GPU-accelerated Large Eddy Simulations of Distributed Electric Propulsion Arrays

Péter Onódi

GPU-accelerated Large Eddy Simulations of Distributed Electric Propulsion Arrays

by

Péter Onódi

in partial fulfillment of the requirements for the degree of

Master of Science
in Aerospace Engineering

at the Delft University of Technology,
to be defended publicly on Friday July 18, 2025 at 14:00.

Responsible thesis supervisor:	Prof. dr. ir. S. Hickel
Company supervisor:	Dr. V. Pasquariello

Thesis committee:

Chairperson:	
Dr. I. Langella,	Delft University of Technology
Examiner:	
Dr. S. J. Hulshoff,	Delft University of Technology
Prof. dr. ir. S. Hickel,	Delft University of Technology
External:	
Dr. V. Pasquariello,	Lilium Aerospace GmbH

Acknowledgement

First of all, I would like to thank my two supervisors. Prof. dr. ir. Stefan Hickel always helped to steer the project in the right direction and his insights were invaluable. Dr. Vito Pasquariello provided indispensable daily support in diverse topics ranging from CUDA coding issues through algorithms to simulation setups. I learned a lot from both of you about LES and aerodynamic software development, and I immensely enjoyed working together with you on this topic.

I would also like to thank the support of the whole aerodynamics team and propulsion department at Lilium, especially my team leads. To Nele, for her compassion in difficult times and for helping getting started. To Iñigo, for the unwavering support, both during the peaks and valleys of the journey. To Amar, for the continuous support in developing my skills and for advocating the project within the company. Thanks to Dr. Alberto Racca both for the initial brainstorming about the thesis topic and for his feedback towards the end. I also owe gratitude to the aerodynamics software development team at Lilium for their excellent work on the solver used in this thesis. Without these solid foundations it would not have been possible to finish. Lilium provided me with all the needed computational resources throughout the project. Although the Lilium Jet will likely never again soar in the sky, I am forever grateful for my fellow Lilians who helped me navigate the difficult transitions in the last turbulent months of the electrifying journey of my thesis.

A special thanks to Susan de Rouw from the academic counsellor team, who helped me revive my MSc after a long break and motivated me in a critical moment.

Lastly, but most importantly, I would like to thank my whole family for their support during my studies. A legtöbb hálával Édesanyámnak tartozom, akinek a szeretete és biztatása nélkül nem jutottam volna el idáig. Mély fájdalommal tölt el, hogy ő már nem érhette meg ezt a pillanatot. Számára mindig a családja volt a legfontosabb, és mindent megtett gyermekei sikeréért és boldogságáért. Az egyik nagy álma az volt, hogy fiai egyetemi végzettséget szerezzenek, amit egy hosszú és rögös út után sajnos már csak túl későn tudok beteljesíteni. Ezt a diplomamunkát az Ő emlékének ajánlom.

Péter Onódi
Gilching, July 2025

CONTENTS

List of Figures

List of Tables

Acronyms

AABB	Axis Aligned Bounding Box
AMR	Adaptive Mesh Refinement
CFD	Computational Fluid Dynamics
DEP	Distributed Electric Propulsion
EOS	Equation of State
eVTOL	Electric Vertical Take-off and Landing
GPU	Graphics Processing Unit
IB	Immersed Boundary
IBM	Immersed Boundary Method
ITR	Inter-processor communication to data redistribution time ratio
LE	Leading Edge
LES	Large Eddy Simulation
LMSR	Locally-matched Multilevel Scratch Remap
MPI	Message Passing Interface
RANS	Reynolds-averaged Navier–Stokes
RPM	Revolution per Minute
SDF	Signed Distance Function
SST	Shear Stress Transport
TE	Trailing Edge
URA	Unified Repartitioning Algorithm
VTOL	Vertical Take-off and Landing
WCS	Wall Cell Size
WMLES	Wall-modelled Large Eddy Simulation

1
Abstract

In distributed electric propulsion (DEP) arrays, complex flow phenomena and strong interaction between the internal flow of the engines and the external aerodynamics demands the use of advanced numerical methods. Separated and unsteady flow during transition and high angle of attack flight, as well as off-design conditions such as engine failures and wind gusts, make scale-resolving methods like Large Eddy Simulation (LES) the most suitable modelling approach. A GPU-accelerated wall-modelled LES (WMLES) solver [2] was successfully employed for eVTOL aerodynamic analysis. Previous work focused on external aerodynamic phenomena, such as transition from hover to cruise flight and hover in ground effect, using a volume source term to model the engines. However, to accurately characterise safety-critical flight conditions, such as the effect of engine failures on adjacent engines or the performance penalty caused by steady wind or gusts during hover, the whole aircraft and multiple engines with resolved turbomachinery components need to be simulated. Since numerous flight conditions need to be analysed with high accuracy to characterise the aerodynamics of an aircraft with DEP, it is key to maximise the computational performance of the solver. In the present work, the main bottlenecks of the solver were identified and removed, which resulted in improvements in the dynamic load balancing, voxelization and the update of the signed distance field. A detailed comparison of the computational performance on four simulation setups shows up to 86% runtime reduction. An eVTOL aircraft is simulated in two off-design tailwind conditions, for which the flowfield and engine performance is analysed. Finally, a comparison with wind tunnel measurements is presented. Results confirm that GPU-accelerated WMLES is a suitable approach to simulate DEP arrays both regarding accuracy and computational cost.

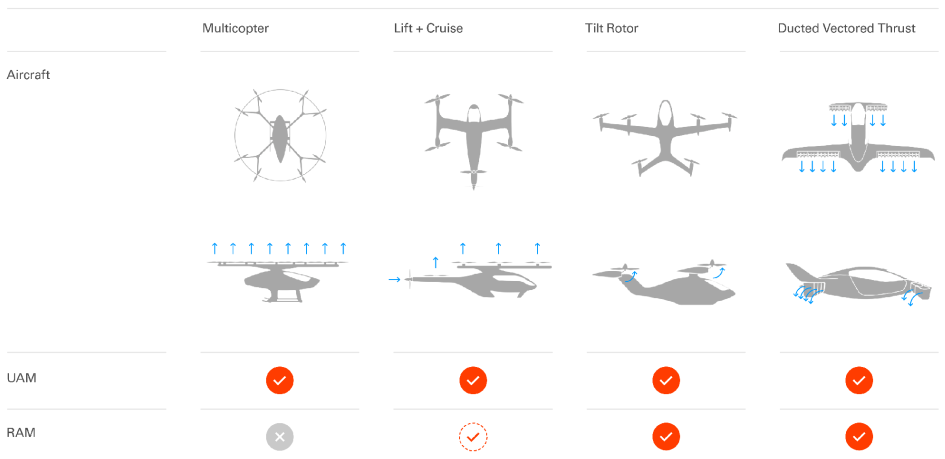
2
Introduction

2.1. Motivation and state of the art

Computational fluid dynamics (CFD) progressed rapidly in the previous decades. Medium fidelity methods, such as Reynolds-averaged Navier Stokes (RANS) simulations are widely adopted in the aerospace industry for the design, analysis and optimisation of aircraft. Reliably predicting unsteady and separated turbulent flows is still a challenge [4, 5], but it is essential to further improve the lift-to-drag ratio of aircraft in off-design conditions, improve the efficiency of engines and to understand the flow physics of novel aircraft configurations with complex aero-propulsive couplings.

Direct numerical simulation (DNS) is likely to have a prohibitively large computational cost for many years, and using wall-resolved large eddy simulation (WRLES) is almost as computationally expensive for high Reynolds number flows. For this reason, wall-modelled LES offers an attractive trade-off between accuracy and computational cost. With constant improvements in wall models and numerical methods, and supported by the rapid pace of development in the field of high performance computing (HPC), it gained popularity in the last years. GPU-based computer architectures are especially suitable for WMLES, making it practical for large-scale use cases.

Several GPU-based WMLES results for the High Lift Common Research Model (CRM-HL) were presented at the 5th AIAA High-lift Prediction Workshop in 2025 [6–9]. WMLES also demonstrated promising results for many other aircraft configurations [10–14]. WMLES also become popular in aerodynamic predictions of propulsion systems [13, 15–20], mainly to evaluate the effect of inflow distortion or to accurately estimate engine noise. Usually an individual propulsion unit is analysed. This can be sufficient for conventional engine installations, but in more tightly integrated propulsion concepts – such as distributed electric propulsion (DEP) – it is important to understand the interaction between the aircraft and the propulsors.



	Multicopter	Lift + Cruise	Tilt Rotor	Ducted Vectored Thrust
Aircraft				
UAM	✓	✓	✓	✓
RAM	✗	✓	✓	✓

Figure 2.1: Comparison of different eVTOL architectures and their suitability to urban air mobility (UAM) and regional air mobility (RAM) [1]

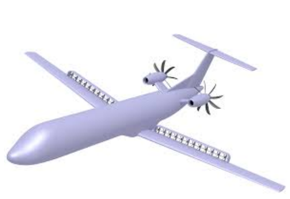
Fig. 2.1 shows an overview of the most common aircraft architectures using DEP in conjunction with VTOL capabilities. The optimal aircraft architecture depends on the range, payload, required hover time and other operational requirements. Propellers offer mechanical simplicity and it is easier to achieve good hover efficiency, whereas carefully designed ducted fans can achieve good high-speed cruise performance (through boundary layer ingestion and using the engine nacelles as lifting surfaces) and reduced noise (by shielding effects or targeting tonal/broadband noise with acoustic liners).

Fig. 2.2 shows a selection of concepts with DEP arrays. These include both conventional take-off and VTOL designs. The ducted arrays are either mounted on the leading edge, trailing edge or they function as a tilt-wing.

(a) Aurora LightningStrike

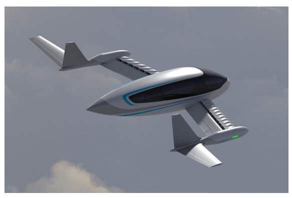
(b) NASA N+3 BWB [21]

(c) ONERA DRAGON [22]


(d) TU Delft OTWDP concept [23]

(e) Lilium Jet (tech. demonstrator)

(f) Pantuo Pantala T1


(g) Whisper Jet

(h) Whisper CLA

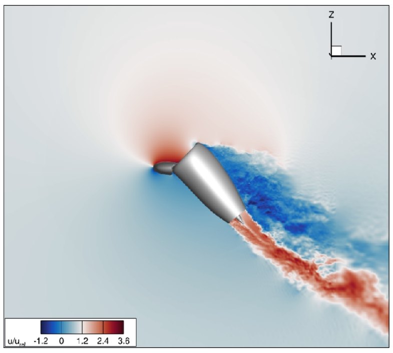
Figure 2.2: Historic and current aircraft concepts and prototypes using DEP arrays (Image sources: Aurora Flight Sciences, NASA, ONERA, TU Delft, Lilium, Pantuo Aviation, Whisper Aero)

A complete review of DEP concepts can be found in [24–26]. The present work will focus only on ducted fan array configurations. Ducted fan arrays are mostly studied with medium fidelity (2D or 3D RANS) methods [22, 27], or investigated experimentally in a wind tunnel [28] or with scaled flight testing [29].

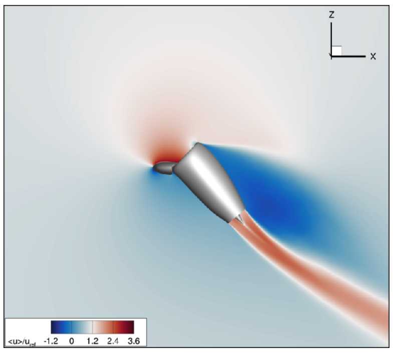
One of the most complex propulsion-related WMLES simulations was done by Powers and Fernandes [30], who simulated a whole wind tunnel circuit, including 6 resolved fans. They do not report the computational cost, but the desired number of fan rotations was not achieved on all configurations of the finest mesh (6 billion cells) due to computational resource constraints.

Aircraft level simulations with resolved propulsion systems are still not commonplace. To accurately predict the coupling between external aerodynamics and engine performance of distributed electric propulsion arrays, especially in off-design conditions, high-fidelity scale-resolving methods are needed.

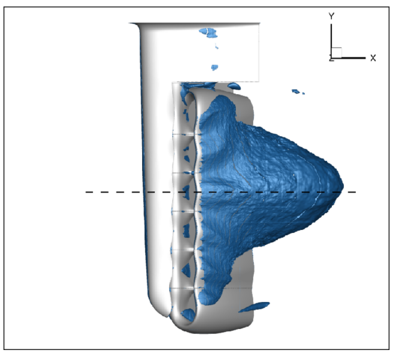
The DEP examples in literature usually focus on the design operating points and use RANS with 2D or quasi-2D setups. The present solver was used to simulate a canard with 6 ducted electric fans (with modelled engines) in a wind tunnel setup [2]. The flowfield at a post-stall condition is shown in Fig. 2.3.



(a) u/u_{ref}



(b) $\langle u\rangle/u_{ref}$



(c) $\langle u\rangle = 0$ iso-surface

Figure 2.3: Instantaneous u/u_{ref} (left) and mean velocity $\langle u\rangle/u_{ref}$ contours (middle) for a y-normal slice cutting through the center of the third inboard flap. Iso-surfaces of $\langle u\rangle = 0$ (right) at 50.3°. Reproduced from [2].

The performance of the present solver was compared with the charLES solver [3]. Comparable turnaround times were achieved on a 12 times smaller hardware in the High Lift 3 Prediction Workshop setup. The performance comparison is repeated in Table 2.1.

Solver	Cell count $\times 10^6$	Non-dim. time step	Wall-clock time to 30 flow passes	Hardware
present solver (mesh1)	72	2.48×10^{-4} 7 min	$8 \times$ A100	
present solver (mesh2)	154	1.22×10^{-4} 27 min	$8 \times$ A100	
present solver (mesh3)	495	6.08×10^{-5} 2.5 h	$8 \times$ A100	
present solver (mesh3-opt)	226	6.14×10^{-5} 1.4 h	$8 \times$ A100	
present solver (mesh3-opt)	226	6.14×10^{-5} 2.9 h	$8 \times$ V100	
charLES	32	6.47×10^{-5} 48 min	$96 \times$ V100	
charLES	157	3.23×10^{-5} 7 h	$96 \times$ V100	

Table 2.1: Computational cost summary for 30 flow passes of the High Lift 3 Prediction Workshop setup at $\alpha = 18.58°$. Timings from charLES solver [3] are given as a point of reference. Reproduced from [2].

Both the canard and the high lift setups are static cases, they do not contain moving geometry and do not use AMR. However, when it is used with moving setups (e.g. the spinning motion of resolved engines, or the aircraft moving in the domain), a significant overhead is added.

2.2. Solver bottlenecks

An abridged version of the workflow and hierarchy of the baseline solver is shown in Fig. 2.4. Components of the solver that are modified in the scope of the present work are highlighted, and they are discussed in detail in the subsequent chapters. The *Run motion solver*, *Advance geometry* and *IB update* steps are only active in moving setups, and the *Apply volume forces* step is only applicable in setups containing modelled engines.

To understand where the overhead is coming from, a static and a moving setup with adaptive mesh refinement (AMR) is compared. The chosen example is a full body setup of an electric vertical take-off and landing (eVTOL) aircraft (shown in Fig. 2.2e) hovering in ground effect. In the static setup the aircraft is hovering at a constant height above ground of $h_0 = 5$ m. The dynamic setup starts from the same position, but the aircraft is moving with a climb speed of $V_c = 1$ m/s. The setup is shown in Fig. 2.5, and the main mesh settings are included in Table 2.2.

Parameter	Value
GPUs	$8 \times$ Nvidia A100
Cells per block	16^3
Domain size	$[100, 100, 100]$ m
Surface refinement level	10
Blocks	64×10^3
Cells	262×10^6
Cut cells	2×10^6

Table 2.2: Aircraft hover in ground effect settings and mesh properties

In a static WMLES simulation (Fig. 2.6a) most of the runtime is spent in the flow solver, such as time-stepping (*Time discretization*), calculating the time step and the sub-grid scale viscosity (*SGS*), applying the boundary conditions on the ghost cells, connecting the flow solution between mesh blocks and mapping the flow solution onto the surface (*Reconstruct*).

In a moving setup (Fig. 2.6b) with the baseline solver, the actual flow solver takes up only a small portion of the runtime. The runtime of kernels related to the flow solution remains on the same order of magnitude. The time step size of the simulation is similar in the static and dynamic setup. This is expected, since the climb speed is low compared to the exhaust jet velocity, which is the limiting factor for the time step size. By far the biggest bottleneck is the load balancing within AMR, which corresponds to three quarters of the runtime. The second biggest contributor is `update_flag` within the IB update, which covers the main steps needed to update the signed distance field as the geometry moves. These impose a prohibitively large computational cost for the targeted use cases, therefore the thesis investigates how these parts of the solver can be accelerated.

It is difficult to evaluate, if these bottlenecks are specific to the present solver or a generic obstacle for large scale Immersed Boundary Method (IBM) simulations with moving geometry and AMR. There are not enough examples in the open literature, and usually they either do not report in detail the computational cost or do not compare static and moving setups with similar complexity.

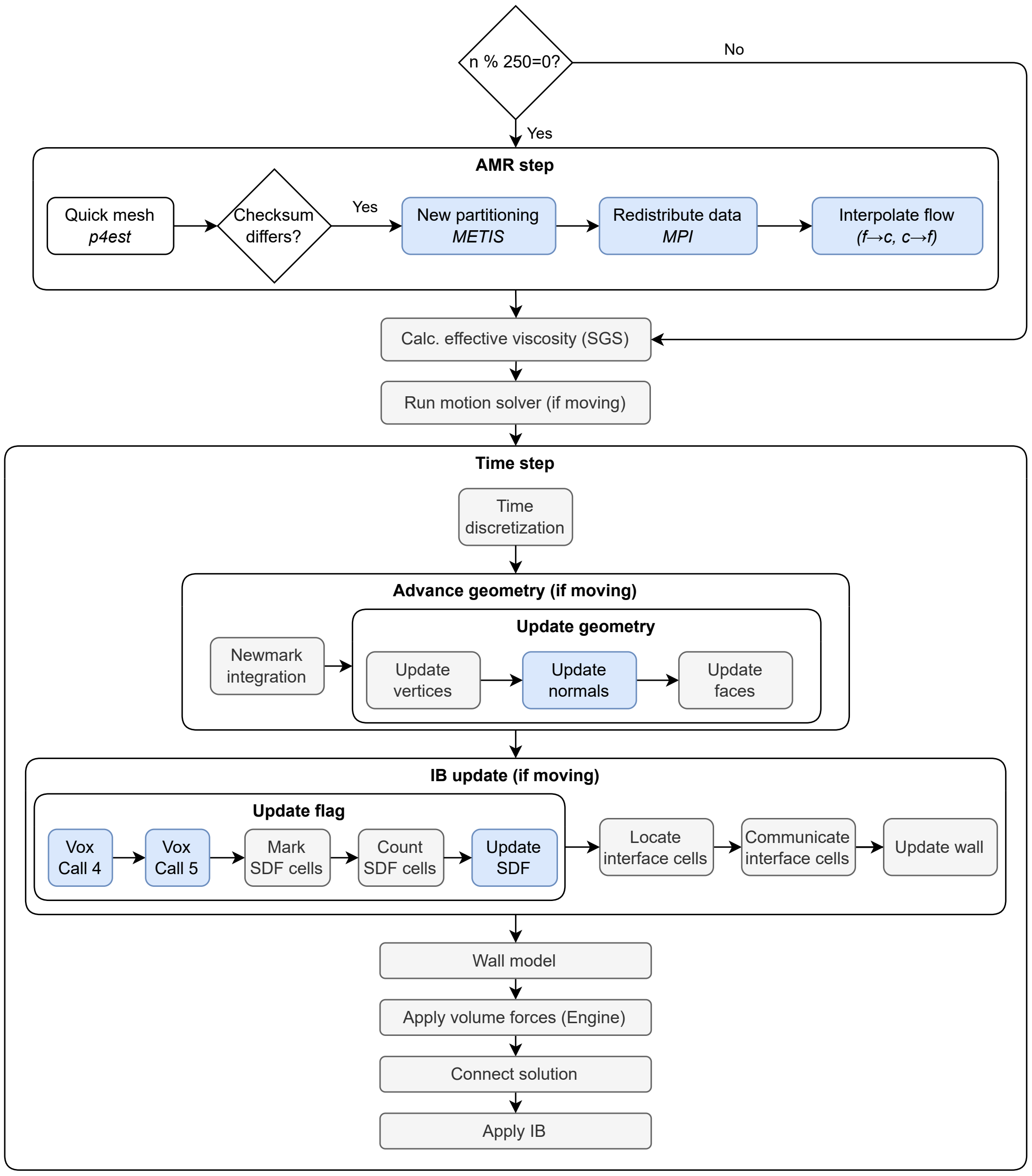


Figure 2.4: Solver workflow

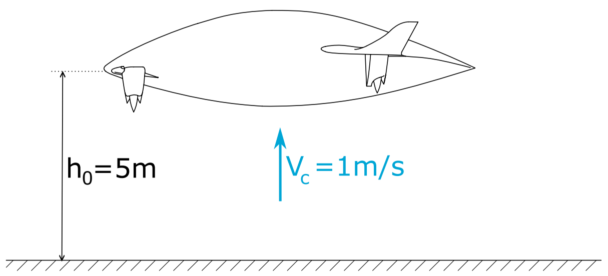
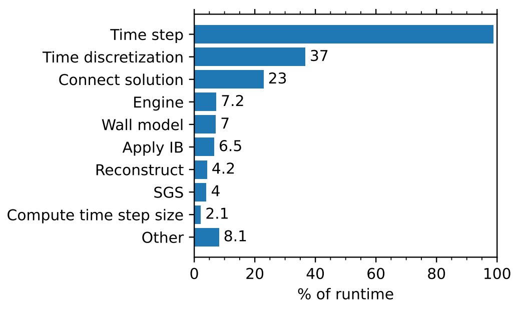
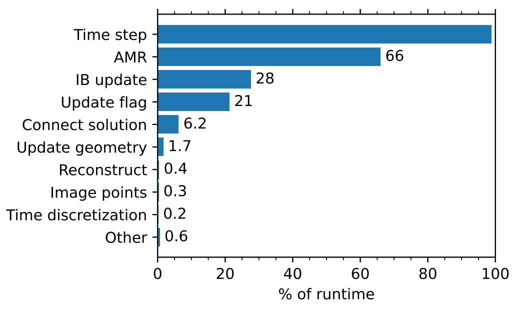


Figure 2.5: Aircraft hover in ground effect setup



(a) Static setup



(b) Moving setup with AMR

Figure 2.6: Comparison of timings in a static and moving case

2.3. CONTRIBUTIONS OF THIS THESIS

The research objective is to develop efficient methods to handle moving geometry and AMR in a GPU-based scale resolving CFD solver, simulate large-scale distributed electric propulsion setups and validate them with measurement data. The selected flight condition is hover in tailwind, since it is an important sizing point of DEP arrays installed on VTOL airplanes, but it is still a research gap in the literature.

The present research is aiming to answer two research questions.

What are the aerodynamic characteristics of DEP arrays in hover with tailwind?

The following sub-questions aid to answer the first research question:

- How strong is the effect of tailwind on the engine inlet flow field?
- How strongly is the engine thrust and torque affected by off-design tailwind conditions?
- How accurately can the present solver predict engine performance compared to wind tunnel data?

What is an efficient implementation of a moving immersed boundary method in a GPU-based high-fidelity WMLES solver for distributed electric propulsion applications?

The second research question is broken down into the following sub-questions:

- How much can the AMR be accelerated by using different partitioning algorithms?
- What speed-up is expected from using efficient algorithms for the immersed boundary update?
- What is runtime of a full aircraft DEP simulation?

The layout of the report is as follows. Chapter 3 introduces the solver used in the study, including the governing equations, the fluid model, a brief explanation of the mesh generation, IBM and finally the load balancing strategy. It is not in the scope of the report to provide a full understanding of the solver, and only the parts that are relevant to present research are highlighted. Chapter 4–6 detail the main bottlenecks of the baseline solver, for each of them introducing the methodology, describing the causes of the bottlenecks and providing improved algorithms to reduce and eliminate them. Chapter 4 contains a detailed investigation of the parameters influencing the load balancing. Since these contributed to selection of parameters of the code version used in all subsequent investigations, they are presented early on. Chapter 5 and Chapter 6 on the other hand focus on the methodology, and the results are relegated to Chapter 7, which compares the baseline and new version in a selection of four different setups ranging from a simple airfoil simulation to a distributed electric propulsion simulation on full aircraft scale. For each setup a detailed comparison of the baseline and optimised solver is provided. Whereas Chapter 7 concentrates on the performance of the solver, Chapter 8 includes an aerodynamic analyses of the last setup – eVTOL aircraft hovering in tailwind at low and high RPM. The CFD results are compared with wind tunnel data, including inlet and outlet total pressure rakes, surface pressure taps, engine mass flow rate and thrust. Finally, conclusions and future work suggestions are discussed in Chapter 9.

3
Methodology

3.1. Navier–Stokes equations

The governing equations are the three-dimensional weekly compressible Navier–Stokes equations. The conservative form of the Navier–Stokes equations is given as:

$$\partial_t \mathbf{U} + \nabla \cdot \mathbf{F}(\mathbf{U}) - \nabla \cdot \mathbf{D}(\mathbf{U}) = 0 \tag{3.1}$$

The state vector $\mathbf{U} = [\rho, \rho u_1, \rho u_2, \rho u_3]$ consists of the density (ρ) and linear momentum (u_i are the velocity components). The flux is divided into an inviscid $\mathbf{F} = [\mathbf{f}_1, \mathbf{f}_2, \mathbf{f}_3]^T$ and viscous part $\mathbf{D} = [\mathbf{d}_1, \mathbf{d}_2, \mathbf{d}_3]^T$, and given as

$$\mathbf{f}_i(\mathbf{U}) = \begin{bmatrix} u_i \rho \\ u_i \rho u_1 + \delta_{i1} p \\ u_i \rho u_2 + \delta_{i2} p \\ u_i \rho u_3 + \delta_{i3} p \end{bmatrix}, \tag{3.2}$$

and

$$\mathbf{d}_i(\mathbf{U}) = [0, \tau_{i1}, \tau_{i2}, \tau_{i3}]^T . \tag{3.3}$$

τ_{ij} is the viscous stress tensor according to the Stokes hypothesis [31]:

$$\tau_{ij} = \mu \left(\partial_j u_i + \partial_i u_j - \frac{2}{3} \delta_{ij} \partial_k u_k \right) \tag{3.4}$$

The dynamic viscosity is the sum of the molecular viscosity of air and the subgrid-scale (SGS) eddy viscosity $\mu = \mu_{air} + \mu_{SGS}$. Throughout this report Vreman's eddy viscosity model [32] is used.

3.2. Fluid model

Air is accurately modelled as an ideal gas mixture with the equation of state (EOS)

$$p = \rho R T , \tag{3.5}$$

which relates pressure p, density ρ and temperature T. The specific gas constant of air is

$$R := \mathscr{R}/M \approx 287.0509011 \mathrm{J/kgK}, \tag{3.6}$$

where the universal gas constant is exact by definition: $\mathscr{R} := 8314.46261815324$ J/kmolK. Dry air is modelled as a mixture of 78.084% N_2, 20.9476% O_2, 0.9365% Ar and 0.0319% CO_2 (mole fractions), which gives for the molar mass $M = 28.9651159$ kg/kmol [33].

Both the pressure and temperature of an ideal gas are non-linear functions of density and internal energy, that is $p = p(\rho, e)$ and $T = T(\rho, e)$. This relation is usually expressed in form of thermodynamic derivatives that are accessible to measurement, notably the specific heats at constant pressure and constant volume

$$c_p := \left.\frac{\partial h}{\partial T}\right|_p , \tag{3.7}$$

$$c_v := \left.\frac{\partial e}{\partial T}\right|_\rho . \tag{3.8}$$

The enthalpy of an ideal gas can be expressed as $h = e + RT$, which yields the convenient relation $c_v = c_p - R$. The specific heat c_p is modelled with the NASA Glenn polynomials [34]

$$c_p \approx R\left(a_1 T^{-2} + a_2 T^{-1} + a_3 + a_4 T + a_5 T^2 + a_6 T^3 + a_7 T^4\right), \tag{3.9}$$

where the coefficients are:

$$\mathbf{a} = \begin{bmatrix} 1.009950160 \times 10^4 \\ -1.968275610 \times 10^2 \\ 5.009155110 \times 10^0 \\ -5.761013730 \times 10^{-3} \\ 1.066859930 \times 10^{-5} \\ -7.940297970 \times 10^{-9} \\ 2.185231910 \times 10^{-12} \end{bmatrix} \tag{3.10}$$

In the present solver, a barotropic EOS is used in order to avoid the high computational cost of solving an energy transport equation. In a barotropic EOS the pressure is the function of only density $p = p(\rho)$, i.e. it does not depend on temperature. This barotropic EOS is obtained by linearising the ideal gas model around a reference state $\{T = T_{ref}, p = p_{ref}\}$:

$$p = A\rho + B , \tag{3.11}$$

with

$$A = \gamma_{ref} R\, T_{ref} \tag{3.12a}$$ $$B = \left(1 - \gamma_{ref}\right) p_{ref} \tag{3.12b}$$

Slope A and integration constant B are obtained by evaluating $\gamma_{ref} = c_p(T_{ref})/(c_p(T_{ref}) - R)$ and Eq. 3.9 at the reference state. The stagnation density ρ_0 and stagnation pressure p_0 are derived assuming quasi-one-dimensional flow in an adiabatic stream tube without viscous losses. The stagnation density is

$$\rho_0 = \rho \exp\left(\frac{u^2}{2A}\right) , \tag{3.13}$$

and the corresponding stagnation pressure is obtained by substituting ρ_0 into Eq. 3.11:

$$\begin{aligned} p_0 &= A\rho \exp\left(\frac{u^2}{2A}\right) + B \\ &= p \exp\left(\frac{u^2}{2A}\right) + B\left[1 - \exp\left(\frac{u^2}{2A}\right)\right] . \end{aligned} \tag{3.14}$$

For air at moderate temperatures and pressures, the dynamic viscosity μ can be modelled with Sutherland's law [35]

$$\mu_{SL}(T) = \mu_{cal}\left(\frac{T}{T_{cal}}\right)^{3/2} \frac{T_{cal} + S_{cal}}{T + S_{cal}} , \tag{3.15}$$

where the calibration coefficients μ_{cal} and S_{cal} depend on the chosen calibration temperature T_{cal} and the desired temperature range. The present solver uses

$$T_{cal} = 273.15\text{K}, \quad \mu_{cal} = 17.16 \times 10^{-6}\text{Pa}\cdot s, \quad S_{cal} = 110.4\text{K} . \tag{3.16}$$

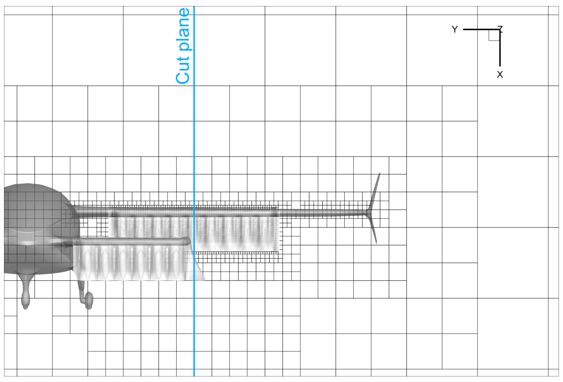
As the temperature differences in low Mach number flows are small, we avoid the cost of computing and storing the local viscosity in every cell and rather approximate it by a global constant. This global value is calculated from the expected wall temperature, that is

$$\mu = \mu_{SL}\left(T_{ref} + \frac{u_{ref}^2}{2c_p(T_{ref})}\right) , \tag{3.17}$$

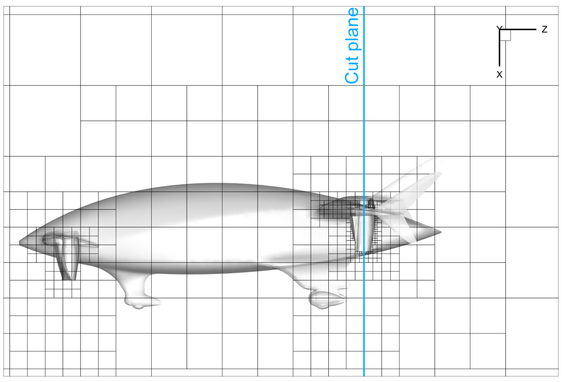
because the accurate modelling of viscous wall shear stresses is most relevant.

3.3. MESH

A Cartesian grid consisting of a forest of octrees is employed, which enables memory-efficient storage of the solution. This is especially important for GPU-based solvers, where the problem size is often constrained by memory limitations. Beside the even more compact representation of the mesh, using unit aspect ratio cells also reduces numerical errors [11] and has advantageous properties in some SGS models [32]. Fig. 3.1 shows the grid blocks around an aircraft setup. A geometry-based mesh refinement around the wall boundaries is included.



(a) Side view



(b) Front view

Figure 3.1: Grid blocks for simulation of a wind tunnel model. Black lines are block edges, the blue lines show the front and side view's cut plane. Each block contains 16^3 cells.

In a Cartesian grid the cell boundaries are not aligned with the flow in a general case, therefore the immersed boundary method is used to enforce the boundary conditions at the walls.

3.4. IMMERSED BOUNDARY METHOD

In the present solver the ghost-cell immersed boundary (IB) method by Mittal et al. [36] is used. In this method the boundary conditions on the wall are imposed implicitly by specifying fluid variables in the cells surrounding the boundary (ghost cells). A signed distance function (SDF) is constructed from the triangulated surface, with the fluid volume satisfying SDF > 0 and SDF ≤ 0 for the solid region. Ghost cells have also a negative SDF, but they have at least one fluid neighbour. Analogously, wall cells have a positive signed distance, but they have at least one ghost cell neighbour. The different cell types are shown in Fig. 3.2.

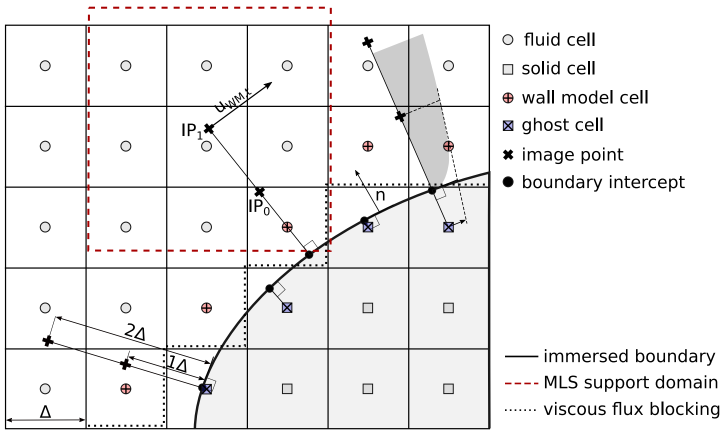


Figure 3.2: Ghost-cell immersed boundary and wall-modelling [2]

An equilibrium wall-stress model is used to estimate the wall-shear stress from the instantaneous wall-parallel velocity at a constant distance from the wall. Two image points IP_0 and IP_1 are placed perpendicular to the wall to interpolate the flow solution with a moving least squares (MLS) interpolation [37].

The workflow of initialising the IB and updating it in moving setups is outlined in Fig. 3.3. The highlighted steps are further discussed in Chapter 5 and 6.

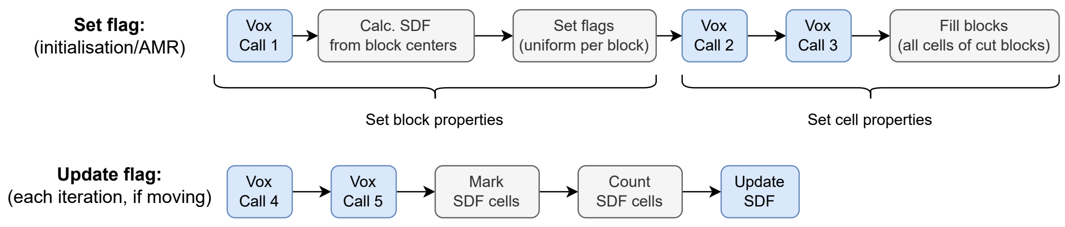


Figure 3.3: IB workflow

3.5. ADAPTIVE MESH REFINEMENT AND LOAD BALANCING

The solver uses a structured multi-block grid, where the forest of octrees is constructed with the *p4est* [38] library. A local mesh refinement is used on immersed boundaries, where the refinement level can be specified separately for each body. Furthermore, additional surface or volume refinements can be added by the user. The *p4est* library creates a Z-ordering of the mesh blocks, and distributes the Z-ordered blocklist between partitions. This can be sufficient, if the computational cost of each block is equal, but it does not guarantee to minimise the inter-partition communication cost. Additionally, an unequal distribution of ghost cells results in an unbalanced load.

To overcome this, the METIS [39] library is used for load balancing in the baseline version (as presented in [2]). For the present work ParMETIS [40] was adopted, which has further adaptive re-partitioning capabilities and parallel partitioning algorithms. The load balancer employs a multi-objective partitioning, which aims to equally distribute both the fluid volume and the IB cells between the partitions, since the IB cells have a higher computational cost than fluid cells.

A moving IB implementation for aeroelastic applications was introduced in [41]. In a moving setup the SDF is recomputed in each timestep, and the cell types near the walls are reassigned. For moving bodies an adaptive mesh refinement (AMR) can be used, which in certain iteration intervals probes the mesh by constructing a new *p4est* forest of octrees. This is a very quick step, and if the new forest differs from the current one, an AMR step is needed. As the geometry is moving, the initial load balancing would degrade due to the uneven distribution of IB cells and because the density of grid blocks is higher around the bodies. Hence, load balancing is needed after the AMR steps.

An overview of the AMR workflow is shown in Fig. 3.4.

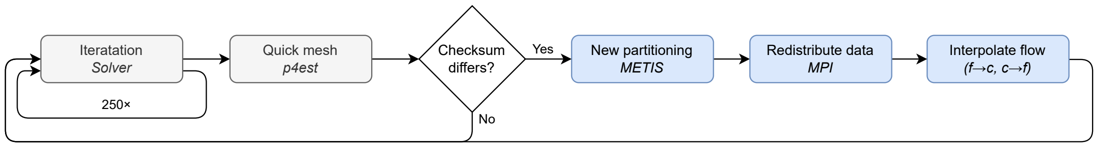


Figure 3.4: AMR workflow

3.6. TEST CASES

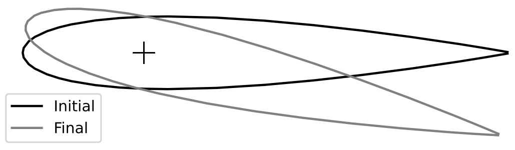
Four test cases with increasing complexity are selected (Fig. 3.5). All of them include moving geometry and AMR. The first and simplest is a pitching airfoil. The others model a distributed electric propulsion system with different fidelity. The test cases are listed below. The mesh settings of all setups are described in detail in Chapter 7.

Pitching airfoil A NACA 0015 airfoil is rotating around the quarter-chord line. The domain is quasi-2D with periodic boundaries on the sides. The domain consists of many grid blocks, which results in a relatively fine mesh resolution in the far field.

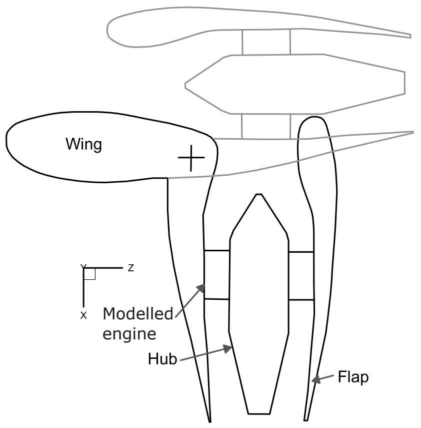
Rotating flap Consists of a wing section, a flap (which also functions as engine nacelle) and an engine. The engine is modelled with a volume source term. The flap and engine are rotating from hover to cruise position around the hinge line at the wing trailing edge. The domain has periodic boundaries to represent an infinite array.

Flap with resolved engine The setup has an engine with resolved blades and stator vanes, and it is solved on a finer mesh and larger domain. The wing and flap are not moving, and they are merged into one body. The domain has periodic boundaries.

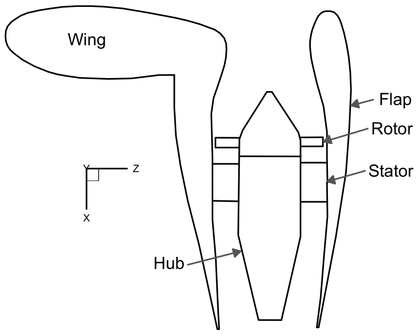
Aircraft wind tunnel model A complete eVTOL aircraft, with resolved engines on one wing, and modelled engines on the other wing and canards. The aircraft has 6 engines on canards and 9 on wings, 30 in total. The flaps and lifting surfaces around the modelled engines, as well as the fuselage, have a much coarser surface mesh resolution than the left wing.



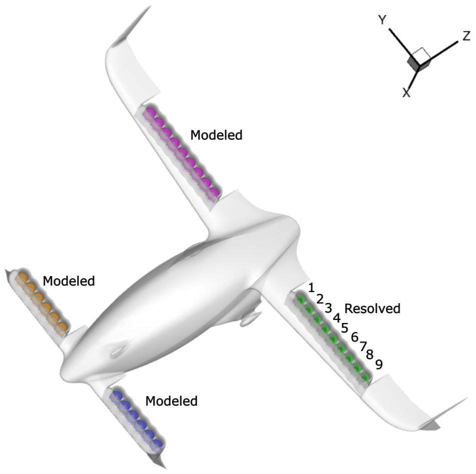
(a) Pitching airfoil



(b) Rotating flap



(c) Flap with resolved engine



(d) Aircraft wind tunnel model

Figure 3.5: Test cases

A single instance of an Nvidia DGX A100 [42] was used for the aircraft wind tunnel model test case and an Nvidia DGX V100 [43] was used for the other cases.

4
LOAD BALANCING IN AMR STEPS

4.1. INTRODUCTION

AMR updates the mesh based on the proximity of IB walls (in case of moving setups) or based on some flow solution dependent criterion. In the present solver the AMR criterion is evaluated at regular intervals. When the mesh needs any refinement or coarsening, a new octree is created and the solution is mapped onto the new octree. Changing the mesh also changes the workload of the partitions, and it is beneficial to balance the workload after mesh adaptation. The redistribution of the workload necessitates data transfer via MPI [44] between partitions. The algorithm is outlined in Fig. 4.1.

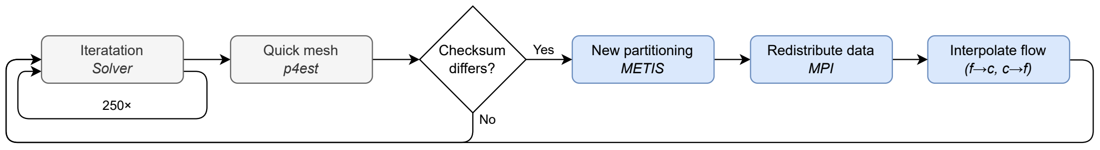


Figure 4.1: AMR workflow

The key concepts and methods used for dynamic load balancing are defined below.

Edge-cut A set of graph edges (in this case cell faces, edges and vertices) of a connected graph, that if disconnected, split the graph into multiple disconnected graphs. In a 3D volume the edges of the edge-cut form a set of surfaces, that split the volume into disjoint parts. In the present work the graph edges are weighted with the amount of data that is exchanged on them, which is different for faces, edges and vertices.

Scratch remapping A new partitioning is created at each update, typically with a high data redistribution cost. Geometric schemes [45–48] usually have a lower data redistribution cost, but a worse quality. Multilevel schemes [49–54] create a higher quality partitioning, but subsequent partitionings are in general not similar to each other [55].

Locally-matched Multilevel Scratch Remap (LMSR) Aims to keep a high overlap with the original partitioning [55]. During the coarsening phase only graph vertices with the same home domain are matched, preserving the partition boundaries.

Diffusion Computes a balanced solution by a small perturbation of the original partition graph. It calculates a flow solution, which determines the amount of work exchange between pairs of partitions.

Wavefront Diffusion (WF) The flow of vertices moves in a wavefront from overweight to underweight subdomains [55]. Contrary to other diffusion algorithms, which select the moved vertices randomly or with the highest impact on the edge-cut (greedy algorithms), WF starts with the partitions that have no required inflow. Next, it iteratively propagates to partitions whose inflow was already satisfied. WF can achieve both lower edge-cut and data redistribution cost compared to other diffusion schemes [55].

Unified Repartitioning Algorithm (URA) Since either scratch-remap or diffusion algorithms can perform better for different graphs, URA employs both LMSR and WF [56]. Each partition creates a partitioning graph, half of them using LMSR and the other half using WF. The most favourable graph is accepted for the new mesh.

Fig. 4.2 shows an example result of the baseline partitioner. The setup is an airfoil in a large domain, where the spanwise direction is 1% of the height (1 block on the coarsest level), so the domain is quasi-2D. The airfoil is rotating pitch-up around the quarter-chord line 0.2° between the initial position (Fig. 4.2a) and the first AMR step (Fig. 4.2c). Although this difference is hard to see and does not necessitate the refinement and coarsening of many blocks, a large portion of the surface cuts through different blocks by the first AMR steps, shifting the balance of the cut cells.

When a new partitioning is created at each AMR step (the baseline approach), a large portion of the data (here 73%) is moved. An improved approach is a scratch-remap, where the partition boundaries are still created from scratch, but the ranks are assigned to minimise the data exchange. For this example even with the remapping 17% of the data is moved, because the partition boundaries changed significantly. This domain without the refinement around the geometry and the cut cell balancing constraint would have a trivial optimal partitioning with 4×2 square partitions. With the geometry and constraint some partitions have pockets around the geometry to take their share of the cut cells and the boundaries in the far-field are distorted to maintain the balance of the blocks. Although the partitions follow the basic layout of the optimal partitioning of the simplified case, there are many solutions with a similar edge-cut and imbalance, but large shifts in the boundaries, which still result in excessive repartitioning costs.

A much better result is achieved with URA — which relies on the previous partitioning — where only 0.1% of the blocks is moved. The majority of the moved blocks is concentrated around the geometry (Fig. 4.5), but part of the far-field also changed (above and to the left of the zoomed area) to compensate for the changes around the moving geometry. The graph with the best cost function is picked, where the cost function considers both edge-cut and data redistribution cost.

This example demonstrates, that the data redistribution cost, which was identified as the main bottleneck of the present solver, can be effectively reduced.

4.2. METHODOLOGY

An epoch can be considered as the time between AMR steps, which consists of the computational time during iterations, the time to redistribute the load (mainly copying data between GPUs) and the repartition time t_{repart}, which is the time to compute the new partition graph.

$$t_{epoch} = \left(t_{comp} + f\left(|E_{cut}|\right)\right) n + g\left(|V_{move}|\right) + t_{repart} \tag{4.1}$$

The cost function of the partitioner aims to (indirectly) minimise the epoch time, and therefore the runtime:

$$\min!\ J = ITR \cdot |E_{cut}| + |V_{move}| \tag{4.2}$$

The ITR factor is the ratio between the inter-processor communication time (proportional to the edge-cut $|E_{cut}|$) and the redistribution time (proportional to the total amount of data redistribution $|V_{move}|$).

In ParMETIS the valid range for ITR is $0.0001 < ITR \leq 1000000$ [40] and $\epsilon > 0$ must apply for the imbalance tolerance.

For the set of vertices $B(q)$ of partition q the partition weight is the weighted sum of the vertices [57]:

$$W(q) = \sum_{v_i \in B(q)} w_i \tag{4.3}$$

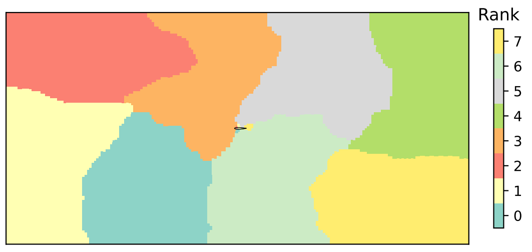
The average partition weight is:

$$\overline{W} = \frac{1}{p} \sum_{i=1}^{p} W_i \tag{4.4}$$

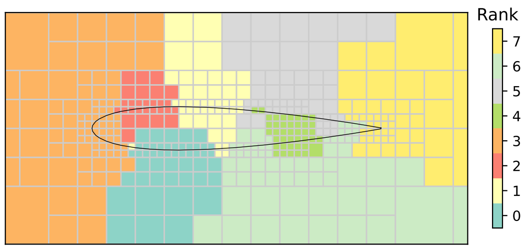
where p is the number of partitions.

A graph is imbalanced if

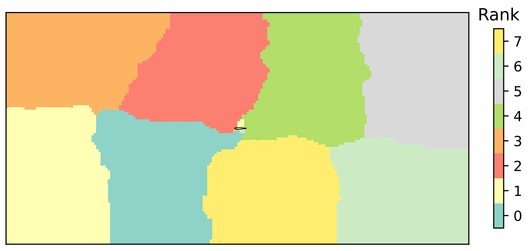
$$\exists q | W(q) > \overline{W}(1+\epsilon) \tag{4.5}$$



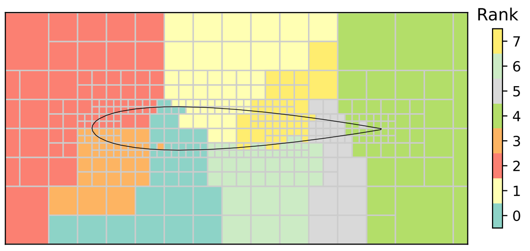
(a) AMR step 0 (initial)



(b) AMR step 0 (zoom to immersed boundary)



(c) AMR step 1



(d) AMR step 1 (zoom to immersed boundary)

Figure 4.2: Initial partitioning and after the first AMR step (scratch partitioning)

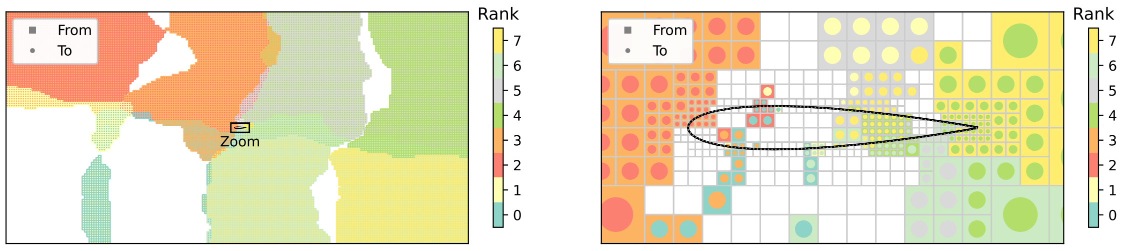


Figure 4.3: Moved blocks in scratch partitioning. Domain *(left)* and zoom at the immersed boundary *(right)*.

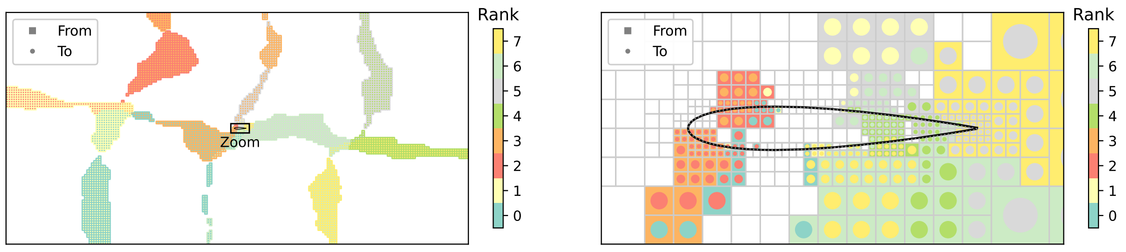


Figure 4.4: Moved blocks in scratch-remap partitioning. Domain *(left)* and zoom at the immersed boundary *(right)*.

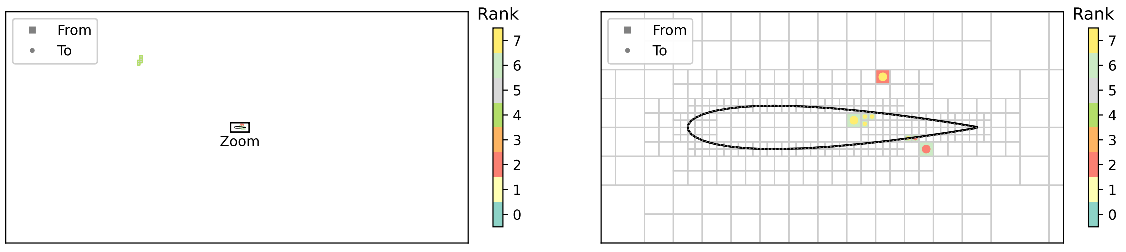


Figure 4.5: Moved blocks in with URA partitioning. Domain *(left)* and zoom at the immersed boundary *(right)*.

where $1+\epsilon$ is the imbalance tolerance. The imbalance shown in the plots of this chapter is the ratio of the partition with the largest weight to the average weight: $\max\left\{W(q)/\overline{W}\right\}$.

Both METIS [39] and ParMETIS [40] use a multilevel graph partitioner. The main steps are coarsening, initial partitioning and refinement (Fig. 4.6).

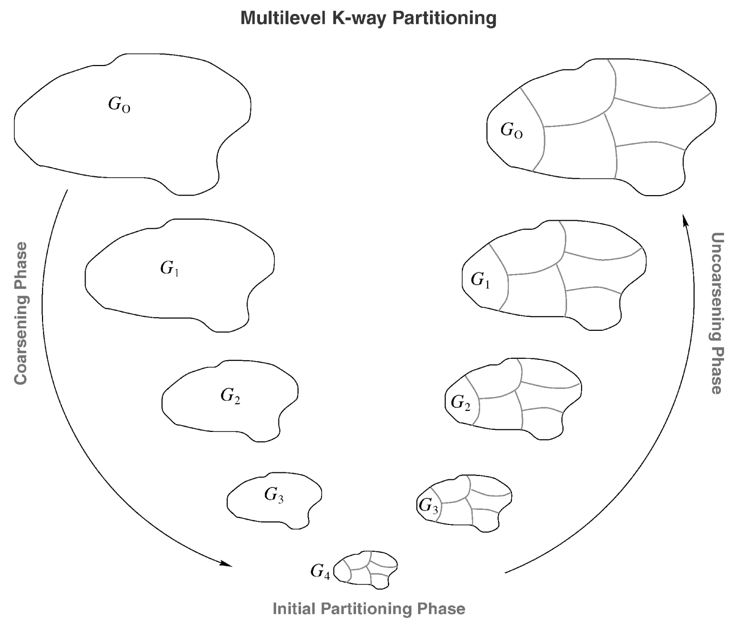
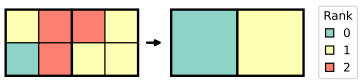


Figure 4.6: The three phases of multilevel k-way graph partitioning [55]

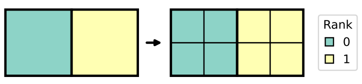
In the coarsening phase graph vertices are collapsed, until the graph is sufficiently small for the initial partitioning. On each level, the graph is refined and uncoarsened. Usually a heuristic algorithm is used for the refinement, which tries to swap graph vertices, and keeps the changes if they provide an improvement in the edge-cut and balance.

After an AMR step the new mesh has a different space-filling curve [38], and a given block is not at the same place in the block list as its parent was. To find its parent, the vertex coordinates of a new block are compared with all old blocks. If the block was refined, they share exactly one vertex, and the opposite vertex is within the parent block's volume.

If the block is coarsened, the rank is inherited from the first block, that is found with the above method (Fig. 4.7a). This is faster than assigning the new rank based on the majority rank among the octants of the previous mesh, and it is a good enough starting point for the partitioner. If the block is refined, all of the refined blocks inherit the rank of the coarser block of the previous mesh (Fig. 4.7b).



(a) Fine to coarse mapping



(b) Coarse to fine mapping

Figure 4.7: Mapping of partitions during mesh refinement

4.3. EFFECT OF ITR AND IMBALANCE TOLERANCE

The ITR factor and the imbalance tolerance are free parameters, and their optimal value depends both on the solver and the mesh. Their effect is investigated in 4 different setups. The setups are described in detail in Chapter 7, but for the following study it suffices to know some key properties of them (included in Table 4.1). The ratio of cut cells to fluid cells is important, because both of them are controlled with balancing constraints. This is needed, because ghost cells need both more computations and memory than volume cells. The aspect ratio of the domain influences the edge-cut, especially thin domains have smaller connections between the partitions.

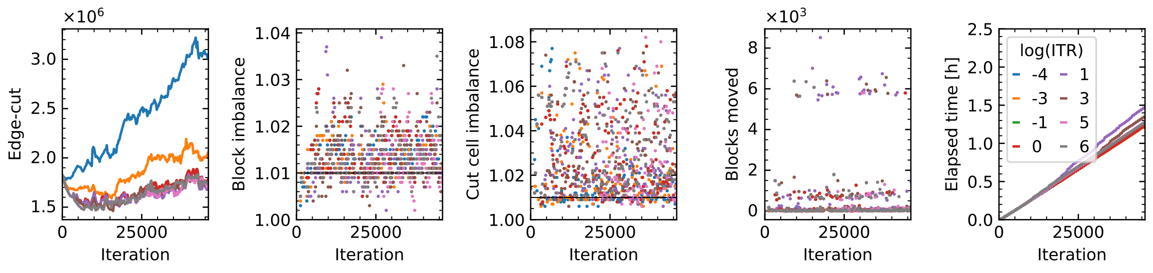
Property	Pitching airfoil	Rotating flap	Slice	Aircraft wind tunnel model
# of cut cells / # of cells [%]	0.1	2.6	1.5	3.3
# Domain aspect ratio	200:100:1	30:30:1	100:100:1	1:1:1
Surface refinement level	3	3	4–6	10–14
# of partitions	4	8	8	8
# of constraints	2	3	2	3
Static bodies	0	1	1	22
Moving bodies	1	2	1	9

Table 4.1: Settings of the four test cases

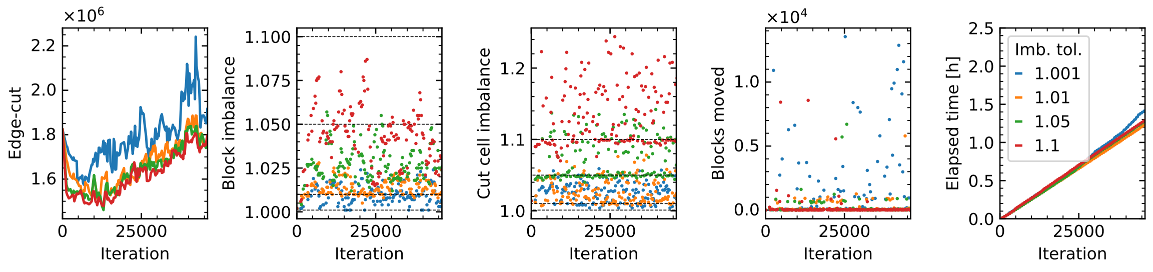
4.3.1. PITCHING AIRFOIL

The first and simplest setup was already introduced to demonstrate different partitioning techniques (Fig. 4.2–4.3). With this setup the whole range of allowed ITR values is explored. The effect of imbalance is tested with 4 different values, where 1.001 demands an almost perfect balancing (as defined in Eq. 4.5). Although the effect of the two parameters is not independent, and it would be desirable to create a design of experiments covering a larger number of combinations, the number of simulations had to be limited.

Fig. 4.8 shows the iteration of the ITR sweep on the top and the imbalance tolerance sweep on the bottom. In both cases the prescribed imbalance tolerance is not satisfied by the partitioner, since it is a target and not a hard constraint. The number of blocks moved forms two distinct clusters. In most cases the hierarchical coarsening and refining converges back to a similar graph as the starting point, swapping nodes only on the finer levels. With low ITR the edge-cut is higher and the difference is increasing as the simulation progresses.



(a) Effect of ITR (1.01 imbalance tolerance)



(b) Effect of imbalance tolerance (ITR=0.1)

Figure 4.8: Variation of edge-cut and imbalance during AMR steps (Pitching airfoil)

Part of this is coming from the increased number of blocks (Fig. 4.9), which is simply a consequence of the orientation of the airfoil, and it would reduce with further pitch. So much weight is put on the redistribution cost, that the partitioning remains virtually unchanged (except due to the coarsening and refinement of the mesh, see Fig. 4.7). A strict imbalance tolerance also results in an up to 20% higher edge-cut, since the partitioner has less freedom to swap blocks, usually yielding a more complex partition boundary.

The results shown in Fig. 4.8 are summarised in Fig. 4.10. The edge-cut improves both with increased ITR and imbalance tolerance, but with diminishing returns. In this specific case the block imbalance constraint is close to the target value, and with loose imbalance tolerances it is even exceeded. On the other hand the cut cells are unevenly distributed, and the target on average is not achieved. The setup has a small (in terms of cut blocks) and simple geometry, which makes it harder to distribute the blocks on the surface without compromising the edge-cut. At first glance this might seem like a problem, but since the geometry has a small

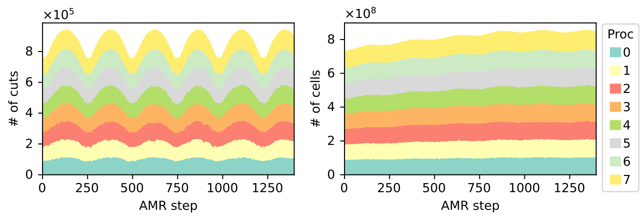
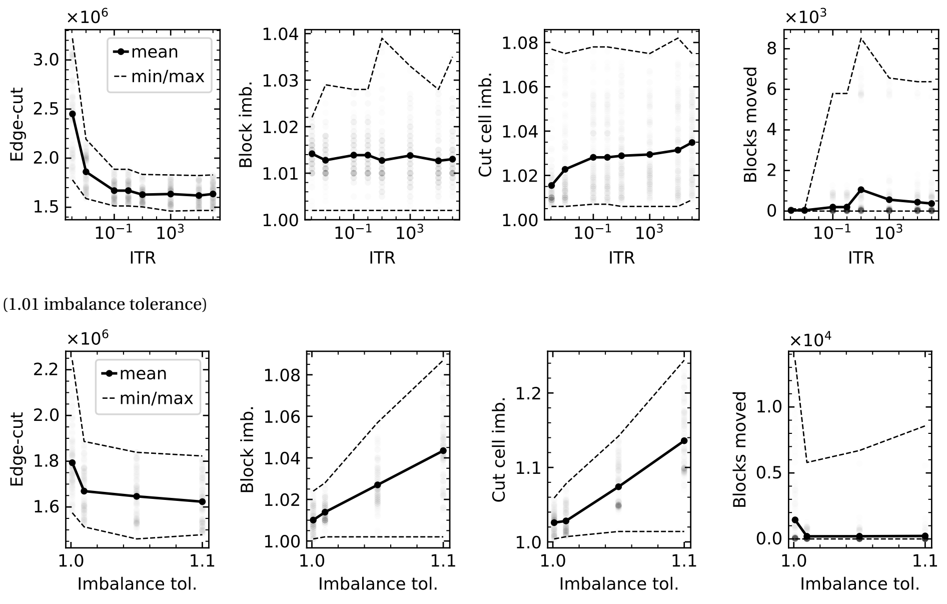


Figure 4.9: Balance of cells and cut cells during AMR steps (Pitching airfoil). The total number of cells and cut cells fluctuates with the pitch angle.

fraction of blocks, the majority of compute time is not spent in the immersed boundary kernels, therefore a higher imbalance is acceptable. Unexpectedly, the largest amount of blocks is moved at middle ITR values, and not at high ITR, where the weight on inter-partition data transfer cost is low.

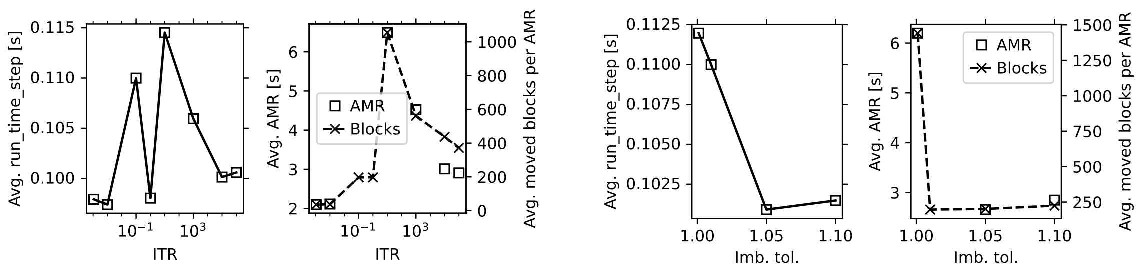
The AMR time is driven by the number of moved blocks (Fig. 4.11), which becomes a bottleneck for the overall runtime. In this setup 1.05 imbalance tolerance is a local minimum for the runtime. With different ITR values the runtime fluctuates by more than 10%, and very low ITR gives the best values, despite the high edge-cut. Since in quasi-2D setups the edge-cut is less influential, the conclusions should not be extrapolated beyond similar domains.



(a) Effect of ITR (1.01 imbalance tolerance)

(b) Effect of imbalance tolerance (ITR=0.1))

Figure 4.10: Average edge-cut and imbalance for different ITR factors and imbalance tolerances (Pitching airfoil)



(a) Effect of ITR (1.01 imbalance tolerance)

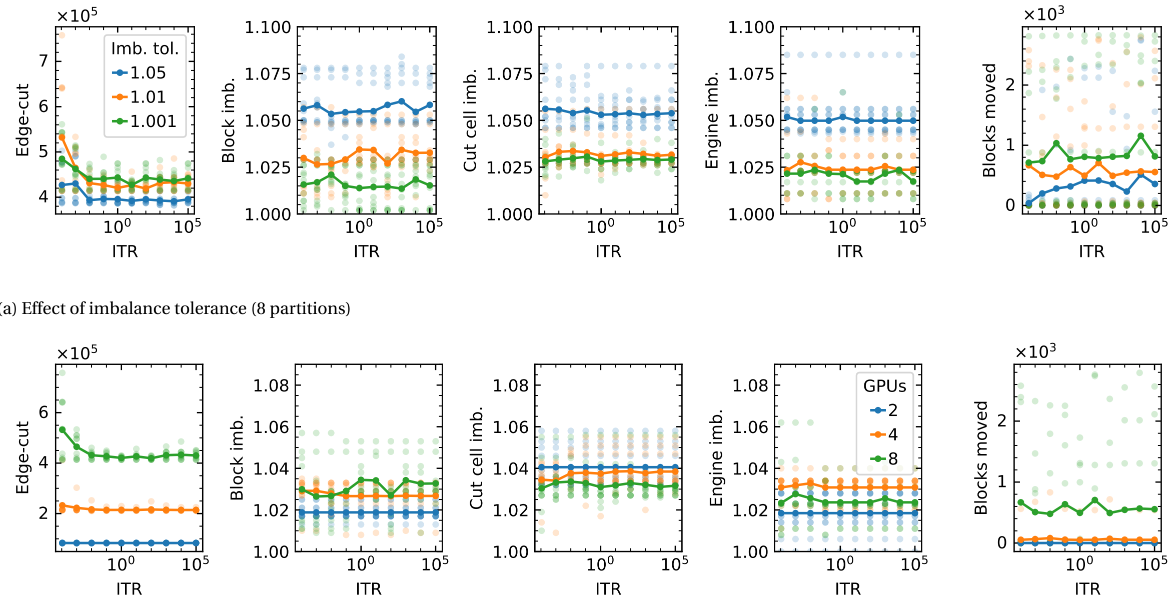
(b) Effect of imbalance tolerance (ITR=0.1)

Figure 4.11: Effect of ITR and imbalance tolerance on the runtime and AMR time (Pitching airfoil)

4.3.2. ROTATING FLAP

The setup has a smaller quasi-2D domain. It contains one static and one moving (rotating) body and an engine volume. A third balancing constraint is used for the engine volume. With this setup the effect of ITR, imbalance tolerance and the number of partitions is explored.

As previously seen, the lowest edge-cut can be achieved with the biggest imbalance tolerance (Fig. 4.12a), but in this setup it simultaneously reduces the number of moved blocks. This does not minimise the runtime (Fig. 4.13a), which is achieved with the tightest imbalance. The trend is the opposite as in the previous setup. This highlights the case-dependent importance of the edge-cut, balance and data transfer. The cut cell and engine volume constraints barely improved with the tightest imbalance tolerance, and due to the coarse mesh around the surfaces and low number of engine blocks the balance of the blocks is the most influential factor.

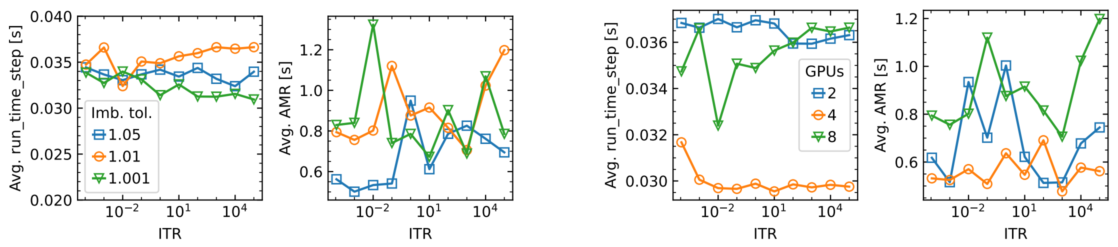


(a) Effect of imbalance tolerance (8 partitions)

(b) Effect of number of partitions (1.01 imbalance tolerance)

Figure 4.12: Average edge-cut and imbalances (Rotating flap)

The edge-cut increases proportionally with the number of partitions (Fig. 4.12b). ITR has a minor effect, except at low values, where it tends to worsen the edge-cut. The trend is different for each balancing constraint, but again the number of partitions is more influential than ITR. An order of magnitude more blocks are moved with 8 partitions. The square domain shape lends itself better to subdividing into two halves or 4 squares, which causes a higher block imbalance and contributes more block movement. In the end, having more computational power gets counterproductive, and in most cases 4 partitions have an up to 20% lower runtime than 8 partitions (Fig. 4.13b). This relatively small example is compute bound, and not memory bound, and this conclusion cannot be extrapolated for larger setups.



(a) Effect of ITR (1.01 imbalance tolerance)

(b) Effect of number of partitions (ITR=0.1)

Figure 4.13: Effect of the imbalance tolerance and number of GPUs on the runtime and AMR times (Rotating flap)

4.3.3. FLAP SLICE

The setup is similar to the rotating flap setup, but it has a larger domain (Table 4.1), the flap is fixed in hover position and the engine volume is replaced by a rotor. Therefore the partitioner uses only 2 balancing constraints, since there is no engine volume in the setup. The trend in edge-cut (Fig. 4.14) by now is familiar from the previous examples. The main difference is, that whereas in the airfoil and rotating flap setup a large part of the geometry was moving, here only the rotor is moving, and the geometry coincides with the initial state with a periodicity of one blade passage. Fig. 4.16–4.17 show two examples with ITR=0.1 and 1.01 imbalance tolerance, where on the left the blades already moved out from their initial blocks and some blocks got refined around the new blade locations and coarsened in the gaps between the blade tips.

One could think, that with so small changes it is not even worth setting up AMR and repartitioning. A uniform mesh refinement cylinder was added around the rotor with the refinement level of the rotor surface, and AMR was turned off. Despite the negligible increase in the number of blocks and an identical time step size this setup had a 10% higher runtime than the worst in Fig. 4.15. This is mainly because the partitioner at the initialisation can take into account only the initial position of the blades to distribute the cut cells between partitions, and as the blades rotate the cut cell distribution is fluctuating. The AMR time is rather small, and not sensitive to the choice of imbalance tolerance and ITR, but some ideas will be introduced in the discussion how the static (no AMR) partitioning could be improved for moving setups with periodic or at least predictable prescribed motion.

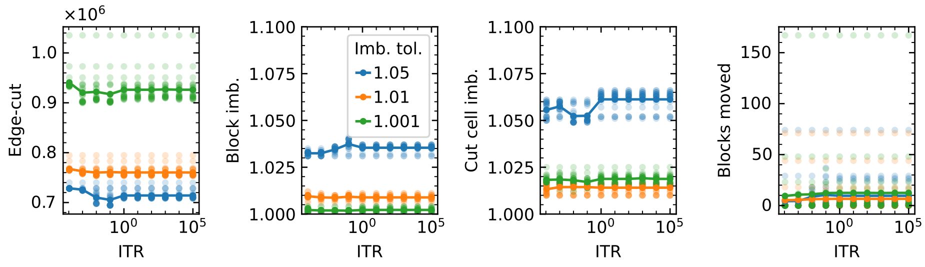


Figure 4.14: Average edge-cut and imbalances (Flap with resolved engine)

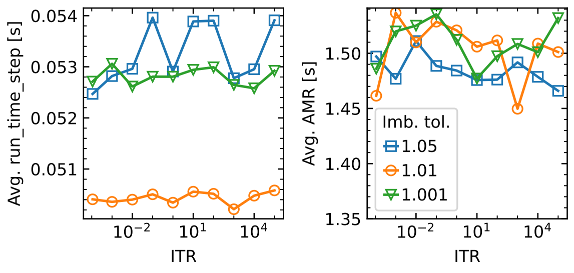
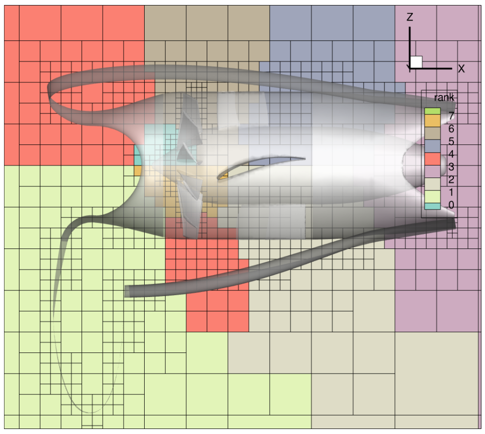


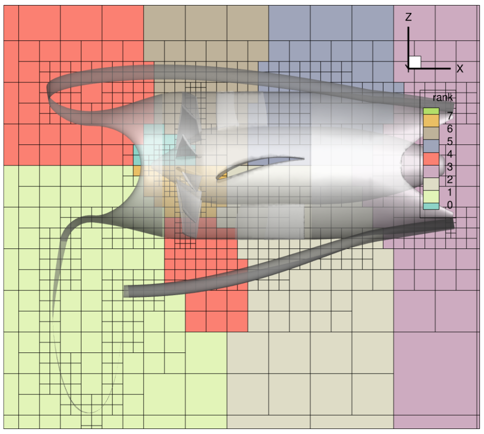
Figure 4.15: Effect of ITR on runtime and AMR time with 1.01 imbalance tolerance (Flap with resolved engine)

4.3.4. AIRCRAFT WIND TUNNEL MODEL

The setup includes an aircraft with 9 resolved engines on the left wing, each similar to the flap slice in the previous example. The rest of the engines are modelled with engine volumes. The setup is the most complex from all examples, not just because of the significantly bigger partitioning graph (due to more blocks), but also because most of the blocks are concentrated around the resolved engines, but the engine volume blocks of the modelled engines also need to be distributed equally between the partitions. Since the rotor geometry and mesh resolution is similar to the Flap with resolved engine setup, in this setup also a low amount of blocks is moved between partitions. The number of moved blocks is half with the lowest ITR compared to the highest ITR, whereas the edge-cut only varies 0.5% (Fig. 4.18). The number of moved blocks barely affects the AMR time (Fig. 4.19) and the runtime also stays almost constant. Unlike in the previous setups, where the AMR time was dominated by the data transfer, here it is only a small fraction.

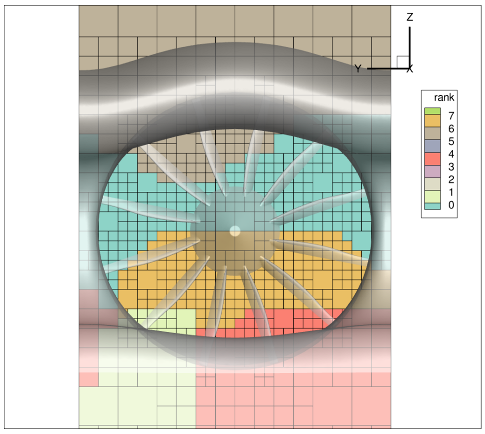


(a) AMR step 3

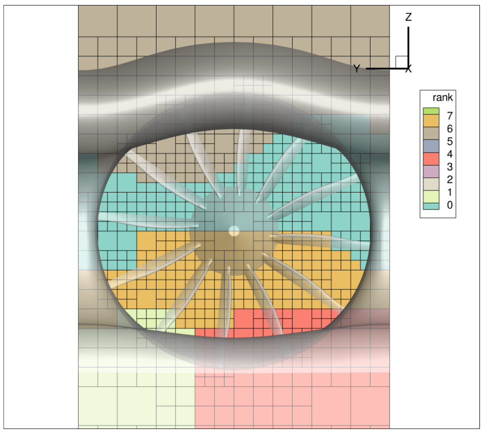


(b) AMR step 30

Figure 4.16: Partitioning of the slice setup in different AMR steps (side view, cut through the symmetry plane)



(a) AMR step 3



(b) AMR step 30

Figure 4.17: Partitioning of the slice setup in different AMR steps (front view, cut through the rotor)

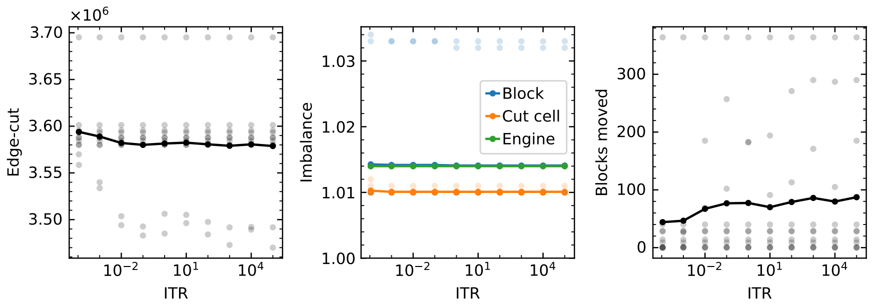


Figure 4.18: Average edge-cut and imbalances (Aircraft wind tunnel model)

Testing if a new partitioning is needed (AMR criterion, denoted as *Criterion* in the legend of Fig. 4.19) is the most expensive, and it takes about 1/3 of the total AMR time. The average time of it is about 1/4, but this step is executed in regular intervals. The subsequent AMR steps are only carried out if the Z-ordering of the new octree mesh is different from the current one, so it can be called more often than the rest of AMR, especially if the CFL number is not driven by the relative speed of the moving geometry. Establishing the intra- and inter-partition connections between the blocks (*Connect*), the re-partitioning of the graph itself (*Re-partition*), and writing an output file (*Write*) of the partition also take significant time, together constituting about 80% of the AMR step. Writing the output file is optional, and this cost can be avoided, if only the simulation results are

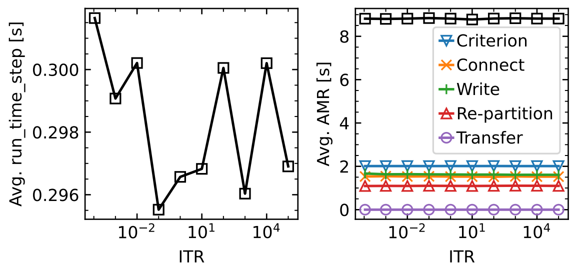


Figure 4.19: Effect of ITR on runtime and AMR time with 1.01 imbalance tolerance (Aircraft wind tunnel model)

needed. Running this setup without AMR could perhaps lead to a different conclusion about the necessity of AMR, but AMR was only 5% of the total runtime, therefore only minor gains could be achieved by this.

4.4. INITIAL REFINEMENT

Partitioning algorithms in general do not provide an optimal partitioning quality. In an adaptive setup, the subsequent re-partition steps guide the partition graph towards better quality. However, during the AMR steps this comes with a repartitioning cost. Instead, the k-way refinement algorithm is applied iteratively on the initial solution, until the edge-cut converges. Since at this stage the data of the mesh blocks is not assigned to partitions, it comes with no performance penalty to change the graph. The target is to minimise the edge-cut given the balancing constraints, and the ITR factor does not play a role in the optimisation objective function. The time of this initial iteration is usually negligible (see Appendix B). Although in the present work only moving setups with AMR are investigated, this refinement is also capable of improving the balancing of static setups.

On small setups, such as the rotating flap, the refinement loop converges in only 1–2 steps. In this setup the partitioning had only minor changes (Fig. 4.20–4.21), except at a high imbalance tolerance, for which the edge-cut was improved by 6% in exchange for making it less balanced (but still within tolerance). With a tight imbalance tolerance both the edge-cut and cut cell imbalance increased, with a similar decrease in block imbalance.

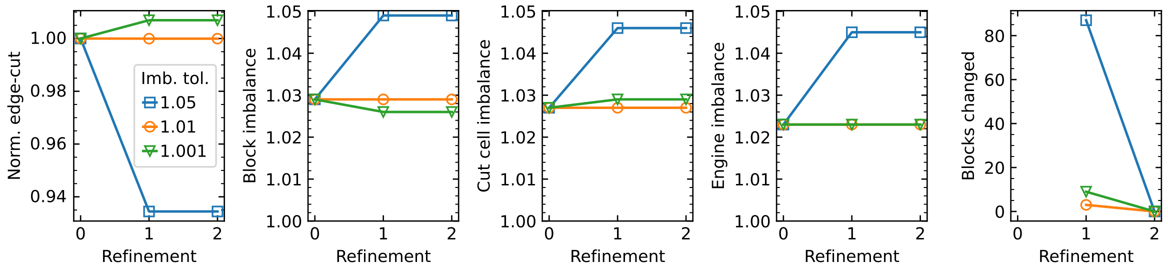


Figure 4.20: Effect of imbalance tolerance on initial refinement (Rotating flap)

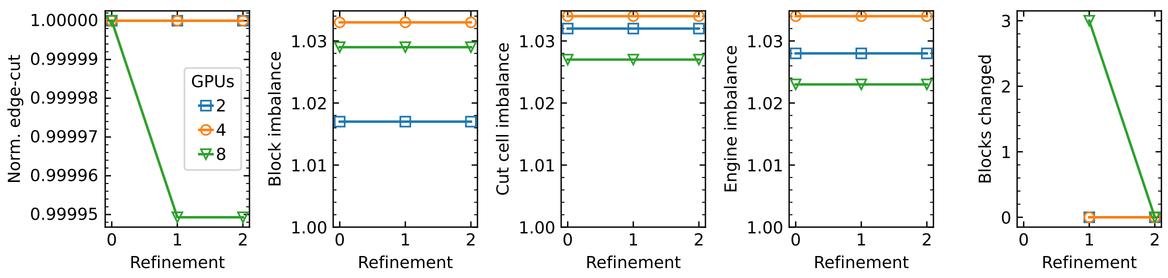


Figure 4.21: Effect of the number of partitions on initial refinement (Rotating flap)

In the flap with resolved engine setup, which has a larger partitioning graph, the edge-cut improved by up to 9% with only small changes in imbalances (Fig. 4.22).

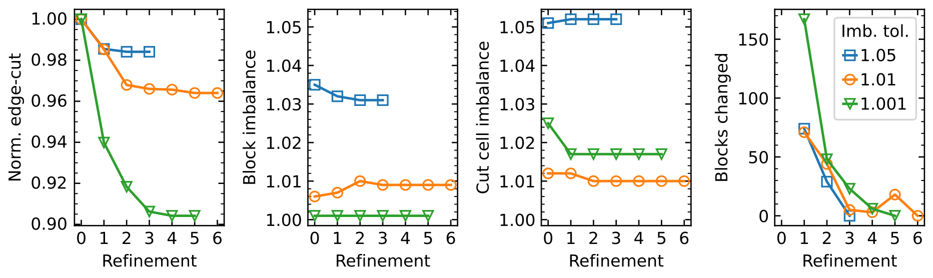


Figure 4.22: Effect of imbalance tolerance on initial refinement (Flap with resolved engine)

The effect of the mesh size is investigated with the wind tunnel setup. The meshes are described in detail in 7.6, but it suffices to know that the 9 engine case is even larger than the *fine* mesh, and the *coarse* and 9 engine case have more uniform surface refinement levels, whereas in the *baseline* and *fine* mesh the refinement is concentrated in a small part of the domain, making them more challenging for the partitioner. All of them have a 1.01 imbalance tolerance and the domain is subdivided into 8 partitions. The *coarse* and 9 engine case show a 6% and 10% edge-cut reduction, and they need the most refinement steps before convergence. In the large partitioning graph of these setups the edge-cut is improved considerably while the imbalance is approximately held on the same level.

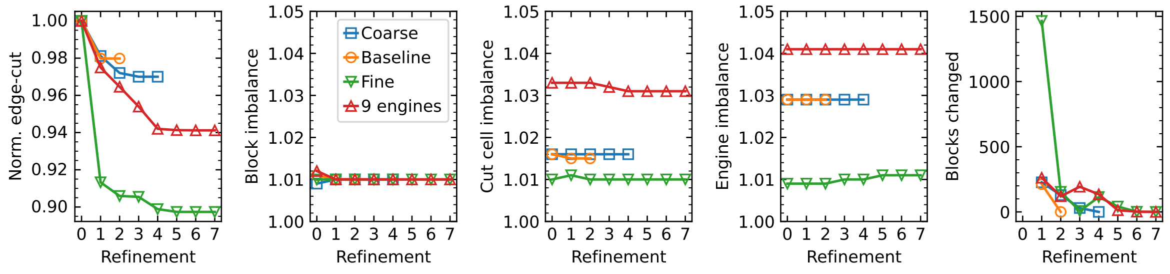


Figure 4.23: Effect of imbalance tolerance on initial refinement (Aircraft wind tunnel model)

The initial refinement, depending on the case, can improve the edge-cut by 10%. When the refinement is not increasing the partitioning quality, it converges in a few steps. Therefore it is worth using it for all cases, because it either manages to reduce the runtime, or it has a negligible overhead.

4.5. Discussion

4.5.1. ITR and imbalance tolerance parameter values

Based on the studies with different setups (full 3D and quasi 2D, with 2 and 3 balancing constraints), we find that the best value for imbalance tolerance and ITR factor varies case by case. A value of ITR=0.1 and 1.01 imbalance tolerance was chosen as baseline for the simulations in the following chapters, with the aim to provide both efficient and robust values for the aircraft setup. This setup is relatively insensitive to ITR, and with higher imbalance tolerances the memory need by a partition could occasionally exceed the hardware limitations, leading to failed simulations.

The choice of these parameters can influence the runtime more than 10%, therefore making them adaptive could bring further improvement and ensure good performance on a wide range of setups. This could be achieved in two ways:

- By collecting data from a larger number of simulations and establishing correlations with geometric and mesh properties (e.g. domain aspect ratios or ratio of cut blocks to all blocks), and setting the parameter values at initialisation.

- Use the timers of different kernels to estimate inter-partition communication cost and redistribution cost at runtime, and adaptively adjust the parameters during the run at regular intervals. A similar approach is used in Zoltan [58] and DRAMA [59].

4.5.2. LOAD BALANCING WITH SIMPLE PRESCRIBED PATHS

URA proved to be a robust and efficient solution for load balancing a diverse set of moving setups. It is a special case, when most of the domain is static, and the moving geometry follows a prescribed path. In engine simulations with an otherwise static simulation the motion is even simpler, since it is periodic with the blade passing frequency. The flap with resolved engine example showed that the initial partitioning is insufficient to create a well-balanced partitioning at all rotor position angles. The balancing quality would drop even further with a lower blade count (especially with the typical 3–5 blades of wind turbines and propellers). An alternative would be to use a cylindrical bounding box volume revolved around the axis of rotation, and split it among the partitions. Splitting only by the number of blocks would be insufficient, since this would not necessarily provide balanced cut cells, and the angular distribution of the blocks would need to be considered. Other motions, such as a simple translation could be statically load balanced analogously. It needs further investigation, whether the increase in mesh size and potentially higher edge-cut is offset by the absence of AMR steps.

4.5.3. MULTI-CONSTRAINED BALANCING

In the ParMETIS library the wavefront diffusion algorithm can handle only 2 balancing constraints. Despite this, the setups including an engine volume block balancing constraint (so 3 in total) also performed well. This is, because the LMSR algorithm (the only one available for more than 2 constraints), unlike other scratch remap methods, respects the original partition boundaries during the coarsening phase. This finding encourages the addition of further balancing constraints, e.g. to equally distribute the moving cut cells (and not just the total cut cells) among partitions, which could be beneficial for setups including a mix of moving and static bodies.

5
Voxelizer

5.1. Introduction

Voxelization means creating a discrete representation of a body, where each voxel stores geometric information. A surface voxelization is used to identify cut blocks. Cut blocks are voxels, which contain inside them or within some proximity of the triangles of the surface. This binary value (true if the block is cut) is used to construct the octree mesh. The voxelizer algorithm is adapted from [60]. The conservative approach is used, where a voxel is marked as cut, if it has any overlap with a triangle of a body.

Beyond the binary criterion to identify cut blocks, the voxelizer is also used to cluster triangles of the geometry to mesh blocks. This ensures the fast look-up of faces in the proximity of cells in subsequent steps. Counting the faces within mesh blocks also helps to distribute the workload between partitions (Chapter 4).

An important part of the voxelizer algorithm is the triangle and box overlap test, based on the separating axis theorem [61]. In this solver the voxelizer is used in five different modes. It is called 3 times during the initialisation and AMR steps and twice in the `update_flag` function during each iteration of moving setups. The first three voxelizer calls are a small fraction of the total runtime. Call 1 and 2 have a similar function to Call 4. Call 3 in the initialisation is analogous to Call 5 in the `update_flag`. Call 4 and 5 are preceding the SDF updates (Chapter 6). An overview is provided in Table 5.1.

	Call	Purpose
	1	Tags cut blocks.
Initialisation	2	Counts the number of cut triangles in each enlarged block.
	3	Stores triangle IDs cutting each block in a linear array.
Update (each time step)	4	Tags cut blocks and counts the number of cut triangles in each enlarged block.
	5	Stores triangle IDs cutting each block in a linear array.

Table 5.1: Voxelizer calls

A schematic of the workflow of the baseline and new version is shown in Fig. 5.1. Call 4 is setting the binary flag and counting the number of faces cutting the block. An array is sized based on the total number of cuts, and in Call 5 it is filled with the IDs of the faces.

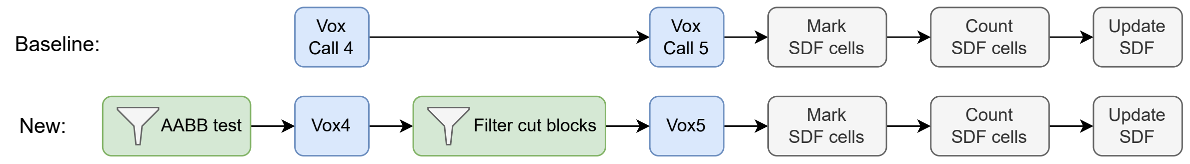


Figure 5.1: Baseline (*top*) and new (*bottom*) `update_flag` workflow

This chapter discusses the improvements within the voxelizer kernels and the filtering of their inputs.

5.2. METHODOLOGY

The baseline voxelizer algorithm is summarised in Alg. 1. The purpose of the voxelizer is to count and collect each surface triangle, that is in the neighbourhood of a block. The kernel launches a thread for each triangle, and the intersection is checked between triangle and block pairs. For this in each thread all local (assigned to the current partition) blocks are evaluated. The overlap checks determine, whether the axis-aligned bounding box (AABB) of the triangle intersects the enlarged block. This step is inexpensive, and for most blocks there is no intersection, and the loop will proceed. The issue with this approach is that for the overlap check the 6 corner coordinates of each block need to be loaded from global memory.

In the revised algorithm (Alg. 2) in Call 4 only the blocks cut by the AABB of the parent body of the triangle are considered (Fig. 5.2). Although this needs an extra overlap check, the amount of tests between bodies and blocks is much lower than between triangles and blocks. A thread is assigned to each block, it loops through the list of bodies, and stores boolean values for the overlaps. The number of overlaps is counted to size the memory allocation of the block index list. In the second pass the list is filled with the overlapping block IDs of each body and is passed to the voxelizer (Call 4).

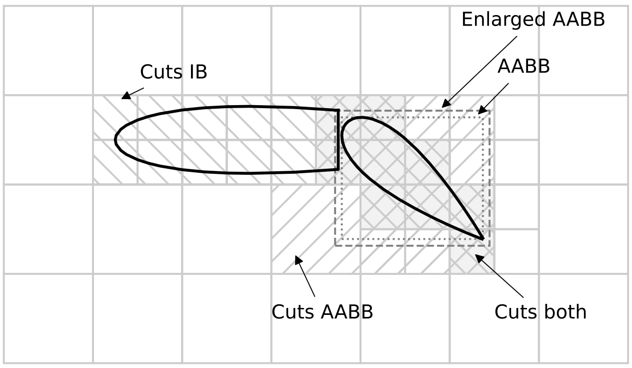


Figure 5.2: Filtering of blocks for voxelizer call 5

This approach is most effective, when the body has a simple shape, and it fills most of the volume of the AABB (e.g. an airfoil). Bodies with more complex topology, like ducted fan nacelles with engines inside, will include many blocks, that will not pass the triangle overlap checks, since they contain large empty spaces or other bodies within the bounding box. The latter further increases the computational cost, since other bodies can have a fine wall mesh resolution, significantly increasing the length of the block list.

In Call 5 the block list can be further reduced, since it is known from Call 4 which blocks are cut. Call 5 takes a block list that is filtered both by AABB overlap and by being cut by any triangle. This filtered blocks are conforming to the body shape, and they constitute the bare minimum that is needed to execute Call 5.

The reduction of the block list proved to be the most effective way to speed up the voxelizer, but minor gains can be achieved also by refactoring the block/triangle overlap tests. Fig. 5.3 illustrates both the bounding box overlap test and the edge projections test.

The projected triangle edge normals are defined in Eq. 5.1, $d_{\mathbf{e}_i}^{xy}$ used in the edge projection test is included in Eq. 5.4. In both equations the xy projection direction is used as an example, and it is similar for $j \in \{xy, yz, zx\}$.

$$\mathbf{n}_{\mathbf{e}_i,y}^{xy} = \left[-\mathbf{e}_{i,y}, \mathbf{e}_{i,x}\right]^T \operatorname{sgn}(\mathbf{n}_z) \tag{5.1}$$

$$\begin{aligned} d_{\mathbf{e}_i}^{xy} &= -\mathbf{n}_{\mathbf{e}_i}^{xy} \cdot \mathbf{v}_{i,xy} + \max\{0, h_{k,x}\mathbf{n}_{\mathbf{e}_i,x}^{xy}\} + \max\{0, h_{k,y}\mathbf{n}_{\mathbf{e}_i,y}^{xy}\} && (5.2)\\ &= -\mathbf{n}_{\mathbf{e}_i}^{xy} \cdot \mathbf{v}_{i,xy} + \left(\max\{0, \mathbf{n}_{\mathbf{e}_i,x}^{xy}\} + \max\{0, \mathbf{n}_{\mathbf{e}_i,y}^{xy}\}\right) h_k && (5.3)\\ &= -a_{\mathbf{e}_i}^{xy} + b_{\mathbf{e}_i}^{xy} h_k && (5.4) \end{aligned}$$

In [61] blocks with arbitrary aspect ratios are considered, but the present solver always uses cubical blocks, which simplifies Eq. 5.4. In the new version $a_{e_i}^{xy} = \mathbf{n}_{\mathbf{e}_i}^{xy} \cdot \mathbf{v}_{i,xy}$ and $b_{\mathbf{e}_i}^{xy} = \max\left\{0, \mathbf{n}_{\mathbf{e}_i,x}^{xy}\right\} + \max\left\{0, \mathbf{n}_{\mathbf{e}_i,y}^{xy}\right\}$ is pre-

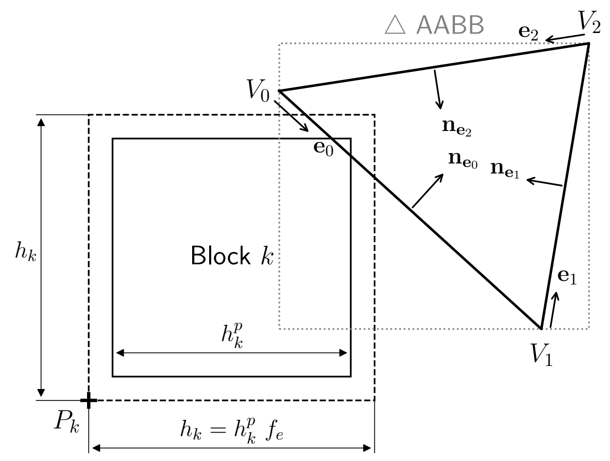


Figure 5.3: Block and triangle overlap and edge projection test (Adapted from [60])

computed before the loop, reducing the computations in exchange for a small increase in the number of registers.

Storing the edge normals $\mathbf{n}_{\mathbf{e}_i}$ was also considered, but the increase in global memory reads offsets the gains of the reduced amount of arithmetic operations. It is not detailed in Alg. 2, but the logic of the overlap and edge checks also changed. In the baseline all 3 overlap checks were tested after all necessary calculations, and similarly the edge checks were done after computing all $d^j_{\mathbf{e}_i}$. In the new version the checks are done one by one. Since the checks are within a loop, this causes thread divergence only on the warp level, and if the triangle IDs are to some extent spatially ordered (as is the case usually in STL files), this can still provide a moderate speed up. The kernel was split up to 3 different functions, for Call 1–3, Call 4 and Call 5. This removes a switch statement from the latter two, and the inputs can be tailored to the specific call.

Various other optimisation options were evaluated, such as reducing the amount of atomic operations with a warp shuffle, changing the order of the edge checks (looping through edges first versus looping through projection directions first), using a larger kernel for triangle-block pairs in order to remove the block loop and launching kernel calls instead of the loop. These proved to be ineffective or detrimental with the tested examples, therefore they were not implemented.

Algorithm 1 Voxelizer (baseline)

```
1: Thread for each △
2: Calc. △ AABB
3: Get $\mathbf{n}$
4: Calc. $\mathbf{e}_i \quad i \in \{0,1,2\}$
5: Calc. $\mathbf{n}_{e_i}$
6: for $k \leftarrow 0, n_{b,local}$ do
7:     Calc. $h_k$
8:     △ AABB-block $k$ overlap test
9:     Calc. $d^j_{\mathbf{e}_i} \quad j \in \{xy, yz, zx\}$
10:    Edge proj. test $\mathbf{n}^j_{\mathbf{e}_i} \cdot \mathbf{P}_k + d^j_{\mathbf{e}_i} < \epsilon$
11:    Mark $b$ as cut (Call 4)
12:    $n_{cuts,k}$ += 1 (Call 4)
13:    Set next △ ID in the list (Call 5)
14: end for
```

Algorithm 2 Voxelizer (new)

```
1: Thread for each △
2: Calc. △ AABB
3: Get $\mathbf{n}$
4: Calc. $\mathbf{e_i} \quad i \in \{0,1,2\}$
5: Calc. $\mathbf{n}_{\mathbf{e}_i}$
6: Calc. $a^j_{\mathbf{e}_i}, b^j_{\mathbf{e}_i} \quad j \in \{xy, yz, zx\}$
7: for $k \leftarrow 0, n_{b,local,filtered}$ do
8:     $k$ = index_list($k$)
9:     Calc. $h_k$
10:    △ AABB-block $k$ overlap test
11:    Edge proj. test $\mathbf{n}^j_{\mathbf{e}_i} \cdot \mathbf{P}_k - a^j_{\mathbf{e}_i} + b^j_{\mathbf{e}_i} h_k < \epsilon$
12:    Mark $b$ as cut (Call 4)
13:    $n_{cuts,k}$ += 1 (Call 4)
14:    Set next △ ID in the list (Call 5)
15: end for
```

5.3. SCALING WITH THE NUMBER OF BLOCKS

Fig. 5.4 shows the effect of changing the number of blocks. The pitching airfoil setup was used (introduced in Section 4.1, see also Section 7.2). The domain size was changed, keeping the mesh resolution around the airfoil constant.

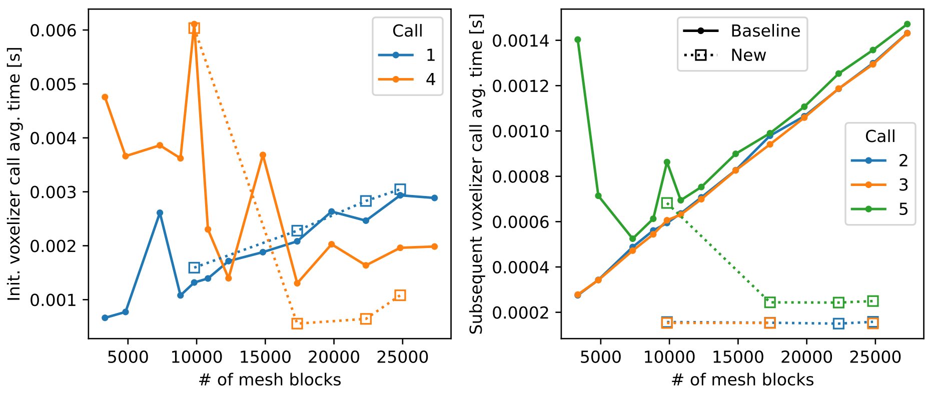


Figure 5.4: Effect of domain size with fixed wall cell size on the initial (*left*) and subsequent (*right*) voxelizer calls (Pitching airfoil)

The initial calls (calling the voxelizer for the first time in the `set_flag` and `update_flag` functions) show a linear behaviour for a large number of blocks, but with fewer blocks both the baseline and new version have a large overhead in Call 4. The exact cause is not known, but the GPUs are underutilised in these cases, and it might be related to a suboptimal selection of the CUDA grid and block size. At large mesh block sizes the new version has an up to 50% better runtime due to the AABB filtering. The subsequent calls show a more striking difference. The baseline scales linearly with the mesh size, whereas the new version has a constant execution time at large mesh sizes. The advantages of the new algorithm can best harnessed in large domains, where either the domain size is big compared to the geometry or it includes volume refinement regions. A complementary study is included in Section 7.6, where the domain size is fixed and the wall cell size is varied. In this case Call 5 scales linearly with the number of mesh blocks.

5.4. DISCUSSION

The voxelizer kernel is accelerated both by the adaption of the blocks and triangle overlap tests to the specifics of the present solver, and to a greater extent by the reduction of the candidate blocks. Fig. 5.5 shows a summary of the speed-up for different test cases. The speed-up is the largest in the setups, where a large fraction of blocks is outside the bounding box of the geometry.

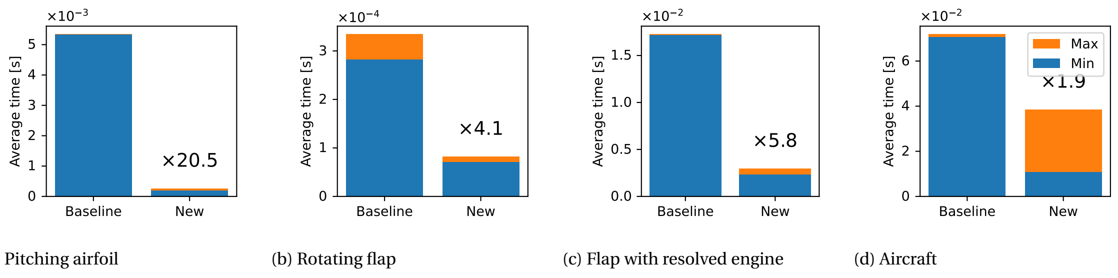


Figure 5.5: Timings of voxelizer kernel Call 4–5 in `update_flag`

Several avenues were explored to further optimise the overlap tests, but this did not result in improvements with the test cases of Chapter 7. The simple AABB block filtering was sufficient to remove the voxelizer from the list of main bottlenecks. It could be further improved by using a body-aligned bounding box (BABB) [62] instead of an AABB. This results in more expensive overlap tests, but better conforms to many common applications, such as rotating motion of slender bodies (pitching airfoil in Chapter 7, or a rotating propeller blade). The shape of more complex geometries, such as aircraft could be better approximated with convex hulls [63, 64] or discrete orientation polytopes (k-DOPs) [65], which form a convex polytope with a prescribed number (k) of faces. These approaches are similar to covering a present in wrapping paper, where for most shapes not much air (in this case uncut mesh blocks) remain within the volume enclosed by the paper.

The special case of ducted fans poses another issue. The engine is represented either by an axisymmetric volume (actuator disc) or includes a rotor with resolved blades, which is enclosed within the duct. With the above approaches, which in essence all represent the geometry with a *convex* bounding volume, the blocks of the engine are included in the block list of the nacelle. The rotor, due to both the surface complexity and thin boundary layer can have a much smaller wall cell size than the nacelle, therefore adding a substantial overhead. Using bounding volume hierarchies (BVHs) [66, 67] could better conform to the geometry and is not limited to convex representations, but it comes with a memory overhead.

6

Signed distance function

6.1. Introduction

When a setup contains moving geometry, the signed distance field needs to be updated at each iteration to correctly mark the cells as fluid, solid, wall model and ghost cells (Fig. 3.2). Furthermore, the SDF is implicitly needed for prescribing boundary conditions through ghost cells. At each iteration the SDF is evaluated on the previous ghost cells and their neighbouring cells (opportunity cells). The opportunity cells are defined as cells, which have at least one interface cell face neighbour. The opportunity and ghost cells (NSDF cells) are marked based on the previous SDF (Fig. 5.1), and a thread is launched for all NSDF cells. For each of the cells the algorithm needs to find the closest triangle and calculate the signed distance to correctly mark the cell type.

6.2. Methodology

The baseline version (Alg. 3) uses the triangle list collected by the voxelizer (Chapter 5) to find the closest triangle to the cell center. The point-to-triangle distance calculation is based on [62]. In the new version (Alg. 4 a simplified version of the distance function is used within the block loop. The baseline distance function returns on top of the squared distance the closest point on the triangle, the hit type (Voronoi region of the point) and other properties, that are needed later. Doing the bare minimum to calculate the distance reduces the number of arithmetic operations and the number of used registers. The disadvantage of this approach is, that the distance calculation has to be repeated with the baseline function after the closest triangle was identified. Since the blocklist often has tens to thousands of elements, this overhead is negligible in most cases.

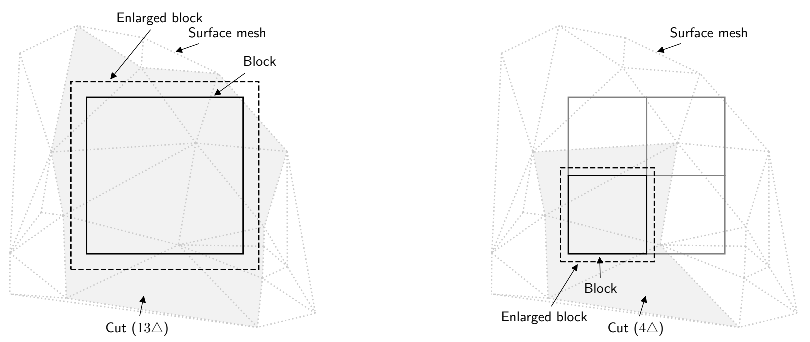
Algorithm 3 Update SDF (baseline)

```
1:  Thread for each NSDF cell
2:  Get block of the cell
3:  Get cell center $\mathbf{P}$
4:  Get cut triangle list of block
5:  $d^2_{min} = \infty$
6:  for $i \leftarrow 0, n_{cut\triangle,b}$ do
7:      Get $\triangle$ vertices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$
8:      $(d^2_i, ID, \mathbf{P}_{hit}, ...) = \text{dist\_to\_tri}(\mathbf{P}, \mathbf{A}, \mathbf{B}, \mathbf{C})$
9:      if $d^2_i < d^2_{min}$ then
10:         $(d^2_{min}, ID^*, \mathbf{P}^*_{hit}, ...) = (d^2_i, ID, \mathbf{P}_{hit}, ...)$
11:     end if
12: end for
13: $\mathbf{n} = \text{get\_normal}(\text{ID}^*, ...)$
14: $SDF = \text{sgn}\left((\mathbf{P} - \mathbf{P}^*_{hit}) \cdot \mathbf{n}\right) \sqrt{d^2_{min}}$
15: if $SDF < 0$ then
16:     Set target flag
17: else
18:     Unset target flag
19: end if
```

Algorithm 4 Update SDF (new)

```
1:  Thread for each NSDF cell
2:  Get block of the cell
3:  Get cell center $\mathbf{P}$
4:  Get cut triangle list of block
5:  $d^2_{min} = \infty$
6:  for $i \leftarrow 0, n_{cut\triangle,b}$ do
7:      Get $\triangle$ vertices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$
8:      $d^2_i = \text{quick\_dist\_to\_tri}(\mathbf{P}, \mathbf{A}, \mathbf{B}, \mathbf{C})$
9:      if $d^2_i < d^2_{min}$ then
10:         $d^2_{min} = d^2_i$
11:         $ID^* = ID_i$
12:     end if
13: end for
14: From $ID^*$ get closest $\triangle$ vertices $\mathbf{A}^*$, $\mathbf{B}^*$ and $\mathbf{C}^*$
15: $(d^2_{min}, ID^*, \mathbf{P}^*_{hit} ...) = \text{dist\_to\_tri}(\mathbf{P}, \mathbf{A}^*, \mathbf{B}^*, \mathbf{C}^*)$
16: $\mathbf{n} = \text{get\_normal}(\text{ID}^*, ...)$
17: $SDF = \text{sgn}\left((\mathbf{P} - \mathbf{P}^*_{hit}) \cdot \mathbf{n}\right) \sqrt{d^2_{min}}$
18: if $SDF < 0$ then
19:     Set target flag
20: else
21:     Unset target flag
22: end if
```

The length of the block list depends on the surface triangulation and the mesh resolution at the wall. If the wall cell size is decreased, less triangles are intersecting the enlarged block (Fig. 6.1). In both the baseline and new algorithm the block is enlarged, since the closest triangle of interface cells can be located in a neighbour block (see Fig. 3.2).



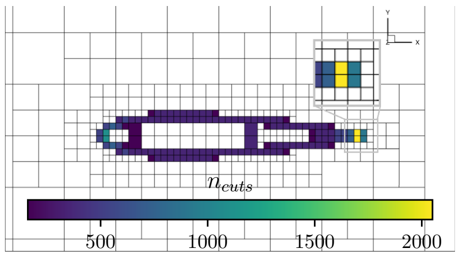
(a) Coarse (b) Fine (coarse block subdivided)

Figure 6.1: Triangle list of a block with different mesh refinement levels

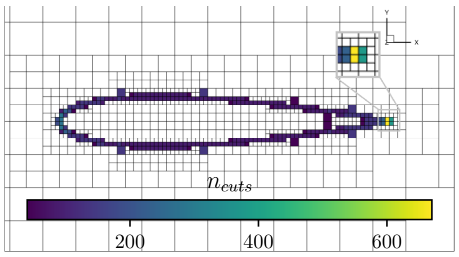
Fig. 6.2 shows an example mesh, where a two level refinement (2^2 times smaller wall cell size) results in an order of magnitude larger block list. In this case most of the cut blocks still have a low number of candidate triangles, but at the trailing edge the triangle list is excessive due to a locally dense surface mesh. This highlights, that even small geometric features can cause bottlenecks.

6.3. SURFACE NORMALS

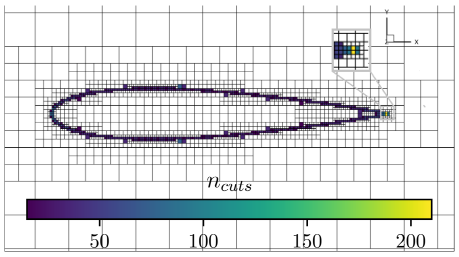
For static setups the SDF only needs to be constructed once at initialisation, and it stays constant throughout the simulation. The baseline is to construct the SDF field anew at each iteration. For rigid bodies a simpler approach is to rotate the SDF field as the body moves, and translating motions do not even need an update of the pseudo-normals.



(a) Level 3



(b) Level 4



(c) Level 5

Figure 6.2: Number of cut cells per block on 3 different meshes

6.3.1. INITIAL PSEUDO-NORMAL CALCULATION

In the present solver bodies are represented as triangulated surfaces. Since at the vertices and edges the surface is not C^1 continuous, using only the triangle normals would be insufficient to unambiguously define the signed distance everywhere. There are many methods to construct pseudo-normals, but angle-weighted pseudo-normals have the advantage that they can always correctly determine if a point is inside or outside a body, and they are not sensitive to the tessellation. The construction of the angle-weighted pseudo-normals (Eq. 6.1) is based on [68]. In Eq. 6.1 α_i are the angles of the incident faces of a vertex. In case of edges the weights are π (so the pseudo-normal is the average of the adjacent face normals), and for faces it is 2π, giving the face normal itself.

$$\mathbf{n}_\alpha = \sum \alpha_i \mathbf{n}_i \tag{6.1}$$

For a given point in space the pseudo-normal of the closest element (vertex, edge or face) of the surface mesh is used. The value of the pseudo-normal in the whole 3D Cartesian space gives the SDF, which is negative inside bodies, and positive outside.

To calculate the pseudo-normals, a thread is launched for each triangle. First the thread calculates the face normal, and thereafter the contribution $\alpha_i \mathbf{n}_i$ is added to the 3 edges and 3 vertices that connect to the triangle. Since the same edge and vertex normals are updated by several threads, atomic additions are used. Atomics lock the memory for a read-modify-write operation, during which other threads cannot access the data. This ensures that the sum is calculated correctly, but it can become a bottleneck, if many triangles share a vertex, because the contributions have to be added sequentially.

6.3.2. UPDATE OF THE PSEUDO-NORMALS

The orientation of the bodies in space is described with the Euler parameters (Eq. 6.2). A complete description of the coordinate system convention and the relation between Euler angles and Euler parameters can be found in [41].

$$\theta = \begin{bmatrix} \theta_0 & \theta_1 & \theta_2 & \theta_3 \end{bmatrix}^T \tag{6.2}$$

The transformation matrix Eq. 6.3 is based on [69]:

$$\mathbf{A} = \begin{bmatrix} 2\left[\theta_0^2+\theta_1^2\right]-1 & 2\left(\theta_1\theta_2-\theta_0\theta_3\right) & 2\left(\theta_1\theta_2-\theta_0\theta_3\right) \\ 2\left(\theta_1\theta_2-\theta_0\theta_3\right) & 2\left[\theta_0^2+\theta_2^2\right]-1 & 2\left(\theta_2\theta_3-\theta_0\theta_1\right) \\ 2\left(\theta_1\theta_3-\theta_0\theta_2\right) & 2\left(\theta_2\theta_3+\theta_0\theta_1\right) & 2\left[\theta_0^2+\theta_3^2\right]-1 \end{bmatrix} \tag{6.3}$$

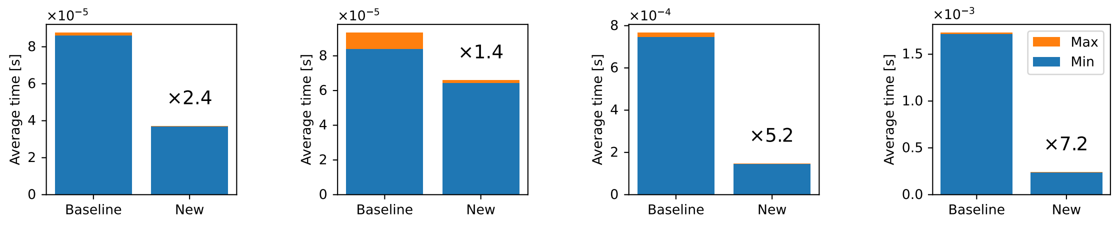
The normal vector of a face, edge or vertex at an iteration is calculated from the current transformation matrix of its parent body and the initial normal vector. The initial normal vectors $\mathbf{n}_0$ are extracted once before applying an Euler transformation and stored in memory.

In iterations, where $\theta \neq \theta_0$, the pseudo-normals are calculated as:

$$\mathbf{n} = \mathbf{A}\mathbf{n_0} \tag{6.4}$$

The main advantage of this method compared to just using the initial normal vector calculation in each iteration is, that it does not use any atomic operations. A vertex can have an arbitrarily large number of face neighbours (e.g. the tip of a cone). This creates a bottleneck in the atomic addition that calculates the angle-weighted vertex normal vector.

Fig. 6.3 shows a summary of the runtime of different setups. The setups are described in detail in Chapter 7, but for this evaluation it suffices to know, that the pitching airfoil setup has a simple quasi-2D geometry with a regular surface mesh topology (any vertex has 6 adjacent faces, except vertices on the periodic boundary), and the other setups have increasingly larger and more complex surface meshes.



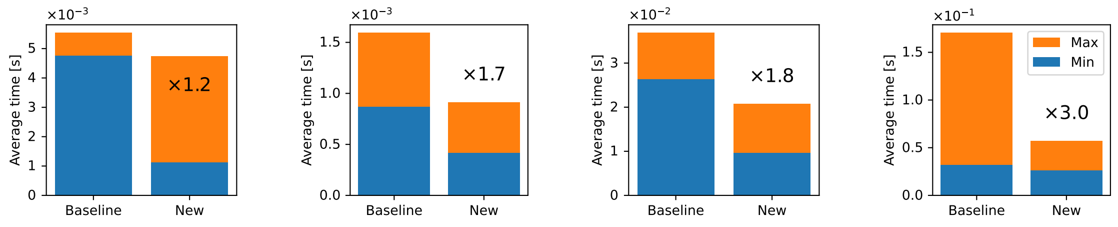
(a) Pitching airfoil (b) Rotating flap (c) Flap with resolved engine (d) Aircraft

Figure 6.3: Timings of STL normal calculations

The pseudo-normal update with the rotation matrix is up to 7 times faster than using the pseudo-normal construction. Surprisingly, the speed-up is the lowest in the rotating flap setup. The least benefit was anticipated for the pitching airfoil, which is not expected to have so severe bottlenecks from the atomic addition due to the regular vertex connectivity. The two geometries have a similar size (pitching airfoil has 362 000 faces and rotating flap around 250 000), but the latter consists of four separate bodies.

6.4. Discussion

Using a simpler and faster point-to triangle distance calculation leads to an up to 3× faster `update_sdf`. The highest gains are achieved in the most complex case. A summary is shown in Fig. 6.4 and Chapter 7 contains a more detailed analysis.



(a) Pitching airfoil (b) Rotating flap (c) Flap with resolved engine (d) Aircraft

Figure 6.4: Timings of `update_sdf` calculations

Furthermore, updating the pseudo-normals with an Euler transformation reduces the runtime up to 4%. Despite this, `update_sdf` remains time consuming, especially in simulation setups with a dense surface mesh and coarse volume mesh. Potential improvements to the point-to-triangle distance calculation are further discussed in Appendix C and a new algorithm with a reduced block list is proposed in Appendix D.

7 CODE OPTIMISATION RESULTS

7.1. INTRODUCTION

In this chapter four simulation setups are described and their results presented, to demonstrate the effect of the changes introduced in the previous chapters. The focus is on performance, and the simulation results are not discussed here. In each setup the geometry and mesh is described, the runtime of the main components of the code is compared between the baseline and the new version, and the code execution is analysed with the Nvidia Nsight Systems[1] and Compute[2] profiling tools.

The four setups are chosen to represent different relevant use cases with increasing complexity. The first setup is an airfoil simulation (Section 7.2), where the whole geometry is moving. The domain is large, with a fine volume mesh resolution and a relatively coarse surface mesh. The second setup (Section 7.3) is also a wing section, but with a flap mounted fan modelled with a volume source term. In this setup the effect of an engine volume on the AMR balancing is investigated. The flap is deflected with a constant rate, while the wing is static. The third setup is also a wing section with periodic boundaries, but with a resolved engine and a more refined mesh (Section 7.4). The final setup (Section 7.5) is a full-body canard aircraft with resolved engines on the left wing and modelled engines on the right wing and on the canards. This setup therefore includes both moving geometry (the spinning engines) and an engine volume balancing constraint. The results on 3 different meshes are compared to evaluate the effect of the total cell count on the performance.

7.2. PITCHING AIRFOIL

A NACA 0015 airfoil is rotating around the quarter-chord line (Fig. 7.1) with 30°/s rate. The setup is similar as in [41, 70, 71]. The domain is quasi 2D with periodic boundaries on the sides. The domain consists of many grid blocks (Table 7.1), which results in a relatively fine mesh resolution in the far field. The surface refinement level can be converted into wall cell size (WCS) with Eq. 7.1, where N is the number of cells per block in one direction (16 or 32 in the setups used in this chapter) and l_{cube} is the the edge length of a block.

$$WCS = \frac{l_{cube}}{N \cdot 2^{level}} \tag{7.1}$$

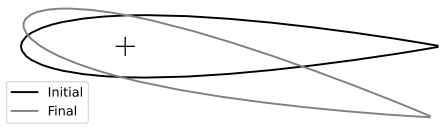


Figure 7.1: Pitching airfoil setup

Compared to the baseline solver, the previously discussed code optimisations result in a 7 times faster simulation (Fig. 7.2). Most of this is thanks to the 48× reduction in AMR, and is related to the reduced re-partitioning cost. The setup is quasi-2D, and as such the partition boundaries have a small area. Therefore

[1]https://developer.nvidia.com/nsight-systems
[2]https://developer.nvidia.com/nsight-compute

Parameter	Value
GPUs	4 × Nvidia V100
Cells per block	32^3
Domain size	$[200, 100, 1] \cdot 0.2 \cdot c$
Surface refinement level	3
Blocks	22×10^3
Cells	731×10^6
Cut cells	0.75×10^6

Table 7.1: Pitching airfoil settings and mesh properties

the edge-cut is less critical in this setup, because data is exchanged on small stripes between the partitions (Fig. 4.2). On the other hand, the fine volume mesh makes it expensive to move large chunks of the volume between partitions, which was a major bottleneck in the baseline code.

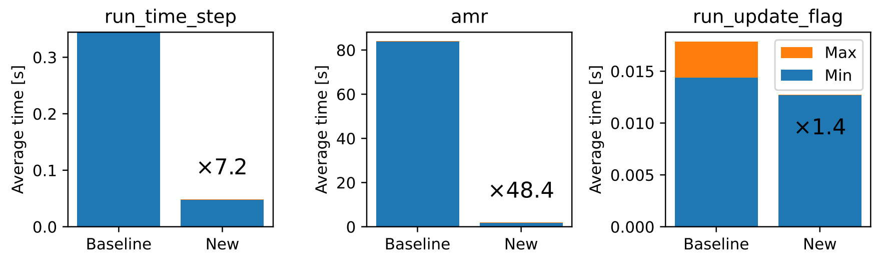
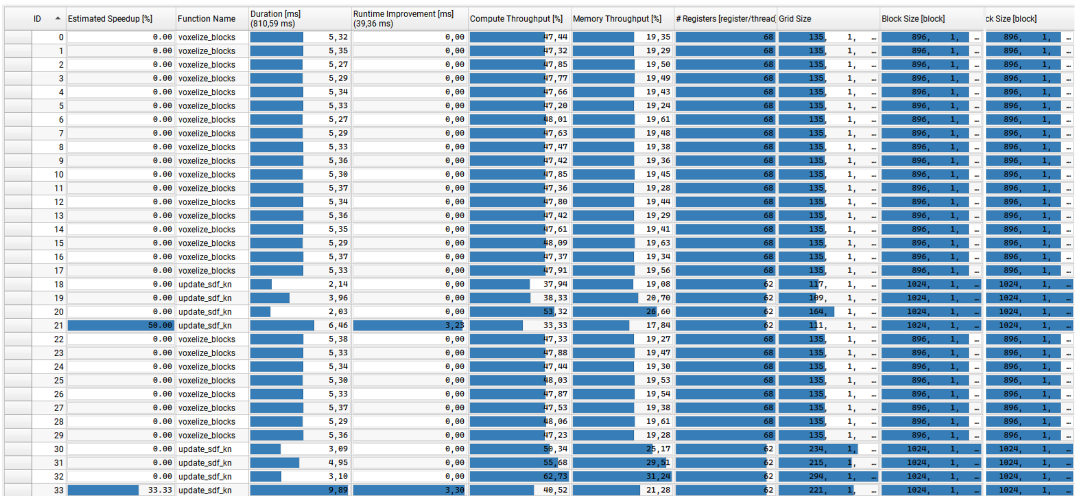


Figure 7.2: Timings (Pitching airfoil)

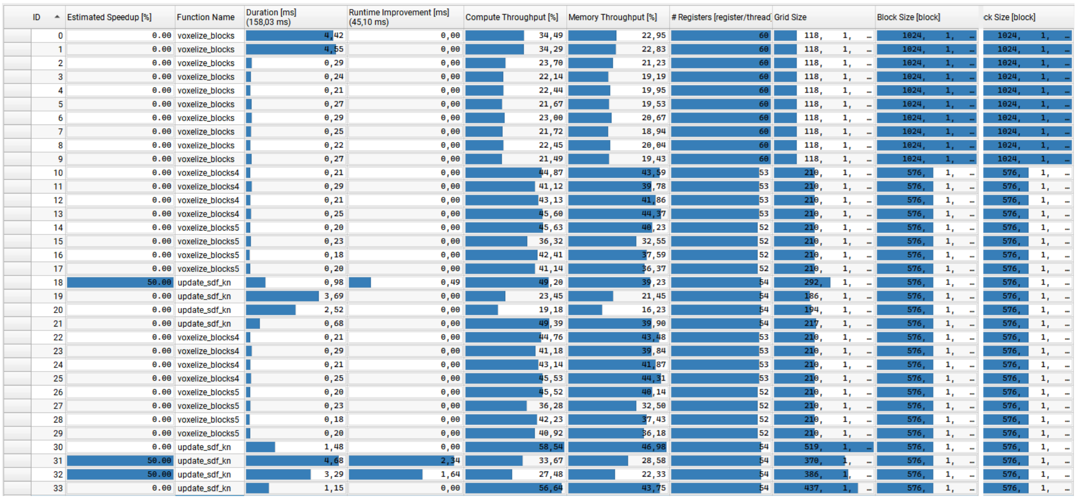
The `update_flag` function also improved by 40%. The first call of this function is broken down in the bottom part of Fig. 7.3–7.4 (starting on line 10). Despite the reduced register usage, the block size is unexpectedly constrained to a lower value in the new version. This was observed also in the other test cases.



ID	Estimated Speedup [%]	Function Name	Duration [ms] (810,59 ms)	Runtime Improvement [ms] (39,36 ms)	Compute Throughput [%]	Memory Throughput [%]	# Registers [register/thread]	Grid Size	Block Size [block]	ck Size [block]
0	0.00	voxelize_blocks	5,32	0,00	47,44	19,35	68	135, 1, …	896, 1, …	896, 1, …
1	0.00	voxelize_blocks	5,35	0,00	47,32	19,29	68	135, 1, …	896, 1, …	896, 1, …
2	0.00	voxelize_blocks	5,27	0,00	47,85	19,50	68	135, 1, …	896, 1, …	896, 1, …
3	0.00	voxelize_blocks	5,29	0,00	47,77	19,49	68	135, 1, …	896, 1, …	896, 1, …
4	0.00	voxelize_blocks	5,34	0,00	47,66	19,43	68	135, 1, …	896, 1, …	896, 1, …
5	0.00	voxelize_blocks	5,33	0,00	47,20	19,24	68	135, 1, …	896, 1, …	896, 1, …
6	0.00	voxelize_blocks	5,27	0,00	48,01	19,61	68	135, 1, …	896, 1, …	896, 1, …
7	0.00	voxelize_blocks	5,29	0,00	47,63	19,48	68	135, 1, …	896, 1, …	896, 1, …
8	0.00	voxelize_blocks	5,33	0,00	47,47	19,38	68	135, 1, …	896, 1, …	896, 1, …
9	0.00	voxelize_blocks	5,36	0,00	47,42	19,36	68	135, 1, …	896, 1, …	896, 1, …
10	0.00	voxelize_blocks	5,30	0,00	47,85	19,45	68	135, 1, …	896, 1, …	896, 1, …
11	0.00	voxelize_blocks	5,37	0,00	47,36	19,28	68	135, 1, …	896, 1, …	896, 1, …
12	0.00	voxelize_blocks	5,34	0,00	47,80	19,44	68	135, 1, …	896, 1, …	896, 1, …
13	0.00	voxelize_blocks	5,36	0,00	47,42	19,29	68	135, 1, …	896, 1, …	896, 1, …
14	0.00	voxelize_blocks	5,35	0,00	47,61	19,41	68	135, 1, …	896, 1, …	896, 1, …
15	0.00	voxelize_blocks	5,29	0,00	48,09	19,63	68	135, 1, …	896, 1, …	896, 1, …
16	0.00	voxelize_blocks	5,37	0,00	47,37	19,34	68	135, 1, …	896, 1, …	896, 1, …
17	0.00	voxelize_blocks	5,33	0,00	47,91	19,56	68	135, 1, …	896, 1, …	896, 1, …
18	0.00	update_sdf_kn	2,14	0,00	37,94	19,08	62	117, 1, …	1024, 1, …	1024, 1, …
19	0.00	update_sdf_kn	3,96	0,00	38,33	20,70	62	109, 1, …	1024, 1, …	1024, 1, …
20	0.00	update_sdf_kn	2,03	0,00	53,32	26,60	62	164, 1, …	1024, 1, …	1024, 1, …
21	50.00	update_sdf_kn	6,46	3,23	33,33	17,84	62	111, 1, …	1024, 1, …	1024, 1, …
22	0.00	voxelize_blocks	5,38	0,00	47,33	19,27	68	135, 1, …	896, 1, …	896, 1, …
23	0.00	voxelize_blocks	5,33	0,00	47,88	19,47	68	135, 1, …	896, 1, …	896, 1, …
24	0.00	voxelize_blocks	5,34	0,00	47,44	19,30	68	135, 1, …	896, 1, …	896, 1, …
25	0.00	voxelize_blocks	5,30	0,00	48,03	19,53	68	135, 1, …	896, 1, …	896, 1, …
26	0.00	voxelize_blocks	5,33	0,00	47,87	19,54	68	135, 1, …	896, 1, …	896, 1, …
27	0.00	voxelize_blocks	5,37	0,00	47,53	19,38	68	135, 1, …	896, 1, …	896, 1, …
28	0.00	voxelize_blocks	5,29	0,00	48,06	19,61	68	135, 1, …	896, 1, …	896, 1, …
29	0.00	voxelize_blocks	5,36	0,00	47,23	19,28	68	135, 1, …	896, 1, …	896, 1, …
30	0.00	update_sdf_kn	3,09	0,00	50,34	25,17	62	234, 1, …	1024, 1, …	1024, 1, …
31	0.00	update_sdf_kn	4,95	0,00	55,68	29,51	62	215, 1, …	1024, 1, …	1024, 1, …
32	0.00	update_sdf_kn	3,10	0,00	62,73	31,24	62	294, 1, …	1024, 1, …	1024, 1, …
33	33.33	update_sdf_kn	9,89	3,30	40,52	21,28	62	221, 1, …	1024, 1, …	1024, 1, …

Figure 7.3: Profiling of the baseline version in CUDA Nsight Compute (Pitching airfoil)

Since an AABB closely fits an airfoil section (especially at the start of the simulation, without pitch), the new voxelizer algorithm greatly reduces the block list already in call 4. Call 5 filters out further blocks, that are not cut, but this only results in negligible time reduction compared to Call 4. Both of them are around 25× faster than the baseline voxelizer. The `update_sdf` function became twice as fast, but remained imbalanced (see Min and Max in Fig. 7.2, which highlights the average time of the slowest and fastest GPU). This is due to the dense surface triangulation near the TE. The locally dense surface mesh increases the cut triangle list of the blocks near the TE surface. The issue is detailed in Appendix D and shown in Fig. D.7. In this setup `update_sdf` remains the biggest bottleneck, which could be remedied either by local volume refinement near the TE or a coarsening of the surface triangulation. The latter would be the preferred option, since it does



ID	Estimated Speedup [%]	Function Name	Duration [ms] (158,03 ms)	Runtime Improvement [ms] (45,10 ms)	Compute Throughput [%]	Memory Throughput [%]	# Registers [register/thread]	Grid Size	Block Size [block]	ck Size [block]
0	0.00	voxelize_blocks	4,42	0,00	34,49	22,95	60	118, 1, …	1024, 1, …	1024, 1, …
1	0.00	voxelize_blocks	4,55	0,00	34,29	22,83	60	118, 1, …	1024, 1, …	1024, 1, …
2	0.00	voxelize_blocks	0,29	0,00	23,70	21,23	60	118, 1, …	1024, 1, …	1024, 1, …
3	0.00	voxelize_blocks	0,24	0,00	22,14	19,19	60	118, 1, …	1024, 1, …	1024, 1, …
4	0.00	voxelize_blocks	0,21	0,00	22,44	19,95	60	118, 1, …	1024, 1, …	1024, 1, …
5	0.00	voxelize_blocks	0,27	0,00	21,67	19,53	60	118, 1, …	1024, 1, …	1024, 1, …
6	0.00	voxelize_blocks	0,29	0,00	23,00	20,67	60	118, 1, …	1024, 1, …	1024, 1, …
7	0.00	voxelize_blocks	0,25	0,00	21,72	18,94	60	118, 1, …	1024, 1, …	1024, 1, …
8	0.00	voxelize_blocks	0,22	0,00	22,45	20,04	60	118, 1, …	1024, 1, …	1024, 1, …
9	0.00	voxelize_blocks	0,27	0,00	21,49	19,43	60	118, 1, …	1024, 1, …	1024, 1, …
10	0.00	voxelize_blocks4	0,21	0,00	44,87	43,59	53	210, 1, …	576, 1, …	576, 1, …
11	0.00	voxelize_blocks4	0,29	0,00	41,12	39,78	53	210, 1, …	576, 1, …	576, 1, …
12	0.00	voxelize_blocks4	0,21	0,00	43,13	41,86	53	210, 1, …	576, 1, …	576, 1, …
13	0.00	voxelize_blocks4	0,25	0,00	45,60	44,37	53	210, 1, …	576, 1, …	576, 1, …
14	0.00	voxelize_blocks5	0,20	0,00	45,63	40,23	52	210, 1, …	576, 1, …	576, 1, …
15	0.00	voxelize_blocks5	0,23	0,00	36,32	32,55	52	210, 1, …	576, 1, …	576, 1, …
16	0.00	voxelize_blocks5	0,18	0,00	42,41	37,59	52	210, 1, …	576, 1, …	576, 1, …
17	0.00	voxelize_blocks5	0,20	0,00	41,14	36,37	52	210, 1, …	576, 1, …	576, 1, …
18	50.00	update_sdf_kn	0,98	0,49	49,20	39,23	54	292, 1, …	576, 1, …	576, 1, …
19	0.00	update_sdf_kn	3,69	0,00	23,45	21,45	54	186, 1, …	576, 1, …	576, 1, …
20	0.00	update_sdf_kn	2,52	0,00	19,18	16,23	54	194, 1, …	576, 1, …	576, 1, …
21	0.00	update_sdf_kn	0,68	0,00	49,39	39,90	54	217, 1, …	576, 1, …	576, 1, …
22	0.00	voxelize_blocks4	0,21	0,00	44,76	43,48	53	210, 1, …	576, 1, …	576, 1, …
23	0.00	voxelize_blocks4	0,29	0,00	41,18	39,84	53	210, 1, …	576, 1, …	576, 1, …
24	0.00	voxelize_blocks4	0,21	0,00	43,14	41,87	53	210, 1, …	576, 1, …	576, 1, …
25	0.00	voxelize_blocks4	0,25	0,00	45,53	44,31	53	210, 1, …	576, 1, …	576, 1, …
26	0.00	voxelize_blocks5	0,20	0,00	45,52	40,14	52	210, 1, …	576, 1, …	576, 1, …
27	0.00	voxelize_blocks5	0,23	0,00	36,28	32,50	52	210, 1, …	576, 1, …	576, 1, …
28	0.00	voxelize_blocks5	0,18	0,00	42,23	37,43	52	210, 1, …	576, 1, …	576, 1, …
29	0.00	voxelize_blocks5	0,20	0,00	40,92	36,18	52	210, 1, …	576, 1, …	576, 1, …
30	0.00	update_sdf_kn	1,48	0,00	58,54	46,98	54	519, 1, …	576, 1, …	576, 1, …
31	50.00	update_sdf_kn	4,68	2,34	33,67	28,58	54	370, 1, …	576, 1, …	576, 1, …
32	50.00	update_sdf_kn	3,29	1,64	27,48	22,33	54	386, 1, …	576, 1, …	576, 1, …
33	0.00	update_sdf_kn	1,15	0,00	56,64	43,75	54	437, 1, …	576, 1, …	576, 1, …

Figure 7.4: Profiling of the new version in CUDA Nsight Compute (Pitching airfoil)

not change the flow solution, and the small curvature towards the TE does not need such a high resolution. Although in this case the bottleneck could be easily removed, on a generic complex geometry only further improvements in the `update_sdf` algorithm could ensure a lower imbalance in this kernel.

7.3. ROTATING FLAP

This setup includes a static section of a wing, and a flap-mounted ducted fan rotating around a hinge (Fig. 7.5). At the beginning the flap is at a 90° angle, corresponding to a hover position of a VTOL aircraft, and during the simulation the flap moves to horizontal position (cruise configuration).

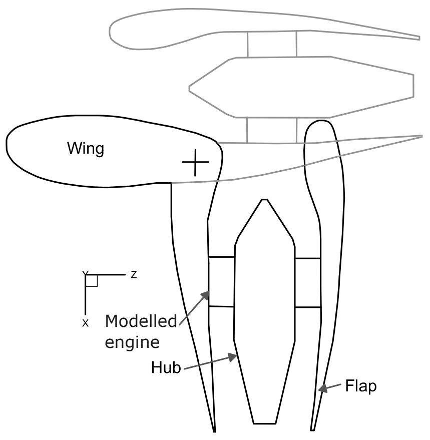


Figure 7.5: Rotating flap setup. Initial vertical position (–) and final horizontal position (–).

Although the setup shows similarities with the previous pitching airfoil setup, the smaller domain size, coarser mesh resolution (Table 7.2) and the presence of an engine volume shifts the computational costs towards different bottlenecks. The domain size is expressed in terms of flap width w.

The smaller number of cells lessens the significance of data migration cost between partitions at AMR steps. Although the AMR time still halved (Fig. 7.6), this has less impact on the total runtime (reduced only by 10%). The `update_flag` function become 1.7× slower than before. With this coarse mesh the voxelizer calls are shorter and most of the time is spent in `update_sdf`. The higher overall time is most likely due to (optional) synchronisation points between kernels in the new version, since both voxelizer (Fig. 5.5) and `update_sdf` (Fig. 6.4) are faster. The CUDA Nsight Systems output was not extracted for these simulation. In other setups the voxelizer and `update_sdf` imbalance cancels out, shortening the `update_flag` execution. The overhead is likely a combination of the synchronisation overhead and the extra filtering steps (see Appendix B).

Parameter	Value
GPUs	8 × Nvidia V100
Cells per block	16^3
Domain size	$[30, 1, 30] \cdot w$
Surface refinement level	3
Blocks	3476
Cells	14.4×10^6
Cut cells	0.38×10^6
Engine blocks	352

Table 7.2: Rotating flap settings and mesh properties

Fig. 7.8 reveals similar imbalance, but a lower runtime of `update_sdf` compared to the baseline version. ID 38–45 show the kernel calls with IB on the first lines, and ID 86–93 the same call with engine volume boundary type. The latter has a large theoretical speed-up, but this comes from the small kernel size, and the speed-up would be applicable only if the hardware would not be used for executing other kernels in parallel. The longer runtime of `update_sdf` is unexpected, and it was only observed in this small setup. The overhead of the block filtering can explain this (Appendix B).

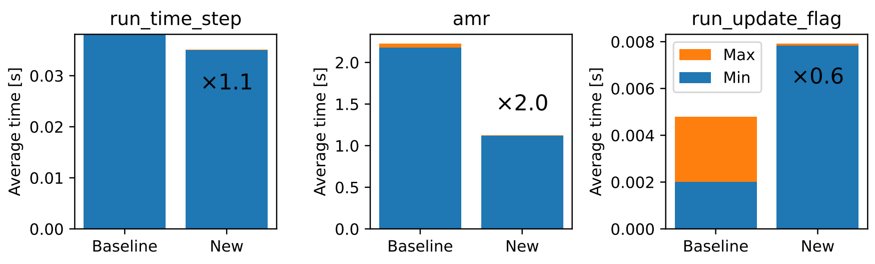
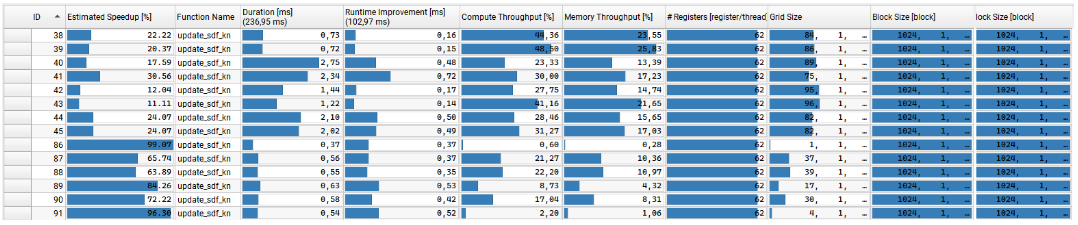


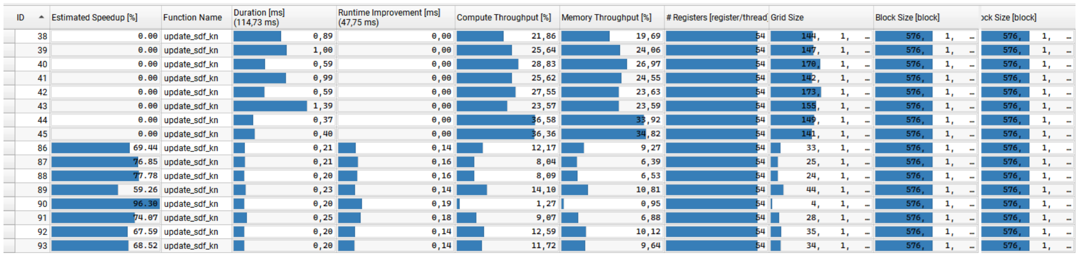
Figure 7.6: Timings (rotating flap)

Fig. 7.7–7.8 show the profiling of the `update_sdf` kernel on 8 GPUs. The first 8 lines show the kernel call with immersed boundary, and the subsequent lines with engine volumes. Two of the partitions in the baseline are not called, since they have no engine volumes assigned to them. In lack of an engine load balancing constraint the rest of the partitions have also a very uneven workload.



ID	Estimated Speedup [%]	Function Name	Duration [ms] (236,95 ms)	Runtime Improvement [ms] (102,97 ms)	Compute Throughput [%]	Memory Throughput [%]	# Registers [register/thread]	Grid Size	Block Size [block]	lock Size [block]
38	22.22	update_sdf_kn	0,73	0,16	44,36	23,55	62	84, 1, …	1024, 1, …	1024, 1, …
39	20.37	update_sdf_kn	0,72	0,15	48,50	25,83	62	86, 1, …	1024, 1, …	1024, 1, …
40	17.59	update_sdf_kn	2,75	0,48	23,33	13,39	62	89, 1, …	1024, 1, …	1024, 1, …
41	30.56	update_sdf_kn	2,34	0,72	30,00	17,23	62	75, 1, …	1024, 1, …	1024, 1, …
42	12.04	update_sdf_kn	1,44	0,17	27,75	14,74	62	95, 1, …	1024, 1, …	1024, 1, …
43	11.11	update_sdf_kn	1,22	0,14	41,16	21,65	62	96, 1, …	1024, 1, …	1024, 1, …
44	24.07	update_sdf_kn	2,10	0,50	28,46	15,65	62	82, 1, …	1024, 1, …	1024, 1, …
45	24.07	update_sdf_kn	2,02	0,49	31,27	17,03	62	82, 1, …	1024, 1, …	1024, 1, …
86	99.07	update_sdf_kn	0,37	0,37	0,60	0,28	62	1, 1, …	1024, 1, …	1024, 1, …
87	65.74	update_sdf_kn	0,56	0,37	21,27	10,36	62	37, 1, …	1024, 1, …	1024, 1, …
88	63.89	update_sdf_kn	0,55	0,35	22,20	10,97	62	39, 1, …	1024, 1, …	1024, 1, …
89	84.26	update_sdf_kn	0,63	0,53	8,73	4,32	62	17, 1, …	1024, 1, …	1024, 1, …
90	72.22	update_sdf_kn	0,58	0,42	17,04	8,31	62	30, 1, …	1024, 1, …	1024, 1, …
91	96.30	update_sdf_kn	0,54	0,52	2,20	1,06	62	4, 1, …	1024, 1, …	1024, 1, …

Figure 7.7: Profiling of the baseline version in CUDA Nsight Compute (Rotating flap)



ID	Estimated Speedup [%]	Function Name	Duration [ms] (114,73 ms)	Runtime Improvement [ms] (47,75 ms)	Compute Throughput [%]	Memory Throughput [%]	# Registers [register/thread]	Grid Size	Block Size [block]	ock Size [block]
38	0.00	update_sdf_kn	0,89	0,00	21,86	19,69	54	144, 1, …	576, 1, …	576, 1, …
39	0.00	update_sdf_kn	1,00	0,00	25,64	24,06	54	147, 1, …	576, 1, …	576, 1, …
40	0.00	update_sdf_kn	0,59	0,00	28,83	26,97	54	170, 1, …	576, 1, …	576, 1, …
41	0.00	update_sdf_kn	0,99	0,00	25,62	24,55	54	142, 1, …	576, 1, …	576, 1, …
42	0.00	update_sdf_kn	0,59	0,00	27,55	23,63	54	173, 1, …	576, 1, …	576, 1, …
43	0.00	update_sdf_kn	1,39	0,00	23,57	23,59	54	155, 1, …	576, 1, …	576, 1, …
44	0.00	update_sdf_kn	0,37	0,00	36,58	33,92	54	149, 1, …	576, 1, …	576, 1, …
45	0.00	update_sdf_kn	0,40	0,00	36,36	34,82	54	141, 1, …	576, 1, …	576, 1, …
86	69.44	update_sdf_kn	0,21	0,14	12,17	9,27	54	33, 1, …	576, 1, …	576, 1, …
87	76.85	update_sdf_kn	0,21	0,16	8,04	6,39	54	25, 1, …	576, 1, …	576, 1, …
88	77.78	update_sdf_kn	0,20	0,16	8,09	6,53	54	24, 1, …	576, 1, …	576, 1, …
89	59.26	update_sdf_kn	0,23	0,14	14,10	10,81	54	44, 1, …	576, 1, …	576, 1, …
90	96.30	update_sdf_kn	0,20	0,19	1,27	0,95	54	4, 1, …	576, 1, …	576, 1, …
91	74.07	update_sdf_kn	0,25	0,18	9,07	6,88	54	28, 1, …	576, 1, …	576, 1, …
92	67.59	update_sdf_kn	0,20	0,14	12,59	10,12	54	35, 1, …	576, 1, …	576, 1, …
93	68.52	update_sdf_kn	0,20	0,14	11,72	9,64	54	34, 1, …	576, 1, …	576, 1, …

Figure 7.8: Profiling of the new version in CUDA Nsight Compute (Rotating flap)

7.4. FLAP WITH RESOLVED ENGINE

The following example also presents a configuration with a wing section and a flap-mounted ducted fan (Fig. 7.9), but the flap angle is fixed, and the engine includes a resolved rotor and stator.

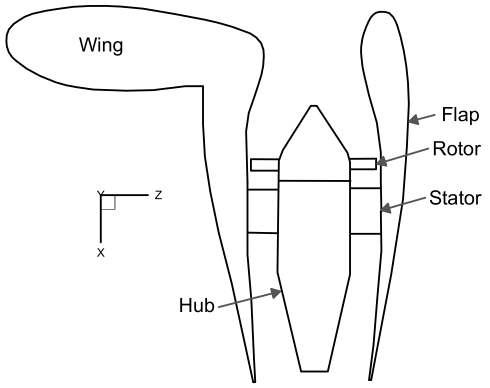


Figure 7.9: The setup of the flap with resolved engine example

The rotor is spinning with a constant RPM. The domain size is increased compared to the previous setup (Table 7.3). The wall cell size is reduced to resolve the boundary layer around the blades and to resolve the geometry at the trailing edge of the flap.

Parameter	Value
GPUs	8 × Nvidia V100
Cells per block	16^3
Domain size	$[100, 1, 100] \cdot w$
Surface refinement level	4-6
Blocks	20×10^3
Cells	81.6×10^6
Cut cells	1.2×10^6

Table 7.3: Flap with resolved engine settings and mesh properties

Due to the higher cell count the AMR data transfer is a bottleneck in the baseline code (Fig. 7.10). Since only a small portion of the domain near the rotor needs to be re-meshed, the scratch remapping algorithm of the baseline code moves unnecessarily big amounts of data between partitions. With URA load balancing the changes in partitioning (and therefore redistribution cost) are small (Fig. 4.16–4.17). The AMR computational cost became almost negligible, and together with a 30% improvement in `update_flag` a 4× total speed-up is achieved.

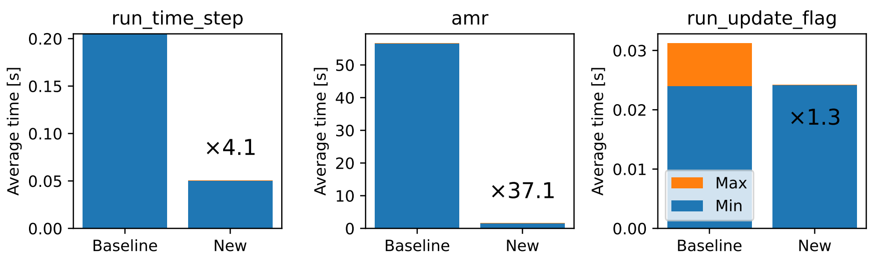


Figure 7.10: Timings (Flap with resolved engine)

The `update_flag` profiling of the baseline (Fig. 7.11) and new (Fig. 7.12) version both show an imbalance in `update_sdf`, although in the new version it is reduced. The voxelizer is balanced in both cases, therefore the extra synchronisation points of the new code do not add a considerable overhead in this setup.

The compute throughput of the voxelizer calls decreased (Fig. 7.13–7.14). The filtered block list causes thread divergence, but decreases the runtime.

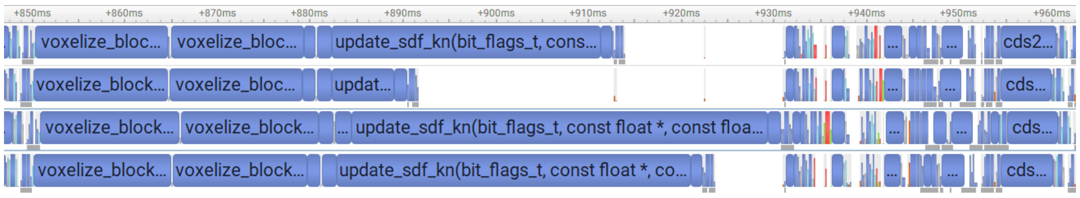


Figure 7.11: Profiling of the baseline version in CUDA Nsight Systems (Flap with resolved engine)

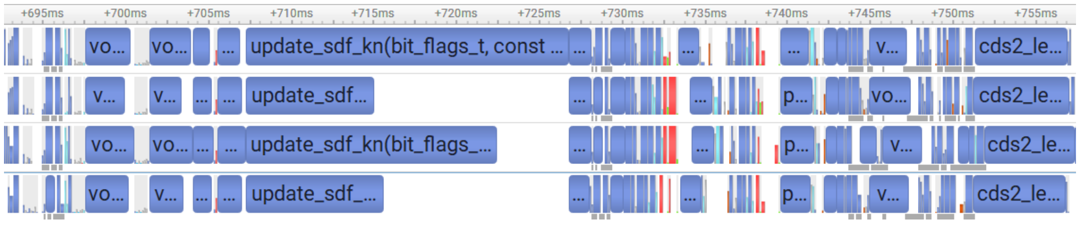
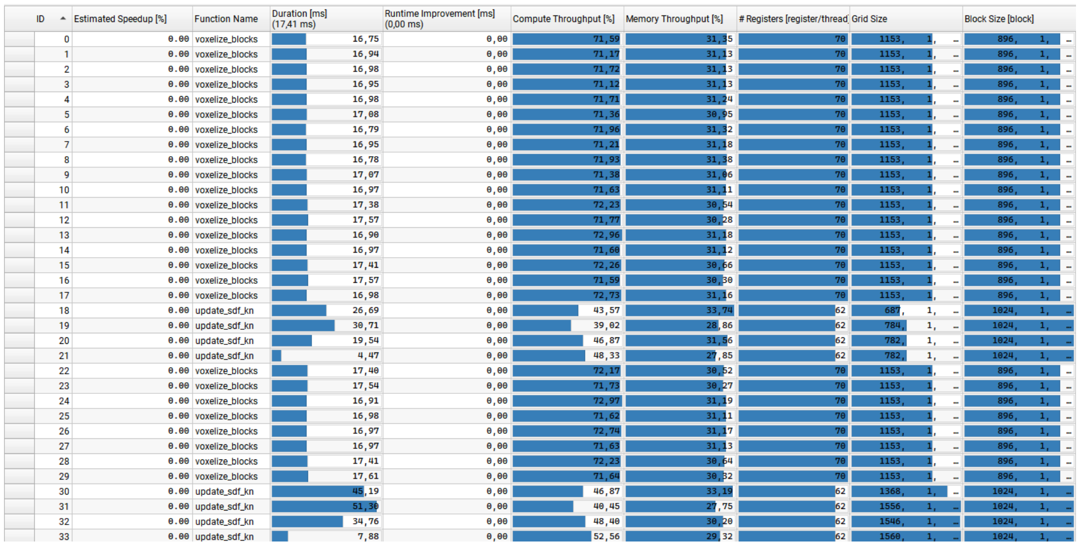
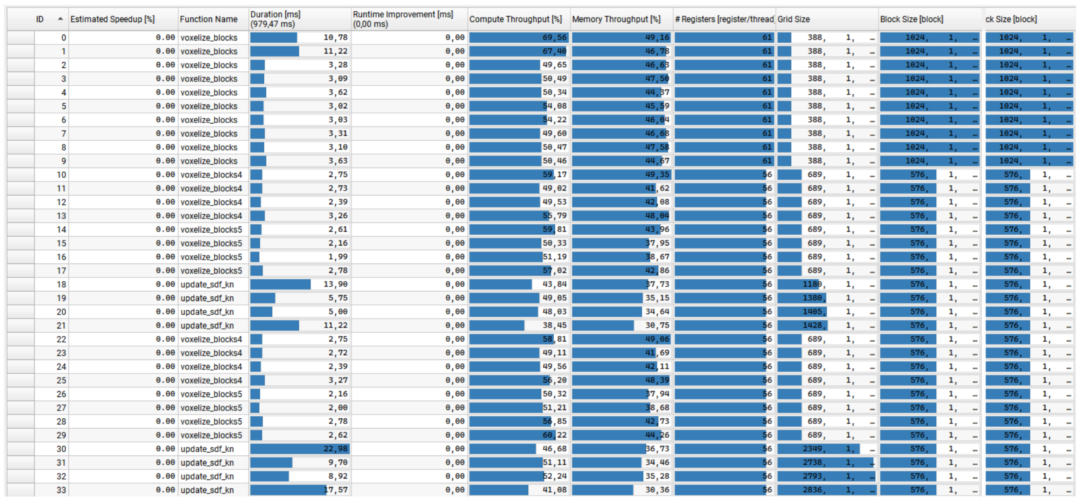


Figure 7.12: Profiling of the new version in CUDA Nsight Systems (Flap with resolved engine)



ID	Estimated Speedup [%]	Function Name	Duration [ms] (17,41 ms)	Runtime Improvement [ms] (0,00 ms)	Compute Throughput [%]	Memory Throughput [%]	# Registers [register/thread]	Grid Size	Block Size [block]
0	0.00	voxelize_blocks	16,75	0,00	71,59	31,35	70	1153, 1, …	896, 1, …
1	0.00	voxelize_blocks	16,94	0,00	71,17	31,13	70	1153, 1, …	896, 1, …
2	0.00	voxelize_blocks	16,98	0,00	71,72	31,13	70	1153, 1, …	896, 1, …
3	0.00	voxelize_blocks	16,95	0,00	71,12	31,13	70	1153, 1, …	896, 1, …
4	0.00	voxelize_blocks	16,98	0,00	71,71	31,24	70	1153, 1, …	896, 1, …
5	0.00	voxelize_blocks	17,08	0,00	71,36	30,95	70	1153, 1, …	896, 1, …
6	0.00	voxelize_blocks	16,79	0,00	71,96	31,32	70	1153, 1, …	896, 1, …
7	0.00	voxelize_blocks	16,95	0,00	71,21	31,18	70	1153, 1, …	896, 1, …
8	0.00	voxelize_blocks	16,78	0,00	71,93	31,38	70	1153, 1, …	896, 1, …
9	0.00	voxelize_blocks	17,07	0,00	71,38	31,06	70	1153, 1, …	896, 1, …
10	0.00	voxelize_blocks	16,97	0,00	71,63	31,11	70	1153, 1, …	896, 1, …
11	0.00	voxelize_blocks	17,38	0,00	72,23	30,54	70	1153, 1, …	896, 1, …
12	0.00	voxelize_blocks	17,57	0,00	71,77	30,28	70	1153, 1, …	896, 1, …
13	0.00	voxelize_blocks	16,90	0,00	72,96	31,18	70	1153, 1, …	896, 1, …
14	0.00	voxelize_blocks	16,97	0,00	71,60	31,12	70	1153, 1, …	896, 1, …
15	0.00	voxelize_blocks	17,41	0,00	72,26	30,66	70	1153, 1, …	896, 1, …
16	0.00	voxelize_blocks	17,57	0,00	71,59	30,30	70	1153, 1, …	896, 1, …
17	0.00	voxelize_blocks	16,98	0,00	72,73	31,16	70	1153, 1, …	896, 1, …
18	0.00	update_sdf_kn	26,69	0,00	43,57	33,74	62	687, 1, …	1024, 1, …
19	0.00	update_sdf_kn	30,71	0,00	39,02	28,86	62	784, 1, …	1024, 1, …
20	0.00	update_sdf_kn	19,54	0,00	46,87	31,56	62	782, 1, …	1024, 1, …
21	0.00	update_sdf_kn	4,47	0,00	48,33	27,85	62	782, 1, …	1024, 1, …
22	0.00	voxelize_blocks	17,40	0,00	72,17	30,52	70	1153, 1, …	896, 1, …
23	0.00	voxelize_blocks	17,54	0,00	71,73	30,27	70	1153, 1, …	896, 1, …
24	0.00	voxelize_blocks	16,91	0,00	72,97	31,19	70	1153, 1, …	896, 1, …
25	0.00	voxelize_blocks	16,98	0,00	71,62	31,11	70	1153, 1, …	896, 1, …
26	0.00	voxelize_blocks	16,97	0,00	72,74	31,17	70	1153, 1, …	896, 1, …
27	0.00	voxelize_blocks	16,97	0,00	71,63	31,13	70	1153, 1, …	896, 1, …
28	0.00	voxelize_blocks	17,41	0,00	72,23	30,64	70	1153, 1, …	896, 1, …
29	0.00	voxelize_blocks	17,61	0,00	71,64	30,32	70	1153, 1, …	896, 1, …
30	0.00	update_sdf_kn	45,19	0,00	46,87	33,19	62	1368, 1, …	1024, 1, …
31	0.00	update_sdf_kn	51,30	0,00	40,45	27,75	62	1556, 1, …	1024, 1, …
32	0.00	update_sdf_kn	34,76	0,00	48,40	30,20	62	1546, 1, …	1024, 1, …
33	0.00	update_sdf_kn	7,88	0,00	52,56	29,32	62	1560, 1, …	1024, 1, …

Figure 7.13: Profiling of the baseline version in CUDA Nsight Compute (Flap with resolved engine)



ID	Estimated Speedup [%]	Function Name	Duration [ms] (979,47 ms)	Runtime Improvement [ms] (0,00 ms)	Compute Throughput [%]	Memory Throughput [%]	# Registers [register/thread]	Grid Size	Block Size [block]	ck Size [block]
0	0.00	voxelize_blocks	10,78	0,00	69,56	49,16	61	388, 1, …	1024, 1, …	1024, 1, …
1	0.00	voxelize_blocks	11,22	0,00	67,40	46,78	61	388, 1, …	1024, 1, …	1024, 1, …
2	0.00	voxelize_blocks	3,28	0,00	49,65	46,63	61	388, 1, …	1024, 1, …	1024, 1, …
3	0.00	voxelize_blocks	3,09	0,00	50,49	47,50	61	388, 1, …	1024, 1, …	1024, 1, …
4	0.00	voxelize_blocks	3,62	0,00	50,34	44,37	61	388, 1, …	1024, 1, …	1024, 1, …
5	0.00	voxelize_blocks	3,02	0,00	54,08	45,59	61	388, 1, …	1024, 1, …	1024, 1, …
6	0.00	voxelize_blocks	3,03	0,00	54,22	46,04	61	388, 1, …	1024, 1, …	1024, 1, …
7	0.00	voxelize_blocks	3,31	0,00	49,60	46,68	61	388, 1, …	1024, 1, …	1024, 1, …
8	0.00	voxelize_blocks	3,10	0,00	50,47	47,58	61	388, 1, …	1024, 1, …	1024, 1, …
9	0.00	voxelize_blocks	3,63	0,00	50,46	44,67	61	388, 1, …	1024, 1, …	1024, 1, …
10	0.00	voxelize_blocks4	2,75	0,00	59,17	49,35	56	689, 1, …	576, 1, …	576, 1, …
11	0.00	voxelize_blocks4	2,73	0,00	49,02	41,62	56	689, 1, …	576, 1, …	576, 1, …
12	0.00	voxelize_blocks4	2,39	0,00	49,53	42,08	56	689, 1, …	576, 1, …	576, 1, …
13	0.00	voxelize_blocks4	3,26	0,00	55,79	48,04	56	689, 1, …	576, 1, …	576, 1, …
14	0.00	voxelize_blocks5	2,61	0,00	59,81	43,96	56	689, 1, …	576, 1, …	576, 1, …
15	0.00	voxelize_blocks5	2,16	0,00	50,33	37,95	56	689, 1, …	576, 1, …	576, 1, …
16	0.00	voxelize_blocks5	1,99	0,00	51,19	38,67	56	689, 1, …	576, 1, …	576, 1, …
17	0.00	voxelize_blocks5	2,78	0,00	57,02	42,86	56	689, 1, …	576, 1, …	576, 1, …
18	0.00	update_sdf_kn	13,90	0,00	43,84	37,73	56	1180, 1, …	576, 1, …	576, 1, …
19	0.00	update_sdf_kn	5,75	0,00	49,05	35,15	56	1380, 1, …	576, 1, …	576, 1, …
20	0.00	update_sdf_kn	5,00	0,00	48,03	34,64	56	1405, 1, …	576, 1, …	576, 1, …
21	0.00	update_sdf_kn	11,22	0,00	38,45	30,75	56	1428, 1, …	576, 1, …	576, 1, …
22	0.00	voxelize_blocks4	2,75	0,00	58,81	49,06	56	689, 1, …	576, 1, …	576, 1, …
23	0.00	voxelize_blocks4	2,72	0,00	49,11	41,69	56	689, 1, …	576, 1, …	576, 1, …
24	0.00	voxelize_blocks4	2,39	0,00	49,56	42,11	56	689, 1, …	576, 1, …	576, 1, …
25	0.00	voxelize_blocks4	3,27	0,00	56,20	48,39	56	689, 1, …	576, 1, …	576, 1, …
26	0.00	voxelize_blocks5	2,16	0,00	50,32	37,94	56	689, 1, …	576, 1, …	576, 1, …
27	0.00	voxelize_blocks5	2,00	0,00	51,21	38,68	56	689, 1, …	576, 1, …	576, 1, …
28	0.00	voxelize_blocks5	2,78	0,00	56,85	42,73	56	689, 1, …	576, 1, …	576, 1, …
29	0.00	voxelize_blocks5	2,62	0,00	60,22	44,26	56	689, 1, …	576, 1, …	576, 1, …
30	0.00	update_sdf_kn	22,98	0,00	46,68	36,73	56	2349, 1, …	576, 1, …	576, 1, …
31	0.00	update_sdf_kn	9,70	0,00	51,11	34,46	56	2738, 1, …	576, 1, …	576, 1, …
32	0.00	update_sdf_kn	8,92	0,00	52,24	35,28	56	2793, 1, …	576, 1, …	576, 1, …
33	0.00	update_sdf_kn	17,57	0,00	41,08	30,36	56	2836, 1, …	576, 1, …	576, 1, …

Figure 7.14: Profiling of the new version in CUDA Nsight Compute (Flap with resolved engine)

7.5. AIRCRAFT WIND TUNNEL MODEL

The final – and most complex – setup is the wind tunnel model of the Lilium Jet [1], an eVTOL aircraft with distributed electric propulsion. The aircraft has 30 engines on the wings and canards, from which the left wing engines are resolved, and the rest of them are modelled (Fig. 7.15). The flaps and lifting surfaces around the modelled engines, as well as the fuselage, have a much coarser surface mesh resolution than the left wing. Such a mixed-fidelity setup poses a challenge both for the load balancing and the immersed boundary algorithms. The simulation results of this setup will be addressed in Chapter 8, and in the following the performance of the baseline and new code is compared.

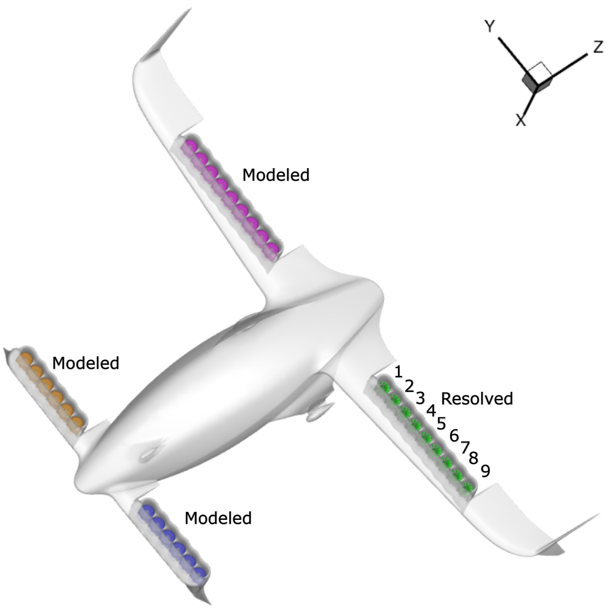


Figure 7.15: Aircraft wind tunnel model geometry

The mesh settings are described in Table 7.4, where the domain size is expressed in terms of aircraft wingspan b. The mesh refinements are detailed in Section 7.6, where the present setup corresponds to the settings of the *baseline* mesh.

Parameter	Value
GPUs	8 × Nvidia A100
Cells per block	16^3
Domain size	$[1,1,1]\cdot 12.9\cdot b$
Surface refinement level	10-14
Blocks	77×10^3
Cells	315.5×10^6
Cut cells	10.3×10^6
Engine blocks	292

Table 7.4: Aircraft wind tunnel model settings and mesh properties

The simulation is 5 times faster with the optimised algorithms (Fig. 7.16), and the AMR has an almost 100× speed-up. With the baseline a more competitive option could be to use a finer uniform mesh resolution around the rotors and turn off the AMR, but this was not tested. The `update_flag` function is 3 times faster, both due to the voxelizer and `update_sdf` improvements.

Snapshots of one iteration profiled with CUDA Nsight Systems are shown in Fig. 7.17–7.18. In the baseline version the voxelizer calls are almost perfectly balanced, since every thread in the kernel is looping through the whole local block list. The SDF update has a large imbalance, driven by locally fine portions of the surface mesh,resulting in different triangle list lengths. Contrary to the baseline version, the new version includes additional GPU synchronisation points, which are not necessary for the correct execution of the `update_flag` algorithm, and could be removed. In this particular iteration this would result in an estimated 10% speed-up

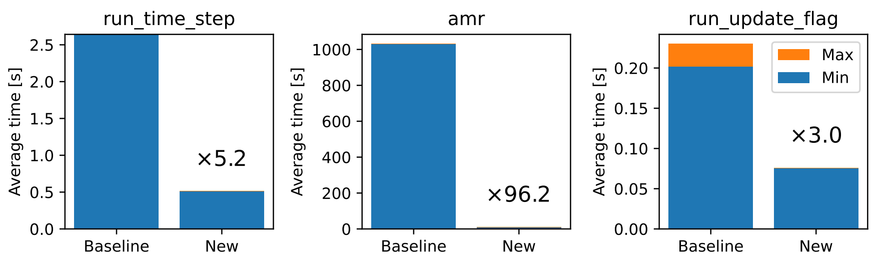


Figure 7.16: Timings of the aircraft wind tunnel model setup

The ratio of the time spent waiting for the synchronisation and the time spent in the kernels is approximately $\sum t_{wait} / \sum t_{kernel} = 0.1$ on Process 0, which has the largest $\sum t_{kernel}$.

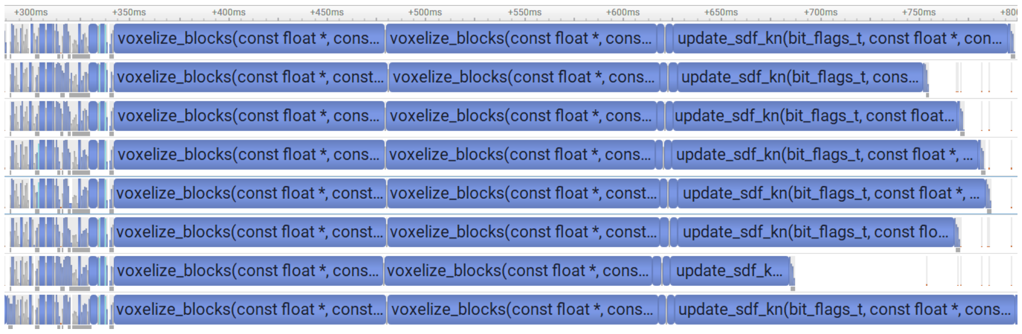

voxelize_blocks(const float *, cons...
update_sdf_kn(bit_flags_t, const float *, con...

Figure 7.17: Profiling the baseline code in CUDA Nsight Systems (Aircraft wind tunnel model)

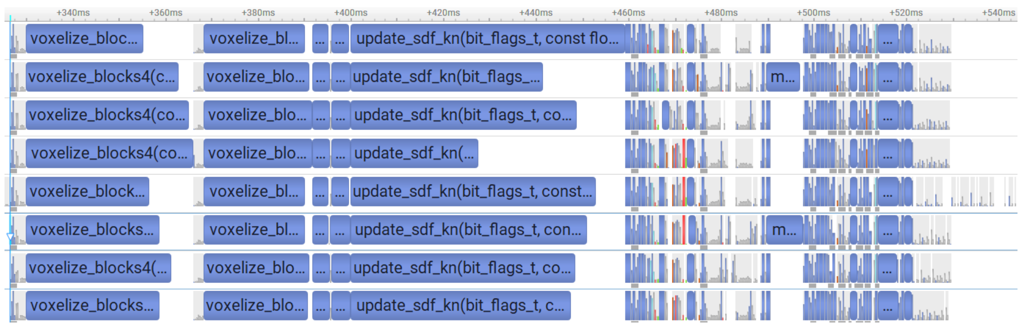
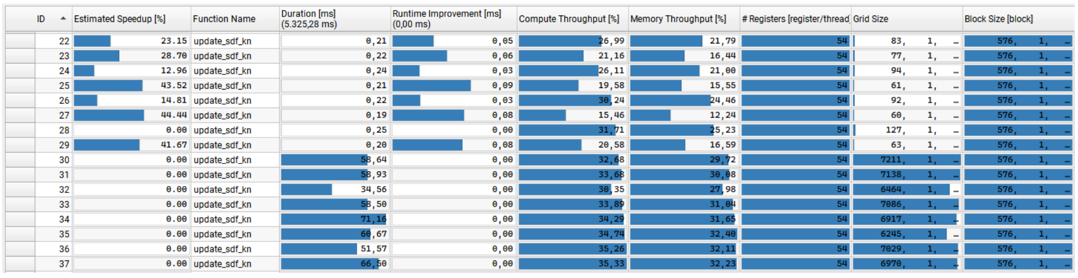


Figure 7.18: Profiling the optimised code in CUDA Nsight Systems (Aircraft wind tunnel model)

Fig. 7.19 shows the profiling of the optimised `update_sdf` kernel. The profiler finds further speed-up opportunities only for the kernel calls of the engine boundaries. These are already much quicker than the call for IB, and it has a small potential to decrease the runtime.



ID	Estimated Speedup [%]	Function Name	Duration [ms] (5.325,28 ms)	Runtime Improvement [ms] (0,00 ms)	Compute Throughput [%]	Memory Throughput [%]	# Registers [register/thread]	Grid Size	Block Size [block]
22	23.15	update_sdf_kn	0,21	0,05	26,99	21,79	54	83, 1, …	576, 1, …
23	28.70	update_sdf_kn	0,22	0,06	21,16	16,44	54	77, 1, …	576, 1, …
24	12.96	update_sdf_kn	0,24	0,03	26,11	21,00	54	94, 1, …	576, 1, …
25	43.52	update_sdf_kn	0,21	0,09	19,58	15,55	54	61, 1, …	576, 1, …
26	14.81	update_sdf_kn	0,22	0,03	30,24	24,46	54	92, 1, …	576, 1, …
27	44.44	update_sdf_kn	0,19	0,08	15,46	12,24	54	60, 1, …	576, 1, …
28	0.00	update_sdf_kn	0,25	0,00	31,71	25,23	54	127, 1, …	576, 1, …
29	41.67	update_sdf_kn	0,20	0,08	20,58	16,59	54	63, 1, …	576, 1, …
30	0.00	update_sdf_kn	58,64	0,00	32,68	29,72	54	7211, 1, …	576, 1, …
31	0.00	update_sdf_kn	58,93	0,00	33,68	30,08	54	7138, 1, …	576, 1, …
32	0.00	update_sdf_kn	34,56	0,00	30,35	27,98	54	6464, 1, …	576, 1, …
33	0.00	update_sdf_kn	58,50	0,00	33,89	31,04	54	7086, 1, …	576, 1, …
34	0.00	update_sdf_kn	71,16	0,00	34,29	31,65	54	6917, 1, …	576, 1, …
35	0.00	update_sdf_kn	60,67	0,00	34,74	32,40	54	6245, 1, …	576, 1, …
36	0.00	update_sdf_kn	51,57	0,00	35,26	32,11	54	7029, 1, …	576, 1, …
37	0.00	update_sdf_kn	66,50	0,00	35,33	32,23	54	6970, 1, …	576, 1, …

Figure 7.19: Profiling the optimised code in CUDA Nsight Compute (Aircraft wind tunnel model). Immersed boundary (first 8 lines) and engine (last 8 lines) SDF update.

7.6. AIRCRAFT MESH CONVERGENCE STUDY

For the mesh convergence study a simplified version of the aircraft setup in Section 7.5 is used, with only one resolved engine on the left wing. Three different meshes are compared, where the *baseline* has similar mesh settings as the 9 engine setup in Section 7.5. The wall cell size of the resolved engine and the flaps on the left wing is varied, whereas the mesh of modelled engines, the fuselage and other flaps and lifting surfaces are kept the same. Table 7.5 summarises the mesh settings.

Mesh	Part	Min cell size [$\times 10^{-3}$]	Number of blocks	Number of cells [$\times 10^{6}$]
Coarse	Rotor	0.8	46474	≈190
	Vanes	0.8		
	Hub	0.8		
	Flaps	0.8		
	Inlet	0.8		
Baseline	Rotor	0.4	51577	≈211
	Vanes	0.4		
	Hub	0.4		
	Flaps	0.8		
	Inlet	0.4		
Fine	Rotor	0.4	117888	≈482
	Vanes	0.2		
	Hub	0.4		
	Flaps	0.4		
	Inlet	0.2		

Table 7.5: Mesh settings for the aircraft wind tunnel model setup

Fig. 7.20 shows the scaling with number of cells. With a 2.5 times bigger mesh the runtime increased by only 30%. The `update_sdf` function is faster on the *baseline* and *fine* mesh due to the shorter block lists (Chapter 6).

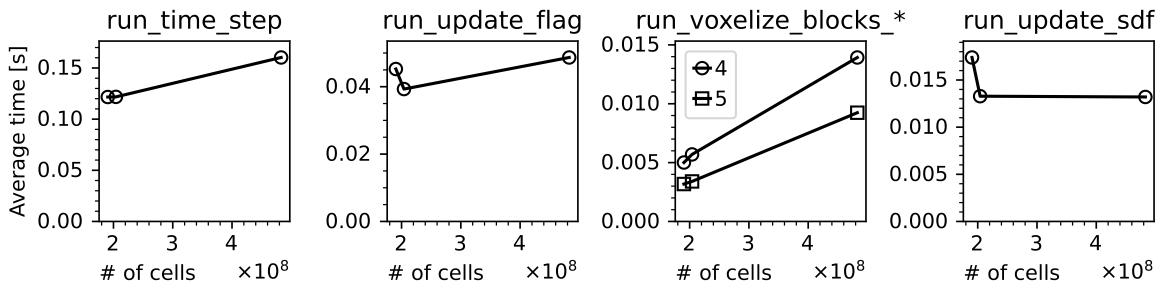


Figure 7.20: Mesh convergence study timings

The above mesh convergence study was repeated with short, 1000 iteration runs (Fig. 7.21). The AMR times are not fully representative, since only 1 or 2 AMR steps occur in all of these simulations. In these initial AMR steps the optimised code is 3 to 6 times faster. It was not feasible to run the *fine* mesh with the baseline code, as the first AMR step did not finish after six hours of waiting.

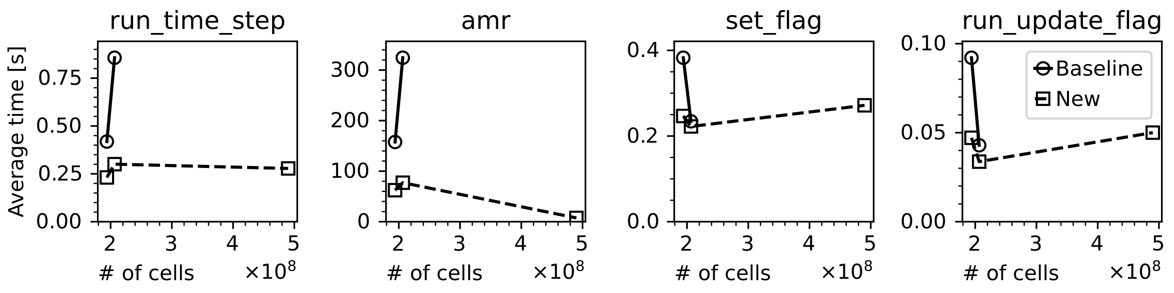


Figure 7.21: Mesh convergence study timings. Comparison of baseline and new.

Table 7.6 shows the partitioning quality statistics from ParMETIS. It is notable, that the edge-cut of the largest mesh improved more than 10% during the initial refinement. On the fine mesh all constraints are (almost) satisfied, whereas the two other meshes violate the 1.01 target.

	Edge-cut		Imbalance		
Mesh	Initial	Final	Blocks	Cut cells	Engine blocks
Coarse	3182778	3008997 (-5.5%)	1.022	1.016	1.029
Baseline	3349306	3281715 (-2%)	1.010	1.015	1.029
Fine	5375575	4823815 (-10.2%)	1.010	1.010	1.011

Table 7.6: Partitioning quality (Aircraft wind tunnel model)

7.7. SCALING WITH RESOLVED ENGINES

Two simulation results are compared. One only contains a single resolved engine, the other has all engines resolved on the left wing. The mesh properties are summarised in Table 7.7. All other mesh settings, e.g. wall cell sizes, are the same for the same type of body.

Engines	# of blocks	# of engine blocks	# of cuts	# of cells	# of cells with buffer
1	49824	2884	5×10^6	204×10^6	291×10^6
9	77190	292	10.2×10^6	316×10^6	450×10^6
Ratio 9 to 1	1.549	0.101	2.041	1.549	1.549

Table 7.7: Mesh properties with 1 and 9 resolved engines

These two setups give insight into how the runtime changes as a larger portion of the propulsion system is modelled with higher fidelity. A selection of timers can be found in Table 7.8. The IB update in both cases takes up about three quarters of the runtime, but the composition of it changes. With the larger setup the voxelizer slowed down due to the increased block list loop. The bounding box of the aircraft body stays the same, but the flaps on the left wing are more refined, therefore the amount of cut blocks increases. The graph partitioning functions `ParMETIS_initial` (called once) `ParMETIS_adaptive` (called at each AMR step) scaled linearly with the mesh size.

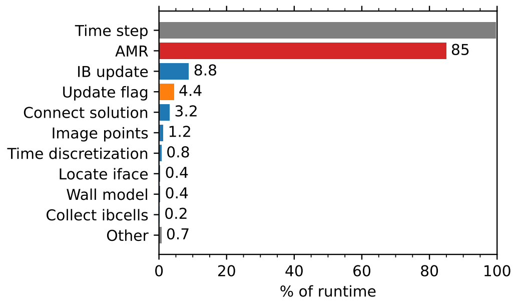
	Average time [s]			Fraction of run [%]	
Timer	1	9	Ratio of 9 to 1	1	9
init_metis_partition	0.706604	1.10766	1.57		
init	10.6528	21.3566	2.00		
run_time_step	0.121768	0.220158	1.81		
run_ib_update	0.0942415	0.161963	1.72	77.36	73.61
run_update_flag	0.0393023	0.0714342	1.82	64.77	65.16
run_update_sdf	0.0132821	0.0302929	2.28	21.89	27.63
run_mark_sdf_cells	0.0121663	0.00454125	0.37	20.05	4.14
run_AMR_step_total	3.15462	7.24407	2.30	9.54	11.72
run_voxelize_blocks_4	0.0056954	0.0184762	3.24	9.39	16.85
run_voxelize_blocks_5	0.00339964	0.0117987	3.47	5.60	10.76
run_update_geom	0.00132233	0.0101732	7.69	1.09	4.62
ParMETIS_adaptive	0.583997	0.928009	1.59		
superstl_bbox	0.000596595	0.00700028	11.73		
superstl_rot_matrix	0.000315042	0.00167081	5.30		
superstl_normals	0.00011488	0.000510219	4.44		
set_flag	0.0157046	0.0169177	1.08		
ParMETIS_initial	0.890523	1.3561	1.52		

Table 7.8: Timings with 1 and 9 resolved engines

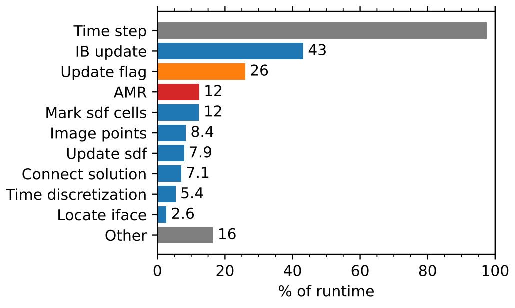
7.8. BOTTLENECKS

The most expensive parts of the code execution in baseline and optimised version are outlined in Fig. 7.22–7.25. The optimised parts of the code (AMR and `update_flag`) are highlighted in red and orange respectively.

AMR was dominant in all test cases except in the rotating flap setup. After the optimisation only takes on the order of 5% of the runtime, except the pitching airfoil setup. The higher AMR fraction in this setup is presumably due to a large number of blocks and a relatively small number of cut cells. This forces more block redistribution to keep the cut cell balance as the airfoil is rotating.

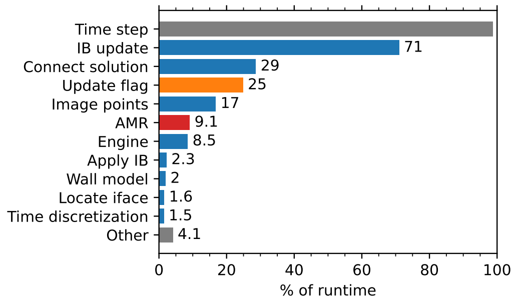


(a) Baseline code

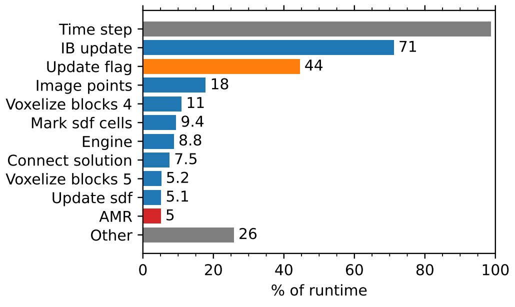


(b) Optimised code

Figure 7.22: Comparison of bottlenecks in the baseline and optimised code (Pitching airfoil)

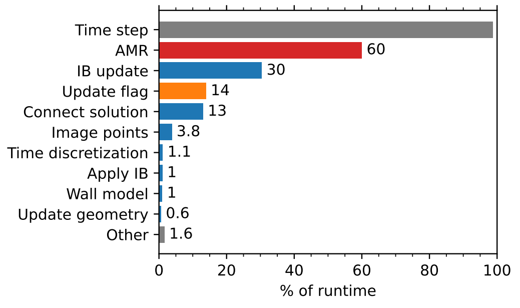


(a) Baseline code

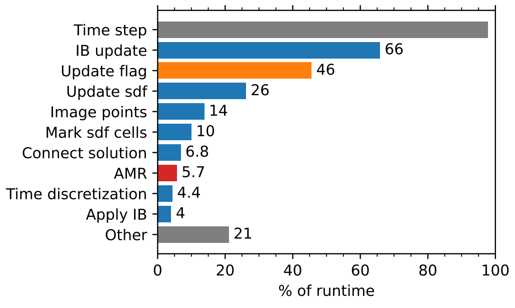


(b) Optimised code

Figure 7.23: Comparison of bottlenecks in the baseline and optimised code (Rotating flap)

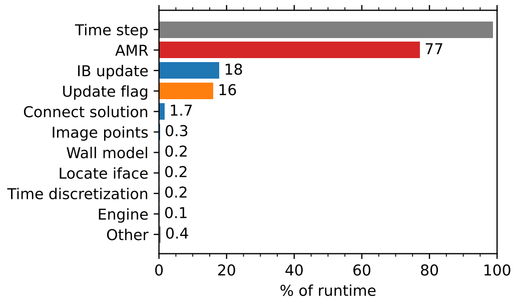


(a) Baseline code

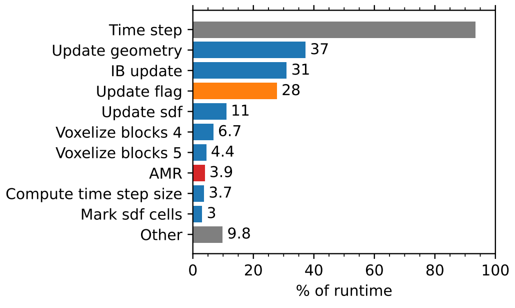


(b) Optimised code

Figure 7.24: Comparison of bottlenecks in the baseline and optimised code (Flap with resolved engine)



(a) Baseline code



(b) Optimised code

Figure 7.25: Comparison of bottlenecks in the baseline and optimised code (Aircraft wind tunnel model)

The IB update (including the voxelizer calls, `update_sdf` and all other steps from Fig. 5.1) became the most time consuming part of the code, taking between 37% to 71% of the runtime. Note, that the absolute time of this part decreased for all test cases except the rotating flap, but by the removal of the AMR bottleneck it is a larger fraction of the runtime in most setups. The `update_flag` function take between a quarter to a half of the runtime. It varies case by cases, if `update_sdf` or the voxelizer is more time-consuming, but `update_sdf` has more potential for further improvements. The `mark_sdf_cells` kernel, locating image points and connecting the solution also became significant contributors to the runtime, but no optimisation opportunities were identified for these.

7.9. Summary

The runtime of all of the four setups was successfully reduced (Table 7.9). The large setups with resolved engines are 4 to 5 times faster than with the baseline code. The largest improvements are coming from load balancing, which in some cases became two orders of magnitudes faster and it is not a bottleneck any more. The voxelizer and `update_sdf` algorithm changes jointly resulted in an up to threefold speed-up of the flag field update, with a larger increase in the most complex setup.

Speed-up	Pitching airfoil	Rotating flap	Flap with resolved engine	Aircraft wind tunnel model
Run	7.2	1.1	4.1	5.2
AMR	48	2	37	96
Update flag	1.4	0.6	1.3	3

Table 7.9: Speed-up of the test cases

One of the setups has an increased `update_flag` time, but the absolute time penalty is small, and this setup still became faster thanks to the AMR improvements. Removing synchronisation points is expected to lessen or eliminate this penalty. Another remedy could be to switch back to the full block list (as in the baseline) on coarse meshes, and only use the new algorithm, when the filtering pays off.

8 DISTRIBUTED PROPULSION ARRAY IN TAILWIND

8.1. INTRODUCTION

Novel aircraft configurations, such as eVTOLs, distributed electric propulsion systems and open rotor turbofans pose many new aerodynamic challenges. They are characterised by a closer coupling of external flow and engine aerodynamics. For eVTOLs, the very distinct cruise and hover operating points and the transition between them often include unsteady flow regimes and separated flow. The accurate prediction of the flowfield and engine performance is crucial for the sizing of the propulsion system, for compliance with regulatory requirements and to ensure a safe operation in the whole flight envelope.

In this chapter an eVTOL aircraft is analysed in tailwind conditions. This situation can be encountered during take-off or landing, either caused by the aircraft manoeuvring (ground speed) or by natural wind. Similar to crosswind and high angle of attack inflow of gas turbine engines [72–74], tailwind on the DEP array of a hovering VTOL aircraft can cause high inlet distortion or inlet separation, which can severely degrade the engine performance. On a DEP array the streamlines are more constrained (Fig. 8.1) than in a stand-alone ducted fan case. The flowfield changes with the geometry (especially the length of the array), the engine RPM and the wind speed.

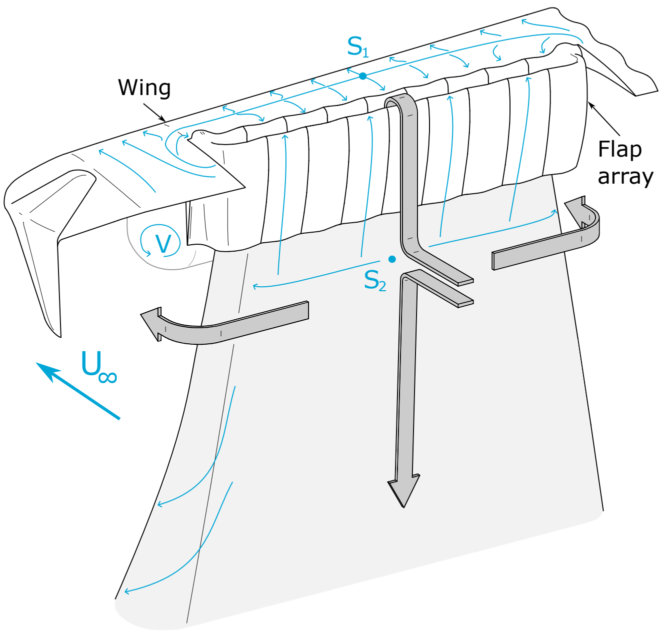
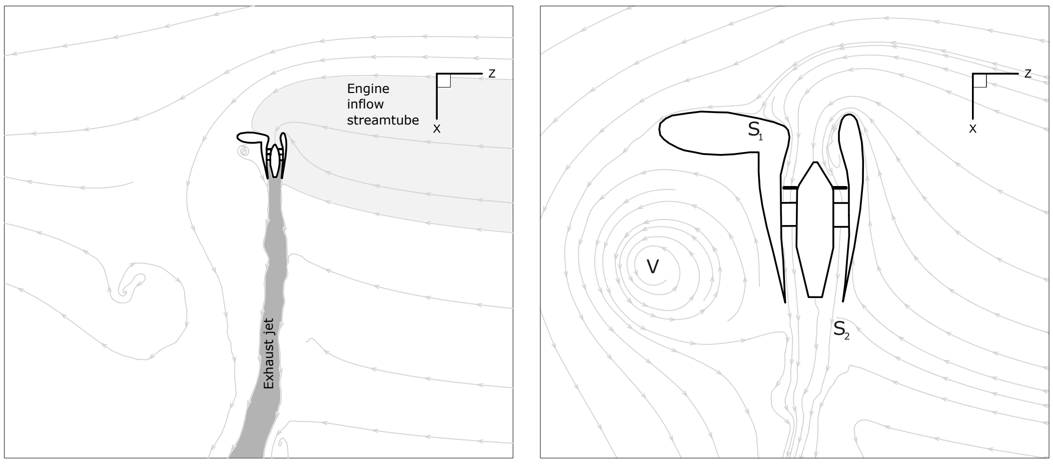


Figure 8.1: Flow topology around a wing-mounted distributed propulsion array in 3D

Fig. 8.2 shows the engine streamtube entering the inlet. In the middle of the array the flow is almost two-dimensional. The flow has a stagnation line S_2 behind the flap. The streamlines entering the engine first turn upwards, before entering the inlet through a U-turn around the leading edge of the flap. There is another stagnation line S_1 on the upper wing surface, where the suction of the engine drives the flow opposite to the freestream velocity vector. A boundary layer develops both on the wing and the flap, before the flow enters the inlet. Below the wing and in front of the flap a stable vortex V develops, and some of the flow going around the wing is circling around this vortex and is finally entrained by the exhaust jet.



(a) Engine inflow and outflow streamtubes (b) Flow topology in the symmetry plane of the propulsion array

Figure 8.2: Flow topology in tailwind

Due to the boundary layer development on the flap top and the wing, its thickness varies along the circumference downstream of the rotor (Fig. 8.3). The inlet has a roughly square shape in the plane of the flap leading edge, which transitions to a circular duct at the rotor. This causes secondary flows in the rotor plane. A stronger tailwind shifts the flap top stagnation line towards (or even beyond) the flap trailing edge, and the large streamline curvature of the high speed flow around the flap LE causes a strong suction peak. This can lead to a separation near the LE. The inflow distortion decreases the stall margin of the fan, and it can cause detachments on the stator vanes, on the hub, or on the duct. Fig. 8.4 shows the non-axisymmetric outlet section of the engine. The main flow features are the wake of the hub and the stator vanes, and in off-design conditions the corners of the duct can have detachment. This is especially likely, if the engine operates far from its design point, and the swirl is not completely removed by the stator vanes.

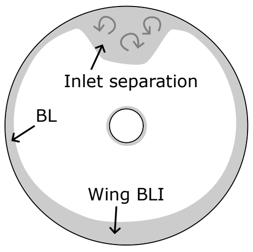


Figure 8.3: Engine intake flow features

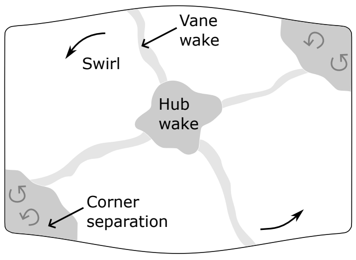


Figure 8.4: Engine exhaust flow features

The rest of this chapter often refers to the stages of the thermodynamic cycle of the engine. These are used as post-processing planes, and they are defined in Fig. 8.5. The measurement planes of the experimental setup are also shown. Note, that the inlet rake (IR) does not coincide with Stage 2, whereas Stage 6 and the outlet rake (OR) are both located at the trailing edge of the flap. Stage 3 and 5 did not have any instrumentation.

In the following aircraft performance parameters are defined. A mesh convergence study of a simplified setup with a single fully resolved engine is presented, which was used to select the appropriate mesh settings for the simulations with 9 resolved engines. The CFD results of a low and high RPM tailwind condition are analysed, and finally they are compared with corresponding wind tunnel data.

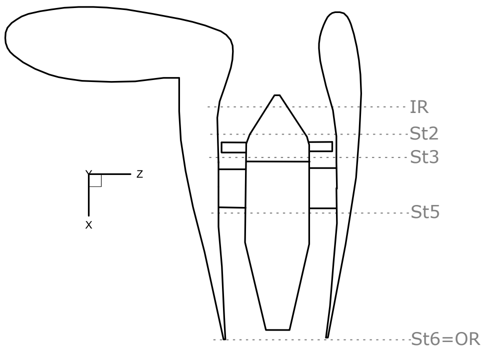


Figure 8.5: Engine stage definition and rake positions

8.2. DEFINITION OF ENGINE PERFORMANCE PARAMETERS

The emphasis is mainly on physical quantities, which can be also derived from the available measurement data. Aircraft level aerodynamic quantities, such as lift and drag coefficient are not investigated due to their small magnitude compared to thrust in the investigated tailwind conditions.

The mass flow rate is not recorded in the output data. The output data has high temporal resolution, whereas the stage surface fields were saved at a lower sampling rate. The mass flow rate is approximated from the data with the area weighted averages of the primary solution variables (Eq. 8.1).

$$\dot{m}(t) = \int_A \rho(t)u(t)dA \cong \left(\frac{1}{A}\int_A \rho(t)dA\right)\cdot\left(\frac{1}{A}\int_A u(t)dA\right)\cdot A = \overline{\rho}(t)\overline{u}(t)A \tag{8.1}$$

The density field is almost uniform, except in the wake regions, which contribute little to the mass flow due to the small velocity. The error from this approximation compared to the exact integral was 0.14%. This is deemed acceptable, and this definition of approximate mass flow rate will be used throughout this chapter.

The thrust is calculated from the freestream and outlet properties. More elaborate thrust bookkeeping systems [75, 76] exist for integrated propulsion systems, but the definition in Eq. 8.2 is more suitable to later allow a more straightforward comparison with experimental data. The engine aerodynamic shaft torque τ is simply the moment around the engine axis on the rotor (both blades and spinner).

$$T = (V_6 - V_\infty)\dot{m} + (p_6 - p_\infty)A_6 \tag{8.2}$$

The non-dimensional mass-flow is defined as the ratio of the engine mass flow and the mass flow of a streamtube in the freestream with a cross-section equal to the fan face area. The thrust (Eq. 8.4) and torque (Eq. 8.5) coefficients are normalised with the shroud diameter d_2 at Stage 2.

$$c_\mu = \frac{\dot{m}}{\rho_\infty U_\infty A_2} \tag{8.3}$$

$$c_T = \frac{T}{\rho_\infty n^2 d_2^4} \tag{8.4}$$

$$c_\tau = \frac{\tau}{\rho_\infty n^2 d_2^5} \tag{8.5}$$

In the following, the data from the last engine rotation is used for averaging.

8.3. MESH CONVERGENCE STUDY

The three meshes used in the aircraft mesh convergence study were already introduced in Section 7.6. Their mesh settings and results are summarised in Table 8.1, where t is the runtime and the coefficients are defined in Eq. 8.3–8.5.

All meshes contain one fully resolved engine in the middle of the left wing. The study focused on the sensitivity of engine flow features and performance on the mesh resolution. The mesh outside the left wing array was kept unchanged, and it is similar to the 9 engine setup. Fig. 8.6 highlights the mesh resolution changes in the symmetry plane of the resolved engine.

The modelled engines of the array were only refined on the *fine* mesh. This reduces the y^+ below 300 on the modelled engines, although this is not critical, since the Proportional-Integral (PI) controller keeps the mass flow close to the target on all meshes. Therefore, the effect of this on the resolved engine is negligible. The *fine* mesh also targets to reduce the $y+$ on the flap LE, which is a high velocity region, and the boundary layer development in this area can have a large impact on the separation point.

Mesh	Part	Min cell size [$\times 10^{-3}$]	Number of blocks	Number of cells [$\times 10^{6}$]	c_μ	c_T	c_τ	t [h]
Coarse	Rotor	0.8	46474	≈190	5.051	1.258	0.14	0.9
	Vanes	0.8						
	Hub	0.8						
	Flaps	0.8						
	Inlet	0.8						
Baseline	Rotor	0.4	51577	≈211	5.1	1.311	0.187	7.5
	Vanes	0.4						
	Hub	0.4						
	Flaps	0.8						
	Inlet	0.4						
Fine	Rotor	0.4	117888	≈482	5.089	1.313	0.184	19.4
	Vanes	0.2						
	Hub	0.4						
	Flaps	0.4						
	Inlet	0.2						

Table 8.1: Mesh settings for the aircraft wind tunnel model setup

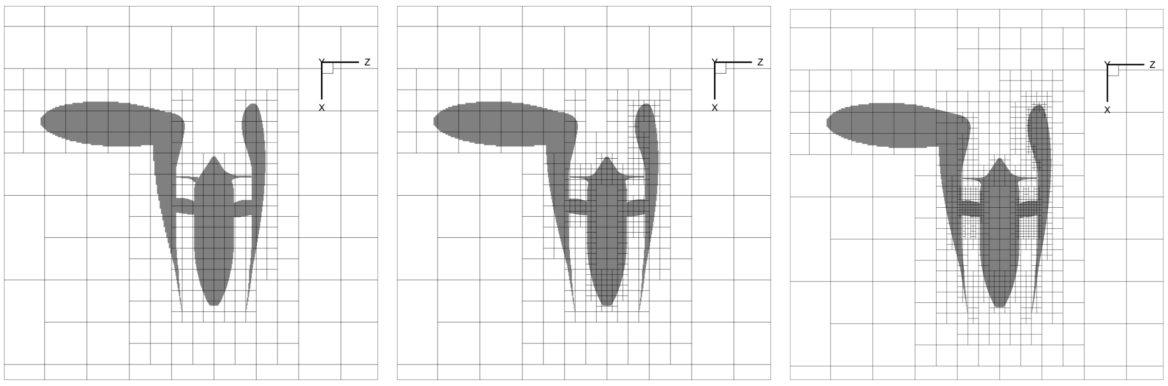
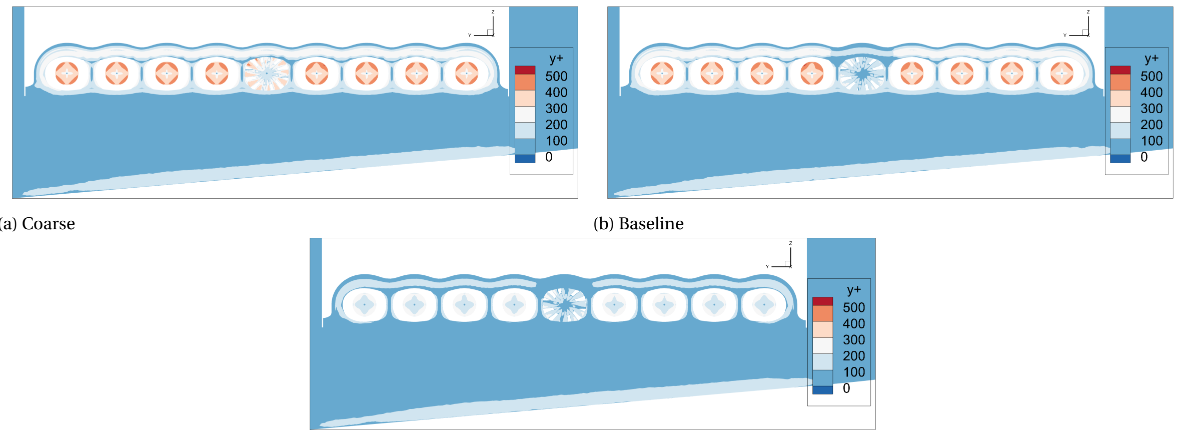

(a) Coarse (b) Baseline (c) Fine

Figure 8.6: Grid blocks with 16^3 cells of the three wind tunnel model meshes

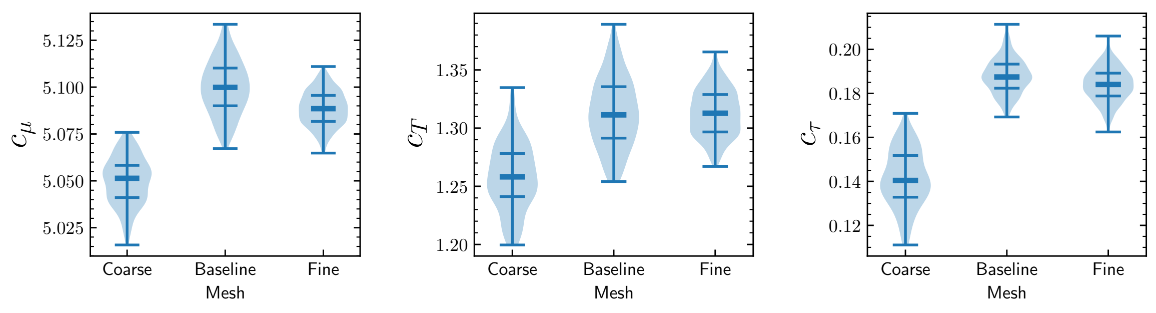


(a) Coarse (b) Baseline (c) Fine

Figure 8.7: Non-dimensional wall distance y^+

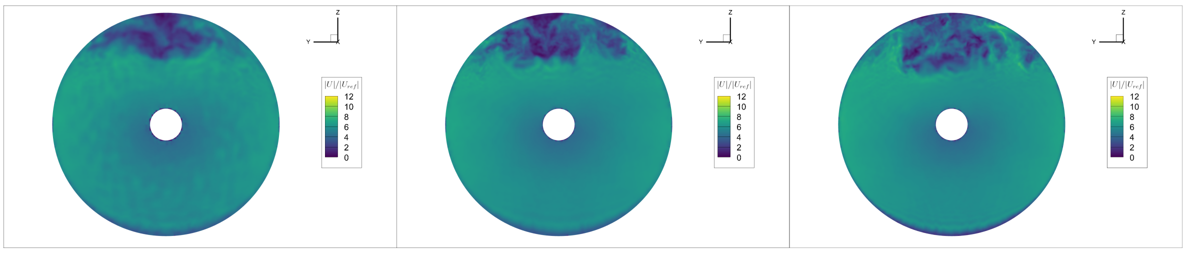
The mass flow on the *coarse* mesh is 1.2% lower than on the *fine* mesh (Fig. 8.8). The forces and moments have a difference approaching 25%, but the engine operates close to the wind-milling state, therefore the absolute magnitude is low.

Snapshots at the same solution time in Fig. 8.9–8.12 highlight some key differences in flow topology. Although the extent of inlet separation (Fig. 8.9) and the distortion patterns after the rotor (Fig. 8.10) are similar, the *coarse* mesh exhibits large vane separations near the hub on all vanes (Fig. 8.11). The corner separations are larger on the *baseline* mesh, but the hub separation causes a larger wake on the *fine* mesh (Fig. 8.12). The resulting similar effective outlet area can explain the small difference in $\dot{m}$.



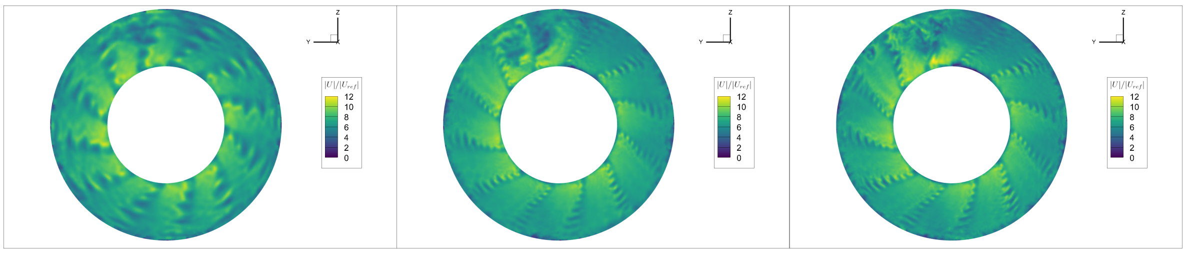
(a) Non-dimensional mass flow rate (b) Thrust coefficient (c) Torque coefficient

Figure 8.8: Comparison of integral quantities with three different mesh resolutions. The plots show the distribution, minimum, maximum inter-quartile range and the median.



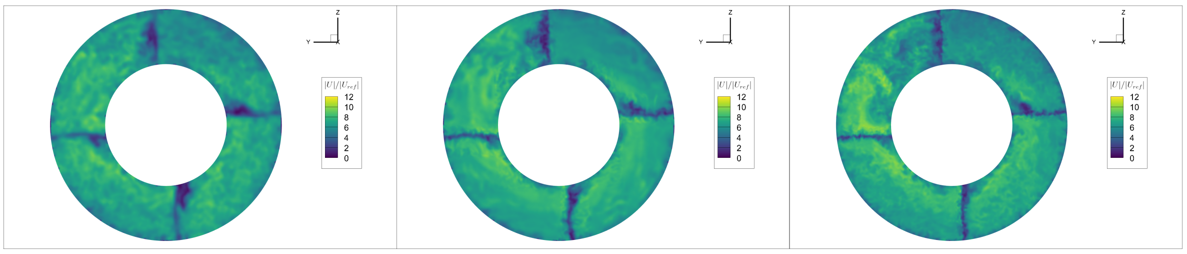
(a) Coarse (b) Baseline (c) Fine

Figure 8.9: Normalised velocity magnitude at Stage 2 on three different meshes



(a) Coarse (b) Baseline (c) Fine

Figure 8.10: Normalized velocity magnitude at Stage 3 on three different meshes

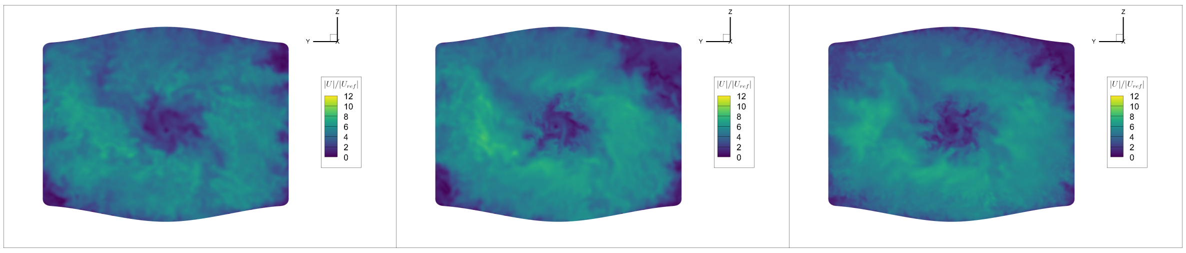


(a) Coarse (b) Baseline (c) Fine

Figure 8.11: Normalised velocity magnitude at Stage 5 on three different meshes

Fig. 8.13 shows the pressure distribution of the flap with the resolved engine. The mesh on the wing and flap bottom (TE–LE_f) remained unchanged. The changes are the biggest at the suction peak, which has 20% higher magnitude on the *fine* mesh. In this condition the suction peak is between two static ports, therefore it is hard to judge the accuracy of the solution in this region. Nevertheless, the changes between the *baseline* and *fine* mesh are smaller (5%) than from *coarse* to *baseline*.

Due to the acceptably small differences in integral quantities and the similar flowfield in the engine the



(a) Coarse (b) Baseline (c) Fine

Figure 8.12: Normalised velocity magnitude at Stage 6 on three different meshes

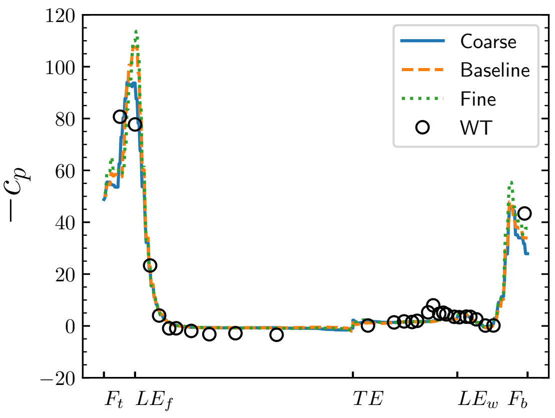


Figure 8.13: Static pressure in the symmetry plane of engine 5 on three different meshes

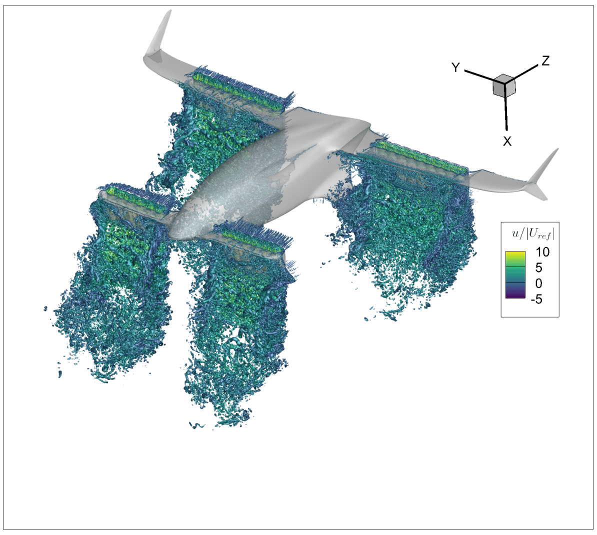
baseline mesh is deemed sufficient, and it is used as the basis of the 9 engine setup discussed in Section 8.4. Note, that the profiling in Section 7.5 showed a similar time step size and a smaller runtime for the *fine* mesh than for the *baseline* mesh. Based on this, accepting the *fine* mesh could be more advantageous, but it would drastically increase the mesh size, exceeding the memory limitations of the used hardware for 9 resolved engines.

8.4. RESULTS

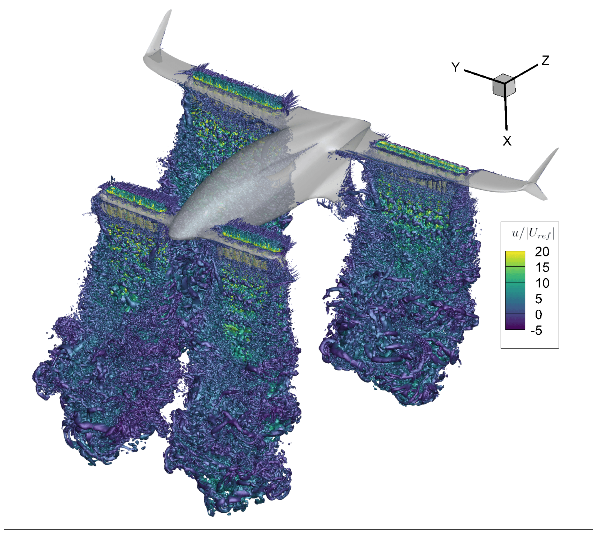
The non-dimensional RPM is defined as $\hat{\Omega} = \Omega/\Omega_{ref}$, where Ω_{ref} is close to the peak RPM of the engine used in the wind tunnel test. In the following, two cases at $\hat{\Omega} = 1/3$ and $\hat{\Omega} = 1$ are compared. The freestream velocity and all other boundary conditions are the same in both cases. Atmospheric pressure is specified on the front $(-Z)$ wall of the domain, which acts as an outlet in tailwind. Boundaries based on Riemann invariants are used to impose the the freestream velocity on all other faces of the cubic domain. Vreman's subgrid scale model [32] and an equilibrium wall stress model is used. The mesh settings are included in Table 7.4 in the previous chapter.

Both simulations ran until stable exhaust jets have developed. Although the engine mass flow rates without wind would converge earlier to a steady value, the jets can shift the stagnation point on the flap and the suction peak at the flap leading edge. Fig. 8.14 shows the jets of the two cases. In the $\hat{\Omega} = 1/3$ case the jets are dissipating in about half the distance from the airplane than in the $\hat{\Omega} = 1$ case. The $\hat{\Omega} = 1$ case has a less prominent vortex below the wing, a straighter jet (Fig. 8.15) and overall it looks similar to a calm air condition. The turbulent structures of the left wing (which has resolved engines) jet look similar to the other arrays.

The velocity and pressure fields at the engine stages are shown in Fig. 8.16–8.17. At $\hat{\Omega} = 1/3$ a large separation bubble is visible (Stage 2) at all engines, which is more pronounced towards the middle of the array. The distortion is still visible downstream of the rotor (Stage 3). Several stator vanes have flow detachment close to the hub or mid-span (Stage 5). There are corner separations in diagonally opposite corners of the outlets (Stage 6). The separation is more pronounced in the upper right quadrant, which is exposed to the inlet distortion. The upper corner separation is absent on the outboard corner engine (on the right in Fig. 8.16). It is worth noting that the corner engines have a different outlet design, with a rounded shape in this area.

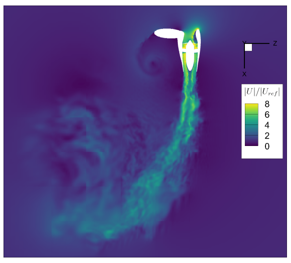


(a) $\hat{\Omega} = 1/3$

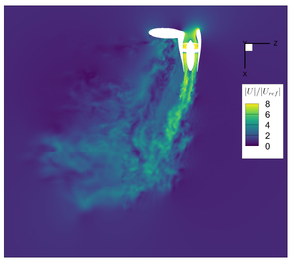


(b) $\hat{\Omega} = 1$

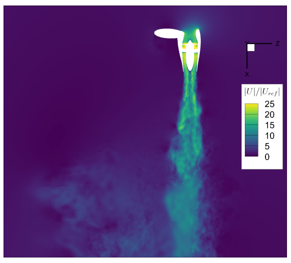
Figure 8.14: Q-criterion=10 iso-surfaces coloured with axial velocity



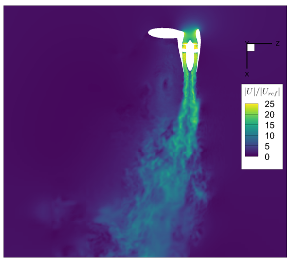
(a) Middle engine ($\hat{\Omega} = 1/3$)



(b) Outboard engine ($\hat{\Omega} = 1/3$)



(c) Middle engine ($\hat{\Omega} = 1$)



(d) Outboard engine ($\hat{\Omega} = 1$)

Figure 8.15: Velocity field in the engine cross-sections

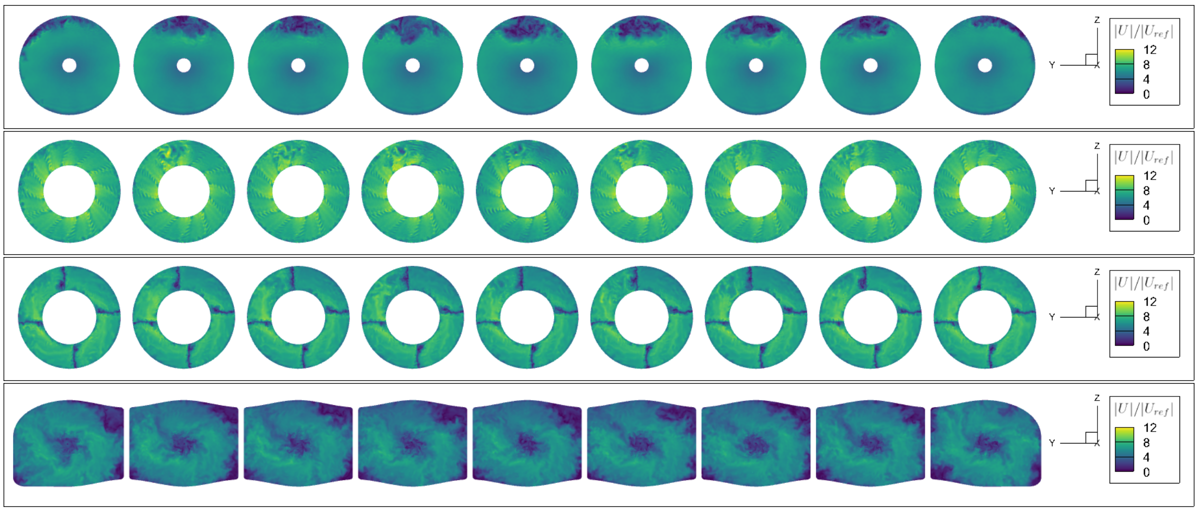


Figure 8.16: Velocity magnitude at Stage 2, 3, 5 and 6 ($\hat{\Omega} = 1/3$)

The pressure field is homogenous at Stage 2 (Fig. 8.17). Acoustic waves are visible downstream of the rotor (Stage 3), and both Stage 3 and 5 show a 4 per revolution pattern. There is a large pressure drop in the hub wake and patches of low pressure in the corner separation zones (Stage 6). The pressure coefficient is lower in engine 5 on this snapshot, this feature is not present at all time steps, only occasionally on some of the engines.

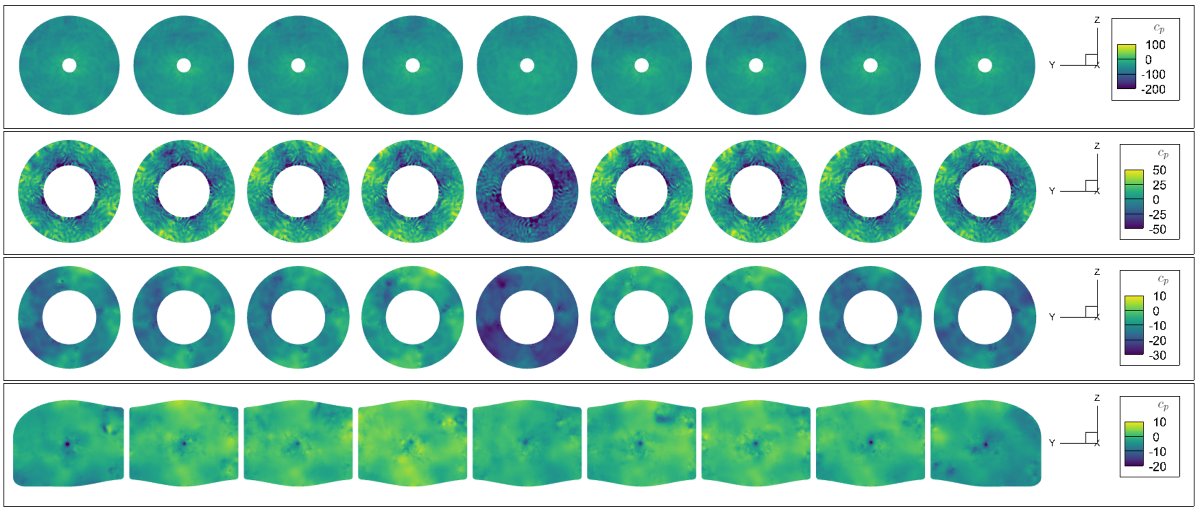


Figure 8.17: Pressure coefficient at Stage 2, 3, 5 and 6 ($\hat{\Omega} = 1/3$)

The inlet distortion is smaller at $\hat{\Omega} = 1$, especially on the corner engines (Fig. 8.18). A thickened boundary layer on the bottom can be attributed to the boundary layer development on the wing (see also Fig. 8.2b). The flowfield is mostly axisymmetric downstream of the rotor (Stage 3), but separations are still present at the vane roots (Stage 5). The corner separations at the outlet (Stage 6) are more symmetric due to the lower inflow distortion, and they have a smaller extent than for $\hat{\Omega} = 1/3$. A large swirl can be observed by comparing the clocking of the vane wakes at Stage 5 and 6.

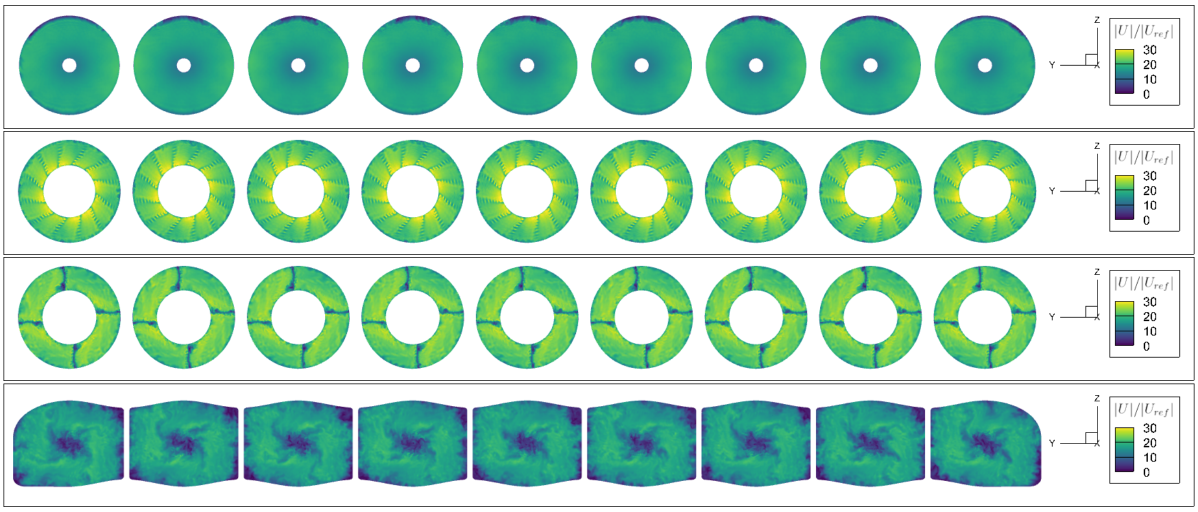


Figure 8.18: Velocity magnitude at Stage 2, 3, 5 and 6 ($\hat{\Omega} = 1$)

The mass flow of $\hat{\Omega} = 1/3$ drops by 5% on the middle engines, showing a parabolic distribution (Fig. 8.20), similar to the distribution of the inlet distortion in Fig. 8.16. The $\hat{\Omega} = 1$ distribution is more uniform along the span. The distribution within an individual engine comes from a periodic fluctuation of the mass flow with the blade passing frequency.

The thrust has a parabolic spanwise distribution at both rotational speeds (Fig. 8.21), with an up to 10% higher thrust on the corner engines at $\hat{\Omega} = 1/3$ and up to 2% at $\hat{\Omega} = 1$.

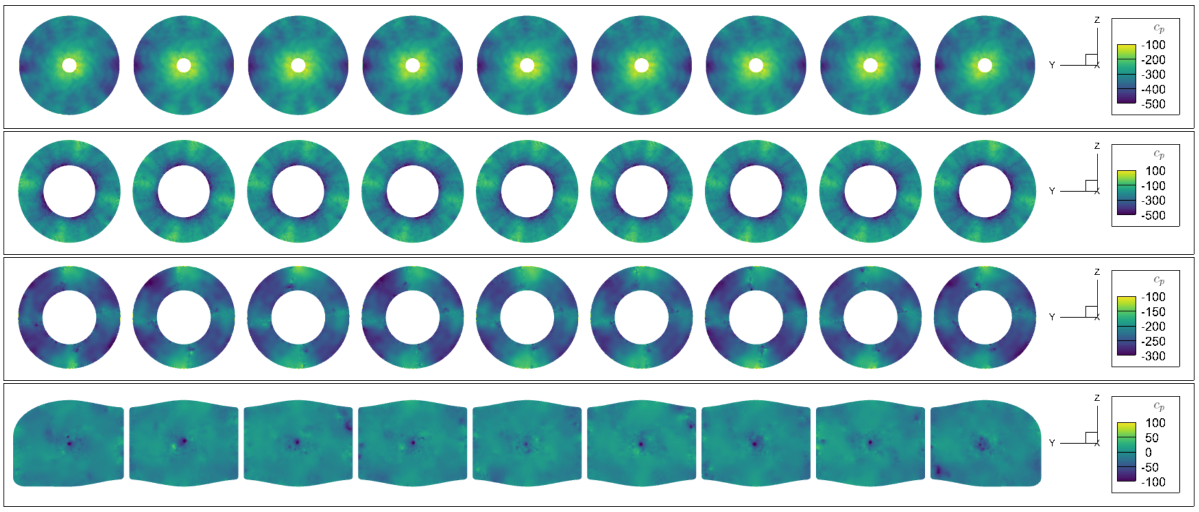


Figure 8.19: Pressure coefficient at Stage 2, 3, 5 and 6 ($\hat{\Omega} = 1$)

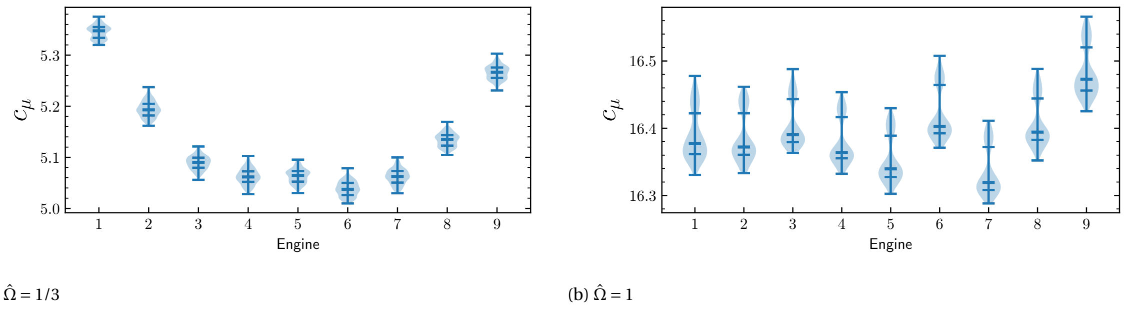


Figure 8.20: Non-dimensional mass flow rates of the engines on the left wing. The plots show the data distribution, minimum, maximum, inter-quartile range and median of each engine.

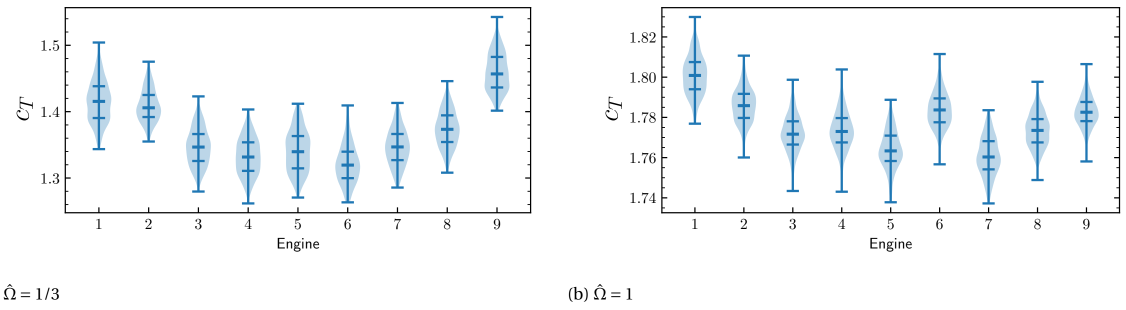
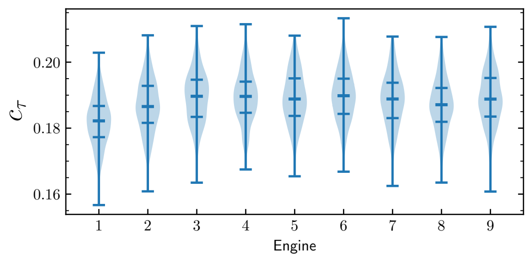
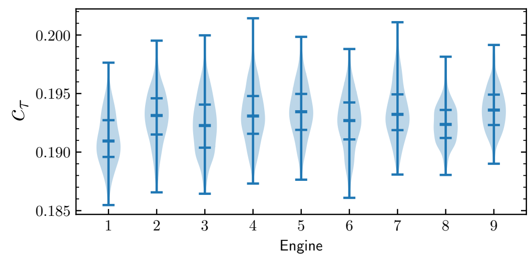


Figure 8.21: Thrust coefficients of the left wing engines. The plots show the data distribution, minimum, maximum, inter-quartile range and median of each engine.

The torque — and hence the shaft power — is almost constant along the span in both cases, with a small drop on the inboard engines (Fig. 8.22).



(a) $\hat{\Omega} = 1/3$



(b) $\hat{\Omega} = 1$

Figure 8.22: Torque coefficients of the left wing engines. The plots show the data distribution, minimum, maximum, inter-quartile range and median of each engine.

8.5. Comparison with wind tunnel results

The two operating points introduced in this chapter are compared with the corresponding wind tunnel results. The measurement data was acquired in the Low Speed Tunnel of the German-Dutch Wind Tunnels [77]. The test setup is shown in Fig. 8.23. The wind tunnel walls and sting were not modelled in CFD, and no correction for blockage was applied. The wind tunnel model was equipped with inlet rakes on two engines of the left wing, outlet rakes on all engines, and static pressure taps on every second flap section. Mass flow rate and thrust were calculated from the outlet rake data, employing a calibration based on an engine static test with more instrumentation.

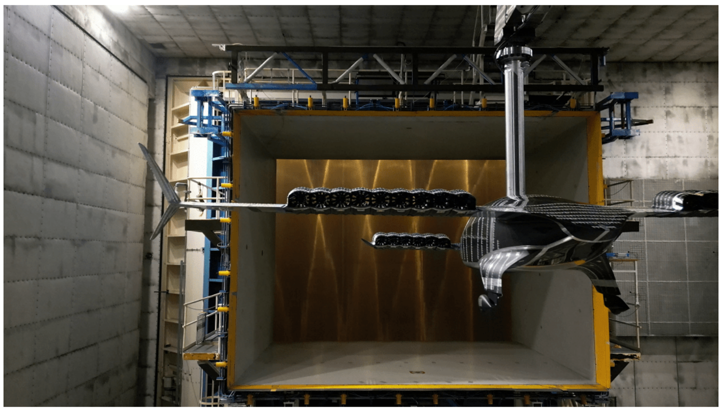
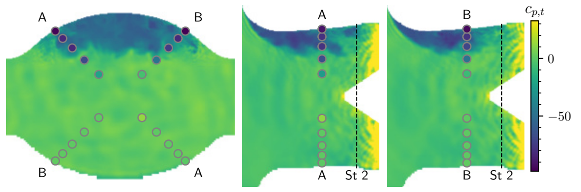


Figure 8.23: The aircraft model in the DNW low-speed wind tunnel (Source: lilium.com, retrieved 28.3.2025)

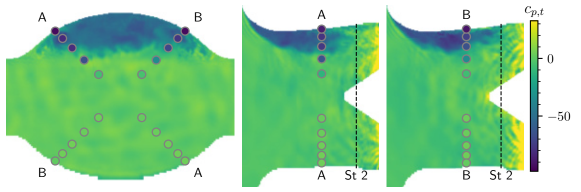
8.5.1. Inlet rakes

Both the CFD results and the inlet total pressure rake measurements indicate a large separation at the top of the inlet (Fig. 8.24). Although the extent of the separation in CFD in the symmetry plane is similar to that of the probes along the diagonals, at the location of the probes both the extent of the separation and the total pressure drop is smaller in CFD (Fig. 8.25).

This can be caused by too much dissipation on an under-resolved mesh, but the mesh convergence study results (Fig. 8.9) do not indicate this. Another possibility is that the total pressure probes do not provide a reliable measurement in this highly distorted inflow. First, the probes are meant to be aligned with the flow direction, but the local flow angles are not known a priori, and the probes were calibrated and placed to measure accurately in normal operating conditions with less severe inflow distortion, with only minor secondary flows in the measurement plane (mainly from the square-to-circular shape transition of the inlet). Since the probes are placed axis-aligned (and not aligned with the streamlines), they will measure only the axis aligned component of the total pressure (Eq. 8.6). In case of a flow reversal the probes would not measure

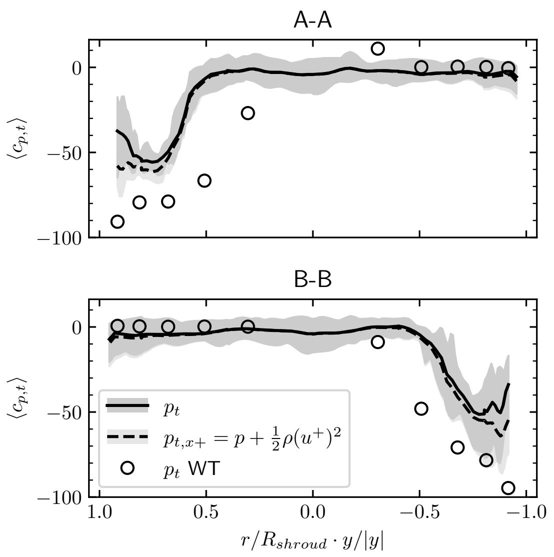


(a) Engine 4

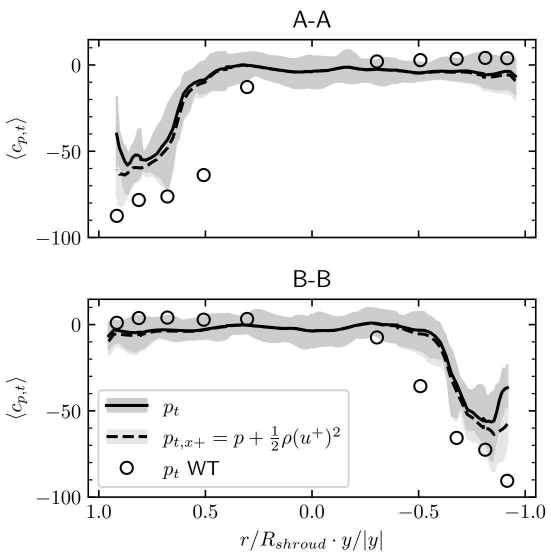


(b) Engine 6

Figure 8.24: Total pressure coefficient at the inlet. Total pressure probes highlighted with circles, instantaneous LES results in the background ($\hat{\Omega} = 1/3$).



(a) Engine 4



(b) Engine 6

Figure 8.25: Total pressure along the inlet rake diagonals ($\hat{\Omega} = 1/3$). The shading shows the temporal fluctuations.

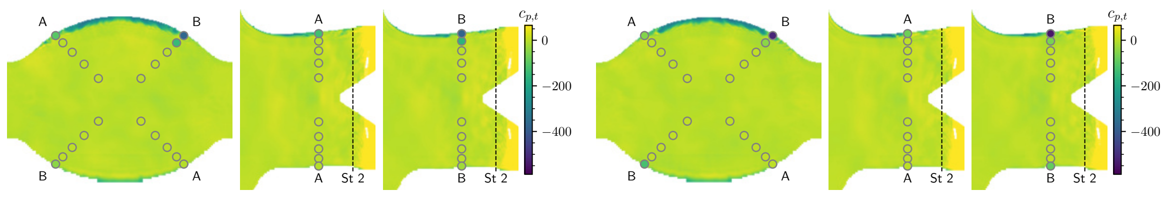
the velocity, therefore only the positive axial velocity component u^+ is considered.

$$c_{p,t,x+} = p + \frac{1}{2}\rho\left(u^+\right)^2 \tag{8.6}$$

If the difference between the local velocity vector and the probe is large, it can lead to flow separation on the probe, which corrupts the measurement. The exact shape of the total pressure probes was not recorded, but according to Gracey [78] the best shapes provide accurate measurements in a 28° range, which was exceeded on the top 2 probes on the upper arms in the $\hat{\Omega} = 1/3$ operating point (Fig. 8.28). The local flow angle is shown in Fig. 8.28, where the angle of attack of the probe is defined based on the local velocity perpendicular to the probe axis:

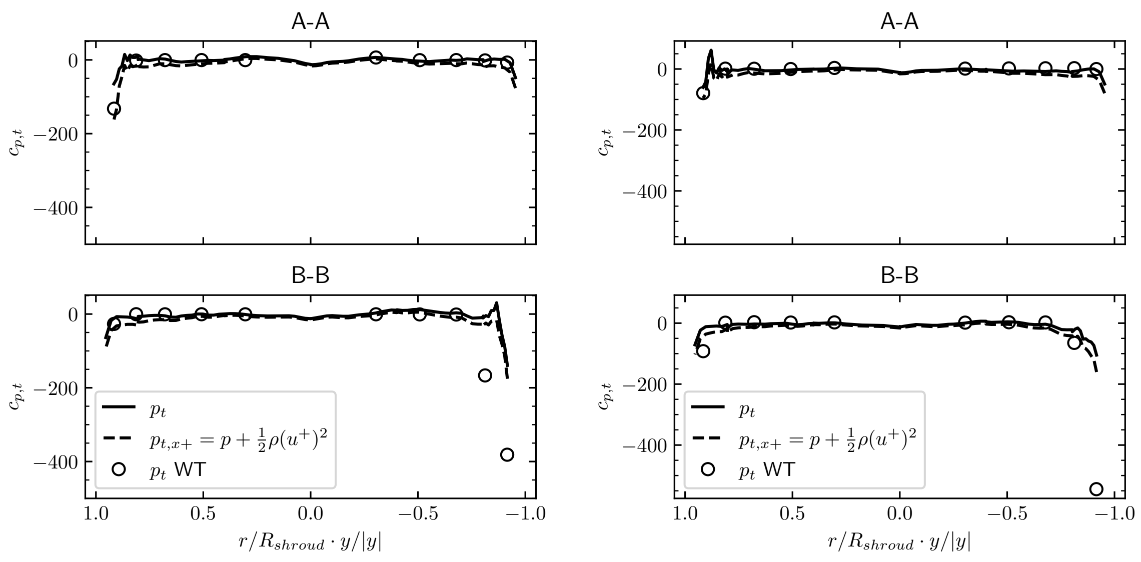
$$\alpha_{probe} = \sin^{-1}\left(\frac{v^2 + w^2}{|U|}\right) \tag{8.7}$$

In the $\hat{\Omega} = 1$ case, the measurements and the CFD flowfield match well in the bulk flow. This is expected, since the flow in the inlet is attached, the bulk flow can be considered as isentropic, therefore the total pressure in the measurement plane matches the freestream (Fig. 8.26). The probes at the highest radius are placed close to the boundary layer. These probes (except the upper A arm on both engines) are close to the CFD values (Fig. 8.27). It is possible that the upper A arm is disturbing the boundary layer or even touching the wall. The nominal distance from the wall is small, and the probe can be misaligned due to assembly tolerances. This was not measured, therefore this hypothesis cannot be confirmed.



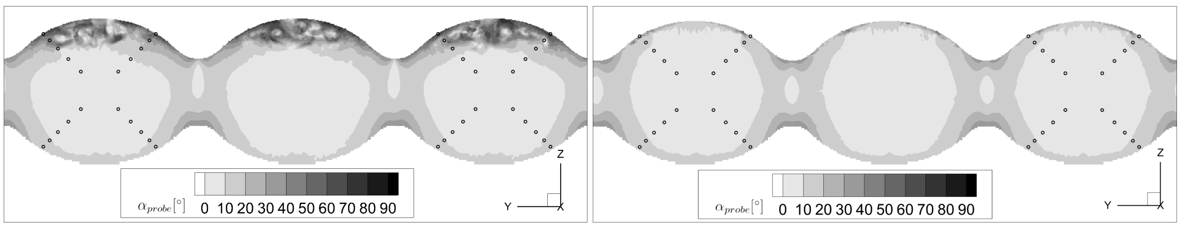
(a) Engine 4 (b) Engine 6

Figure 8.26: Total pressure coefficient at the inlet. Total pressure probes highlighted with circles, instantaneous LES results in the background ($\hat{\Omega} = 1$).



(a) Engine 4 (b) Engine 6

Figure 8.27: Instantaneous total pressure coefficient along the inlet rake diagonals ($\hat{\Omega} = 1$)



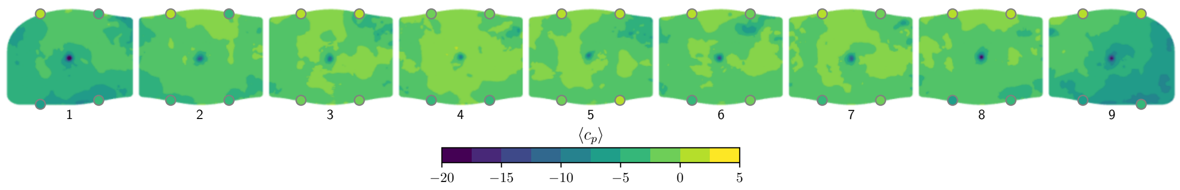
(a) $\hat{\Omega} = 1/3$ (b) $\hat{\Omega} = 1$

Figure 8.28: Inlet total pressure probe angle of attack in engine 4–6

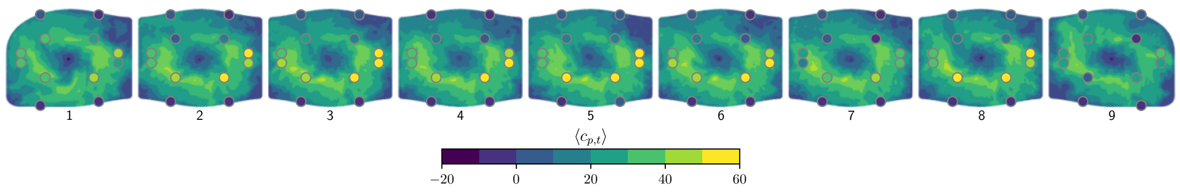
8.5.2. Outlet rakes

The outlet of each engine was equipped with a total pressure rake and pressure taps (Fig. 8.29). A pair of pressure taps was placed on the top and bottom of each outlet. At $\hat{\Omega} = 1/3$ the static pressure field is mostly homogenous, except the hub wakes and some corner separation regions show a pressure drop. The total pressure field has higher gradients due to the many flow features, such as corner separations and wakes. It would not be possible to reconstruct such a complex flowfield from merely 8 point values, but in the neighbourhood of the probes the CFD flowfield shows similarities, e.g. the top 2 pressure probes have a lower value than the bottom ones. The "north east" probes in engine 7 and 9 presumably show a faulty value due to leakage.

At $\hat{\Omega} = 1$ the pressure is homogenous, except in a small area downstream of the hub.

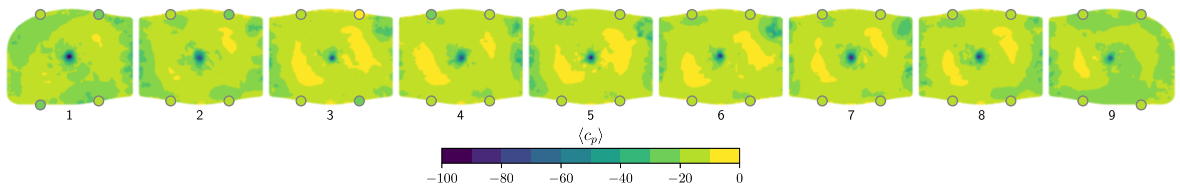


(a) Static pressure coefficient

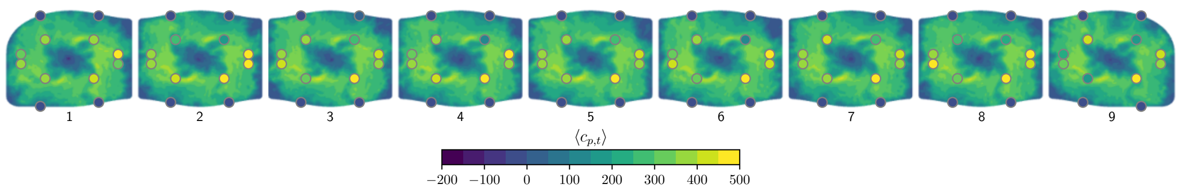


(b) Total pressure coefficient

Figure 8.29: Averaged pressure at Stage 6. Pressure probes marked with circles ($\hat{\Omega} = 1/3$).



(a) Static pressure coefficient



(b) Total pressure coefficient

Figure 8.30: Averaged pressure at Stage 6. Pressure probes marked with circles ($\hat{\Omega} = 1$).

8.5.3. SURFACE PRESSURE

The wind tunnel model surface was instrumented with 30 static pressure probes in each section. The sections are located in the symmetry plane of flap 1, 3, 5, 7 and 9. The accuracy of the static pressure probes is $\pm 2.3 \cdot c_p$. The corrections applied to the wind tunnel data are described in Appendix F. Fig. 8.31 shows the location of the taps and the pressure distribution extracted from the averaged CFD solution.

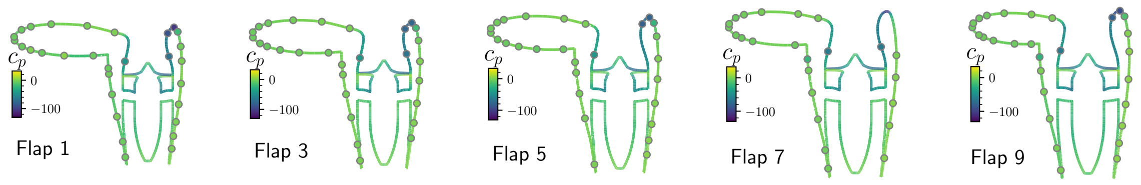


Figure 8.31: Engine slice pressure distribution. Pressure probes marked with circles ($\hat{\Omega} = 1/3$).

Fig. 8.32 contains the same information, but the values are easier to read. This plot shows the pressure distribution along a surface parameter (Fig. 8.33), which runs from the top of the fan face (Stage 2) along the inlet to the flap leading edge (LE_f), it goes around the flap to the outlet (TE_t). The pressure distribution between the top and bottom of the outlet (TE_t and TE_b) is not shown, since the engine duct was not instrumented with pressure probes in this cross-section (TE_t and TE_b are marked as a single point TE). The line continues from the TE on the bottom of the flap, it wraps around the wing and returns back to Stage 2. The CFD results show a good match. The suction peak is under-predicted up to 25% on the middle flaps (3, 5 and 7).

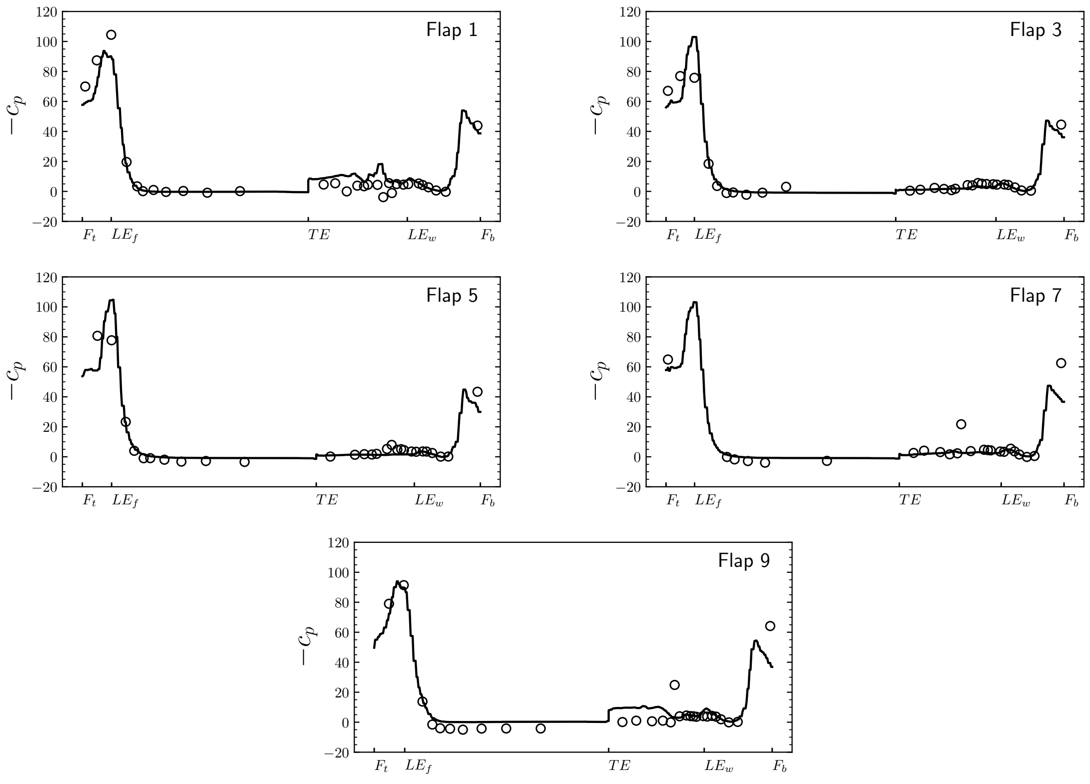


Figure 8.32: Engine slice pressure distribution along the surface, starting from the top of Stage 2 (F_t), going clockwise around the flap. Pressure probes marked with circles ($\hat{\Omega} = 1/3$).

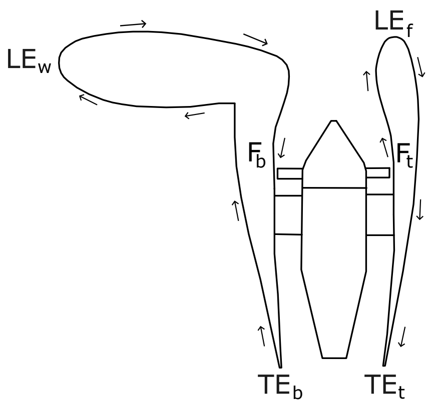


Figure 8.33: Surface parameter

At $\hat{\Omega} = 1$ there is a strong suction peak at the flap LE and inside the engine inlets (Fig. 8.34). The pressure inside the inlet matches well the experimental data (Fig. 8.35), but the suction peak is under-predicted by up to 25% on the middle flaps. On the corner flaps (1 and 9) the pressure matches well the measurement. On the wing and flap bottom the surface pressure is close to the atmospheric pressure.

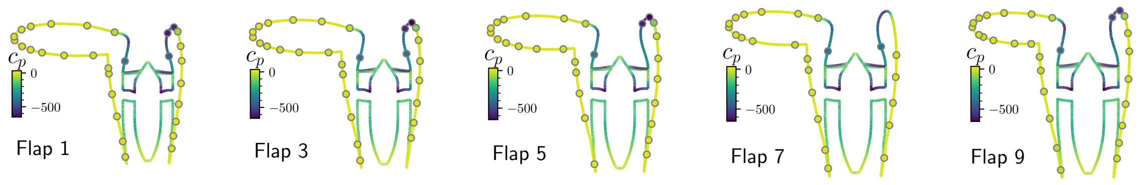


Figure 8.34: Engine slice pressure distribution. Pressure probes marked with circles ($\hat{\Omega} = 1$).

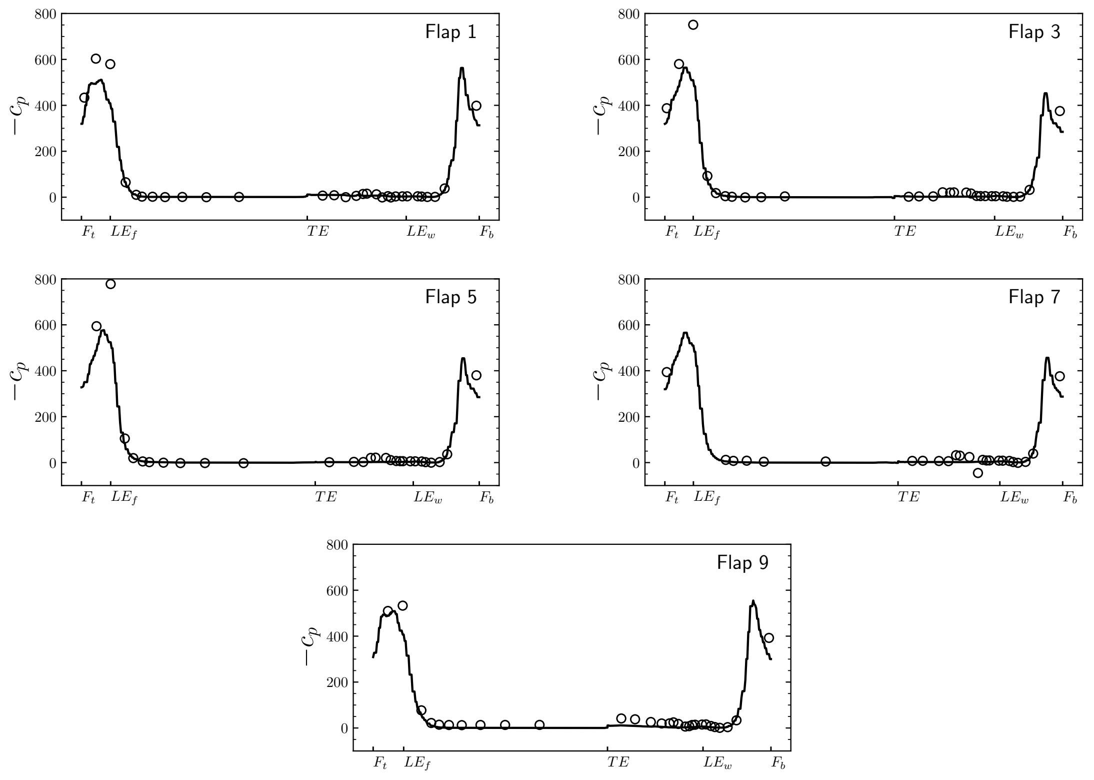


Figure 8.35: Engine slice pressure distribution along the surface, starting from the top of Stage 2 (F_b), going clockwise around the flap. Pressure probes marked with circles ($\hat{\Omega} = 1$).

8.5.4. Integral quantities

The small number of total pressure probes on the outlet rakes makes it difficult to reconstruct integral quantities, such as mass flow or thrust, purely from the direct measurements. The engines used in the wind tunnel were calibrated in a static test, where the engines were equipped with a bellmouth and the operating point was regulated with the back-pressure of the outlet. The mass flow was measured with a traversing hot wire and the total axial force — which approximates well the thrust in this setup — was measured with a balance. Beside the clean inflow baseline, the measurements were also repeated with a distortion screen, and calibration curves were established for clean and distorted inflow. Fig. 8.36 and 8.37 show both the raw measurement, the calibrated values and the 95% confidence interval of the measurement. The measured values are offset by 0.2 on the X axis for better readability.

The CFD results show a good match, and except engine 7 and 9 the values are within the confidence interval of the measurement. The thrust values at $\hat{\Omega} = 1/3$ have a large confidence interval, because the force magnitude is low in this operating point, increasing the measurement uncertainty. The uncertainty of RPM and freestream velocity was not considered in the confidence interval calculation, assuming they have a much smaller uncertainty than the mass flow rate and the thrust. Both variables have a slightly parabolic distribution in CFD at $\hat{\Omega} = 1/3$, showing lower values towards the center of the flap, which correlates with the extent of the inlet separation. This is not visible in the experimental results, likely because the outlet

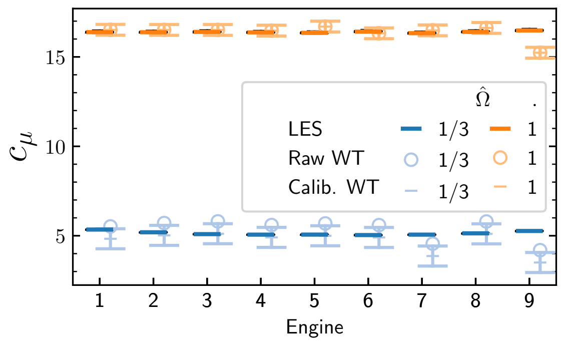


Figure 8.36: Non-dimensional mass flow rates on the left wing

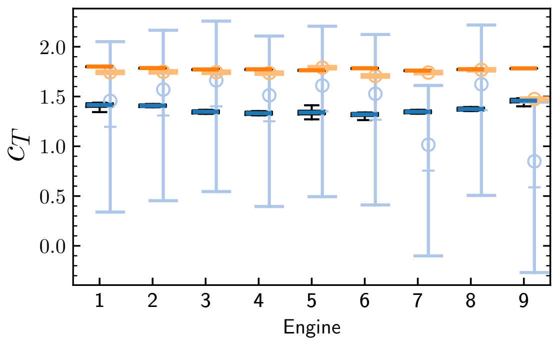


Figure 8.37: Thrust coefficient on the left wing

rake cannot capture changes in corner separation and hub wake regions, since the total pressure probes are positioned far from these. For the calibration cases this error is removed, but the installed engines have a different inlet distortion pattern, which on top of this also varies along the span.

8.6. Summary

An eVTOL aircraft with distributed electric propulsion was simulated with WMLES using a mesh with 316 million cells. The simulations had a one day turnaround time on 8 Nvidia A100 GPUs, which is sufficient to enable the evaluation of many certification relevant off-design conditions of the propulsion system or even to iterate the inlet or engine design based on high-fidelity simulation results. The CFD results were compared with wind tunnel measurements. The engine mass flow and thrust are within the measurement uncertainty of the experimental values. The LES flowfield captures the main flow features observed in the experiments, such as the inlet separation at the top of the inlets and inhomogeneities in the outlet total pressure field. The extent of the inlet separation is slightly under-predicted, and the total pressure probe readings in the outlet show higher values than the CFD. The static pressure taps on the surface show a good match, except a mismatch in suction peak magnitude on the middle flaps.

9
Conclusion and discussion

9.1. Conclusion

The thesis aimed to optimise an existing GPU-native high-fidelity Wall-modelled Large Eddy Simulation (WMLES) moving Immersed Boundary (IB) solver, enabling large-scale distributed electric propulsion applications. The main bottlenecks of the baseline solver were presented, that needed to be tackled to enable this use case. The impact of these bottlenecks, namely the data redistribution cost in Adaptive Mesh Refinement (AMR) steps and the IB update in moving geometry were evaluated on a static and dynamic setup.

Several dynamic load balancing concepts and algorithms were presented and evaluated. A parallel partitioner using locally-matched multilevel scratch remap (LMSR) and wavefront diffusion was integrated into the present solver. An additional constraint was added to the multi-objective load balancing to equally distribute modelled engines between partitions. An iterative initial refinement step was added to improve the partitioning quality at the start of the simulation. The second bottleneck was traced back to the voxelizer and to the signed distance function (SDF) update. Block filtering steps were added in the voxelizer kernel to reduce the amount of computations and global memory usage. The SDF update was accelerated by using an optimised distance function. Modifications of this algorithm were explored and tested with various geometries. Finally, an alternative SDF update algorithm was designed and implemented. This showed promising performance on simpler test cases, but suffered from robustness issues.

Two off-design tailwind condition were simulated with the optimised version of the WMLES solver. The high RPM operating point showed an increased inlet distortion, but no inlet separation, whereas in the low RPM case all engines showed large inlet separations on all resolved engines. The simulation results were compared with wind tunnel data, with a focus on the inlet distortion and engine performance. The solver accurately predicted the mass flow and thrust of the engines within the measurement uncertainty, and it accurately captured the main flow features. With large inflow distortion there were discrepancies in the total pressure field, which should be investigated in the future.

The goal of the present work was to answer the following research questions:

How strong is the effect of tailwind on the engine inlet flow field?

Two tailwind cases were simulated to represent relevant off-design conditions. In the high RPM case, the attached inlet flow and thickened boundary layer at the top of the inlet match well the wind tunnel results. At low RPM both CFD and the total pressure rakes of the measurement show a large inlet separation. The extent of the separation and the total pressure drop show discrepancies, which can be attributed either to numerical errors or to the probe placement. The outlet rake data shows similar patterns as the CFD flowfield. There are discrepancies in the total pressure magnitude, the WT data showing higher variations. Pressure tap data extracted from the flap and wing surface matches well the CFD flowfield, except the simulations underpredicting the suction peaks at the flap top leading edges. There is no available data to compare the shape or vortical structures of the exhaust jets.

How strongly is the engine thrust and torque affected by off-design tailwind conditions?

At the low RPM operating point, the CFD results show a 6% thrust drop on the middle engines compared to the less affected corner engines, which are close to their clean inflow operating point. The mass flow follows a similar trend. The trend of thrust distribution along the wingspan is less clear in the measurement data, which at this operating point can be explained by the large measurement uncertainty. The torque varies 5%, and it is the lowest on one of the corner engines. This can be attributed to the inlet distortion lowering the compressor efficiency in the middle engines. At high RPM the variation in thrust and torque is only 2%, resulting in only a minor performance penalty. Some of the loss sources, such as the outlet corner separations and vane separations are presumably also present to some extent in similar operating points at calm air conditions, and are not merely a consequence of the increased inlet distortion in tailwind. This can partially mask the potential performance penalty due to tailwind. The observations are only valid for the investigated geometry. Small changes in the flap and fan design can considerably change the performance of the engines in tailwind.

How accurately can the present solver predict engine performance compared to wind tunnel data?

Integral quantities extracted from the flowfield were compared with engine mass flow rate and thrust recorded in the wind tunnel. In the measurement the integral quantities are reconstructed from the outlet rake static and total pressure measurements and calibrated with hot wire and load balance data from a single engine static test setup. Except two engines in the low RPM and one engine in the high RPM case, the mass flows fall within the confidence interval of the measurement. In these three engines the outlet rake probes have a constant bias compared to the other engines at the same operating point. Likely causes include leakages in the piping of the data acquisition system and assembly issues causing an efficiency drop in these engines. No physical explanation was identified that could explain the effect with the nominal aircraft geometry. The mass flow drop also propagates to the thrust calculated from the measurement data, showing similar trends. The other engines at low RPM are within the confidence interval of the measurement, but due to the low force magnitudes the measurement has large error bars. At high RPM the simulation predicts a higher thrust on most engines. The highest deviation between CFD and experiment is 4%, but the measurement also shows up to 5% difference between adjacent engines, which is unexpected in this condition.

How much can the AMR be accelerated by using different partitioning algorithms?

The largest bottleneck was the data redistribution cost of the baseline partitioner during AMR steps. Replacing it with an adaptive re-partitioner using the Unified Repartitioning Algorithm removed this bottleneck. In the baseline version the AMR took up to 85% of the runtime, and with the optimised code it does not exceed 12%, and in most cases it is only about a 5% fraction. Using an iterative initial refinement of the partitioning graph further reduced the runtimes, and the up to 10% edge-cut reduction achieved with this technique is also applicable for static setups.

What speed-up is expected from using efficient algorithms for the immersed boundary update?

The voxelizer and SDF update steps of the IB framework were reworked to achieve a faster geometry update in moving setups. The update flag function containing these two steps became 3 times faster in a full aircraft simulation, but in one of the smaller test cases it caused a marginal penalty. The SDF update can still take about a quarter of the runtime. Both the baseline and the optimised algorithm can become imbalanced, when a locally fine surface triangulation is combined with a coarse volume mesh. Hence, the performance impact of the optimisation of these functions is very case-dependent.

What is runtime of a full aircraft DEP simulation?

A one-day turnaround time was achieved for full aircraft simulations with 9 resolved engines. The simulations were running until the exhaust jets fully developed, which took 0.1–0.2s physical time based on the engine RPM. Aircraft integral forces, engine mass flow rates and thrust converged in less than half of this time. For use cases not involving ground effect, canard-wing interaction or other large time scale flow features this reduces runtime can be sufficient. Simulations of the full aircraft with only one resolved engine and running for 0.1s physical time converged in 8 hours. A single resolved engine can already provide useful insights about installed engine performance and BLI effects. Overnight simulations can enable fast design cycles or analysing a large number of flight condition. Since the presented DEP use case was simulated for the first time with high fidelity methods, it was not possible how these runtimes compare to other solvers.

9.2. FUTURE WORK

LOAD BALANCING

The switch from a scratch remapping partitioning to the Unified Repartitioning Algorithm brought the biggest performance improvements with a more than 50× reduction in the AMR time in large simulation setups. On the other hand, there is still room for improvement. Studying different parameter values of the imbalance tolerance and the inter-processor communication to data redistribution time ratio (ITR) showed no clear trends and there is no universally optimal value for them in the present solver. At the same time, the runtime varied up to 10% with different combinations of these parameters. Adapting ITR and imbalance tolerances based on the mesh or solver settings, or adjusting them dynamically during the run based on iteration and AMR timers could lead to potential runtime reduction with a small implementation effort.

SIGNED DISTANCE FUNCTION UPDATE

Despite improvements in the point-to-triangle distance calculation, and after extensive kernel optimisations, SDF remained one of the bottlenecks of the solver. The larger simulation setups still spend 1 to 30% of the runtime in this kernel, and currently it needs careful surface and volume meshing to mitigate the bottleneck. Both in improved versions of the baseline algorithm, and in an alternative approach using triangle proximity instead of the triangle list of grid blocks constructed by the voxelizer, the algorithm becomes unbalanced when the surface mesh has a high concentration of faces or extreme aspect ratios. It is proposed to explore algorithms, that either use a finer voxelization than the discretisation of the fluid domain or refine the voxelization to limit the number of potential candidates during the closest triangle search.

FURTHER PROPULSION TEST CASES

The solver showed a reasonably good match with wind tunnel data, but a measurement with more extensive instrumentation could provide further insight. The wind tunnel model with 30 engines and flap actuators, and hundreds of static and total pressure probes is among the most mechanically complex wind tunnel models. But due to the small size and limited number of measurement points in the inlet and outlet it is difficult to accurately estimate engine performance parameters, such as mass flow, thrust, shaft power and inlet distortion. Wind tunnel or engine static tests equipped with hot-wire measurements, particle image velocimetry or other high resolution instruments could provide a more unambiguous reference. Existing datasets, such as [28, 79, 80] could be used for this purpose.

A
Applicability of barotropic model

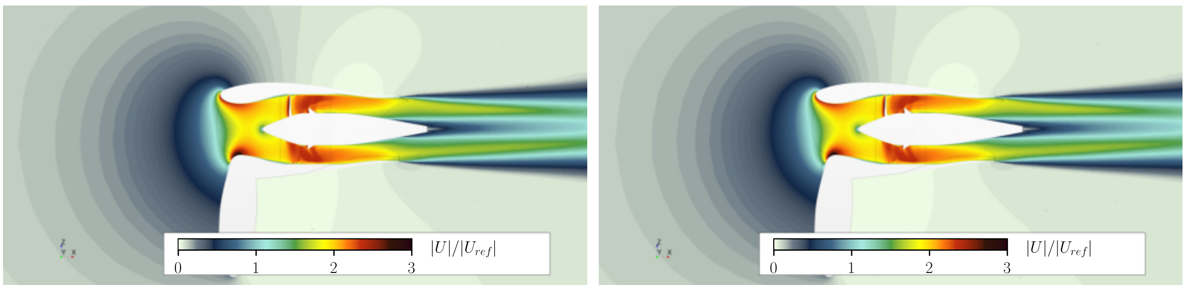
A.1. Introduction

The LES solver uses a barotropic fluid model (Chapter 3). This approach simplifies the governing equations compared to an ideal gas model, but it is only applicable for low Mach number flows. In use cases, such as the hover of a rcraft with tailwind described in Chapter 8, the freestream is low Mach number, but high speed flow can occur on the blade tips and on the inlet lip, when the engine RPM is high. Implementing the ideal gas model, especially the addition of the energy equation, would need considerable rework of the present solver, and it would without doubt result in higher runtimes.

Therefore, a different approach was taken, and the barotropic equation of state was implemented in a commercial solver (Siemens Star CCM+) to compare the fluid models. The following study quantifies the error between the barotropic fluid model and the (perfect) ideal gas fluid model with a geometry similar to the one used in the Slice setup in Section 7.4. In this case the geometry is not scaled to the wind tunnel model size and a different engine design is included. Throughout this study steady RANS is used with Menter's SST turbulence model [81].

A.2. Results

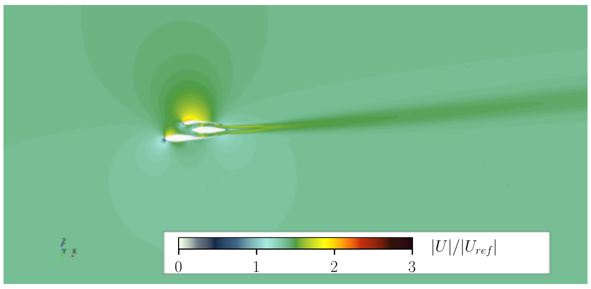
In hover the flap is deflected 90°, and the boundary conditions represent calm air. In cruise the flap is not deflected relative to the wing and a velocity inlet boundary condition is used with the cruise speed. The velocity fields in hover (Fig. A.1) are almost indistinguishable. In cruise (Fig. A.2) there are slight differences in the velocity contours below the wing and around the exhaust jet. The flowfield within the engine duct is nearly identical.



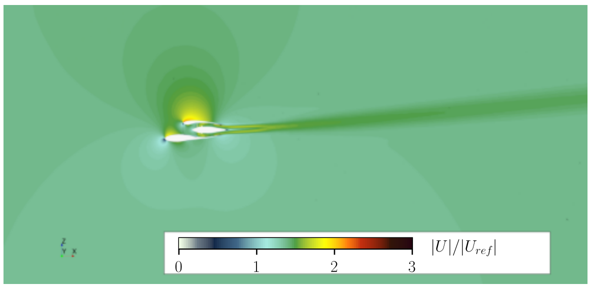
(a) Ideal gas (b) Barotropic fluid

Figure A.1: Normalised velocity field in hover ($\hat{\Omega} = 1$)

Fig. A.3a shows the non-dimensional mass flow rate (c_μ), thrust coefficient (c_T), engine shaft torque coefficient (c_τ), lift coefficient (c_L) and drag coefficient (c_D). The engine coefficients are defined in Eq. 8.3–8.5. The thrust is 2% higher with the barotropic assumption and the drag coefficient differs by almost 20%. This may seem as a large error, but in this flight condition, for an infinite array of propulsion units (since periodic boundaries are used on the side), the aerodynamic forces are low, and the surfaces wetted by the engine



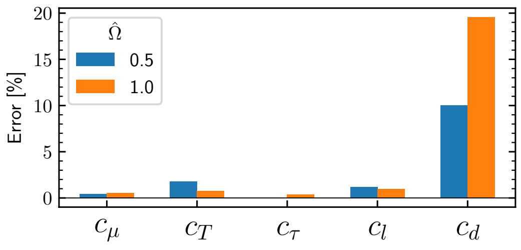
(a) Ideal gas



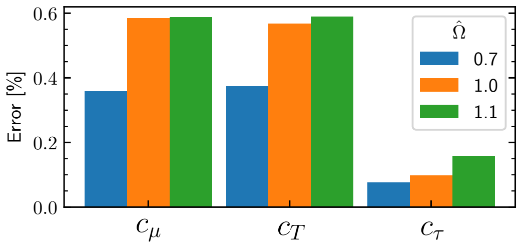
(b) Barotropic fluid

Figure A.2: Normalised velocity field in cruise ($\hat{\Omega} = 0.5$)

streamtube are accounted for in the thrust instead of lift and drag. Fig. A.3b shows the error in hover. The lift and drag is omitted, because the magnitude of them is negligible in this flight condition. The error with all normalised rotational speed $\hat{\Omega}$ is within 0.6%, and the barotropic model predicts higher values for all quantities.



(a) Cruise



(b) Hover

Figure A.3: Model error in cruise and hover

The effect of the viscosity modelling was evaluated separately for a cruise case (Fig. A.4). In both cases Sutherland's law was used, but in the reference case it is calculated with the local temperature, and in the other case the viscosity is constant in the whole flowfield, calculated from the reference static temperature and reference velocity (chosen as the cruise velocity). The error of all integral quantities is well below 0.1%, therefore this approximation is well-suited for this use case.

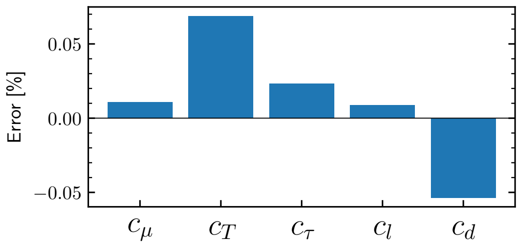


Figure A.4: Error of using Sutherland's law with a constant reference temperature instead of the temperature field ($\hat{\Omega} = 0.7$)

A.3. CONCLUSION

The setups in the present work are closer to the hover operating points evaluated in this study. The errors are deemed acceptable, therefore it is not necessary to change the fluid model or add the energy equation for propulsion simulations with moderate blade tip Mach numbers. In Chapter 8 at $\hat{\Omega} = 1/3$ the tip Mach number is considerably lower than at the lowest hover RPM in this study, therefore the errors due to the barotropic fluid model are expected to be smaller.

B
TIMINGS OF MINOR KERNELS

B.1. UPDATE FLAG

The voxelizer described in Chapter 5 and `update_sdf` introduced in Chapter 6 are the most time consuming parts of `update_flag`. On the other hand, the new algorithms necessitate some additional steps between the kernel calls (Fig. 5.1). Although in most cases these are negligible, in order to keep an accurate bookkeeping, the measurements are provided in Table B.1 together with the voxelizer and `update_sdf` runtimes. The values are the average execution time on the slowest rank, and they are also included as a percentage of the average time step.

The *run_get_stl_block_indices* timer entails the following steps:

- Memory allocation
- *run_check_bbox_overlap* timer (`check_bbox_overlap` kernel)
- Calculate the index and number of blocks.
- $\sum$ number of blocks (length of the index list)
- Fill index list

The *run_blocklist_update5* timer contains the following steps:

- *run_check_bbox_overlap_and_cut* timer (`check_bbox_overlap_and_cut` kernel)
- Calculate the index and number of blocks.
- $\sum$ number of blocks (length of the index list)
- Fill index list

Timer	Pitching airfoil	Rotating flap	Flap with resolved engine	Aircraft wind tunnel model
run_get_stl_block_indices	4.7978e-04 (1.00%)	8.6279e-04 (2.46%)	8.2792e-04 (1.64%)	2.0600e-03 (0.40%)
run_check_bbox_overlap	1.5367e-04 (0.32%)	2.1673e-04 (0.62%)	2.1819e-04 (0.43%)	1.5061e-04 (0.03%)
run_voxelize_blocks_4	4.9005e-04 (1.02%)	1.9381e-03 (5.53%)	1.4909e-03 (2.95%)	1.8412e-02 (3.59%)
run_blocklist_update5	3.7664e-04 (0.79%)	5.3872e-04 (1.54%)	2.4086e-04 (0.48%)	1.0778e-03 (0.21%)
run_check_bbox_overlap_and_cut	2.0539e-05 (0.04%)	2.8624e-05 (0.08%)	2.7497e-05 (0.05%)	4.3329e-05 (0.01%)
run_voxelize_blocks_5	4.5252e-04 (0.94%)	9.1926e-04 (2.62%)	1.2291e-03 (2.43%)	1.1995e-02 (2.34%)
run_update_sdf	3.8497e-03 (8.02%)	9.1024e-04 (2.60%)	1.4109e-02 (27.94%)	6.3866e-02 (12.47%)

Table B.1: Timings of small kernels in `update_flag`

In the rotating flap setup both `update_sdf` and the voxelizer take only a small portion of the iteration time. On the other hand the small kernels' runtime is within the same order of magnitude. This overhead explains, why unlike the other larger setups, the rotating flap showed a small slow-down of `update_flag` (7.3).

B.2. LOAD BALANCING

The runtime and AMR time of the baseline and new code was investigated in Chapter 4 and it was compared on 4 different setups in Chapter 7. The AMR time is usually dominated by the data transfer between partitions during repartitioning, but in case of setups with only small mesh changes it is relevant to look at the time it takes to run the repartitioning algorithm itself. In the new version, a parallel partitioner is used [40].

The results confirm, that this reduces the partitioning time up to 5 times (with 8 GPUs), but the gain is smaller for a larger graph size (Aircraft wind tunnel model in Table B.2). In Table B.2 the times are also shown as the percentage of average AMR times. The timer *init_metis_partition* is less than two times slower than the repartitioning during an AMR step (*ParMETIS_adaptive*). The initial partitioning entails a k-way partitioning step and up to 10 graph refinements (until the edge-cut is not improving). Spending extra time on the refinement at initialisation is well worth it, since it can improve the edge-cut up to 10% (Chapter 4).

Timer	Pitching airfoil	Rotating flap	Flap with resolved engine	Aircraft wind tunnel model
ParMETIS_adaptive	1.4811e-01 (8.54%)	9.6285e-02 (8.60%)	2.2623e-01 (14.80%)	9.3060e-01 (8.68%)
init_metis_partition	2.9231e-01 (16.86%)	1.3137e-01 (11.73%)	3.5109e-01 (22.97%)	1.6157e+00 (15.06%)
amr_metis	2.8429e-01 (16.40%)	1.1983e-01 (10.70%)	3.2938e-01 (21.55%)	1.1053e+00 (10.30%)
amr_metis (baseline)	8.6940e-01 (1.04%)	9.1528e-01 (41.11%)	1.5675e+00 (2.77%)	2.5373e+00 (0.25%)
# of GPUs	4	8	8	8
# of blocks [$\times 10^3$]	22	3.5	20	77

Table B.2: Timings of partitioning kernels

C

Point-to-triangle distance projected in 2D

C.1. Introduction

In Chapter 6, improving the point-to-triangle distance calculation between cells and body faces was identified as one of the most promising options to accelerate the SDF update. Minor tweaks of the algorithm reduced the runtime of the kernel, but `update_sdf` remained one of the bottlenecks is some setups, e.g. the pitching airfoil.

The baseline 3D point-to-triangle distance calculation [62] is an efficient way to calculate the distance in a generic case. In the SDF calculation, cells within a block (with the baseline algorithm) share the potential triangle candidates. With any other triangle filtering approach adjacent cells would likely share some triangle candidates, therefore a given triangle would be used for distance calculation many times.

The idea behind the new approach is to reduce the arithmetic operations of a distance check in exchange for the one-time cost of pre-computing triangle properties and storing them. The generic 3D problem can always be turned into a 2D problem, where the point is projected onto the plane of the triangle. The in-plane squared distance is calculated analogously to Ericson's 3D method, and the squared distance from the plane is added to it. A similar 2D approach was used by [82], but in the present work Ericson's [62] algorithm — which is highly optimised for efficient execution on GPUs — was taken as the basis.

The present algorithms differs from the algorithm used in the previous chapters (Alg. 4) mainly in the steps within the for loop of Alg. 5. In the algorithm and in Section C.2 the non-dimensionalised version ($2D_{nd}$) of is described. The $2D$ version only differs only in some details from $2D_{nd}$. The edge length is stored instead of its square, and on line 10 there is no need to scale the result of the distance function with the edge length.

Algorithm 5 Update SDF ($2D_{nd}$)

```
1:  Thread for each NSDF cell
2:  Get block of the cell
3:  Get cell center $P$
4:  Get cut triangle list of block
5:  $d^2_{min} = \infty$
6:  for $i \leftarrow 0, n_{cut\triangle,b}$ do
7:      Get $\mathbf{R}^{-1}$, $\mathbf{C}'$, $|AB|^2$
8:      $\mathbf{P}' = \mathbf{R}^{-1}\mathbf{P}$
9:      $d^2_{xy,i}$ = quick_dist_to_tri_2D($\mathbf{P}', \mathbf{C}'$)
10:     $d^2_i = (d^2_{xy,i} + (P')^2_z)|AB|^2$
11:     if $d^2_i < d^2_{min}$ then
12:         $d^2_{min} = d^2_i$
13:         $ID^* = ID_i$
14:     end if
15: end for
16: From $ID^*$ get closest $\triangle$ vertices $\mathbf{A}^*$, $\mathbf{B}^*$ and $\mathbf{C}^*$
17: $(d^2_{min}, ID^*, \mathbf{P}^*_{hit}...)$ = dist_to_tri($\mathbf{P}, \mathbf{A}^*, \mathbf{B}^*, \mathbf{C}^*$)
18: $\mathbf{n}$ = get_normal(ID, ...)
19: $SDF = \mathrm{sgn}((\mathbf{P} - \mathbf{P}_{hit}) \cdot \mathbf{n})\sqrt{d^2_{min}}$
20: if $SDF < 0$ then
21:     Set target flag
22: else
23:     Unset target flag
24: end if
```

C.2. METHODOLOGY

The unit normal vector of the triangle from the cross-product of two unit edge vectors is defined as:

$$\mathbf{n}_\triangle = \frac{\mathbf{AB}}{|AB|} \times \frac{\mathbf{AC}}{|AC|} \tag{C.1}$$

The 3×3 transformation matrix rotates the coordinates into the local triangle coordinate system and scales it, so that the AB edge is unit length. This step is explained in detail in Section C.4.

$$\mathbf{T}^{-1} = \frac{1}{|AB|}\left[\frac{\mathbf{AB}}{|AB|}, \mathbf{n}_\triangle \times \frac{\mathbf{AB}}{|AB|}, \mathbf{n}_\triangle\right]^{-1} \tag{C.2}$$

A point can be transformed into the local coordinate system the following way:

$$\mathbf{P}' = \mathbf{T}^{-1}(\mathbf{P} - \mathbf{A}) \tag{C.3}$$

The A and B points do not have to be explicitly transformed, their new coordinates are always identical. At this step it is more efficient to calculate the C point coordinates from the edge lengths, here denoted as $a = |AB|, b = |BC|, c = |AC|$ and store them. The x and y coordinates of C are calculated as the intersection of two circles of radius b and c drawn around the B and A vertices. The positive y solution is kept in accordance with the coordinate system convention. An alternative would be to use Eq. C.3. Using Eq. C.4 only 3 scalars (an edge length and C_x', C_y') need to be stored in memory per triangle.

$$\mathbf{A}' = \begin{bmatrix}0\\0\\0\end{bmatrix} \quad \mathbf{B}' = \begin{bmatrix}0\\1\\0\end{bmatrix} \quad \mathbf{C}' = \begin{bmatrix}C_x'\\C_y'\\0\end{bmatrix} = \begin{bmatrix}\frac{1}{2} + \frac{1}{2}\left(c^2 - b^2\right)\frac{1}{a^2}\\ \sqrt{\frac{c^2}{a^2} - C_x'^2}\\ 0\end{bmatrix} \tag{C.4}$$

In the following the prime symbol ($'$) marking the local triangle coordinate system will be omitted for conciseness. The in-plane distance d_{xy} is calculated differently for each Voronoi region. The calculations simplify compared to the generic 3D case, but they need an additional matrix transformations for point P. The

Voronoi region checks are ordered as follows, to start with the easier computations. The geometric meaning of d_2, d_6, d_4 and d_{43} are described in [62]. If any of the criteria are satisfied, the algorithm stops and returns the in-plane squared distance.

$$P_y \le 0 \wedge 0 \le P_x \le 1 \quad \Rightarrow \quad d_{xy}^2 = P_y^2 \quad \boxed{E_0} \tag{C.5}$$

$$d_2 = \mathbf{C} \cdot \mathbf{P} \tag{C.6}$$

$$P_x \le 0 \wedge d_2 \le 0 \quad \Rightarrow \quad d_{xy}^2 = \mathbf{P} \cdot \mathbf{P} \quad \boxed{V_0} \tag{C.7}$$

$$d_6 = \mathbf{C} \cdot \mathbf{CP} \tag{C.8}$$

$$d_6 \ge 0 \wedge CP_x \le d_6 \quad \Rightarrow \quad d_{xy}^2 = \mathbf{CP} \cdot \mathbf{CP} \quad \boxed{V_2} \tag{C.9}$$

$$d_6 < 0 \wedge d_2 \ge 0 \wedge CP_x d_2 - P_x d_6 \le 0 \Rightarrow d_{xy}^2 = \left(\frac{d_2}{d_2 - d_6}\mathbf{C} - \mathbf{P}\right) \cdot \left(\frac{d_2}{d_2 - d_6}\mathbf{C} - \mathbf{P}\right) \quad \boxed{E_2} \tag{C.10}$$

$$d_4 = \mathbf{C} \cdot \mathbf{BP} \tag{C.11}$$

$$BP_x \ge 0 \wedge d_4 \le BP_x \quad \Rightarrow \quad d_{xy}^2 = \mathbf{BP} \cdot \mathbf{BP} \quad \boxed{V_1} \tag{C.12}$$

$$d_{43} = d_4 - BP_x \tag{C.13}$$

$$d_{43} \ge 0 \wedge CP_0 \ge d6 \ge BP_x d_6 - CP_x d_4 \le 0 \Rightarrow d_{xy}^2 = \begin{bmatrix} w(C_x - 1) - P_x + 1 \\ wC_x - P_y \end{bmatrix} \cdot \begin{bmatrix} w(C_x - 1) - P_x + 1 \\ wC_x - P_y \end{bmatrix} \quad \boxed{E_1} \tag{C.14}$$

Otherwise the point projects to the face Voronoi region of the triangle:

$$d_{xy}^2 = 0 \quad \boxed{F} \tag{C.15}$$

The distance from the plane of the triangle is simply the Z coordinate of the transformed point:

$$d_z^2 = {P_z}^2 \tag{C.16}$$

The total distance squared is the sum of the distance squared from the triangle in the plane of the triangle and the distance from the plane of the triangle:

$$d^2 = d_{xy}^2 + d_z^2 \tag{C.17}$$

Fig. C.1 shows a triangle in the local coordinate system of the $2D_{nd}$ method, and 3 example points in a face (P_0), edge (P_1) and vertex (P_2) Voronoi region, and their projected distance d from the triangle. Points within the face Voronoi region F have a 0 projected distance.

C.3. PERFORMANCE

The performance of the algorithms is data dependent. Each algorithm is checking the Voronoi regions in a given (computationally efficient) order, and the distance of the point from the triangle changes the statistical distribution of hit types (Voronoi regions), considering random points and random triangles with a uniform distribution. Fig. C.2 shows the results of the distance calculations between a million random triangles and points. The triangles vertices are randomly placed in a unit cube centred around 0, and the points are placed in a cube of size r also centred around 0. The r parameter in essence captures the effect of the ratio between block and triangle sizes. When r is small, many points will project to the face region. This slows down all algorithms, since the face region is tested last.

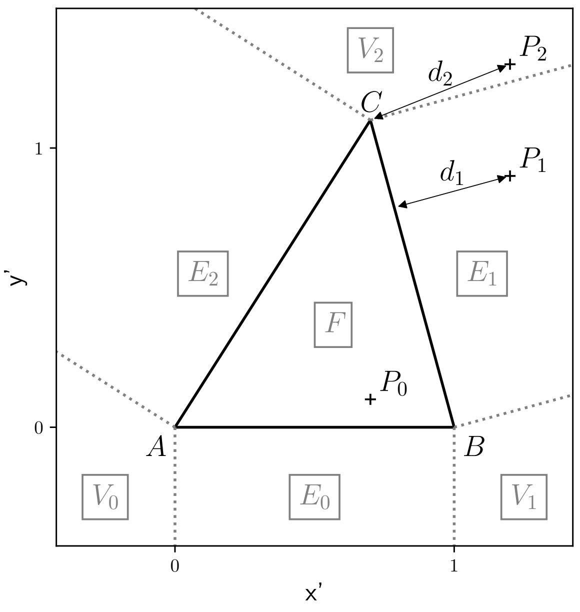
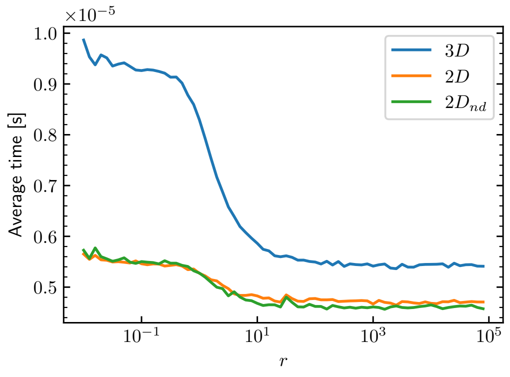
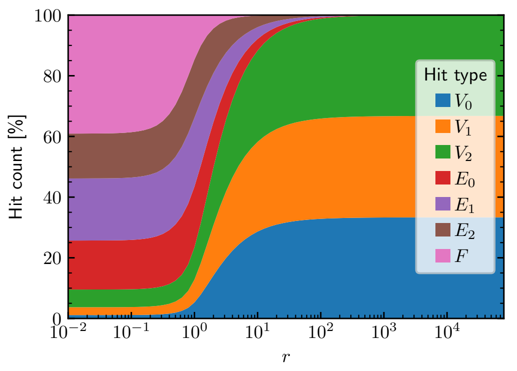


Figure C.1: Parametrisation of a triangle in 2D, showing Voronoi regions

When r is large, most points will fall into the vertex Voronoi regions (this could be illustrated by zooming out from Fig. C.1). Reaching the first two vertex checks requires just a few arithmetic operations, which explains why the run time drops more than 30% with high r. The $2D$ algorithms have similar performance, with the non-dimensional version having a slight advantage. Both are around 40% faster with small r and about 15 % faster with large r in the prototype implementation. Implementing the same $2D$ and $2D_{nd}$ algorithms in the solver resulted only in an order of magnitude smaller speed-up.



(a) Timing

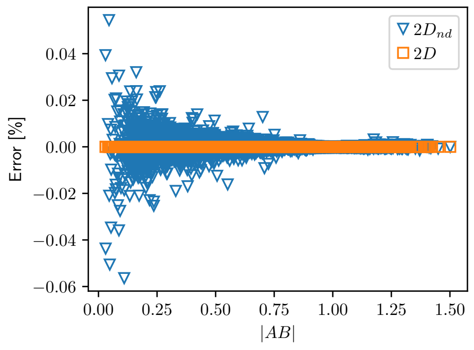


(b) Distribution of Voronoi regions

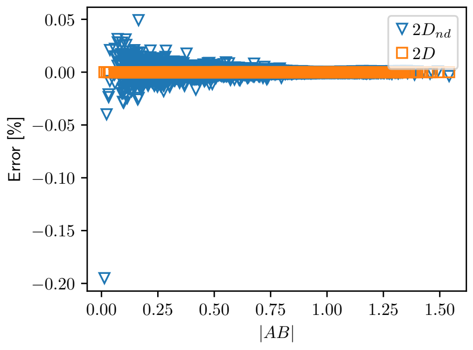
Figure C.2: Timing of the point to triangle distance algorithms implemented in Python

When implemented in the present solver, both the $2D$ and the $2D_{nd}$ showed erroneous behaviour with geometries containing triangles with extremely large aspect ratios. This behaviour was only observed, when single precision variables were used in the function. With double precision it worked as expected, but it diminishes the speed-up of the new algorithms.

Fig. C.3 shows the errors of the squared distance with single and double precision implemented in Python. The single precision implementation has a larger error, but it is still not expected to result in large errors in the closest triangle selection within the SDF update. The error is larger towards smaller edge lengths $|AB|$ in the $2D_{nd}$ algorithm. This is expected, since this is used for the scaling of the coordinate system (Eq. C.23). Using the longest edge for normalisation would provide additional numerical robustness, but the more complex data structure and logic would also slow it down.



(a) Double precision



(b) Single precision

Figure C.3: Relative error of distance compared to the 3D algorithm

The $2D$ point-to-triangle distance algorithms were promising during the prototype implementation in Python, but in the present solver they proved to be less robust and provided less speed-up. Although the robustness issues came up only with highly skewed triangles, it was decided not to include the $2D$ algorithm in the studies carried out in Chapter 7–8.

C.4. INVERSE MATRIX CALCULATION

A computationally efficient 3×3 matrix inversion method is the following:

$$\mathbf{R}^{-1} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}^{-1} = \frac{1}{\det(\mathbf{R})} \begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix}^{T} = \frac{1}{\det(\mathbf{R})} \begin{bmatrix} A & D & G \\ B & E & H \\ C & F & I \end{bmatrix} \tag{C.18}$$

In Eq. C.18 capital letters are calculated as a 2D cross-product of the original matrix, excluding the row and column of the same lowercase letter (see Eq. C.21). In the triangle rotation matrix (Eq. C.18) one vector is constructed as the cross-product of the two other vectors. Because the original matrix and the inverse include the same cross-products, the amount of computation can be reduced by using this identity for this special case.

$$\frac{\mathbf{AB}}{|\mathbf{AB}|} = \begin{bmatrix} a \\ d \\ g \end{bmatrix} \quad \mathbf{n}_{\triangle} = \begin{bmatrix} c \\ f \\ i \end{bmatrix} \tag{C.19}$$

The elements of the second column are calculated with a cross-product (Eq. C.2).

$$b = fg - id \quad e = ia - cg \quad h = cd - fa \tag{C.20}$$

The elements from the inverse matrix needed for the determinant are precomputed ($b = B$, and it is already known).

$$A = ei - fh \quad C = dh - eg \tag{C.21}$$

The determinant with Sarrus' rule is defined as:

$$\det(\mathbf{R}) = aA + bB + cC = aA + b^2 + cC \tag{C.22}$$

Eq. C.20 is substituted into Eq. C.18, simplifying the second row.

$$\mathbf{R}^{-1} = \frac{1}{\det(\mathbf{R})} \begin{bmatrix} A & (-bi + ch) & (bf - ce) \\ b & e & h \\ C & (-ah + bg) & (ae - bd) \end{bmatrix} \tag{C.23}$$

Lastly the rotation matrix is scaled with the edge length to obtain the final transformation matrix (only in the $2D_{nd}$ algorithm):

$$\mathbf{T}^{-1} = \frac{1}{|AB|}\mathbf{R}^{-1} \tag{C.24}$$

D
SDF UPDATE BASED ON TRIANGLE NEIGHBOURS

D.1. Introduction

After the changes in AMR load balancing and voxelizer improvements, currently `update_sdf` is the most expensive part of the solver (see Chapter 8). In Chapter 6, speeding up the point-to-triangle distance calculation and reducing the triangle list were identified as the biggest levers to improve `update_sdf`. The point-to-triangle distance is further discussed in Appendix C. In the present appendix the potential to reduce the triangle list is investigated. A new algorithm, that is relying on identifying the neighbours of each triangle of the surface mesh in a wall cell size dependent search radius is introduced. Results are shown with simple test cases to highlight the scaling of the present and baseline algorithm. Finally the advantages and shortcomings of the new algorithm are discussed.

D.2. Methodology

As in the rest of the solver, it is assumed that faces of the surface are disjoint, therefore the triangles do not need to be tested for intersection. At least one of the closest points of any two triangles lies on the boundary of one of the triangles [62]. The two possible cases are a pair of edges from each triangle or the vertex of one triangle is the closest to the face of the other triangle (Fig. D.1).

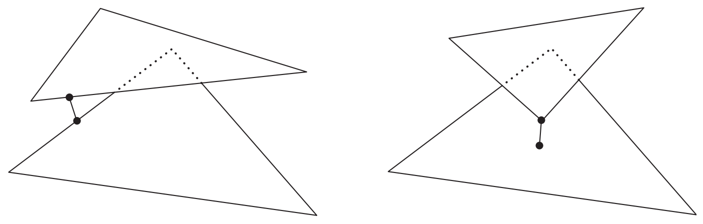

(a) (b)

Figure D.1: The closest pair of points between two triangles *a)* lying on an edge from either triangle or *b)* a vertex of one triangle and an interior point of the other triangle [62]

There are 4 possible cases for the closest point between two line segments Fig. D.2. The segment-to-segment distance calculation from [62] is adapted.

The search radius d_e is used to construct the neighbour list of each triangle (Alg. 6). Currently the algorithm is executed for all triangles on all GPUs. Parallelising the task could significantly shorten the runtime of this kernel. The simplest approach would be to just launch the kernel with $n_{\triangle}/n_{GPU}$ threads on each GPU. This would reduce the workload but decrease the occupancy, and except for very large surface meshes would not improve the execution time. A more refined approach would be to use the voxelization (Chapter 5), and only look for triangle neighbours in an (enlarged) block which owns $\triangle_i$. Due to the enlargement a triangle can be included in more than one block, and the enlargement ensures that picking any of them includes a sufficiently large neighbourhood of the triangle. Since this kernel is called only once, and even for the large

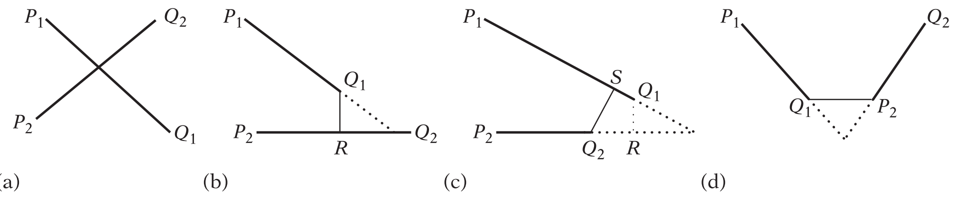


Figure D.2: Closest points *a)* inside both segments, *b)* and *c)* inside one segment endpoint of other, *d)* endpoints of both segments [83]

aircraft wind tunnel model example it only took 3 seconds to run Alg. 6, it was decided not to pursue this option.

Algorithm 6 Find triangle neighbours

```
1:  Thread for each IB type $\triangle_i$
2:  Get body of the $\triangle_i$
3:  Get wall cell size of the body
4:  Calc. enlargement
5:  Get $\triangle_i$ vertices $\mathbf{A}_i$, $\mathbf{B}_i$ and $\mathbf{C}_i$
6:  Calc. AABB of $\triangle_i$
7:  for $j \leftarrow 0, n_{\triangle,IB}$ do
8:      Get $\triangle_j$ vertices $\mathbf{A}_j$, $\mathbf{B}_j$ and $\mathbf{C}_j$
9:      Calc. AABB of $\triangle_j$
10:     AABB overlap check of $\triangle_i$ and $\triangle_j$
11:     Triangle distance check $\triangle_i$, $\triangle_j$
12:     $n_i$ += 1 (Call 1)
13:     $n$ += 1 (Call 1)
14:     Save $ID_j$ to $\triangle_i$ list (Call 2)
15: end for
```

Fig. D.3 illustrates an example triangle $\triangle_0$ and several other triangles in its neighbourhood. For simplicity, all triangles are coplanar, but the distance calculation works with arbitrary 3D orientations. Triangles sharing a vertex or edge ($\triangle_1$) are always included in the neighbour list, since $d_e = 0$, and the triangle list is also self-referencing the triangle itself. Triangle $\triangle_2$ is closer than $d_e = 2 \cdot WCS$ and is included in the list, whereas $\triangle_3$ is excluded. With a larger WCS triangle $\triangle_3$ would be also included.

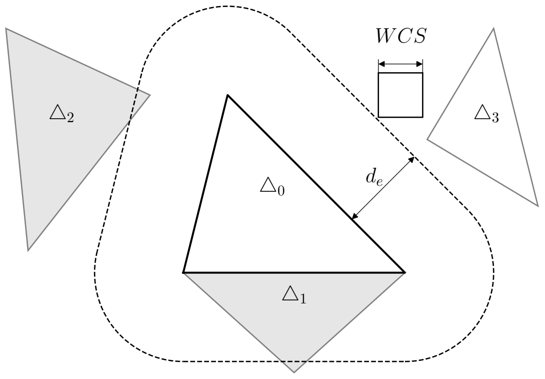


Figure D.3: Zone for collecting triangle neighbours for $\triangle_0$ in simplified a 2D surface mesh

For each triangle pair first the axis-aligned bounding boxes (AABB) are calculated. This check for triangles requires few arithmetic operations, and most triangle pairs will not pass this test. The AABB test is similar to the overlap tests in Chapter 5. After this the 3×3 edge pairs are checked and the 3 point and triangle pairs (maximum 15 tests). If any of these checks is a match (the distance is less than the enlargement), the algorithm will skip the remaining checks. It is only important to know, that the triangle pair distance is less than the enlargement, the exact value of the distance or finding the closest points on the triangles, is not relevant for this application. The enlargement has to be at least $\sqrt{3} \cdot WCS$ (the body diagonal of a cell on the surface), assuming that the CFL number stays below one (hence the mesh can move only 1 cell length in one time step). In the following studies a value of $d_e = 2 \cdot WCS > \sqrt{3} \cdot WCS$ was used for the enlargement, to add a safety factor for numerical errors.

The algorithm does two passes, in a similar manner as the voxelizer (Call 4 and 5). In the first pass it is counting the neighbours of each triangle (n_i) and the total number of neighbours (n). A size array (element i is n_i) and index array (element i is $\sum_0^i n_j$) is constructed, and in the next pass the elements of a continuous n-sized array are filled. In the worst case the array size can be n^2, which would occur, if the geometry size is on the order of the size of a single cell. This is unrealistic for any use case, because it would not resolve any geometric details. But for large geometries (e.g. a full aircraft) with a coarse mesh storing the triangle connectivity can lead to excessive memory usage.

In the initial time step and after each ep still the baseline `update_sdf` algorithm (Alg. 4) is used, since the previous triangle ID field is not available. The new algorithm is presented in Alg. 7.

Algorithm 7 Update SDF (new)

```
1:  Thread for each NSDF cell
2:  Get block of the cell
3:  Get cell center $\mathbf{P}$
4:  Get previous $\triangle ID_{t-1}$ from the neighbour cells
5:  $d^2_{min} = \infty$
6:  for $i \leftarrow 0, n_{\triangle ID,t-1}$ do
7:      Get $\triangle$ vertices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$
8:      $d_i^2 = \text{quick\_dist\_to\_tri}(\mathbf{P},\mathbf{A},\mathbf{B},\mathbf{C})$
9:      if $d_i^2 < d^2_{min}$ then
10:         $d^2_{min} = d_i^2$
11:         $ID^* = ID_i$
12:     end if
13: end for
14: From $ID^*$ get closest $\triangle$ vertices $\mathbf{A}^*$, $\mathbf{B}^*$ and $\mathbf{C}^*$
15: $(d^2_{min}, ID^*, \mathbf{P}^*_{hit}...) = \text{dist\_to\_tri}(\mathbf{P},\mathbf{A}^*,\mathbf{B}^*,\mathbf{C}^*)$
16: $\mathbf{n} = \text{get\_normal}(\text{ID}, ...)$
17: $SDF = \text{sgn}\left(\left(\mathbf{P} - \mathbf{P}^*_{hit}\right) \cdot \mathbf{n}\right) \sqrt{d^2_{min}}$
18: if $SDF < 0$ then
19:     Set target flag
20: else
21:     Unset target flag
22: end if
```

The previous triangle ID for a given cell is picked from the cell itself, if it had an assigned triangle ID in the previous time step. If the triangle ID is not assigned, the face neighbours of the cell are iterated. Because an opportunity cell (a cell that might become a ghost cell in the next time step) has at least one face neighbour ghost cell, at least one element of this stencil yields a triangle ID. In case if cells on block boundaries with different refinement levels, the following rules are applied:

Coarse to fine If the cell in the buffer layer on the coarse side has a triangle ID, it is mapped to the 8 smaller blocks on the fine side, if they do not have a triangle ID.

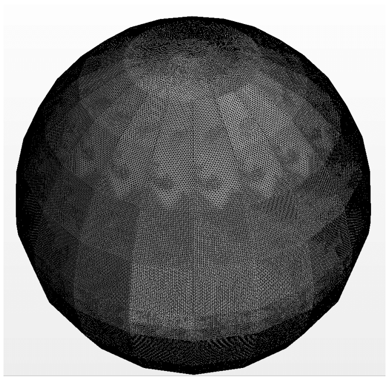
Fine to coarse The blocks on the fine side are iterated, and the first triangle ID hit is assigned to the coarse side, if it does not have a triangle ID.

The triangle IDs are deallocate, if a cell ceases to be an opportunity cell. This is especially important for periodic motion, or when two opposite walls of a body are passing through the same region of the domain. Otherwise the closest triangle could be searched on the wrong triangle list, leading to an erroneous SDF and wrong tagging of solid, fluid and ghost cells.

After obtaining the previous triangle ID with the above rules, the neighbour triangle list of this triangle is iterated to find the closest triangle to the cell, in an analogous fashion to the Block-based algorithm (Alg. 4).

D.3. Results

It was previously identified, that the fineness of the volume mesh compared to the surface mesh has a large impact on the performance of the SDF algorithm. A test case is constructed, where the target triangle size / wall cell size ratio (△/□) is varied by changing the surface mesh refinement. A sphere is placed in the center of the cubic domain. The sphere is first discretised with the coarsest target triangle size, and for the more refined cases this polyhedron (Fig. D.4a) is used to keep the volume mesh exactly the same. Using a different discretised sphere could slightly change the number of cut cells, since the surface area of the body decreases as a polyhedron with less faces is constructed from point on the exact sphere surface.


(a) Uniform surface mesh

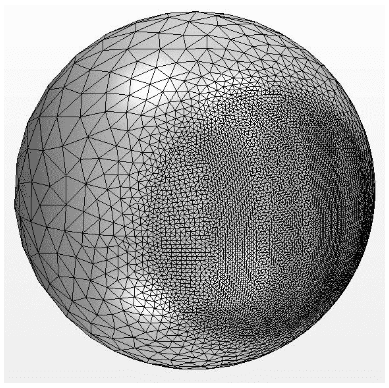

(b) Sphere with refinement

Figure D.4: Uniform and on one side refined surface triangulation of a sphere

The other test case is also a sphere, but with 10% of the body along one axis refined (Fig. D.4b). This test can demonstrate how the algorithms are performing, when △/□ is uneven.

Two versions of `update_sdf` are compared, the one introduced in Chapter 6 will be referred as the Block-based (BB) algorithm, and the new approach introduced above as Triangle-based (TB). The BB contains the improvements from Chapter 6. The baseline BB algorithm is expected to scale similarly, but with higher runtimes in general, because of the longer point-to-triangle distance calculation in each iteration of the block loop.

BB scales linearly with △/□ on a uniform mesh (Fig. D.5). At small △/□ the TB is more than 3 times faster. On the mesh with the refinement the BB scales with density of triangles in the refined area. For this case the gain with TB is a more than 8× speed-up.

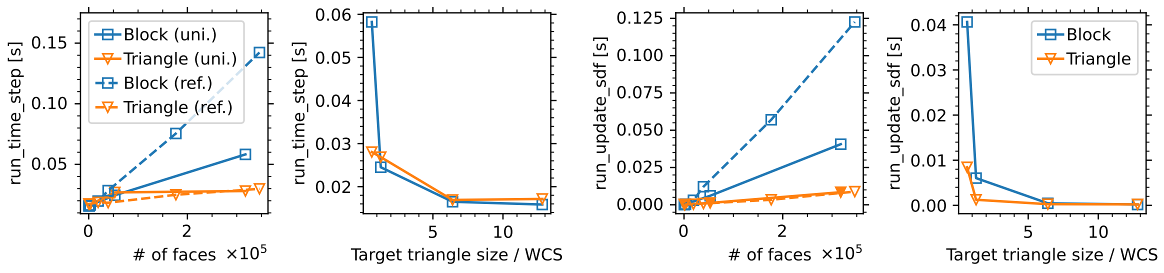


Figure D.5: Timings of a uniform and refined spherical surface mesh with block and triangle SDF update

Although TB was very promising during the initial testing, with more complex geometries a few bottlenecks and robustness issues were identified. In the pitching airfoil setup presented in Section 7.2 TB is very unbalanced and it is slower on one of the partitions than BB. The slowest partition contains the TE of the airfoil, which has a very fine surface mesh on the lower and upper surface close to the edges (Fig. D.7). The low △/□ in this area is further exacerbated by the high aspect ratio of the triangles. In the spanwise direction only two layers of triangles are included in the neighbour list, but in the chordwise (x) and thickness (z) directions many triangles are included (some close to the TE too thin to be drawn in Fig. D.7).

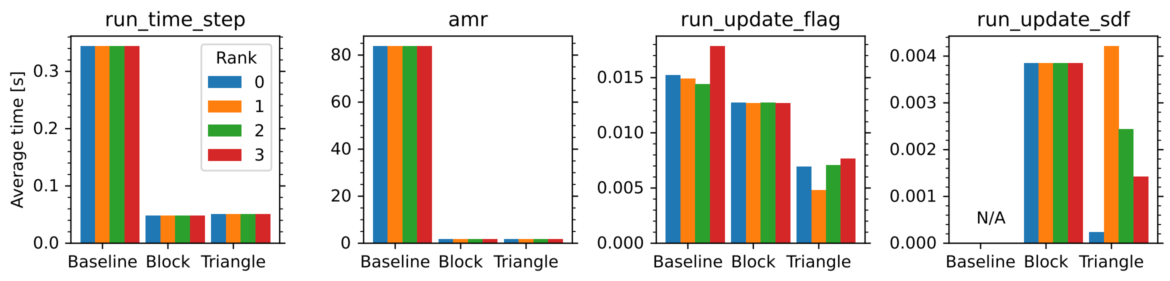


Figure D.6: Timings of the pitching airfoil setup with the baseline code, block-based SDF update and triangle neighbour-based SDF update

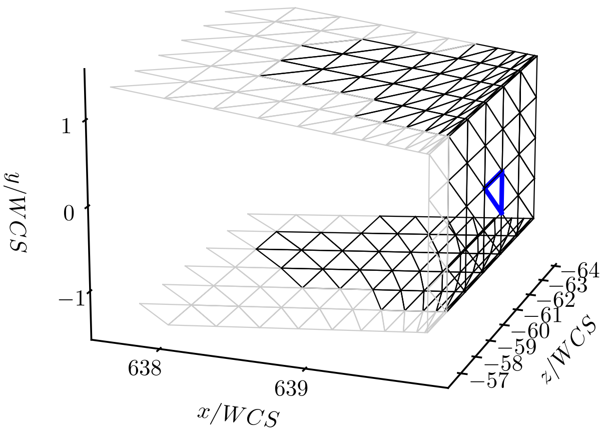


Figure D.7: Triangle connectivity on a coarse volume mesh, close to the trailing edge

In other use cases it was observed, that with coarse volume mesh resolution on thin geometries (like a TE or cone tip) the triangle IDs were incorrectly assigned. This can cause severe errors in the solution, when the wall cell size approaches the thickness of the geometry. For such thin geometries a different number of cells can be marked as solid cells in the thickness direction in consecutive time steps (e.g. 1, 0, 1, ...). Some of the triangle neighbour information is lost, and the algorithm selects the closest triangle from an incomplete list of neighbours, which can cause an erroneous SDF value. For certain shapes, such as knife edges and pointy tips, a volume mesh refinement does not remedy this issue.

D.4. DISCUSSION

Despite the good performance of TB on surface meshes with simple shapes the algorithm shows insufficient robustness on complex geometries and similar bottlenecks to BB. Since `update_sdf` remains the main bottleneck for moving cases, it is proposed to investigate alternative approaches. The above studies showed, that the $\triangle/\square$ is the critical parameter for the current architecture (using threads per NSDF cells to check each triangle candidate) of the `update_sdf` kernel. A possible approach would be to use a different voxelization for storing the block triangle lists than the discretisation of the fluid domain. The simplest approach would be to just subdivide the fluid blocks n times, until the triangle lists are sufficiently small. A more flexible approach would be to subdivide locally until a computationally efficient $\triangle/\square$ is reached. Both of these would need additional memory allocation, and would need extensive profiling with different surface meshes. A simple experiment with the pitching airfoil setup showed a 1.8× speed-up of `update_sdf`, when blocks with 16^3 cells were used in stead of 32^3 cells.

E
Convergence plots

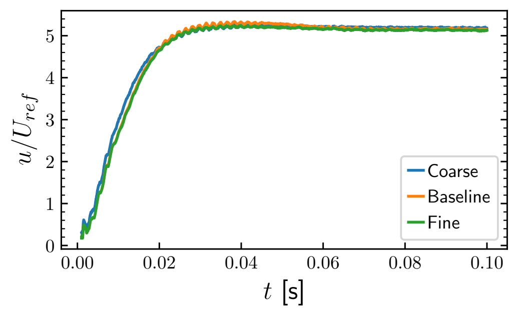
The convergence plots of the mesh convergence study (8.3) are included in E.1, and the plots belonging to the simulations presented in 8.5 can be found in E.2.

E.1. Aircraft wind tunnel model mesh convergence study with 1 engine

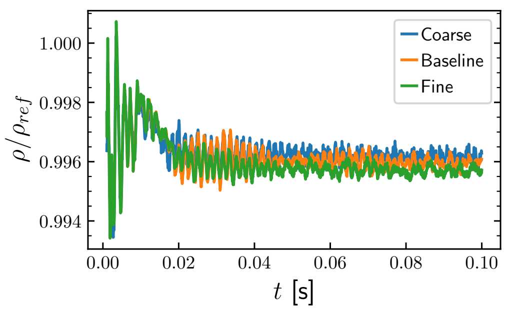
The convergence plots of the simulation in 7.6 and 8.3 are shown in Fig. E.1. Forces are normalised with the dynamic pressure of the freestream flow and the wing area:

$$c_x = \frac{F_x}{\frac{1}{2}\rho_{ref} U_{ref}^2} \tag{E.1}$$

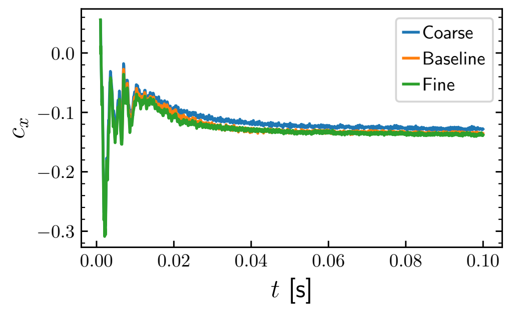
The last 6 milliseconds were used for time averaged quantities.



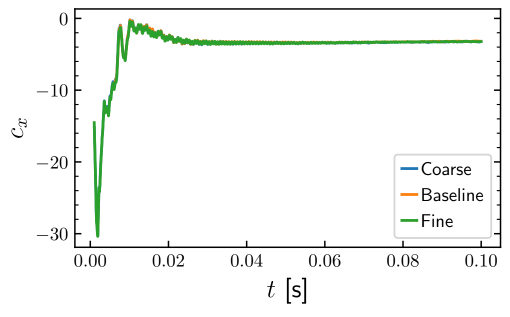
(a) Stage 2 axial velocity



(b) Stage 2 density



(c) Axial force coefficient on the resolved flap (Flap 5)

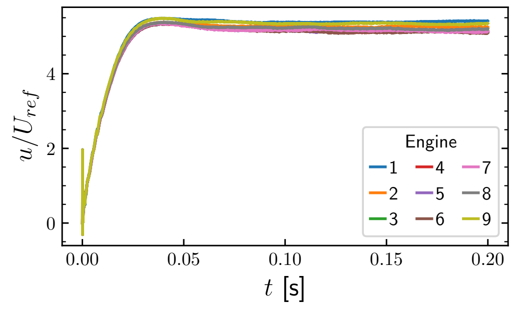


(d) Aircraft axial (vertical) force coefficient

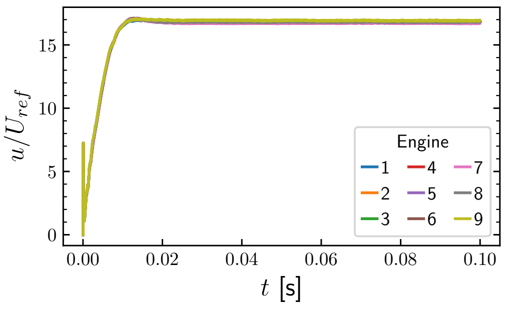
Figure E.1: Convergence on 3 different meshes

E.2. AIRCRAFT WIND TUNNEL MODEL WITH 9 RESOLVED ENGINES

The convergence plots of the simulations from 8.4–8.5 are shown in Fig. E.2–E.5. The low RPM results are shown on the left, and the high RPM results on the right of the plots.

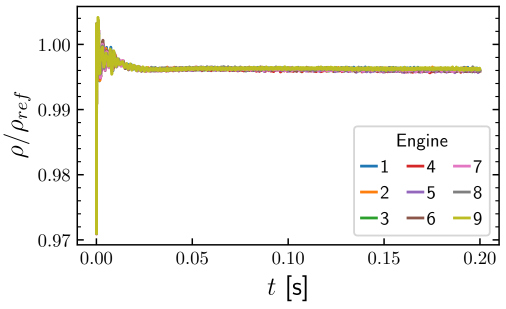


(a) $\hat{\Omega} = 1/3$

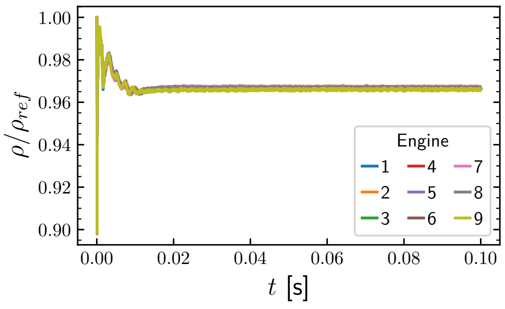


(b) $\hat{\Omega} = 1$

Figure E.2: Stage 2 axial velocity

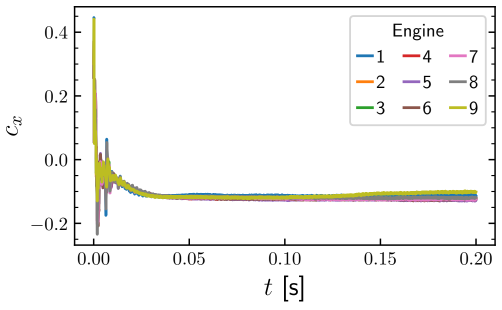


(a) $\hat{\Omega} = 1/3$

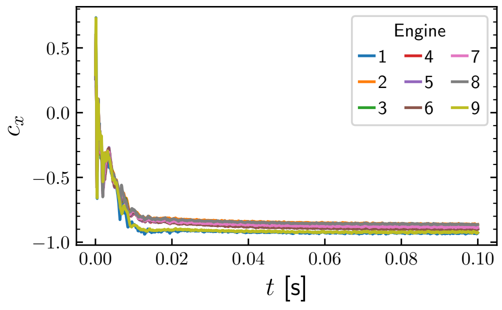


(b) $\hat{\Omega} = 1$

Figure E.3: Stage 2 density



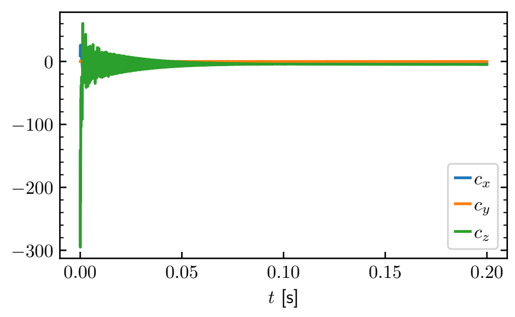
(a) $\hat{\Omega} = 1/3$



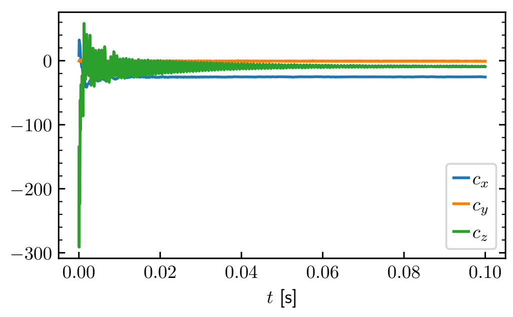
(b) $\hat{\Omega} = 1$

Figure E.4: Flap axial force coefficient

In the low $\hat{\Omega} = 1/3$ case the last 6 ms, in the $\hat{\Omega} = 1$ case the last 2 ms were used for averaging integral quantities and the flowfield.



(a) $\hat{\Omega} = 1/3$



(b) $\hat{\Omega} = 1$

Figure E.5: Aircraft force coefficients

F
Pressure tap corrections

Chapter 8 compared the surface pressure distribution in 5 spanwise locations with the wind tunnel measurement results. Several of the 150 tap show faulty values. They were either omitted or corrected in Chapter 8, and this appendix aims to describe the reasoning and methodology behind this.

The numbering of the pressure taps is shown in Fig. F.1. Note that numbering has a clockwise direction, and starts and ends at the trailing edge, unlike the surface parameter used in Chapter 8, which goes counter-clockwise and starts and ends at the fan face. Tap 3 was not part of the dataset for any of the sections.

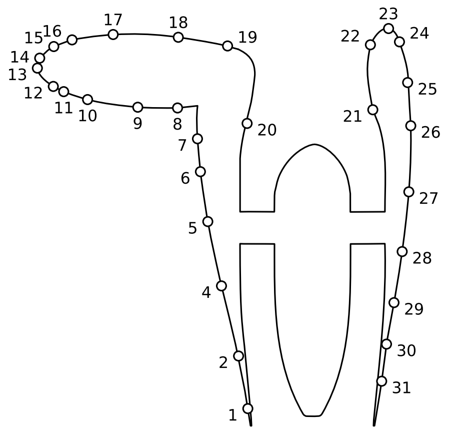


Figure F.1: Pressure tap numbering of flaps and wing sections

All taps that had an exactly 0 Pa value in both $\hat{\Omega} = 1/3$ and $\hat{\Omega} = 1$ operating points are excluded. These taps were presumably disconnected from the data acquisition system or had a severe leakage (e.g. taps 22–26 on flap 7 and tap 21 on flap 5 and 9). Furthermore, on flaps 7 and 9 all taps located on the flap surface (but not the ones on the wing surface of the same section) had a constant bias compared to other flaps. On flap 7 tap 9 had a bias of approximately $50 \cdot c_p$. This tap is located on the wing bottom, where the pressure is expected to be close to atmospheric in tailwind. A correction of $c_p := c_p + \Delta c_p$ was added (Table F.1), where Δc_p was estimated from the values on the flap and wing bottom, which are expected to be close to 0 and have only a small variation on the other flaps.

Flap	Taps	Δc_p
7	1-7, 21-31	20
9	1-7, 21-31	25

Table F.1: Correction of flap surface pressure taps at $\hat{\Omega} = 1/3$

The correction is only applied to taps 1–7 on the flap bottom and taps 21–31 on the flap top. The values on the wing remain unchanged. The correction was only applied at $\hat{\Omega} = 1/3$. A similar error might be present at $\hat{\Omega} = 1$, but due to the higher c_p magnitudes does not distort the results as much. The original and corrected pressure readings are summarised in Fig. F.2-F.3. The effect of the Δc_p correction is visible in Fig. F.2. After the correction, flap 7 and 9 closely match the pressure distribution of the other flaps, and show the physically expected behaviour (close to atmospheric pressure in the low speed regions, such as the flap bottom).

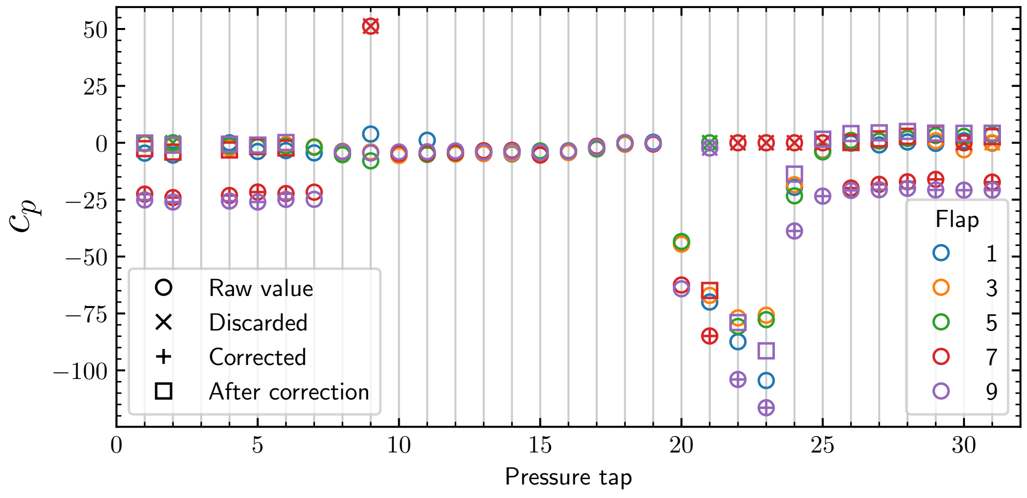


Figure F.2: Raw and corrected pressure tap readings ($\hat{\Omega} = 1/3$)

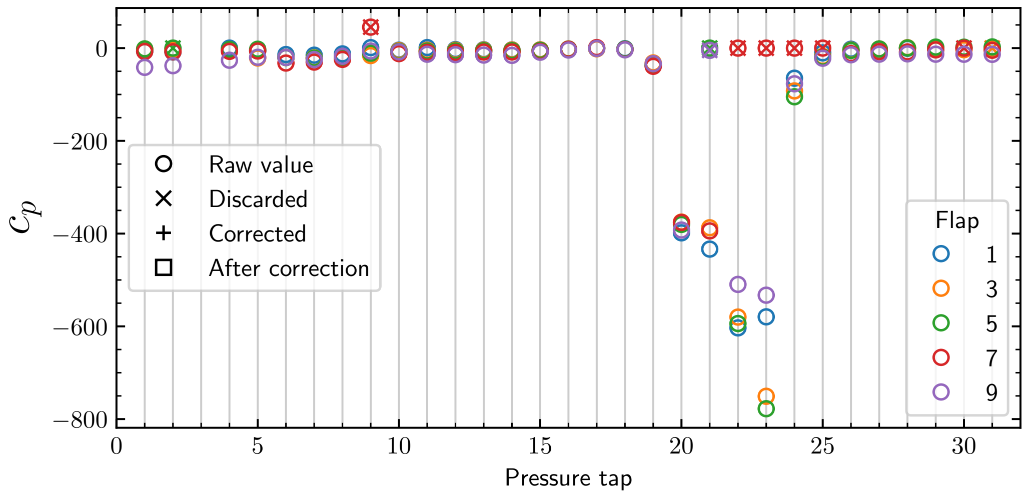


Figure F.3: Raw and corrected pressure tap readings ($\hat{\Omega} = 1$)

Bibliography

[1] P. Nathen, *Architectural performance assessment of an electric vertical take-off and landing (e-VTOL) aircraft based on a ducted vectored thrust concept,* (2021).

[2] V. Pasquariello, Y. Bunk, S. Eberhardt, P.-H. Huang, J. Matheis, M. Ugolotti, and S. Hickel, *GPU-accelerated simulations for eVTOL aerodynamic analysis,* in *AIAA SCITECH Forum: Applied Computational Fluid Dynamics V* (American Institute of Aeronautics and Astronautics (AIAA), 2023).

[3] K. A. Goc, O. Lehmkuhl, G. I. Park, S. T. Bose, and P. Moin, *Large eddy simulation of aircraft at affordable cost: a milestone in computational fluid dynamics,* Flow **1** (2021), 10.1017/flo.2021.17.

[4] J. Slotnick, A. Khodadoust, J. Alonso, W. Gropp, and D. Mavriplis, *CFD Vision 2030 Study: A Path to Revolutionary Computational Aerosciences*, Tech. Rep. (NASA Langley Research Center, 2014).

[5] A. W. Cary, J. R. Chawner, E. P. Duque, W. Gropp, B. Kleb, R. Kolonay, E. Nielsen, and B. Smith, *CFD vision 2030 roadmap: Progress and perspectives,* in *AIAA Aviation and Aeronautics Forum and Exposition, AIAA AVIATION Forum 2021* (American Institute of Aeronautics and Astronautics Inc, AIAA, 2021).

[6] L. Wang, C. Rumsey, W. K. Anderson, E. J. Nielsen, P. S. Iyer, A. Wick, A. Walden, G. Nastac, M. W. Lohry, and B. Diskin, *WMLES for the fifth high-lift prediction workshop cases using FUN3D,* in *AIAA SCITECH Forum: High-Lift-Prediction-Workshop (HLPW-5)* (2025) pp. 1–40.

[7] E. Sozer, M. F. Barad, J. A. Housman, F. Cadieux, V. C. B. Sousa, E. Dumlupinar, K. Saurabh, K. Ravikumar, and J. Duensing, *LAVA WMLES results for the 5th high-lift prediction workshop,* in *AIAA SCITECH Forum: High-Lift-Prediction-Workshop (HLPW-5): Scale-Resolving Simulations I* (2025) pp. 1–25.

[8] P. A. G. Ciloni, M. R. N. Lopez, V. G. E. Abicalil, J. L. P. Costa, R. M. Granzoto, E. S. Molina, and L. C. Scalabrin, *HLPW5-aerodynamic evaluation of different high lift configurations through CFD RANS, DDES and WMLES,* in *AIAA SCITECH Forum: High-Lift-Prediction-Workshop (HLPW-5): Scale-Resolving Simulations II* (2025) pp. 1–43.

[9] A. Hantla and Z. J. Wang, *High-order wall-modeled large eddy simulation of HLPW-5 benchmarks,* in *AIAA SCITECH 2025 Forum: High-Lift-Prediction-Workshop (HLPW-5)* (2025) pp. 1–20.

[10] H. Asada and S. Kawai, *LES of full aircraft configuration using non-dissipative KEEP scheme with conservative explicit filter,* in *AIAA Science and Technology Forum and Exposition, AIAA SciTech Forum 2022* (American Institute of Aeronautics and Astronautics Inc, AIAA, 2022).

[11] J. B. Angel, A. S. Ghate, G. K. W. Kenway, M. L. Wong, and C. C. Kiris, *Validation of immersed boundary wall-modelled large eddy simulations with F-16XL flight data,* in *AIAA AVIATION FORUM AND ASCEND: Scale-Resolving Simulation Including DNS/LES/Hybrid Methods II* (2024).

[12] Y. Ling, S. Costa, G. Arranz, M. Mani, A. Lozano-Durán, and K. A. Goc, *Building-block-flow model for large-eddy simulation: GPU implementation and application to the CRM-HL,* in *AIAA Aviation Forum and ASCEND, 2024* (American Institute of Aeronautics and Astronautics Inc, AIAA, 2024).

[13] C. W. Jackson, D. Appelhans, J. M. Derlaga, and P. G. Buning, *GPU implementation of the OVERFLOW CFD code,* in *AIAA SciTech Forum and Exposition, 2024* (American Institute of Aeronautics and Astronautics Inc, AIAA, 2024) pp. 1–18.

[14] R. Agrawal, S. T. Bose, and P. Moin, *A non-equilibrium wall model: application to the qinetiq high-lift aircraft,* in *AIAA Science and Technology Forum and Exposition, AIAA SciTech Forum 2025* (American Institute of Aeronautics and Astronautics Inc, AIAA, 2025).

[15] G. A. Brés, K. Wang, M. Emory, C. B. Ivey, and S. T. Bose, *GPU-accelerated large-eddy simulations of the NASA fan noise source diagnostic test benchmark,* in *AIAA Aviation and Aeronautics Forum and Exposition, AIAA AVIATION Forum 2023* (American Institute of Aeronautics and Astronautics Inc, AIAA, 2023) pp. 1–18.

[16] G. A. Brés, K. Wang, C. B. Ivey, S. T. Bose, G. Okubo, T. Kamatsuchi, and H. Tanaka, *Predictions of fan noise and performance at transonic operating conditions using GPU-accelerated large-eddy simulations,* in *30th AIAA/CEAS Aeroacoustics Conference, 2024* (American Institute of Aeronautics and Astronautics Inc, AIAA, 2024) pp. 1–18.

[17] H. A. Abid, A. P. Markesteijn, I. Solntsev, and S. A. Karabasov, *Numerical investigation of propeller boundary layer ingestion noise using CABARET on rotating meshes,* in *30th AIAA/CEAS Aeroacoustics Conference, 2024* (American Institute of Aeronautics and Astronautics Inc, AIAA, 2024) pp. 1–11.

[18] C. Brehm, M. F. Barad, and C. C. Kiris, *Development of immersed boundary computational aeroacoustic prediction capabilities for open-rotor noise,* Journal of Computational Physics **388**, 690 (2019).

[19] D. Zuber, J. Larsson, C. Brehm, J. Mcquaid, W. V. Noordt, B.-Y. Min, and B. E. Wake, *Wall-modeled large eddy simulation using the immersed boundary method of the HVAB rotor,* in *AIAA SciTech Forum* (2025).

[20] Z. Duan and Z. J. Wang, *High-order wall-modeled large eddy simulation of the HVAB rotor from the hover prediction workshop,* in *AIAA Science and Technology Forum and Exposition, AIAA SciTech Forum 2025* (American Institute of Aeronautics and Astronautics Inc, AIAA, 2025).

[21] J. L. Felder, H. D. Kim, and G. V. Brown, *Turboelectric distributed propulsion engine cycle analysis for hybrid-wing-body aircraft,* in *47th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition* (2009).

[22] P. Schmollgruber, C. Döll, J. Hermetz, R. Liaboeuf, and M. Ridel, *Multidisciplinary exploration of DRAGON: an ONERA hybrid electric distributed propulsion concept,* in *AIAA Scitech Forum* (2019) pp. 1–23.

[23] R. de Vries and R. Vos, *Aerodynamic performance benefits of over-the-wing distributed propulsion for hybrid-electric transport aircraft,* Journal of Aircraft **60**, 1201 (2023).

[24] H. D. Kim, *Distributed propulsion vehicles,* in *27th international congress of the aeronautical sciences* (ICAS, 2010).

[25] H. D. Kim, A. T. Perry, and P. J. Ansell, *A review of distributed electric propulsion concepts for air vehicle technology,* in *2018 AIAA/IEEE Electric Aircraft Technologies Symposium* (American Institute of Aeronautics and Astronautics, 2018) pp. 1–21.

[26] A. T. Perry, *The effects of aero-propulsive coupling on aircraft with distributed propulsion systems,* Ph.D. thesis, University of Illinois at Urbana-Champaign (2020).

[27] J. Hermetz, M. Ridel, and C. Doll, *Distributed electric propulsion for small business aircraft a concept-plane for key-technologies investigations,* in *ICAS* (2016) pp. 1–11.

[28] A. T. Perry, P. J. Ansell, and M. F. Kerho, *Aero-propulsive and propulsor cross-coupling effects on a distributed propulsion system,* Journal of Aircraft **55**, 1 (2018).

[29] K. Pieper, A. Perry, P. Ansell, and T. Bretl, *Design and development of a dynamically, scaled distributed electric propulsion aircraft testbed,* in *2018 AIAA/IEEE Electric Aircraft Technologies Symposium, EATS 2018* (Institute of Electrical and Electronics Engineers Inc., 2018).

[30] R. W. Powers and C. P. Fernandes, *Coupled fan motor assembly and wind tunnel circuit predictions using WMLES,* in *AIAA SCITECH Forum: CFD on Large-Scale Meshes for Applied Aerodynamics* (2025) pp. 1–16.

[31] G. G. Stokes, *On the theories of the internal friction of fluids in motion, and of the equilibrium and motion of elastic solids.* Mathematical and Physical Paper (1845).

[32] A. W. Vreman, *An eddy-viscosity subgrid-scale model for turbulent shear flow: Algebraic theory and applications,* Physics of Fluids **16**, 3670 (2004).

[33] S. Gordon, *Thermodynamic and Transport Combustion Properties of Hydrocarbons With Air, I-Properties in SI Units*, Tech. Rep. (NASA Lewis Research Center, 1982).

[34] B. J. Mcbride, M. J. Zehe, and S. Gordon, *NASA Glenn Coefficients for Calculating Thermodynamic Properties of Individual Species*, Tech. Rep. (NASA/TP-2002-211556, 2002).

[35] W. Sutherland, *LII. the viscosity of gases and molecular force,* The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science **36**, 507 (1893).

[36] R. Mittal, H. Dong, M. Bozkurttas, F. M. Najjar, A. Vargas, and A. von Loebbecke, *A versatile sharp interface immersed boundary method for incompressible flows with complex boundaries,* Journal of Computational Physics **227**, 4825 (2008).

[37] M. Vanella and E. Balaras, *A moving-least-squares reconstruction for embedded-boundary formulations,* Journal of Computational Physics **228**, 6617 (2009).

[38] C. Burstedde, L. C. Wilcox, and O. Ghattas, *P4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees,* SIAM Journal on Scientific Computing **33**, 1103 (2011).

[39] G. Karypis and V. Kumar, *METIS—A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes and Computing Fill-Reducing Ordering of Sparse Matrices*, Tech. Rep. (University of Minnesota, Department of Computer Science / Army HPC Research Center, 1997).

[40] G. Karypis and K. Schloegel, *PARMETIS - Parallel Graph Partitioning and Sparse Matrix Ordering Library Version 4.0*, Tech. Rep. (University of Minnesota, Department of Computer Science and Engineering, 2013).

[41] Y. Bunk, *Towards high-fidelity aero-servo-elasticity analysis,* Master's thesis, TU Delft (2022).

[42] Nvidia, *NVIDIA A100 Tensor Core GPU Architecture*, Tech. Rep. (2020).

[43] Nvidia, *NVIDIA DGX-1 With Tesla V100 System Architecture*, Tech. Rep. (2017).

[44] D. Walker, *Standard for Message-Passing in a Distributed Memory Environment*, Tech. Rep. (Oak Ridge National Laboratory, U.S. Department of Energy, 1992).

[45] C.-W. Ou, J. Ranka, and G. Fox, *Fast and parallel mapping algorithms for irregular problems,* Journal of Supercomputing **10**, 119 (1996).

[46] A. K. Patra and D. W. Kim, *Efficient Mesh Partitioning For Adaptive HP Finite Element Meshes*, Tech. Rep. (Dept. of Mechanical Eng., State University of New York, Buffalo, 1970).

[47] A. Sohn and H. Simon, *S-HARP: A Parallel Dynamic Spectral Partitioner*, Tech. Rep. (Dept. of Computer and Information Science, New Jersey Institute of Technology, 1998).

[48] J. R. Pilkington and S. B. Baden, *Dynamic partitioning of non-uniform structured workloads with space-filling curves,* IEEE Transactions on Parallel and Distributed Systems **7**, 288 (1995).

[49] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graph,* Society for Industrial and Applied Mathematics **20**, 359 (1998).

[50] B. Monien, R. Preis, and R. Diekmann, *Quality matching and local improvement for multilevel graph-partitioning*, Tech. Rep. (Univ. of Paderborn, 2000).

[51] B. W. Kernighan and S. Lin, *An efficient heuristic procedure for partitioning graphs,* The Bell System Technical J. **49** (1969), 10.1002/j.1538-7305.1970.tb01770.x.

[52] B. Hendrickson and R. Leland, *An improved spectral graph partitioning algorithm for mapping parallel computations,* SIAM J. Scientific Computing **16**, 452 (1995).

[53] S. Hauck and G. Borriello, *An evaluation of bipartitioning techniques,* in *Proc. Conf. Advanced Research in VLSI* (1995) pp. 383–402.

[54] J. Cong and L. Smith, *A parallel bottom-up clustering algorithm with applications to circuit partitioning in VLSI design,* in *30th ACM/IEEE Design Automation Conference* (1993) pp. 755–760.

[55] K. Schloegel, G. Karypis, and V. Kumar, *Wavefront diffusion and LMSR: Algorithms for dynamic repartitioning of adaptive meshes,* IEEE Transactions on Parallel and Distributed Systems **12**, 451 (2001).

[56] K. Schloegel, G. Karypis, and V. Kumar, *A unified algorithm for load-balancing adaptive scientific simulations,* in *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing* (IEEE, 2000) p. 59.

[57] K. Schloegel, G. Karypis, and V. Kumar, *Multilevel Diffusion Schemes for Repartitioning of Adaptive Meshes*, Tech. Rep. (1997).

[58] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, *Zoltan data management services for parallel dynamic applications,* Computing in Science & Engineering **4**, 90 (2002).

[59] D. Roose, K. U. Leuven, B. Maerten, A. Basermann, J. Fingberg, and G. Lonsdale, *DRAMA: A Library for Parallel Dynamic Load Balancing of Finite Element Applications*, Tech. Rep. (1999).

[60] M. Schwarz and H. P. Seidel, *Fast parallel surface and solid voxelization on GPUs,* ACM Transactions on Graphics **29**, 9 (2010).

[61] T. Akenine-Möller, *Fast 3D triangle-box overlap testing,* Journal of Graphics Tools **6**, 29 (2001).

[62] C. Ericson, *Real-Time Collision Detection* (Elsevier, 2005).

[63] C. B. Barber and D. P. Dobkin, *The quickhull algorithm for convex hulls,* ACM Trans. Math. Softw. **22**, 469 (1996).

[64] T. M. Chan, *A Minimalist's Implementation of the 3-d Divide-and-Conquer Convex Hull Algorithm*, Tech. Rep. (2003).

[65] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan, *Efficient collision detection using bounding volume hierarchies of k-DOPs,* IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS **4**, 21 (1998).

[66] R. Ytterlid and E. Shellshear, *BVH split strategies for fast distance queries,* Journal of Computer Graphics Techniques BVH Split Strategies for Fast Distance Queries **4** (2015).

[67] Y. Gu, Y. He, K. Fatahalian, and G. Blelloch, *Efficient BVH construction via approximate agglomerative clustering,* in *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13 (Association for Computing Machinery, New York, NY, USA, 2013) p. 81–88.

[68] J. A. Bærentzen and H. Aanæs, *Signed distance computation using the angle weighted pseudonormal,* IEEE Transactions on Visualization and Computer Graphics **11**, 243 (2005).

[69] A. A. Shabana, *Dynamics of Multibody Systems* (Cambridge University Press, 2020).

[70] M. Chávez-Modena, J. L. Martínez, J. A. Cabello, and E. Ferrer, *Simulations of aerodynamic separated flows using the lattice Boltzmann solver XFlow,* Energies **13** (2020), 10.3390/en13195146.

[71] W. Haase, M. Braza, and A. Revell, *DESider – A European Effort on Hybrid RANS-LES Modelling* (Springer, 2009).

[72] A. Peters, Z. S. Spakovszky, W. K. Lord, and B. Rose, *Ultra-short nacelles for low fan pressure ratio propulsors,* in *Proceedings of the ASME Turbo Expo*, Vol. 1A (American Society of Mechanical Engineers (ASME), 2014) pp. 1–15.

[73] L. Lobuono, D. MacManus, R. Christie, and L. Boscagli, *Unsteady aerodynamics of a coupled compact intake-fan in crosswind,* in *34th Congress of the International Council of the Aeronautical Sciences* (ICAS Proceedings, 2024).

[74] B. K. Hodder, *An investigation of Engine Influence On Inlet Performance*, Tech. Rep. (NASA Ames Research Center, 1981).

[75] T. S. Tse and C. A. Hall, *Aerodynamics and power balance of a distributed aft-fuselage boundary layer ingesting aircraft,* MDPI Aerospace **10**, 1 (2023).

[76] A. L. Habermann, J. Bijewitz, A. Seitz, and M. Hornung, *Performance bookkeeping for aircraft configurations with fuselage wake-filling propulsion integration,* CEAS Aeronautical Journal (2019), 10.1007/s13272-019-00434-w.

[77] M. Seidel and F. Jaarsma, *The German–Dutch low speed wind tunnel DNW,* The Aeronautical Journal **82**, 167–173 (1978).

[78] W. Gracey, *Wind-tunnel investigation of a number of total-pressure tubes at high angles of attack subsonic, transonic, and supersonic speeds*, Tech. Rep. (NACA-TR-1303, 1953).

[79] D. R. Boldman, C. Iek, D. P. Hwang, M. Larkin, P. S. Pratt, and W. E. Hartford, *Effect of a rotating propeller on the separation angle of attack and distortion in ducted propeller inlets,* in *31st Aerospace Sciences Meeting and Exhibit* (AIAA, 1993) pp. 1–14.

[80] S. Tambe, U. B. Oseguera, and A. G. Rao, *Performance of a low speed axial fan under distortion: An experimental investigation,* in *Proceedings of ASME Turbo Expo 2020: Turbine Technical Conference and Exposition* (ASME, 2020) pp. 1–11.

[81] F. R. Menter, *Two-equation eddy-viscosity turbulence models for engineering applications,* AIAA Journal **32**, 1598 (1994).

[82] M. W. Jones, *3D Distance from a Point to a Triangle*, Tech. Rep. (Department of Computer Science, University of Wales Swansea (CSR-5-95), 1995).

[83] V. J. Lumelsky, *On fast computation of distance between line segments,* Information Processing Letters **21**, 55 (1985).