UNIVERSITY OF SOUTHAMPTON

FACULTY OF ENGINEERING AND THE ENVIRONMENT

Engineering Sciences

Robust Automated Computational Fluid Dynamics Analysis and Design Optimisation of Rim Driven Thrusters

by

Aleksander J. Dubas

Thesis for the degree of Doctor of Philosophy

October 2014

ABSTRACT

UNIVERSITY OF SOUTHAMPTON

FACULTY OF ENGINEERING AND THE ENVIRONMENT

Doctor of Philosophy   Engineering Sciences

ROBUST AUTOMATED COMPUTATIONAL FLUID DYNAMICS ANALYSIS AND DESIGN OPTIMISATION OF RIM DRIVEN THRUSTERS

by Aleksander J Dubas

The rim driven thruster is a novel electromagnetic marine propulsion device that uses a motor in its casing to drive a propeller by its rim. There are many interacting flow features posing a number of challenges when it comes to simulating the device with computational fluid dynamics. The primary concern is finding a suitable simulation method to capture the flow behaviour accurately, though a secondary challenge is created by the complex interactions creating a rugged design landscape that is difficult to optimise.

A steady-state simulation method has been developed and a verification and validation process was conducted on a B4-70 standard series propeller as a baseline case. Results show a great sensitivity to computational domain size below a radial distance of five propeller diameters. The Re-Normalisation Group (RNG) k-ϵ and k-ω Shear Stress Transport (SST) turbulence models were compared and the k-ω SST model was found to be the most robust due to its better handling of separation that occurs at low propeller advance ratios.

To investigate the capture of rotor-stator interaction by the frozen rotor formulation an unsteady simulation method was developed. The unsteady method was also verified and validated, showing good agreement for a standard series propeller, and subsequently applied to rim driven thruster simulations. The results show the frozen rotor formulation does capture some variation and has reasonable agreement with thrust variation over one rotation, but does not predict the variation in torque accurately and thus is considered insufficient for rotor-stator interaction modelling.

While the capture of rotor-stator interaction is flawed in frozen rotors, if the stators are omitted, the steady state simulation method is suitable for performance prediction. Given the computational cost of full unsteady simulation, steady state was chosen for the objective function calculation method for the design optimisation. A library of functions was written to robustly automate the geometry creation, mesh generation, solution and post-processing. An initial design study of the sensitivity of 13 parameters showed that the most significant variables were pitch distribution, thickness distribution and hub diameter. These were factored into a second design optimisation study of six parameters, using Kriging for surrogate modelling, to produce an improved rim driven thruster design.

The improved design features a greater pitch at the tip exploiting the lack of tip-leakage experienced with rim drive. A high sensitivity of the hydrofoil to Reynolds number was discovered and exploited by increasing the blade thickness and pitch to make the blade section produce more force over a greater area of the blade. The open water efficiency of the improved design is 0.06 higher than the baseline design, showing the optimisation was a success.

Contents

1 Introduction **13**

1.1 Motivation 13

1.2 Aims and Objectives 14

1.3 Thesis Outline 15

2 Propeller Design and Computational Fluid Dynamics Analysis **17**

2.1 General Background 17

2.1.1 Anatomy of The Propeller 17

2.1.2 Non-Dimensional Parameters 19

2.1.3 The 100mm Rim Driven Thruster 20

2.2 Background to Computational Fluid Dynamics 22

2.2.1 Methods of Numerical Solutions 22

2.3 Reynolds Averaged Navier Stokes Equations Solutions 25

2.3.1 Turbulence Modelling 26

2.3.2 k-ϵ Turbulence Model 27

2.3.3 k-ω Turbulence Model 28

2.3.4 RNG k-ϵ Turbulence Model 28

2.3.5 k-ω SST Turbulence Model 29

2.3.6 Turbulent Boundary Layers 30

2.3.7 Turbulent Intensity and Turbulent Viscosity Ratio 31

3 State of the Art in Design Optimisation and Numerical Analysis of Propeller Performance **35**

3.1 The State of the Art in Computational Fluid Dynamics of Hydraulic Turbomachinery and Propulsors 35

3.2 Turbulence Model Selection 44

3.3 Ducted and Rim Driven Propulsors 45

3.4 Design Optimisation of Marine Propulsors 46

4 Computational Fluid Dynamics Methods **49**

4.1 Geometry Creation 51

4.1.1 Propeller Surface Visual Basic Macro 51
4.1.2 Automated Geometry Generation 51
4.2 Meshing for Steady State Simulations 54
4.2.1 Automated Mesh Generation 56
4.3 Steady State Solution Setup 60
4.4 Automation of Steady State Solution 63
4.4.1 Simulation Case Setup 64
4.5 Running The Simulation 65
4.5.1 Automatically Running The Simulation From Python 66
4.6 Automated Post Processing 67
4.7 Example: Steady State Simulation of Wageningen B4-70 67
4.8 Unsteady Simulation Method 69
4.9 Unsteady Meshing 69
4.9.1 Dynamic Meshing 70
4.10 Unsteady Solution Set Up for OpenFOAM 71
4.11 Solving Unsteady Simulations 73

5 Verification and Validation of Computational Fluid Dynamics 75
5.1 Steady Method Verification 75
5.1.1 Boundary Distance Investigation 76
5.2 Steady Method Validation 81
5.2.1 Convergence Problems at Low Advance Ratios 81
5.2.2 Bollard Pull Condition 83
5.2.3 Summary 84
5.3 Unsteady Method Verification and Validation 85
5.3.1 Numerical Start Up of Wageningen B4-70 Propeller 90

6 Results for Simulation of Rim Driven Thrusters 93
6.1 Preliminary Results for 70mm Rim Driven Thruster 93
6.1.1 Analysis of Preliminary 70mm Rim Driven Thruster Results 96
6.2 Raw Results for the 100mm Rim Driven Thruster 98
6.3 Analytical Annulus Models 102
6.3.1 Radial Gap Models 102
6.3.2 Axial Gap Models 104
6.4 Results for 100mm Rim Driven Thruster Including Annulus Models 106
6.4.1 Annulus Analytical Modelling Summary 108
6.5 Computational Fluid Dynamics Simulation of the Annulus 108
6.6 Steady State Simulations Summary 109
6.6.1 Key Design Areas 110
6.7 Unsteady Simulation Results 110

6.7.1 Start Up of 100mm Rim Driven Thruster 111
6.7.2 Unsteady 100mm Rim Driven Thruster Results 112
6.7.3 Origins of the Unsteady Force Variations 117

7 Design Optimisation Study of the 100mm Rim Driven Thruster 121

7.1 Parameterisation and Parameter Selection 121
7.2 Surrogate Modelling 132
7.2.1 Kriging 132
7.2.2 Optimisation Strategy 135
7.2.3 Surrogate Model Visualisation 136
7.2.4 Surrogate Modelling Validation 148
7.3 Results from Design Optimisation 153
7.3.1 Device Performance 153
7.3.2 Blade Pressure Profiles 158
7.3.3 Cavitation 159

8 Conclusions 165

8.1 Conclusions 165
8.2 Suggestions for Future Work 168

A Propeller Blade Surface Co-ordinate Generating Program 177

B Stereolithographic Format Generation and Writing Program 193

C OpenFOAM Automation Functions 207

D Optimisation Functions 235

Declaration of Authorship

I, Aleksander J Dubas, declare that this thesis and the work presented in it are my own and have been generated by me as the result of my own original research.

Thesis Title: *Robust Automated Computational Fluid Dynamics Analysis and Design Optimisation of Rim Driven Thrusters.*

I confirm that:

- This work was done wholly or mainly while in canditature for a research degree at this University;
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- Where I have consulted the published work of others, this is always clearly attributed;
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- Either none of this work has been published before submission, or parts of this work have been published as listed in the publications section.

Signed: .... Date: ....

Acknowledgements

I would like to thank my supervisors for their support in the production of this work and their timely feedback on all that I have written throughout my canditature. Secondly, this thesis would not have been possible without the support of my friends and family through the highs and the lows of the last few years.

The help of TSL Technology Ltd in the generation of blade geometry and provision of geometry files and accompanying experimental data for existing rim driven thrusters has been invaluable and is much appreciated. This work was supported by an EPSRC Doctoral Training Centre grant (EP/G03690X/1). Finally I would like to acknowledge the use of the IRIDIS High Performance Computing Facility, and associated support services at the University of Southampton, in the completion of this work.

List of Publications and Presentations

Published Papers

- Dubas A J, Bressloff N W, Fangohr H, Sharkh S M - *"Computational Fluid Dynamics Simulation of a Rim Driven Thruster"* in Proceedings of the Open Source CFD International Conference, Paris, France, 2011

Submitted Papers

- Dubas A J, Bressloff N W, Sharkh S M - *"Numerical Modelling of Rotor-Stators Interaction in Rim Driven Thrusters"* submitted to Ocean Engineering

Papers in Preparation

- Dubas A J, Bressloff N W, Sharkh S M - *"Optimisation of Rim Drive Propellers using Surrogate Modelling"*

Presentations

- *"Robust Automated Simulation of the Complex Flow Features in a Rim Driven Thruster"*, Poster at Student Conference on Complexity Sciences, Oxford, UK, August 2013

- *"Simulating the Complex Flow Features of a Rim Driven Thruster"*, Poster at 1st Postgraduate Conference in Engineering Sciences, Southampton, UK, November 2012

- *"Numerical Simulation of the Complex Flow Through Rim Driven Thrusters"*, Talk at European Fluid Mechanics Conference 9, Rome, Italy, September 2012

- *"Simulating the Complex Flow Features of a Rim Driven Thruster"*, Talk at Student Conference on Complexity Sciences, Gloucester, UK, August 2012
- *"Computational Fluid Dynamics Simulation of a Rim Driven Thruster"*, Talk at Open Source CFD International Conference, Paris, France, October 2011
- *"Complexity in Computationally Simulating a Rim Driven Thruster"*, Talk at Student Conference on Complexity Sciences, Winchester, UK, August 2011

Chapter 1

Introduction

1.1 Motivation

Marine propulsion has been achieved through a large variety of different methods since people first set sail on the sea, from the oars and sails of early mariners to more recent ideas such as paddle wheels and the screw propeller. While the design of the screw propeller could be originally attributed to Leonardo Da Vinci or perhaps even to Archimedes, the technology as a method of marine propulsion was widely adopted and developed in the 19th century as the design of engines to drive the propellers also improved (Carlton, 2007).

Recent developments in marine propulsion and electrical machines have led to the development of the rim driven thruster which is a propulsive device that is, as the name implies, driven by the rim rather than the more conventional method of a shaft (Sharkh et al., 2001). It is this device that has been chosen to be studied in this project as there are some additional features to the device compared to a typical propeller that add interest to the flow, and compose a different design environment that has not yet been fully explored.

While there are similarities between rim drive devices and ducted propellers, there are sufficient differences in the hydrodynamics that there are novel aspects to the design space arising from the rim drive. In particular the presence of the rim and its connection to the propeller blades change what is possible in the blade design, both hydrodynamically and structurally. To the author's best knowledge, a surrogate modelling based design optimisation study of rim driven thrusters using Reynolds-Averaged Navier-Stokes computational fluid dynamics has not previously been conducted.

1.2 Aims and Objectives

The aims of this project are to optimise and improve upon the design of a rim driven thruster and to gain a better understanding of the sources of hydrodynamic losses. Insight into how different design parameters affect the performance of the device is desired and the key design areas are to be highlighted. Due to the encapsulated nature of a rim driven thruster, experimentally visualising and probing the flow within is a physically challenging process. For this reason, computational methods are instead employed in this project, with the additional benefit of being able to rapidly and inexpensively evaluate changes without having to remanufacture prototypes. However, computational methods are not without their pitfalls and a number of potential difficulties in the modelling must be resolved, mitigated or avoided.

The performance derived from a propeller, and the inevitable losses, arise from the interaction of a large number of flow features, each adding to the complexity of the flow and rendering the task of its simulation more difficult. Viscous effects and boundary layer development over the propeller are important to model as their contribution is not insignificant, though a good first approximation can be obtained using inviscid methods. The typical Reynolds numbers of propeller operating conditions are moderate to high and consequently the flow is turbulent which then requires either sufficient temporal and spatial resolution to simulate the turbulence directly, or a model of the effects of the turbulence on the mean flow. At the larger scales, there are tip and hub vortices which should not be neglected, as well as radial pumping along the blade, which renders two dimensional approximations such as blade element methods inaccurate. Some propellers operate at a scale and speed at which a significant portion of the boundary layer on the blade is laminar and thus accurate transition prediction is integral to a good solution, which is known to be intrinsically difficult. The wake of upstream elements in the flow can impart a non-uniform inflow to the propeller which can cause unsteady effects as the blades rotate through the non-uniform wake. Finally, as the medium of operation is typically water, if the propeller is sufficiently loaded the local pressure in the flow field can drop below the vapour pressure of the fluid causing it to change to the gaseous phase, and then back to liquid again when the pressure recovers, in a phenomenon known as cavitation. This adds further complexity to the computational simulation of the flow as the transport of both liquid and gaseous phases must be included in the model as well as the transition between them.

In addition to all these modelling challenges for an open water propeller, there are additional features on a rim driven thruster that need consideration. There is a ducting that houses the motor that drives the rim driven thruster, having a hydrodynamic effect on the inflow to the propeller which is dependant on its own contribution to thrust or drag. Also, there are stators that support the axle on which the propeller rotates which can be located in a variety of configurations upstream or downstream of the propeller that can cause either beneficial or detrimental interactions with the rotor. Finally, the rim that forms the part of the motor drive that is attached to the propeller tips can create interesting flow features, such as Taylor-Couette vortices (Batten, 2002), between itself and the duct when it rotates. Thus a significant effort is required to ensure that the computational simulations output a result which is relevant to the real flow.

Once the computational method has been developed, if it is made robust and automated, it enables the further use of the computational method in a iterative design optimisation study. However, due to the time taken to evaluate a single design, direct optimisation is not practicable, and a surrogate modelling approach is used instead. In this particular case, Kriging is used as the surrogate model, which uses radial basis functions to compose the response surface.

To summarise, the aims of improving and understanding the design of a rim driven thruster can be achieved through satisfaction of the following objectives:

- Develop, verify and validate a computational method for simulating rim driven thrusters.
- Use computational simulation results to identify key design regions.
- Automate the computational performance evaluation in a robust manner with respect to different geometries.
- Optimise the design of a rim driven thruster using surrogate modelling.
- Visualise the response surface to gain insight into the significance of design parameters.

With these completed, further knowledge of the design of rim driven thrusters will be gained, leading to the improved efficiency of marine propulsors and the subsequent reduction of environmental impact that improved efficiency brings.

1.3 Thesis Outline

This chapter, Chapter 1, begins by introducing the motivation to this work, the aims it sets out to achieve and the deliverable objectives through which the aims will be fulfilled. The next chapter, Chapter 2, gives a background of foundation knowledge in the fields of propeller design and computational fluid dynamics, upon which the following chapter, Chapter 3, builds upon to bring the reader up to date with the state of the art and review the contemporary literature in the context of this work.

The fourth chapter of this thesis details the methods employed to robustly evaluate and automate the computational fluid dynamics solutions. This includes all stages from geometry creation, through meshing and solution setup, to post processing the solution data for both steady state and unsteady simulation methods. These methods are subsequently employed to verify and validate the simulation model against a test case of a Wageningen B4-70 propeller in Chapter 5.

Application of the simulation method to rim driven thrusters is presented in Chapter 6, first for a 70mm thruster and later a 100mm thruster. The results from these simulations are used to yield insight into key design areas and highlight flow features of interest in the rim driven thrusters while enabling a critique of the methodology when applied to rim driven thrusters as opposed to the open water propeller study in Chapter 5.

Moving forward with the insights gained from Chapter 6, Chapter 7 details a design optimisation study undertaken on the 100mm rim driven thruster. Initially parameterised into 13 parameters, this is subsequently reduced to six key parameters, which are optimised to maximise open water efficiency using a Kriging surrogate model searched by genetic algorithm. The optimised design is then compared to the baseline to ascertain where the improvements originated.

Finally, the conclusion of this thesis is given in Chapter 8, with some proposals for future work that may be undertaken to build upon the work presented herein.

Chapter 2

Propeller Design and Computational Fluid Dynamics Analysis

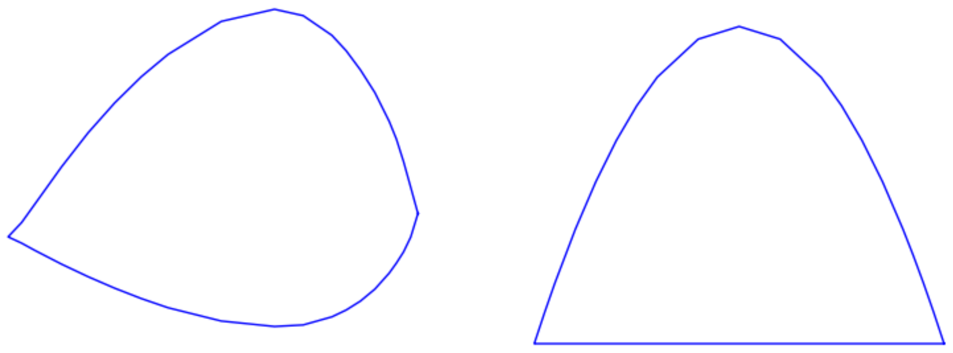
2.1 General Background

2.1.1 Anatomy of The Propeller

There are many parts to a propeller, which are often referred to in a variety of different ways, thus the convention used in this report is outlined here. A propeller typically consists of two main parts, the propeller hub and the propeller blade. The hub is the central part of the propeller onto which the blades are mounted and has a typical diameter of 15% to 20% of the overall diameter of the propeller. The blades are mounted on to the hub; the end of the blade which is on the hub is the blade root and the outer end is the blade tip.

A number of geometrical parameters must be specified to fully describe the blades of a propeller. The first of these is the propeller diameter against which all other dimensions of the blade are usually normalised. The non-dimensional distance along the blade is given by $x = r/R$, where r is the radial position and R is the propeller radius; half the diameter.

The propeller blade can be divided into hydrodynamic sections along its radial length, over which the flow can be thought to pass over in a two-dimensional sense. Although, in practice, radial pumping and vortical structures from the blade tip and root add three-dimensional flow characteristics that invalidate this two-dimensional simplification. The local angle of incidence to the flow of the blade sections is dependent on the advance velocity, radial location and speed of rotation as well as the installed angle, which is typically defined through a concept known as pitch. The term pitch comes from the original design of the screw propeller, which was



(a) Normalised blade section at r/R = 0.2 (b) Normalised blade section at r/R = 0.9

Figure 2.1: Examples of blade sections from a Wageningen B series propeller.

based on the same principles as a screw, and refers to the distance that the screw would travel in one full turn without slipping. Commonly, it is expressed non-dimensionally as a pitch ratio, which is the ratio of the pitch to the diameter, with typical values of pitch ratio ranging from 0.5 to 1.4 although higher pitch ratios are used in some applications. While there are fixed, constant pitched propellers in use, it is not uncommon for the pitch to vary with radial location along the blade or for the entire pitch of the propeller to be changeable during operation to improve efficiency where loads are variable.

As well as the local angle of incidence, the hydrodynamics of each blade section are also affected by its size and shape, which are typically varying along the length of the blade. The chord is the length of the blade section from the leading edge to the trailing edge and is typically given by values normalised by the diameter for each radial station. Similarly, the thickness of the blade section varies along the blade and is normalised by the diameter. However, as thickness varies from leading edge to trailing edge, it is only the maximum thickness that is usually given. Thus the remainder of the profile of the blade sections must be defined, which can be constant along the length of the blade but is usually varied. Typically, more hydrodynamically streamlined sections (e.g. Figure 2.1a), such as NACA aerofoil sections, are used at the root of the blade with the section progressing to a more ogival shape (e.g. Figure 2.1b) towards the tip of the blade as cavitation becomes a concern.

There are three commonly used methods for measuring the area of a propeller; the simplest method is to measure the area of the disc swept by the propellers, which can be found from the radius using $A_{disc} = \pi r^2$. Similarly, if the propeller is viewed from directly astern, the projected area of the propeller onto this plane can be measured. This gives the projected area and can also be used to calculate the blade area ratio, which is the ratio of the projected area to the disc area. Finally there is the expanded area (A_E), also called the developed area (A_D), which is the area that would be occupied if the propeller blades were flattened. The expanded area ratio (EAR, also known as disc-area ratio, DAR) can then be calculated from this as the ratio of the expanded area to the disc area.

It can be easily shown that the larger the diameter of a propeller, the greater the propulsive efficiency. However often there are geometric constraints that limit the diameter of a propeller that can be fitted, or the required torque, which also increases with propeller diameter, cannot be delivered by the installed powerplant and gearbox combination. To overcome the geometrical or torque delivery constraints, marine propulsion engineers designed a method of increasing the effective diameter without actually increasing the diameter. This is done by raking the propeller, adding slope either forwards or backwards when viewed from the side, and the propeller rake is conventionally defined as positive when backwards. In some applications, where it is important to minimise noise or vibration, propellers are skewed, which can be seen by a backwards sweeping of the blade contour when viewed along the rotational axis. This allows each radial section of the blade to enter the water at different times and thus have less synchronous pressure pulse effect when, for example, cutting across the non-uniform wake of the hull.

2.1.2 Non-Dimensional Parameters

To evaluate the performance of a propulsive device in a manner that can be compared to other devices of different sizes and operating conditions, non-dimensional parameters are used to normalise performance against values it may depend upon. For propellers there are four key non-dimensional parameters: thrust coefficient, K_T, torque coefficient, K_Q, open water efficiency, η_o, and advance coefficient, J. Also of importance is the relative effect of viscous and inertial forces in the flow which is typically characterised by the Reynolds number R_n and also, if examining the cavitation properties of a propeller, the cavitation number, σ_0, is of interest here. Thus the non-dimensional parameters for a propeller are as follows:

- Thrust Coefficient:
$$K_T = \frac{T}{\rho n^2 D^4} \tag{2.1}$$
where T is the thrust [N], ρ is the fluid density [kg/m^3], n is the rotational speed [revs/s] and D is the propeller diameter [m].

- Torque Coefficient:
$$K_Q = \frac{Q}{\rho n^2 D^5} \tag{2.2}$$
where Q is the torque [Nm].

- Open Water Efficiency:

$$\eta_O = \frac{TV_a}{2\pi n Q} = \frac{K_T}{K_Q}\frac{J}{2\pi} \tag{2.3}$$

  where V_a is the advance velocity [m/s].

- Advance Coefficient:

$$J = \frac{V_a}{nD} \tag{2.4}$$

- Reynolds Number:

$$R_n = \frac{\rho n D^2}{\mu} \tag{2.5}$$

  where μ is the dynamic viscosity [Ns/m^2].

- Cavitation Number:

$$\sigma_0 = \frac{p_0 - e}{\frac{1}{2}\rho n^2 D^2} \tag{2.6}$$

  where $(p_0 - e)$ is the local static pressure [N/m^2]. Taken from the Bernoulli relationship where p_0 is the total pressure and e is the dynamic pressure.

2.1.3 The 100mm Rim Driven Thruster

The device that is the ultimate focus of this study is a novel electromagnetic device developed at the University of Southampton (Sharkh et al., 2001) that drives the propeller by its tips, rather than by the shaft, using a rim fixed to the propeller tips and thus given the name 'Rim Driven Thruster'. Preliminary results were obtained using a 70mm thruster geometry but, due to insufficient experimental data, the study subsequently switched focus to the 100mm IntegratedThruster™ as produced and sold by TSL Technology Ltd, who have kindly provided geometrical and experimental data for this study.

A picture of the rim driven thruster is shown in Figure 2.2 which shows its similarities to a ducted propeller. While the ducting of the rim driven thruster is essential to its operation as it houses the motor for the device, there are both advantages and disadvantages to ducting a propeller. The duct can be shaped such that the inflow to the propeller is increased, which increases efficiency at low speeds, although the additional drag from having a duct eventually leads to diminishing efficiency at higher speeds, thus the common application of ducted propellers is for low speed and heavily loaded devices such as those on tug boats. Alternatively, the duct can be shaped so as to reduce inflow speeds to the propeller, which will consequently increase the local static pressure on the blades through the Bernoulli effect, reducing the amount of cavitation but with a penalty to efficiency. Consequently, a decelerating type of duct is typically only applied where noise reduction is more important than efficiency, for example a military submarine propulsion system. The advantages of driving a ducted propeller by the rim are that there is no need for a driveshaft or gearbox which increases the compactness of the device and also enables the rim driven thruster to be better suited to bi-directional operation. Due to the bi-directional design of the IntegratedThruster™ the ducting does not have as much impact on the performance as a Kort nozzle might (Carlton, 2007, pp. 15–17), however there are unidirectional designs available with a preferrential thrust direction, thus asymmetric ducts are also of interest for rim driven thrusters, but not covered in the scope of this work.

The parts that make up a rim driven thruster can be seen in the cut-through diagram in Figure 2.3. From the outside to the centreline of the diagram; first there is the duct, or casing, that houses the motor and stator windings. Next is the rim, which also forms part of the motor and this is attached to the tips of the blades. There are stators attached to the duct, whose primary purpose is to hold the hub and shaft bearings in place. Finally there is the shaft, which is solely for locating the blades, rather than transmitting torque to the blades as it is in conventional propellers. For convenience and contrast to the non-rotating components of the device, the shaft, blades and rim are henceforth collectively referred to as the rotor. To reduce vibration in the device it is typical to have a different number of stators to blades, in the case of the 70mm and 100mm IntegratedThruster™ there are three stators (as seen in Figure 2.2) and four blades.

2.2 Background to Computational Fluid Dynamics

To simulate the hydrodynamics of the rim driven thruster, this project proposes to use computational fluid dynamics methods. While it could fill many books to detail all the available methods, the most commonly used methods in marine propulsion are summarised here and the chosen method for this project is described in more detail.

2.2.1 Methods of Numerical Solutions

The complexity of fluid motion makes the governing equations difficult to solve using traditional mathematical calculus, put eloquently by Leonhard Euler himself: *"If it is not permitted to us to penetrate to a complete knowledge concerning the motions of fluids, it is not to mechanics, or to the insufficiency of the known principles of motion, that we must attribute the cause. It is analysis itself which abandons us here."* Indeed, to find the solution to all but a simplified subset of flows, numerical methods must be employed. Numerical solution of differential equations pre-date the advent of the modern computer although increasing computational power has enabled quicker and higher fidelity solutions to equations for which few analytical solutions are known.

Early attempts at computational fluid dynamics of propellers involved using simplifications to make the problem tractable such as potential flow solutions which assume that the flow is both inviscid and irrotational. When obtaining a potential flow solution, there are two commonly employed methods of geometry description. The first is to take many radial slices of the blade and compute a local two-dimensional flow based on the inflow and rotational velocity of each slice and then integrate the contribution from each slice along the blade. This method is known as the blade element method and can be extended to include the effects of viscosity through using a viscous-inviscid interaction solver when computing the local two-dimensional flow. However, blade element method has a major limitation in its lack of three-dimensionality, as its formulation inherently does not account for any radial components of the velocity field. However, blade element method can produce a reasonable approximation when combined with momentum theory (Benini, 2004).

An alternative potential flow solution to the blade element method is to use full three-dimensional panel methods (Kerwin et al., 1987; Hughes et al., 2000). This is where the surface of the propeller is represented by a number of panels that consist of potential flow elements, for example sources and doublets, and control points. A linear problem is constructed by formulating a system of equations stating that the sum of velocity contributions from each potential flow element must be tangential to the surface at every control point. The system of equations is then solved to find the strength of each potential flow element, allowing the velocity at any point in the field to be calculated from the contributions from every flow element. While this method is three-dimensional, it does not account for the effects of viscosity or vorticity, both of which form a significant part of the flow around a propeller. The importance of vortices in rotating devices has lead to recent development of meshless vortex methods (Zhu et al., 2012), which can be combined with panel methods to provide an efficient solution method that also captures the key vortical elements.

Both blade element and panel methods are numerical methods that fall into the category of boundary element methods. These only require a mesh over the boundaries of interest (i.e. the blade surface) and thus are typically more computationally efficient than finite volume methods but are limited in their ability to model complex flow features. In contrast, finite element, finite difference and finite volume methods require a meshing of the entire fluid volume and consequently the effects of the far field boundaries must be taken into consideration. However, the meshing of the entire fluid domain, provided it is sufficient in resolution, allows the capture of more complex flow features by the solution. While both finite element and finite difference methods are used in computational fluid dynamics, finite volume methods are much more prevalent and widely used in many applications including popular commercial computational fluid dynamics packages such as ANSYS FLUENT.

As a finite volume method can model vorticity and viscosity effects, it is necessary to consider the effects of turbulence; whether the flow is at sufficient Reynolds number to be turbulent and whether the mesh is sufficiently fine to capture all the relevant scales of motion. Typically, propeller flows are turbulent but it is not practicable to resolve all the turbulent scales, thus it is common in computational fluid dynamics to utilise some form of turbulence modelling. Perhaps the most prevalent models are Reynolds-Averaged Navier-Stokes (RANS) models which only solve for the mean flow and use a turbulence model to estimate the effects of the turbulent fluctuations on the mean flow. RANS solutions are a good compromise between accuracy and computational requirements and have consequently been chosen for the design optimisation study in this project. The RANS equations are not a fully closed system of equations by themselves and require a turbulence closure model to make them complete. The selection of turbulence closure model can have a great impact on accuracy, solution time and robustness and further details on RANS methods can be found in Section 2.3 as well as a more in depth look at RANS turbulence modelling in Section 2.3.1. It is worth clarifying that finite volume methods can also be used to solve laminar, inviscid and irrotational flows.

With sufficient computational power, it is possible to resolve the larger turbulent scales in the simulation and model the unresolvable smaller scales known as sub-grid scales (SGS). This is done with Large Eddy Simulation (LES), which still requires a turbulence model, however as the small unresolved scales are less energetic and less specific to the flow, the results obtained with LES are typically more accurate than those achieved with RANS. Solution times for LES are substantially longer than those for RANS and results are inherently unsteady so time averaging is necessary to find ‘steady-state’ solutions for variables such as thrust and torque. Large eddy simulation does suffer from a requirement to have a very refined mesh near to walls which increases computational cost significantly to achieve a good result. One solution to this drawback is to use Detached Eddy Simulation (DES).

Detached eddy simulation is a hybrid method which exploits the advantages of both LES and RANS. The main principal behind DES is to use a RANS turbulence model in regions that are close to the walls and LES away from the walls. This gives the method the cost benefits of using RANS in the wall region while retaining the unsteady flow and resolved large eddies captured by LES. However the use of DES in computational fluid dynamics simulations of marine propulsors is not as widespread as that of RANS and LES methods.

To complete the spectrum of available methods in computational fluid dynamics, it is necessary to mention Direct Numerical Simulation (DNS), where all the time and length scales of turbulence are resolved. An idea of the order of magnitude of simulation size can be gleaned from the fact that, for highly periodic homogeneous flows, the required spatial resolution scales with $R_n^{9/4}$ and the required temporal resolution scales with $R_n^{3/4}$ where R_n is the integral scale Reynolds number or ‘turbulence’ Reynolds number and usually smaller than the large scale flow Reynolds number. Thus the computational cost of direct numerical simulation becomes prohibitively expensive even at moderate Reynolds numbers and it is certainly not practicable as a method at the typical Reynolds numbers at which marine propulsors operate.

2.3 Reynolds Averaged Navier Stokes Equations Solutions

The instantaneous velocity at a single point in a steady state turbulent flow can effectively be viewed as a (pseudo-) random fluctuation, u' about some mean velocity, U. This forms the basis of the Reynolds decomposition which is the beginning of the derivation of the Reynolds-Averaged Navier-Stokes equations. Similarly the pressure can be decomposed into mean, p, and fluctuating, p', parts. In the case of compressible turbulent flows, the density is also decomposed in mean and fluctuating parts, ρ and ρ' respectively. However as all the flows considered in this thesis are incompressible, the decomposition of density is omitted henceforth. Starting with the Navier-Stokes equations (for incompressible fluids, that is density ρ = constant):

$$\frac{\partial u_j}{\partial x_j} = 0 \tag{2.7}$$

$$\frac{\partial u_i}{\partial t} + \frac{\partial u_j u_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial p}{\partial x_i} + \frac{\partial \tau_{ji}}{\partial x_j} + f_i \tag{2.8}$$

$$\tau_{ji} = -\frac{2}{3}\nu\frac{\partial u_k}{\partial x_k}\delta_{ij} + \nu\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) \tag{2.9}$$

where u_i is the ith component of velocity, t is time, x_i is the ith component of displacement, ρ is the density, p is the pressure, τ_{ji} is the viscous stress tensor, f_i is the ith component of the body force, ν is the kinematic viscosity and δ_{ij} is the Kronecker delta which evaluates to 1 if $i = j$ and 0 in all other cases. These equations undergo a Reynolds decomposition where the velocity is split into a mean and a fluctuating part, $u_i = U_i + u'_i$ leading to the following equations:

$$\frac{\partial U_j}{\partial x_j} + \frac{\partial u'_j}{\partial x_j} = 0 \tag{2.10}$$

$$\frac{\partial U_i}{\partial t} + \frac{\partial u'_i}{\partial t} + \frac{\partial U_j U_i}{\partial x_j} + \frac{\partial U_j u'_i}{\partial x_j} + \frac{\partial u'_j U_i}{\partial x_j} + \frac{\partial u'_j u'_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial p}{\partial x_i} - \frac{1}{\rho}\frac{\partial p'}{\partial x_i} + \frac{\partial \tau_{ji}}{\partial x_j} + \frac{\partial \tau'_{ji}}{\partial x_j} + f_i \tag{2.11}$$

Following the decomposition, the equations are then time averaged, such that:

$$\overline{\frac{\partial u'_j}{\partial x_j}}, \overline{\frac{\partial U_i}{\partial t}}, \overline{\frac{\partial u'_i}{\partial t}}, \overline{\frac{\partial U_j u'_i}{\partial x_j}}, \overline{\frac{\partial u'_j U_i}{\partial x_j}}, \overline{\frac{\partial p'}{\partial x_j}}. \overline{\frac{\partial \tau'_{ji}}{\partial x_j}} = 0 \qquad \overline{\frac{\partial U_j U_i}{\partial x_j}} = \frac{\partial U_j U_i}{\partial x_j} \tag{2.12}$$

This ultimately leads to the Reynolds-Averaged Navier-Stokes equations which are much the same as the Navier-Stokes equations except the solution variables are now mean (filtered) variables and there is an additional tensor term:

$$\frac{\partial U_j}{\partial x_j} = 0 \tag{2.13}$$

$$\frac{\partial U_j U_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial p}{\partial x_i} + \frac{\partial \tau_{ji}}{\partial x_j} + f_i - \frac{\overline{u'_j u'_i}}{\partial x_j} \tag{2.14}$$

While the Reynolds stress tensor, $\overline{u'_j u'_i}$, originates from the advective term on the left hand side of the equation it is often treated as a source term and placed on the right hand side as it effectively represents the effect of the turbulence on the mean flow.

As there are now extra solution variables with no extra equations, the system of equations is no longer closed. Therefore to solve these equations, additional equations must be found for the Reynolds stress tensor, for which there are two predominant classifications. There are those which are based on the Boussinesq approximation which utilises the concept of eddy viscosity:

$$-\overline{u'_i u'_j} = \nu_t \left(\frac{\partial U_i}{\partial x_j} + \frac{\partial U_j}{\partial x_i}\right) - \frac{2}{3}\left(k + \nu_t \frac{\partial U_k}{\partial x_k}\right)\delta_{ij} \tag{2.15}$$

which requires some method of determining the scalar eddy viscosity, ν_t, and the turbulent kinetic energy, $k = \frac{1}{2}\overline{u'_i u'_i}$. The most common method for determining k and ν_t is to solve additional transport equations for k and another variable that can be used to obtain an eddy viscosity, although other methods such as algebraic and one-equation models are also available, or indeed the assumption of a constant eddy viscosity in ocean current simulations. The Boussinesq models used in this work are discussed in further details in Section 2.3.1.

Alternative to turbulence models based on the Boussinesq approximation are Reynolds Stress Models (RSM) which explicitly solve transport equations for each of the six components of the Reynolds stress tensor. This leads to a minimum of six additional equations to solve, which is significantly more expensive in computational resources, and the transport equations for the Reynolds stresses also include triple products of the fluctuating part of the velocity, $\overline{u'_i u'_j u'_k}$, thus still requiring a turbulence model to close the system of equations.

2.3.1 Turbulence Modelling

There are a large number of turbulence models for Reynolds-Averaged Navier-Stokes closure available to the computational fluid dynamics practitioner (Wilcox, 1994). However, as each model is not a perfect representation of turbulence and there are a large variety of flows that could be simulated, each model is stronger at simulating and predicting some flow features than others. One analogy that describes the situation of the turbulence model is that of a fitted sheet that is too small for the mattress; it is impossible to cover the entire mattress with the sheet and therefore it must be used to cover the corner that is most likely to be used. For simulating the flow around a rim driven thruster, it is suggested that either the k-ϵ or k-ω families of turbulence model are best (see Section 3.2) and the basic principles of each are outlined in Section 2.3.2 and Section 2.3.3 respectively. The selected turbulence models of RNG k-ϵ and k-ω SST are also detailed in Sections 2.3.4 and 2.3.5 respectively.

2.3.2 k-ϵ Turbulence Model

Though there are many k-ϵ turbulence models, the 'standard' model has come to be accepted as the one proposed by Launder and Sharma (1974). The k-ϵ family of turbulence models is based on solving two additional transport equations for the turbulent kinetic energy, k, and the turbulent kinetic energy dissipation rate, ϵ, and determining the turbulent eddy viscosity, ν_t, using these two variables. The turbulent kinetic energy transport equation for steady state incompressible flow is as follows:

$$\frac{\partial k U_j}{\partial x_j} = \frac{\partial}{\partial x_j}\left[\left(\nu + \frac{\nu_t}{\sigma_k}\right)\frac{\partial k}{\partial x_j}\right] + P_k - \epsilon \tag{2.16}$$

where σ_k is a calibration constant and P_k, the production of turbulent kinetic energy, is given by:

$$P_k = 2\nu_t S_{ij}^2 \tag{2.17}$$

where S_{ij} is the mean rate of strain tensor:

$$S_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) \tag{2.18}$$

Similarly the transport equation for the turbulent kinetic energy dissipation rate, ϵ, is as follows:

$$\frac{\partial \epsilon U_j}{\partial x_j} = \frac{\partial}{\partial x_j}\left[\left(\nu + \frac{\nu_t}{\sigma_\epsilon}\right)\frac{\partial \epsilon}{\partial x_j}\right] + C_{1\epsilon}\frac{\epsilon}{k}P_k - C_{2\epsilon}\frac{\epsilon^2}{k} \tag{2.19}$$

Finally complete closure of the system of equations is achieved by relating the scalar eddy viscosity to k and ϵ:

$$\nu_t = C_\mu \frac{k^2}{\epsilon} \tag{2.20}$$

There are a number of calibration constants for this model and while these vary between applications, typical values are given here:

$$C_{1\epsilon} = 1.44, \quad C_{2\epsilon} = 1.92, \quad C_\mu = 0.09, \quad \sigma_k = 1.0, \quad \sigma_\epsilon = 1.3$$

2.3.3 k-ω Turbulence Model

The first two equation model of turbulence was a k-ω turbulence model proposed by Kolmogorov (1942), however, this model was flawed and after many years and much development, the accepted 'standard' k-ω model came to be the one proposed by Wilcox (1988). Similar to the k-ϵ models, the k-ω family of turbulence models solve a transport equation for turbulent kinetic energy, k, but differ in the choice of second variable, which is ω, the specific turbulent kinetic energy dissipation rate (i.e. the dissipation per unit turbulent kinetic energy $\omega = \epsilon/k$). With this difference, the turbulent kinetic energy transport equation becomes:

$$\frac{\partial k U_j}{\partial x_j} = \frac{\partial}{\partial x_j}\left[(\nu + \sigma_1 \nu_t)\frac{\partial k}{\partial x_j}\right] + P_k - \beta_1 k \omega \tag{2.21}$$

where the production of turbulent kinetic energy, P_k, is given by:

$$P_k = \tau_{ij}\frac{\partial U_i}{\partial x_j} \tag{2.22}$$

Similarly the transport equation for the specific turbulent kinetic energy dissipation rate, ω, is:

$$\frac{\partial \omega U_j}{\partial x_j} = \frac{\partial}{\partial x_j}\left[(\nu + \sigma_2 \nu_t)\frac{\partial \omega}{\partial x_j}\right] + \alpha\frac{\omega}{k}P_k - \beta_2 \omega^2 \tag{2.23}$$

Finally to close the system of equation, again a relationship between k, ω and ν_t is provided:

$$\nu_t = \frac{k}{\omega} \tag{2.24}$$

Similar to the closure for the k-ϵ models, calibration coefficients for k-ω vary between applications but typical values are as follow:

$$\alpha = \frac{5}{9},\quad \beta_1 = \frac{9}{100},\quad \beta_2 = \frac{3}{40},\quad \sigma_1 = \frac{1}{2},\quad \sigma_2 = \frac{1}{2}$$

2.3.4 RNG k-ϵ Turbulence Model

From the models detailed above, the two models used in this report have been developed; the Re-Normalisation Group (RNG) k-ϵ model which is outlined in this section and the k-ω Shear Stress Transport model which is outlined in Section 2.3.5. The RNG k-ϵ model is identical to the standard k-ϵ model in Section 2.3.2 except that $C_{2\epsilon}$ is replaced with $C^*_{2\epsilon}$ which is defined as:

$$C^*_{2\epsilon} = C_{2\epsilon} + \frac{C_\mu \eta^3(1 - \eta/\eta_0)}{1 + \beta\eta^3} \tag{2.25}$$

where $\eta = Sk/\epsilon$ and $S = (2S_{ij}S_{ij})^{1/2}$, with the calibration coefficients adjusted as follows:

$$C_{1\epsilon} = 1.42, \quad C_{2\epsilon} = 1.68, \quad C_\mu = 0.0845,$$

$$\sigma_k = 0.7194, \quad \sigma_\epsilon = 0.7194, \quad \eta_0 = 4.38, \quad \beta = 0.012$$

to complete the RNG k-ϵ model.

The RNG k-ϵ model was developed by Yakhot et al. (1992) to account for turbulent diffusion at all scales of motion rather than the single length scale based turbulent diffusion in the standard k-ϵ model. It is reported that the RNG k-ϵ model is best suited to rotating flows, thus it is likely to be good for modelling flows of marine propulsors, although it is typically most favoured for indoor air simulations.

2.3.5 k-ω SST Turbulence Model

The k-ω SST (Shear Stress Transport) Model has many similarities to the standard k-ω but features some minimisations and maximisations to combine the best of k-ω and k-ϵ turbulence models. The transport equation for the turbulent kinetic energy in this case is:

$$\frac{\partial k U_j}{\partial x_j} = \frac{\partial}{\partial x_j}\left[(\nu + \sigma_k \nu_t)\frac{\partial k}{\partial x_j}\right] + P_k - \beta^* k\omega \tag{2.26}$$

which from the outset looks identical to the standard k-ω model, but differs as the production of turbulent kinetic energy P_k is defined by:

$$P_k = \min\left(\tau_{ij}\frac{\partial U_i}{\partial x_j}, 10\beta^* k\omega\right) \tag{2.27}$$

The transport equation for the specific turbulent kinetic energy dissipation rate, ω, differs further from the standard k-ω model:

$$\frac{\partial \omega U_j}{\partial x_j} = \frac{\partial}{\partial x_j}\left[(\nu + \sigma_\omega \nu_t)\frac{\partial \omega}{\partial x_j}\right] + \alpha S^2 - \beta\omega^2 + 2(1-F_1)\sigma_{\omega 2}\frac{1}{\omega}\frac{k}{x_i}\frac{\omega}{x_i} \tag{2.28}$$

From k and ω, the turbulence viscosity can be calculated using the following relation:

$$\nu_t = \frac{\alpha_1 k}{\max(\alpha_1\omega, SF_2)} \tag{2.29}$$

Finally the remaining undefined terms are calculated from the following auxiliary relations, calculating the unsubscripted coefficients from Equation 2.34:

$$F_1 = \tanh\left\{\left\{\min\left[\max\left(\frac{\sqrt{k}}{\beta^*\omega y}, \frac{500\nu}{y^2\omega}\right), \frac{4\sigma_{\omega 2}k}{C_{Dk\omega}y^2}\right]\right\}^4\right\} \tag{2.30}$$

$$C_{Dk\omega} = \max\left(2\rho\sigma_{\omega 2}\frac{1}{\omega}\frac{\partial k}{\partial x_i}\frac{\partial \omega}{\partial x_i}, 10^{-10}\right) \tag{2.31}$$

$$F_2 = \tanh\left[\left[\max\left(\frac{2\sqrt{k}}{\beta^*\omega y}, \frac{500\nu}{y^2\omega}\right)\right]^2\right] \tag{2.32}$$

$$S = \sqrt{2}\,|\mathrm{symm}(\nabla\mathbf{U})| \tag{2.33}$$

Similar to the closure for the k-ω model, calibration coefficients for k-ω SST vary between applications but typical values are as follow:

$$\alpha_1 = \frac{5}{9}, \quad \alpha_2 = 0.44, \quad \beta_1 = \frac{3}{40}, \quad \beta_2 = 0.0828, \quad \beta^* = \frac{9}{100}$$

$$\sigma_{k1} = 0.85, \quad \sigma_{k2} = 1, \quad \sigma_{\omega 1} = 0.5, \quad \sigma_{\omega 2} = 0.856$$

Where a coefficient is unsubscripted, it is found from the following relationship:

$$\phi = \phi_1 F_1 + \phi_2(1 - F_1) \tag{2.34}$$

which finally completes the model.

The SST formulation of the k-ω model is designed to combine the strengths of both k-ω and k-ϵ models. It uses a k-ω formulation in the boundary layer and is applicable all the way to the wall without the addition of damping functions and in the freestream it switches to a k-ϵ behaviour to counteract the common k-ω feature of the model being too sensitive to freestream turbulence properties. The k-ω SST is often reported as being a good model for adverse pressure gradients and separating flows and that is its primary reason for being chosen for this investigation (see Section 5.2.1).

2.3.6 Turbulent Boundary Layers

An important consideration when modelling turbulent boundary layers is the mesh resolution and the near-wall treatment of the chosen turbulence model. The reasoning behind this can be found by examining the structure of a turbulent boundary layer, which can be divided into two distinct layers, the inner and outer layer. When velocity and distance are normalised into wall units, u^+ and y^+ respectively, the inner layer is assumed to have the same velocity profile in every turbulent boundary layer. The wall units are defined as:

$$u^+ = \frac{u}{u_\tau}, \quad y^+ = \frac{y u_\tau}{\nu} \tag{2.35}$$

where y is the distance away from the wall and u_τ is the friction velocity given by:

$$u_\tau = \sqrt{\frac{\tau_w}{\rho}} \tag{2.36}$$

The inner layer of the turbulent boundary layer can be subdivided into three sublayers. First, closest to the wall, is the viscous sublayer where turbulent eddies are limited in size by their proximity to the wall. The viscous sublayer extends of the region of $0 \leq y^+ \leq 5$ and the velocity profile is given by:

$$u^+ = y^+ \tag{2.37}$$

Following the viscous sublayer is a buffer region where the velocity profile undergoes a transition from the viscous sublayer to the third and final sublayer within the inner layer known as the log-law region. The log-law region extends from approximately $y^+ = 30$ to the end of the inner layer (the size of which varies) and within this region, under the present assumptions, the velocity profile follows the relationship after which it is named, the log-law of the wall:

$$u^+ = \frac{1}{\kappa} \ln y^+ + A \tag{2.38}$$

where κ is Karman's constant whose value is the topic of much debate but is typically reported to lie in the region $0.38 \leq \kappa \leq 0.43$ and in engineering is most commonly taken as 0.41. Similarly A is not definitely determined, however a typical value is approximately 5.5. After the log-law region is the outer layer, which depends fully on the external flow.

There are two different types of wall treatments for turbulence models, typically referred to as low and high Reynolds number wall treatments. When the Reynolds number is low, it is more computationally affordable to resolve the entire turbulent boundary layer, thus low Reynolds number wall treatments are designed to have the distance of the first grid point away from the wall, y_1^+, within the viscous sublayer and ideally $y_1^+ \leq 1$. Conversely, high Reynolds number wall treatments are designed to model the viscous sublayer and buffer region and begin resolving in the log-law region. In this case, good values of y_1^+ are typically in the region of $30 \leq y_1^+ \leq 50$.

2.3.7 Turbulent Intensity and Turbulent Viscosity Ratio

Where there is background turbulence, a turbulent inflow or other initially turbulent flow, there is need to calculate the turbulence quantities (k, ω, ϵ or otherwise) from some model independent value. A few concepts exist in this area but this work solely uses the turbulent intensity and viscosity ratio which are detailed in this Section.

The turbulent intensity is a measure of the level of turbulent velocity fluctuation. It is expressed as the ratio of the root-mean-square of the turbulent velocity fluctuations, u', to the mean velocity, U, as follows:

$$I = \frac{u'}{U}$$

The turbulent intensity, I, can be equated to the turbulent kinetic energy through the equation below:

$$u' = \sqrt{\frac{2}{3}k}$$

and thus for a given turbulent intensity, the value of k can be found with Equation (2.39).

$$k = \frac{3}{2}u'^2 = \frac{3}{2}(IU)^2 \tag{2.39}$$

Typical values for the turbulent intensity very depending on the flow and certain simulations can be very sensitive to its value. As a rough guide, values less that 1% are considered low levels of turbulence, 5% is a medium level of turbulence and up to 20% is a high level. It is rare for inlet turbulent intensity to be higher than 20%, although this may be exceeded in some flow regions away from the inlet. Due to the rotating machinery and typical operating environment of rim driven thrusters, a turbulent intensity of 10% is used throughout this work.

To finish defining the turbulence quantities, a second parameter needs to be estimated, in this case the turbulent viscosity ratio is used. This is the ratio of the effective turbulent viscosity, μ_t, to the fluid dynamic viscosity, μ. For this work, the value to the ratio is estimated to be 0.1 or 10%, thus the value of ϵ or ω can be calculated from rearranging Equations (2.20) and (2.24) into (2.40) and (2.41) respectively.

$$\epsilon = C_\mu \frac{k^2}{\mu_t} = C_\mu \frac{k^2}{0.1\mu} \tag{2.40}$$

$$\omega = \frac{k}{\mu_t} = \frac{k}{0.1\mu} \tag{2.41}$$

This completes the necessary background to computational fluid dynamics and RANS turbulence modelling.

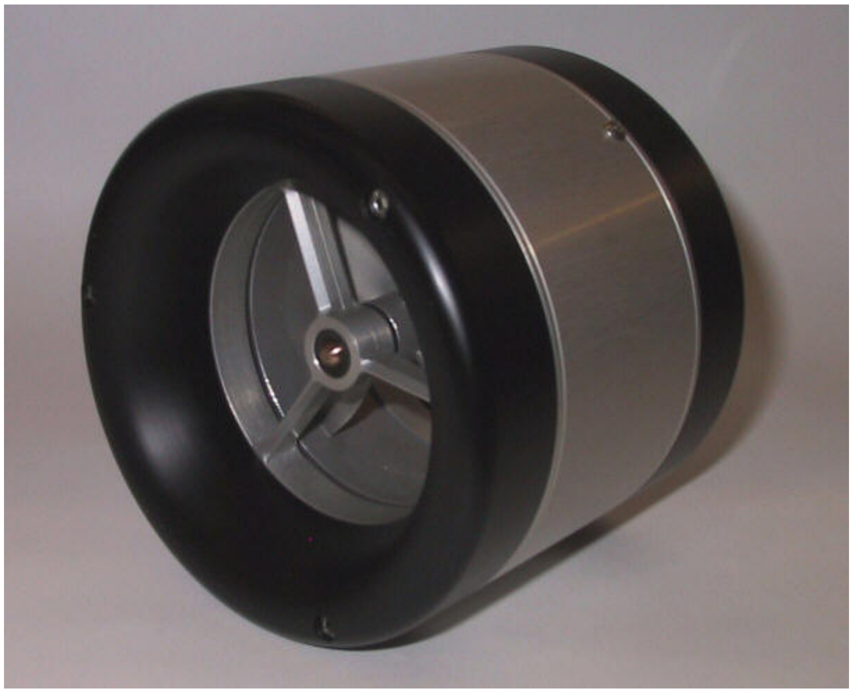


Figure 2.2: Picture of the IntegratedThruster™ from TSL Technology Ltd.

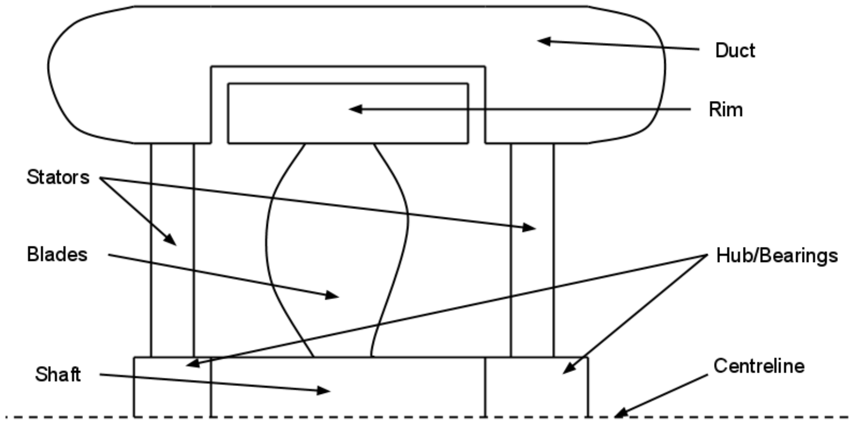


Figure 2.3: Diagram detailing the anatomy of a rim driven thruster.

Chapter 3

State of the Art in Design Optimisation and Numerical Analysis of Propeller Performance

In order to place the present work in the context of current research, the following section outlines the techniques used and results of recent publications. So as to not exclude potentially useful papers from the literature search, it is inclusive of ducted and unducted propulsors as well as hydraulic turbomachinery such as pumps and turbines. Although these machines may be designed for a different purpose, they are very similar in terms of Computational Fluid Dynamics (CFD) simulation methodology.

3.1 The State of the Art in Computational Fluid Dynamics of Hydraulic Turbomachinery and Propulsors

Using a computer to predict the performance of a device is not a recent idea. Caster (1973) made predictions of ducted propellers using a combination of linearised duct theory, Lerbs' moderately loaded propeller theory, and Hough and Ordway's approach to computing induced velocities. The interaction between the duct and propeller flows was considered via a velocity coupling which was iterated until the propeller inflow (induced and freestream) did not change. Despite this method admittedly neglecting centrifugal force, slipstream contraction and the influence of the duct on the propeller wake, the agreement with experimental data was reasonable and a gain in efficiency of 10% in the design condition was achieved.

More recently, Salvatore et al. (2009) presented a good overview of the state of the art by comparing seven different computational models' ability to predict cavitation on an INSEAN E779A propeller. The simulations compared by Salvatore et al. comprised one Blade Element Method (BEM) code, five Reynolds-Averaged Navier-Stokes (RANS) codes and one Large Eddy Simulation (LES) code. For non-cavitating flows, all methods are found to produce results with 1.2% of the experimental values, though it is worth noting that BEM suffered from underpredicting the torque and that LES achieved the best results. When cavitation modelling is introduced it is found that all methods overpredict the cavity extension and Salvatore et al. questioned the reliability of current CFD models' prediction of dynamic cavitation effects such as pressure fluctuations, noise and erosion. While LES was shown to give the best results here, given the similar performance of simpler models, the computational expense of resolving the large turbulent eddies is perhaps not justified.

Perhaps the most ubiquitous methods for propeller performance prediction are those that can be classified as boundary element methods, not to be confused with blade element methods, which is a subset of the former. Occasionally these are referred to as mesh-free methods, but this is a slight misnomer as surfaces still need to be 'meshed', though the fluid domain itself is not meshed which is where the name arises. The prevalence and attraction of boundary element methods is their lower solution times which enabled them to be computed when computational resources were less abundant than today (Caster, 1973). More recently, Benini (2004) showed the relevance and accurate performance prediction of blade element methods for light and moderately loaded propellers. Also, even in current work, they are attractive where the propeller is not the main focus of the research or when solution time needs to be minimised for a design optimisation study.

For investigating the maximum efficiencies of various marine propulsors, Brockett (2003) used an inviscid lifting-line representation of rotors coupled with a panel method for hub and duct surfaces. Viscosity was accounted for by Brockett through an empirical method, presumably to keep solution time down as a large number of cases were investigated. Similarly, Pashias et al. (2003) used a three-dimensional surface panel method to investigate design cases of a rim driven thruster with a rotationally symmetric blade section designed using a two-dimensional viscous-inviscid interaction solver (Drela, 1989). Despite the relative simplicity of the CFD method employed, validation against an open water propeller showed good agreement and an improvement of 5% in bollard pull efficiency over the initial design was reported. Celik and Guner (2007) used a lifting line theory for the design of stators or guide vanes and compared the results to RANS solutions. Though an improvement of 5%

to propulsive efficiency was reported with the addition of stators, the simulation did not report any substantial verification or validation, which does not lend confidence to the results.

By adding a user defined function into a commercial code, Phillips et al. (2008) modelled the effect of a propeller as an actuator disc using blade element momentum theory to extract the performance coefficients of the propeller. This allowed Phillips et al. to investigate the flow over the body of an AUV (Autonomous Underwater Vehicle) using RANS without devoting too much computational time to calculating the propeller flow. Similarly, Choi et al. (2010) modelled a hull flow using RANS with the propeller disc flow solved using a potential flow solver. Validation against experimental data showed that this method had at most a 5.7% error in resistance (drag) and Choi et al. concluded that there is scope to apply computational methods at the initial hull-form design stage but enhanced accuracy is desirable. It is possible that a more accurate result could be achieved if RANS were also used to model the propeller, but this would be at the expense of longer solution time, which is not desirable in the initial design stages. Also, Berger et al. (2011) investigated propeller-hull interaction by using a boundary element code for the propeller and a commercial RANS code for the hull. This work differed from the aforementioned in that unsteadiness is included using a time step of three degrees of propeller rotation. The novel coupling method presented is shown to drastically reduce computation time but even Berger et al. concluded that the results are not completely satisfactory.

A design study of contra-rotating propellers was performed by Koronowicz et al. (2010) using lifting line and lifting surface methods. An iterative procedure was used to account for the effect of the forward propeller on the aft propeller and *vice versa*. A pseudo-unsteady simulation was performed including a unsteady sheet cavitation bubble model with a non-uniform inflow and the results concluded that, in some conditions, contra-rotating propellers can be better than single propellers and consequently should be considered when doing a design study. Yakovlev et al. (2011) used a two-dimensional foil lattice method for the design optimisation of a rim driven propulsor. An increase in propulsive efficiency of 0.4% was reported even though the model was very simplified to make it quick to calculate for optimisation. Similarly, Zeng and Kuiper (2012) used a lifting surface method to perform the design optimisation of a propeller with an objective of a maximum cavitation inception speed (*id est* delaying the onset of cavitation as much as possible). The optimisation was performed using a genetic algorithm, which is a very expensive method in terms of number of function evaluations, so the speed of boundary element methods was necessary here. However, the preliminary results reported do show an improvement in cavitation inception behaviour.

Blade frequency noise prediction is usually calculated in the frequency domain, however, Ye et al. (2012) took advantage of the speed of potential based surface panel methods to perform a time domain prediction of a cavitating propeller. By doing this, it was found that cavitation noise attenuates more slowly than non-cavitation noise. A relatively large time-step of 3 degrees of propeller rotation does bring in to question the maximum frequency resolution, but the lower frequencies were most likely to have been captured well.

Recent innovations to boundary element methods try to increase the physical modelling, to include flow vorticity for example, while still preserving the mesh-free character of the solution. One such work is that of Zhu et al. (2012) where a Lagrangian vortex method is outlined to include vortices in a panel or boundary element method, though it could also be used within a finite volume method. The method was applied to simulating unsteady flow in hydraulic turbines, but the results had a greater than 10% error in off-design conditions. Zhu et al. make an interesting point that no turbulence model accounts for centrifugal or Coriolis forces (also known as first and second Coriolis forces), however these forces are only an artefact of angular momentum conservation in a rotating reference frame (Coriolis, 1935) and do not manifest in the inertial reference frame in which most unsteady simulations are performed.

Further extensions to boundary element methods and blade element momentum theory were performed by Leone et al. (2013) to apply the methods to self pitching propellers. The results obtained for blade element methods were accurate for a pitch ratio of 0.8 but significantly worsened at higher pitch ratios up to an error of approximately 20%. Leone et al. found that panel methods performed slighty better with regards to performance prediction accuracy.

It is uncommon to find reports of the Euler equations being solved numerically using volume meshed methods, as they provide little benefit over boundary element methods. However it is a valid solution method and included here for completeness. An interesting result was achieved by Bousquet and Gardarein (2003) when trying to model unsteady propeller-wing interactions using a finite volume method solving the (inviscid) Euler equations. Bousquet and Gardarein used fourth and second order discretisations to achieve a high accuracy and good agreement with pressure distributions in the validation case, but the integral normal force coefficient differred from experimental values by up to 15% in the unsteady simulation. Carcangiu et al. (2011) investigated the design of an urban vertical axis wind turbine using a commercial finite element method but, as the investigation was focussed on the inlet and outlet to the turbine, the rotating part of the impeller was neglected in the model.

A more popular method for solution in investigations where the propulsor is of primary interest is to solve the RANS equations. This is most likely because, with modern computing power, it presents the best trade off between solution time and physical accuracy, though the turbulence effects still remain modelled rather than simulated. Rhee and Joshi (2005) investigated the solution of a marine propeller flow using RANS with a validation against experimental results. A commercial solver, FLUENT, was used and the verification study found that domain size is not as important as mesh resolution, which is in contrast to the author's findings in Section 5.1 where domain size is found to be important. With a total axial length of 1.22 propeller diameters and radial diameter of 2.86 propeller diameters, the domain used by Rhee and Joshi is smaller than the size recommended in Section 5.1, and this is likely to be a contribution to the errors of 8% in thrust and 11% in torque reported, though Rhee and Joshi conjecture that the errors are due to insufficient mesh resolution in the hub area. Using the same commercial package, Lam et al. (2006) investigated the propeller wash at the bollard pull condition. A structured grid of approximately 71,000 cells was used but rotational symmetry was exploited such that this equates to a full propeller mesh of approximately 213,000 cells. For validation, the results were compared against laser-doppler anemometry (LDA) data, however discrepancies were found. Also the results for the structured and unstructured meshes differed significantly enough that it was concluded that further investigation was needed.

A better result was achieved using FLUENT by Da-Qing (2006), where thrust and torque were reported to be within 3% and 5% of experimental values respectively. Although this work highlights the potential for integral values to give a misleading representation of accuracy, as examining pressure distributions shows discrepancies around the blade tip area. An interesting validation method was also used here of comparing pictures of a paint test with skin friction contours as well as comparing particle image velocimetry (PIV) measurements of vorticity in the wake which did show a good agreement. Huang et al. (2007) also used the popular commercial code of FLUENT to investigate the effects of thrust fins on thrust efficiency. Although the mesh used does not look completely accurate in its representation of the geometry, this discretisation error doesn't appear to have a large effect as thrust and torque profiles match well with experimental data.

The work presented by Li and Wang (2007) on investigating the effect of the axial gap between inducer and impeller of a three blade axial pump is interesting in its quasi-unsteady methodology. The device investigated comprises a six bladed inducer, three bladed impeller and 11 vaned diffuser which denies any rotational symmetry and the unsteady interaction is represented by a sequence of seven steady state simulations. This method seems to produce good results in this case and comparison of velocity profiles shows no significant differences. However, it is not a true unsteady method and Li and Wang admitted it does not consider the unsteady flow features between impeller and diffuser. Conversely, Petit et al. (2009) were less convinced that using a steady-state frozen rotor formulation yields good results and concluded that it is not accurate enough for rotor-stator interaction due to improper treatment of the impeller wakes. Although this finding was with a model of a centrifugal rather than axial pump, the hypothesis would stand for axial flow devices too.

A prediction of the effective wake and performance of a rim driven tunnel thruster was made by Kinnas et al. (2009), using FLUENT to achieve a RANS solution, and then combining this with a vortex-lattice method to predict cavitation. Some results achieved had a large 50% error between experimental and computational results, but results closer to the design condition had a smaller error. However, cavitation prediction was consistently close to experimental observation. A performance assessment of ducted propellers by Funeno (2009) using RANS gave a recommendation of suitable boundary conditions for this simulation of a combination of a velocity inlet, pressure outlet and slip walls. Again, use of a frozen rotor formulation here made design difficult due to the interaction of nozzle, propeller and gear housing flows.

Contrary to other recent works (Phillips et al., 2008; Choi et al., 2010; Berger et al., 2011), Abramowski et al. (2010) used a full RANS formulation for the propeller when investigating the effect of different hull forms on propulsive efficiency. The accuracy, compared with experimental results, was very high at advance ratios greater than 0.2 but was found to be lower at low advance ratios. This was attributed to possible geometrical differences, but an alternative explanation is offered in Section 5.2.2 herein. Vesting and Bensow (2011) chose to use RANS coupled with a vortex-lattice method to predict cavitation in an attempt to optimise the design of a propeller blade with respect to both propulsive efficiency and cavitation performance at the same time. Unfortunately no comparison with experimental data is made and the reported improvement of 8.5% in propulsive efficiency cannot be taken as definitive, though it is likely that there was some improvement on the original design if the measurement errors were effectively systematic in nature.

Good agreement with experimental particle image velocimetry (PIV) measurements was achieved by Liu et al. (2012a) using RANS to investigate the internal flow of a two-bladed centrifugal pump. This result was calculated using a first order Gaussian upwind discretisation scheme as the initially selected total variation diminishing (TVD) scheme was diverging. The cavitation performance of ship propellers was investigated by Zhu and Fang (2012) with good agreement with experimental results except at low advance ratio. It is possible that the poor results at low advance ratio are due to poor prediction of the cavitation and the consequential impact of this. An axial flow water turbine with a similar motor construction to a rim driven thruster was analysed by Wang et al. (2012) with a conclusion that a nozzle and diffuser can inrease the pressure drop across a turbine and thus extract more power. Finally, Cao et al. (2012) used RANS for the solution of a rim driven propulsor but neglected the stators in the model to remove the need to simulate unsteady rotor-stator interaction.

The increasing popularity and power of RANS methods is shown in recent research, where a study into the effect of rake angle by Hayati et al. (2012) had near perfect agreement with experimental results up to an advance ratio of 0.6. It is also arguable that more insight gained into the physics behind the trends observed than would have been achieved had the study used boundary element methods. Hayati et al. (2013) modelled the propeller-hull interaction, similar to that of Phillips et al. (2008), but using RANS for the entire flowfield. This work highlighted the significant difference between open water tests of marine propellers and that of their in-service performance interaction with other flows and under off-design conditions such as vehicle pitch.

Few studies have directly compared results of steady RANS and unsteady RANS (URANS) as very often there are not any significant unsteady flow features to capture so as to justify the resource investment of an unsteady simulation. However, in comparing RANS and URANS methods for marine propellers, Kaufmann and Bertram (2011) did not find RANS to be categorically worse than URANS. The results clearly showed that, when using multiple reference frame (MRF) method in RANS, the size of the rotating reference frame zone has a significant (10%) impact on results. Also, the findings for the URANS simulations showed a clear dependency of results on time step, consequently Kaufmann and Bertram recommended a time step of one degree of propeller rotation as a suitable trade-off between accuracy and computation time. The ability of OpenFOAM to produce URANS results as accurate as those calculated by FLUENT was documented by Muntean et al. (2009). This improves the confidence in having no benefit in accuracy from commercial software, or the corollary that the use of OpenFOAM does not penalise the potential accuracy of results. Petit et al. (2009) made a recommendation in preference of URANS due to the significantly unsteady nature of the rotor-stator interaction in a centrifugal pump and consequently achieved better results, although the velocity profiles are only qualitatively similar to experimental measurements and suffer some quantitive error.

Unsteady methods are often not employed when a steady method will suffice as they require temporal resolution and thus more computation time. However, very often propellers are acting in non-uniform wakes, usually due to the hull ‘shadow’, and thus the blades see a time-varying inflow as they undergo one rotation. This is the most common reason for employing unsteady methods in the investigation of propellers, though often large eddy simulations (LES) are preferred to URANS. However, Liu et al. (2012b) utilised URANS for investigating cavitating flows around skewed propellers using a measured wake for the inflow boundary condition. Liu et al. concluded that a skew angle of 20 degrees is best for minimum cavitation. In this work URANS allowed the use of relatively coarse grids of 700 thousand and one million cells as would be an acceptable resolution in a steady RANS simulation, whereas LES is likely to require a greater mesh density to resolve large eddies in the flow.

Morros et al. (2011) performed a URANS simulation of a centrifugal turbine to evaluate the risk of fatigue failure and found a 25% force fluctuation with a good confidence as results were within 2% of experimental data. Also investigating a turbine, though axial rather than centrifugal, Lloyd et al. (2011) used a URANS formulation though the necessity for unsteady modelling in this case is not clear.

To perform a transient analysis of a single channel pump, Auvinen et al. (2010) used URANS with a pre-solution from RANS and a large time-step preconditioning phase to prepare the solution prior to transient analysis. Validation against laser doppler velocimetry (LDV) measurements showed a varying error from 4% to 10% though it is conjectured that this is due to simplification of the computational model. Similar to the findings of Kaufmann and Bertram (2011), a high sensitivity to time-step was observed. An axial flow pump was analysed using URANS by Zhang et al. (2010) with a maximum error, against experimental measurements with a five hole probe, of 4.54%. A timestep of three degrees was used which, given the results of Kaufmann and Bertram (2011), may have been the major contributing factor to the error. Although the qualitative findings of increased pressure fluctuations towards the blade tip and small amounts of pre-rotation are probably still valid. In order to predict the vortex rope in a swirl flow generator, Petit et al. (2010) used URANS and validated against laser doppler velocimetry (LDV) velocity profiles. Visualisations of the vortex rope looked qualitatively good, however tangential velocities were underestimated, which was possibly due to using first order discretisation schemes.

It seems more popular for propeller flows, particularly with respect to simulating cavitation, to use large eddy simulation (LES) as many find it is the best available method to reproduce all the flow features of interest. However, a conclusive method is yet to be developed for cavitation, as using LES to find cavitation around a twisted hydrofoil, Lu et al. (2010) found cavity extent was underpredicted and the numerical simulation was unable to predict the cavity collapse. Results from Di Felice et al. (2009) using LES to model a submarine propeller flow were not much more accurate than those from RANS and URANS methods with results for thrust and torque around 5% of experimental values. Although a rather low resolution (for LES) mesh of 4.467 million cells was used which is perhaps the reason the results are not more accurate. A close agreement between the CFD and LDV measurements of velocity fields suggest that the modelling was suitable even if the accuracy was not perfect. The LES performed by Liefvendahl et al. (2010) of a submarine propeller and also propeller-hull interaction used a more substantial mesh of up to 13 million cells. As the rotation was modelled using mesh deformation methods, 18 topologically different meshes were generated, corresponding to 20 degrees of rotation each, so as to preserve mesh quality as cells deformed. Liefvendahl et al. failed to exploit the rotational periodicity of the meshed seven bladed propeller as the same mesh can be used every 51.42857 degrees of rotation, thus reducing the number of topological meshes required or alternatively increasing the mesh quality with the same number of topological meshes. It was found that the thrust on each blade fluctuates by approximately 20% through each rotation. Bensow and Bark (2010) investigated dynamic cavitation using implicit LES and found an overprediction of the cavity extent but with good shape and location and good agreement with pressure distributions. Although sufficient discrepancies between numerical and experimental results were reported that the method is not conclusively a good one. However, Alin et al. (2010) performed a comparison of RANS, DES and LES for solution of flow around a submarine and found that LES had the best agreement with experimental results and captured more flow features. It is likely that the subgrid model plays a key role in the accuracy of LES and it is not surprising that implicit LES, where the subgrid model is the truncation error in the discretisation, has not reported very good results in the literature.

The final option, that is not frequently utilised in the literature, is to use detached eddy simulation (DES) which exploits the capabilities of LES where the mesh resolution is sufficient to resolve the larger eddies but switches to a URANS formulation in regions where it is not. This enables DES to work on grids without the significant requirement for near-wall resolution that LES has, attempting to provide the advantages of both LES and URANS in one method. Kornev et al. (2011) developed a hybrid URANS-LES model for the ship stern area from tests from a significant number of URANS and LES models. These include the linear k-ϵ, non-linear k-ϵ, k-ω SST and $k\epsilon v^2 f$ models from URANS and the simple Samgorinsky, dynamic Smagorinsky and the dynamic one-equation eddy models from LES. The best results in terms of accuracy and numerical stability were achieved with a combination of the k-ω SST and dynamic Samgorinsky models. The other interesting finding reported by Kornev et al. is that instantaneous wake velocities deviate sufficiently from the mean values to negatively influence the accuracy of propulsion and unsteady load predictions made with a mean wake assumption.

For the most accurate simulations and most resolution of flow features that are achievable with current computational capabilities, the literature confirms LES as the leading tool, which is expected as it involves the least modelling out of the above methods. However, the additional computational expense, in both spatial and temporal resolution, required for an accurate LES preclude it from being useful as a method where rapid design iteration is important (for example in design optimisation). It is also shown in the literature that RANS simulations have been used to a great degree of accuracy and, as it is substantially quicker to solve, would be preferrable provided it can reach the desired level of accuracy when simulating a rim driven thruster. Although boundary element methods or blade elements methods would be the best option in terms of speed of solution, their lack of accuracy in highly loaded and off-design flow conditions render them unsuitable for accurately capturing the entire design space in a design optimisation study.

3.2 Turbulence Model Selection

Selecting a turbulence model is not a simple task as all models have their benefits and drawbacks in accuracy and stability when solving different flow features. An editorial on RANS modelling by Spalart (2009) discussed the state of the art in RANS turbulence modelling which has stagnated somewhat since 1992. Prediction of transition to turbulence by turbulence models is still a weak point in all models, which was also raised by Batten et al. (2009) where the transition point is found to 'creep' upstream. Corson et al. (2009) also found transition to be a weakness of the Spalart-Allmaras turbulence model as well as massive separation, unsteady flow and near wall modelling. Conversely, Menter (2009) showed a good performance of transition prediction, with the k-ω SST model, as well as good performance in predicting boundary layers with adverse pressure gradients. However, due to the high Reynolds number operation of marine propulsors, transition is not a critical part of the flow and the limitations of transition prediction are not overly concerning.

For the modelling of marine flows, there are conflicting results on the most suitable turbulence model. Zhang et al. (2006) stated a preference for k-ω SST for ship wake flows and others have also chosen to use it (Da-Qing, 2006; Funeno, 2009; Berger et al., 2011; Kaufmann and Bertram, 2011; Lloyd et al., 2011; Cao et al., 2012), although others chose to use RNG k-ϵ (Huang et al., 2007; Abramowski et al., 2010; Zhu and Fang, 2012) as it is supposed to be the best for swirling flows. In investigating the prediction by standard, Realizable and RNG k-ϵ models of the flow past an inclined flat plate (*confer* flow past a blade section), Castelli et al. (2012) found the Realizable k-ϵ model, when combined with standard wall functions, to be the most accurate. However at low angles of incidence (9 degrees), the results of Castelli et al. show the RNG k-ϵ model to be the most accurate. Also used in literature is the standard k-ϵ model (Choi et al., 2010; Liu et al., 2012b) and was found to be the closest to experimental measurements when compared to standard k-ω and RNG k-ϵ models by Lam et al. (2006).

For centrifugal and axial pump flows both the standard k-ϵ model (Petit et al., 2010; Morros et al., 2011) and the RNG k-ϵ model (Li and Wang, 2007; Zhang et al., 2010) are commonly used. However the standard k-ϵ model is found to produce more accurate results when compared to the k-ω SST model by Auvinen et al. (2010) and both the RNG k-ϵ and k-ω SST models by Liu et al. (2012a) though these are both studies of centrifugal pump flows.

For RANS simulations of ducted and rim driven propulsors, the prevalent model is the k-ω SST model. This model is also the one chosen by Kornev et al. (2011) in a comprehensive attempt to find the best URANS and LES models for the ship stern area. A preference and better accuracy has been shown for the RNG k-ϵ model in some marine propulsion cases due to its suitability for swirling flow. Hence it is decided that for the numerical modelling of a rim driven thruster either the RNG k-ϵ or k-ω SST turbulence models are the most suited for the purpose.

3.3 Ducted and Rim Driven Propulsors

As there as many similarities between ducted and rim driven propulsors, many of the design and experimental findings for the former are still valid considerations when designing rim driven thrusters. English and Rowe (1973) found that while ducted propellers typically have lower open water efficiencies, they improve the hull efficiency and they induce a greater proportion of hull boundary layer fluid, thus leading to a higher wake fraction. They also found that steerable ducted propellers allow for an improved turning circle over conventional propeller and rudder configurations. Another reported advantage of the ducted propeller is thrust loading can be transferred to the duct, reducing propeller loading and increasing cavitation performance. This advantage is also reported by Brockett (2003) who derived, through a lifting-line method, the maximum efficiency of a ducted propulsor of $\sqrt{C_T/J^2} = 4.3$ at an advance ratio of $J = 4.3$. However the effects of the blade tips and clearance region are detrimental to efficiency.

For rim driven propulsors, Lea et al. (2003) stated the advantages of rim drive as increased propulsion efficiency, increased arrangement flexibility, decreased weight and increased harbour maneuverability. Lea et al. found the open water efficiency of a rim driven propulsor to be 4.5% greater than the equivalent hub-driven variant. One advantage found, that rim driven propulsors have over ducted propulsors, is that the rim allows a significant hydrodynamic loading on the blade tip without generating a tip vortex and the associated losses (*ibidem*, Kinnas et al. 2009).

A hubless design of rim driven thruster was tested by Yakovlev et al. (2011)

which has the advantages of not entrapping debris such as cables or fishing lines. However, in terms of blade stresses, Yakovlev et al. found a combination of both a hub and rim reduces the stresses experienced by the blades four fold over a hubless design and six fold over a conventional propeller.

The distribution of loading on the blades of a rim driven thruster was investigated numerically by Cao et al. (2012) who found the maximum hydrodynamic loading occurs typically at the tip. This work simplified the complexity of a rim driven thruster in two ways, first neglecting the stators from the modelled and secondly replacing the flow in the gap between rim and duct with an empirical model. The reported torque contribution of the rim based on these models totals 27% of the total torque losses.

The flow in the gap region features many possible states and as it constitutes part of the motor the design is a trade-off between motor electromagnetic and hydrodynamic efficiency as well as being constrained by machining tolerances. In terms of the physical flow features, the Taylor-Couette flow region has been investigated both experimentally (Batten et al., 2004) and numerically (Batten et al., 2002; Lin et al., 2010). Although, in terms of effect on the device, Lea et al. (2003) did not find an efficiency maximum but found the highest efficiency was with the smallest gap tested.

3.4 Design Optimisation of Marine Propulsors

Due to the complexity of marine propulsors, they do not lend themselves to simple optimisation, but a few attempts with varying success have been made. To improve the bollard pull efficiency of a rim driven thruster, Pashias et al. (2003) optimised seperate components in a sequential fashion. First the two-dimensional blade section was designed to be rotationally symmetric, such that the section shape is the same for both directions of rotation. Then the duct profile was chosen from a selection of designs, followed by the duct length and finally the blade area ratio. Despite the simplicity of this approach, an improvement of 5% in bollard pull efficiency was reported. It should be noted that, like Cao et al. (2012), the stators were neglected from this study as their effect on efficiency was assumed to be less than 1%. Yakovlev et al. (2011) also tried to improve the efficiency of a rim driven thruster using a two parameter optimisation to both maximise propeller efficiency and minimise pressure reduction on the blade surface to minimise cavitation. The results reported here were not a significant improvement as the efficiency increase reported was only 0.4%. A penalty of 30% more pressure reduction was also reported, which would increase the amount of cavitation and is not desirable.

For the optimising the design of marine propellers, Benini (2003) defines a multi-objective method for maximising both the efficiency and thrust of a B-series propeller, subject to a cavitation based constraint. Benini uses a novel fitness function to optimise genetic diversity as well as Pareto optimality for an evolutionary algorithm based optmisation. Conversely, Gaafary et al. (2011) use a single objective function of open water efficiency to select the optimal B-series propeller, choosing to formulate other requirements as constraints imposed by cavitation, material strength and propeller thrust. This allows a more general single-objective constrained optimisation method to be used, for which the example in the paper is a commercial optimiser called LINGO.

Design optimisation of a propeller blade with ten design variables was performed by Vesting and Bensow (2011) using response surface modelling (RSM) and a genetic algorithm to search the output response surface. This method produced a reduction in required power of 8.5% although the noise produced by the propeller was increased. If evaluation of the objective function is sufficiently inexpensive, then a genetic algorithm may be applied directly as done by Zeng and Kuiper (2012) to find a propeller with the highest cavitation inception speed. Preliminary results reported an improvement of inception speed by two knots.

Where the objective function is expensive to evaluate and/or the dimensionality of the problem is high, response surface modelling, also known as surrogate modelling, can be used to represent the true objective with a fitted model. Forrester et al. (2008) makes a good reference for surrogate modelling techniques, particularly favouring one known a Kriging name after Daniel Krige, who originally developed it as a mine valuation method (Krige, 1951). Jones et al. (1998) showed that global optimization of response surface modelling could be improved through the balancing of exploration and exploitation through means of the standard error.

While the body of work on optimization of propellers is not exhaustive, the literature suggests that even with the simplest of computational fluid dynamics models, there are improvements to be found. Multi-objective optimisation has been found as the least productive of approaches reviewed here, with response surface modelling yielding the best improvements and perhaps the best method to apply to rim driven propulsors. Due to the relatively long evaluation times in the chosen analysis method, that is RANS simulation, the best way to optimise the performance would be through response surface modelling, in this case Kriging is chosen, with a single objective function that can be rapidly searched using a genetic algorithm.

Chapter 4

Computational Fluid Dynamics Methods

There are a number of steps in creating computational fluid dynamics simulations, each being typically performed with separate, specialised pieces of software. However, this is not always necessarily the case as some CFD packages are available that provide all the necessary tools to generate geometry, mesh the domain, solve the equations and post process the results. In this project, open source software is used in preference to commercial codes where possible, primarily to avoid problems with licence contention, but also because open source allows complete interrogation and modification of the code. A number of different programs are used to create, solve and post-process the simulations, but the majority of applications come from the software package OpenFOAM (Open Field Operation And Manipulation), an open source collection of utilities and solvers for computational fluid dynamics. However, for geometry generation, SolidWorks, Visual Basic and Python were used. Meshing was subsequently handled by blockMesh and snappyHexMesh, two meshing utilities provided in the OpenFOAM package, and steady state solution was also performed with the OpenFOAM solver MRFSimpleFoam. Post-processing was primarily performed using ParaView, an open-source, multi-platform data analysis and visualization application that is bundled as third party software complimentary to OpenFOAM. Python was also used for some post-processing (graphs) and scripting purposes. The complete solution procedure is outlined in Figure 4.1 and further details of the simulation process, and the role of each package in the methodology, is described in the following sections.

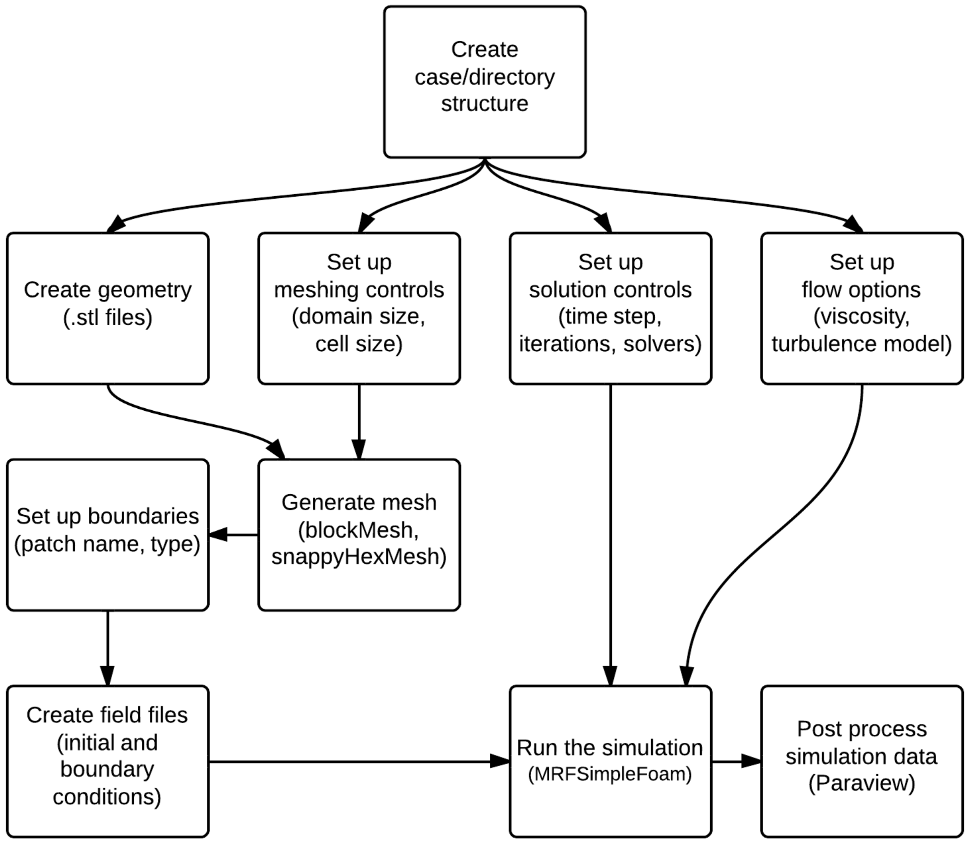

Create case/directory structure
Create geometry (.stl files)
Set up meshing controls (domain size, cell size)
Set up solution controls (time step, iterations, solvers)
Set up flow options (viscosity, turbulence model)
Set up boundaries (patch name, type)
Generate mesh (blockMesh, snappyHexMesh)
Create field files (initial and boundary conditions)
Run the simulation (MRFSimpleFoam)
Post process simulation data (Paraview)

Figure 4.1: Outline flowchart of the solution procedure.

4.1 Geometry Creation

Two methods were used to generate the propeller geometry definition. Both methods ultimately generated a sterolithographic (.stl) format file that is required by the meshing program snappyHexMesh. The first method used a proprietary Visual Basic macro, provided by TSL Technology Ltd., to automate SolidWorks to generate the propeller surface whereas the second method was written in the Python language and was designed so that the geometry creation can be automated.

4.1.1 Propeller Surface Visual Basic Macro

The Visual Basic program for propeller surface generation took details of the propeller section and distributions of section, thickness, chord, pitch and rake along the radius from a Microsoft Excel spreadsheet containing the geometrical data provided by TSL Technology Ltd. With these inputs the program then automatically provided a lofted propeller surface in SolidWorks that was then converted into a solid body. This program only provides a single blade, therefore it is necessary to repeat the solid body in a circular pattern and include a propeller hub. Following this, the resulting geometry is exported into .stl format for reading into snappyHexMesh.

4.1.2 Automated Geometry Generation

While the Visual Basic program developed by TSL Technology Ltd. provides all the necessary functionality for generating the propeller blade geometry, it was decided that the workflow could be improved by developing a geometry creation method in Python. The advantages to writing a new geometry creation method are two-fold, first Python is available on the GNU/Linux platform that OpenFOAM is designed to run on, preventing the need to switch from Microsoft Windows to GNU/Linux to run SolidWorks and OpenFOAM, respectively. Secondly, removing the need to switch platform and using a scriptable language such as Python allows for the easy automation of the entire simulation process, speeding up the time it takes to evaluate different design iterations.

The programming language of Python was chosen over other languages for a number of reasons, many of which are not specific to this application. As a high level language, it benefits from quicker development time, primarily due to the readability of the code enforced by its syntax rules, but also attributable to rapid debugging at the interpreter prompt. A trade-off for this lexical benefit is that Python will not run as fast as perhaps a compiled language might, but the computational time of the task of automation is small compared to the time taken to write the code, thus it is more efficient to minimise writing (and reading) time than to minimise the relatively small run time. Other advantages of Python are that it is free, popular and has many libraries for a large variety of scientific tasks. It also supports multiple programming paradigms for improved flexibility and because everything in Python is an object, it is easy to program in a modular, re-usable fashion.

There are two parts to the automated geometry generation process, the first is to generate data to represent the geometry (Section 4.1.2) and the second is to store those data in a format that is readable by the computational fluid dynamics code (Section 4.1.2).

Blade Co-ordinate Generation

To generate the blade geometry as a set of co-ordinates, a set of Python functions were written and are listed in Appendix A. The primary function for generating the blade geometry is `bladegen` which takes as arguments functions for the blade section, chord, thickness and pitch as well as the number of degrees of rake. The `bladegen` function then calculates the co-ordinates of the blade at 15%, 20%, 25%, 30%, 40%, 50%, 60%, 70%, 80%, 90% and 100% of the radius and returns them. This results in a set of three dimensional curves, which can be lofted to produce a solid blade surface.

The process of generating the co-ordinates begins with the non-dimensional sections of the blade being placed at each radial station in the x-z plane, with the y axis forming the radial direction of the blade. Next, the section is scaled to the correct chord and thickness, by multiplication of the values returned by the chord and thickness functions. It should be noted here that the non-dimensional section should be specified such that the origin (0, 0) coincides with the generator line of the blade. The pitch is applied next by calculating the rotation angle and rotating the x-z plane about the y axis. Then the rake is added by translating the z co-ordinates according to their radial position to give the required rake angle. Finally, the co-ordinates, which lie on planes of constant y value, are mapped onto cylindrical planes of constant radius, such that the radial slices are consistent with the propeller definition.

To enable the program to be used in an extensible and modular fashion, the program is designed to take pitch, chord and thickness in the form of functions of radial position and diameter. The advantage of this is that any form of function can be specified, without changing the underlying code, to give constant, linear, quadratic and any other distribution of the geometry variables.

To validate the output of the blade geometry generation program, the blade contours produced by the Python program and the Visual Basic program provided by TSL Technology Ltd. were compared and found to match. Visualisations of the geometry were also compared to pictures (Gerr, 2001; Carlton, 2007) and propeller diagrams in Kuiper (1992) and there were no perceptable differences between them, thus validating the output of the blade geometry generation program.

STereoLithographic (.stl) File Format

The OpenFOAM meshing utility snappyHexMesh uses .stl (stereolithographic) files as the input of the geometry. These files specify a geometry through a number of triangular facets, consisting of three vertices given by cartesian co-ordinates and a face normal vector. The specification for the file format allows two different types: ASCII, which is written in characters, and binary, which is stored in a numerical format that is not human readable. As snappyHexMesh only reads ASCII format .stl files, the binary .stl format will not be considered further.

The .stl file format opens with a line to describe the solid by name:

```
solid <name>
```

and similarly the file must finish with a closing statement:

```
endsolid <name>
```

In the enclosed region between each of these statements, the triangular facets that make up the solid surface are listed. For each triangle with a unit normal vector given by `<ni> <nj> <nk>` and vertex co-ordinates given by `<v1x> <v1y> <v1z>`, `<v2x> <v2y> <v2z>` and `<v3x> <v3y> <v3z>`, the resulting code for this facet is as follows:

```
facet normal <ni> <nj> <nk>
    outer loop
        vertex <v1x> <v1y> <v1z>
        vertex <v2x> <v2y> <v2z>
        vertex <v3x> <v3y> <v3z>
    endloop
endfacet
```

where each number is given in floating point format.

To generate .stl files from co-ordinate data, a series of functions were written in Python and are listed in Appendix B. As the geometric data in blade generation is predominantly based on lofting profiles, that is interpolating through a series of two dimensional profiles, the functions were written to either generate a surface between two profiles (in 3D space) or to fill in a profile so as to close the lofted surfaces at either end.

The function `oneface` takes a set of co-ordinates and returns a string representing the co-ordinates as a face in .stl format. This is done by first splitting the co-ordinates into an upper and lower set. Triangles are then added to the .stl string in a pairwise fashion between co-ordinates from the upper and lower sets. It should be noted that for the best representation of a blade section, the upper and lower sets of co-ordinates should be split at the leading and trailing edges, thus best preserving the blade curvature. However, very often the final blade section will be beyond the intersection of blade and hub, or blade and rim, and consequently its shape is not of utmost importance.

The lofting between two three dimensional lines is provided by the function `twoface`, which takes two sets of co-ordinates, one for each line, and returns a string that represents the face lofted between these two lines. This function requires an equal number of co-ordinates for both lines, as the triangles are created in a similar pairwise fashion as the function `oneface`. Another caution required is that both sets of co-ordinates are given from the same reference point, *id est* leading or trailing edge, and also progress in the same direction, otherwise the produced loft will be skewed.

By combining the strings from subsequent `oneface` and `twoface` outputs, `writestl` finishes the formatting of the .stl files with the correct header and footer and writes the result to an ASCII .stl file ready for reading in to snappyHexMesh.

4.2 Meshing for Steady State Simulations

All the meshing in this work was performed using the programs blockMesh and snappyHexMesh, which are open source mesh programs and both part of the OpenFOAM software package. The process of mesh generation using these programs begins with the generation of a baseline hexahedral mesh using the program blockMesh. It is at this point where both the extent of the computational domain and the largest cell size must be chosen. The baseline mesh defines the largest possible cell size (base size) as snappyHexMesh will only split cells, not combine them. Initially, to make sure the computational domain was sufficiently large, it was designed to extend six propeller diameters in the radial and upstream directions and 12 propellers diameters in the downstream direction. The minimum required computational domain size was later investigated as part of the verification procedure in Section 5.1. The base cell size was selected to be a cube with a side length of one quarter of the propeller diameter, 17.5mm.

Once the base mesh was generated with blockMesh (Figure 4.2a), the program snappyHexMesh was used to finish the meshing procedure, as blockMesh is not well suited to complex geometries such as that of marine propulsors. There are three distinct stages in the meshing procedure with snappyHexMesh, first a castellated mesh is produced (Figure 4.2b), then the castellated mesh is fitted (snapped) to the surface (Figure 4.2c) and finally a layer mesh is grown on the surface (Figure 4.2d).

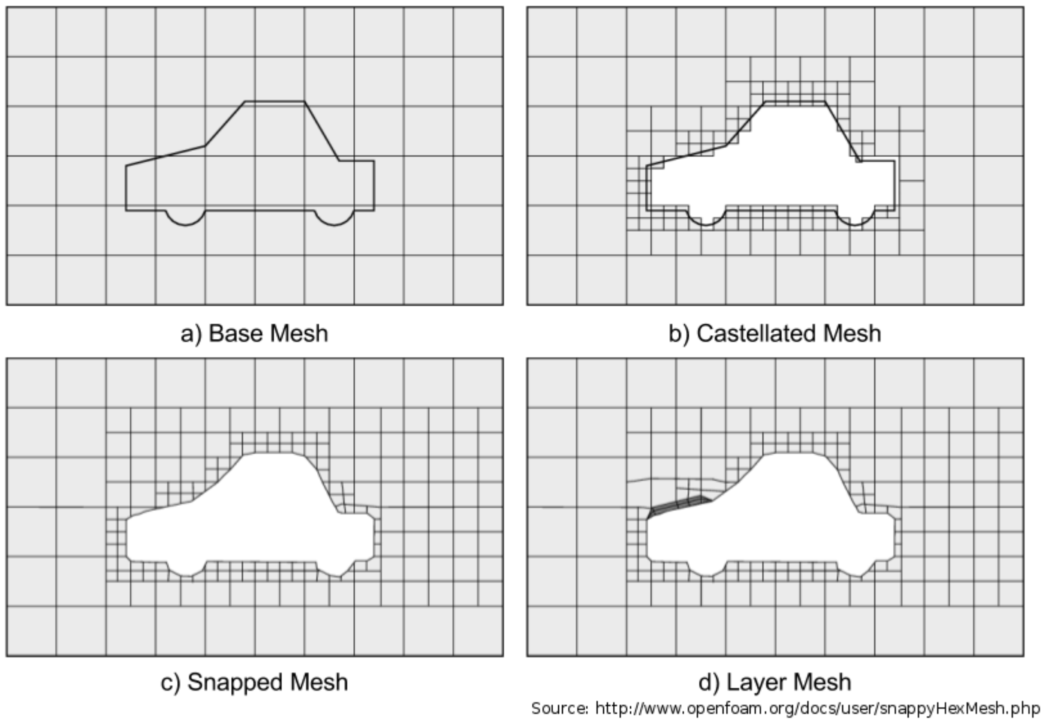


Figure 4.2: Outline of the meshing procedure in snappyHexMesh.

The surfaces used in snappyHexMesh must be provided in .stl format which, after reading the surfaces in, begins by refining the base mesh next to surfaces and regions specified by the user by splitting cells. This is done non-discriminantly, therefore cells that ultimately end up being removed, because they are 'inside' the propeller, are also refined. Once the refinement stage has finished, snappyHexMesh searches out all the cells in the fluid domain from a location specified by the user and removes any unreachable cells to leave a refined castellated mesh. It would be more memory efficient to remove 'interior' cells after each stage of refinement but it would be less time efficient.

The next stage of the meshing process is the snapping stage after which snappyHexMesh is named and involves moving vertices such that they lie on the surface but without violating any of the mesh quality constraints defined by the user. The application of the refinement and snapping stages to a Wageningen B4-70 propeller are clearly visible in Figure 4.3 with the final resultant mesh shown in Figure 4.4. It is noted here, as it can be clearly seen in Figure 4.4, that the tip of the Wageningen B4-70 propeller is difficult to recreate using the geometry method of Sections 4.1.2 and 4.1.2 as at the tip, both the thickness and chord become zero. Therefore, to generate this open water propeller geometry, the tip is cut at 99% of the radius to produce a viable final section.

Depending on the complexity of the geometry and the level of mesh quality

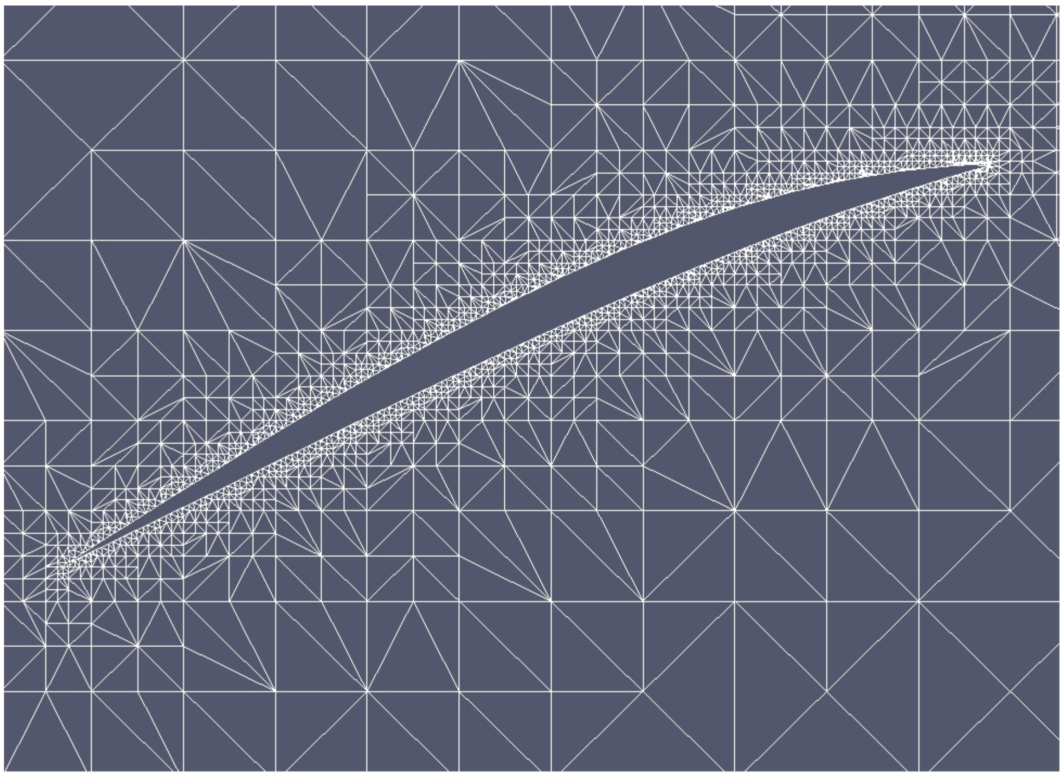


Figure 4.3: Slice through Wageningen B4-70 propeller mesh.

desired, this stage of the meshing procedure can take a considerable amount of time for complex geometries and if running on a multi-user cluster this stage is best executed in batch.

The final stage of the mesh generation procedure in snappyHexMesh is the creation of a layer mesh. This involves displacing the mesh on the surface and inserting extra cells of a user specified height so that y^+ may be controlled as well the number and distribution of cells used to resolve the boundary layer. Adding a layer mesh is a time consuming process and it may take multiple iterations of meshing follow by solution to achieve a suitable value for y^+ (see Section 2.3.6 for the importance of y^+).

4.2.1 Automated Mesh Generation

The automatic generation of the mesh is performed using the geometry from Section 4.1.2 with some of the functions listed in Appendix C. The automation was designed to be as general as possible, whilst still containing the requisite features for modelling propellers and rim driven thrusters. There are three main parts to the mesh generation as previously outlined in Section 4.2: the base meshing done with blockMesh, the geometry snapping done with snappyHexMesh and the feature edge snapping improvement.

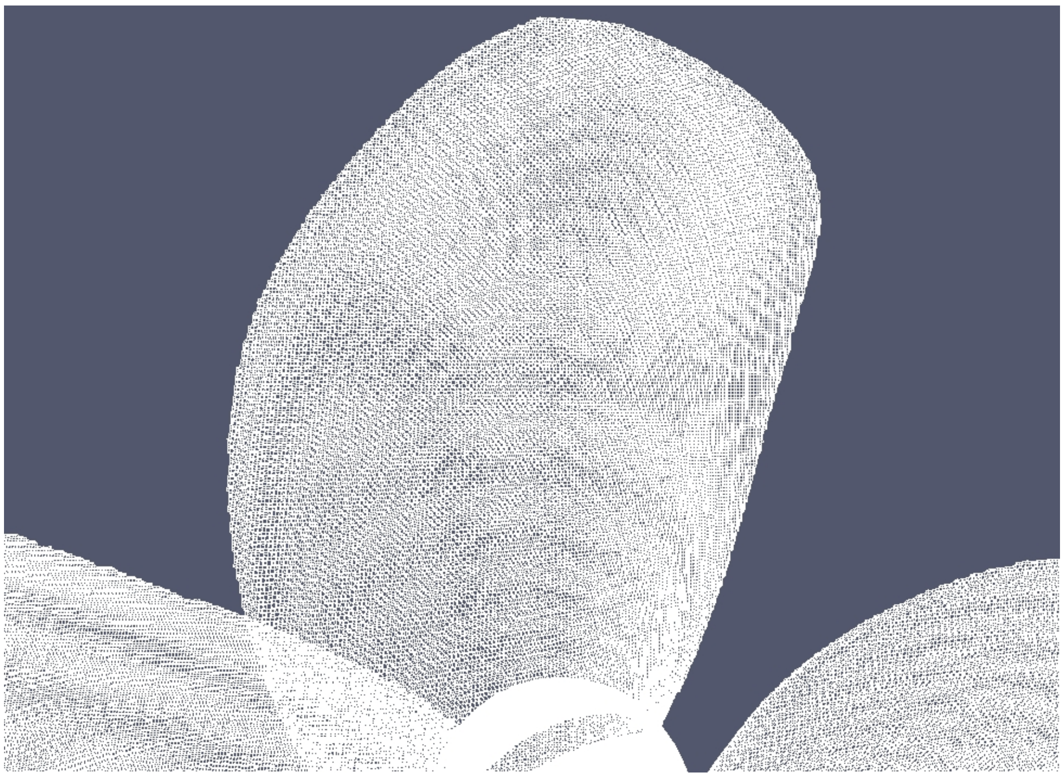


Figure 4.4: Meshed Wageningen B4-70 propeller.

makeDomain

Specifying the computational domain is done using the `makeDomain` function which takes as input the limits of the domain and writes a blockMeshDict file to generate the domain. To maintain the generality of the function, the default boundaries are referred to as xmin, xmax, ymin, ymax, zmin and zmax, denoting the minimum and maximum x, y and z values respectively. The function also takes a base size parameter which defines how large the mesh cells will be and the total number of cells in the base mesh can be calculated by:

$$n_{\text{cells}} = \frac{x_{\max} - x_{\min}}{\text{base size}} \times \frac{y_{\max} - y_{\min}}{\text{base size}} \times \frac{z_{\max} - z_{\min}}{\text{base size}}$$

Once the `makeDomain` function has created the blockMeshDict file, the base mesh is generated by running the blockMesh program. This can be done within the script using the `run` wrapper function which executes a command in the shell and automatically stores the output in a log file, as well as optionally forwarding the output to the console too. The 'silent running' option in the `run` function is included to suppress the output when an automated code is called multiple times, or when the simulations are running otherwise unattended (*exempli gratia* in batch on a supercomputer), and this also reduces the time taken to execute.

snappyHexMesh

Automated generation of a meshed geometry from the base mesh and .stl files is then completed using the OpenFOAM utility snappyHexMesh. The function `snappyHexMeshDict` creates the requisite control file to generate the final mesh. As mesh generation is specific to the geometry and problem setup, the function takes a large number of parameters, as well as many optional parameters for the mesh quality, the defaults of which should generally produce a good mesh. If snappy-HexMesh is able to snap the mesh correctly, then the mesh produced should have a maximum non-orthogonality of 60 degrees and a maximum skewness of five, except at the boundaries where this requirement is relaxed to 20.

The primary arguments that are required are: ‘stls’, a list of the names of the .stl files; ‘lvls’, a list of the required mesh refinement on the respective .stl surface and ‘loc’, the co-ordinates of a location in the meshed region. There are two further keyword arguments that are also useful, especially in the case of rim driven thruster meshing, and these are: ‘MRFrotor’, which takes a list of three parameters, defining the start co-ordinate, end co-ordinate and radius of a rotating region, and uses these to create a cellZone for using multiple reference frame rotation; and ‘features’, which takes the number of iterations to use for feature edge snapping which vastly improves the capture of the geometry by the mesh.

Once the snappyHexMeshDict file has been created, the mesh is created with the `run("snappyHexMesh")` command. This creates a snapped hexahedral mesh, but this still required some further modification before it is ready for solution. An empty patch called `rotor_region0` is created in the process of creating the rotating region, that requires defining correctly with a patch type of empty. For correcting the patch type, a function called `changePatchType` exists to change the patch type to any required, therefore the command `changePatchType("rotor_region0", "empty")` will fix the patch.

As the automated generation of the initial and boundary condition files, performed by the function `makeFieldsiKO` (see Section 4.4.1), requires knowledge of which patches are to be treated as inlets and outlets, the patches need to be renamed. In the rim driven thruster case the patch in the positive z direction, that is the direction in which the thrust is produced, is the inlet boundary which the `blockMesh` function names `zmax`. The opposite patch is the outlet boundary, which the `blockMesh` function names `zmin`. A function `renamePatch` was written that takes the patch name and the desired new name, and renames the patch to the new name. This allows the renaming of `zmax` and `zmin` to `inlet` and `outlet` respectively, such that the boundary fields are generated correctly.

Robust Automated Generation

Automated mesh generation needs to produce high quality meshes for a large number of geometries which to some extent can be tuned by the mesh quality parameters in the snappyHexMeshDict file. The meshing parameters which were found to be most critical in overall quality were nCellsBetweenLevels, tolerance and nSolveIter. The nCellsBetweenLevels parameter controls the number of cells in each successive level of refinement. Having more cells in a level improves the ability of the mesh to move, while reducing the likelihood of solution divergence, but comes at a cost of requiring more cells and subsequent computational memory and time.

The parameters tolerance and nSolveIter are part of the mesh snapping controls and affect how well the geometry is captured. Increasing the tolerance increases the amount the mesh will move towards the .stl geometry during the snapping phase. The higher the tolerance, the better the capture of the geometry, however if the tolerance is too high, points will snap to the wrong patch. It was found setting the tolerance to between 1.0 and 1.7 is sufficient for a good mesh snapping without any patches going awry. Similarly the nSolveIter parameter controls the number of iterations in the snapping phase of the meshing. The number of iterations must be a trade-off, as more iterations lead to a better snapping, but with diminishing returns and increased mesh generation time. 30 iterations were found to be sufficient to produce the desired mesh without any noticeable improvement beyond this number.

Feature edge snapping can further improve geometry capture, especially of sharp edges that are typically not well captured by snappyHexMesh alone. There are two ways that feature edge snapping can be performed, either with a utility called snapEdge, or with the inbuilt feature edge snapping in snappyHexMesh. Functionality for both methods of feature edge snapping has been included, although snapEdge requires *a priori* compilation from source code as it is a third party OpenFOAM utility.

First the snapEdge functionality was included in the function `snapSTL` which, given a list of .stl files, writes the requisite snapEdgeDict dictionary with some default parameters and then calls the snapEdge utility. However, a more robust method of feature edge snapping was found in snappyHexMesh version 2.1.0, despite being listed as an experimental feature at the time. This method performs the feature edge snapping during the snapping stage of the meshing, rather than retrospectively applying it after the meshing is completed. It is enabled by passing the argument `features=10` to the `snappyHexMeshDict` function, which will prompt the function to add the necessary features sub-dictionary, extract the edge mesh (.eMesh) files from the .stls, and perform 10 iterations of feature edge snapping during the mesh snapping phase.

4.3 Steady State Solution Setup

Once the meshing is finished, the rest of the set up for solution in OpenFOAM can be performed; this involves writing a number of control dictionaries that specify the solution schemes to be used. Figure 4.5 shows the resulting case structure and the location and names of files that need to be written for the example Wageningen B4-70 case. The OpenFOAM solver chosen for simulating steady state flows is MRFSimpleFoam, which uses moving reference frames and the SIMPLE (Semi-Implicit Method for Pressure Linked Equations) algorithm to handle pressure-velocity coupling. The key dictionaries for solving with MRFSimpleFoam are fvSchemes, fvSolution and controlDict which are located in the system directory as well as the transportProperties, RASProperties and MRFZones dictionaries located in the constant directory.

The fvSchemes dictionary is where the numerical schemes for calculating the gradient, divergence and laplacian of variables as well as the finite volume interpolation schemes are specified. Different schemes were tested in an attempt to improve convergence and robustness during the verification and validation procedure and the baseline schemes chosen for the solution in this work are Gaussian schemes using linear interpolation with the exception of the turbulence quantities k, ϵ and ω which use an upwind interpolation scheme. All the discretisation schemes are second order accurate in space and the linear and upwind interpolation schemes are second and first order respectively.

Control of the solution and choices such as matrix solvers, preconditioners and inner iterations are done through the fvSolution dictionary. Here, for every variable that is solved for, the user can specify the matrix solver and preconditioner. There are also two tolerance levels that are given for the solution of each variable; the absolute tolerance and the relative tolerance. The absolute tolerance gives some convergence control and the variable will stop the solution if the residual falls below this value. The relative tolerance is the value residuals must achieve, relative to the initial residual at the beginning of the timestep, for solution to continue onto the next variable, thus allowing control over the inner iterations of the solver. The matrix solvers chosen were Preconditioned Conjugate Gradient (PCG) and Preconditioned Bi-Conjugate Gradient (PBiCG) for symmetric and asymmetric matrices, respectively. The pressure matrix, however, was solved using a generalised Geometric-Algebraic Multi-Grid (GAMG) solver to speed up convergence. The preconditioning of the solution matrices was performed using Diagonal Incomplete-Cholesky (DIC) for the PCG solver and Diagonal Incomplete-LU (DILU) for the PBiCG solver. After an issue with the turbulence quantities, k and ϵ, reaching the absolute tolerance too quickly and the solution of these variables halting prematurely, all the absolute tolerances were set to 1×10^{-12}. Experimentation with different relative tolerances found that a relative tolerance of 0.01 worked best for speed and convergence.

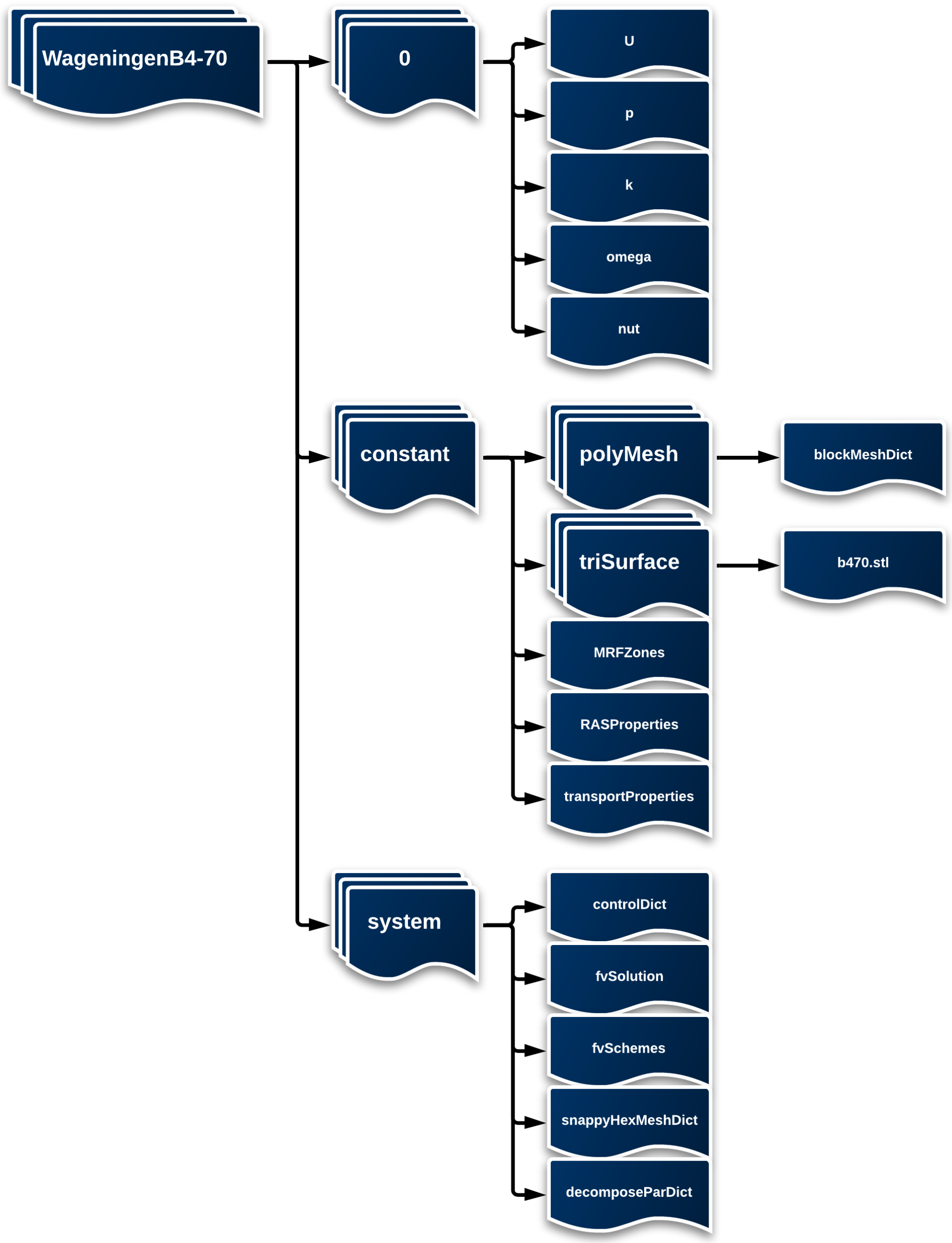


Figure 4.5: Outline diagram of the OpenFOAM case file and directory structure.

Top level control of the simulation is done through the controlDict dictionary, through which the number of iterations, size of timestep and frequency of saving are controlled. It is possible to set up controlDict dictionary to be re-read every timestep, such that changes may be made during computation in response to the equation residuals if desired. The controlDict dictionary is also the file where force calculations are specified by giving details of density, centre of rotation and the patches over which the forces are to be calculated. If the force calculations are specified prior to solution, the frequency of output can be given such that a history of the force values during the solution can be viewed and used to monitor convergence. Additional post-processing forces calculations can also be added after solution and computed for each saved timestep using the `execFlowFunctionObjects` command.

Viscosity and the model used for the working fluid are set in the transportProperties dictionary; set as a Newtonian fluid with a constant kinematic viscosity of 1.307×10^{-6}m^2/s. In the RASProperties dictionary, it is possible to select the turbulence model and turn turbulence 'on' or 'off' and in the MRFZones dictionary, where the rotating region is specified, the axis about which rotation occurs and angular velocity are set.

The problem specific parts of the solution setup are the boundary and initial conditions. In OpenFOAM these are both specified in one file per solution variable in contrast to a lot of commercial CFD packages which allow the selection of a specific boundary 'type' that sets up all the fields on that boundary automatically. This has the advantage of being able to specify entirely custom boundary conditions on a fieldwise basis as well as being entirely transparent as to the exact condition being applied to each field. Initial conditions for the problem have been chosen to be a single uniform value, the advance speed in the case of velocity, the outlet pressure in the case of pressure and the initial turbulence quantities are based on the turbulent intensity and turbulence viscosity ratio (see Section 2.3.7).

For the verification procedure the advance velocity was fixed at 1 m/s, and as the forward thrust direction is the z-axis, the velocity vector is (0 0 -1). Using a turbulent intensity of 2.5%, this lead to a value of the turbulent kinetic energy, $k = 0.000625$ m^2/s^2, and using a turbulence viscosity ratio of 10 yielded a turbulent dissipation rate of $\epsilon = 0.00269$ m^2/s^3. The boundary conditions are set differently for each type of boundary, with the key options of the four types used in the simulations shown in Figure 4.6. For a wall type boundary, *id est* the surface of the propeller, the velocity is set to 0, pressure is set to zeroGradient and the turbulence quantities are set to use wall functions. A zeroGradient boundary condition is a Neumann condition such that the gradient of the pressure in the surface normal direction is equal to 0. The specific boundary condition names within OpenFOAM for the wall functions are kqRWallFunction, epsilonWallFunction, omegaWallFunction and nutWallFunction

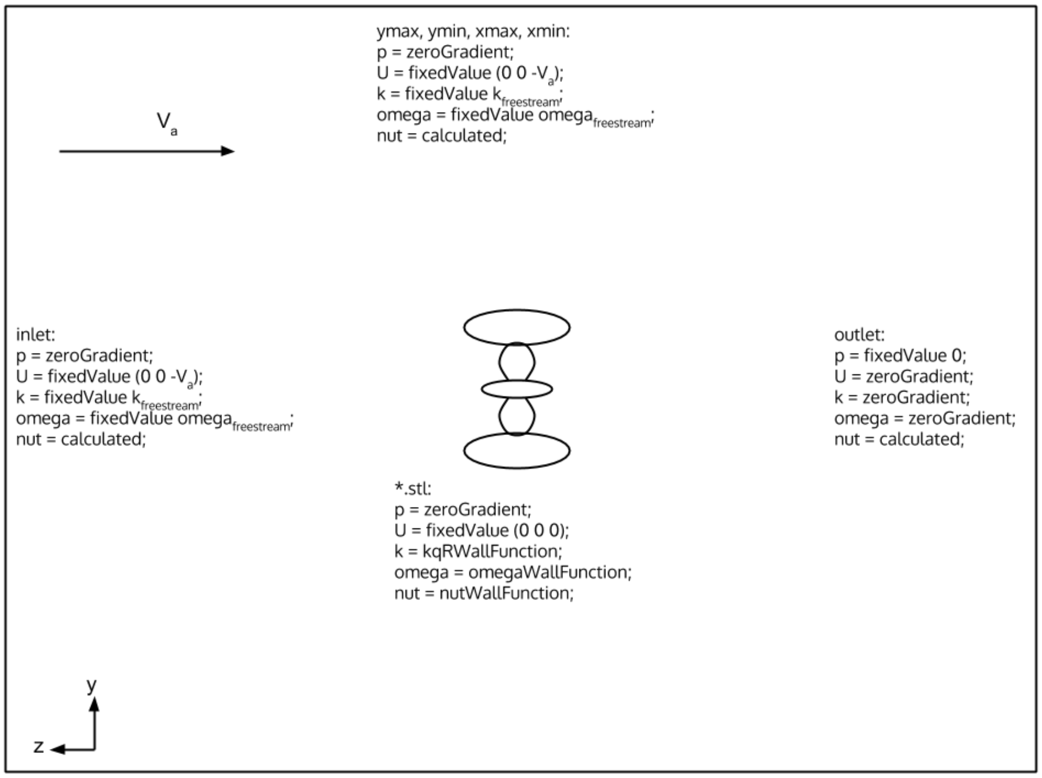


Figure 4.6: Diagram of the four boundary conditions.

for the scalar variables of k, ϵ, ω and ν_τ, respectively.

At the outlet the pressure is set to 0 as it is a relative quantity and the remaining fields are set as zeroGradient boundaries. For the inlet, the velocity obviously takes on the value of the advance speed, pressure is set as zeroGradient and the turbulent quantites set up to reflect the freestream turbulent intensity and turbulence viscosity ratio. Similarly for the outer walls of the domain, the velocity is set (as a vector) to be the same as the inlet, essentially preventing any fluid from traversing across the boundary but without acting as a 'real' (*id est* no-slip) wall. The pressure boundary condition at these outer walls is set to zeroGradient and the turbulence quantities are set up the same as the inlet and initial conditions.

4.4 Automation of Steady State Solution

To automate the above process of setting up the simulation, a number of functions were created to calculate the variables and write the necessary dictionaries. All the functions required to do these tasks are listed in Appendix C, completing the functionality required to automate design evaluation for the purpose of optimising rim driven thrusters. The procedure for setting up a simulation using these functions is outlined in Section 4.4.1 below.

4.4.1 Simulation Case Setup

Initially, the simulation case setup requires the making of a case directory, and the requisite directory structure within it. This is carried out by the function `makeCase` which takes as an argument the name of the case and an optional parameter to overwrite the case if it exists, to reduce data storage requirements in some use cases.

Writing the dictionary to control the simulation is performed by the `controlDict` function. This takes the parameters of the solver name, end time and time step, along with an optional choice of write interval, which is the number of iterations between each successive saving of the field files to disk. If this optional parameter is not given, then the default is to write the data at the end of the simulation only.

Selection of the discretisation schemes and the control of the solvers inner loops comes from the fvSchemes and fvSolution dictionaries respectively. These can be automatically created using the Python functions of the same name. Suitable, robust defaults have been chosen such that these dictionaries can be written with calls to the function requiring no parameters (*id est* `fvSchemes(); fvSolution();`). However, functionality to select discretisation schemes, in the fvSchemes dictionary, and custom inner iteration tolerances, in the fvSolution dictionary, has also been included.

Viscosity for the solution is set using the transportProperties dictionary and function of the same name, with the default kinematic viscosity equal to that used throughout this thesis ($1.307 \times 10^{-6}\mathrm{m}^2/\mathrm{s}$). Similarly, the turbulence properties of the simulation are set through the RASProperties dictionary and the function to create it is named likewise. This function takes a parameter of the turbulence model name, and two boolean parameters of whether turbulence should be on and whether to print the turbulence coefficients.

To complete the global settings for a marine propulsor simulation, the rotation must be specified which can be done using the MRFZones dictionary created using the `MRFZones` function. This function requires two arguments; the name of the rotating region and the number of revolutions per minute. There are also options to change the centre of rotation, which defaults to the origin, and the axis of rotation which defaults to the z-axis. It is also possible to include names of patches in the rotating region that do not rotate, although this feature is not required for rim driven thrusters.

Perhaps the most important function in the `ajopenfoam` package is the function `makeFieldsiKO`, named as such as it creates the internal and boundary fields (makeFields) for incompressible flow (i) with the k-ω turbulence model (KO). For a given initial velocity, this function assigns the relevant boundary conditions according to the name and type of the patches, extracted from the constant/polyMesh/boundary file. Turbulent intensity and viscosity ratio, defaulting to 0.1 and 10 respectively, can be passed to the function which then calculates the freestream turbulent kinetic energy (k) and specific turbulent kinetic energy dissipation rate (ω) automatically. For wall boundary conditions, the parameter `wallFunctions` can be set to either `True` or `False` depending on whether wall functions are desired. Two further optional functionalities are included through the parameters `inlet_pressure` and `rot_omega`, the former specifying the inlet pressure for a pressure driven flow and the latter specifying the rotation rate for rotating wall boundary conditions. With all this information, the function computes and writes the requisite files for `p`, `U`, `k`, `omega` and `nut`, which give the initial and boundary conditions for their respective fields.

4.5 Running The Simulation

After the entire problem has been set up, the numerical solution process can begin. Depending on the size and expected run-time of the problem, each case was run either locally on a PC or on the cluster Iridis3. To run the case locally it is simply a matter of changing the current directory to the relevant case directory and using the commands:

```
MRFSimpleFoam >> log.MRFSimpleFoam &
tail -f log
```

This runs the simulation and appends the output to the log file, the ampersand telling it to run in the background. The second line then outputs the log to the console as it is written, allowing the residuals and solution progress to be monitored. Alternatively, if it is desirable to run the program in the foreground, so that it can be easily exited if it begins to diverge for example, this is possible with the command `MRFSimpleFoam | tee log.MRFSimpleFoam` which writes the output to the file `log.MRFSimpleFoam` as well as the console.

When the number of simulations or their size or runtime rendered it impractical to run them locally, the Iridis3 supercomputer was used to solve the simulations. There is some trade-off when deciding whether execution should be performed in batch on the supercomputer as the typical queue length is approximately 12 hours. For example if three simulations of length five hours were required, it would take 15 hours to run them sequentially locally or a total (including queueing time) of 17 hours to run them in a parallel batch on the supercomputer. Additionally, where a number of different solution configurations are being tested, some of which are likely to fail within a few iterations, it is a fruitless exercise to wait the entire length of a 12 hour queue for the solution to fail after five minutes of runtime. Actual execution of programs on the Iridis3 supercomputer is done through submitting the required commands to the distributed resource manager, Torque in the case of Iridis3.

As the solution size gets larger or if a reduction in solution time is desired, it is possible to run the solution in parallel. It is simpler to generate the mesh and set up the field files in serial and then to decompose the domain into parallel regions for the solution, recombining them after solution for serial post processing. However, if the desired mesh is too large to fit into RAM on a single node (22 Gigabytes on Iridis3, which experience shows is enough for roughly 5.5 million cells), then it may be necessary to also generate the mesh in parallel. First of all, the decomposition must be set up in the decomposeParDict dictionary, where the number of subdomains and how the domain is to be split is specified. Then the decomposition, solution and reconstruction is performed with the following commands:

```
decomposePar | tee log.decomposePar
mpirun -np 4 MRFSimpleFoam | tee log.MRFSimpleFoam
reconstructPar | tee log.reconstructPar
```

These will run the solution in the foreground, as they must be executed sequentially, alternatively they can be run in batch on a cluster or the commands can be entered into a script which can be run in the background to enable local running in the background while preserving the sequential order of execution.

4.5.1 Automatically Running The Simulation From Python

The simulation can be automatically run from within a script, after it is meshed and set up, by using the `run` function. This function automatically directs the output of whichever program is run into a log file with the option of including it as output of the Python function also. For example, to run `MRFSimpleFoam`, the function call would be `run("MRFSimpleFoam")` and would store the output in the file log.MRFSimpleFoam.

If automatic parallel running is desired, the functionality is also included. First, the `decomposeParDict` function will write a dictionary of the same name with the required number of processes in the specified decomposition. The domain can then be decomposed with the command `run("decomposePar")` which will store the output of the domain decomposition utility in the file log.decomposePar.

Once the domain has been decomposed, it is then possible to run the simulation in parallel by using the `run` function. For example, a four process simulation would be invoked with the command: `run("mpirun -np 4 MRFSimpleFoam")`. In this case the output is stored in the file named log.mpirun. Similarly, reconstruction of the domain back into a single mesh can be automated with the command: `run("reconstructPar")`.

4.6 Automated Post Processing

There are many ways in which one might want to post-process a simulation, depending on the purpose of the simulation and data required. The post-processing functions in `ajopenfoam.py` are focussed only on getting the required data for the design optimisation of rim driven thrusters. Consequently, the two main functions are `residuals` and `forces` which extract the residuals and forces (and moments) respectively, allowing the subsequent calculation of thrust and torque coefficients and from this derive the efficiency of the device.

Convergence is a necessary, although not sufficient, condition for a good computational fluid dynamics simulation result. The equation residuals can be extracted from the solution log file using the `residuals` function which returns a dictionary containing lists of the residuals with the solution variables as the dictionary keys. The exception to this are the time step continuity errors, where the sum of the local residuals, the global residual and the cumulative residual are stored with the keys `"local cont"`, `"global cont"` and `"cum cont"` respectively. Thus the continuity residual, which is perhaps the best indicator of convergence or lack thereof, can be checked with the command:
`assert residuals("log.MRFSimpleFoam")["local cont"] < 1e-4`.

Once there is some confidence that the solution has obtained a sensible solution, the performance metrics of interest must be extracted. This is done with the `forces` function, which extracts the force and moment histories stored by the force function objects, that are set up either *a priori* or *posteriori* with the `forceFuncObj` function. The `forces` function returns a set of lists of pressure based and viscous based forces calculated from the normal and shear stresses on a face respectively. Summing the outputs for the z-axis will yield the thrust and torque which can then be normalised against fluid density, rotation rate and diameter to give the preferred non-dimensional co-efficients.

4.7 Example: Steady State Simulation of Wageningen B4-70

To illustrate the entirety of the automation functions and general simulation process, here follows an example that computes the thrust coefficient, torque coefficient and efficiency for a 70mm Wageningen B4-70 propeller at an advance speed of one metre per second and rotation rate of 3000 revolutions per minute.

```
from ajopenfoam import *
from ajblades import outputb470stl
```

```
# set up the case
makeCase("b470-70mm")
controlDict("MRFSimpleFoam", "500", "1")
forceFuncObj("prop", "b470.stl_b470", 1027)
decomposeParDict()
fvSchemes()
fvSolution()
MRFZones("rotor", 3000)
RASProperties("kOmegaSST")
transportProperties()

# create the geometry
outputb470stl(0.07)
os.system("mv b470.stl constant/triSurface/b470.stl")

# create the mesh
makeDomain(-0.42, -0.42, -0.42, 0.42, 0.42, 0.21, 0.0175)
snappyHexMeshDict(["b470.stl"], ["(6 6)"], [0, 0, 0.1],
                  MRFrotor=[[0, 0, -0.05], [0, 0, 0.05], 0.05],
                  features=10)
run("blockMesh")
run("snappyHexMesh -overwrite")

# set up boundary conditions
changePatchType("rotor_region0", "empty")
renamePatch("zmax", "inlet")
renamePatch("zmin", "outlet")
makeFieldsiKO([0, 0, -1])

# run the simulation
run("MRFSimpleFoam")
assert residuals("log.MRFSimpleFoam")["local cont"] < 1e-4

# extract the performance coefficients
force = forces("prop")
J = 1 / (50*0.07)
KT = (force[3][-1] + force[6][-1]) / (1027*50**2*0.07**4)
KQ = -(force[9][-1] + force[12][-1]) / (1027*50**2*0.07**5)
eta = (KT*J) / (2*3.14159*KQ)
```

4.8 Unsteady Simulation Method

Much of the solution set up and execution for the unsteady simulations is the same as for the steady state simulations. The main differences are that the time dimension needs to be resolved, which raises the question as to what resolution should be used, and the geometry needs to rotate to give a realistic representation of the unsteady flow.

The geometry creation for the unsteady simulations is the same process as for the steady state simulations. The unsteady rotational movement is imparted at the meshing stage, although it is theoretically possible to use multiple geometries and mesh them all separately, this is not an efficient way to model movement; as a timestep equivalent to 0.5 degrees would require 720 separate geometries and associated meshes!

4.9 Unsteady Meshing

The unsteady simulation meshes were also created with blockMesh and snappyHexMesh, but other meshing programs such as Netgen, Harpoon and Gmsh were also tried. Harpoon functions in much the same way as snappyHexMesh, starting with a base hexahedral mesh and refining to a surface level, then snapping to the surface before growing a layer mesh. Harpoon has the advantage of having a Graphical User Interface (GUI) unlike snappyHexMesh, allowing easy inspection of the mesh at each stage of generation, however exporting into an OpenFOAM format was not natively supported in the version tested and thus export had to be preformed via an intermediate file format.

Conversely, both Gmsh and Netgen start by meshing the surfaces and then 'grow' the mesh into the volume. This process has the advantage of capturing surfaces more accurately, which is important for the unsteady case when meshing of the interface between rotating and static regions. However, the main problem found with both Gmsh and Netgen, is that surface imperfections (*exempli gratia* small surface holes, perhaps introduced by SolidWorks .stl exporter) caused the meshing to fail.

As it was important to make sure that the interface in the unsteady simulations was perfectly cylindrical, many parameters were adjusted in snappyHexMesh to attempt to get the best surface capturing possible. The findings were primarily that the snapControls dictionary in the file snappyHexMeshDict could be adjusted to increase the number of snapping iterations, it was also found that a snapping 'tolerance' of between 1.0 and 2.0 produced the best surfaces. If more mesh cells are not undesirable then the refinement level on the interface could be increased, and a good compromise is found by increasing the maximum refinement level but retaining a relatively small minimum refinement level. Alternatively, surface capturing could be improved using a third party utility called snapEdge which focusses on making sure feature edges are well defined in a mesh, although a similar functionality is natively provided in snappyHexMesh from OpenFOAM version 2.0 onwards.

There are two main ways of creating a mesh with the two coincident patches required for a sliding interface unsteady simulation. First, the rotating and static regions of the mesh can be meshed separately and then merged together, preserving the boundaries between regions. Alternatively, and perhaps preferrably as it only requires a single mesh generation, the mesh can be generated as a whole with a rotating region defined in snappyHexMesh. This region can then be used to define a faceZone which can be converted into two independent patches for the interface. To do this, two empty patches must be created in the boundary file, called AMI1 and AMI2 in this work (for Arbitrary Mesh Interface). If the faceZone is called rotor then the commands to populate the interface patches are as follow:

```
createBaffles -internalFacesOnly -overwrite rotor '(AMI1 AMI2)' | tee
    log.createBaffles
mergeOrSplitBaffles -split -overwrite | tee log.mergeOrSplitBaffles
```

This method will even work for parallel meshes with the inclusion of the -parallel flag.

4.9.1 Dynamic Meshing

There are a number of methods for performing dynamic meshing and they are outlined here, with the reason for choosing a sliding interface over other methods of dynamic meshing. To begin with, there are simple mesh deformation methods, where the number and connectivity of cells are unchanged and only the location is changed. Mesh deformation is not a particularly successful method for large displacements or rotation as mesh quality, particularly cell skew and aspect ratio, typically reduces beyond a reasonable level, although it has been made to work for rotation (Liefvendahl et al., 2010). If the movement is linear, for example a piston, then the problem of cell aspect ratio changing can be addressed by the addition or destruction of cells at the boundaries, but this technique is of little use in rotation as the primary issue is the cell skew at the edge of the rotational region.

Preventing the cells at the blade tips in a rotating geometry becoming too skewed can be achieved through topological changes where, at a rotating boundary, the connectivity between cells is cut, moved and reattached. In this way, original mesh quality is preserved throughout the mesh with the exception of the cells adjacent to the rotating boundary. However, changing the mesh in this way every time step is expensive and has the constraint of a 1:1 mapping at each step of rotation (*id est* not good for unstructured meshes).

Therefore a sliding mesh interface can be used, such as a Generalised Grid Interface (GGI, Beaudoin and Jasak (2008)) or Arbitary Mesh Interface (AMI, Farrell and Maddison (2011)). These weight the flux between cells across the rotational boundary such that the cells do not need to be directly matched or connected. This achieves a similar accuracy to a topological change method but allows for hanging nodes and removes the requirement for the topological detachment and reattachment.

Another option for the dynamic meshing of rotation is using immersed boundary methods. These have a fixed background mesh through which the boundary moves, updating the position of the walls within the background mesh at each timestep. This type of method was not implemented in OpenFOAM at the time of investigation and although its existence is acknowledged, it is not covered in any further depth here.

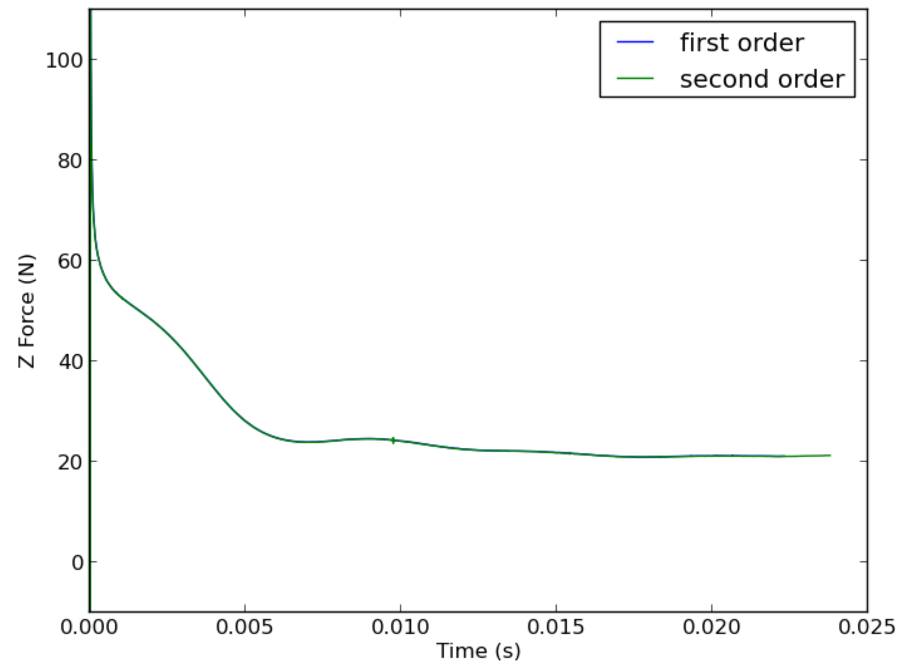
Given all the options for dynamic meshing, the sliding interface method was chosen as it preserves mesh quality (*id est* cells do not change size or shape, only position) and a quicker solution time over a changing topology method.

4.10 Unsteady Solution Set Up for OpenFOAM

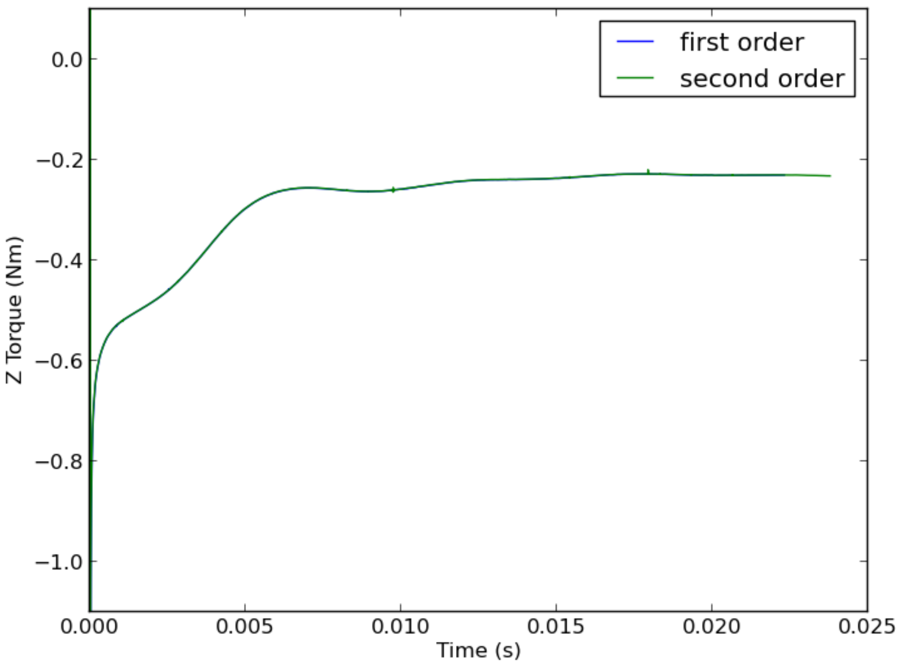
The set up for the unsteady simulations proceeds in a similar manner to the steady state case but with a number of necessary additional parts. The OpenFOAM solver chosen for the unsteady simulations is pimpleDyMFoam, which uses the SIMPLE (Semi-Implicit Method for Pressure Linked Equations) and PISO (Pressure Implicit Splitting of Operators) algorithms to handle pressure-velocity coupling for the inner (within a timestep) and outer (across timesteps) iterations, respectively. The key dictionaries which differ to steady state simulation when solving with pimpleDyMFoam are fvSchemes, fvSolution and controlDict which are located in the system directory and also the dynamicMeshDict dictionary located in the constant directory.

The numerical schemes chosen for the unsteady simulation were initially second order Gaussian schemes for calculating the gradient, divergence and Laplacian with a first order discretisation in time. However, while it is typically more stable to use a first order time discretisation, it is more accurate and less diffusive to use a second order time discretisation. The backward differencing scheme, listed as a second order implicit scheme in the OpenFOAM User Guide, was found to be the most stable second order time discretisation and Figures 4.7a and 4.7b show that the first order scheme is accurate enough, but the second order scheme works from the outset and so is chosen for the base case.

For the unsteady simulation the same matrix solvers were chosen, that is Preconditioned Conjugate Gradient (PCG) and Preconditioned Bi-Conjugate Gradient



(a) Force History



(b) Torque History

Figure 4.7: Solution histories for Wageningen B4-70 propeller using first and second order time discretisation schemes.

(PBiCG) for symmetric and asymmetric matrices, respectively, except for the pressure which was solved using a generalised Geometric-Algebraic Multi-Grid (GAMG) solver to speed up convergence. The preconditioning of the solution matrices was performed using Diagonal Incomplete-Cholesky (DIC) for the PCG solver and Diagonal Incomplete-LU (DILU) for the PBiCG solver.

In an unsteady simulation the timestep size is important and has been shown to be crucial to the final result (Kaufmann and Bertram, 2011). If the timestep is too large then the solution will diverge, although OpenFOAM has the option to dynamically change the timestep size to prevent this happening by limiting the maximum Courant number. As the nominal rotational speed for the unsteady simulation is 3000 revolutions per minute, this corresponds to a complete single revolution every 0.02 seconds. A timestep which corresponds to one degree of rotation, as recommended by Kaufmann and Bertram (*ibidem*), would evaluate to $0.02/360 = 5.5555 \times 10^{-5}$ seconds. However to give a rounded timestep that is smaller than this, a timestep of 2×10^{-5} seconds was initally chosen. This timestep was found to be unstable and was later refined to 2×10^{-6} seconds, corresponding to a maximum Courant number of approximately 0.3.

The boundary and initial conditions are almost identical to those of the steady state simulation, although the mesh rotation and sliding interface must be accounted for. The boundary condition for the AMI patches was a `cyclicAMI`, with the corresponding patch specified along with the type in the constant/polyMesh/boundary file. The other thing that is important to specify on any moving surfaces, is the velocity boundary condition should be a `movingWallVelocity`, and thus the surface velocity is calculated from the mesh motion.

4.11 Solving Unsteady Simulations

The solution for the unsteady simulations is the same as for the steady counterpart, with the replacement of `MRFSimpleFoam` with `pimpleDyMFoam`. However, due to the significantly longer solution times required to resolve the time domain, all simulations were run in batch on the Iridis3 supercomputer. As the solution needs to be restarted when maximum wall time (the maximum allocated time per supercomputer job) is reached, the command that is run is:

```
pimpleDyMFoam -parallel >> log.pimpleDyMFoam
```

This appends the output to the same log after the solution is restarted, keeping a linear log of solution progress. Alternatively, one may wish to write to separate log files for each restart, which can be achieved by changing the file name at the end of the command each time it is restarted.

Chapter 5

Verification and Validation of Computational Fluid Dynamics

Best practice guidelines for computational fluid dynamics outline a number of processes to go through to ensure the quality of the results produced and increase confidence in the simulation's representation of reality. Investigations should be conducted to verify that the results are independent of any increase in mesh resolution and any increase in domain size such that there is no benefit to accuracy from increasing the number of cells. It is also good practice to validate the results against any available experimental data. If the experimental data are available, then it is better to validate simulations against field values, comparing distributions of variables such as velocity and pressure rather than aggregate data such as forces and torques.

Throughout the verification and validation process a 70mm diameter Wageningen B4-70 propeller geometry was used, as the geometry and reliable, published experimental data were available. Therefore all the figures and results within this chapter pertain to the Wageningen B4-70 propeller, and not either of the rim driven thruster geometries in this thesis.

5.1 Steady Method Verification

Mesh verification for this work first involved using a reasonable exterior mesh and some adjustments to the layer mesh to find a good layer mesh at a reasonable y^+ value that could be fixed for the remainder of the mesh verification. Initially, the RNG k-ϵ turbulence model was used with a high Reynolds number wall function, requiring the y^+ value to lie within the log-law region of the turbulent boundary layer (see Section 2.3.6). Once a suitable layer mesh was found that yielded results with acceptable y^+ values in the range of $30 \simeq y^+ \simeq 50$, different meshes with increasing

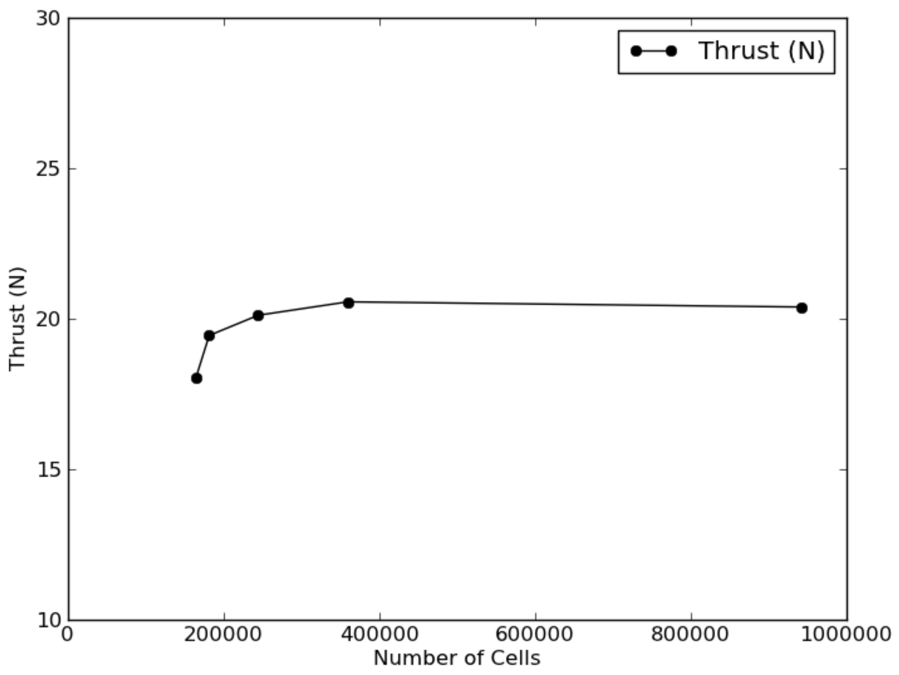


Figure 5.1: Mesh dependency study showing the effect of mesh resolution on thrust calculated.

levels of surface refinement were investigated to find the level of refinement at which any further increase did not cause any change in the results.

Figure 5.1 shows the results of the mesh dependency study. It can be seen from this figure that a mesh in excess of 400,000 cells is sufficient and any further increase in resolution does not change the result. This corresponds to a surface refinement level of six (*id est* the base mesh is refined six times), with a base mesh size of 17.5mm this gives a typical edge length of $\frac{17.5}{2^6} = 0.2734375$mm. To ensure the surface mesh was sufficient for all explored cases, a surface refinement level of seven was settled upon. At a refinement level of seven, the typical edge length becomes 0.13671875mm for a base mesh size of 17.5mm. The effect of the surface refinement level on the represented geometry can be clearly seen by comparing Figures 5.2 and 5.3, where Figure 5.2 shows the coarsest mesh at surface refinement level three and Figure 5.3 shows the level of acceptable mesh at surface refinement level six.

5.1.1 Boundary Distance Investigation

After the mesh settings at the surface were defined, the size of the computational domain was investigated. This was done by changing one dimension of the domain size while keeping the other two dimensions fixed at a baseline size that should be

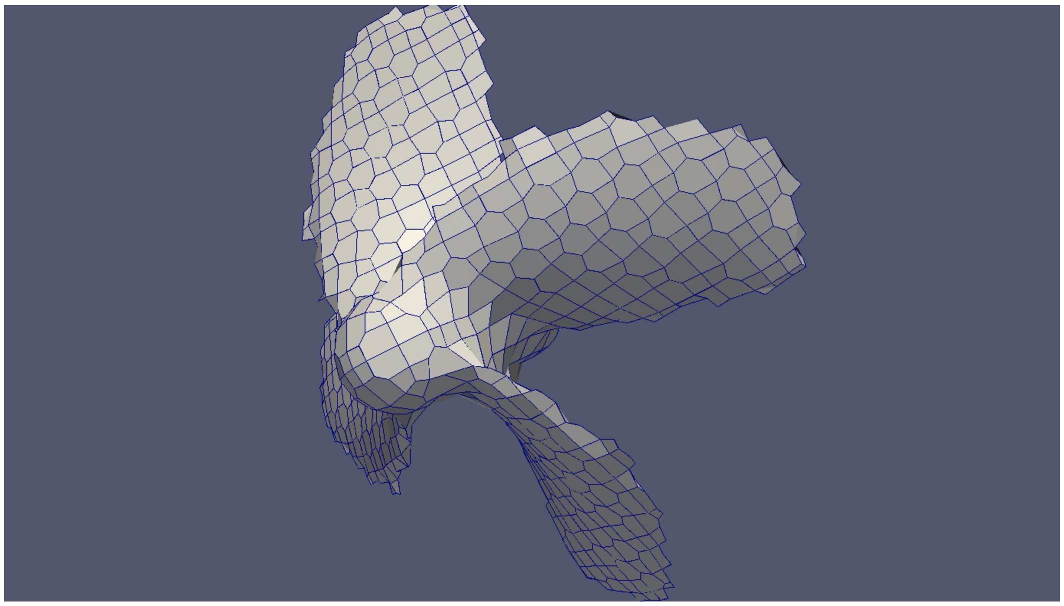


Figure 5.2: Visualisation showing a coarse surface mesh.

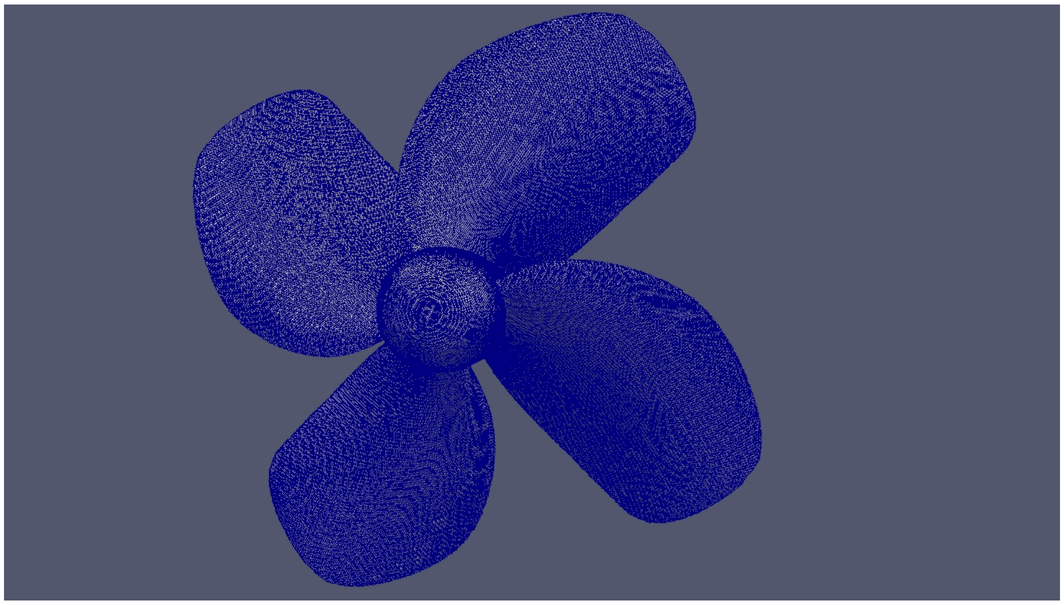


Figure 5.3: Visualisation showing a fine surface mesh.

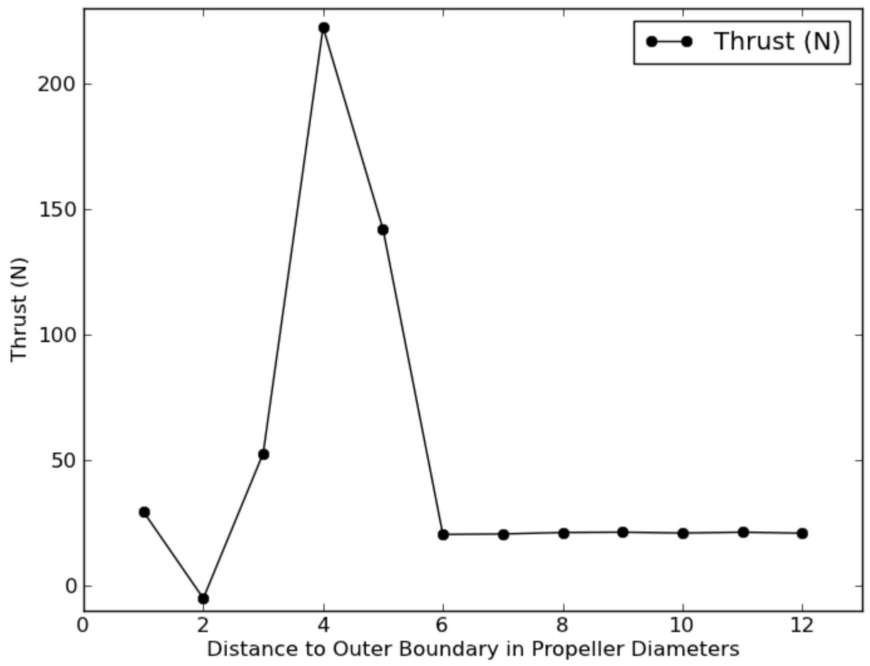


Figure 5.4: Computational domain size study showing the effect of distance to domain walls on thrust calculated.

sufficient based on experience and 'rule of thumb' guidelines. The baseline size of the domain was six propeller diameters to the inlet and radial boundaries and 12 propeller diameters to the outlet boundary. This study was conducted at a single advance ratio of 0.3, which was selected as this is the advance ratio used in the design optimisation study conducted later on in Chapter 7.

The first variable investigated and the one that was found to have the most profound effect on the results was the distance from the propeller tip to the domain walls (i.e. the radial size of the domain, although the domain is a cuboid, so in this case the minimum radial size). As can be seen in Figure 5.4, the results oscillate considerably before settling down to a reasonable value after increasing the computational domain width above six propeller diameters (420mm). This oscillation is highly unexpected and unphysical, characterised by thrust values up to ten times higher than they should be. It is presumed that this phenomenon is due to a combination of the frozen rotor formulation and the boundary condition on the radial walls producing unphysical pressure waves.

Similarly the effect of the distance from the propeller to the inlet was investigated, although it was found to have a less dramatic impact. Figure 5.5 shows the results of the study and an inlet distance of six propeller diameters was chosen. Although the results in Figure 5.5 indicate an inlet distance of five propeller diameters

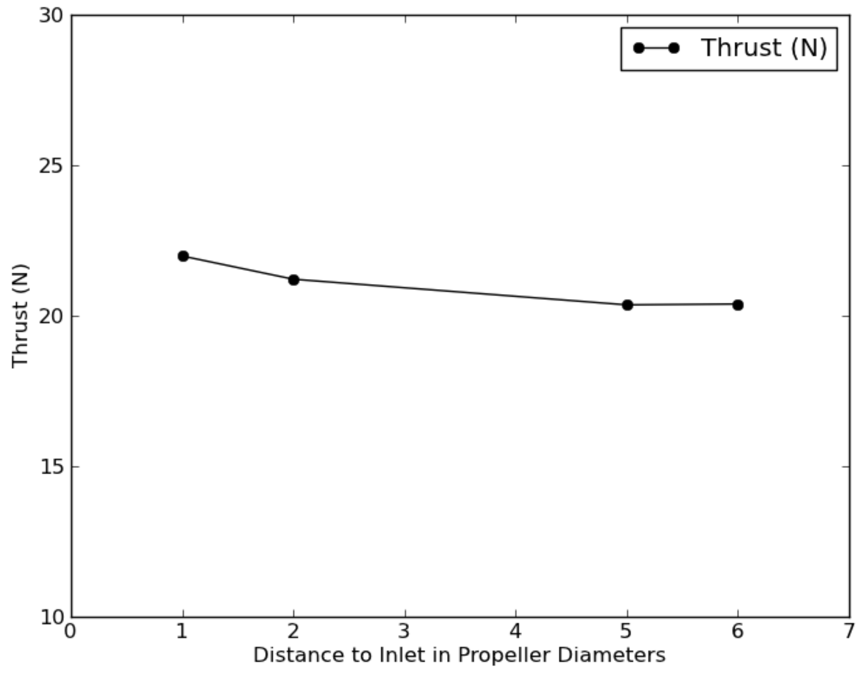


Figure 5.5: Computational domain size study showing the effect of distance to domain inlet on thrust calculated.

is sufficient, six propeller diameters were used to allow for variation in the required distance away from the verification test case.

When the distance to the oulet was investigated, it seemed to impart little variability to the results, although using an outlet distance of two propeller diameters lead to a solution that did not converge. The results in Figure 5.6 indicate an approximate minimum outlet distance of six propeller diameters. However, as the required outlet distance may be larger at higher values of the advance ratio than the test case, it was decided that the baseline outlet distance of 12 propeller diameters should be used. This should also improve the accuracy of the wake, although this is not the primary interest in this investigation.

The extremely large variation in the results in Figure 5.4 is interesting and prompted further investigation. Each case up to seven propeller diameters was repeated and a number of infill points between five and seven propeller diameters, the critical transition region, were also included. This was to investigate the transition from the large oscillations to a steady result and to check that the initial results were not anomalies. As can be seen in Figure 5.7, the results of the repeated investigation are the same as those in Figure 5.4 and thus the findings were confirmed.

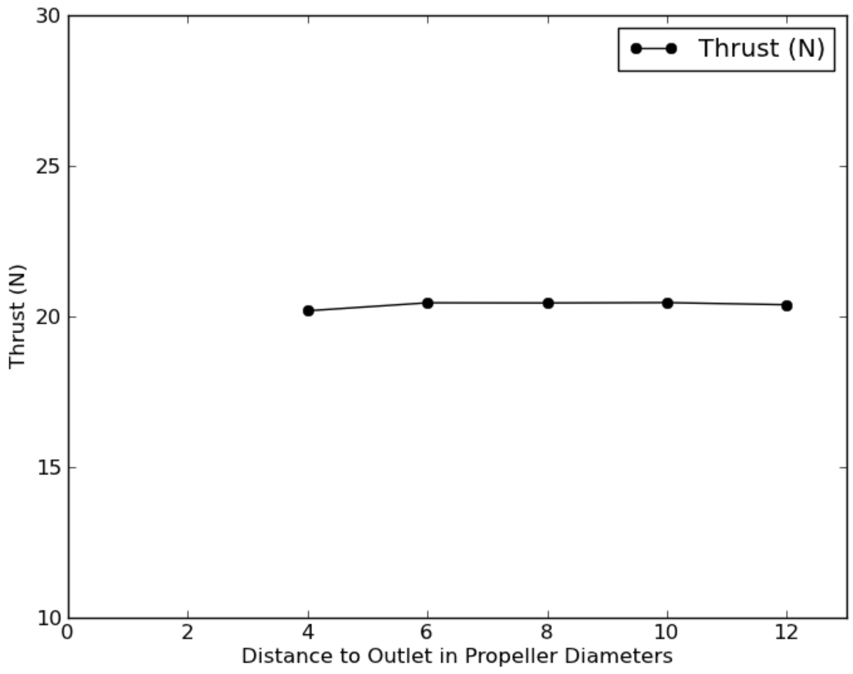


Figure 5.6: Computational domain size study showing the effect of distance to domain outlet on thrust calculated.

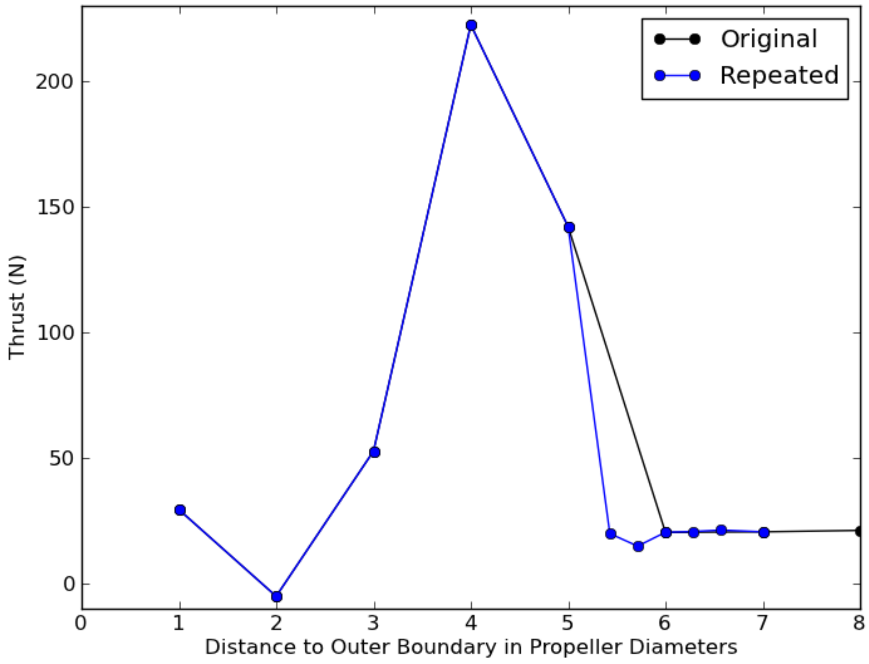


Figure 5.7: Repeated computational domain size study showing the effect of distance to domain walls on thrust calculated.

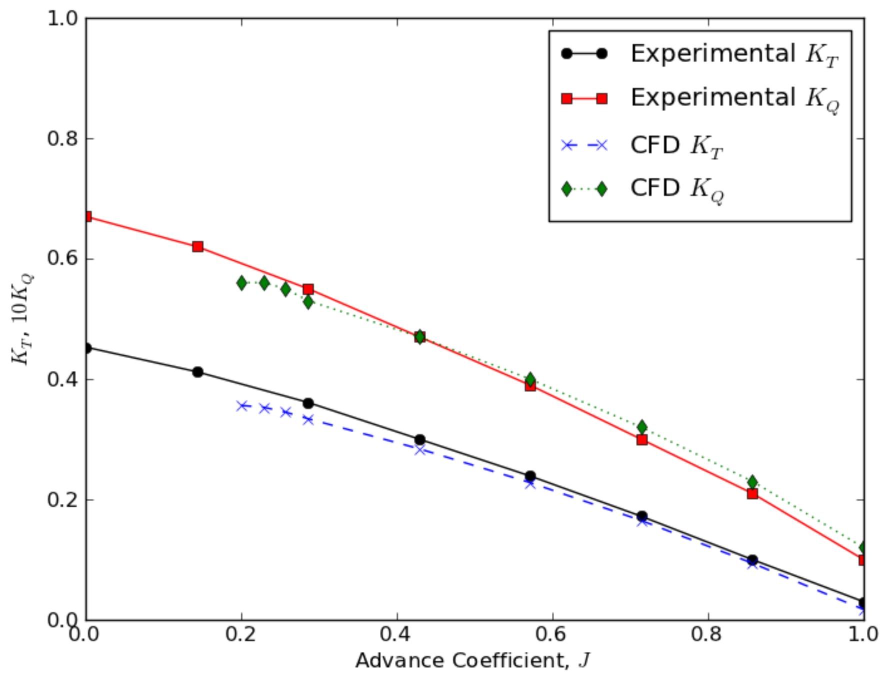


Figure 5.8: Validation against experimental data for the Wageningen B4-70 propeller using RNG k-ϵ turbulence model.

5.2 Steady Method Validation

Once the computational domain size and mesh resolution had been systematically verified, the work proceeded to investigate other advance ratios, thus allowing a performance characteristic of the propeller to be built and compared against the experimental data from MARIN. The results of the validation procedure are displayed in Figure 5.8. It was originally planned to simulate data points at 0.5 m/s advance speed intervals, however, the 0 m/s and 0.5 m/s advance speed cases did not converge, prompting further investigation. First, the advance speed was decremented in smaller 0.1 m/s intervals from the 1 m/s case until divergence occured. These results are plotted in Figure 5.8 and the slowest converged case was used to try and inform the reason for solution divergence. Different methods were then attempted to achieve convergence, as detailed in Section 5.2.1.

5.2.1 Convergence Problems at Low Advance Ratios

Using the RNG k-ϵ turbulence model at low advance speeds caused problems with solution convergence. To try and get the solution to converge a number of different methods were attempted. First, to see whether it was instability due to the use of second order discretisation schemes, a more diffusive but robust first order scheme

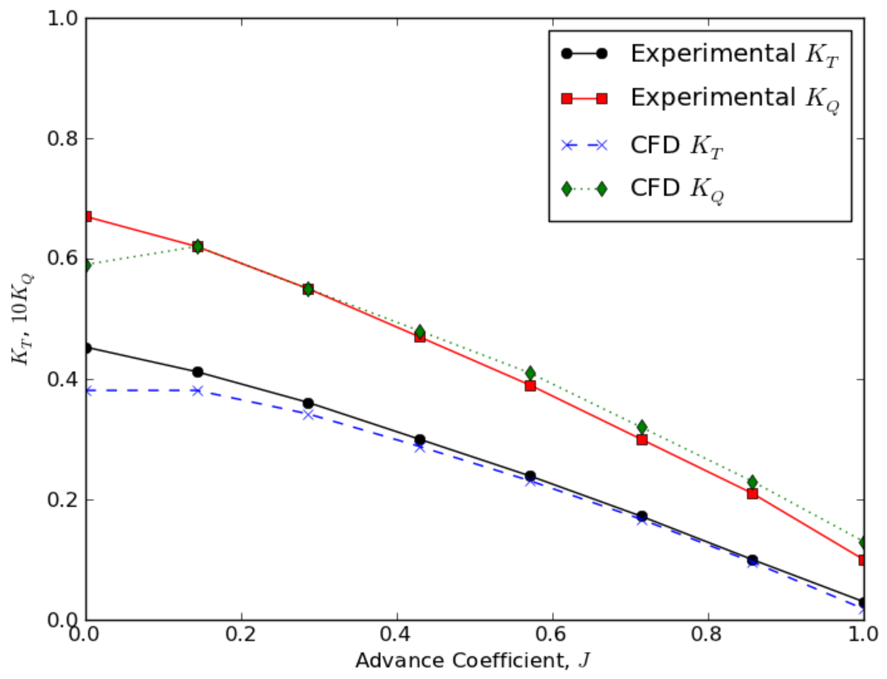


Figure 5.9: Validation against experimental data for the Wageningen B4-70 propeller using k-ω SST turbulence model.

was tried but this did not yield any success. Another approach was to try using a pre-converged solution (at higher advance speed) to seed the initial conditions of the flow field, however this also did not yield any success. Similarly, a slow increase of rotational speed, Ω, was tried with no success. Other numerical parameters were varied to try and improve the numerical stability including the timestep size and the relative tolerance (thus the number of inner iterations) but solution convergence was still not achieved.

One solution to the problem that worked was reducing the advance ratio, J, by increasing the rotational speed, Ω, rather than reducing the velocity. The solutions with increased Ω at 6000 rpm and 12000 rpm both converged, granting some insight into the source of the divergence. As the rotational speed is the dominant speed (*id est* larger than the advance speed), the primary difference between a solution at half the advance speed and a solution at double the rotational speed is the Reynolds number. At low advance ratios, the propeller blades are typically at a larger angle of attack to the incident flow, thus at lower Reynolds numbers, where viscous forces are more dominant, separation is more likely to occur. It was conjectured that it was this low speed separation that was the source of the instability and the k-ω SST turbulence model was tried as it handles low speed separation better.

As can be seen in Figure 5.9, the k-ω SST model did indeed succeed in getting converged results at low advance speeds. The validation also shows good agreement with the experimental data, thus increasing confidence in the capabilities of the computational fluid dynamics method. The slight differences between the experimental and CFD results can be primarily attributed to subtle differences between the experimental and computational geometries at the blade tips and the filleting at the blade roots. There is a large discrepancy between the experimental and numerical results at the bollard pull (0 m/s advance speed) condition and this is discussed in further detail in the following Section 5.2.2.

5.2.2 Bollard Pull Condition

Thrust produced when stationary is measured experimentally by a bollard pull test and the state of statically thrusting is often referred to as the bollard pull condition. In the validation results in Section 5.2.1, it can be clearly seen that the static thrust does not match the quoted experimental value. The propeller characteristics produced by the work of Abramowski et al. (2010) also feature a reduction in thrust at low advance ratios compared to experimental data. It is conjectured in the paper that the reason for this is a difference between the computational geometry and real geometry. However, this is not likely to be the cause as an error from a geometric discrepancy would be a systematic error, consequently it would reduce the thrust throughout the entire range of advance ratios and not just at the bollard pull condition.

A more likely explanation of the observed results is that the computational set up and boundary conditions do not realistically reflect what is occuring in the real world flow. As there is no inflow to begin with, the propeller must induce a flow through itself. However, there is no inflow to feed mass into the computational domain, thus the induced flow cannot flow out of the domain without violating mass conservation. The only remaining option is for the flow to recirculate, but this recirculation would typically require a larger computational domain than a moving case and thus it is possible that the computational domain is wrongly configured when extrapolated from a moving case to a static thrust case. As the computational domain extends a significant distance from the propeller, it is likely that even the recirculation induced when there is no advance speed should be able to move around the propeller unimpeded by the computational boundaries.

An alternative theory to the reduction in static thrust is that the experimental data for a pseudo-static thrust is obtained through either extrapolation or instantaneous thrust measurement to get a ‘true’ representation of the thrust at zero advance ratio. As the induced circulation (shown in Figure 5.10) and local velocity produced by static thrusting would mean that the local advance ratio is no longer zero and thus not technically correct despite having a global advance ratio that

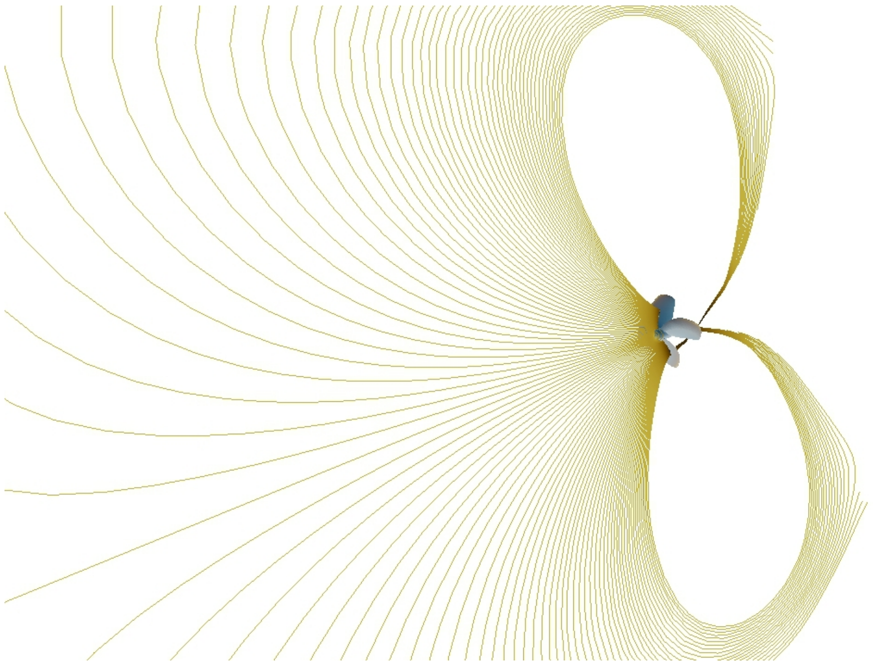


Figure 5.10: Streamlines showing recirculation about a Wageningen B4-70 propeller at bollard pull.

is zero. Details of how experimental bollard pull measurements are conducted in Carlton (2007) give a differentiation between the instantaneous bollard pull value and a continuous bollard pull value. It is thought that the experimentally reported values for the bollard pull condition are the instantaneous bollard pull condition, that is prior to any steady state recirculation being induced, and this is the reason for the difference between experimental data and computational fluid dynamics simulations, which by definition report the steady state bollard pull value.

5.2.3 Summary

A thorough verification of both mesh resolution and computational domain size has been conducted and a baseline computational domain for all steady-state simulations has been set. The required computational domain size for this exercise is approximately six propeller diameters to the walls, inlet and outlet, although double the required outlet distance was chosen in anticipation of high advance ratio cases where more momentum will be advected downstream.

It is interesting to find that this domain size is in excess of some simulations reported in the literature. For example, the simulations by Huang et al. (2007) used only a two propeller diameter radius and a total domain length of only five propeller diameters, less than the length to the inlet in the present work. This equates to a blockage ratio of 6.25%, higher than the recommended maximum of 1.5%. Similarly, Rhee and Joshi (2005) used only a domain with a radius of 1.43

propeller diameters, equating to a blockage ratio of 12.2%, and a total domain length of only 1.32 propeller diameters. In the work by Bensow and Bark (2010) the domain only had a domain width 1.5 times the propeller diameter, although in this case, the reasoning was to match the size of the experimental cavitation tunnel. The blockage is very significant in this case, a total of 44.4%, although in this situation it was replicating experimental conditions, which would have suffered the same amount of flow blockage in the cavitation tunnel. It may be the case that the boundary conditions used by the above works were more suited to a smaller domain, although other possibilities include a limitation of computational resources, as a larger domain would require both more memory and more computing time.

The mesh verification and the required surface resolution to properly represent the propeller in a computational space is visibly finer than the level of mesh resolution used by Celik and Guner (2007). A visual inspection and comparison of the mesh would suggest that the surface resolution is insufficient, although a trade off between speed of solution and accuracy must always be made in computational fluid dynamics, and while the results may not be experimentally accurate, their conclusions may be still be valid.

5.3 Unsteady Method Verification and Validation

For the purposes of gaining confidence in the numerical simulation, the verification and validation procedure for the unsteady method was conducted in the same systematic manner as for the steady state simulations. However as time-accuracy is also required, the time histories of force and torque are compared for each mesh level and domain size. As the test case of a Wageningen B4-70 propeller exhibits no unsteady phenomena, the results of unsteady simulation are rather trivial and comparable to the steady state. Hence the purpose of this procedure is purely to confirm the method functions as expected and as a learning process for the necessary tools.

Mesh level was found to be important to time accuracy as shown in Figures 5.11 and 5.12, although results for the Wageningen B4-70 propeller settle down to the same steady state result. A mesh level of 7, which corresponds to roughly 500,000 cells and a surface resolution of 0.27mm is almost as accurate as a mesh level of 8 which has roughly 1,200,000 cells and takes significantly longer to solve with only an increment of surface resolution to 0.14mm, thus a mesh level of 7 is a suitable compromise, though a higher resolution should be used if practicable.

The result is not as sensitive to distance to the outer boundary as it was for the steady simulations. It is shown in Figures 5.13 and 5.14 that the boundary need only be two propeller diameters away and is not severely affected by only being

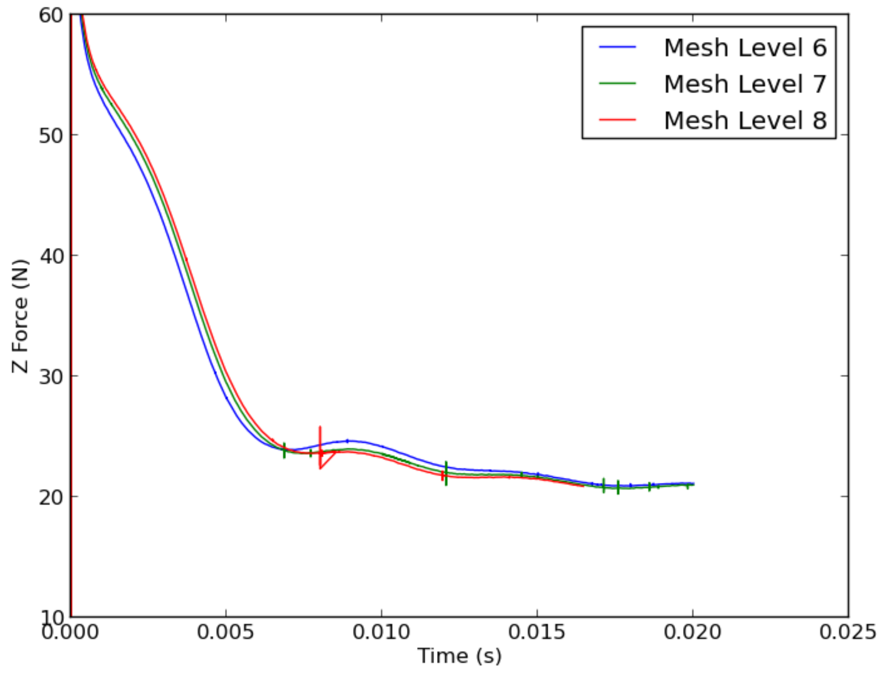


Figure 5.11: Force history for increasing mesh level.

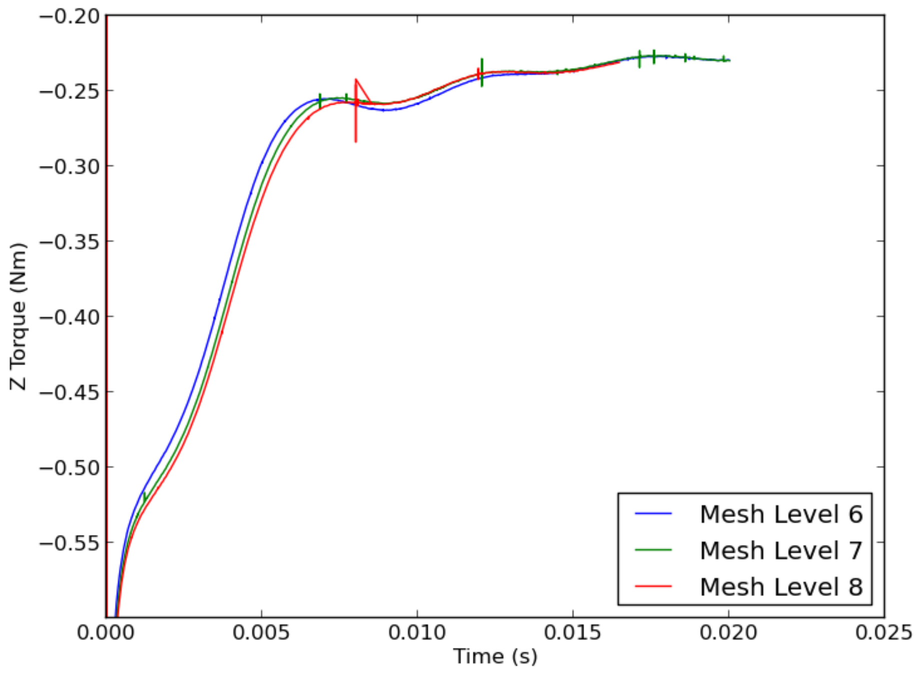


Figure 5.12: Torque history for increasing mesh level.

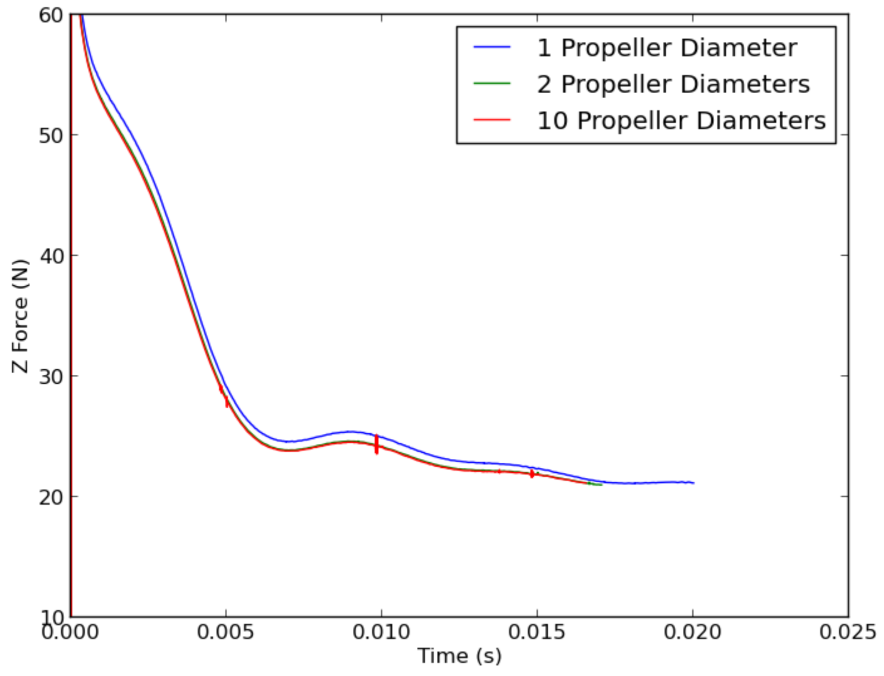


Figure 5.13: Force history for increasing distance to outer boundary.

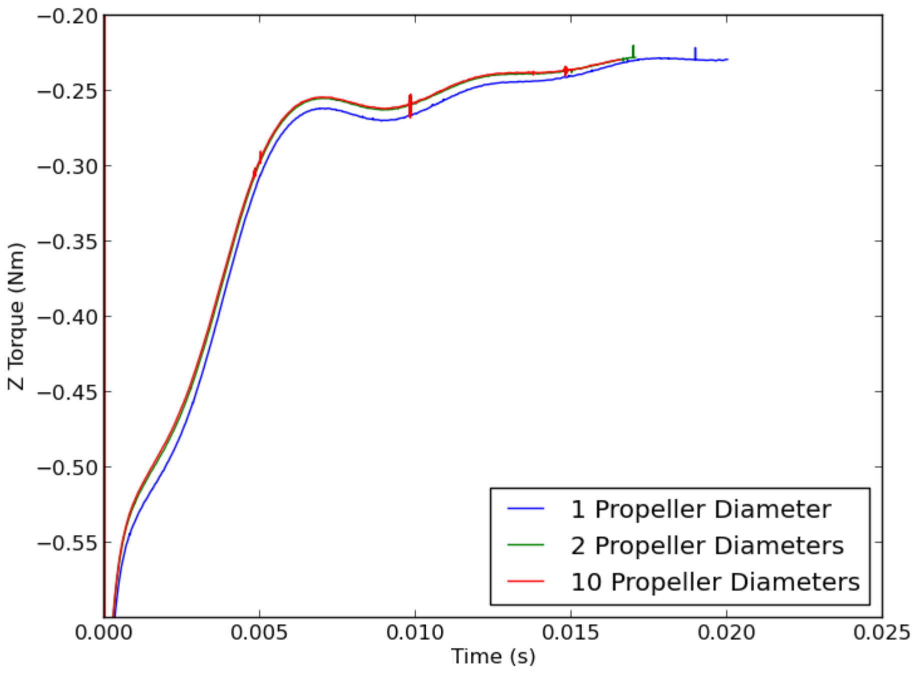


Figure 5.14: Torque history for increasing distance to outer boundary.

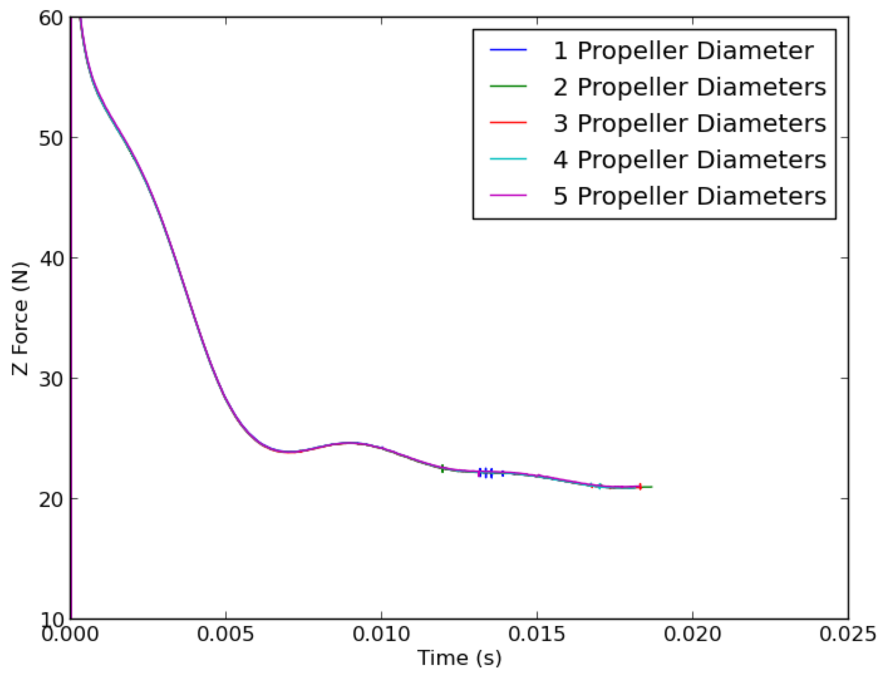


Figure 5.15: Force history for increasing distance to inlet boundary.

one propeller diameter away. A full range of propeller diameters were tested in the verification procedure, however as all the lines between two and ten propeller diameters are coincident, that is they lie over each other, it is not informative to plot them in Figures 5.13 and 5.14. A distance of two propeller diameters is also considered as best practice in experimental procedure and thus chosen as the domain size for the unsteady simulations.

Distance to the inlet boundary seems to have little impact on the results as shown in Figures 5.15 and 5.16. Consequently a distance of two propeller diameters to the inlet boundary was chosen, to allow for the duct flow in the rim driven thruster to be unaffected.

Outlet boundary distance also does not have much impact on the quantative value of the results but if it is too small it can lead to solution divergence as shown in Figures 5.17 and 5.18. As in Figures 5.13 and 5.14, a full range of domain sizes were tested and the unchanged lines between two and ten propeller diameters were omitted from Figures 5.17 and 5.18 for clarity. To allow for the wake at higher advance ratios than the verification test case, an outlet distance of four propeller diameters was selected.

For the unsteady simulations, with a distance to the boundary of two propeller diameters, the blockage ratio for this domain size is rather high at 4.9%. However,

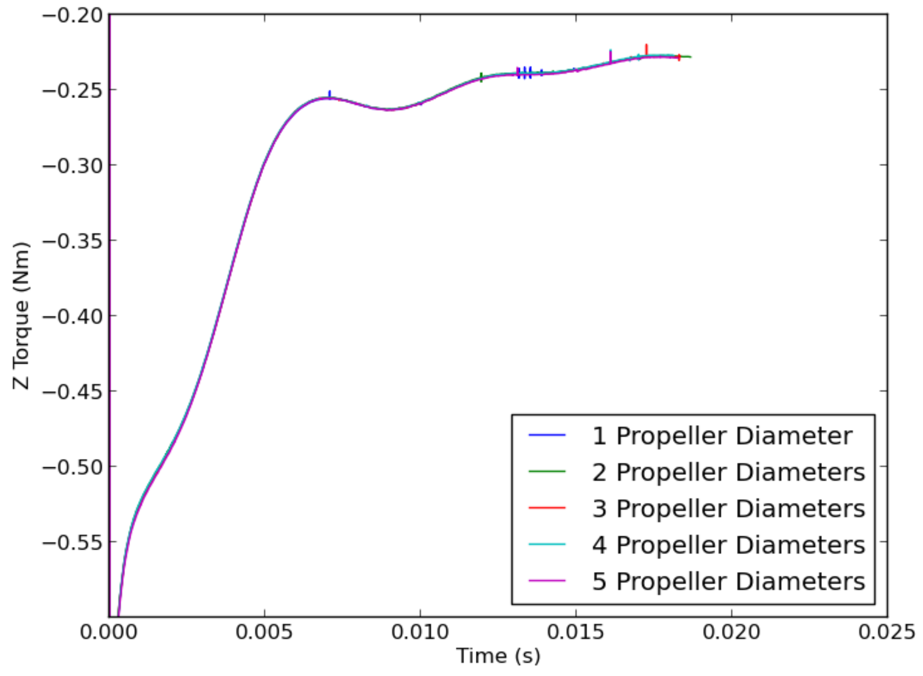


Figure 5.16: Torque history for increasing distance to inlet boundary.

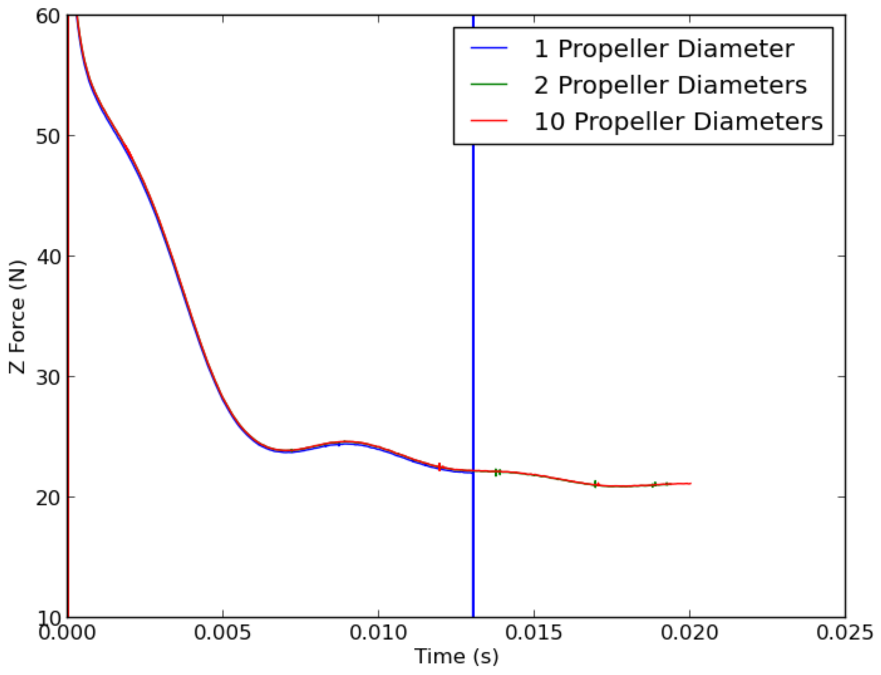


Figure 5.17: Force history for increasing distance to outlet boundary.

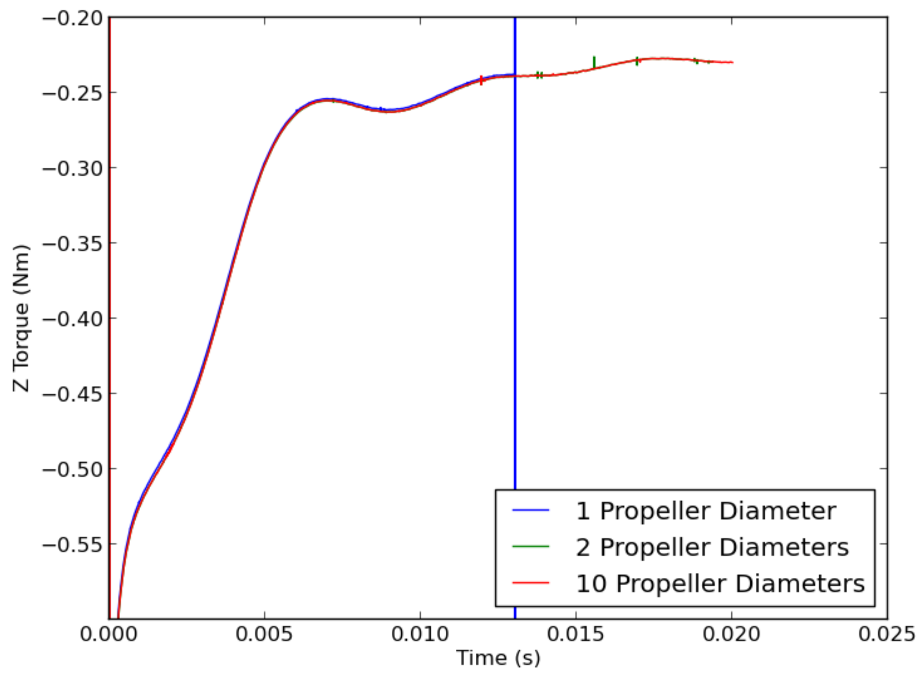


Figure 5.18: Torque history for increasing distance to outlet boundary.

as the verification procedure above has shown, this level of blockage has no impact on the simulation results.

Measurement	Pressure	Viscous	Total	Exp.	Error (%)
Thrust (N)	21.6021	-0.309489	21.292611	22.254	4.32
Torque (Nm)	0.2135	0.0180987	0.2315987	0.237	2.28

Table 5.1: Validation results for unsteady simulation at an advance ratio of 0.286

To validate the unsteady simulation, the results for the base case are compared against the experimental data for the same condition in Table 5.1. At a low advance ratio of 0.286, the error between the calculated thrust and published experimental data is 4.32% and for torque is 2.28%. This is reasonably close, especially when the differences between simulation and experiment at low advance ratios are considered, as previously discussed in Section 5.2.2.

5.3.1 Numerical Start Up of Wageningen B4-70 Propeller

To estimate how many revolutions need to be simulated before a periodic state is reached, the numerical start up of the Wageningen B4-70 propeller in a quiescent flow is shown in Figures 5.19 and 5.20. As a general rule of thumb in unsteady computational fluid dynamics, three revolutions are typically required to remove

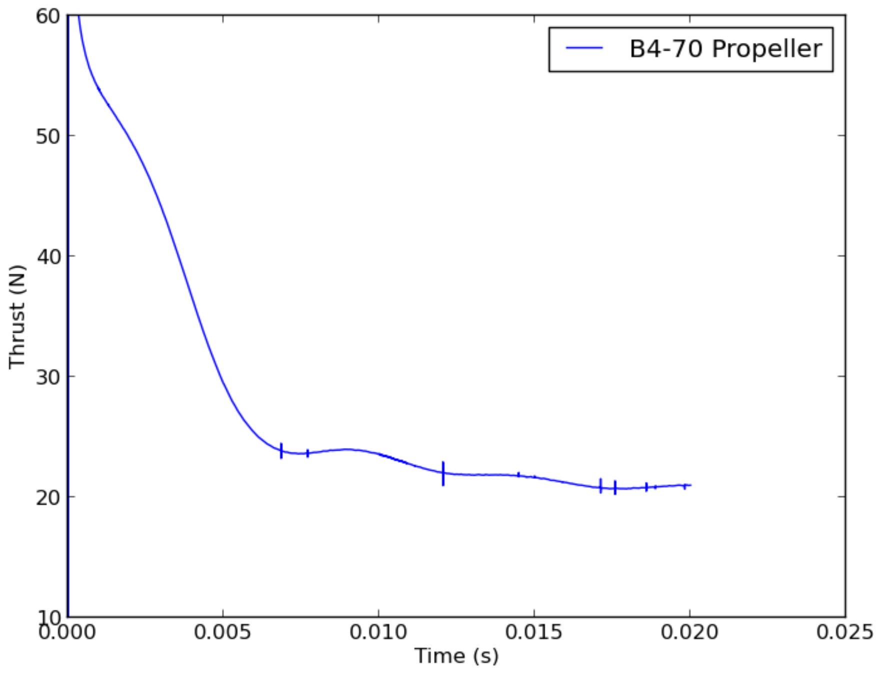


Figure 5.19: Force history during start up of Wageningen B4-70 propeller at 3000 RPM.

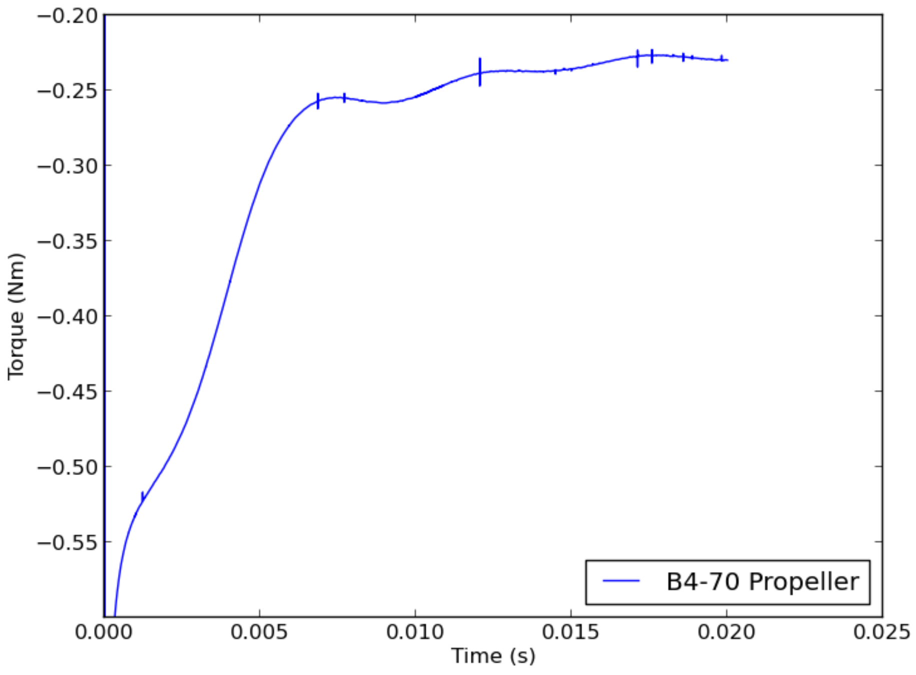


Figure 5.20: Torque history during start up of Wageningen B4-70 propeller at 3000 RPM.

any transience. This is also observed in the results for the unsteady simulation of the Wageningen B4-70 propeller, where a steady state is observed after three complete revolutions, which is expected as there are no unsteady features in the flow.

The computational results for the start up of the Wageningen B4-70 propeller in a quiescent flow shown in Figures 5.19 and 5.20 look incorrect at a first glance as a slow ramp up to the steady state value is expected. However, when the fact that immediately after start up there is no induced flow is considered, the results are more sensible. Without the induced axial flow, the momentary effective angle of incidence is higher, leading to the higher values of both thrust and torque before the induction of an axial flow tempers this status to a steady state. In reality the propeller does not experience such forces, as there is rotational inertia which is not modelled in the present work. If an inertialess propeller were to exist such that it would start up at the desired number of revolutions per minute instantaneously then the present results would be, at the very least qualitatively if not quantitively, correct. However, as the system takes time to increase rotational velocity, a different transient path to the steady state is observed in experiment. It is stressed that the start up observed here is purely a numerical phenomenon and not a representation of a real process.

Chapter 6

Results for Simulation of Rim Driven Thrusters

6.1 Preliminary Results for 70mm Rim Driven Thruster

After the verification and validation procedures had been completed on the standard series Wageningen B4-70 propeller geometry, preliminary work began on modelling the 70mm IntegratedThruster™. Figure 6.1 shows a meshed representation of the 70mm IntegratedThruster™coloured to show rotating (yellow) and static (blue) regions, with a (red) slicing plane to show the meshing of the internal field. This mesh comprised a total of 1,121,519 cells including 85,046 cells in the surface layer mesh.

As the preliminary results were initially obtained using the RNG k-ϵ turbulence model, there was no convergence for low advance ratios as also seen in Section 5.2.1. However, the results using the RNG k-ϵ turbulence model are shown in Figure 6.2, though no torque values are displayed for comparison as the available experimental data for the 70mm thruster are limited and do not contain torque measurements. However, it is clear from the thrust values that there is a large difference between the experimental data and the results from the computational fluid dynamics simulation.

These preliminary results may be improved upon when the k-ω SST turbulence model is used, and Figure 6.3 shows these results compared against those from the RNG k-ϵ turbulence model. It is shown that there is little difference between the turbulence models and this suggests that the difference between experimental and numerical results is not due to the turbulence modelling. The solution convergence at low advance ratios is shown in Figure 6.3 and thus k-ω SST was chosen as the sole turbulence model henceforth.

There are a number of sources for the difference between the experimental and computational results, aside from those aforementioned in the validation in Section 5.2, all of which contribute to a greater or less extent. Firstly, the frozen rotor

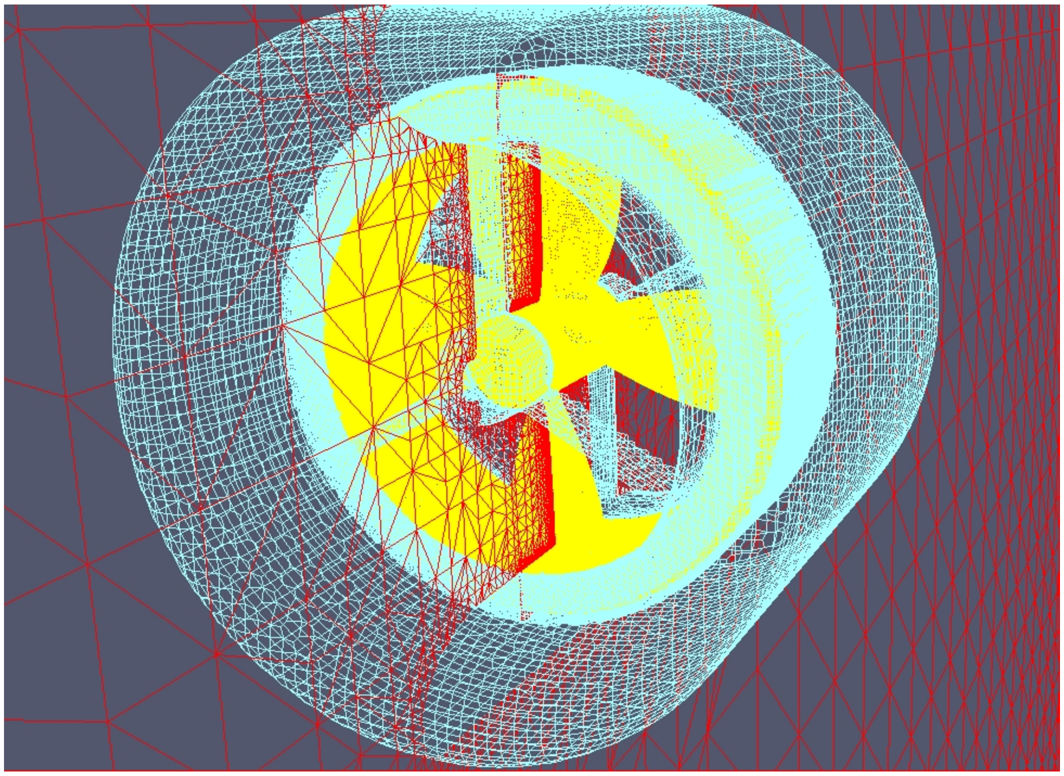


Figure 6.1: Meshed 70mm Integrated Thruster™.

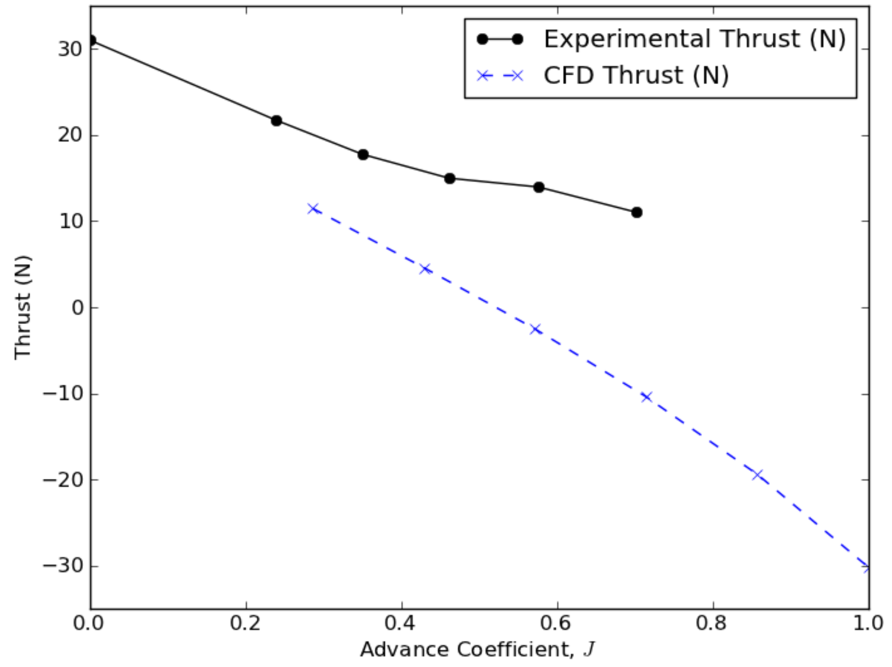


Figure 6.2: Preliminary results for the 70mm Rim Driven Thruster using RNG k-ϵ turbulence model.

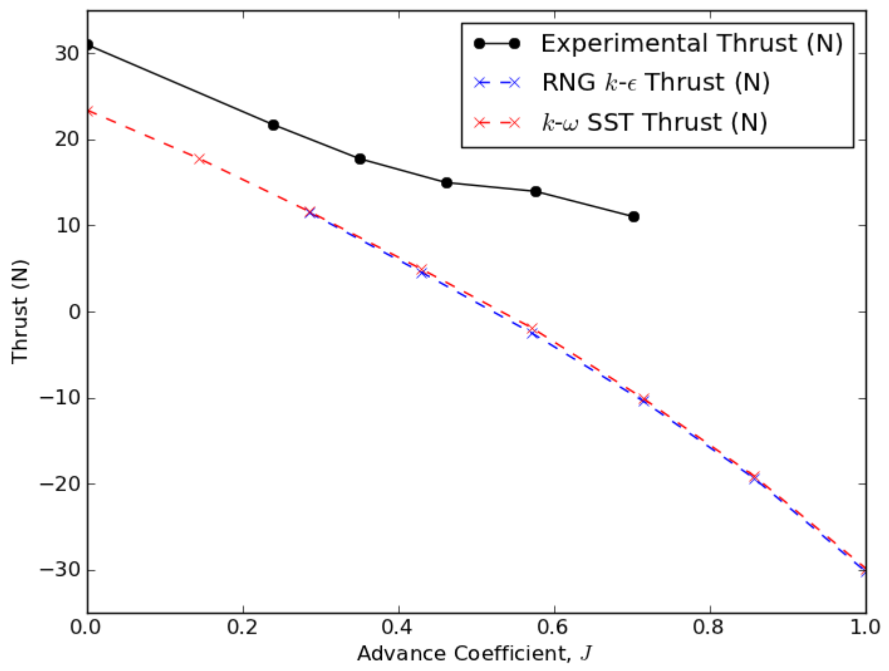


Figure 6.3: Preliminary results for the 70mm Rim Driven Thruster using k-ω SST turbulence model.

formulation for the interface between rotating and non-rotating reference frames may not be capturing the rotor-stator interaction correctly, leading to inaccuracies in the computational results. This phenomenon is covered in more depth in Sections 6.1.1 and 6.7.2, and is shown to be a significant source of error in the steady state simulation of rim driven thrusters. The way the annulus is modelled and resolved in the simulation is also another source of discrepancy, as it is a complex region of the flow and a correct estimation of torque and thrust produced in this region is critical to performance prediction. Calculating the torque and thrust in the annulus region whilst keeping computational expense down is covered in further in Section 6.3.

Finally, there is a source of error that does not derive from the use of computational fluid dynamics and that is the experiments. The lack of torque measurements and the unexpected step in the thrust trendline observed at an advance ratio of 0.6 in the experimental data, but not in the computational data, seen in Figure 6.2 impart a reduced confidence in the validity of the experimental results. There is also some evidence, in a project by Nimmo (2011), to suggest that the experimental method is flawed and may overpredict the thrust by virtue of overestimating the drag on the supporting structure in the towing tank tests.

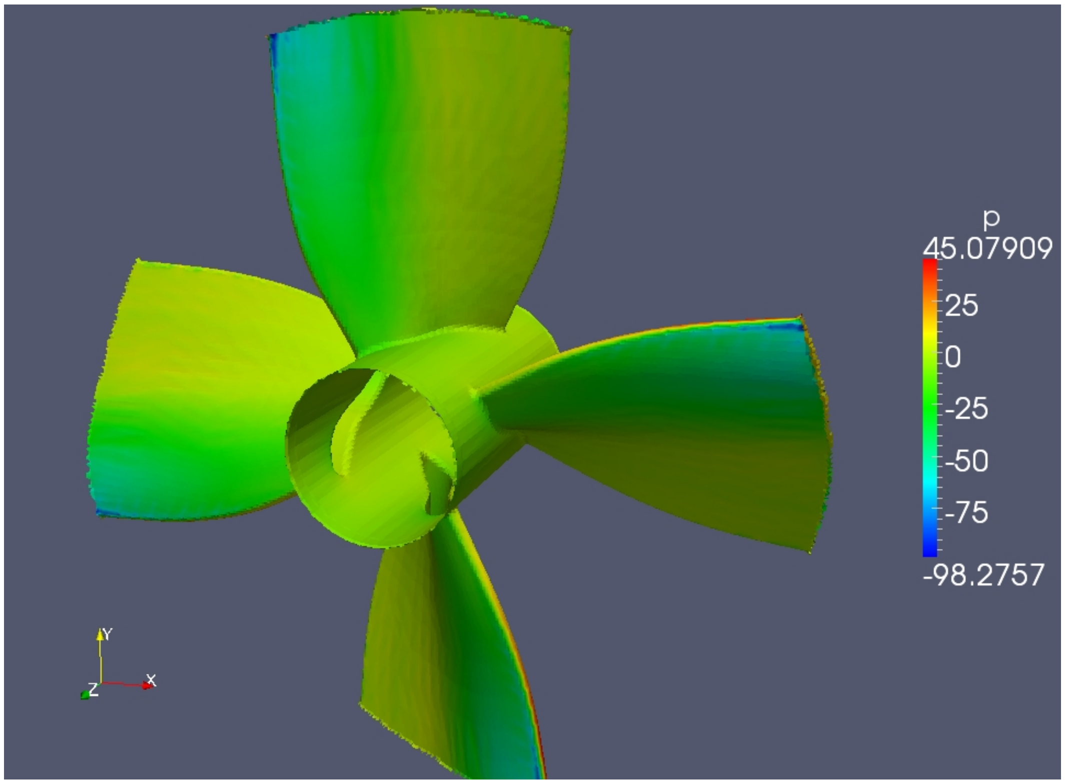


Figure 6.4: Preliminary results for the 70mm Rim Driven Thruster showing the pressure distribution on the blades. Units are simulation units of m^2/s^2 which are normalised against density due to incompressibility.

6.1.1 Analysis of Preliminary 70mm Rim Driven Thruster Results

Although there are differences between the experimental data and the simulation results, some insights can be gained from looking at qualitative features of the flow. Figure 6.4 shows the pressure distribution over the blades of the device, where the highly negative pressure region at the leading edge of the tip shows that the blade is working very hard here. The pressure towards the trailing edge of the blades shows that very little, if any, propulsive force is being generated here. The reason for this result is partly due to the design of the blade section being symmetric for bi-directional operation, although the hard working blade tip could be a result of the duct reducing the inflow velocity at the tip and thus increasing the local angle of attack. This would suggest that some gain in efficiency could be produced through optimising the blade pitch distribution for the augmented inflow caused by the ducting.

One of the reasons for the difference between computational and experimental results was thought to be due to the ‘frozen rotor’ formulation of the interface in MRFSimpleFoam, which was discussed previously in Section 6.1. The effect of the frozen rotor was proven by changing the relative position of the rotor to the stators

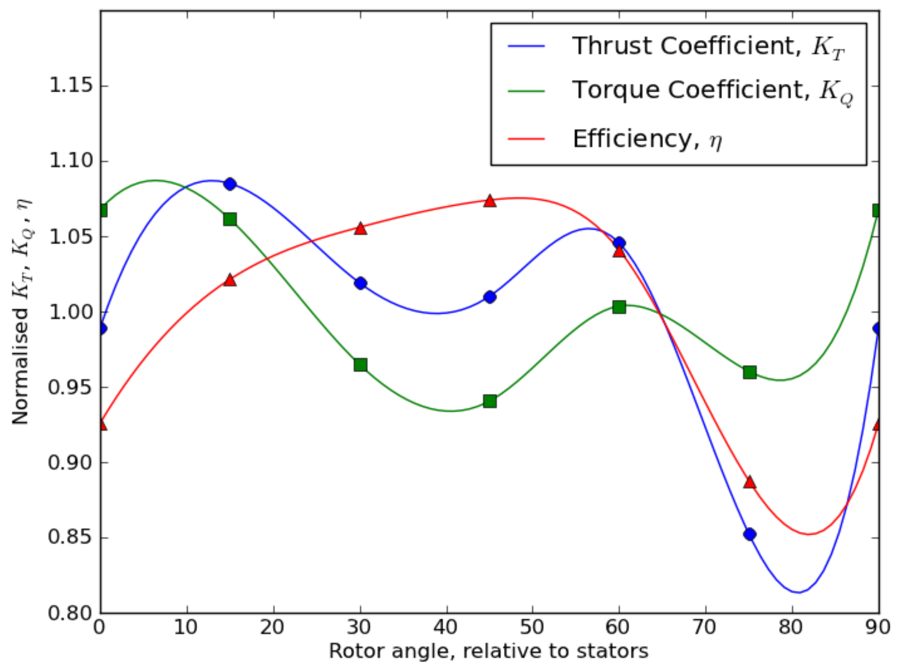


Figure 6.5: Change in simulation results for thrust, torque and efficiency as rotor position is varied.

and solving for every 15 degrees. Only six simulations were required as the geometry is rotationally periodic every 90 degrees and the results of this are shown in Figure 6.5. There is a fluctuation of up to 20% from the average in the results showing that a single result cannot account for the average performance in the device.

However, if an average is taken of all the individual rotor positions for each advance ratio, some interesting data can be derived from the steady state simulations. Figure 6.6 shows the variation of the average components of thrust force, which is drag when negative, as the advance ratio changes. Similarly, the sources of torque can be broken down into their component parts as shown in Figure 6.7. Here torque on the blades is coloured blue, which can be considered as the losses incurred due to making thrust, and torque on the rim is coloured green, which can be considered primarily as the hydrodynamic losses in the annulus region.

The parts in Figure 6.7 can be further broken down into pressure based torque, that is the moment derived from wall normal stresses, and viscous based torque, that is the moment derived from wall parallel shear stresses. Torque on the blades is primarily from static pressure acting on the blade and, as expected, the torque on the rim is predominantly from shear stress, or skin friction.

The reduction of thrust on the blades as the advance ratio increases shown in Figure 6.6 is simply explained by the moving of the propeller blades away from

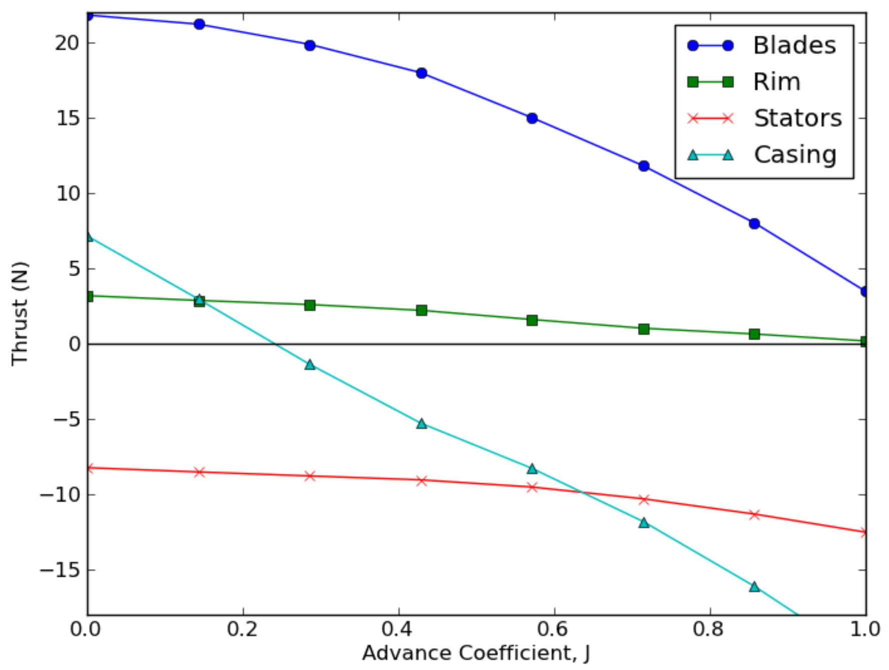


Figure 6.6: Breakdown of thrust/drag sources of the 70mm Rim Driven Thruster against advance ratio.

their design condition, thus the blade sections produce less lift, which in turn equals less thrust. Similarly if the propeller blades were approximated to an actuator disc, the decrease in thrust would be manifest as a reduction in pressure difference across the disc and this reduced pressure differential would also be evident on the rim, thus reducing the rim thrust in proportion with the blades. The increased rearward pressure caused by the propeller blades also manifests itself on the duct, contributing to the large forwards thrust at low advance ratios, but this is quickly swamped by the drag experienced as the advance ratio, and consequently forward velocity, increases.

6.2 Raw Results for the 100mm Rim Driven Thruster

Following the preliminary work on the 70mm rim driven thruster, it was decided that the experimental data for the 70mm device were not sufficiently thorough nor was there sufficient confidence in the accuracy of the data and there were no values of uncertainty in the experimental data available either. All subsequent work was conducted using a 100mm thruster geometry as the available experimental data included estimates of hydrodynamic torque, calculated by subtracting the known losses in the electrical machine from the total power consumption and then dividing

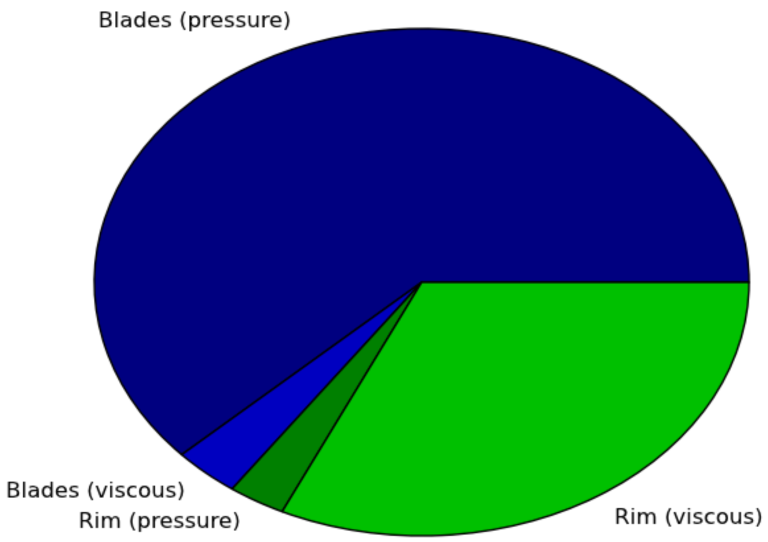


Figure 6.7: Breakdown of torque sources of the 70mm Rim Driven Thruster.

by the rotation rate. The experimental results for the 100mm thruster are considered to better reflect the actual performance of the device, however there is still likely to be some overprediction of the thrust as covered in Section 6.1.

The simulations for the 70mm thruster included the annulus region, that is the axial and radial gap between the rotor rim and ducting, also referred to as the rim gap. To keep the mesh at a tractable number of cells, and at a resolution in keeping with the validation study, the transverse (*id est* wall normal) resolution of the rim gap was only a total of four cells. Full resolution of the annulus, that is with a minimum of 40 cells across the gap, in the current geometry would require in the order of tens of millions of cells as well as additional temporal requirements for the resulting internal flow regime to reach a steady state (see Batten (2002) for a full account of computational fluid dynamics modelling of Taylor-Couette flow).

As solution time is ideally to be minimised to reduce optimisation turn-around time, subsequent simulations for the 100mm thruster were carried out with the annulus region excluded from the computational fluid dynamics calculation. Instead, the effect of the annulus region on both the torque and the thrust was modelled separately, thus reducing the computational cost of the simulations, without any significant penalty to accuracy as the annulus region was previously not sufficiently resolved to be considered accurate. The choice of model for the annulus is discussed

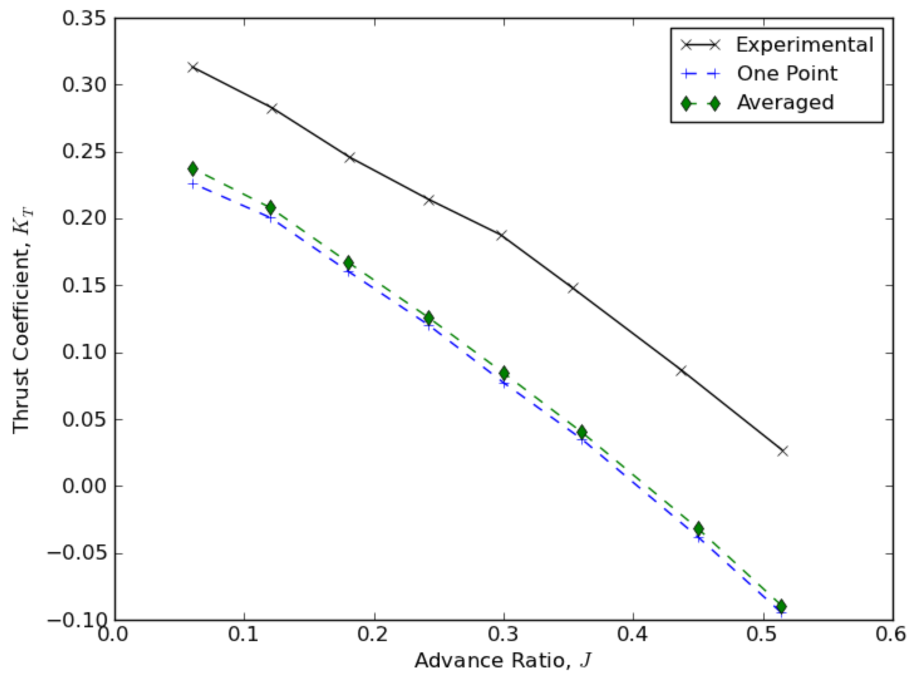


Figure 6.8: Raw thrust coefficient data for the 100mm Rim Driven Thruster without annulus models.

further in Section 6.3.

Figures 6.8, 6.9 and 6.10 show the raw computational fluid dynamics data for the thrust coefficient, torque coefficient and efficiency respectively. They show the difference between evaluating the performance for one rotor position and averaging over multiple rotor positions, thus confirming the inaccuracy of a single frozen-rotor simulation hypothesised in Section 6.1.1. Compared to the experimental data for thrust and torque, it is clear that averaging over multiple rotor positions generally produces a more accurate result. Although the change to thrust is marginal, the torque prediction is closer to the experimental data, however a benefit to efficiency accuracy is neither clearly nor systematically shown.

It is worth noting here that the efficiency of the 100mm thruster is quite low at only 20% . This is primarily a function of the size of the device, as it is well known that larger diameters increase efficiency. The diminuitive size of the 100mm thruster means a lower Reynolds number flow and consequently the viscous losses are proportionally higher than for larger propellers.

A deficit of thrust and torque is shown in Figures 6.8 and 6.9 respectively, which is also expected from the raw data as the annulus models are not included, and these would augment both the thrust and torque closer to the experimental results. The effect of the annulus models on the efficiency, as depicted in Figure 6.10, is

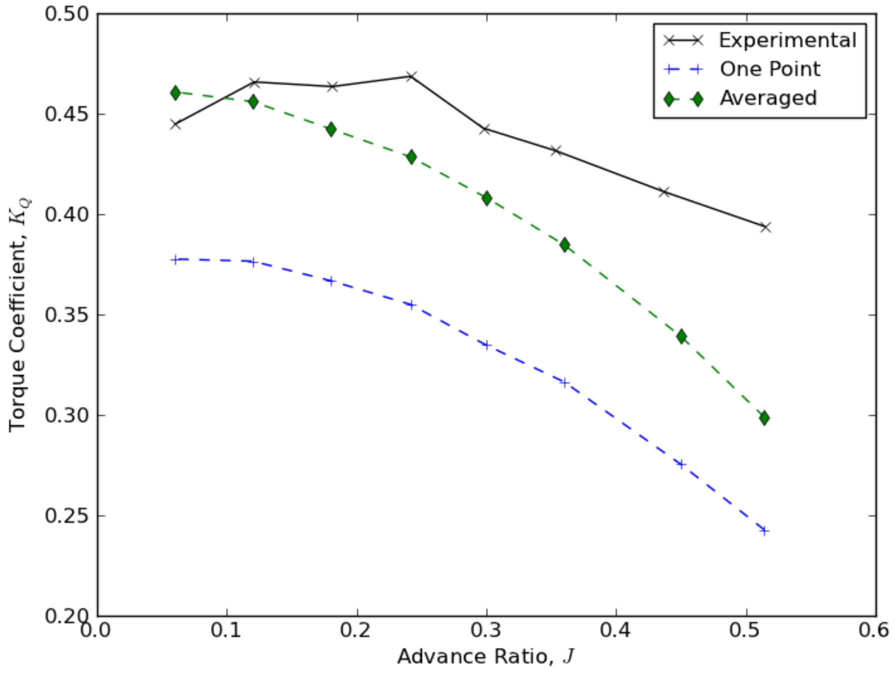


Figure 6.9: Raw torque coefficient data for the 100mm Rim Driven Thruster without annulus models.

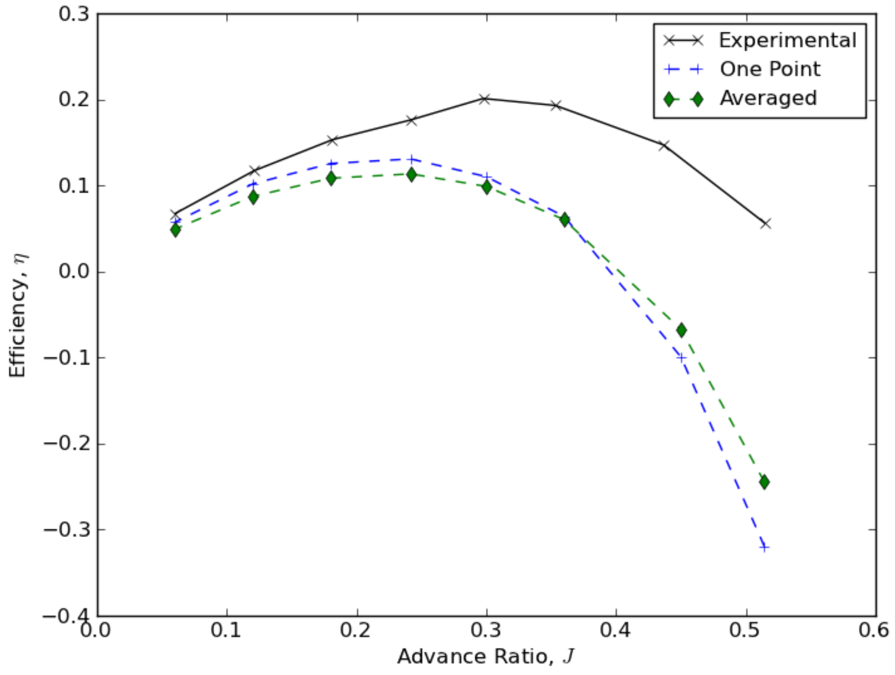


Figure 6.10: Raw efficiency data for the 100mm Rim Driven Thruster without annulus models.

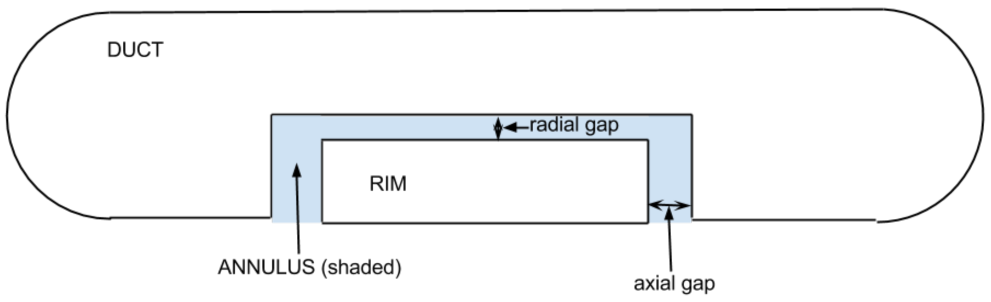


Figure 6.11: Diagram of annulus showing radial and axial gaps.

not possible to predict as it is dependent on the relative magnitudes of thrust and torque produced in the annulus.

6.3 Analytical Annulus Models

There are a number of effects produced by the annulus flow that need to be captured. The most important effect relating to device performance is the viscous friction on the rotor that causes significant torque losses in the device. However, as there is an axial pressure gradient, there is an axial flow in the thrust direction that is also subjection to friction losses. However, the friction 'losses' in this case resolve to a force in the axial direction contributing to the thrust in a positive manner and increasing the performance of the thruster. It is also possible in some cases that the axial pressure gradient acts to reduce the torque losses in the rim gap (Manna and Vacca, 2009) by modification of the tangential velocity profile.

Typically analytical models of the annulus either cover the radial gap, which is a Taylor-Couette flow, or the axial gap (*id est* the front and back faces of the rotor, see Figure 6.11) and none have been found that consider the interactivity of the two flow regimes. Consequently, we shall look at the models for the two areas separately below.

6.3.1 Radial Gap Models

Three models are defined in this section, the first two for torque estimation and a third for axial viscous forces. The first model is derived here from the assumption of a linear velocity profile as a method for derivation of torque based on velocity profile. The second radial gap model given is the Bilgen and Boulos model, which is empirically derived and subsequently used as a more accurate model than a linear velocity assumption. The third model is given to attempt axial viscous force estimation.

To begin with, an approximation of the torque losses can be made by assuming the velocity profile in the gap is linear. In practice this is not the case, but a linear approximation gives a best-case scenario, that is the one that gives the minimum torque out of the possible velocity profiles.

The skin friction, or the wall shear stress, τ_w, that contributes towards the torque losses in the annulus, can be found for a Newtonian fluid from the following relationship:

$$\tau_w = \mu \frac{\partial U_\theta}{\partial r} \tag{6.1}$$

where μ is the dynamic viscosity, U_θ is the rotational velocity and r is the radial co-ordinate.

As there is an assumption of a linear velocity profile, the velocity gradient at the wall is equal to the gradient across the rim gap. Using the relationship $U_\theta = \omega r$, at the outer wall ($r = r_2$) the non-slip condition requires that $U_\theta = 0$ and at the inner wall ($r = r_1$) the velocity is $U_\theta = \omega r_1$. Also, the change in r between these two velocity conditions is $r_2 - r_1$, which can be substituted into Equation 6.1 to yield the following:

$$\tau_w = \mu \frac{0 - \omega r_1}{r_2 - r_1}.$$

From the equation for the wall shear stress, the torque can be derived by multiplying by the area, $\pi {r_1}^2$, and the moment arm, r_1. This yields Equation 6.2 below:

$$Q_{rim} = \frac{-\mu \omega \pi {r_1}^4}{r_2 - r_1} \tag{6.2}$$

where Q_{rim} is the torque on the rotor due to the radial annulus gap.

Perhaps the most comprehensive experimental study on the torque produced in the annulus of a rotating machine was conducted by Bilgen and Boulos (1973). By combining their own and other's experimental data, four regimes based on gap Reynolds number (Equation (6.3)) were identified and empirical relationships were derived for each of these. The four regimes and their equations are as follow:

$$Re = \frac{\rho \omega r_1 (r_2 - r_1)}{\mu} \tag{6.3}$$

$$C_M = 10(\frac{r_2 - r_1}{r_1})^{0.3} Re^{-1.0}, \quad \text{with } Re \leq 64 \tag{6.4}$$

$$C_M = 2(\frac{r_2 - r_1}{r_1})^{0.3} Re^{-0.6}, \quad \text{with } 64 < Re < 500,\ (r_2 - r_1)/(r_1) > 0.07 \tag{6.5}$$

$$C_M = 1.03(\frac{r_2 - r_1}{r_1})^{0.3} Re^{-0.5}, \quad \text{with } 500 \leq Re \leq 10^4 \tag{6.6}$$

$$C_M = 0.065(\frac{r_2 - r_1}{r_1})^{0.3} Re^{-0.2}, \quad \text{with } Re > 10^4 \tag{6.7}$$

where ρ is the density of the working fluid.

For completeness it is necessary to define the moment coefficient in relation to torque as multiple definitions exist and it is not immediately clear which one is used. Throughout this work, the definition used in Bilgen and Boulos (1973) will be taken, and the torque is calculated as in Equation (6.8):

$$Q_{rim} = \frac{1}{2}\rho\pi\omega^2 r_1^4 L C_M \tag{6.8}$$

where L is the (axial) length of the rim and C_M is the moment coefficient.

As there is a pressure differential across the annulus, it follows that an axial flow is induced, which in turn will exert a shear force on the thruster, both through axial friction on the rim and friction on the outer casing. An estimate of the contribution of the annulus to the thrust can be made by first assuming that the pressure differential is equal to the ratio of shear force to flow area as per Equation (6.9) which can be rearranged for thrust as Equation (6.10).

$$\Delta P = T_{\text{gap}}/A_{\text{gap}} \tag{6.9}$$

$$T_{\text{gap}} = \Delta P/A_{\text{gap}} \tag{6.10}$$

where ΔP is the pressure differential across the annulus, T_{gap} is the thrust contribution of the annulus and A_{gap} is the flow area through the annulus. To resolve this in terms of the thruster geometry, the flow area can be expressed as:

$$A_{\text{gap}} = (\pi r_2^2 - \pi r_1^2) \tag{6.11}$$

where r_1 and r_2 are the inner and outer annulus radii respectively. Finally, if the blade thrust generation is modelled as an actuator disc, then an expression for pressure differential can be derived from the pressure thrust on the blades, T_{blades}, divided by the blade disc area:

$$\Delta P = T_{\text{blades}}/\pi r^2 \tag{6.12}$$

Combining Equations (6.10), (6.11) and (6.12) will yield the following equation for the annulus thrust in terms of geometry and thrust generated by the blades:

$$T_{\text{gap}} = \frac{r_2^2 - r_1^2}{r^2} T_{\text{blades}} \tag{6.13}$$

6.3.2 Axial Gap Models

Two models are also given in this section, with the first again based on a linear approximation to ascertain a lower bound for torque and the second empirically derived by Daily and Nece which is more accurate and ultimately chosen.

In a similar fashion to the radial gap, a first approximation of the torque contribution of the axial gap can be made by assuming a linear velocity profile. Starting from Equation 6.1 and apply the non-slip condition at the duct axial wall, $U_\theta = 0$, and at the rotor axial wall, $U_\theta = \omega r$, where r in this case varies along the axial wall. Then, if the distance to the duct and rotor axial walls are given by l_2 and l_1 respectively, the equation for the wall shear stress becomes:

$$\tau_w = \mu \frac{0 - \omega r}{l_2 - l_1}.$$

To get the force, the wall shear stress must be multiplied by the area, and then be multiplied by the moment arm to get the torque. However, both the wall shear stress and moment arms are a function of radius and thus the torque is given by the following integral:

$$Q_{axial} = \int_{r_1}^{r_2} \frac{-2\mu\omega\pi}{l_2 - l_1} r^3 \mathrm{d}r$$

which when evaluated results in:

$$Q_{axial} = \frac{-\mu\omega\pi(r_2^4 - r_1^4)}{2(l_2 - l_1)}$$

where Q_{axial} is the torque on the rotor due to the axial annulus gap.

Finally, considering that there is both a front and rear axial face on the rotor, the torque contribution is thusly doubled, leading to Equation 6.14:

$$Q_{axial} = \frac{-\mu\omega\pi(r_2^4 - r_1^4)}{l_2 - l_1} \tag{6.14}$$

which is very similar to Equation 6.2 which was derived under the same assumptions.

In an attempt to measure the friction on the axial faces of a spinning disc, Daily and Nece (1960) found four different empirical regimes based on a tip Reynolds number (Equation (6.15)) and a spacing ratio, which using the present nomenclature is defined in Equation (6.16).

$$Re_r = \frac{\rho\omega r_2^2}{\mu} \tag{6.15}$$

$$s_r = (l_2 - l_1)/r_2 \tag{6.16}$$

where Re_r is the tip Reynolds number and s_r is the spacing ratio. As the spacing ratio is very small and the tip Reynolds number very high, only one regime applies to the 100mm rim driven thruster, whose moment coefficient is given by Equation (6.17) below:

$$C_M = \frac{0.16}{\left(\frac{l_2 - l_1}{r_1}\right)^{0.167} Re_r^{0.25}} \tag{6.17}$$

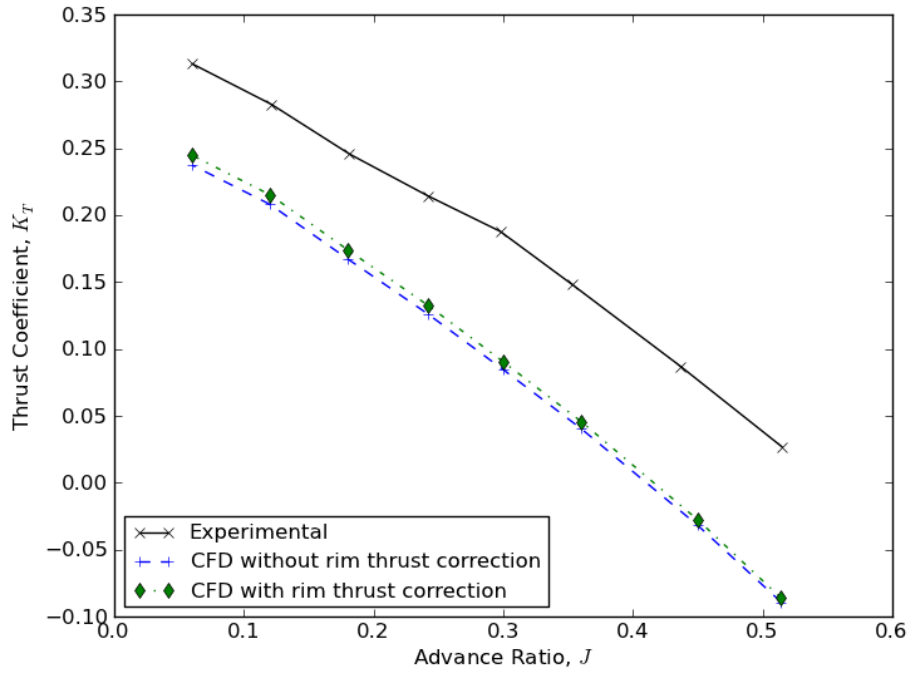


Figure 6.12: Thrust coefficient for the 100mm Rim Driven Thruster.

6.4 Results for 100mm Rim Driven Thruster Including Annulus Models

Applying the annulus models to the CFD results for the 100mm rim driven thruster (Figures 6.8, 6.9 and 6.10) gives the thrust coefficient, torque coefficient and efficiency curves shown in Figures 6.12, 6.13 and 6.14 respectively. The original raw data, averaged over multiple rotor positions, are included in the Figures for comparison.

In Figure 6.12 it is shown that even with the proposed rim thrust model for the annulus, the increased thrust does not match the experimental data. The discrepancy between the results can be attibuted to either inaccuracy in the thrust model, computational fluid dynamics simulation or the experimental results, this is discussed further in Section 6.4.1.

The results including the torque models are shown in Figure 6.13, where the experimental data lie between the results for the simulation and the results including the Bilgen and Boulos model alone. With both annulus torque models included, the results are significantly higher than the experimental results, which can be attributed to the interactivity between the flows in the radial and axial gaps not being included. It is also possible that the presence of the axial pressure gradient also affects the

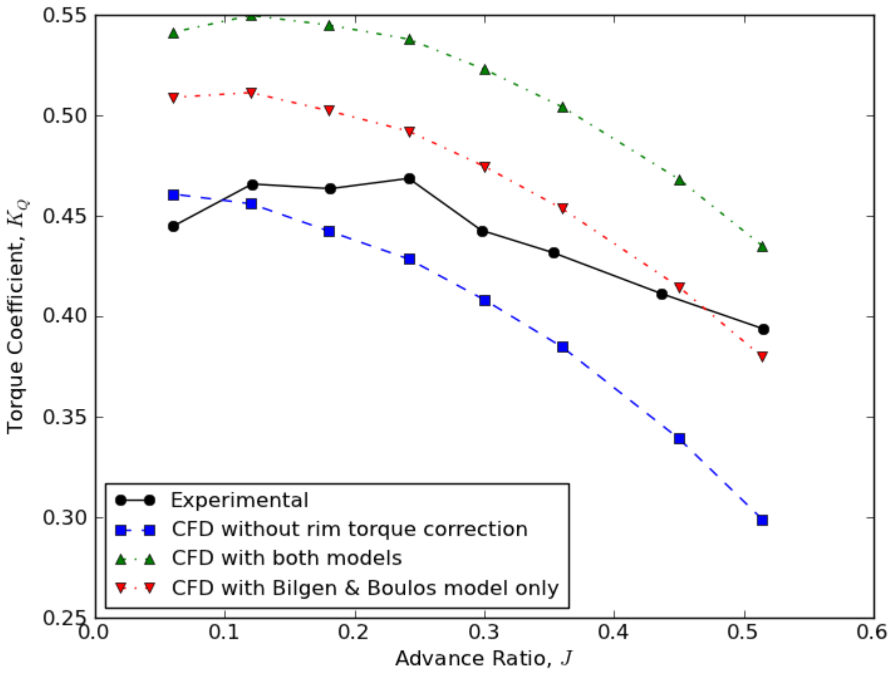


Figure 6.13: Torque coefficient for the 100mm Rim Driven Thruster.

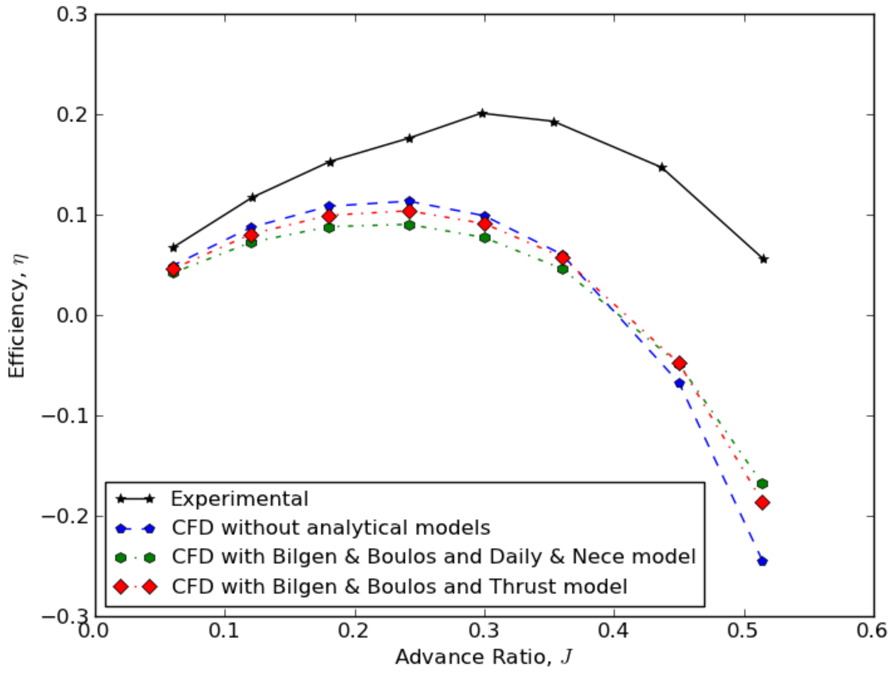


Figure 6.14: Efficiency for the 100mm Rim Driven Thruster.

flow and this is discussed further in Section 6.4.1.

Finally, Figure 6.14 shows how the efficiency varies with various combinations of annulus models. All combinations show a similar trend to the experimental results, but also under-predict the device efficiency. This deficiency in efficiency is a consequence of the thrust under-prediction shown in Figure 6.12.

6.4.1 Annulus Analytical Modelling Summary

Preliminary results attempted to simulate the annulus flow within the rim driven thruster, which produced some interesting results, including the ability to break down the origin of components of thrust and torque losses. However, as the annulus gap is very narrow, only four cells of resolution spanned the gap, and performance predictions here will be inaccurate and the complex flow regimes will not be simulated.

Analytical models were chosen to replace the explicit simulation of the annulus, these have the advantage of 'capturing' different flow regimes in the annulus through regime dependent equations and are also computationally inexpensive to evaluate. However, as the axial gap and radial gap were studied in isolation of each other and any thrust producing surfaces, the analytical models do not account well for the interactivity between the axial and radial gap flows or for the effect of the axial pressure gradient.

A compromise between the two above solutions is proposed and outlined in the following Section (6.5), where a separate computational fluid dynamics simulation is used to exploit the rotational symmetry and evaluate the performance contribution of the annulus including flow interactivity and the pressure gradient. This method would be preferable to the 'brute force' method of increasing the resolution of the preliminary results to accurately resolve the annulus region, which is estimated to require in excess of ten million cells.

6.5 Computational Fluid Dynamics Simulation of the Annulus

A further method for estimating the performance contribution of the annulus, without the extensive computational resource use of including it within a full simulation, is to use a separate computational fluid dynamics simulation. Removed from the simulation of the rim driven thruster, a computational fluid dynamics simulation of the annulus can exploit its rotational symmetry. In this way, both the interactivity of the axial and radial gap flows can be accounted for as well inclusion of the effects of the pressure gradient upon the flow. As the simulation is not trivial (Batten, 2002), only a proposed method is outlined here without any verified results having been obtained.

A computational fluid dynamics simulation of the annulus could be set up as a pressure driven flow with a single rotating inner wall and one static outer wall. The rotational symmetry means that only 1/36th of the domain needs to be simulated, that is a ten degree wedge, although this requires verification. The outlet boundary condition is a pressure outlet, set to a gauge pressure of zero, and the inlet boundary condition is a pressure inlet, its value derived through reverse engineering the pressure based thrust on the blades using actuator disc theory as in Equation (6.12).

If a sufficient number of operating conditions are pre-solved for a fixed geometry of annulus, then a further gain in computational efficiency without significant sacrifice of accuracy is possible. Two two-dimensional surrogate models could be created from pre-solved data points for thrust and torque contributions of the annulus. For a fixed geometry only two independent variables are required; the pressure differential and the rotation rate. This allows for a far quicker estimate of the annulus performance contribution, however accuracy may be reduced in regions where the flow changes regime.

6.6 Steady State Simulations Summary

It is clear from the results of the steady state simulations that the rotor-stator interaction is not correctly captured by the frozen rotor formulation in MRFSimpleFoam. However, the method has been thoroughly verified and validated and some confidence is gained from the results, if the impact of the rotor-stator interaction is neglected. Discrepancies at low advance ratios betwen computational and experimental data are considered primarily to be due to differences in methodologies and whether the induced flow is included as part of the bollard pull condition or not.

The primary contribution of these initial steady-state results is to inform the parameter selection in future design optimisation studies. It is shown that pitch distribution, duct profile and rim gap geometry are likely to yield best results when optimised. The stators also have an unquantified impact, however as they cannot be modelled satisfactorily using a steady state method, they must be excluded from the design optimisation study, as full unsteady simulations would be too time consuming for such an iterative procedure.

As the steady state simulations do not capture the rotor-stator interaction correctly, this prompted an investigation into the contribution and significance of this flow feature using the unsteady simulation method, detailed in Section 4.8. The results of this investigation into unsteady flow features are presented in Section 6.7.

6.6.1 Key Design Areas

Although the steady state simulations fall short of accurately predicting the performance of a rim driven thruster, they do provide insights into what parts of the design should be parameterised for a design optimisation study, once the caveats and limitations are considered. As the number of simulations required increases exponentially with the number of parameters, it is important to know what should and should not be neglected. The selection and elimination of design paramaters is further discussed in Section 7.1, here we only discuss those that the steady state simulations show to be important.

Firstly, Figure 6.4 shows an uneven loading distribution that could be improved through changing the pitch distribution along the blade and the blade section design, although the blade section must be bi-directional so it is possible there is little to be gained here. As there is boundary layer growth on the duct and no tip leakage on a rim driven thruster, the design principles will differ slightly from those for ducted and unducted propellers, as the blade tips can have more loading without a tip vortex penalty to efficiency. Although the loading of the blades must not be so high as to nucleate cavitation, so there is an upper limit to the tip loading from this, despite the freedom from tip leakage penalties.

Secondly, Figure 6.6 shows that the duct or casing of the rim driven thruster is a significant contributor to the thrust at low advance ratios and drag at higher advance ratios. While there are some constraints on the duct design as it must be large enough to contain the motor stator windings, some shaping could be done to either minimise its drag or maximise its thrust producing capabilities. The duct profiling, even for bi-directional operation, is likely to be highly dependent on advance speed, with a thicker duct for maximising the thrust produced at low advance speeds and a thinner duct for minimising the profile drag when at a faster design condition.

Finally, the significant contribution of the rim, and the gap flow between it and the casing, to the torque losses in this propulsor configuration are shown in Figure 6.7. While Lea et al. (2003) experimentally investigated the effect of the size of the gap, they did not find an optimum and it is possible that this could also be improved in the current rim driven thruster design. It is certain that the increased torque compared to an un-rimmed propulsor will lead to a different blade design if efficiency, that is the ratio of thrust to torque, is to be maximised.

6.7 Unsteady Simulation Results

The results for the unsteady simulations of the 100mm rim driven thruster are investigated in this section, examining both the start up transient and the rotor-stator interaction that the unsteady simulation was intended for. A comparison of the un-

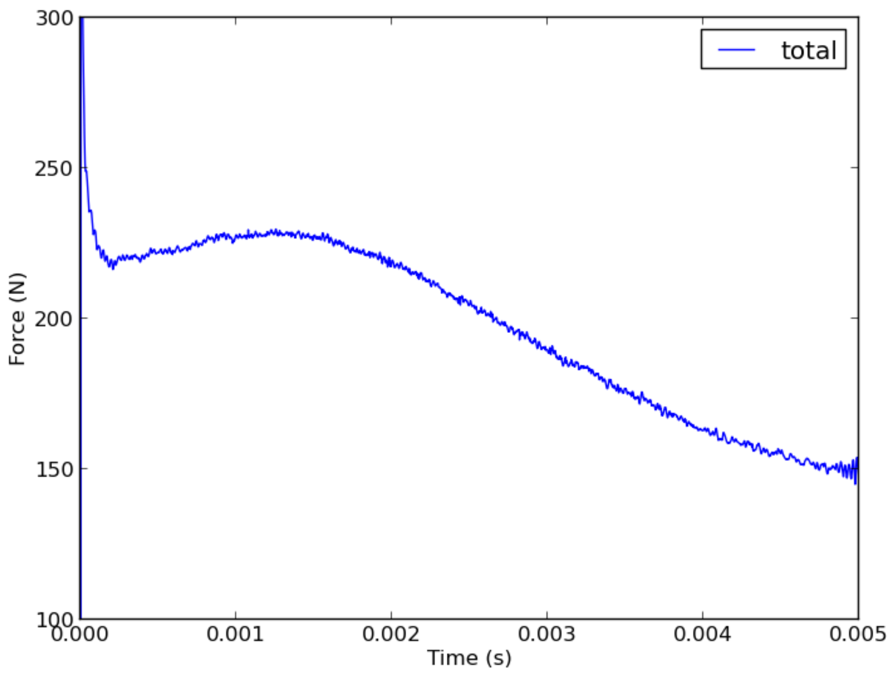


Figure 6.15: Thrust history during start up of 100mm Rim Driven Thruster at 3000 RPM.

steady simulation to a frozen-rotor pseudo-unsteady methodology is made showing that frozen-rotor methods do not capture rotor-stator interaction accurately.

6.7.1 Start Up of 100mm Rim Driven Thruster

Application of the unsteady method to the rim driven thruster is substantially more challenging than the verification case of the Wageningen B4-70 propeller and experiences a different start up flow due to the many components of the device. Figures 6.15 and 6.16 show the thrust and torque for the first quarter revolution of the simulation and is qualitatively different to the start up of the open water propeller in Figures 5.19 and 5.20. By the time it has completed one quarter revolution, the open water propeller has almost reached its steady state thrust and torque values, whereas the rim driven thruster is still far from the expected 49 N and 1.15 Nm respectively. The total forces have been broken down into pressure and viscous contributions in Figures 6.19, 6.20, 6.17 and 6.18 showing the split of forces on the rotating (blades and rim) and static (stators and casing) parts of the rim driven thruster. Figures 6.17 and 6.18 show little change over time of the viscous contribution to total thrust and torque and it is indeed the smaller component of the total forces. In contrast, Figures 6.19 and 6.20 shows that the pressure contribution to the force is still de-

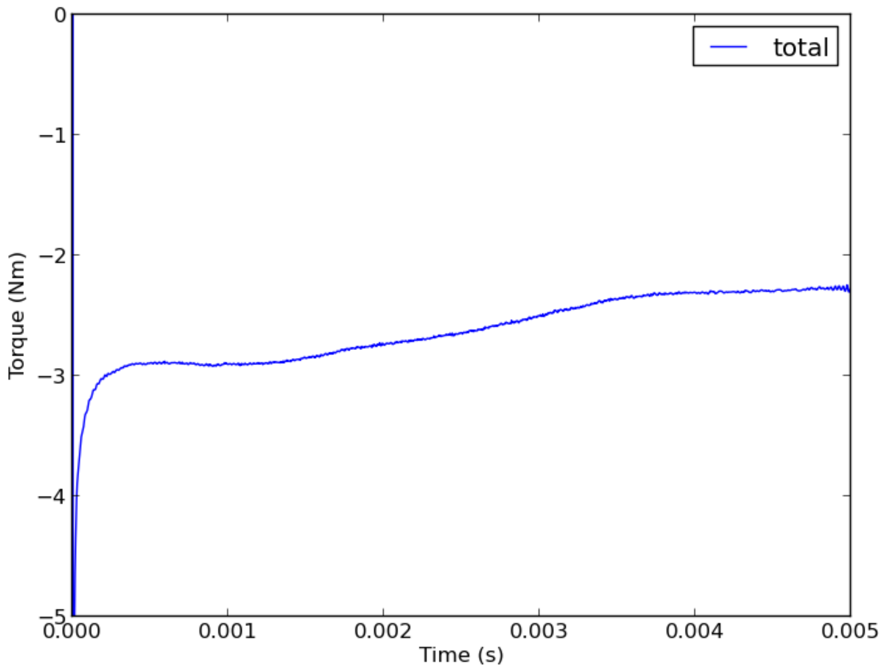


Figure 6.16: Torque history during start up of 100mm Rim Driven Thruster at 3000 RPM.

veloping, particularly on the rotating parts of the device. This is because of the complex rotor-stator interaction during the development of an induced flow.

6.7.2 Unsteady 100mm Rim Driven Thruster Results

After three complete revolutions, a periodic unsteady regime is reached, and is shown in Figures 6.21 and 6.22. These Figures show the thrust and torque variation over one complete revolution, normalised against the average thrust and torque to enable direct comparison of the variation between the frozen rotor and unsteady methods. As there are four blades, the unsteady results are split into four equal 90 degree periods and overlaid to show that there is little variation between the passing of each blade. The frozen rotor results were only measured for one quarter revolution as they are steady state and thus the periodicity does not need to be confirmed.

Comparing the unsteady results with the data from multiple frozen-rotor simulations at different rotor positions, Figure 6.21 shows the force variation and Figure 6.22 shows the torque variation. To enable direct comparison of the fluctuation without any systematic errors, the results are normalised against the average value. While there is some qualitative agreement in the trends between the two simulation methodologies in the thrust prediction in Figure 6.21, the torque prediction in Figure

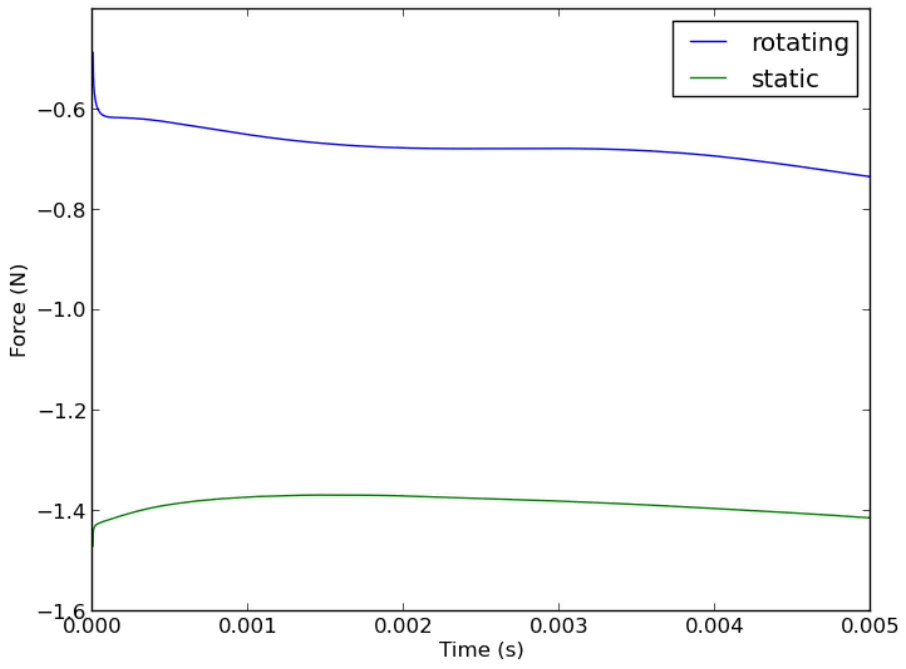


Figure 6.17: Viscous thrust history during start up of 100mm Rim Driven Thruster at 3000 RPM.

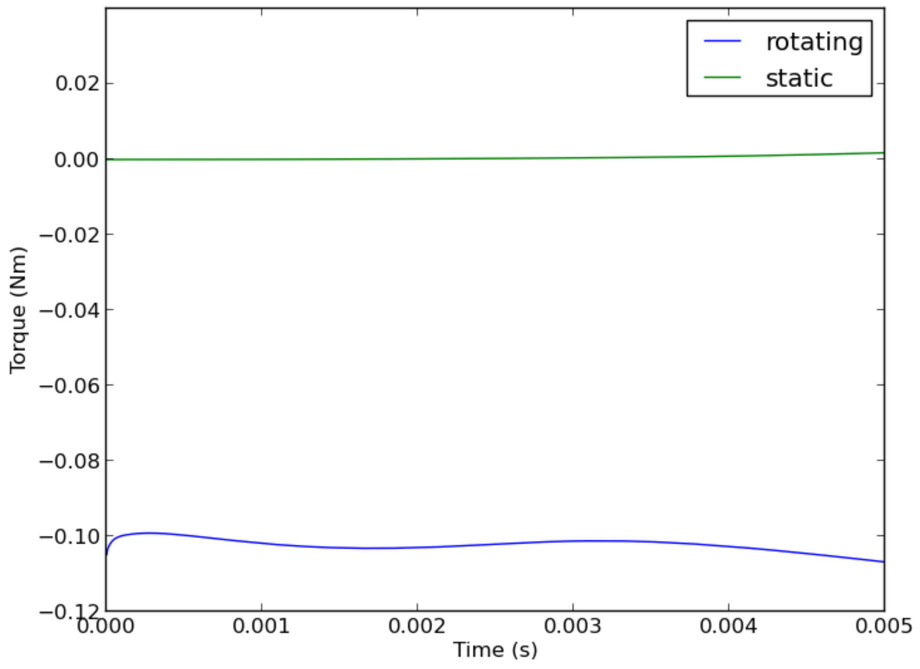


Figure 6.18: Viscous torque history during start up of 100mm Rim Driven Thruster at 3000 RPM.

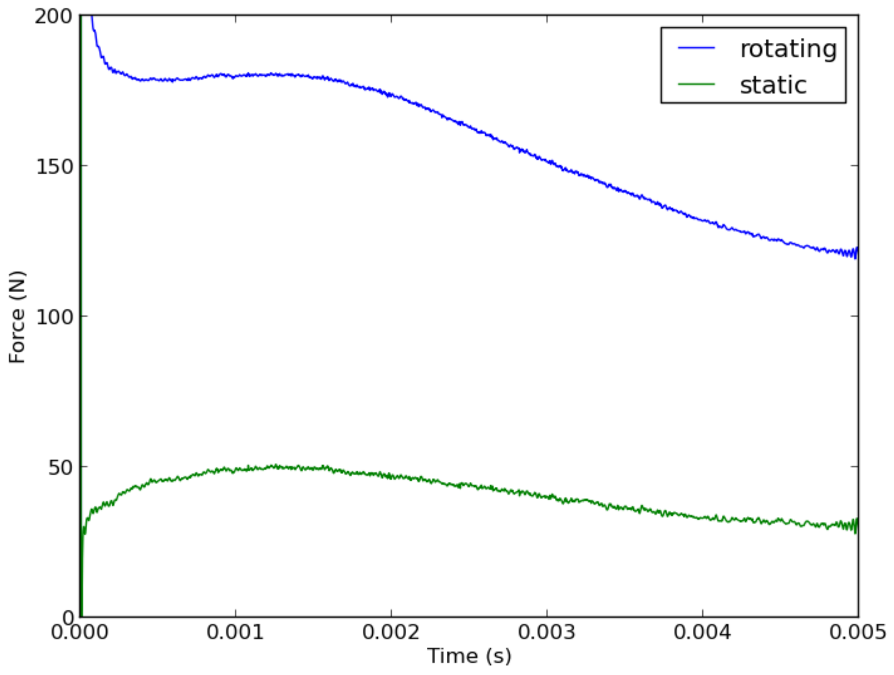


Figure 6.19: Pressure thrust history during start up of 100mm Rim Driven Thruster at 3000 RPM.

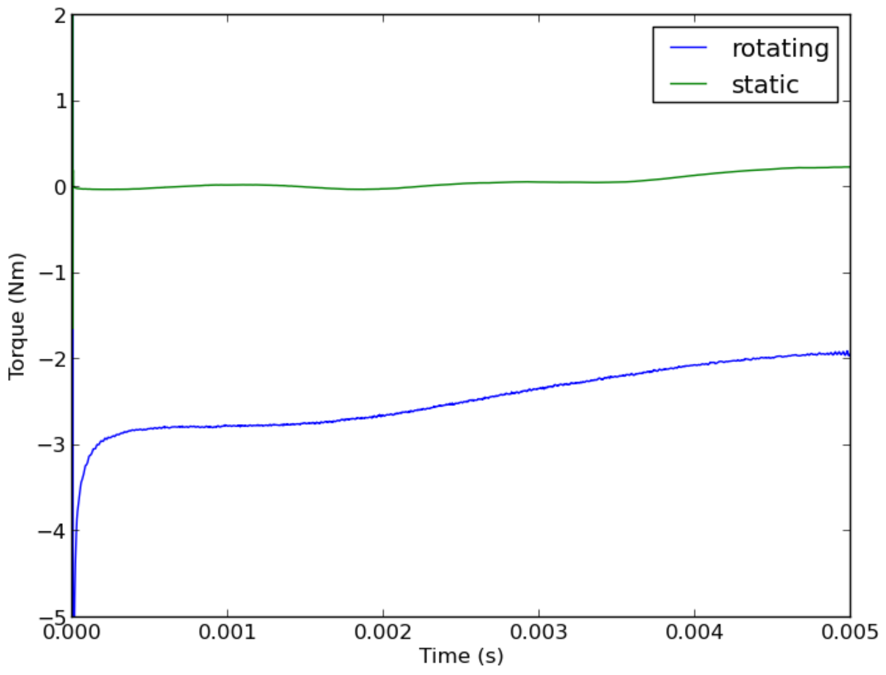


Figure 6.20: Pressure torque history during start up of 100mm Rim Driven Thruster at 3000 RPM.

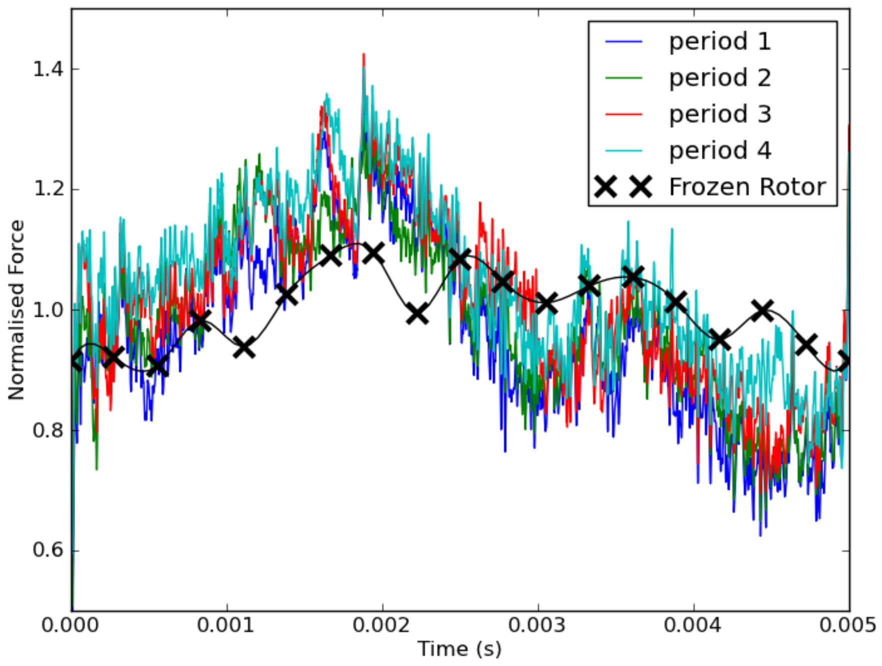


Figure 6.21: Normalised thrust over one period of 100mm Rim Driven Thruster at 3000 RPM.

6.22 differs significantly in character and does not capture the rotor-stator interaction like the unsteady simulations. It can therefore be concluded that frozen-rotor simulations cannot be used to capture the rotor-stator interaction.

The major drawback with the rotation formulation in MRFSimpleFoam is that the interfaces between static and rotating regions are frozen in place. Consequently the outflow from the static region is passed unchanged to the rotating region and vice versa. This does not pose a problem in situations where the inflow and outflow geometries are axi-symmetric as the frozen rotor method is valid here. It can also be argued that, if the flow velocity in the axial direction is significantly larger than the rotational velocity, such that an embedded particle in flow traversing the rotor would not perceive a significant movement, then the frozen rotor would be a good approximation. However, for the rim driven thruster in this case, even at the hub where the radial velocity of the rotor is low, a typical speed ratio at 1.5 metres per second advance and 3000 revolutions per minute is:

$$\frac{V_a}{\omega r} = \frac{1.5}{314.15 \times 0.01} = 0.477.$$

One option for the treatment of the interface, and one most often used in commercial software, is to treat the interface between static and rotating regions as

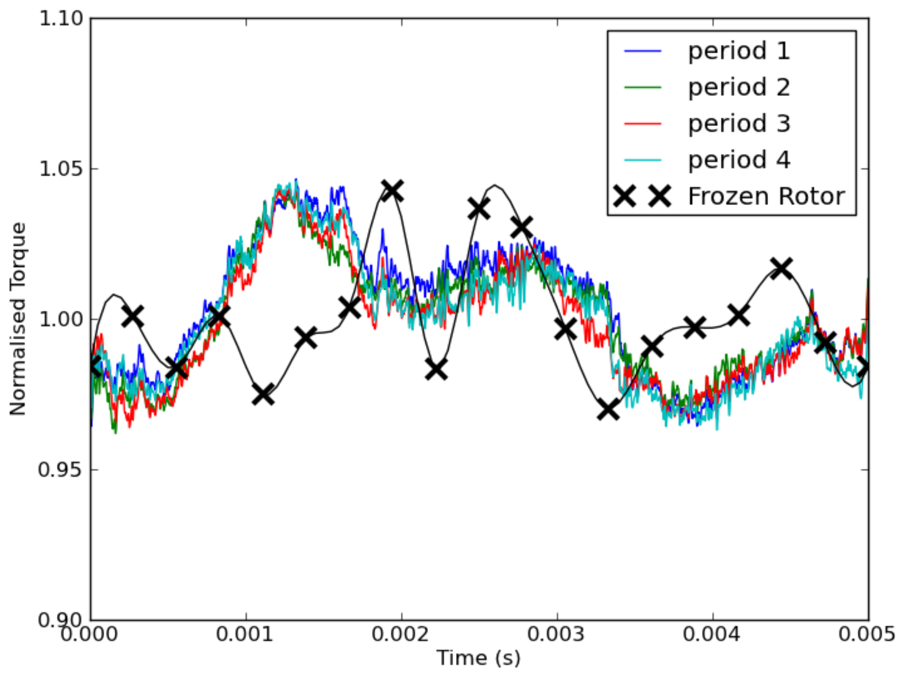


Figure 6.22: Normalised torque over one period of 100mm Rim Driven Thruster at 3000 RPM.

a mixing plane, circumferentially averaging the velocity vectors to provide an axi-symmetric inflow. Applying a similar reasoning to the above, this is a good approximation if a particle traversing the rotor with the flow in the axial direction experiences the effects of many blades of the rotor. This is the converse of the frozen rotor simulation and thus requires rotational velocity to be much larger than axial velocities. Using the above condition and inverting the ratio, $\omega r \ / \ V_a$ is approximately 2 at the hub and exceeds 10 at the blade tips. Thus it should be noted that in most cases of rotor-stator interaction a mixing plane will produce a better approximation than a frozen rotor despite not being a physically accurate depiction.

Caveats considered, there can still be some information gleaned from frozen rotor results in simulations where rotor-stator interaction is important, such as those performed by Li and Wang (2007). However, for the present simulations, the rotor-stator interaction inhibits the rapid analysis of device performance, requiring either an expensive transient simulation or omission of the phenomenon entirely. For the continuing purpose of simulating rim driven thrusters to optimise their design, it was decided that the latter should occur and the stators be excluded from the model. Consequently, the simulations in Chapter 7 do not feature the stators and are thus completely axi-symmetric, making the frozen-rotor formulation a valid assumption to make.

6.7.3 Origins of the Unsteady Force Variations

To take a closer look at the origins of the variation in thrust and torque, the streamlines past a stator and onto a blade at two different points in time, with 45 degrees phase difference, are depicted in Figures 6.23 and 6.24. It can be seen between these two points that the flow past the stators is deflected by different amounts and thus there is a significant variation in the effective angle of attack of both the stators and the blades. This is the major driver behind the fluctuation of the thrust and torque seen in Figures 6.21 and 6.22.

The cause of the experienced oscillation is the passage of the low pressure region on the blade back (the suction side) past the rear of the stator. As the low pressure region approaches the stator (from the left in Figures 6.23 and 6.24), it sucks the flow past the stators towards it, leading to an increased effective inflow angle. When the low pressure region has passed the stator, it acts as suction in the other direction, reducing the effective inflow angle here. A similar effect in reverse, caused by the higher pressure on the blade face, is experienced by the downstream stators, although it is a lesser effect in this device as the trailing edge of the bi-directional section is not as hard working as the leading edge.

This varying load has significant implications for noise and fatigue analysis, particularly for the stators which are experiencing a cyclic load with the passing of every blade. A consequence of this is that the stators must not be made too thin, although as they support the shaft and bearings of the device, this structural requirement is already met. Similar design considerations also apply to the rotor as the observed interaction is mutual.

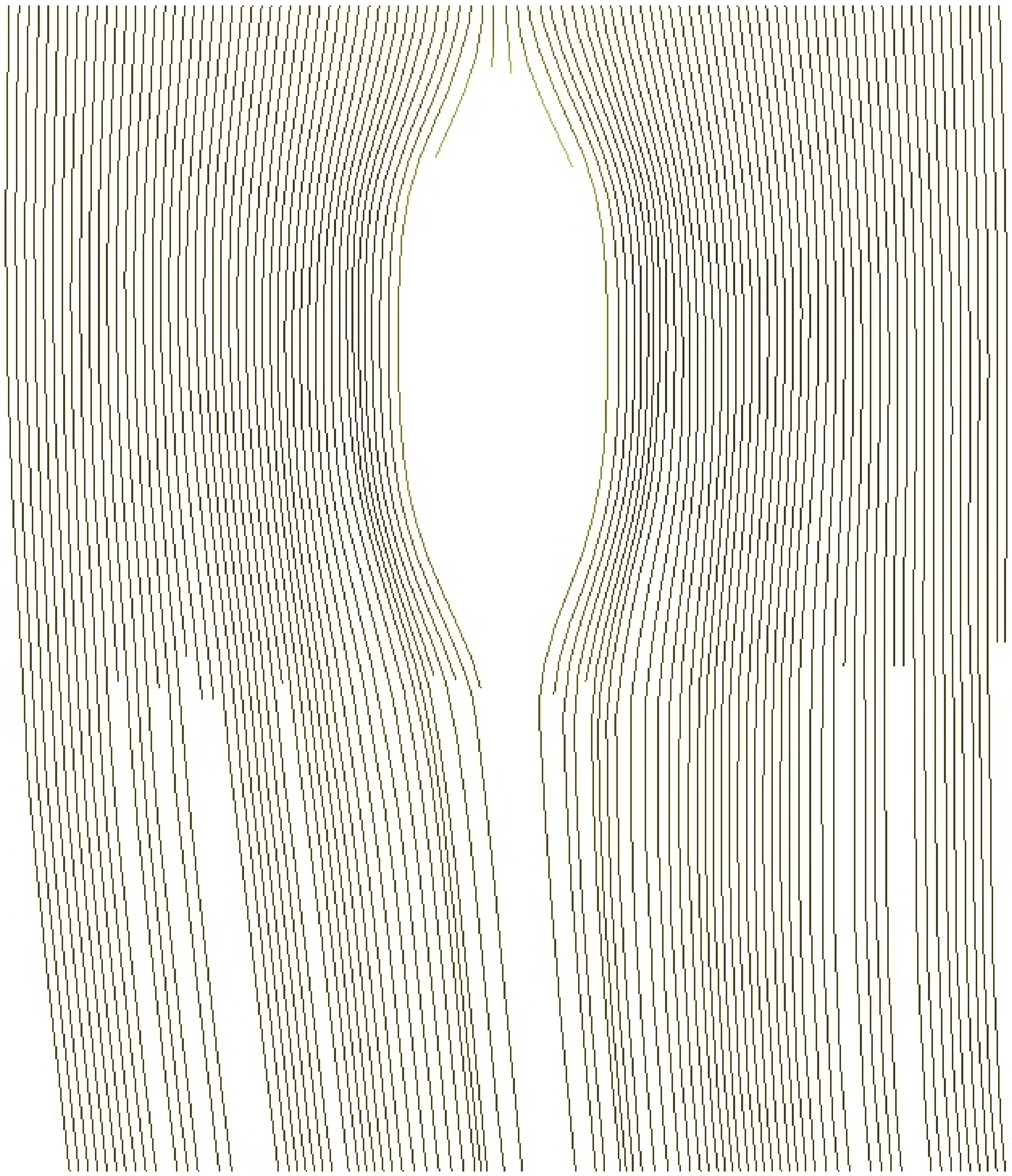


Figure 6.23: Streamlines past stator of 100mm Rim Driven Thruster at 3000 RPM.

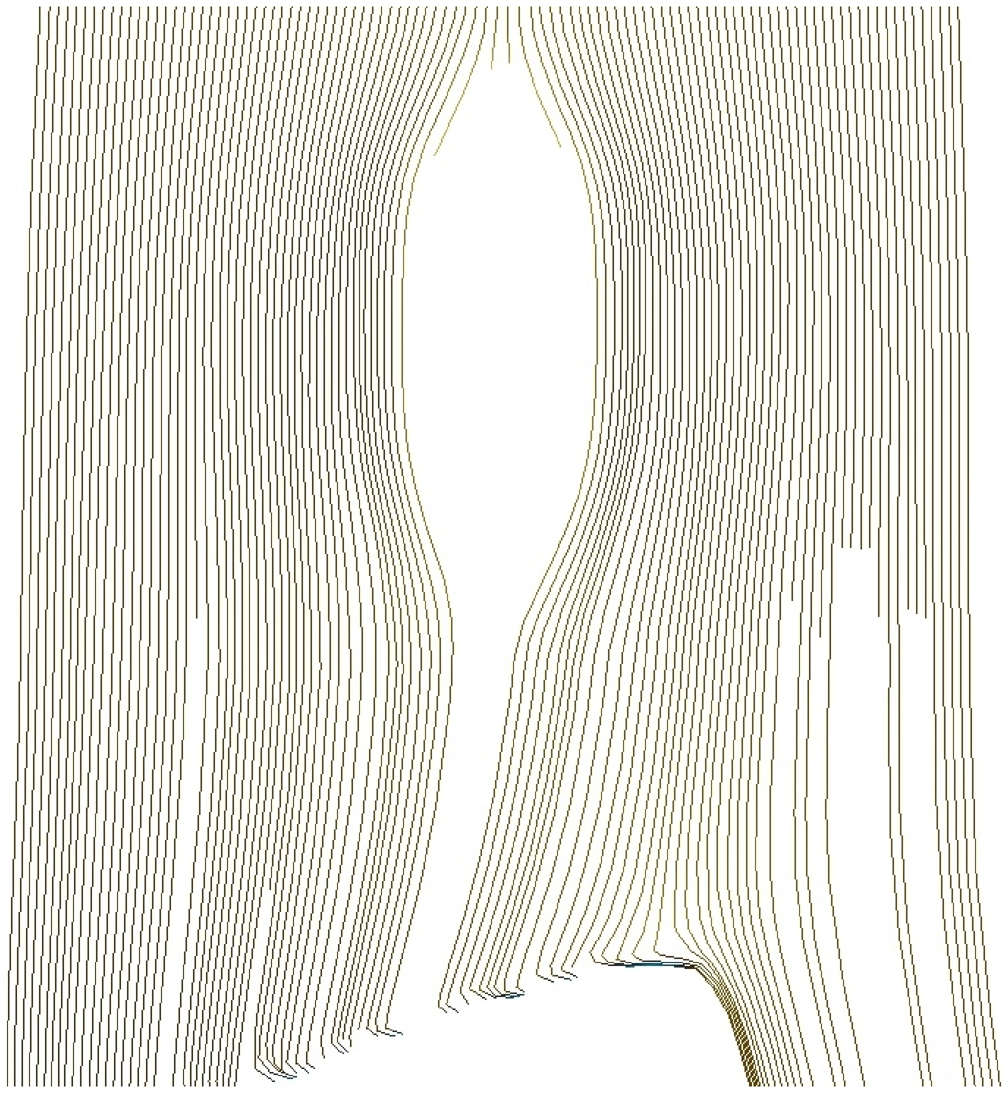


Figure 6.24: Streamlines past stator of 100mm Rim Driven Thruster at 3000 RPM, one half period later.

Chapter 7

Design Optimisation Study of the 100mm Rim Driven Thruster

This chapter details the design optimisation study carried out on the 100mm rim driven thruster, including parameterisation, surrogate modelling, infill strategy and an examination of the results. Throughout this chapter, stators are excluded from the model, and to compare like for like, a simplified model of the 100mm with very similar performance characteristics is used as the baseline device. The design condition at which the optimisation takes place is at an advance speed of 1.5 m/s and rotation rate of 3000 revolutions per minute in imitation of the peak efficiency of the baseline device.

7.1 Parameterisation and Parameter Selection

To be able to change the design geometry without introducing too many dimensions into the design space, the geometry must be parameterised and bounded. The parameterisation must be chosen such that a sufficient number of different design possibilities are explored, without wasting analysis time on designs that are not physically feasible. For the rim driven thruster there are many possible design parameters, some with negligible effect and others profoundly affecting the operational envelope of the device.

The selection of initially explored parameters for the rim driven thruster were informed from previous computational fluid dynamics analysis in Section 6. To reduce the analysis time and to remove any ambiguity caused by the rotor-stator interaction, the stators were omitted through the entirety of the design optimisation process. This left three primary regions of interest for parameterisation: the duct, the blades and the hub. The rim-gap is also a region of interest, but as the design of the rim gap also affects the electromagnetic efficiency of the motor, this region was omitted from the parameterisation. Parameterisation of the duct was performed with three variables, one governing each of duct length, width and shape. These variables were the duct radius, duct extra length and a parameter for shaping the ends of the duct, called the radial bias. Implementation of the extra length and radial bias was done by placing a co-ordinate at the requisite distance and radial position and interpolating using a Bezier spline from the duct base shape.

In the case of propeller blades, there are many possible variables to choose from. The thickness, chord, pitch and blade section all need to be specified and the distribution of these along the blade can also vary. To maintain the bi-directionality of the device, the blade section was fixed with the commercial bi-directional blade section. The remaining variables of thickness, chord and pitch were allowed to vary along the length of the blade and thus a quadratic distribution was chosen for each. To specify a quadratic distribution requires three parameters, which for this study were the value at the root, 50% radius and blade tip, giving a total of nine parameters to define the blade geometry. Finally, there may be some hydrodynamic benefit to changing the hub, so the hub diameter was also chosen as a design parameter to investigate. The complete set of initial parameters are illustrated in Figure 7.1.

Investigation of infeasible designs is prevented by selecting upper and lower bounds to the parameters. Table 7.1 shows the selected design parameters and the chosen bounds. For the duct, the lower bound of the diameter is limited by the size of the motor enclosed in the rim and while there is not a physical upper bound, a diameter greater than 60% larger then the propeller diameter was chosen as the maximum duct diameter. As for the end shaping of the duct profile, the radial bias, as a normalised parameter, has a clear bounding between zero and one referring to a radial bias towards and away from the centre respectively. The lower bound of the duct extra length is also zero, corresponding to a flat duct end profile. For the upper bound of the duct extra length, a maximum of one quarter of the propeller diameter was chosen. Thus there are 13 design parameters and these are summarised in Table 7.1 along with their upper and lower bounds.

The range of pitch distribution, given as a pitch to diameter ratio, is bounded between a ratio of 0.5 and 1.5 as this is the practical range for most propellers (Gerr, 2001). Minimum blade thickness is governed by structural requirements, thus a lower bound was chosen at 0.03 and an upper bound of 0.12 was imposed, although theoretically designs thicker than this are possible, they are unlikely to be efficient designs. The range for the chord distribution was selected by looking at the range of chord distributions used on current rim driven propulsors, which range from 0.22 to 0.33, and was expanded to 0.15 to 0.35 to include a bit more of the

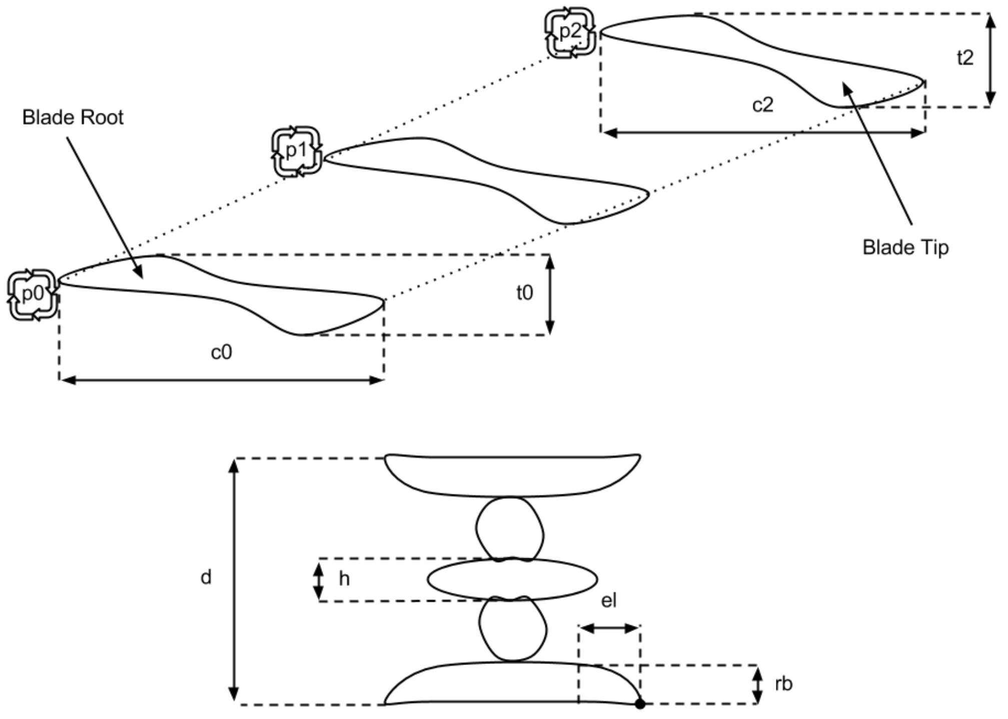


Figure 7.1: Diagram illustrating the parameterisation of a rim driven thruster.

theoretical design space.

A lower limit for the hub diameter was selected as 0.15 of the propeller diameter, as this is the smallest radius at which the blade generation program creates sections. A maximum was selected at 0.35 of the propeller diameter, giving a good range of diameters while still keeping a large blade disc area.

To reduce the number of dimensions and isolate those that have little or no impact, or those that have a linear relationship and thus need to be either minimised or maximised, the thirteen initial parameters were swept along their dimension to see their response. In each sweep the other twelve parameters were fixed at the baseline point as listed in Table 7.1. Performing a sensitivity analysis in this way has a drawback in that the interactivity of the parameters is excluded. Consequently, the effect of parameters that may otherwise seem inactive on other parts of the device is not captured in the sensitivity analysis.

The effect on performance that the variation of the chord at the root, mid-blade and tip is shown in Figures 7.2, 7.3 and 7.4 respectively. In all cases some variation in thrust is observed, often with a large increase at a c/D value of approximately 0.275, which may be attributed to the Reynolds number sensitivity of the blade section. The only reasonable conclusion that can be drawn on the chord, from Figure 7.2, is that the root chord should be minimised to reduce the torque.

Parameter	Baseline	Lower Bound	Upper Bound
Root P/D (quadratic)	1.0	0.5	1.5
Halfway P/D (quadratic)	1.0	0.5	1.5
Tip P/D (quadratic)	1.0	0.5	1.5
Root c/D (quadratic)	0.25	0.15	0.35
Halfway c/D (quadratic)	0.25	0.15	0.35
Tip c/D (quadratic)	0.25	0.15	0.35
Root t/D (quadratic)	0.045	0.03	0.12
Halfway t/D (quadratic)	0.045	0.03	0.12
Tip t/D (quadratic)	0.045	0.03	0.12
Hub Diameter [m]	0.022	0.015	0.035
Duct Outer Diameter [m]	0.149	0.14	0.16
Duct End Length [m]	0.0055	0.0	0.025
Duct Radial Bias	0.5	0.0	1.0

Table 7.1: Table of initial design parameters and their bounds.

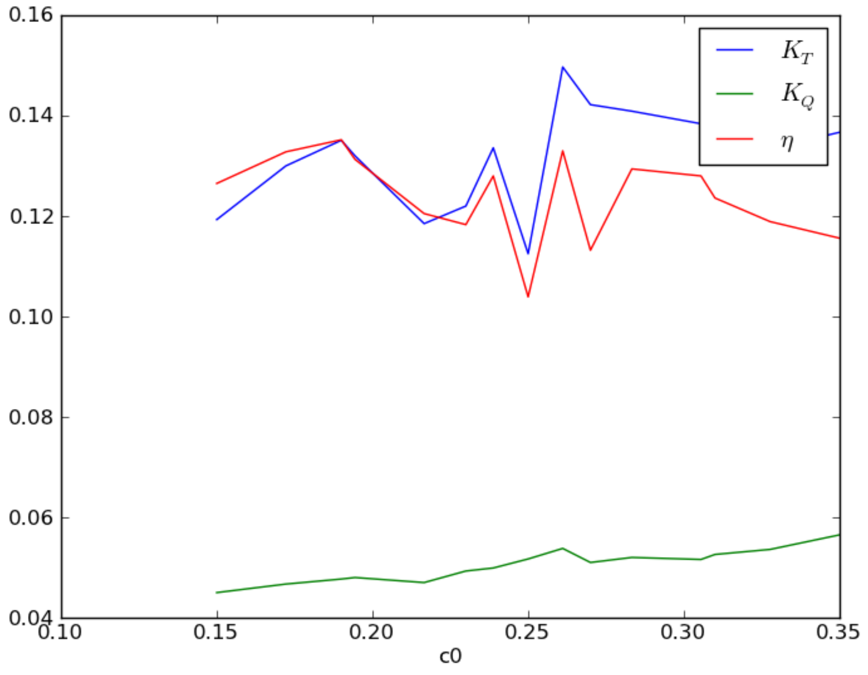


Figure 7.2: Variation of performance with root chord/diameter ratio.

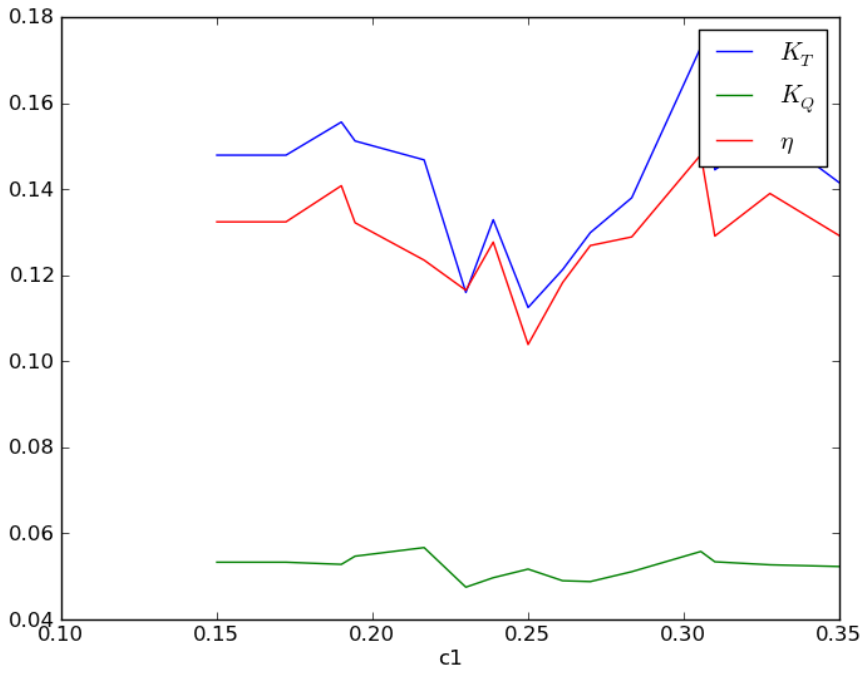


Figure 7.3: Variation of performance with halfway chord/diameter ratio.

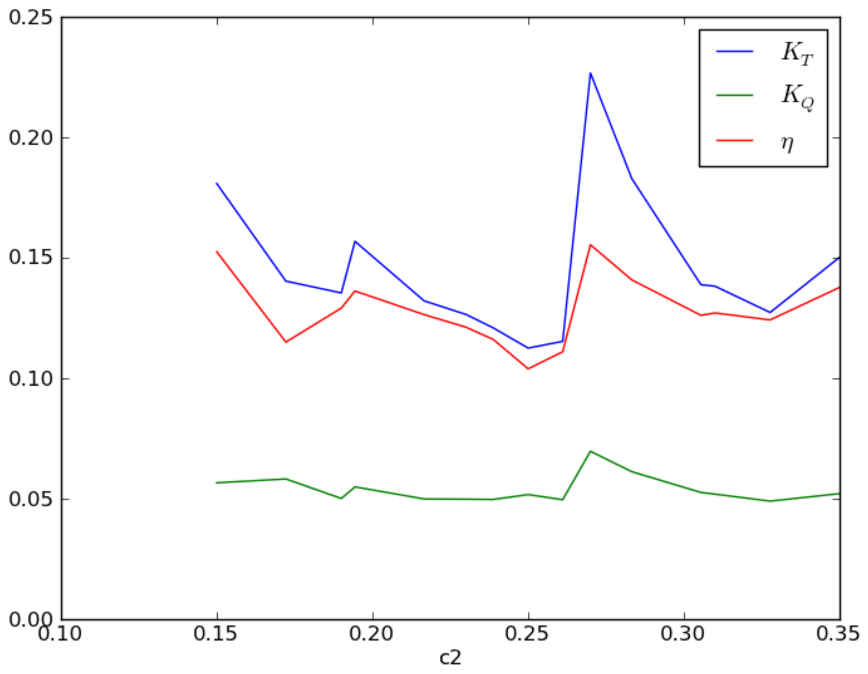


Figure 7.4: Variation of performance with tip chord/diameter ratio.

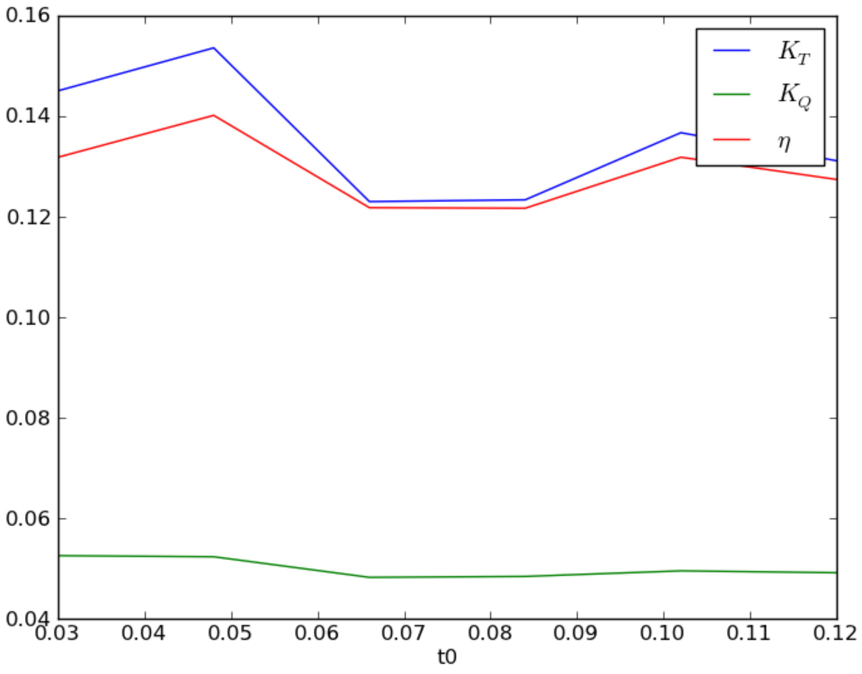


Figure 7.5: Variation of performance with root thickness/diameter ratio.

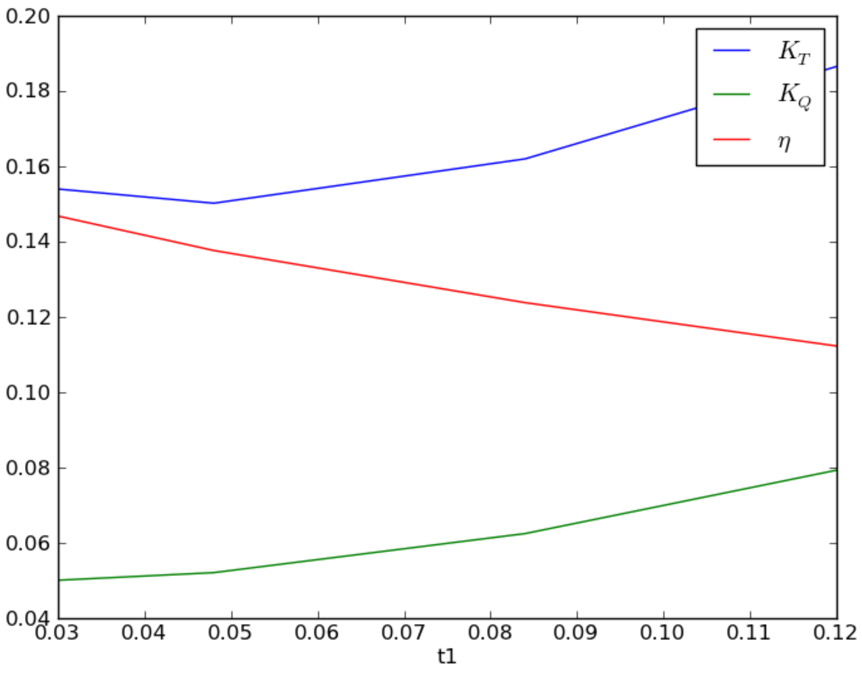


Figure 7.6: Variation of performance with halfway thickness/diameter ratio.

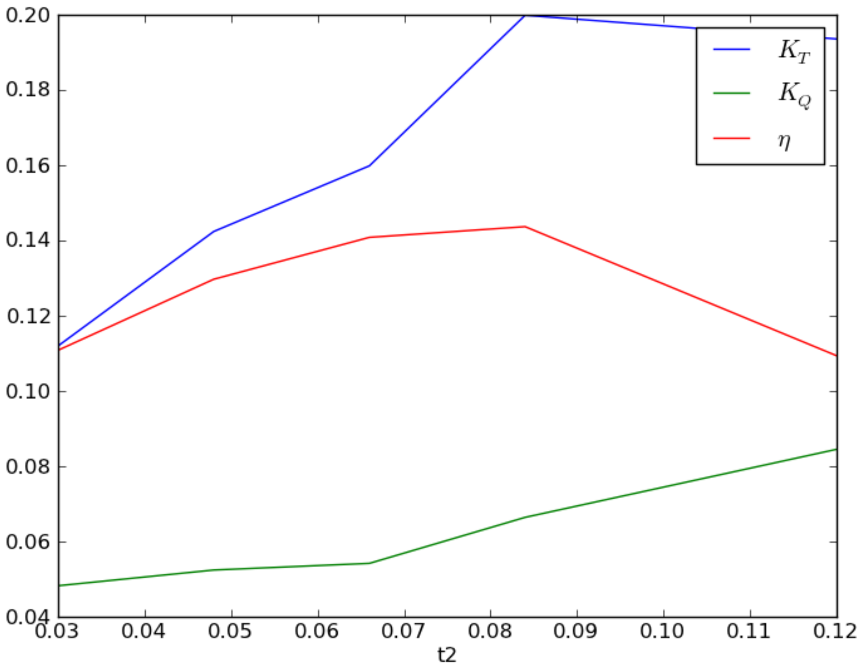


Figure 7.7: Variation of performance with tip thickness/diameter ratio.

How the performance is affected by the thickness of the blades at the root, mid-blade and tip is shown in Figures 7.5, 7.6 and 7.7 respectively. There is some variation in performance from the thickness at the root, with two apparent local optima due to changing thrust seen in Figure 7.5. Figure 7.6 shows what is intuitively expected to happen with increased thickness that is proportional to torque and to a lesser extent thrust, causing a reduction in efficiency with increasing thickness. From this, the classic engineering trade-off of wanting to make blade sections as thin as possible for hydrodynamic reasons and wanting to make them as thick as possible for structural strength would be a logical conclusion. However, most interestingly, Figure 7.7 shows an unexpected result that while torque still increases with thickness at the tip, thrust increases (initially) at a greater rate leading to a peak in efficiency at a thickness of approximately 0.08 (normalised by the diameter).

Figures 7.8, 7.9 and 7.10 show the performance in relation to the pitch ratio at the root, mid-blade and tip respectively. Both Figure 7.9 and 7.10 show an expected response to pitch ratio. As the pitch ratio (and thus the local angle of incidence) increases, both the thrust and torque increase (as ‘lift’ and drag on local sections inceases) until the local blade section stalls and does not produce any further thrust. However, the pitch ratio at the root does not seem to follow this pattern in Figure 7.8 and instead shows the torque decreasing with increasing pitch with the thrust

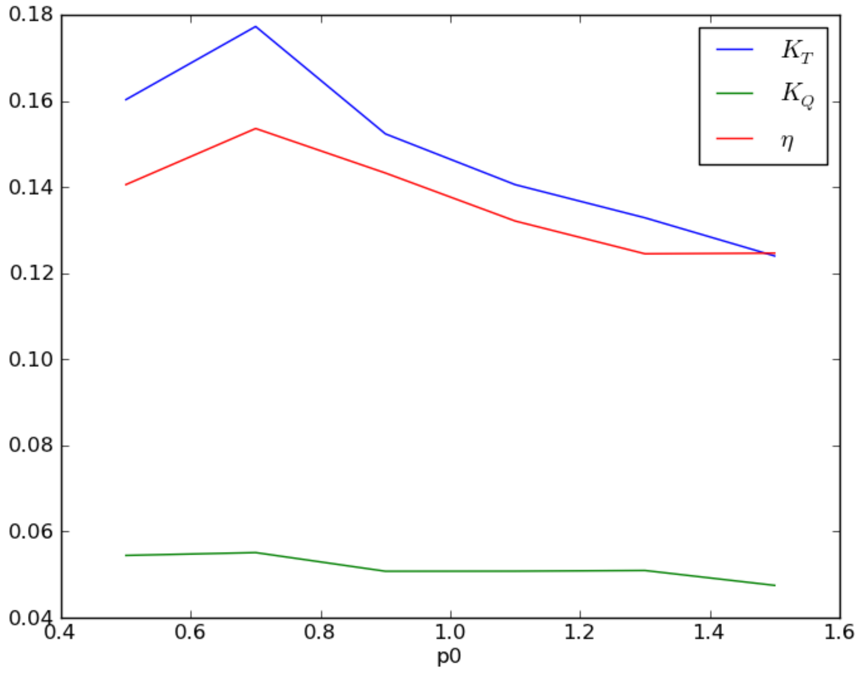


Figure 7.8: Variation of performance with root pitch ratio.

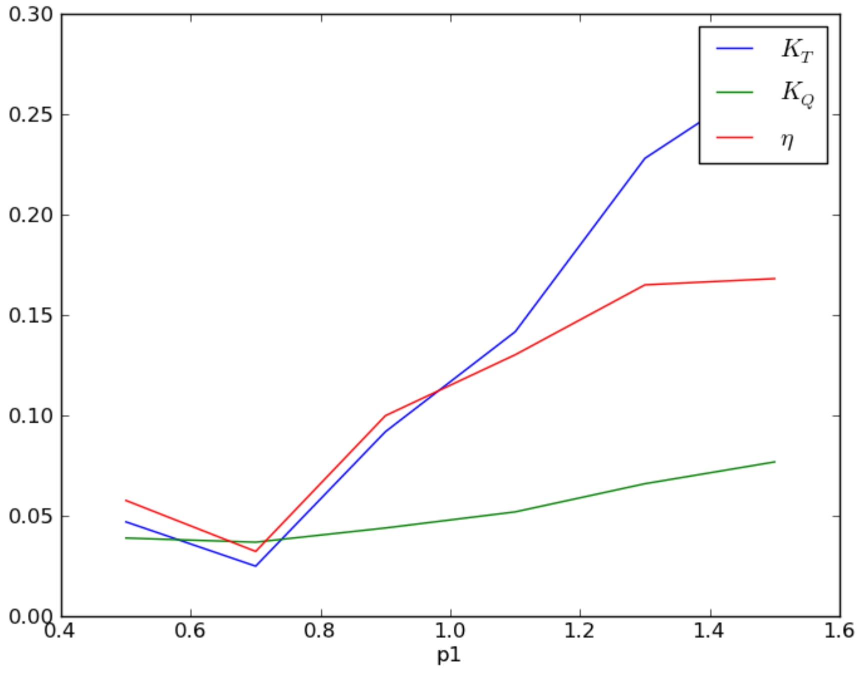


Figure 7.9: Variation of performance with halfway pitch ratio.

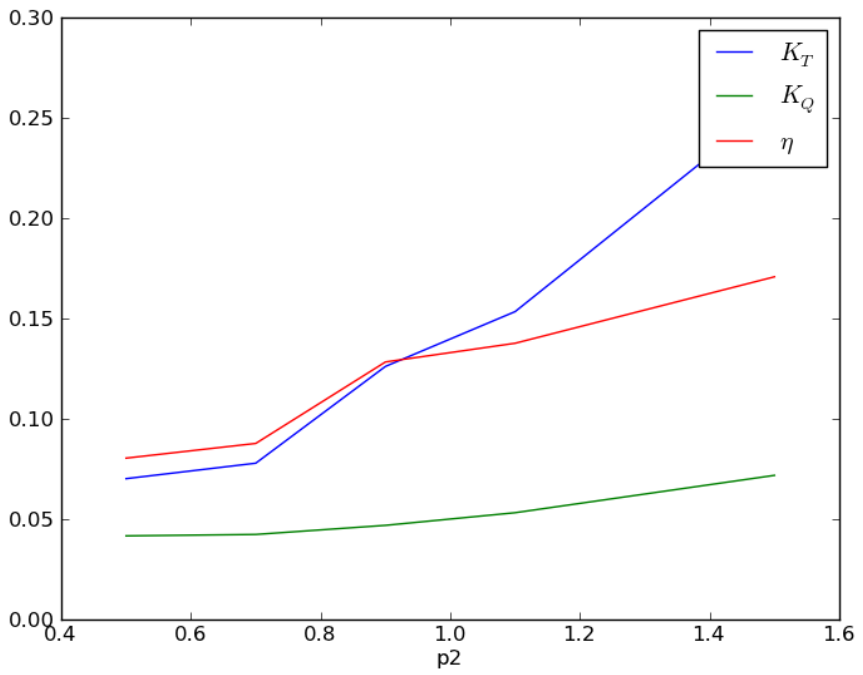


Figure 7.10: Variation of performance with tip pitch ratio.

peaking at a pitch ratio of approximately 0.7. Another unexpected anomaly is found in Figure 7.9 where the thrust reduces with increased pitch ratio at low pitch-ratios. Both these cases highlight the interesting response of performance with pitch found in rim-driven thrusters.

The results for the sensitivity analysis of the duct diameter are shown in Figure 7.11 and are as expected with efficiency decreasing with increasing diameter predominantly due to the increased drag of the larger duct frontal area, manifested in the graph as reducing thrust coefficient. An interesting point to note is the interactivity of the duct flow with the blade flow showing an increase in torque at the lowest diameter but this does not offset the increased drag of the larger duct. Consequently, for this operating condition, maximum efficiency is obtained when duct diameter is minimised but might have a non-trivial optimum at lower advance speeds or in the bollard pull condition. It is also worth noting that increasing the duct outer diameter without increasing the blade tip diameter introduces different physics into the study.

The effect of the duct profile end shaping on performance can be seen in Figures 7.12 and 7.13. Figure 7.12 shows the effect of extra length which does not seem to have a substantial conclusive impact. Similarly, the effect of the radial bias of the end profile on performance shown in Figure 7.13 is noisy and has no obvious trend.

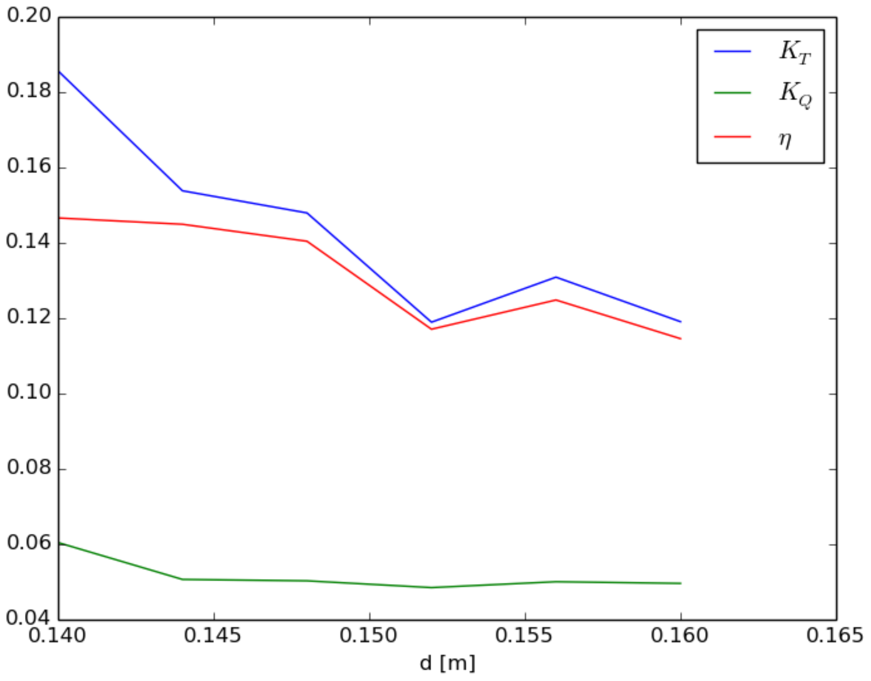


Figure 7.11: Variation of performance with duct outer diameter in metres.

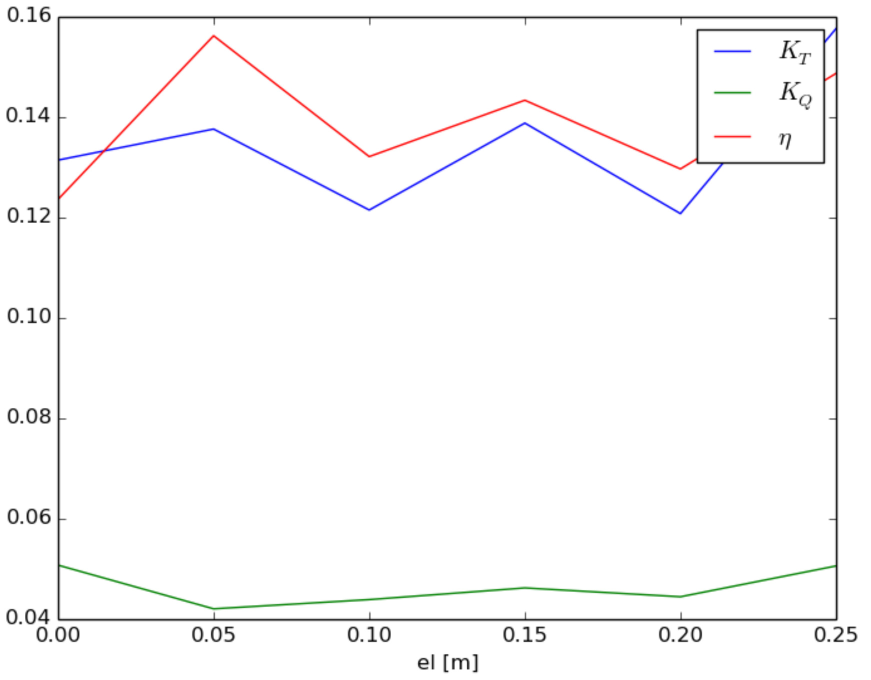


Figure 7.12: Variation of performance with duct extended length in metres.

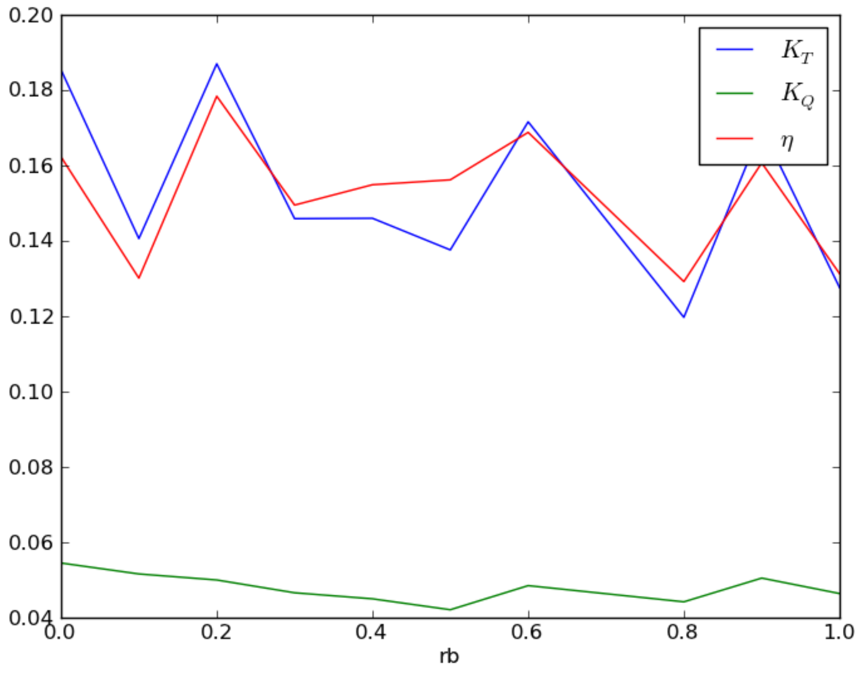


Figure 7.13: Variation of performance with duct radial bias.

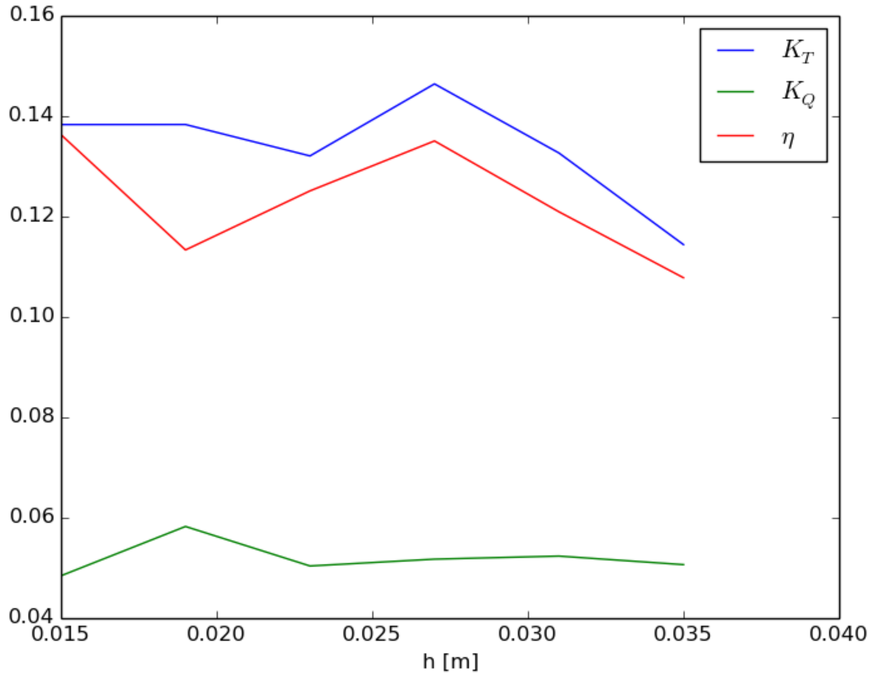


Figure 7.14: Variation of performance with hub diameter in metres.

However, these parameters will almost certainly have some interactivity between them and, as previously discussed for the duct diameter, might be more significant at lower advance ratios or at the bollard pull condition.

Finally, the hub diameter has two possible optimum performance points shown in Figure 7.14, one at the lowest diameter and one at a diameter of approximately 0.0275. Additionally it is expected that the size of the hub will impact the inflow to the blade root and consequently exhibit an interaction with the root pitch ratio parameter, $p0$.

Consequently, the initial thirteen parameters can be reduced to only those that are likely to yield a significant improvement in efficiency and these are the pitch ratio, thickness and hub diameter. To further reduce the dimensionality of the design space, it was decided that the thickness distribution should be represented by a linear function rather than a quadratic one, varying in the blade spanwise (*id est* radial) direction. From the sensitivity analysis of the initial parameters, it is reasonable to expect that the design optimisation will improve the efficiency by at least 0.02 over the baseline case, which corresponds to a percentage improvement of approximately 14%.

7.2 Surrogate Modelling

As the evaluation of the objective function takes too much time for the direct optimisation of the geometry, a surrogate model can be used to represent the design landscape and thus speed up the optimisation process. A generalised flow diagram for the optimisation procedure using surrogate modelling is depicted in Figure 7.15. A Kriging surrogate model based on Forrester et al. (2008) was primarily used and the code is listed in Appendix D. In converting the code from MATLAB®to the Python language, the code was also refactored into an object oriented paradigm so that multiple surrogate models could exist under different namespaces, rather than storing the data of all models in the global namespace.

7.2.1 Kriging

Kriging is a surrogate modelling technique named after Daniel Krige, its inventor, whom used it as a statistical approach for the valuation of mines in South Africa (Krige, 1951). Later work by Sacks et al. (1989) applied the method to engineering design through the approximation of computational results. The core of the Kriging method is a basis function of the form:

$$\psi^{(i)} = \exp - \left(\sum_{j=1}^{k} \theta_j |x_j^{(i)} - x_j|^{p_j} \right) \tag{7.1}$$

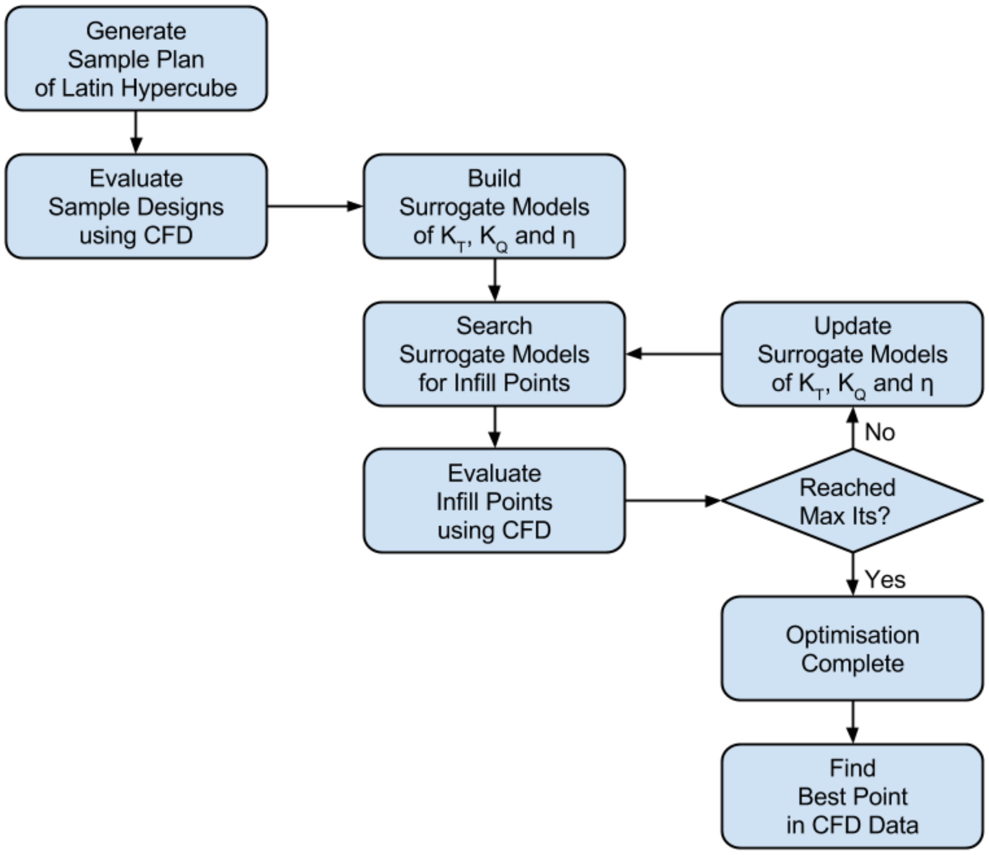

Generate Sample Plan of Latin Hypercube
Evaluate Sample Designs using CFD
Build Surrogate Models of K_T, K_Q and η
Search Surrogate Models for Infill Points
Update Surrogate Models of K_T, K_Q and η
Evaluate Infill Points using CFD
Reached Max Its?
No
Yes
Optimisation Complete
Find Best Point in CFD Data

Figure 7.15: Generalised flow diagram for optimisation procedure.

where $\psi^{(i)}$ is the value of the basis function, k is the number of dimensions, and θ_j and p_j are tuning parameters in the jth dimensions. It is worth noting that if $p_j = 2$ for all j and θ_j is constant for all j then the Kriging basis function is equivalent to the Gaussian basis function:

$$\psi(r) = \exp\left(\frac{-r^2}{2\sigma^2}\right) \tag{7.2}$$

where r is the radial distance and σ is the tuning parameter in this case.

As a surrogate modelling technique, Kriging has a number of advantages. Over simple linear and polynomial radial basis functions, it has better gradient capture where the objective function does not have a polynomial behaviour. It also performs better over a Gaussian radial basis function as it is directionally tunable, something that is particularly important where the rate of change in one dimension is unlikely to be similar to the other dimensions. The better estimation of Kriging does come at a cost computationally in both memory and time, and the tuning of parameters requires an internal optimisation with the Kriging itself that may not be able to find a suitable parameter set depending on the bounds given.

Another key, albeit not exclusive, advantage of Kriging is the ability to apply statistical analysis to estimate the mean squared error (Sacks et al., 1989) and subsequently explore the design space based on the likelihood of finding an improved design. A simple way of doing this is to minimise the statistical lower bound:

$$LB(\vec{x}) = \hat{y}(\vec{x}) - A\hat{s}(\vec{x}) \tag{7.3}$$

where LB is the lower bound, $\hat{y}$ is the estimate of the function, $\hat{s}$ is the mean squared error and A is a constant that controls the degree of exploration. As A in Equation 7.3 tends to 0 there is pure exploitation, and as $A \to \infty$ there is pure exploration.

An improvement on the statistical lower bound is the expected improvement where the expected value of the probability distribution of the error at some point $\vec{x}$ is compared with the value of the current minimum in the data. Formulaically, the expected improvement can be expressed as:

$$E[I(\vec{x})] = (y_{min} - \hat{y}(\vec{x}))\left[\frac{1}{2} + \frac{1}{2}\text{erf}\left(\frac{y_{min} - \hat{y}(\vec{x})}{\hat{s}\sqrt{2}}\right)\right] + \hat{s}\frac{1}{\sqrt{2\pi}}\exp\left[\frac{-(y_{min} - \hat{y}(\vec{x}))^2}{2\hat{s}^2}\right] \tag{7.4}$$

where $E[I(\vec{x})]$ is the expected improvement, y_{min} is the current minimum value and erf() is the error function, the implementation of which is documented within Appendix D. As $E[I(\vec{x})] = 0$ when $\hat{s} = 0$, it can be shown that an infill procedure based on maximising expected improvement will eventually find the global optimum.

7.2.2 Optimisation Strategy

There are a number of strategies that can be used for a design optimisation study and the selection of strategy is a balance of exploitation, that is improving designs within a local design space, and exploration, searching the global design space for better designs. For a given number of design evaluations, there must also be some weighted allocation between the initial sample plan and infill points whose location is based on whether exploitation or exploration is required.

The plan for the initial sampling of the design space was generated as a random six-dimensional latin hypercube. To find a latin hypercube that covered as much of the design space as possible, 500 were randomly generated and compared using the Morris-Mitchell criterion (Morris and Mitchell, 1995). The number of points in the initial sample plan was selected by using the rule of thumb of using ten times the number of dimensions.

For the infill points a hybrid strategy was chosen, such that the surrogate could be used to understand the design space as well as exploit it. The maximum number of evaluations was selected as 100 with 60 of these allocated to the initial sample plan, 61 including the baseline case in the initial sample plan also, this left 39 to be used for infill points. As there are three parameters of interest, the thrust, torque and efficiency, it was decided that the surrogate should have infill points based on each of these. It is worth noting that while the thrust and torque surrogates are used to decide infill points and gain insight into the effects of design parameters, they are not treated as optimisation objectives, thus goal of the optimisation is still solely to maximise the efficiency, η.

For the thrust, the maximum value is of interest, therefore three infill points were chosen based on the maximum thrust, the maximum statistical upper bound of thrust and the maximum expected improvement of thrust. Similarly for the efficiency, three infill points were selected for the maximum value, statistical upper bound and expected improvement. However, for the torque, the final three infill points were chosen based on the minimum value of torque, minimum statistical lower bound and maximum expected improvement, the expected improvement always being maximised for the correct inversion of the response surface. Four successive rounds of nine infill points were allocated to this hybrid exploration and exploitation strategy, sufficient to start to draw some conclusions about the design space and the three remaining infill points were used for pure maximisation of the efficiency. Ideally, maximisation of the efficiency should continue until the expected improvement of the surrogate model tends to zero, but with a limited number of design evaluations, the best design after reaching the maximum number of evaluations must be taken.

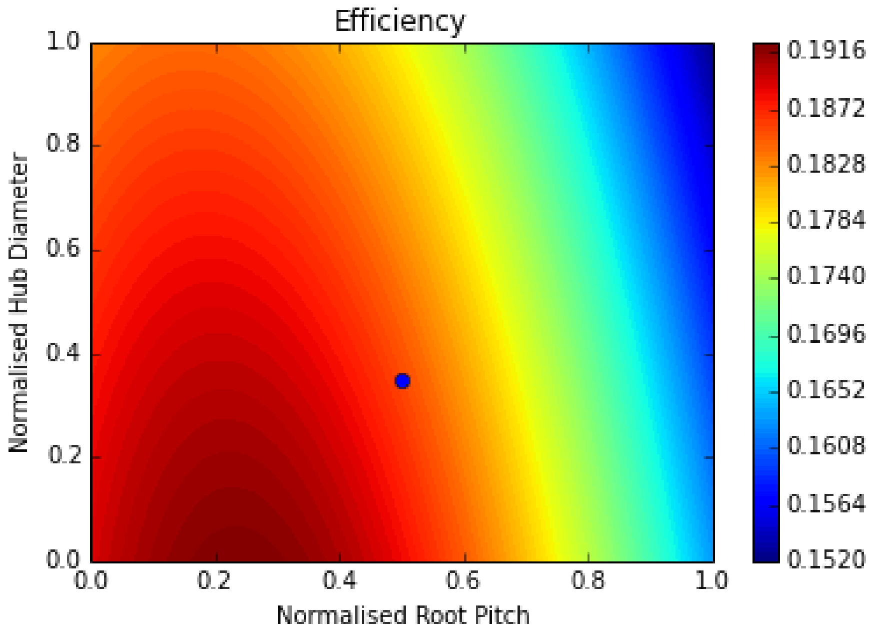


Figure 7.16: Visualisation of efficiency against normalised root pitch and hub diameter, pivoted about the base design.

7.2.3 Surrogate Model Visualisation

Some understanding of the design landscape can be gleaned through its visualisation or, in this case, the visualisation of the surrogate model. To visualise the design landscape, two parameters are plotted on the x and y axes, with a third dimension for the objective function added through the use of colour. This enables us to explore only the two plotted parameters and how they vary about a fixed point, or pivot, in the design space. However this does allow insight into the interactivity between design parameters, and multiple pivots, that is the points at which the unplotted parameters are fixed, can be investigated to further increase understanding.

Examining Figure 7.16, which shows how efficiency changes when root pitch ratio and hub diameter are varied within their bounds, while all other design parameters are kept at their baseline value, it is clear that some gain in efficiency is possible through a slight reduction in both root pitch and hub diameter. With the rest of the parameters fixed at the baseline, it is apparent that hub diameter should be minimised, while the root pitch has some optimum value near a normalised value of 0.2. This also holds true when the remaining parameters are fixed at the optimum, as shown in Figure 7.25.

Analysing the variation of efficiency with root and tip pitch, with all other parameters fixed at their baseline values, as shown in Figure 7.17, shows that there is a distinct optimum value of normalised root and tip pitch ratios at 0.2 and 0.6,

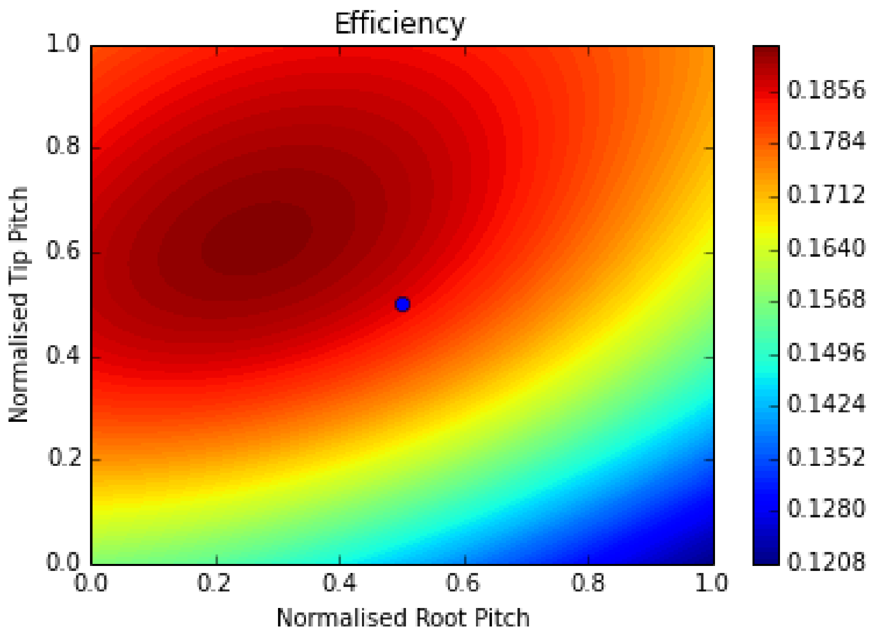


Figure 7.17: Visualisation of efficiency against normalised root pitch and tip pitch, pivoted about the base design.

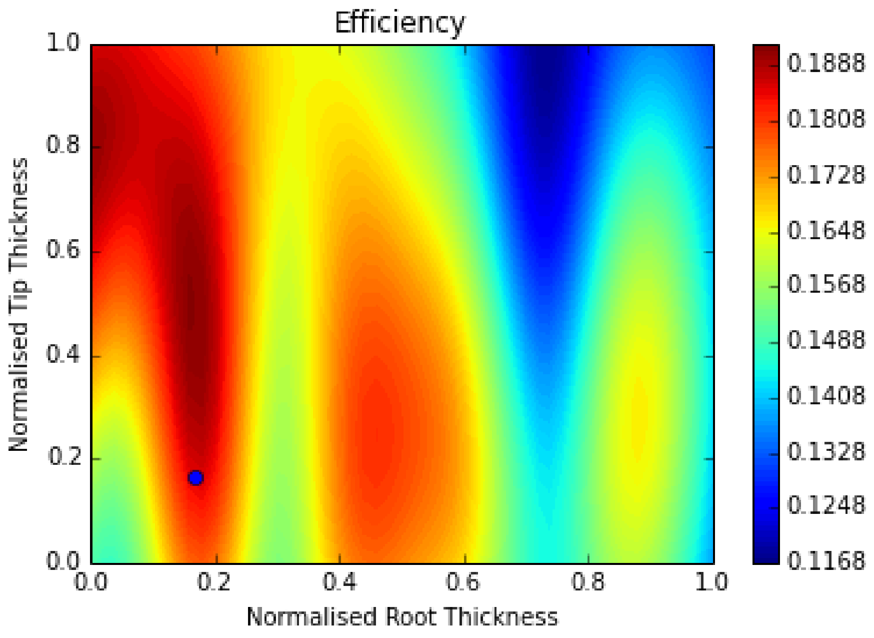


Figure 7.18: Visualisation of efficiency against normalised root thickness and tip thickness, pivoted about the base design.

respectively. There is also a more general trend of increasing efficiency with increasing tip pitch ratio and decreasing efficiency with increasing root pitch ratio. Interestingly, this observation is not immediately manifest when examining how the torque and thrust coefficients vary across the same parameters (Figures 7.20 and 7.23, respectively) which primarily show a variation with tip pitch ratio only.

Perhaps the most dynamic design landscape is the one where root and tip thickness are varied, again with the other design parameters fixed at their baseline values, as shown in Figure 7.18. While there may be an adage in hydrodynamics that thinner is always better, thus placing fluid dynamics in eternal conflict with structural mechanics where thicker is always stronger, in this case of a bi-directional rim-driven thruster the best (hydrodynamic) design is not at the minimum thickness. The thickness at the root in particular shows substantial peaks and troughs in efficiency as it is changed and the tip thickness has a maximum efficiency at a value that depends on root thickness.

The source of this peculiar efficiency landscape is not from the changes in thrust output, which is shown in Figure 7.21 and despite having a central peak for root thickness, predominantly shows a strong increase in thrust with tip thickness. However, further insight is gained from considering the torque, as shown in Figure 7.24, which shows that the increased thrust with increasing tip thickness is coupled with an increase in torque. Furthermore, the torque can vary substantially with small changes in tip and root thickness. This suggests that the efficiency changes are based largely upon the drag on the blades, and thus arises from the changing flow field around the blade sections as the effective thickness to chord ratio is varied.

Figure 7.19 shows the variation of thrust coefficient when changing hub diameter and root pitch ratio, with all other parameters fixed at their baseline values. There is a clear peak at a normalised pitch ratio of approximately 0.6 and maximum hub diameter, but the variable that has the most impact on thrust here is the pitch ratio. Subsequently, Figure 7.20 shows that the tip pitch ratio has much more effect on the thrust than the root pitch ratio, with peak thrust being produced at minimum root and maximum tip pitch ratios.

It is very interesting to compare Figures 7.19 and 7.20 with Figures 7.28 and 7.29 respectively. These show that changing the pivot point, that is the values to which the remaining parameters are fixed, changes the way the performance responds to changes in the design. Conversely to Figure 7.19, Figure 7.28 shows a more significant variation in thrust with hub diameter, with the addition of the peak thrust being produced at a minimum hub diameter rather than at the maximum. It should however be noted, that the magnitude of variation is much smaller, with a difference of 0.0324 across the range of both parameters, compared with 0.072 in Figure 7.19.

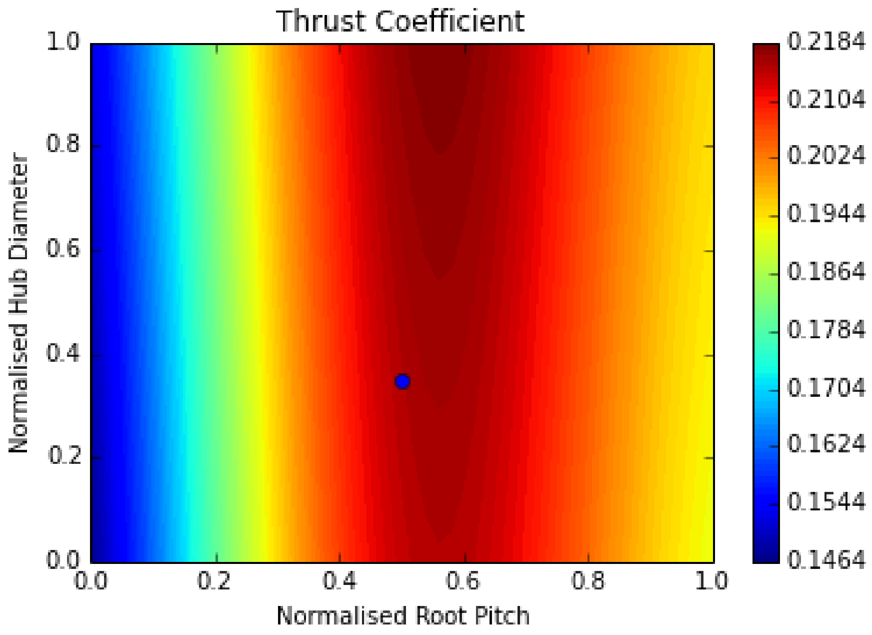


Figure 7.19: Visualisation of thrust coefficient against normalised root pitch and hub diameter, pivoted about the base design.

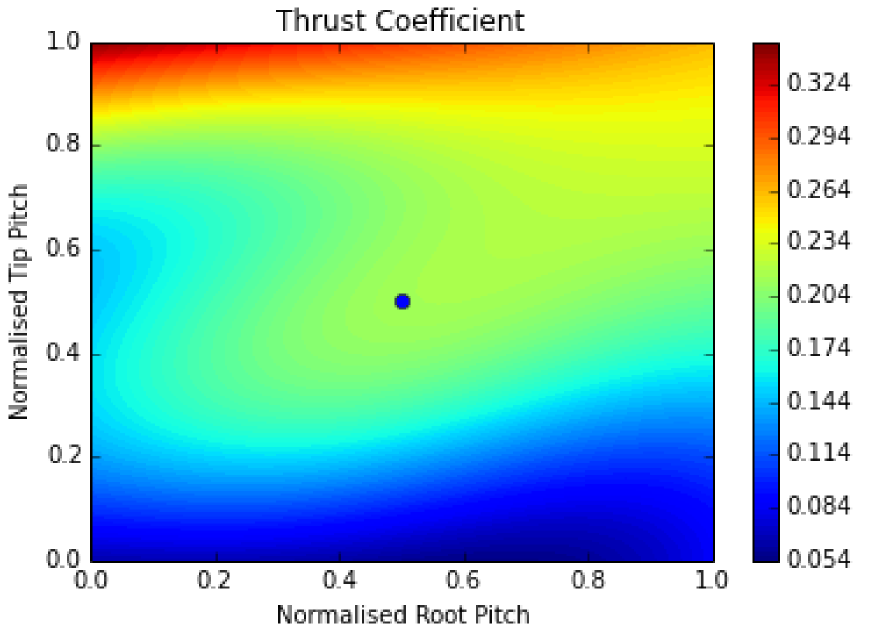


Figure 7.20: Visualisation of thrust coefficient against normalised root pitch and tip pitch, pivoted about the base design.

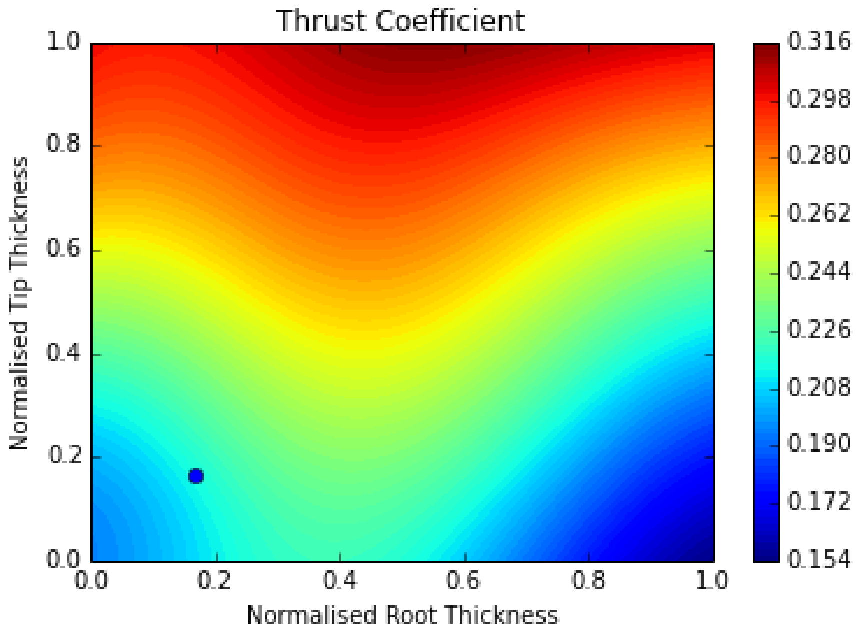


Figure 7.21: Visualisation of thrust coefficient against normalised root thickness and tip thickness, pivoted about the base design.

Similarly Figure 7.29 shows an increased significance of root pitch ratio, when compared to Figure 7.20, with a peak thrust now at minimum root pitch ratio, although still at maximum tip pitch ratio. Finally comparing the thrust landscapes between Figures 7.21 and 7.30, we can see it changing from almost a linear landscape, where thrust is proportional to tip thickness, to a complex landscape with peak thrust produced in the corner of maximum tip thickness and minimum root thickness.

Analysing Figure 7.22, which shows the variation of torque coefficient against normalised root pitch and hub diameter with all other parameters kept at their baseline values, very little variation of torque can be seen and the prevailing effect is due to root pitch that peaks at a normalised value of 0.3. Contrasting this with Figure 7.31, which shows the same but fixes all remaining parameters at the optimum point, there is much more change of torque with root pitch in the latter, with the peak torque shifting to occur at a normalised root pitch of 0.7. Both Figures show that hub diameter does not have much effect on the torque.

Due to the superior moment arm, it is not surprising to see a greater effect on torque by changing the tip pitch than by changing the root pitch as shown in Figure 7.23. The proportional relationship between tip pitch and torque is also as one might expect. However, when comparing this with the same slice of parameters but fixing the remaining parameters at their optimum values, as per Figure 7.32, there is a much greater impact on torque by the root pitch, which concurs with the

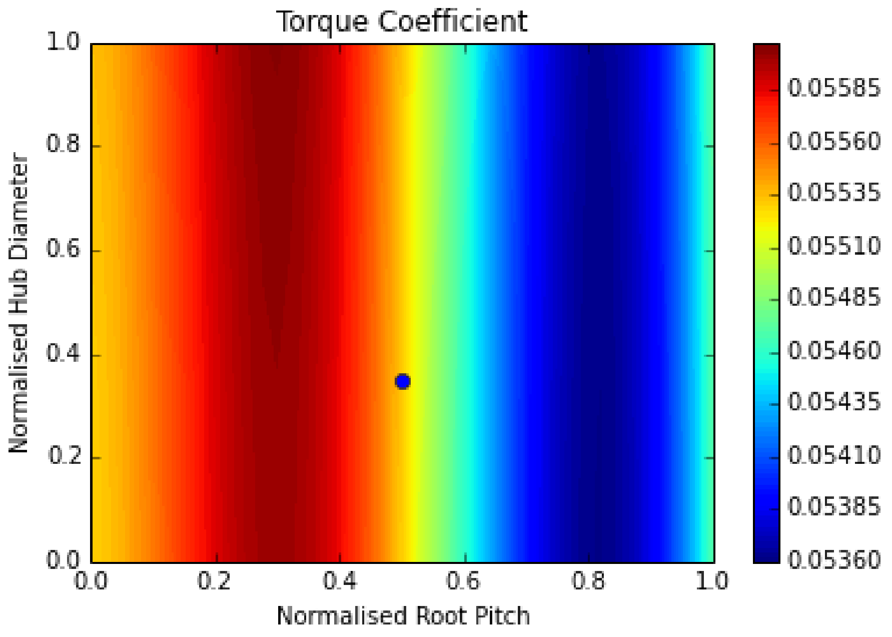


Figure 7.22: Visualisation of torque coefficient against normalised root pitch and hub diameter, pivoted about the base design.

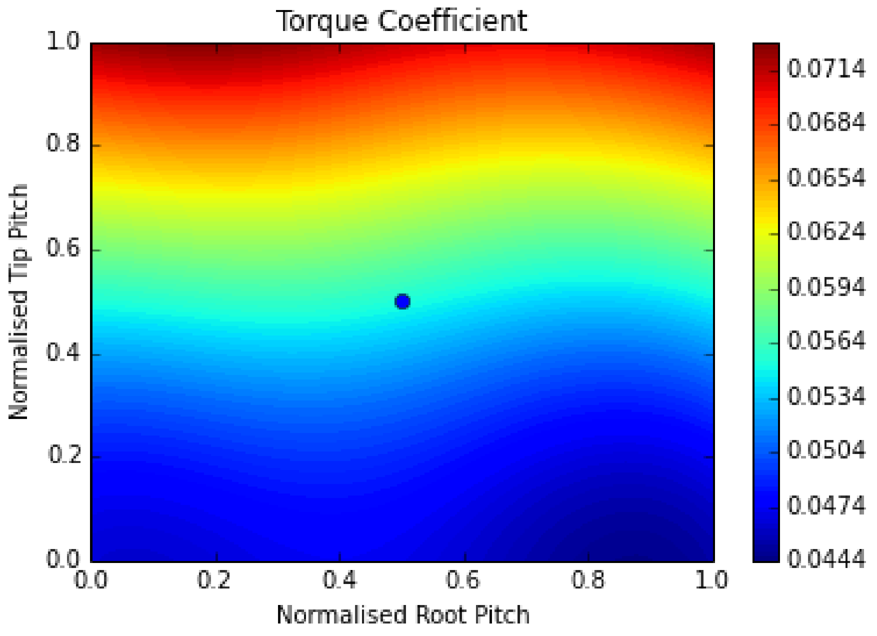


Figure 7.23: Visualisation of torque coefficient against normalised root pitch and tip pitch, pivoted about the base design.

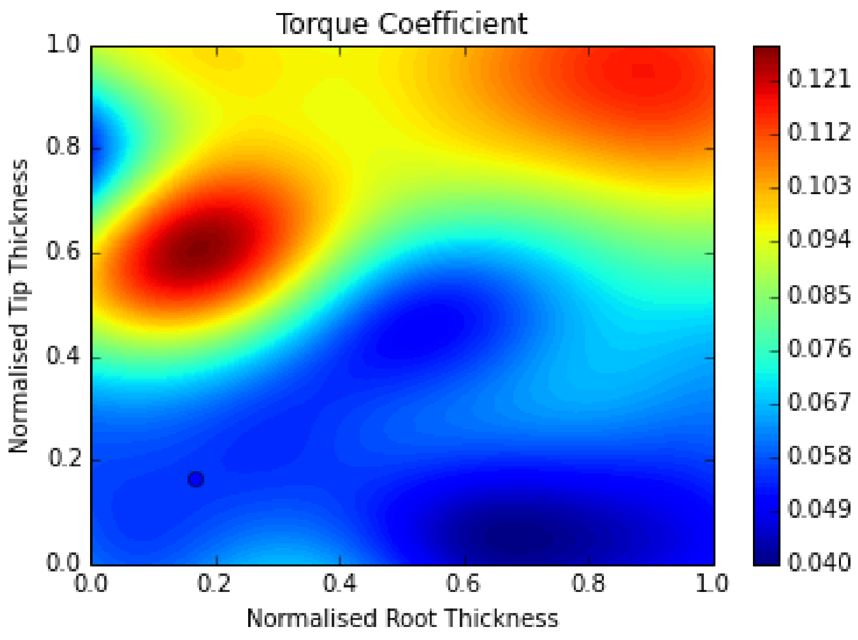


Figure 7.24: Visualisation of torque coefficient against normalised root thickness and tip thickness, pivoted about the base design.

observations on Figure 7.31.

Similar to the thrust landscapes shown in Figures 7.21 and 7.30, the torque variation with tip and root blade thickness, shown in Figures 7.24 and 7.33 respectively, is a complex landscape with multiple peaks. As can be expected, as it has a greater moment arm, the thickness at the tip has a more profound effect on the torque than the root thickness, and a general trend of greater thickness leading to greater section drag resulting in a greater torque is observed. However, this general trend is augmented by the foil hydrodynamics, with the large peaks in torque most likely to be due to flow separation, which is why the root thickness has such a significant effect as the local Reynolds number is lower and thus the flow more likely to separate.

The variation in efficiency with root pitch and hub diameter with all other parameters kept at the optimum value shown in Figure 7.25, is qualitatively similar to the landscape observed about the baseline parameters that is shown in Figure 7.16. It should be noted that, quantatively, the values in Figure 7.25 are much higher and the lowest efficiency in Figure 7.25 is still higher than the highest efficiency in Figure 7.16.

The same comparison of qualitative similarity but quantitative increase can be made between efficiency plots for pitch variation, pivoted about the baseline and optimum points in Figures 7.17 and 7.26 respectively.

Plotting the efficiency variation with blade thickness does have a significant alteration of landscape between the baseline case (Figure 7.18) and the optimum

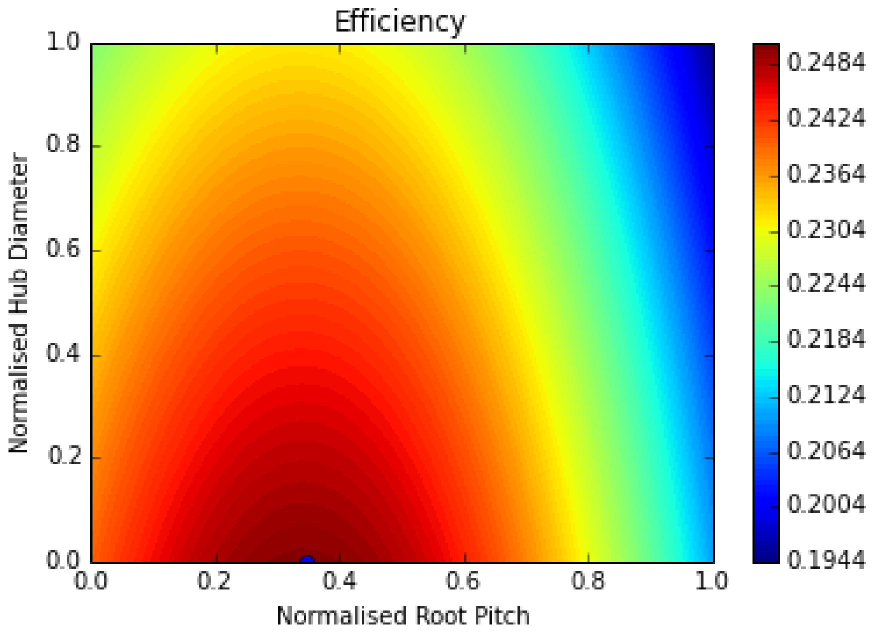


Figure 7.25: Visualisation of efficiency against normalised root pitch and hub diameter, pivoted about the optimum design.

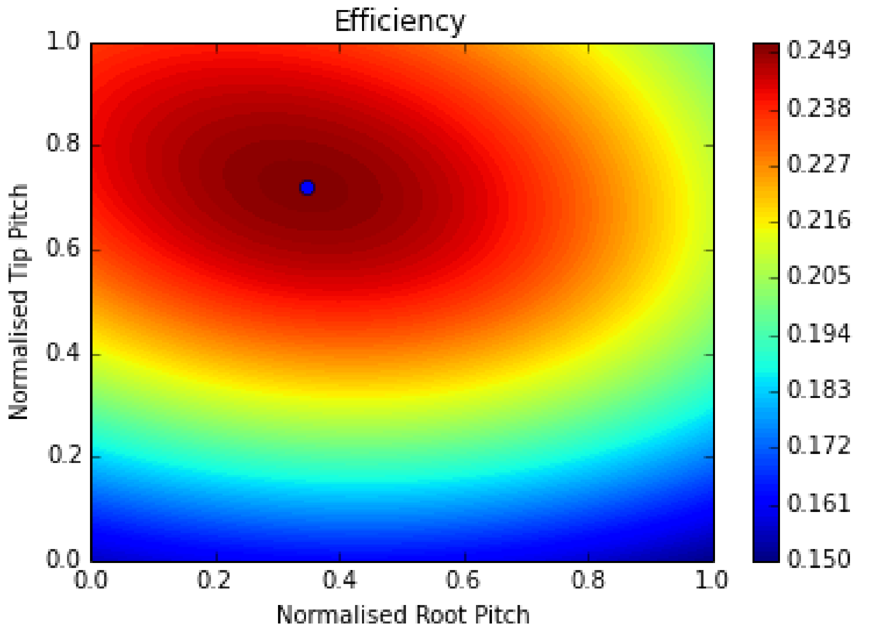


Figure 7.26: Visualisation of efficiency against normalised root pitch and tip pitch, pivoted about the optimum design.

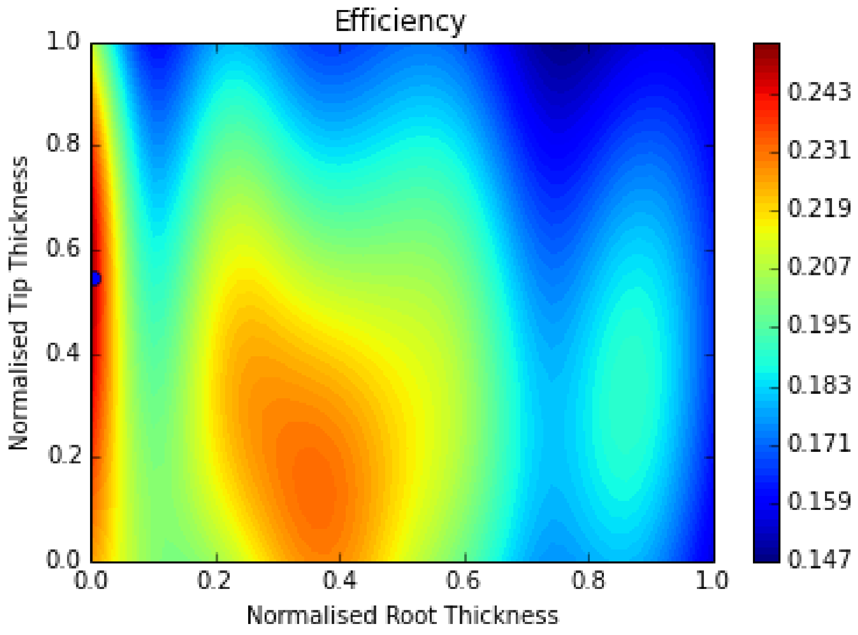


Figure 7.27: Visualisation of efficiency against normalised root thickness and tip thickness, pivoted about the optimum design.

case (Figure 7.27). While there is still a general quantitative increase across the entire landscape for the latter, which is to be expected as all other parameters are at their optimum values, the central peaks seen in Figure 7.18 are diminished in Figure 7.27 and the overall peak value has clearly moved to a minimum root thickness. In practice, depending on the application of the rim driven thruster, it may be better to use a design at the local optimum seen at a normalised root thickness of approximately 0.4 and normalised tip thickness of approximately 0.2, as the increased root thickness will be structurally superior.

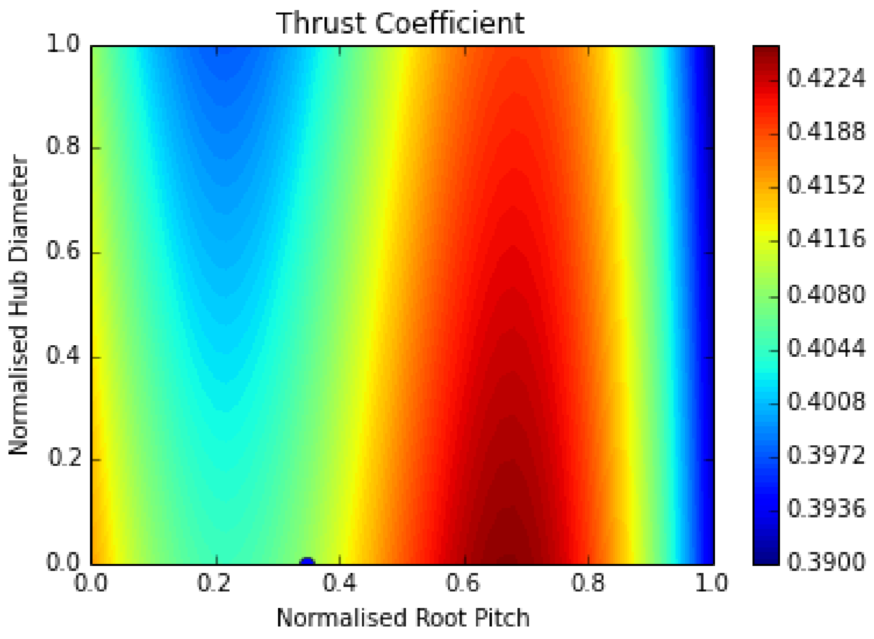


Figure 7.28: Visualisation of thrust coefficient against normalised root pitch and hub diameter, pivoted about the optimum design.

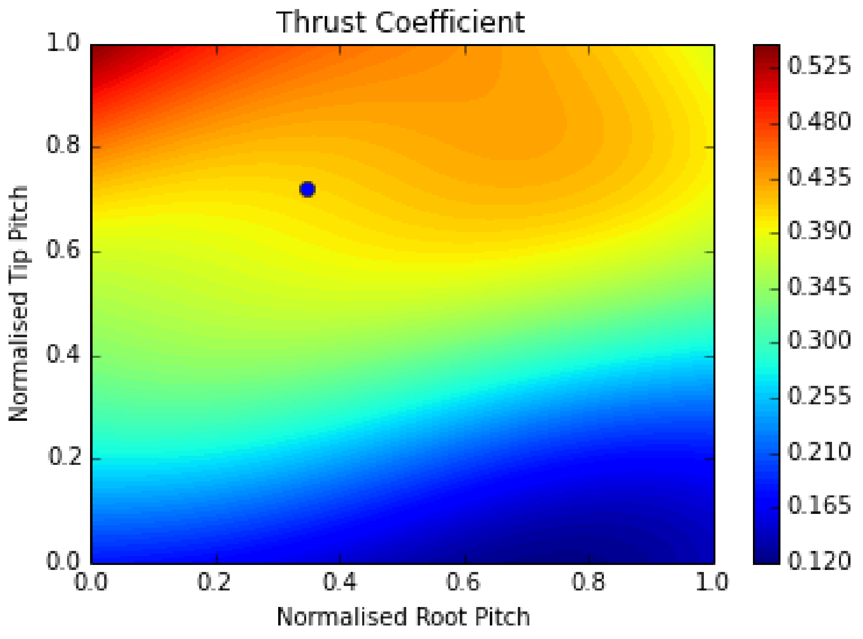


Figure 7.29: Visualisation of thrust coefficient against normalised root pitch and tip pitch, pivoted about the optimum design.

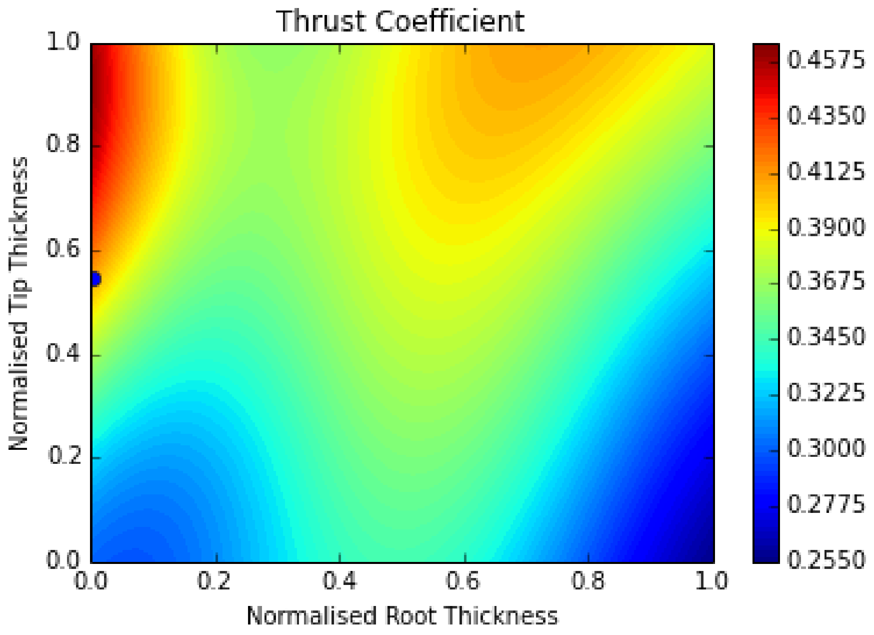


Figure 7.30: Visualisation of thrust coefficient against normalised root thickness and tip thickness, pivoted about the optimum design.

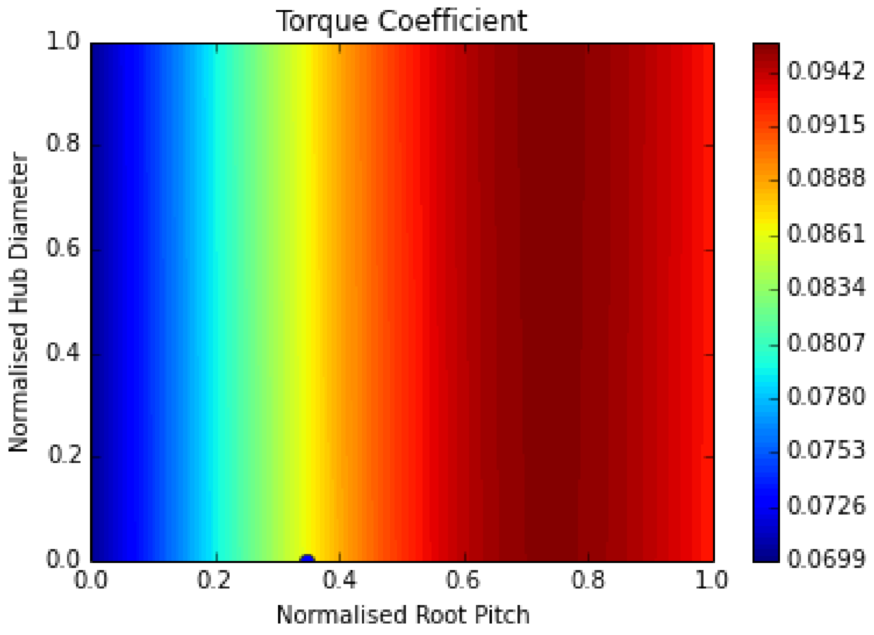


Figure 7.31: Visualisation of torque coefficient against normalised root pitch and hub diameter, pivoted about the optimum design.

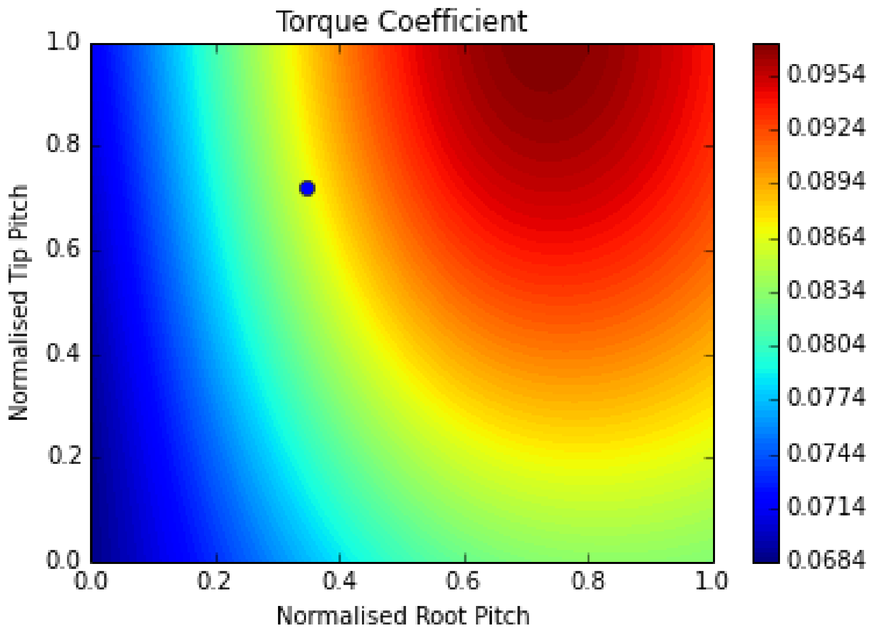


Figure 7.32: Visualisation of torque coefficient against normalised root pitch and tip pitch, pivoted about the optimum design.

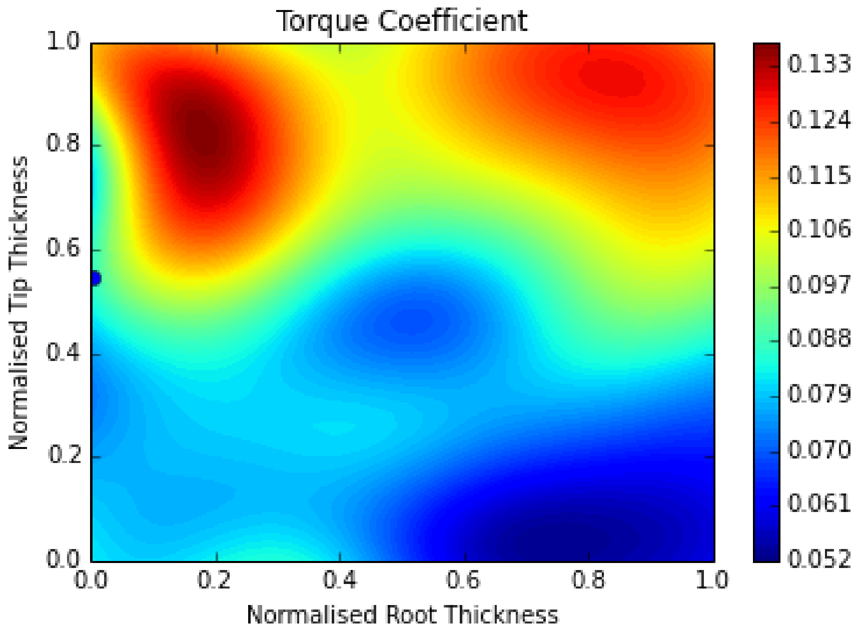


Figure 7.33: Visualisation of torque coefficient against normalised root thickness and tip thickness, pivoted about the optimum design.

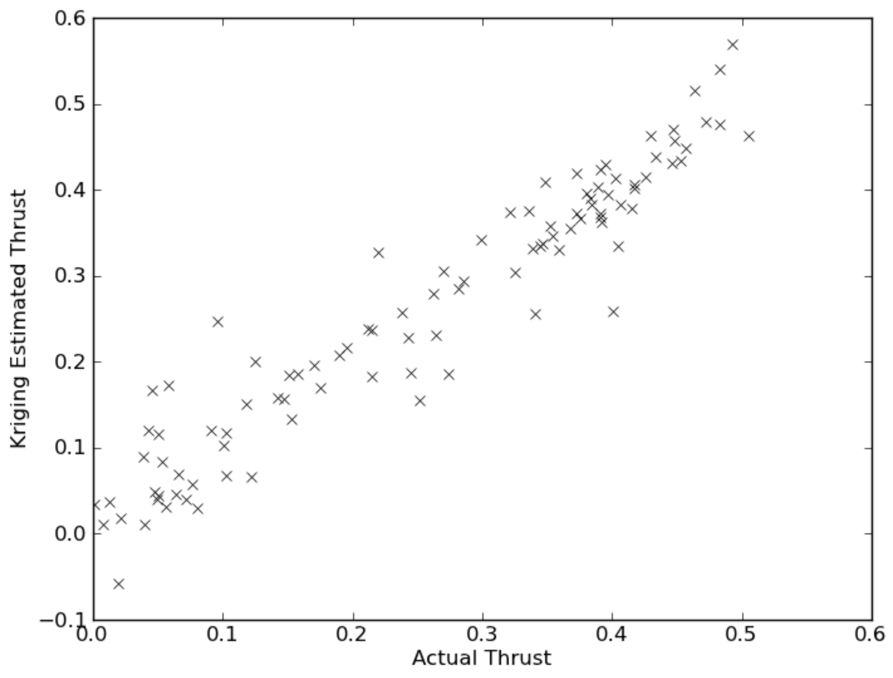


Figure 7.34: Thrust estimated by Kriging surrogate model against thrust calculated by computational fluid dynamics, where the estimated point is left out of the surrogate model construction. $R^2 = 0.921$

7.2.4 Surrogate Modelling Validation

As surrogate models are another kind of computational model, that may or may not be giving an accurate answer when interrogated, it is worthwhile validating the modelling method, which in this case is Kriging, to see how well the modelled data fits the computational fluid dynamics output. Perhaps the best way of doing this is to construct what is known as a 'leave one out plot', where each point in a dataset is excluded in turn, a response surface is constructed and an estimate of the point is made, which is plotted against its actual value. This has been done for thrust, torque and efficiency (Figures 7.34, 7.35 and 7.36 respectively) and allows us to see how close a value predicted by the surrogate model is to its value calculated by computational fluid dynamics, with a perfect surrogate model having all points on the $y = x$ line.

Looking at the estimated thrust compared to the 'actual' thrust, or in this case the thrust predicted by the computational fluid dynamics which is assumed to mirror reality for the purposes of this validation, Figure 7.34 shows us that the Kriging generally has a good agreement. Almost all points lie within 0.1 of the actual value, and greater agreement exists at points in the landscape where thrust

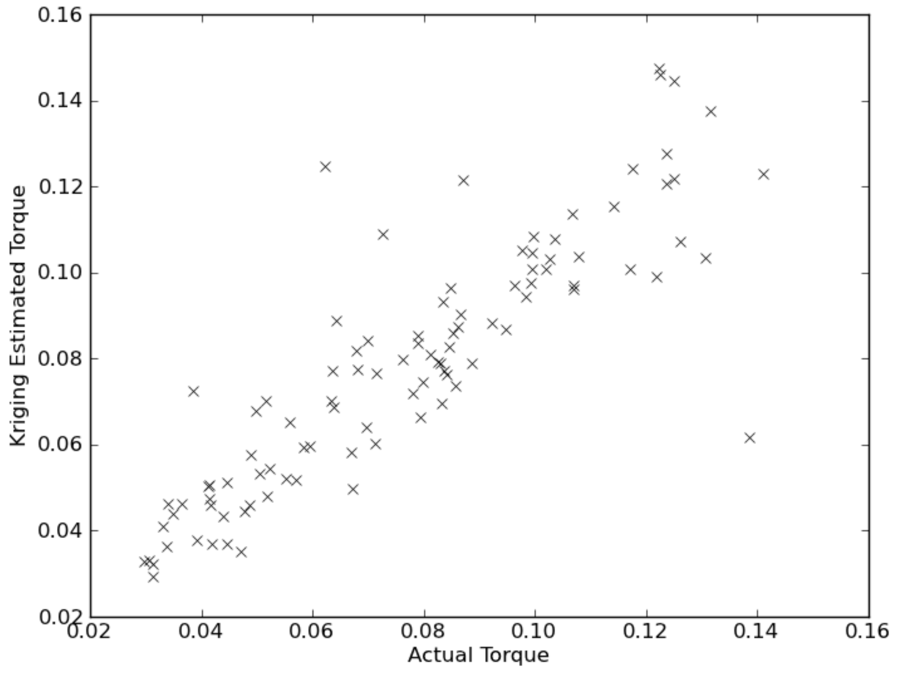


Figure 7.35: Torque estimated by Kriging surrogate model against torque calculated by computational fluid dynamics, where the estimated point is left out of the surrogate model construction. $R^2 = 0.801$

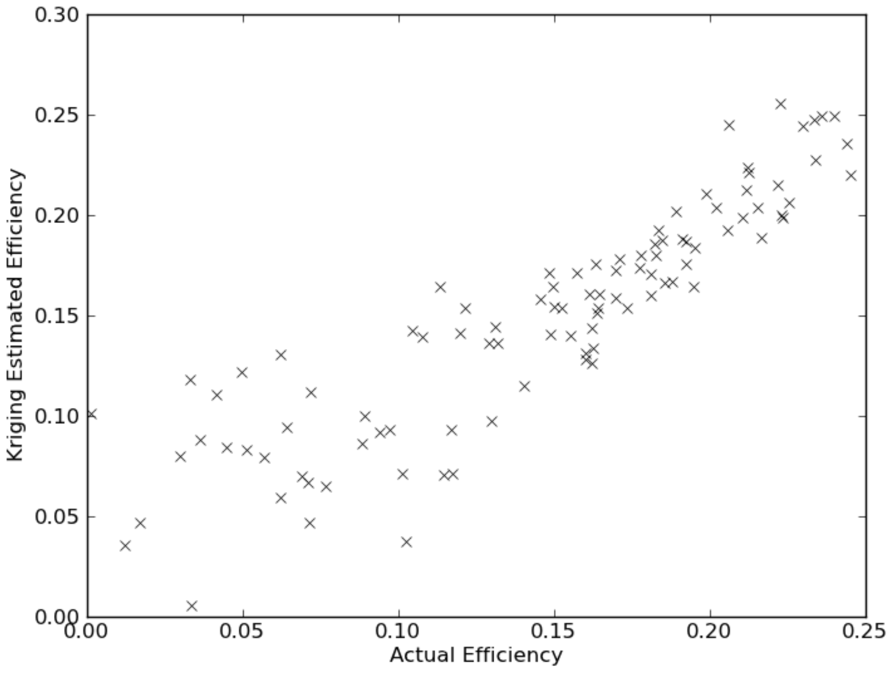


Figure 7.36: Efficiency estimated by Kriging surrogate model against efficiency calculated by computational fluid dynamics, where the estimated point is left out of the surrogate model construction. $R^2 = 0.793$

is higher, due to the greater density of points making up the model in these regions as this is where the optimisation is focussed.

The agreement between Kriging predicted and 'actual' torque, shown in Figure 7.35, at first glance seems to be considerably worse. However, aside from a number of outlying points which skew perceptions, the majority of points are estimated to within 0.01 of their 'actual' values. This is a good achievement considering the undulating, sharp gradiented landscape that is being captured as portrayed in Figures 7.24, 7.32 and 7.33.

The correlation between prediction and measurement of device efficiency depicted in Figure 7.36 shows an interesting trend of increasing agreement with increasing efficiency. This can be expected of a Kriging model that has been used to optimise efficiency, as the higher efficiency regions of the design landscape would be the focus of exploitation infill points and thus the higher number of points, the better the local tuning and thus predictive accuracy.

To quantify the agreements seen in Figures 7.34, 7.35 and 7.36, the root-mean-square error was taken from the results. The values for K_T, K_Q and η were 0.0451, 0.0151 and 0.0286 respectively. Which, taking into account the relative magnitudes of each of the coefficients, results in a percentage error of approximately 10%. This is not a perfect fit, which is perhaps in part due to the noise that comes with numerical simulations and distorts any attempts to fit a surrogate model to it. However, the calculation of root-mean-square error values also includes the outliers that are in the regions where designs are not particularly feasible, thus the fit of the surrogate model is better than first quantified in the regions of interest that are being exploited and explored and worse elsewhere.

The optimisation history in Figure 7.37 tells an interesting story, and at first the optimisation seems to have no convergence, but the first 60 iterations are purely a space filling sample plan and thus any design improvements here (of which there are two) are simply co-incident to the exploration of the design space and subsequent building of a surrogate model. From iteration number 61 onwards, during the update stage, there are only four further improvements on the best design found in the initial sample. The random efficiencies found in the initial sample stage also continue to manifest, although this is not due to a poor optimisation, but an infill strategy that also tries to build a good surrogate model for understanding the design space away from the optimum (see Section 7.2.2). The effects of pure exploitation on the optimisation results can be seen in the last ten iterations, where the evaluated efficiencies are all above 0.22, apart from two outliers.

A criticism of the optimisation performed is that the update strategy may not be the most efficient use of iterations. In terms of producing the best design possible, this is definitely true, however the surrogate model should be a better fit through-

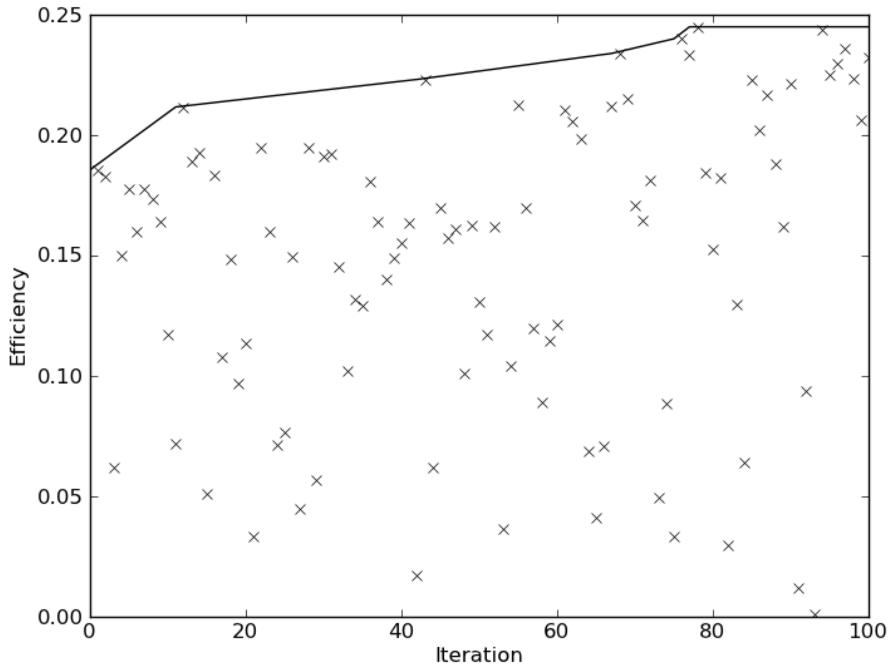


Figure 7.37: Optimisation history of CFD function values.

out the design domain because of the varied update scheme. It is also arguable that the initial sample size is larger than necessary, even though it complies with the 'ten times the number of dimensions' rule of thumb. If an infill strategy based on purely a maximised expected improvement were to be used, then less coverage of an initial sample is required as maximum expected improvement balances both exploration and exploitation and will reach the global optimum given enough iterations, therefore it does not need such a detailed initial surrogate model.

It is also seen in the optimisation history in Figure 7.37 that most of the design improvement came from the random exploration of the initial sample plan. This suggests that further improvement could be obtained by continuing the optimisation for more iterations, which would also refine the surrogate model further.

The Kriging model does not fit the data perfectly, as shown in the leave-one-out plots in Figures 7.34, 7.35 and 7.36, which could suggest that it is a poor choice of model or method. However, if the converse were true and the surrogate model fit the data perfectly, there would not be any need for further infill points as the optimum of the surrogate model would be the optimum of the data being modelled. Thus the general trend of agreement and improvement of agreement in regions of interest (*id est* high efficiency regions) means that the fit of the surrogate model is suitable for this purpose.

7.3 Results from Design Optimisation

A 100mm rim driven thruster design was optimised using the method outlined in the previous sections. From a base geometry that was similar to that of the experimental device, albeit without stators, the optimisation process increased the propulsive efficiency, at the design condition of 3000 revolutions per minute and 1.5

m/s

advance speed, from 18.5% to 24.5%; an absolute increase of 6%. The baseline and optimised geometries are outlined and compared in this section.

The baseline geometry had a non-dimensional co-ordinate of (0.5, 0.5, 0.5, 0.167, 0.167, 0.35). This corresponds to a linear pitch/diameter ratio distribution of 1.0 along the blade, a constant thickness/diameter ratio of 0.045 and a hub diameter of 0.022m. At the design condition, the propulsive efficiency was 18.5% with the complete performance curve shown in Figure 7.40.

After one hundred iterations of designs, an optimal design was found at a non-dimensional co-ordinate of (0.4418, 0.8543, 0.6870, 0.0007, 0.3479, 0.01). This corresponds to a geometry with a pitch diameter ratio distribution that is quadratic with a value of 0.9418 at the origin, 1.3543 halfway to the tip and 1.1870 at the blade tip. The thickness/diameter ratio was linearly distributed from a value of 0.030063 at the origin to a value of 0.061311 at the tip and the hub diameter is 0.0152m. At the design condition, the propulsive efficiency was 24.5% with the complete performance curve shown in Figure 7.40.

7.3.1 Device Performance

The optimisation was of a single operating point of the device, but in reality the performance of the device over a range of advance ratios is important. Thus the characteristics of the optimised device are compared against the baseline device as well the experimental data for the 100mm rim driven thruster in Figures 7.38, 7.39 and 7.40 showing the thrust coefficient, torque coefficient and efficiency respectively.

Figure 7.38 shows the thrust coefficient against advance ratio for the three aforementioned cases. The optimised device has a significantly higher thrust across the range of advance ratios than the baseline design, which in turn has a higher thrust than the experimental data for the device it is emulating, but this is in part due to the omission of stators from the CFD modelling.

Plotting the torque coefficient against advance ratio in Figure 7.39 shows a similar hierarchy to the thrust coefficient with the optimised design being higher than the baseline and experimental designs. This is counter-intuitive, as for a greater efficiency, the torque should be lower, so higher torque should yield a lower effi-

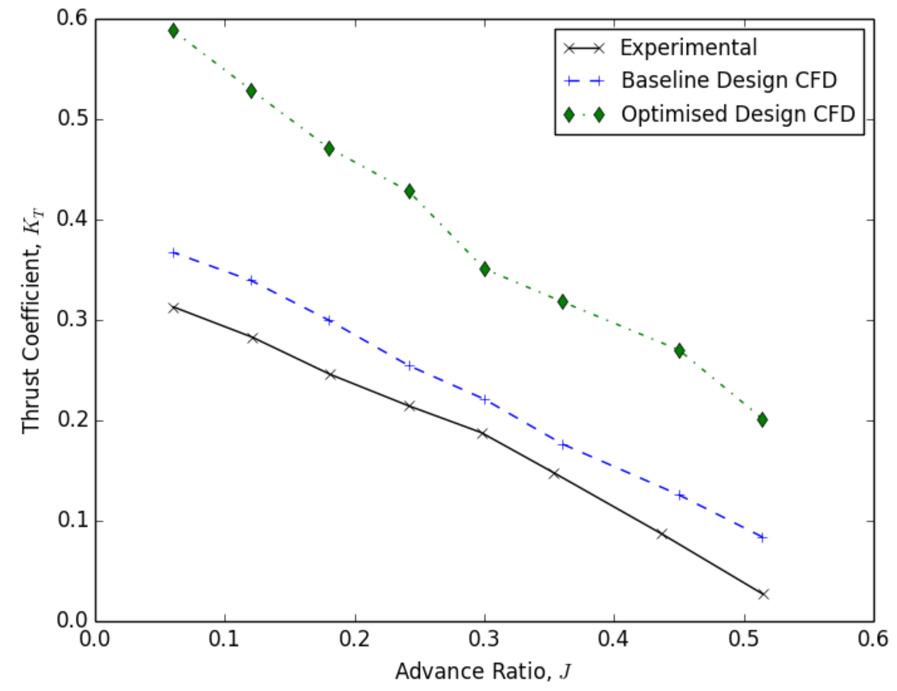


Figure 7.38: Thrust performance of experimental, baseline and optimised 100mm Rim Driven Thrusters.

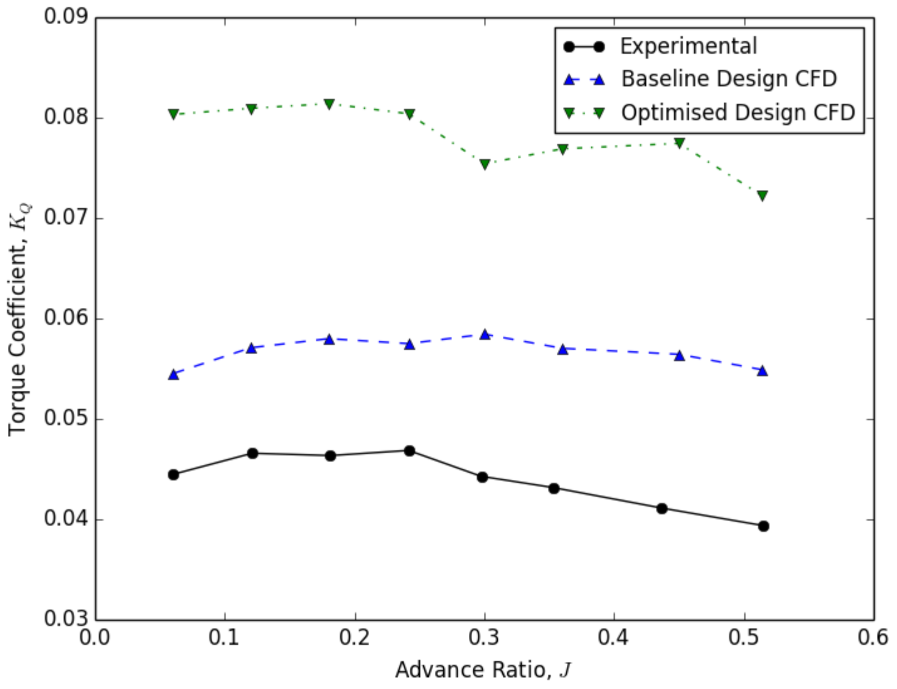


Figure 7.39: Torque performance of experimental, baseline and optimised 100mm Rim Driven Thrusters.

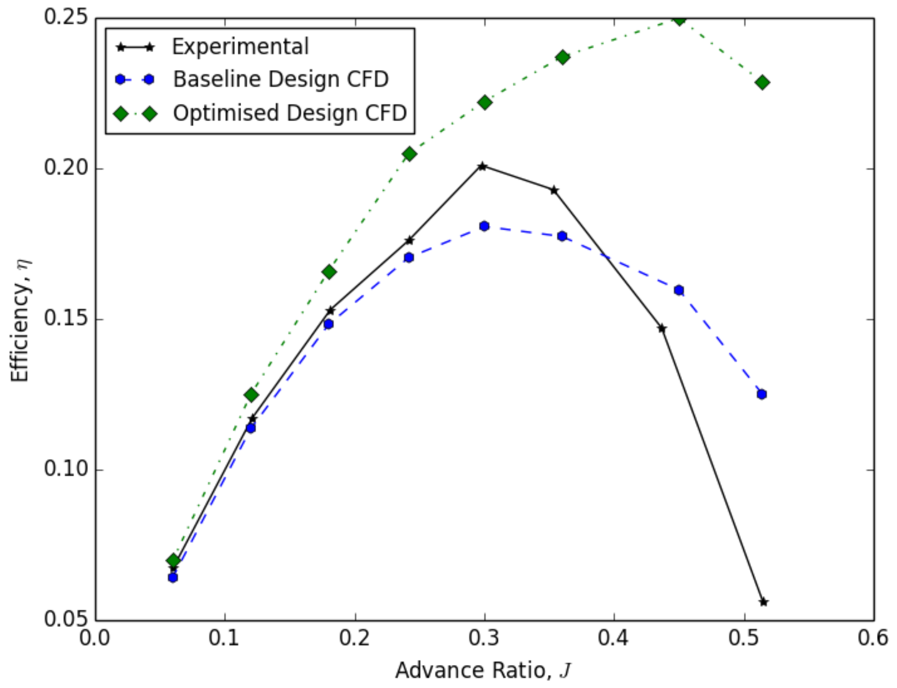


Figure 7.40: Efficiency of experimental, baseline and optimised 100mm Rim Driven Thrusters.

ciency. However, when examining the efficiency against advance ratio in Figure 7.40, the optimised device is clearly more efficient, as the efficiency is the ratio of thrust to torque and it therefore has increased the thrust proportionally more than the increase in torque to yield a greater efficiency.

It is interesting to note that the advance ratio for peak efficiency of the optimised device is different to that of the baseline design and thus different to the design condition. This is further evidence that the optimisation performed is sub-optimal and greater increase in efficiency could be gained from performing more iterations, which should also shift the peak efficiency advance ratio to the design condition if the global optimum design is found.

Further insight can be gained by examining how the contributions of each part vary between the baseline and optimised designs. By decomposing the thrust, torque and efficiency on a component-wise basis, the mechanisms by which the optimised device achieves a greater efficiency can be exposed. It should be noted here that when derived for an individual component, the listed efficiency is not a ‘contribution’ to efficiency *per se*, but the ratio of its thrust contribution to its torque requirement.

The primary contributors to thrust are the blades and the duct, and thus their contribution in both baseline and optimised designs are shown in Figure 7.41. The general trend observed here is that the optimised device produces a higher thrust

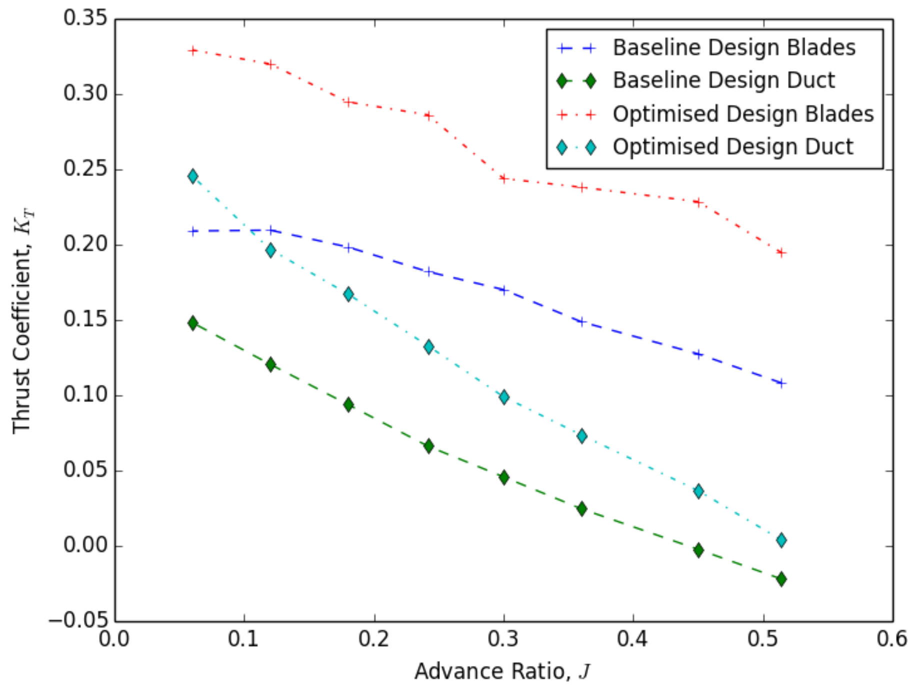


Figure 7.41: Comparison of decomposed thrust performance between baseline and optimised 100mm Rim Driven Thrusters.

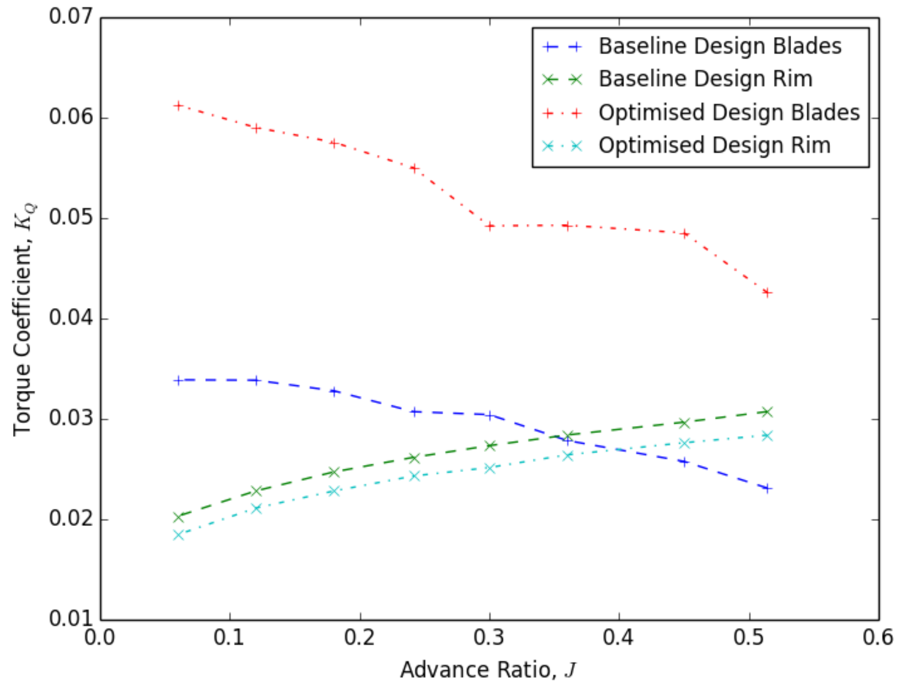


Figure 7.42: Comparison of decomposed torque performance between baseline and optimised 100mm Rim Driven Thrusters.

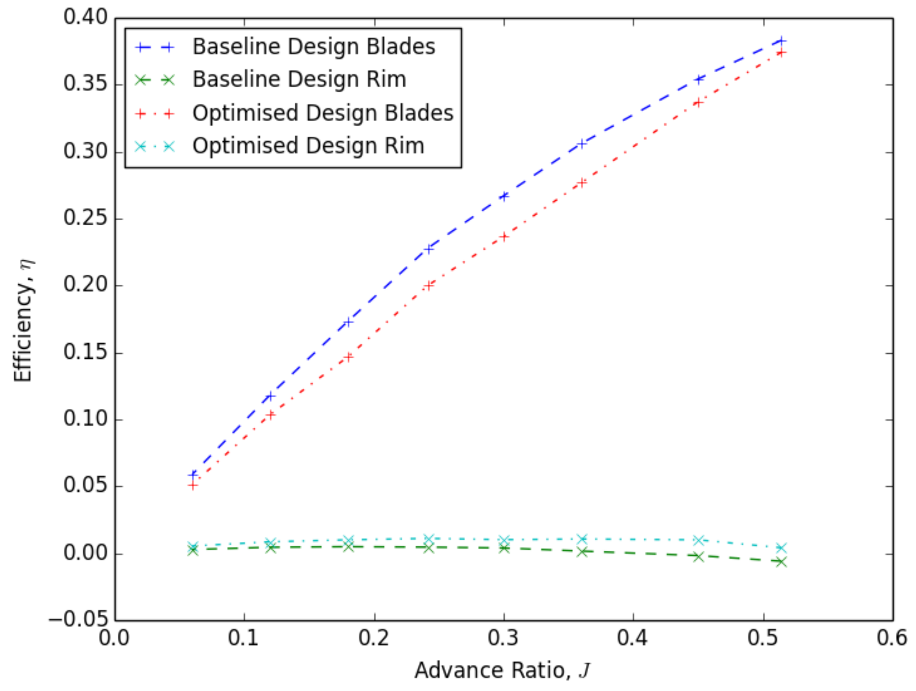


Figure 7.43: Comparison of decomposed component efficiency between baseline and optimised 100mm Rim Driven Thrusters.

on both the blades and the duct, the latter possibly as a consequence of the former. Approximating the blades to an actuator disk, an increase in thrust would equate to an increased pressure difference, also resulting in an increased pressure difference on the duct, thus the net force in the forward direction is increased for both parts.

Torque losses occur on either the blades, rim or hub, but the latter losses are negligible due to the smaller moment arm. Thus the primary concern is with only the torque on the blades and rim, which are shown in Figure 7.42. To some extent, the torque on the blades can be interpereted as the losses involved in producing thrust, whereas the torque on the rim is purely detrimental. The significantly greater torque in the optimised device can then be seen as expected, given the much greater thrust produced, and the reduced torque on the rim is also the intuitive direction to proceed. It is notable that the ratio of rim torque to blade torque is significantly smaller in the optimised design than the baseline design, where rim losses exceed 50% of the total torque above advance ratios of 0.36.

Finally, looking at how component efficiency varies for the blades and rim in Figure 7.43 shows that component efficiency for the rim is very small because very little thrust is produced compared to its torque loss. It is also shown in Figure 7.43 that the component efficiency of both the blades and the rim are increased in the optimised device which, in tandem with the increased thrust on the duct, leads to

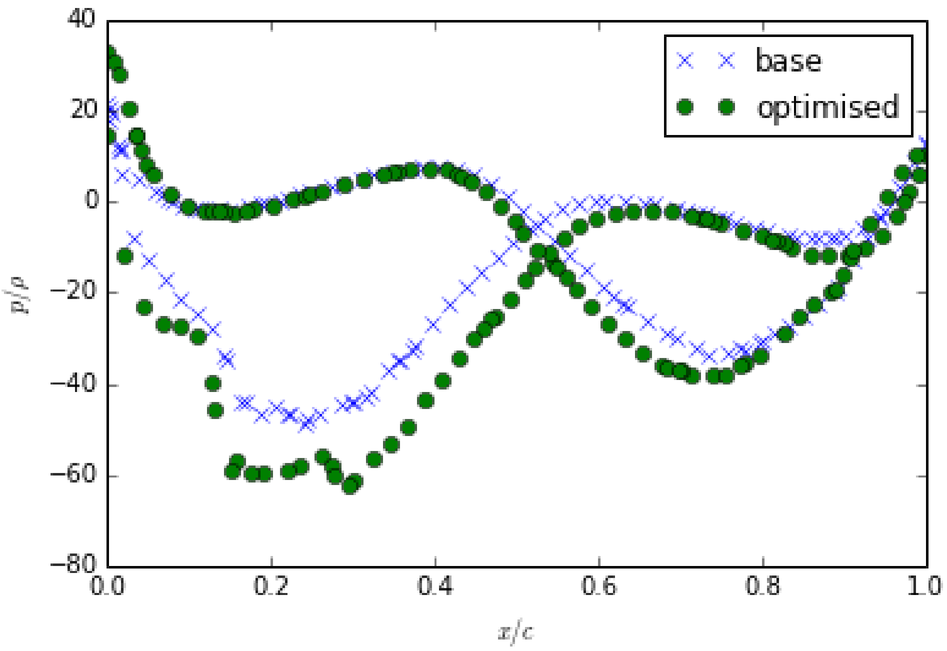


Figure 7.44: Plot of pressure against x co-ordinate on the blades for base and optimised designs at 50% radius.

the performance gains seen in Figure 7.40.

7.3.2 Blade Pressure Profiles

Figures 7.44 and 7.45 show the pressure distribution around the blade for both the baseline and optimised designs. Figure 7.44 shows the pressure around a section halfway along the blade and Figure 7.45 shows the pressure around a section closer to the tip, at 70% along the blade. In both these figures, the leading edge is located at $x/c = 0$ and the trailing edge is consequently at $x/c = 1$. Furthermore, the line for each case that is in the negative region towards the leading edge is the blade face pressure and the line that is in the positive region towards the leading edge corresponds to the pressure on the blade back.

It can be seen from Figures 7.44 and 7.45 that the improvement in performance from the optimised design originates from harder working blades. The optimised design shows overall higher pressures around the blade, as well as a slightly increased area of working section, that is the region where the pressure on the back of the blade exceeds that of the face.

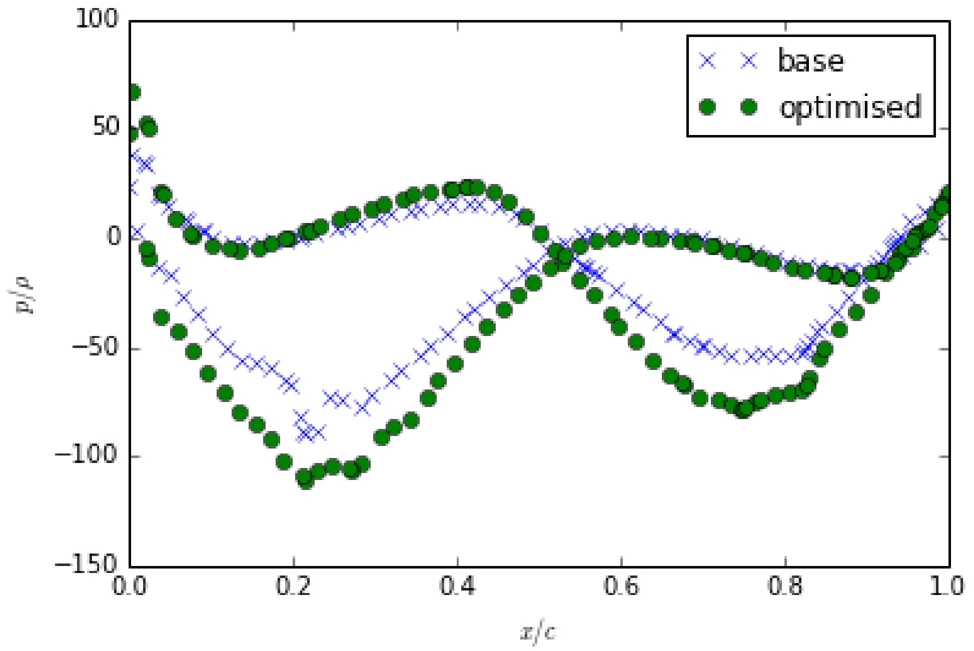


Figure 7.45: Plot of pressure against x co-ordinate on the blades for base and optimised designs at 70% radius.

7.3.3 Cavitation

While cavitation has been excluded from the analyses in this work, as the typical operating conditions of the rim driven thruster are at depths where cavitation is unlikely, it is informative to check the cavitation performance of the optimised design in comparison with the baseline case. This is done from the computational fluid dynamics results by finding the locations within the flow that are below the vapour pressure of water. Although this only highlights the origins of cavitating flow and makes no attempt to model its subsequent transport, it is still informative.

In the original design specification for the device, the operating static pressure condition was defined as 10 metres below sea level (Sharkh et al., 2003). Therefore two operating conditions are considered in this analysis, surface operation at an absolute pressure of 100kPa (Figures 7.46, 7.47, 7.48 and 7.49) and subsurface operation at approximately 10 metres depth, represented by a static pressure of 200kPa (Figures 7.50 and 7.51).

For a given vapour pressure of $P_v = 2.34$kPa, taken from ITS-90 (International Temperature Scale of 1990) for a temperature of 20 degrees Celcius, the gauge pressure at which cavitation occurs is given by Equation (7.5).

$$P_{\text{gauge}} = P_v - P_{\text{ref}} \tag{7.5}$$

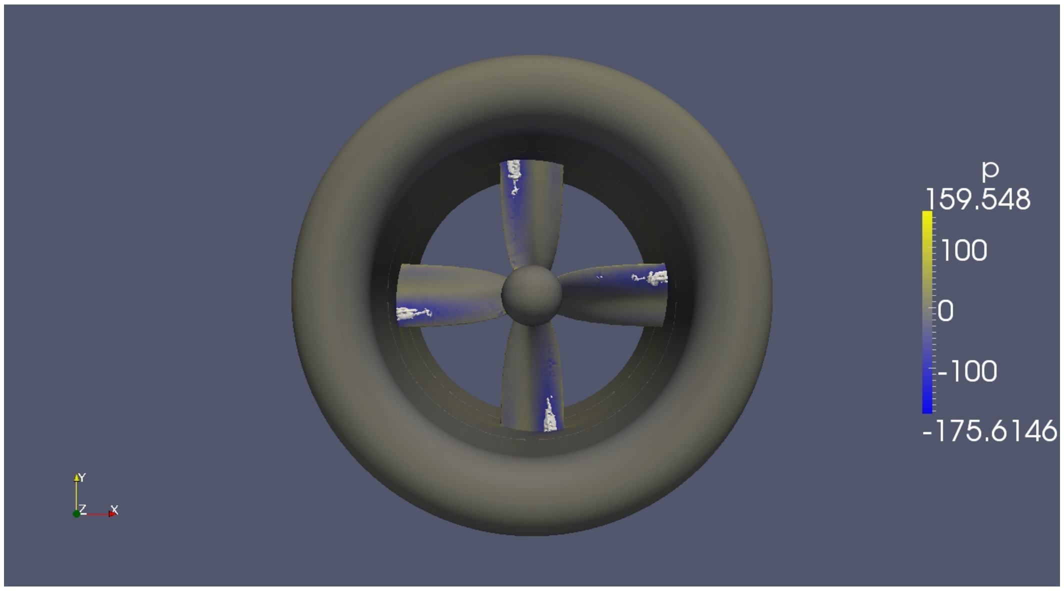


Figure 7.46: Surface pressure plot on base design, front view with cavitation shown for surface propulsion case.

where P_{ref} is the reference pressure, *id est* the absolute pressure at which gauge pressure is zero, which is 100kPa or 200kPa for surface or subsurface operation respectively. Similarly, if we approximate the pressure profile for a depth to be a linear profile of surface pressure (100kPa) plus 10kPa per metre of depth, d, then if the minimum pressure in a flow field, $P_{\min}$, is known the cavitation free operating depth is given by Equation (7.6).

$$d = \frac{1}{10}\left(P_v - P_{\min} - 100\right). \tag{7.6}$$

Figure 7.46 shows cavitation on the front (thus the blade face) of the base design in the 100kPa absolute pressure case. There is minor cavitation on the blade face, whereas the blade back has no cavitation in Figure 7.47. The optimised design in the surface propulsion condition shows heavier cavitation in Figure 7.48 and even includes unexpected cavitation occuring on the blade back in Figure 7.49. However, in the original design condition at a 10 metre depth for cavitation free operation, Figures 7.50 and 7.51 show that the optimised design is not expected to cavitate, thus making the optimised design a successful one in that respect.

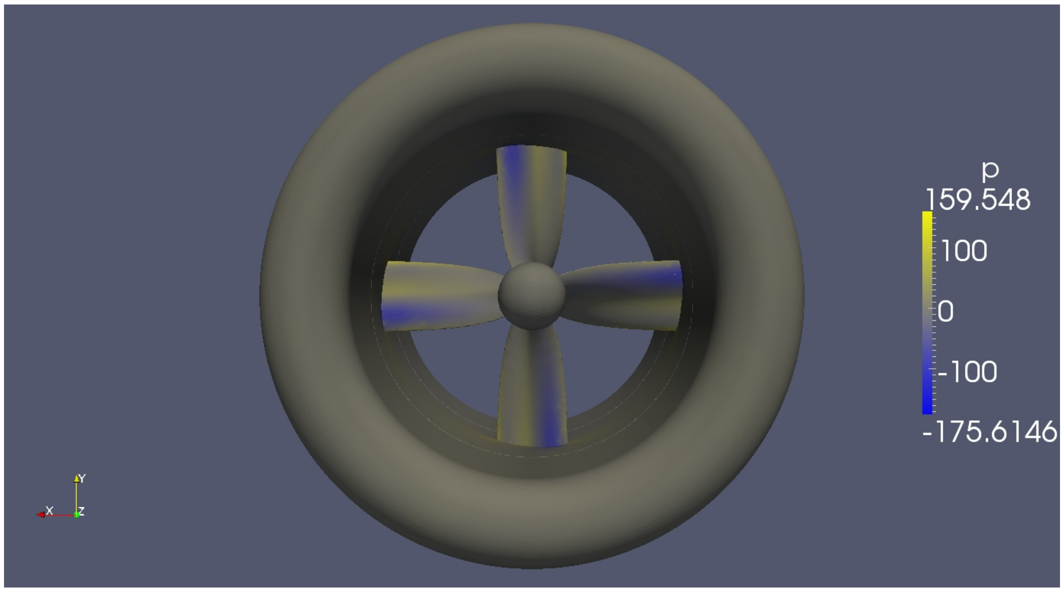


Figure 7.47: Surface pressure plot on base design, rear view with cavitation shown for surface propulsion case.

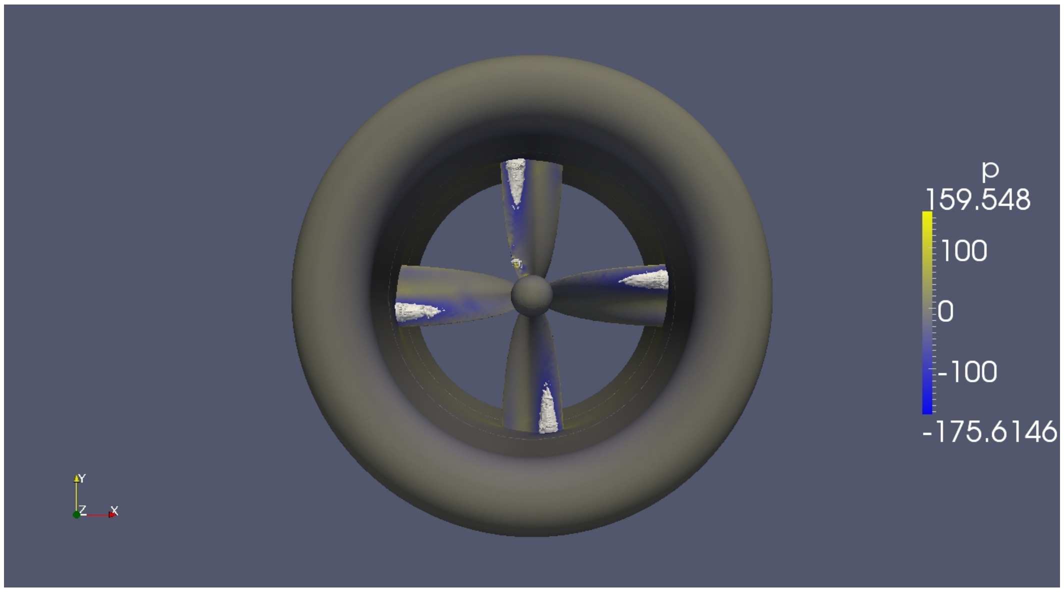


Figure 7.48: Surface pressure plot on optimised design, front view with cavitation shown for surface propulsion case.

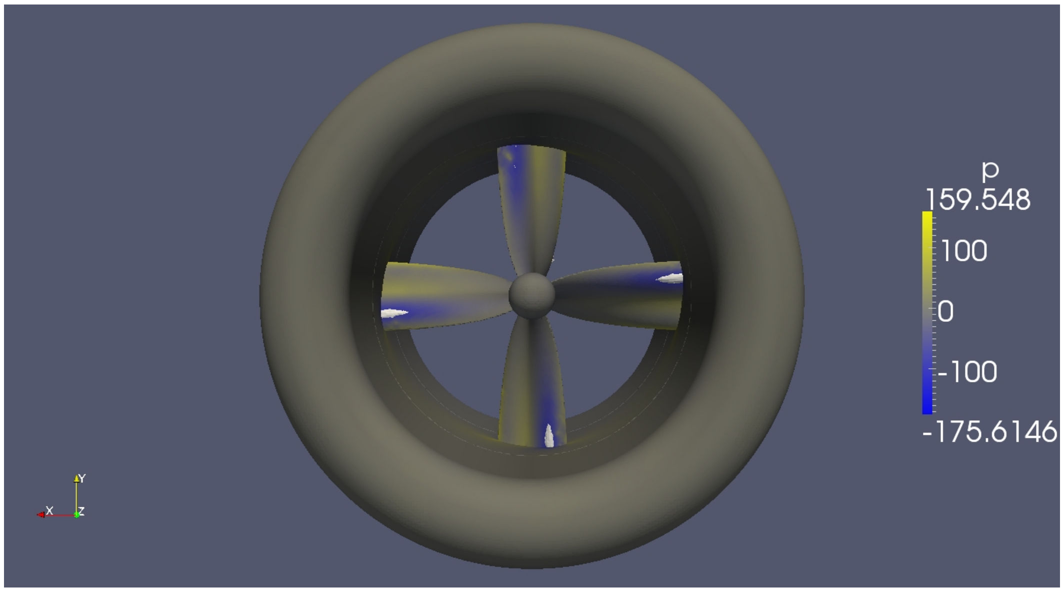


Figure 7.49: Surface pressure plot on optimised design, rear view with cavitation shown for surface propulsion case.

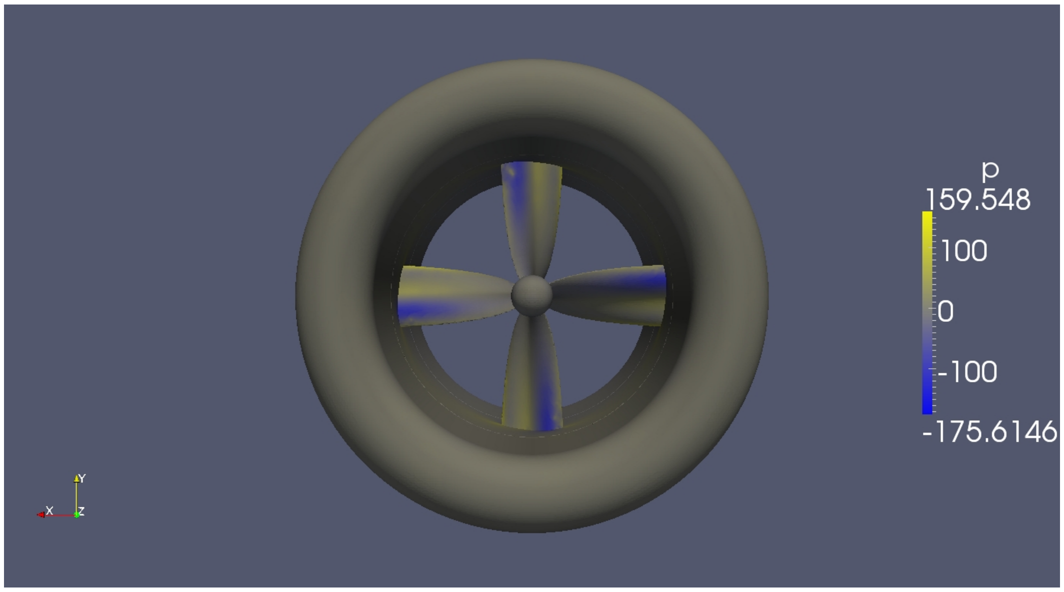


Figure 7.50: Surface pressure plot on optimised design, front view with cavitation shown for 10 metre depth case.

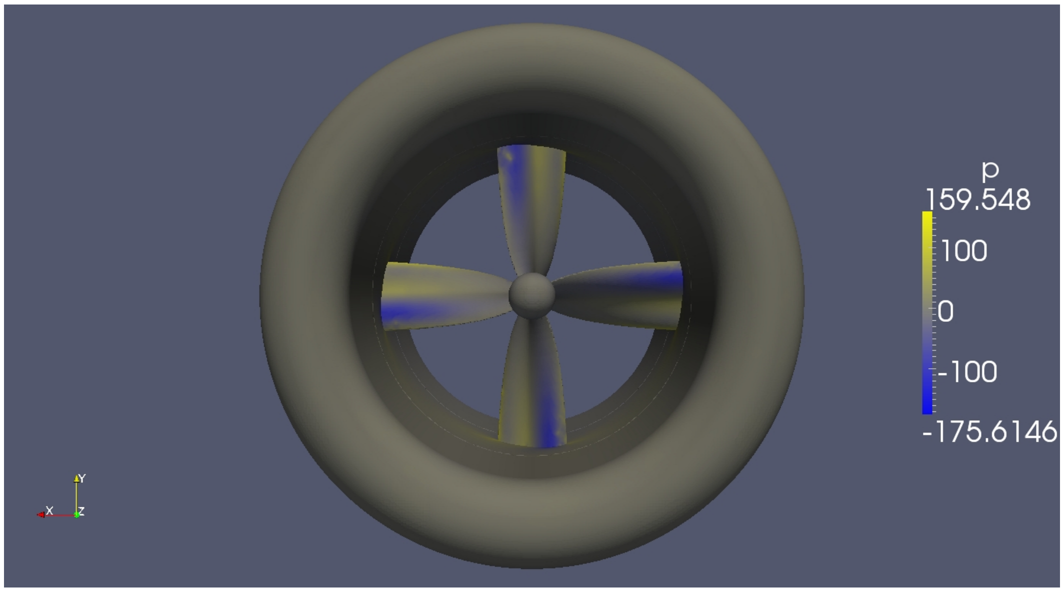


Figure 7.51: Surface pressure plot on optimised design, rear view with cavitation shown for 10 metre depth case.

Chapter 8

Conclusions and Future Work

In summary, the primary original contributions of this thesis are:

- Insights into the unsteady rotor-stator interaction in rim driven thrusters and recommendations on the simulation thereof.
- Design insights into the best pitch and thickness distribution for rim driven thrusters.
- An improved design of 100mm rim driven thruster which is 6% more efficient.
- A robust and automated design optimisation methodology and extensible framework that is suitable for complex design landscapes.

8.1 Conclusions

A thorough mesh verification and validation procedure has been conducted for both steady and unsteady computational fluid dynamics methods outlined in this work. It was found that for steady state simulations using MRFSimpleFoam a domain size extending five propeller diameters to the inlet and six propeller diameters in the radial direction is sufficient. Unsteady simulations using pimpleDyMFoam were found to require a smaller domain with a distance of two propeller diameters to any domain wall being sufficient. This provides domain size independence despite a relatively high blockage ratio. Both MRFSimpleFoam and pimpleDyMFoam solvers were validated using a Wageningen B-Series propeller geometry and reported performance to within 5% of experimental results across a range of advance ratios. A discrepancy between experimental and computational data was apparent at low advance ratios, but this was attributed to the simulating the sustained bollard pull condition, rather than the instantaneous bollard pull condition that is given in the experimental data.

For solving low advance ratio flows of marine propulsors, the k-ω SST turbulence model was found to be more robust due to its better performance in adverse pressure gradients and separation handling. The RNG k-ϵ turbulence model was also investigated and was found to produce good agreement with experimental data except at low advance ratios whereupon the solutions diverged.

MRFSimpleFoam was found to be unsuitable as a program for the complete modelling of a rim driven thruster due to the inaccurate capture of rotor-stator interaction by the 'frozen rotor' treatment of the interface between rotating and stationary reference frames. A pseudo-unsteady simulation of the interaction can be made with MRFSimpleFoam by performing multiple steady-state simulations at different rotor positions. This can improve the accuracy of results if an average of these rotor positions is subsequently taken, however, this is not an accurate description of the flowfield. The 'frozen rotor' formulation of MRFSimpleFoam does make it a better solver for pre-solution of unsteady simulation compared to contemporary mixing plane methods.

Results from unsteady simulation of the rotor-stator interaction concurred with the hypothesis that the flowfield is not accurately captured. The unsteady results also showed thrust loading varies by up to 40% of the mean through one rotation and torque loading varies by 5% of the mean. Further investigation of the unsteady results showed flow oscillation around the stators, subsequently affecting the angle of incidience of the incoming flow to the blades, thus identifying the cause of such a significant variation which could be mitigated by increasing the gap between stator and rotor.

From the steady-state results, it is possible to deduce the components of the rim driven thruster which are most likely to yield improvements in a design optimisation study. This is helpful in informing the dimensions that need to be parameterised, as the number of parameters should ideally be kept to a minimum. The pitch distribution could be improved for the special case of rim mounted blades in a duct as boundary layer development on the duct reduces the inflow speed at the blade tips. Viscous torque on the rim is shown to contribute to approximately a third of the hydrodynamic losses in a rim driven thruster so any reduction here would be beneficial to efficiency. A significant portion of thrust at low advance ratios, and drag at higher advance ratios, is attributed to the casing and so there is scope here for the casing to be optimised to a particular operating regime, whether it be low speed manouevering or high speed propulsion. Finally, there may be some benefit to shaping the stators to improve the interacting rotor-stator flow.

To find an accurate estimate of the losses in the annulus between the rim and casing, a combination of established analytical models have been tested as well as a theoretical minimum bound based on a linear velocity profile. The Bilgen & Boulos and Daily & Nece models were found to overpredict torque in combination, which is due to not accounting for the interactivity between regions or the effect of the axial pressure gradient. A need to investigate this in future work been identified and a proposed computational method to approach this problem is outlined in Section 6.5.

An iterative design optimisation of a 100mm rim driven thruster has been facilitated through the automation of geometry and mesh generation as well as subsequent post processing to calculate and return an objective function. In the process, some useful libraries of functions that create geometry, write files in stereolithographic (.stl) format and interface with OpenFOAM via the command line have been created. Due to their modular design, these allow the subsequent automation of design optimisation studies for cases beyond that of rim driven thrusters or even marine propulsion.

Surrogate modelling techniques have been applied to not only allow the search and optimisation of the design space, but to also enable efficient parametric interrogation to allow visualisation and understanding of the produced response surface. In particular this has highlighted the sensitivity of the bi-directional foil section, as small changes in thickness can have a significant impact on the section hydrodynamics and consequently on the overall performance of the device. Therefore an optimum thickness must be selected with consideration of the operating condition, in particular the Reynolds number experienced by the blade sections.

Search, optimisation and update of the surrogate model was performed using a genetic algorithm, to reach an improved device design. The improved design differs from the baseline case by having increased pitch, in particular at the tip of the blade, also combined with an increased blade thickness at the tip, and a minimised hub diameter to allow a greater working area. Due to the lack of tip leakage losses, the blade tips of a rim driven thruster are able to work harder without incurring the efficiency penalty of their rimless counterparts. A general predisposition towards greater thrust in a rim driven thruster is also desirable due to the significant torque losses on the rim which must be mitigated.

Overall, a computational fluid dynamics simulation method has been investigated, verified, validated and employed in a design optimisation study. As a result of the optimisation of the surrogate model, insight has been gained into the hydrodynamic design of rim driven thrusters and design with a theoretical improvement of efficiency of 6% was found. A robust framework has been created for the design optimisation of rim driven thrusters that is easily employed, interrogated and extended for future works.

8.2 Suggestions for Future Work

It is the nature of research projects that they are extensible and can take many tangents, and this thesis represents only one of many trajectories of investigation. The avenues that were left unexplored, or that have consequently opened up, constitute the final section of this thesis so that they may be pursued by those that follow.

In terms of the modelling and simulation of rim driven thrusters, the key unknown is the power lost to viscous torque in the annulus region. A potential future computational study of this region was outlined in Section 6.5 with the interactivity of the end effects and axial pressure gradient posing questions that are unanswered here. The simulation could also be further extended to compare the unsteady RANS calculation with a large eddy simulation of the rotor-stator interaction, to ascertain the validity of unsteady RANS in this application.

The optimisation conducted in this work could also be extended, firstly through further iteration of the existing study and secondly through extension of the framework to investigate other questions. Two key things to implement and investigate are functional constraints and different optimisation strategies including gradient based methods, which could be compared to the original work here. Further extension could be the implementation of either multi-objective optimisation (*exempli gratia* increase efficiency while minimising minimum pressure) or multi-fidelity models using the computationally expensive unsteady simulations to include the effects of rotor-stator interaction as the higher fidelity model.

References

T. Abramowski, K. Zelazny, and T. Szelangiewicz. Numerical analysis of influence of ship hull form modification on ship resistance and propulsion characteristics - part III - influence of hull form modification on screw propeller efficiency. *Polish Maritime Research*, 17(1):10–13, 2010.

N. Alin, R. E. Bensow, C. Fureby, T. Huuva, and U. Svennberg. Current capabilities of DES and LES for submarines at straight course. *Journal of Ship Research*, 54 (3):184–196, 2010.

M. Auvinen, J. Ala-Juusela, N. Pedersen, and T. Siikonen. Time-accurate turbomachinery simulations with open-source CFD; flow analysis of a single-channel pump with OpenFOAM. In *Fifth European Conference on Computational Fluid Dynamics, Lisbon, Portugal*, 2010.

P. Batten, U. Goldberg, O. Peroomian, and S. Chakravarthy. Recommendations and best practice for the current state of the art in turbulence modelling. *International Journal of Computational Fluid Dynamics*, 23(4):363–374, 2009.

W. M. J. Batten. *Numerical Predictions and Experimental Analysis of Small Clearance Ratio Taylor-Couette Flows.* PhD thesis, University of Southampton, School of Engineering Sciences, August 2002.

W. M. J. Batten, N. W. Bressloff, and S. R. Turnock. Transition from vortex to wall driven turbulence production in the Taylor-Couette system with a rotating inner cylinder. *International Journal for Numerical Methods in Fluids*, 38:207–226, 2002.

W. M. J. Batten, S. R. Turnock, N. W. Bressloff, and S. M. Abu-Sharkh. Turbulent Taylor-Couette vortex flow between large radius ratio concentric cylinders. *Experiments in Fluids*, 36:419–421, 2004.

M. Beaudoin and H. Jasak. Development of a generalized grid interface for turbomachinery simulations with OpenFOAM. In *Open Source CFD International Conference*, 2008.

E. Benini. Multiobjective design optimization of B-screw series propellers using evolutionary algorithms. *Marine Technology*, 40(4):229–238, 2003.

E. Benini. Significance of blade element theory in performance prediction of marine propellers. *Ocean Engineering*, 31:957–974, 2004.

R. E. Bensow and G. Bark. Implicit LES predictions of the cavitating flow on a propeller. *Journal of Fluids Engineering*, 132, 2010.

S. Berger, M. Druckenbrod, M. Greve, M. Abdel-Maksoud, and L. Greitsch. An efficient method for the investigation of propeller hull interaction. In *Proceedings of the 14th Numerical Towing Tank Symposium*, 2011.

E. Bilgen and R. Boulos. Functional dependence of torque coefficient of coaxial cylinders on gap width and Reynolds numbers. *Transactions of ASME, Journal of Fluids Engineering*, 95(1):122–126, 1973.

J. M. Bousquet and P. Gardarein. Improvements on computations of high speed propeller unsteady aerodynamics. *Aerospace Science and Technology*, 7:465–472, 2003.

T. E. Brockett. On the maximum efficiency of some marine propulsors in open water. *International Shipbuilding Progress*, 50:147–169, 2003.

Q.-M. Cao, F.-W. Hong, D.-H. Tang, F.-L. Hu, and L.-Z. Lu. Prediction of loading distribution and hydrodynamic measurements for propeller blades in a rim driven thruster. *Journal of Hydrodynamics*, 24:50–57, 2012.

S. Carcangiu, A. Fanni, and A. Montisci. Computational fluid dynamics simulations of an innovative system of wind power generation. In *Proceedings of the 2011 COMSOL Conference in Stuttgart*, 2011.

J. Carlton. *Marine Propellers and Propulsion: Second Edition*. Elsevier, 2007.

M. R. Castelli, P. Cioppa, and E. Benini. Influence of turbulence model, grid resolution and free-stream turbulence intensity on the numerical simulation of the flow field around an inclined flat plate. *World Academy of Science, Engineering and Technology*, 64:192–197, 2012.

E. B. Caster. Ducted propeller designs for improved backing performance. In *Symposium on Ducted Propellers*, number 7, 1973.

F. Celik and M. Guner. Energy saving device of stator for marine propellers. *Ocean Engineering*, 34:850–855, 2007.

J. E. Choi, K.-S. Min, J. H. Kim, S. B. Lee, and H. W. Seo. Resistance and propulsion characteristics of various commercial ships based on CFD results. *Ocean Engineering*, 37:549–566, 2010.

G. Coriolis. Sur les équations du movement relatif des systèmes de corps. *J. Ec. Polytech.*, 15:142–154, 1935.

D. Corson, R. Jaiman, and F. Shakib. Industrial application of RANS modelling: Capabilities and needs. *International Journal of Computational Fluid Dynamics*, 23(4):337–347, 2009.

L. Da-Qing. Validation of RANS predictions of open water performance of a highly skewed propeller with experiments. In *Conference of Global Chinese Scholars on Hydrodynamics*, pages 520–528, 2006.

J. W. Daily and R. E. Nece. Chamber dimension effects on induced flow and frictional resistance of enclosed rotating disks. *Journal of Fluids Engineering*, 82: 217–230, 1960.

F. Di Felice, M. Felli, M. Liefvendahl, and U. Svennberg. Numerical and experimental analysis of the wake behaviour of a generic submarine propeller. In *First International Symposium on Marine Propulsors*, 2009.

M. Drela. XFOIL: An analysis and design system for low Reynolds number airfoils. In *Conference on Low Reynolds Number Airfoil Aerodynamics, University of Notre Dame*, 1989.

J. W. English and S. J. Rowe. Some aspects of ducted propeller propulsion. In *Symposium on Ducted Propellers*, number 3, 1973.

P. E. Farrell and J. R. Maddison. Conservative interpolation between volume meshes by local Galerkin projection. *Computer Methods in Applied Mechanics and Engineering*, 200:89–100, 2011.

A. Forrester, A. Sobester, and A. Keane. *Engineering Design via Surrogate Modelling*. Wiley, 2008.

I. Funeno. Hydrodynamic optimal design of ducted azimuth thrusters. In *First International Symposium on Marine Propulsors*, June 2009.

M. M. Gaafary, H. S. El-Kilani, and M. M. Moustafa. Optimum design of B-series marine propellers. *Alexandria Engineering Journal*, 50:13–18, 2011.

D. Gerr. *Propeller Handbook*. International Marine, 2001.

A. N. Hayati, S. M. Hashemi, and M. Shams. A study on the effect of the rake angle on the performance of marine propellers. *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, 226: 940–955, 2012.

A. N. Hayati, S. M. Hashemi, and M. Shams. A study on the behind-hull performance of marine propellers astern autonomous underwater vehicles at diverse angles of attack. *Ocean Engineering*, 59:152–163, 2013.

S. Huang, X.-Y. Zhu, C.-Y. Guo, and X. Chang. CFD simulation of propeller and rudder performance when using additonal thrust fins. *Journal of Marine Science and Application*, 6(4):27–31, 2007.

A. W. Hughes, S. R. Turnock, and S. M. Sharkh. CFD modelling of a novel electromagnetic tip-driven thruster. In *Proceedings of the Tenth International Offshore and Polar Engineering Conference*, volume 2, pages 294–298, 2000.

D. Jones, M. Schonlau, and W. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global Optimization*, 13:455–492, 1998.

J. Kaufmann and V. Bertram. Comparison of multi-reference frame and sliding interface propeller models for RANSE computations of ship-propeller interaction. In *Proceedings of the 14th Numerical Towing Tank Symposium*, 2011.

J. E. Kerwin, S. A. Kinnas, J.-T. Lee, and W. Z. Shih. A surface panel method for the hydrodynamic analysis of ducted propellers. *Transactions of the Society of Naval Architects and Marine Engineers*, 95, 1987.

S. A. Kinnas, S.-H. Chang, L. He, and J. T. Johannessen. Performance prediction of a cavitating rim driven tunnel thruster. In *First International Symposium on Marine Propulsors*, 2009.

A. N. Kolmogorov. Equations of turbulent motion of an incompressible fluid. *Izvestia Academy of Sciences, USSR; Physics*, 6(1–2):56–58, 1942.

N. Kornev, A. Taranov, and E. Shchukin. Development of hybrid URANS-LES methods for flow simulation in the ship stern area. In *Proceedings of the 14th Numerical Towing Tank Symposium*, 2011.

T. Koronowicz, Z. Krzemianowksi, T. Tuszkowska, and J. A. Szantyr. A complete design of contra-rotating propellers using the new computer system. *Polish Maritime Research*, 17(1):14–24, 2010.

D. G. Krige. A statistical approach to some basic mine valuation problems on the witwatersrand. *Journal of the Chemical, Metallurgical and Mining Society of South Africa*, December 1951.

G. Kuiper. *The Wageningen Propeller Series.* MARIN, 1992.

W. Lam, D. J. Robinson, G. A. Hamill, S. Raghunathan, and C. Kee. Simulation of a ship's propeller wash. In *Proceedings of the Sixteenth (2006) International Offshore and Polar Engineering Conference*, pages 457–462, 2006.

B. E. Launder and B. I. Sharma. Application of the energy dissipation model of turbulence to the calculation of flow near a spinning disc. *Letters in Heat and Mass Transfer*, 1(2):131–138, 1974.

M. Lea, D. Thompson, B Van Blarcom, J. Eaton, J. Friesch, and J. Richards. Scale model testing of a commercial rim-driven propulsor pod. *Journal of Ship Production*, 19(2):121–130, 2003.

S. Leone, C. Testa, L. Greco, and F. Salvatore. Computational analysis of self-pitching propellers performance in open water. *Ocean Engineering*, 64:122–134, 2013.

Y. Li and F. Wang. Numerical investigation of performance of an axial-flow pump with inducer. *Journal of Hydrodynamics*, 19:705–711, 2007.

M. Liefvendahl, N. Alin, M. Chapuis, C. Fureby, U. Svennberg, and C. Troeng. Ship and propulsor hydrodynamics. In *Fifth European Conference on Computational Fluid Dynamics*, 2010.

H.-C. Lin, B.-C. Chen, and Y.-F. Chen. The lowest stability and bifurcation in supercritical Taylor vortices. *International Journal of Computational Fluid Dynamics*, 24(6):227–233, 2010.

H. Liu, Y. Ren, K. Wang, D. Wu, W. Ru, and M. Tan. Research of inner flow in a double blades pump based on OpenFOAM. *Journal of Hydrodynamics*, 24: 226–234, 2012a.

Y. Liu, P. Zhao, Q. Wang, and Z. Chen. URANS computation of cavitating flows around skewed propellers. *Journal of Hydrodynamics*, 24:339–346, 2012b.

T. P. Lloyd, S. R. Turnock, and V. F. Humphrey. Unsteady CFD of a marine current turbine using OpenFOAM with generalised grid interface. In *Proceedings of the 14th Numerical Towing Tank Symposium*, 2011.

N.-X. Lu, R. E. Bensow, and G. Bark. LES of unsteady cavitation on the delft twisted foil. *Journal of Hydrodynamics*, 22(5):742–749, 2010.

M. Manna and A. Vacca. Torque reduction in Taylor-Couette flows subject to an axial pressure gradient. *Journal of Fluid Mechanics*, 639:373–401, 2009.

F. R. Menter. Review of the shear-stress transport model experience from an industrial perspective. *International Journal of Computational Fluid Dynamics*, 23 (4):305–316, 2009.

M. D. Morris and T. J. Mitchell. Exploratory designs for computational experiments. *Technometrics*, 33:161–174, 1995.

C. S. Morros, J. M. F. Oro, and K. M. A. Diaz. Numerical modelling and flow analysis of a centrifugal pump runnning as a turbine: Unsteady flow structures and its effects on the global performance. *International Journal for Numerical Methods in Fluids*, 65:542–562, 2011.

S. Muntean, H. Nilsson, and R. F. Susan-Resiga. 3D numerical analysis of the unsteady turbulent swirling flow in a conical diffuser using FLUENT and OpenFOAM. In *3rd IAHR International Meeting of the Workgroup on Cavitation and Dynamic Problems in Hydraulic Machinery and Systems*, number C4, pages 155–164, 2009.

M. Nimmo. CFD of a rim driven thruster. 3rd Year Individual Project, University of Southampton, 2011.

C. Pashias, S. R. Turnock, and S. M. Sharkh. Design optimisation of a bi-directional integrated thruster. In *Proceedings of the Propellers/Shafting Symposium*, pages 1–13, 2003.

O. Petit, M. Page, M. Beaudoin, and H. Nilsson. The ERCOFTAC centrifugal pump OpenFOAM case-study. In *3rd IAHR International Meeting of the Workgroup on Cavitation and Dynamic Problems in Hydraulic Machinery and Systems*, 2009.

O. Petit, A. I. Bosioc, H. Nilsson, S. Muntean, and R. F. Susan-Resiga. A swirl generator case study for OpenFOAM. In *25th IAHR Symposium on Hydraulic Machinery and Systems*, 2010.

A. B. Phillips, S. R. Turnock, and M. Furlong. Comparisons of CFD simulations and in-service data for the self propelled performance of an autonomous underwater vehicle. In *27th Symposium on Naval Hydrodynamics*, 2008.

S. H. Rhee and S. Joshi. Computational validation for flow around a marine propeller using unstructured mesh based Navier-Stokes solver. *JSME International Journal, Series B*, 48(3):562–570, 2005.

J. Sacks, W. J. Welch, T. J. Mitchell, and H. P. Wynn. Design and analysis of computer experiments. *Statistical Science*, 4(4):409–423, November 1989.

F. Salvatore, H. Streckwall, and T. van Terwisga. Propeller cavitation modelling by CFD - results from the VIRTUE 2008 Rome workshop. In *First International Symposium on Marine Propulsors*, 2009.

S. M. Sharkh, S. R. Turnock, and G. Draper. Performance of a tip-driven electric thruster for unmanned water vehicles. In *Proceedings of the International Offshore and Polar Engineering Conference*, volume 2, pages 321–324, 2001.

S. M. Sharkh, S. R. Turnock, and A. W. Hughes. Design and performance of an electric tip-driven thruster. In *Proceedings of the Institution of Mechanical Engineers, Part M: Journal of Engineering for the Maritime Environment*, volume 217, 2003.

P. R. Spalart. RANS modelling into a second century. *International Journal of Computational Fluid Dynamics*, 23(4):291–293, 2009.

F. Vesting and R. Bensow. Propeller blade optimisation applying response surface methodology. In *Proceedings of the 14th Numerical Towing Tank Symposium*, 2011.

J. Wang, J. Piechna, and N. Muller. A novel design of composite water turbine using CFD. *Journal of Hydrodynamics*, 24:11–16, 2012.

D. C. Wilcox. Reassessment of the scale determining equations for advanced turbulence models. *American Institute of Aeronautics and Astronautics Journal*, 26 (11):1299–1310, 1988.

D. C. Wilcox. *Turbulence Modeling for CFD.* DCW Industries, 1994.

V. Yakhot, S. A. Orszag, S. Thangam, T. B. Gatski, and C. G. Speziale. Development of turbulence models for shear flows by a double expansion technique. *Physics of Fluids A*, 4(7):1510–1520, 1992.

A. Y. Yakovlev, M. A. Sokolov, and N. V. Marinich. Numerical design and experimental verification of a rim-driver thruster. In *Second International Symposium on Marine Propulsors*, 2011.

J. Ye, Y. Xiong, F. Li, and Z. Wang. Numerical prediction of blade frequency noise of cavitating propeller. *Journal of Hydrodynamics*, 24:371–377, 2012.

Z. Zeng and G. Kuiper. Blade section design of marine propellers with maximum cavitation inception speed. *Journal of Hydrodynamics*, 24:65–75, 2012.

D. Zhang, W. Shi, B. Chen, and X. Guan. Unsteady flow analysis and experimental investigation of axial-flow pump. *Journal of Hydrodynamics*, 22:35–43, 2010.

Z.-R. Zhang, H. Liu, S.-P. Zhu, and F. Zhao. Application of CFD in ship engineering design practice and ship hydrodynamics. In *Conference of Global Chinese Scholars on Hydrodynamics*, pages 315–322, 2006.

B. Zhu, H. Wang, L. Wang, and S. Cao. Three-dimensional vortex simulation of unsteady flow in hydraulic turbines. *International Journal for Numerical Methods in Fluids*, 69:1679–1700, 2012.

Z. Zhu and S. Fang. Numerical investigation of cavitation performance of ship propellers. *Journal of Hydrodynamics*, 24:347–353, 2012.

Appendix A

Propeller Blade Surface Co-ordinate Generating Program

This is a collection of Python functions written to automate the generation of co-ordinate data for the geometry of a propeller, with particular focus on replicating the data for a Wageningen B4-70 propeller with section data and radial distributions obtained from Kuiper (1992).

```
"""
ajblades.py

Propeller blade functions package,
Generates a set of x,y,z co-ordinates for a propeller blade
based on blade profiles at radial stations.

Written to be easily extendable to any type of propeller.

(c) copyright Aleksander Dubas 2011-2013
"""


def bladegen(section, chord, thickness, pitch, D, rake=0, edgefactor=1.0):
    """
    Function to generate blades, takes arguments of four functions and the
    propeller diameter, returns 11 lists of lists which are x,y,z co-ordinates
    at radial stations of 0.15 to 1, with varied steps,
    i.e. r02=[[xs],[ys],[zs]] at radial station of r/R = 0.2

    Parameters
    ----------
    section:    function
        A function that returns a normalised set of co-ordinates
        of the blade section
        inputs are r/R
    chord:  function
        A function that returns the absolute chord
        inputs are r/R and D
    thickness:  function
        A function that returns the abolute thickness
        inputs are r/R and D
    pitch:  function
        A function that returns the pitch angle in degrees
        inputs are r/R
    rake:   number  (default 0)
        Rake angle in degrees, defined as positive aft
    edgefactor: number  (default 1.0)
        Factor by which to scale the outer points.
        e.g. doming(0.99)/slicing(1.01) to create the desired blade tip.

    Returns
    -------
    r015:   n length list of 3D co-ordinates
        Points for the 15% blade section.
    r02:   n length list of 3D co-ordinates
        Points for the 20% blade section.
    r025:   n length list of 3D co-ordinates
        Points for the 25% blade section.
    r03:   n length list of 3D co-ordinates
```

```
␣␣␣␣␣␣␣␣Points␣for␣the␣30%␣blade␣section.
␣␣␣␣r04:␣␣␣n␣length␣list␣of␣3D␣co−ordinates
␣␣␣␣␣␣␣␣Points␣for␣the␣40%␣blade␣section.
␣␣␣␣r05:␣␣␣n␣length␣list␣of␣3D␣co−ordinates
␣␣␣␣␣␣␣␣Points␣for␣the␣50%␣blade␣section.
␣␣␣␣r06:␣␣␣n␣length␣list␣of␣3D␣co−ordinates
␣␣␣␣␣␣␣␣Points␣for␣the␣60%␣blade␣section.
␣␣␣␣r07:␣␣␣n␣length␣list␣of␣3D␣co−ordinates
␣␣␣␣␣␣␣␣Points␣for␣the␣70%␣blade␣section.
␣␣␣␣r08:␣␣␣n␣length␣list␣of␣3D␣co−ordinates
␣␣␣␣␣␣␣␣Points␣for␣the␣80%␣blade␣section.
␣␣␣␣r09:␣␣␣n␣length␣list␣of␣3D␣co−ordinates
␣␣␣␣␣␣␣␣Points␣for␣the␣90%␣blade␣section.
␣␣␣␣r10:␣␣␣n␣length␣list␣of␣3D␣co−ordinates
␣␣␣␣␣␣␣␣Points␣for␣the␣tip␣blade␣section.

␣␣␣␣Co−ordinates␣assume␣z+␣as␣the␣thrust␣direction.
␣␣␣␣"""
    from math import tan, radians

    # generate section geometry and store in zs, xs
    # note L.E. is +z, T.E. is −z and suction is +x, pressure is −x
    r015zs, r015xs = section(0.15)
    r02zs, r02xs = section(0.2)
    r025zs, r025xs = section(0.25)
    r03zs, r03xs = section(0.3)
    r04zs, r04xs = section(0.4)
    r05zs, r05xs = section(0.5)
    r06zs, r06xs = section(0.6)
    r07zs, r07xs = section(0.7)
    r08zs, r08xs = section(0.8)
    r09zs, r09xs = section(0.9)
    r1zs, r1xs = section(1.0)

    # generate section ys
    r015ys = []
    r02ys = []
    r025ys = []
    r03ys = []
    r04ys = []
    r05ys = []
    r06ys = []
    r07ys = []
    r08ys = []
    r09ys = []
    r1ys = []
    for z in r015zs:
        r015ys.append(0.5*D*0.15)
    for z in r02zs:
        r02ys.append(0.5*D*0.2)
    for z in r025zs:
        r025ys.append(0.5*D*0.25)
    for z in r03zs:
        r03ys.append(0.5*D*0.3)
    for z in r04zs:
        r04ys.append(0.5*D*0.4)
    for z in r05zs:
        r05ys.append(0.5*D*0.5)
    for z in r06zs:
        r06ys.append(0.5*D*0.6)
    for z in r07zs:
        r07ys.append(0.5*D*0.7)
    for z in r08zs:
        r08ys.append(0.5*D*0.8)
    for z in r09zs:
        r09ys.append(0.5*D*0.9)
    for z in r1zs:
        r1ys.append(edgefactor*0.5*D)

    # generate chord and multiply geometry zs by chord
    r015zs = map(lambda x: x*chord(0.15, D), r015zs)
    r02zs = map(lambda x: x*chord(0.2, D), r02zs)
    r025zs = map(lambda x: x*chord(0.25, D), r025zs)
    r03zs = map(lambda x: x*chord(0.3, D), r03zs)
    r04zs = map(lambda x: x*chord(0.4, D), r04zs)
    r05zs = map(lambda x: x*chord(0.5, D), r05zs)
    r06zs = map(lambda x: x*chord(0.6, D), r06zs)
    r07zs = map(lambda x: x*chord(0.7, D), r07zs)
    r08zs = map(lambda x: x*chord(0.8, D), r08zs)
    r09zs = map(lambda x: x*chord(0.9, D), r09zs)
    r1zs = map(lambda x: x*chord(1.0, D), r1zs)

    # generate thickness and multiply xs by thickness
    r015xs = map(lambda x: x*thickness(0.15, D), r015xs)
    r02xs = map(lambda x: x*thickness(0.2, D), r02xs)
    r025xs = map(lambda x: x*thickness(0.25, D), r025xs)
    r03xs = map(lambda x: x*thickness(0.3, D), r03xs)
    r04xs = map(lambda x: x*thickness(0.4, D), r04xs)
    r05xs = map(lambda x: x*thickness(0.5, D), r05xs)
    r06xs = map(lambda x: x*thickness(0.6, D), r06xs)
    r07xs = map(lambda x: x*thickness(0.7, D), r07xs)
    r08xs = map(lambda x: x*thickness(0.8, D), r08xs)
    r09xs = map(lambda x: x*thickness(0.9, D), r09xs)
    r1xs = map(lambda x: x*thickness(1.0, D), r1xs)

    # generate pitch angle and rotate co-ordinates about this angle
    # if L.E. (+z) faces right with suction face (+x) on top
    # then positive pitch is clockwise
    r015zs, r015xs = rotatecw(r015zs, r015xs, pitch(0.15))
    r02zs, r02xs = rotatecw(r02zs, r02xs, pitch(0.2))
    r025zs, r025xs = rotatecw(r025zs, r025xs, pitch(0.25))
    r03zs, r03xs = rotatecw(r03zs, r03xs, pitch(0.3))
    r04zs, r04xs = rotatecw(r04zs, r04xs, pitch(0.4))
    r05zs, r05xs = rotatecw(r05zs, r05xs, pitch(0.5))
    r06zs, r06xs = rotatecw(r06zs, r06xs, pitch(0.6))
    r07zs, r07xs = rotatecw(r07zs, r07xs, pitch(0.7))
    r08zs, r08xs = rotatecw(r08zs, r08xs, pitch(0.8))
    r09zs, r09xs = rotatecw(r09zs, r09xs, pitch(0.9))
    r1zs, r1xs = rotatecw(r1zs, r1xs, pitch(1.0))

    # apply propeller rake
    if rake != 0:
        r015zs = map(lambda x: x - (0.15*0.5*D)*tan(radians(rake)), r015zs)
        r02zs = map(lambda x: x - (0.2*0.5*D)*tan(radians(rake)), r02zs)
        r025zs = map(lambda x: x - (0.25*0.5*D)*tan(radians(rake)), r025zs)
        r03zs = map(lambda x: x - (0.3*0.5*D)*tan(radians(rake)), r03zs)
        r04zs = map(lambda x: x - (0.4*0.5*D)*tan(radians(rake)), r04zs)
        r05zs = map(lambda x: x - (0.5*0.5*D)*tan(radians(rake)), r05zs)
        r06zs = map(lambda x: x - (0.6*0.5*D)*tan(radians(rake)), r06zs)
        r07zs = map(lambda x: x - (0.7*0.5*D)*tan(radians(rake)), r07zs)
        r08zs = map(lambda x: x - (0.8*0.5*D)*tan(radians(rake)), r08zs)
        r09zs = map(lambda x: x - (0.9*0.5*D)*tan(radians(rake)), r09zs)
        r1zs = map(lambda x: x - (1.01*0.5*D)*tan(radians(rake)), r1zs)

    # wrap co-ordinates onto a cylindrical section
    r015xs, r015ys = map2cyl(r015xs, r015ys)
    r02xs, r02ys = map2cyl(r02xs, r02ys)
    r025xs, r025ys = map2cyl(r025xs, r025ys)
    r03xs, r03ys = map2cyl(r03xs, r03ys)
    r04xs, r04ys = map2cyl(r04xs, r04ys)
    r05xs, r05ys = map2cyl(r05xs, r05ys)
    r06xs, r06ys = map2cyl(r06xs, r06ys)
    r07xs, r07ys = map2cyl(r07xs, r07ys)
    r08xs, r08ys = map2cyl(r08xs, r08ys)
    r09xs, r09ys = map2cyl(r09xs, r09ys)
    r1xs, r1ys = map2cyl(r1xs, r1ys)

    # make list of lists
    r015 = [r015xs, r015ys, r015zs]
    r02 = [r02xs, r02ys, r02zs]
    r025 = [r025xs, r025ys, r025zs]
    r03 = [r03xs, r03ys, r03zs]
    r04 = [r04xs, r04ys, r04zs]
    r05 = [r05xs, r05ys, r05zs]
    r06 = [r06xs, r06ys, r06zs]
    r07 = [r07xs, r07ys, r07zs]
    r08 = [r08xs, r08ys, r08zs]
    r09 = [r09xs, r09ys, r09zs]
    r1 = [r1xs, r1ys, r1zs]

    return r015, r02, r025, r03, r04, r05, r06, r07, r08, r09, r1


def outputb470files(D):
    """
    Writes the co-ordinate files for a B4-70 propeller with diameter D.

    Parameters
    ----------
    D: number
        Diameter of B4-70 propeller.

    Returns
    -------
    None
    """
    # get co-ordinates
    r015, r02, r025, r03, r04, r05, r06, r07, r08, r09, r1 =\
        bladegen(b4567s, b470c, b4t, b4p10, D, rake=15)

    # open files
    f015 = file("r015.txt", "w")
    f02 = file("r02.txt", "w")
    f025 = file("r025.txt", "w")
    f03 = file("r03.txt", "w")
    f04 = file("r04.txt", "w")
    f05 = file("r05.txt", "w")
    f06 = file("r06.txt", "w")
    f07 = file("r07.txt", "w")
    f08 = file("r08.txt", "w")
    f09 = file("r09.txt", "w")
    f1 = file("r1.txt", "w")

    # write co-ordinates
    sep = ","
    for i in range(len(r015[0])):
        f015.write(str(r015[0][i])+sep+str(r015[1][i])+sep+str(r015[2][i])+"\n")
    for i in range(len(r02[0])):
        f02.write(str(r02[0][i])+sep+str(r02[1][i])+sep+str(r02[2][i])+"\n")
    for i in range(len(r025[0])):
        f025.write(str(r025[0][i])+sep+str(r025[1][i])+sep+str(r025[2][i])+"\n")
    for i in range(len(r03[0])):
        f03.write(str(r03[0][i])+sep+str(r03[1][i])+sep+str(r03[2][i])+"\n")
    for i in range(len(r04[0])):
        f04.write(str(r04[0][i])+sep+str(r04[1][i])+sep+str(r04[2][i])+"\n")
    for i in range(len(r05[0])):
        f05.write(str(r05[0][i])+sep+str(r05[1][i])+sep+str(r05[2][i])+"\n")
    for i in range(len(r06[0])):
        f06.write(str(r06[0][i])+sep+str(r06[1][i])+sep+str(r06[2][i])+"\n")
    for i in range(len(r07[0])):
        f07.write(str(r07[0][i])+sep+str(r07[1][i])+sep+str(r07[2][i])+"\n")
    for i in range(len(r08[0])):
        f08.write(str(r08[0][i])+sep+str(r08[1][i])+sep+str(r08[2][i])+"\n")
    for i in range(len(r09[0])):
        f09.write(str(r09[0][i])+sep+str(r09[1][i])+sep+str(r09[2][i])+"\n")
    for i in range(len(r1[0])):
        f1.write(str(r1[0][i])+sep+str(r1[1][i])+sep+str(r1[2][i])+"\n")

    # close files
    f015.close()
    f02.close()
    f025.close()
    f03.close()
    f04.close()
    f05.close()
    f06.close()
    f07.close()
    f08.close()
    f09.close()
    f1.close()

    return


def b4567s(rR):
    """
    Section generation function of the Wageningen B4-70

    Parameters
    ----------
    rR: number
        Normalised radial location, between 0 (root) and 1 (tip).

    Returns
    -------
    xfs:    list
        List of x co-ordinates.
    yfs:    list
        List of y co-ordinates.
    """

    # set trailing and leading edge thicknesses
    tte = 0.001
    tle = 0.001
    tmax = 1

    # create lists
    ps = [-1.0, -0.95, -0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.2, 0,
          0.2, 0.4, 0.5, 0.6, 0.7, 0.8, 0.85, 0.9, 0.95, 1]
    yts = []
    ybs = []

    # find ys in terms of ps
    for p in ps:
        if p < 0:
            ybs.append(bv1(rR, p)*(tmax-tte))
            yts.append((bv1(rR, p)+bv2(rR, p))*(tmax-tte)+tte)
        if p >= 0:
            ybs.append(bv1(rR, p)*(tmax-tle))
            yts.append((bv1(rR, p)+bv2(rR, p))*(tmax-tle)+tle)

    # find a/c and b/c
    a, b = b4567ab(rR)

    # scale ps in terms of xs
    xs = []
    for p in ps:
        if p < 0:
            xtemp = (1-b)*p
            xtemp = xtemp + (a - b)
        if p >= 0:
            xtemp = b*p
            xtemp = xtemp + (a - b)
        xs.append(xtemp)

    # add additional anti-peanut point at leading and trailing edges
    rte = 0.5*(yts[0] - ybs[0])
    yte = 0.5*(yts[0] + ybs[0])
    xte = xs[0] - rte
    rle = 0.5*(yts[-1] - ybs[-1])
    yle = 0.5*(yts[-1] + ybs[-1])
    xle = xs[-1] + rle

    # create and output single list of co-ordinates
    # from leading edge over top surface in anti-clockwise fashion
    xfs = [xle] + xs[::-1] + [xte] + xs + [xle]
    yfs = [yle] + yts[::-1] + [yte] + ybs + [yle]

    # remove duplicate points
    for i in range(len(xfs)-1):
        if xfs[i] == xfs[i+1] and yfs[i] == yfs[i+1]:
            print "Removing␣duplicate␣point␣at␣" + str(xfs.pop(i)),
            print ",␣" + str(yfs.pop(i)) + "␣at␣radius␣" + str(rR)
    return xfs, yfs


def b470c(rR, D):
    """
␣␣␣␣Returns␣the␣absolute␣chord␣length␣of␣a␣B4-70␣prop.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣rR:␣number
␣␣␣␣␣␣␣␣Normalised␣radial␣location,␣between␣0␣(root)␣and␣1␣(tip).
␣␣␣␣D:␣␣number
␣␣␣␣␣␣␣␣Diameter␣of␣B4-70␣propeller.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣c:␣␣number
␣␣␣␣␣␣␣␣Absolute␣chord␣length.
␣␣␣␣"""
    cZDA = b4567c(rR)
    return cZDA*D*0.7/4.0


def b4t(rR, D):
    """
␣␣␣␣Returns␣the␣absolute␣thickness␣of␣a␣4␣bladed␣B-Series␣propeller.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣rR:␣number
␣␣␣␣␣␣␣␣Normalised␣radial␣location,␣between␣0␣(root)␣and␣1␣(tip).
␣␣␣␣D:␣␣number
␣␣␣␣␣␣␣␣Diameter␣of␣B4␣propeller.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣t:␣␣number
␣␣␣␣␣␣␣␣Absolute␣thickness.
␣␣␣␣"""
    if rR < 0.2:
        t = 0.045 - (0.0084*rR/0.2)
    if rR >= 0.2 and rR < 0.3:
        t = 0.0366 - (0.0042*((rR-0.2)/0.1))
    if rR >= 0.3 and rR < 0.4:
        t = 0.0324 - (0.0042*((rR-0.3)/0.1))
    if rR >= 0.4 and rR < 0.5:
        t = 0.0282 - (0.0042*((rR-0.4)/0.1))
    if rR >= 0.5 and rR < 0.6:
        t = 0.024 - (0.0042*((rR-0.5)/0.1))
    if rR >= 0.6 and rR < 0.7:
        t = 0.0198 - (0.0042*((rR-0.6)/0.1))
    if rR >= 0.7 and rR < 0.8:
        t = 0.0156 - (0.0042*((rR-0.7)/0.1))
    if rR >= 0.8 and rR < 0.9:
        t = 0.0114 - (0.0042*((rR-0.8)/0.1))
    if rR >= 0.9:
        t = 0.0072 - (0.0062*((rR-0.9)/0.1))

    return t*D


def p10(rR):
    """
␣␣␣␣Returns␣the␣pitch␣angle␣in␣degrees␣for␣a␣square␣wheel␣prop␣(P/D␣=␣1.0).

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣rR:␣number
␣␣␣␣␣␣␣␣Normalised␣radial␣location,␣between␣0␣(root)␣and␣1␣(tip).

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣p:␣␣number
␣␣␣␣␣␣␣␣Pitch␣angle␣in␣degrees.
␣␣␣␣"""
```

```
    from math import atan, degrees, pi

    return degrees(atan(pi*rR/1.0))


def b4p10(rR):
    """
␣␣␣␣Returns␣the␣pitch␣angle␣in␣degrees␣for␣a␣square␣wheel␣B4␣series␣prop.

␣␣␣␣Parameters
␣␣␣␣−−−−−−−−−−
␣␣␣␣rR:␣number
␣␣␣␣␣␣␣␣Normalised␣radial␣location,␣between␣0␣(root)␣and␣1␣(tip).

␣␣␣␣Returns
␣␣␣␣−−−−−−−
␣␣␣␣p:␣␣number
␣␣␣␣␣␣␣␣Pitch␣angle␣in␣degrees.
␣␣␣␣"""
    from math import atan, degrees, pi

    if rR > 0.5:
        P = 1.0
    if rR <= 0.5:
        P = 0.7 + 0.6*rR

    return degrees(atan(pi*rR/P))


def bv1(rR, P):
    """
␣␣␣␣Returns␣V1␣for␣a␣B−Series␣propeller␣for␣a␣given␣r/R␣and␣P.
␣␣␣␣"""

    # create lookup table, first index is P, second is r/R
    v1tab = [[0,0,0.0522,0.1467,0.2306,0.2598,0.2826,0.3],          # P = -1.0
             [0,0,0.0420,0.12,0.204,0.2372,0.263,0.2824],           # P = -0.95
             [0,0,0.033,0.0972,0.179,0.2115,0.24,0.265],            # P = -0.9
             [0,0,0.019,0.063,0.1333,0.1651,0.1967,0.23],           # P = -0.8
             [0,0,0.01,0.0395,0.0943,0.1246,0.157,0.195],           # P = -0.7
             [0,0,0.004,0.0214,0.0623,0.0899,0.1207,0.1610],        # P = -0.6
             [0,0,0.0012,0.0116,0.0376,0.0579,0.088,0.128],         # P = -0.5
             [0,0,0,0.0044,0.0202,0.035,0.0592,0.0955],             # P = -0.4
             [0,0,0,0,0.0033,0.0084,0.0172,0.0365],                 # P = -0.2
             [0,0,0,0,0,0,0,0],                                     # P = 0
             [0,0,0,0,0.0027,0.0031,0.0049,0.0096],                 # P = 0.2
             [0,0,0,0.0033,0.0148,0.0224,0.0304,0.0384],            # P = 0.4
             [0,0,0.0008,0.009,0.03,0.0417,0.052,0.0615],           # P = 0.5
             [0,0,0.0034,0.0189,0.0503,0.0669,0.0804,0.092],        # P = 0.6
             [0,0,0.0085,0.0357,0.079,0.1008,0.118,0.132],          # P = 0.7
             [0,0.0006,0.0211,0.0637,0.1191,0.1465,0.1685,0.187],   # P = 0.8
             [0,0.0022,0.0328,0.0833,0.1445,0.1747,0.2,0.223],      # P = 0.85
             [0,0.0067,0.05,0.1088,0.176,0.2068,0.2353,0.2642],     # P = 0.9
             [0,0.0169,0.0778,0.1467,0.2186,0.2513,0.2821,0.315],   # P = 0.95
             [0,0.0382,0.1278,0.2181,0.2923,0.3256,0.356,0.386]]    # P = 1.0

    if P < -0.975:
        pindex = 0
    if P >= -0.975 and P < -0.925:
        pindex = 1
    if P >= -0.925 and P < -0.85:
        pindex = 2
    if P >= -0.85 and P < -0.75:
        pindex = 3
    if P >= -0.75 and P < -0.65:
        pindex = 4
    if P >= -0.65 and P < -0.55:
        pindex = 5
    if P >= -0.55 and P < -0.45:
        pindex = 6
    if P >= -0.45 and P < -0.3:
        pindex = 7
    if P >= -0.3 and P < -0.1:
        pindex = 8
    if P >= -0.1 and P < 0.1:
        pindex = 9
    if P >= 0.1 and P < 0.3:
        pindex = 10
    if P >= 0.3 and P < 0.45:
        pindex = 11
    if P >= 0.45 and P < 0.55:
        pindex = 12
    if P >= 0.55 and P < 0.65:
        pindex = 13
    if P >= 0.65 and P < 0.75:
        pindex = 14
    if P >= 0.75 and P < 0.825:
        pindex = 15
    if P >= 0.825 and P < 0.875:
        pindex = 16
    if P >= 0.875 and P < 0.925:
        pindex = 17
    if P >= 0.925 and P < 0.975:
        pindex = 18
    if P >= 0.975:
        pindex = 19

    if rR >= 0.65 and rR <= 1:
        rindex = 0
    if rR >= 0.55 and rR < 0.65:
        rindex = 1
    if rR >= 0.45 and rR < 0.55:
        rindex = 2
    if rR >= 0.35 and rR < 0.45:
        rindex = 3
    if rR >= 0.275 and rR < 0.35:
        rindex = 4
    if rR >= 0.225 and rR < 0.275:
        rindex = 5
    if rR >= 0.175 and rR < 0.225:
        rindex = 6
    if rR >= 0 and rR < 0.175:
        rindex = 7

    return v1tab[pindex][rindex]


def bv2(rR, P):
    """
    Returns V2 for a B-Series propeller for a given r/R and P
    """

    # create lookup table, first index is P, second is r/R
    v2tab = [[0,0,0,0,0,0,0,0,0,0,0],
                                                                  # P = -1.0
             [0.0975,0.0975,0.0975,0.0975,0.0965,0.0950,0.0905,0.08,0.0725,0.064,0.054],
                    # P = -0.95
             [0.19,0.19,0.19,0.19,0.1885,0.1865,0.181,0.167,0.1567,0.1455,0.1325],
                          # P = -0.9
             [0.36,0.36,0.36,0.36,0.3585,0.3569,0.35,0.336,0.3228,0.306,0.287],
                             # P = -0.8
             [0.51,0.51,0.51,0.51,0.511,0.514,0.504,0.4885,0.474,0.4535,0.428],
                             # P = -0.7
             [0.64,0.64,0.64,0.64,0.6415,0.6439,0.6353,0.6195,0.605,0.5842,0.5585],
                         # P = -0.6
             [0.75,0.75,0.75,0.75,0.753,0.758,0.7525,0.7335,0.7184,0.6995,0.677],
                          # P = -0.5
             [0.84,0.84,0.84,0.84,0.8426,0.8456,0.8415,0.8265,0.8139,0.7984,0.7805],
                        # P = -0.4
             [0.96,0.96,0.96,0.96,0.9613,0.9639,0.9645,0.9583,0.9519,0.9446,0.936],
                         # P = -0.2
             [1,1,1,1,1,1,1,1,1,1,1],
                                                                           # P = 0
             [0.96,0.9615,0.9635,0.9675,0.969,0.971,0.9725,0.975,0.9751,0.975,0.976],
                       # P = 0.2
             [0.84,0.845,0.852,0.866,0.879,0.888,0.8933,0.892,0.8899,0.8875,0.8825],
                        # P = 0.4
             [0.75,0.755,0.7635,0.785,0.809,0.8275,0.8345,0.8315,0.8259,0.817,0.8055],
                      # P = 0.5
             [0.64,0.6455,0.6545,0.6840,0.72,0.7478,0.7593,0.752,0.7415,0.7277,0.7105],
                     # P = 0.6
             [0.51,0.516,0.5265,0.5615,0.606,0.643,0.659,0.6505,0.6359,0.619,0.5995],
                       # P = 0.7
             [0.36,0.366,0.3765,0.414,0.462,0.5039,0.522,0.513,0.4982,0.4777,0.452],
                        # P = 0.8
             [0.2775,0.283,0.2925,0.33,0.3775,0.4135,0.4335,0.4265,0.4108,0.3905,0.3665],
                   # P = 0.85
             [0.19,0.195,0.2028,0.2337,0.272,0.3056,0.3235,0.3197,0.3042,0.284,0.26],
                       # P = 0.9
             [0.0975,0.1,0.105,0.124,0.1485,0.175,0.1935,0.189,0.1758,0.156,0.13],
                          # P = 0.95
             [0,0,0,0,0,0,0,0,0,0,0]]
                                                                           # P = 1.0

    if P < -0.975:
        pindex = 0
    if P >= -0.975 and P < -0.925:
        pindex = 1
    if P >= -0.925 and P < -0.85:
        pindex = 2
    if P >= -0.85 and P < -0.75:
        pindex = 3
    if P >= -0.75 and P < -0.65:
        pindex = 4
    if P >= -0.65 and P < -0.55:
        pindex = 5
    if P >= -0.55 and P < -0.45:
        pindex = 6
    if P >= -0.45 and P < -0.3:
        pindex = 7
    if P >= -0.3 and P < -0.1:
        pindex = 8
    if P >= -0.1 and P < 0.1:
        pindex = 9
    if P >= 0.1 and P < 0.3:
        pindex = 10
    if P >= 0.3 and P < 0.45:
        pindex = 11
    if P >= 0.45 and P < 0.55:
        pindex = 12
    if P >= 0.55 and P < 0.65:
        pindex = 13
    if P >= 0.65 and P < 0.75:
        pindex = 14
    if P >= 0.75 and P < 0.825:
        pindex = 15
    if P >= 0.825 and P < 0.875:
        pindex = 16
    if P >= 0.875 and P < 0.925:
        pindex = 17
    if P >= 0.925 and P < 0.975:
        pindex = 18
    if P >= 0.975:
        pindex = 19

    if rR >= 0.875 and rR <= 1:
        rindex = 0
    if rR >= 0.825 and rR < 0.875:
        rindex = 1
    if rR >= 0.75 and rR < 0.825:
        rindex = 2
    if rR >= 0.65 and rR < 0.75:
        rindex = 3
    if rR >= 0.55 and rR < 0.65:
        rindex = 4
    if rR >= 0.45 and rR < 0.55:
        rindex = 5
    if rR >= 0.35 and rR < 0.45:
        rindex = 6
    if rR >= 0.275 and rR < 0.35:
        rindex = 7
    if rR >= 0.225 and rR < 0.275:
        rindex = 8
    if rR >= 0.175 and rR < 0.225:
        rindex = 9
    if rR >= 0 and rR < 0.175:
        rindex = 10

    return v2tab[pindex][rindex]


def b4567ab(rR):
    """
    Returns a/c and b/c for a 4, 5, 6 or 7 bladed B-Series Propeller.
    If r/R is not exactly tabulated then the value is interpolated.
    """
    if rR < 0.2:
        a = 0.625 - (0.008*rR/0.2)
        b = 0.35
    if rR >= 0.2 and rR < 0.3:
        a = 0.617 - (0.004*((rR-0.2)/0.1))
        b = 0.35
    if rR >= 0.3 and rR < 0.4:
        a = 0.613 - (0.012*((rR-0.3)/0.1))
        b = 0.35 + (0.001*((rR-0.3)/0.1))
    if rR >= 0.4 and rR < 0.5:
        a = 0.601 - (0.015*((rR-0.4)/0.1))
        b = 0.351 + (0.004*((rR-0.4)/0.1))
    if rR >= 0.5 and rR < 0.6:
        a = 0.586 - (0.025*((rR-0.5)/0.1))
        b = 0.355 + (0.034*((rR-0.5)/0.1))
    if rR >= 0.6 and rR < 0.7:
        a = 0.561 - (0.037*((rR-0.6)/0.1))
        b = 0.389 + (0.054*((rR-0.6)/0.1))
    if rR >= 0.7 and rR < 0.8:
        a = 0.524 - (0.061*((rR-0.7)/0.1))
        b = 0.443 + (0.036*((rR-0.7)/0.1))
    if rR >= 0.8 and rR < 0.9:
        a = 0.463 - (0.112*((rR-0.8)/0.1))
        b = 0.479 + (0.021*((rR-0.8)/0.1))
    if rR >= 0.9:
        a = 0.351 - (0.35*((rR-0.9)/0.1))
        b = 0.5

    return a, b


def b4567c(rR):
    """
    Returns c/D.Z/EAR for a 4, 5, 6 or 7 bladed B-Series Propeller.
    If r/R is not exactly tabulated then the value is interpolated.
    """
    if rR < 0.2:
        c = 1.222 + (0.44*rR/0.2)
    if rR >= 0.2 and rR < 0.3:
        c = 1.662 + (0.22*((rR-0.2)/0.1))
    if rR >= 0.3 and rR < 0.4:
        c = 1.882 + (0.162*((rR-0.3)/0.1))
    if rR >= 0.4 and rR < 0.5:
        c = 2.050 + (0.102*((rR-0.4)/0.1))
    if rR >= 0.5 and rR < 0.6:
        c = 2.152 + (0.035*((rR-0.5)/0.1))
    if rR >= 0.6 and rR < 0.7:
        c = 2.187 - (0.043*((rR-0.6)/0.1))
    if rR >= 0.7 and rR < 0.8:
        c = 2.144 - (0.174*((rR-0.7)/0.1))
    if rR >= 0.8 and rR < 0.9:
        c = 1.970 - (0.388*((rR-0.8)/0.1))
    if rR >= 0.9:
        c = 1.582 - (1.482*((rR-0.9)/0.1))

    return c


def ogival(rR):
    """
␣␣␣␣Returns␣blade␣section␣data␣for␣an␣ogival␣section.
␣␣␣␣This␣is␣the␣same␣as␣the␣rR␣=␣1.0␣section␣of␣a␣B-series␣propeller,
␣␣␣␣however␣co-ordinates␣are␣shifted␣to␣be␣centred␣on␣the␣design␣line.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣rR:␣number
␣␣␣␣␣␣␣␣Normalised␣radial␣location,␣between␣0␣(root)␣and␣1␣(tip).

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣xs:␣list
␣␣␣␣␣␣␣␣X␣co-ordinates␣of␣ogival.
␣␣␣␣ys:␣list
␣␣␣␣␣␣␣␣Y␣co-ordinates␣of␣ogival.
␣␣␣␣"""

    return map(lambda x: x+0.5, b4567s(1.0)[0]), b4567s(1.0)[1]


def pitchvar(root, half, seven, tip):
    """
␣␣␣␣Creates␣a␣function␣to␣describe␣a␣variable␣pitch␣distribution,
␣␣␣␣with␣P/D␣ratios␣set␣at:
␣␣␣␣root␣(r/R␣=␣0),
␣␣␣␣half␣(r/R␣=␣0.5),
␣␣␣␣seven␣(r/R␣=␣0.7),
␣␣␣␣tip␣(r/R␣=␣1.0).

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣root:␣␣␣number
␣␣␣␣␣␣␣␣Pitch␣at␣root.
␣␣␣␣half:␣␣␣number
␣␣␣␣␣␣␣␣Pitch␣halfway␣up␣the␣blade.
␣␣␣␣seven:␣␣number
␣␣␣␣␣␣␣␣Pitch␣at␣70%␣section.
␣␣␣␣tip:␣␣␣␣number
␣␣␣␣␣␣␣␣Pitch␣at␣tip.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣pitchfunc:␣␣function
␣␣␣␣␣␣␣␣Function␣that␣calculates␣a␣pitch␣based␣on␣r/R␣parameter.
␣␣␣␣"""

    def pitchfunc(rR):
        from math import degrees, atan, pi
        if 0.0 <= rR <= 0.5:
            PDR = ((rR-0.0)/0.5)*(half-root)+root
        if 0.5 < rR <= 0.7:
            PDR = ((rR-0.5)/0.2)*(seven-half)+half
        if 0.7 < rR <= 1.0:
            PDR = ((rR-0.7)/0.3)*(tip-seven)+seven
        return degrees(atan(pi*rR/PDR))

    return pitchfunc


def pitchquad(root, half, tip):
    """
␣␣␣␣Creates␣a␣function␣to␣describe␣a␣quadratic␣pitch␣distribution,
␣␣␣␣passing␣through␣points␣at␣the␣root,␣half␣way␣and␣tip.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣root:␣␣␣number
␣␣␣␣␣␣␣␣Pitch␣at␣root.
␣␣␣␣half:␣␣␣number
␣␣␣␣␣␣␣␣Pitch␣halfway␣up␣the␣blade.
␣␣␣␣tip:␣␣␣␣number
␣␣␣␣␣␣␣␣Pitch␣at␣tip.
```

```
␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣pitchfunc:␣␣function
␣␣␣␣␣␣␣␣Function␣that␣calculates␣a␣pitch␣based␣on␣r/R␣parameter.
␣␣␣␣"""
    from numpy import polyfit
    pfit = polyfit([0.0, 0.5, 1.0], [root, half, tip], 2)

    def pitchfunc(rR):
        from math import degrees, atan, pi
        PDR = pfit[0]*(rR**2)+pfit[1]*(rR)+pfit[2]
        return degrees(atan(pi*rR/PDR))

    return pitchfunc


def constchord(cD):
    """
␣␣␣␣Creates␣a␣function␣to␣describe␣a␣constant␣chord␣distribution,
␣␣␣␣with␣input␣of␣cD,␣the␣chord/diameter␣ratio.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣cD:␣number
␣␣␣␣␣␣␣␣Chord/diameter␣ratio.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣chordfunc:␣␣function
␣␣␣␣␣␣␣␣Function␣that␣calculates␣absolute␣chord␣based␣of␣r/R␣and␣D.
␣␣␣␣"""

    def chordfunc(rR, D):
        return cD*D

    return chordfunc


def chordquad(root, half, tip):
    """
␣␣␣␣Creates␣a␣function␣to␣describe␣a␣quadratic␣chord␣distribution,
␣␣␣␣with␣an␣input␣of␣chord/diameter␣ratio␣at␣the␣root,␣half␣way␣and␣tip.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣root:␣␣␣number
␣␣␣␣␣␣␣␣Chord/diameter␣ratio␣at␣root.
␣␣␣␣half:␣␣␣number
␣␣␣␣␣␣␣␣Chord/diameter␣ratio␣halfway␣up␣the␣blade.
␣␣␣␣tip:␣␣␣␣number
␣␣␣␣␣␣␣␣Chord/diameter␣ratio␣at␣tip.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣chordfunc:␣␣function
␣␣␣␣␣␣␣␣Function␣that␣calculates␣absolute␣chord␣based␣of␣r/R␣and␣D.
␣␣␣␣"""
    from numpy import polyfit
    pfit = polyfit([0.0, 0.5, 1.0], [root, half, tip], 2)

    def chordfunc(rR, D):
        cD = pfit[0]*(rR**2)+pfit[1]*(rR)+pfit[2]
        return cD*D

    return chordfunc


def constthick(tD):
    """
␣␣␣␣Creates␣a␣function␣to␣describe␣a␣constant␣thickness␣distribution,
␣␣␣␣with␣input␣of␣tD␣the␣thickness/diameter␣ratio.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣tD:␣number
␣␣␣␣␣␣␣␣Thickness/diameter␣ratio.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣thickfunc:␣␣function
␣␣␣␣␣␣␣␣Function␣that␣calculates␣absolute␣thickness␣based␣of␣r/R␣and␣D.
␣␣␣␣"""

    def thickfunc(rR, D):
        return tD*D

    return thickfunc


def thicklinear(root, tip):
    """
    Creates a function to describe a linear thickness distribution,
    with input of the thickness/diameter ratio at the root and tip.

    Parameters
    ----------
    root:   number
        Thickness/diameter ratio at the root.
    tip:    number
        Thickness/diameter ratio at the tip.

    Returns
    -------
    thickfunc:  function
        Function that calculates absolute thickness based of r/R and D.
    """

    def thickfunc(rR, D):
        return (root+(tip-root)*rR)*D

    return thickfunc


def thickquad(root, half, tip):
    """
    Creates a function to describe a quadratic thickness distribution,
    with an input of thickness/diameter ratio at the root, half way and tip.

    Parameters
    ----------
    root:   number
        Thickness/diameter ratio at the root.
    half:   number
        Thickness/diameter ratio halfway along the blade.
    tip:    number
        Thickness/diameter ratio at the tip.

    Returns
    -------
    thickfunc:  function
        Function that calculates absolute thickness based of r/R and D.
    """
    from numpy import polyfit
    pfit = polyfit([0.0, 0.5, 1.0], [root, half, tip], 2)

    def thickfunc(rR, D):
        tD = pfit[0]*(rR**2)+pfit[1]*(rR)+pfit[2]
        return tD*D

    return thickfunc


def showb4567s(rR):
    """
    Uses pylab to plot a figure of the section
    of a B4-70 prop at a certain radial location.
    """
    import pylab
    xs, ys = b4567s(rR)
    pylab.plot(xs, ys)
    pylab.axis([-1, 1, -0.5, 1.5])
    pylab.show()
    return


def outputb4567csv(filename):
    """
    Outputs blade sections for a B4-70 prop at various radial stations
    into a .csv file that can be read into excel
    and used to generate sections in SolidWorks.
    """

    # create co-ordinate data
    r02x, r02y = b4567s(0.2)
    r02y = map(lambda x: x*100, r02y)
    r025x, r025y = b4567s(0.25)
    r025y = map(lambda x: x*100, r025y)
    r03x, r03y = b4567s(0.3)
    r03y = map(lambda x: x*100, r03y)
    r04x, r04y = b4567s(0.4)
    r04y = map(lambda x: x*100, r04y)
    r05x, r05y = b4567s(0.5)
    r05y = map(lambda x: x*100, r05y)
    r06x, r06y = b4567s(0.6)
    r06y = map(lambda x: x*100, r06y)
    r07x, r07y = b4567s(0.7)
    r07y = map(lambda x: x*100, r07y)
    r08x, r08y = b4567s(0.8)
    r08y = map(lambda x: x*100, r08y)
    r09x, r09y = b4567s(0.9)
    r09y = map(lambda x: x*100, r09y)
    r10x, r10y = b4567s(1.0)
    r10y = map(lambda x: x*100, r10y)
    # open file
    fout = file(filename+".csv", "w")

    # write header
    lens = len(r02x)
    fout.write(str(lens) + ",R0.2," + str(lens) + ",R0.25," + str(lens) +
               ",R0.3," + str(lens) + ",R0.4," + str(lens) + ",R0.5," +
               str(lens) + ",R0.6," + str(lens) + ",R0.7," + str(lens) +
               ",R0.8," + str(lens) + ",R0.9," + str(lens) + ",R1.0\n")

    # write remaining data
    for i in range(lens):
        fout.write(str(r02x[i])+","+str(r02y[i])+",")
        fout.write(str(r025x[i])+","+str(r025y[i])+",")
        fout.write(str(r03x[i])+","+str(r03y[i])+",")
        fout.write(str(r04x[i])+","+str(r04y[i])+",")
        fout.write(str(r05x[i])+","+str(r05y[i])+",")
        fout.write(str(r06x[i])+","+str(r06y[i])+",")
        fout.write(str(r07x[i])+","+str(r07y[i])+",")
        fout.write(str(r08x[i])+","+str(r08y[i])+",")
        fout.write(str(r09x[i])+","+str(r09y[i])+",")
        fout.write(str(r10x[i])+","+str(r10y[i])+"\n")

    # close output file
    fout.close()

    return


def rotatecw(xs, ys, angle):
    """
    Function to rotate a set of co-ordinates clockwise
    about the origin through an angle of angle degrees.
    """
    from math import sin, cos, atan2, radians

    # get number of co-ordinates
    lens = len(xs)

    # create empty output lists
    x2s = []
    y2s = []

    for i in range(lens):
        # get theta A
        tA = atan2(ys[i], xs[i])

        # calculate theta B
        tB = tA - radians(angle)

        # calculate r
        r = (xs[i]*xs[i] + ys[i]*ys[i])**0.5

        # calculate rotated co-ordinates
        x2s.append(r*cos(tB))
        y2s.append(r*sin(tB))

    return x2s, y2s


def map2cyl(xs, ys):
    """
    Takes a set of co-ordinates and maps them onto a cylindrical surface.
    Uses the first y value as the radius of the cylinder about the origin.
    """
    from math import sin, cos, pi

    # get global constants
    r = ys[0]
    lens = len(xs)

    # create empty output lists
    x2s = []
    y2s = []

    # check r
    if r == 0:
        print "Error: cannot operate map2cyl function with 0 radius"
        return None

    for i in range(lens):
        # calculate absolute angle (polar)
        t = 0.5*pi - xs[i]/float(r)

        # create new co-ordinates based on this angle
        x2s.append(r*cos(t))
        y2s.append(r*sin(t))

    return x2s, y2s
```

```
def outputb470stl(D):
    """
    Creates a .stl file of a B4-70 propeller with P/D ratio of 1.0,
    diameter of D and hub diameter of 0.2*D; saving this as b470.stl
    """

    import ajstl

    # get co-ordinates
    r015, r02, r025, r03, r04, r05, r06, r07, r08, r09, r1 =\
        bladegen(b4567s, b470c, b4t, b4p10, D, rake=15)

    # create an empty facetstring
    stlstr = ""

    # create the first blade
    stlstr += ajstl.oneface(r02[0], r02[1], r02[2])
    stlstr += ajstl.twoface(r02[0], r02[1], r02[2], r025[0], r025[1], r025[2])
    stlstr += ajstl.twoface(r025[0], r025[1], r025[2], r03[0], r03[1], r03[2])
    stlstr += ajstl.twoface(r03[0], r03[1], r03[2], r04[0], r04[1], r04[2])
    stlstr += ajstl.twoface(r04[0], r04[1], r04[2], r05[0], r05[1], r05[2])
    stlstr += ajstl.twoface(r05[0], r05[1], r05[2], r06[0], r06[1], r06[2])
    stlstr += ajstl.twoface(r06[0], r06[1], r06[2], r07[0], r07[1], r07[2])
    stlstr += ajstl.twoface(r07[0], r07[1], r07[2], r08[0], r08[1], r08[2])
    stlstr += ajstl.twoface(r08[0], r08[1], r08[2], r09[0], r09[1], r09[2])
    stlstr += ajstl.twoface(r09[0], r09[1], r09[2], r1[0], r1[1], r1[2])
    stlstr += ajstl.oneface(r1[0][::-1], r1[1][::-1], r1[2][::-1])

    # rotate blade by 90 degrees
    r015[0], r015[1] = rotatecw(r015[0], r015[1], 90)
    r02[0], r02[1] = rotatecw(r02[0], r02[1], 90)
    r025[0], r025[1] = rotatecw(r025[0], r025[1], 90)
    r03[0], r03[1] = rotatecw(r03[0], r03[1], 90)
    r04[0], r04[1] = rotatecw(r04[0], r04[1], 90)
    r05[0], r05[1] = rotatecw(r05[0], r05[1], 90)
    r06[0], r06[1] = rotatecw(r06[0], r06[1], 90)
    r07[0], r07[1] = rotatecw(r07[0], r07[1], 90)
    r08[0], r08[1] = rotatecw(r08[0], r08[1], 90)
    r09[0], r09[1] = rotatecw(r09[0], r09[1], 90)
    r1[0], r1[1] = rotatecw(r1[0], r1[1], 90)

    # create the second blade
    stlstr += ajstl.oneface(r02[0], r02[1], r02[2])
    stlstr += ajstl.twoface(r02[0], r02[1], r02[2], r025[0], r025[1], r025[2])
    stlstr += ajstl.twoface(r025[0], r025[1], r025[2], r03[0], r03[1], r03[2])
    stlstr += ajstl.twoface(r03[0], r03[1], r03[2], r04[0], r04[1], r04[2])
    stlstr += ajstl.twoface(r04[0], r04[1], r04[2], r05[0], r05[1], r05[2])
    stlstr += ajstl.twoface(r05[0], r05[1], r05[2], r06[0], r06[1], r06[2])
    stlstr += ajstl.twoface(r06[0], r06[1], r06[2], r07[0], r07[1], r07[2])
    stlstr += ajstl.twoface(r07[0], r07[1], r07[2], r08[0], r08[1], r08[2])
    stlstr += ajstl.twoface(r08[0], r08[1], r08[2], r09[0], r09[1], r09[2])
    stlstr += ajstl.twoface(r09[0], r09[1], r09[2], r1[0], r1[1], r1[2])
    stlstr += ajstl.oneface(r1[0][::-1], r1[1][::-1], r1[2][::-1])

    # rotate blade by 90 degrees
    r015[0], r015[1] = rotatecw(r015[0], r015[1], 90)
    r02[0], r02[1] = rotatecw(r02[0], r02[1], 90)
    r025[0], r025[1] = rotatecw(r025[0], r025[1], 90)
    r03[0], r03[1] = rotatecw(r03[0], r03[1], 90)
    r04[0], r04[1] = rotatecw(r04[0], r04[1], 90)
    r05[0], r05[1] = rotatecw(r05[0], r05[1], 90)
    r06[0], r06[1] = rotatecw(r06[0], r06[1], 90)
    r07[0], r07[1] = rotatecw(r07[0], r07[1], 90)
    r08[0], r08[1] = rotatecw(r08[0], r08[1], 90)
    r09[0], r09[1] = rotatecw(r09[0], r09[1], 90)
    r1[0], r1[1] = rotatecw(r1[0], r1[1], 90)

    # create the third blade
    stlstr += ajstl.oneface(r02[0], r02[1], r02[2])
    stlstr += ajstl.twoface(r02[0], r02[1], r02[2], r025[0], r025[1], r025[2])
    stlstr += ajstl.twoface(r025[0], r025[1], r025[2], r03[0], r03[1], r03[2])
    stlstr += ajstl.twoface(r03[0], r03[1], r03[2], r04[0], r04[1], r04[2])
    stlstr += ajstl.twoface(r04[0], r04[1], r04[2], r05[0], r05[1], r05[2])
    stlstr += ajstl.twoface(r05[0], r05[1], r05[2], r06[0], r06[1], r06[2])
    stlstr += ajstl.twoface(r06[0], r06[1], r06[2], r07[0], r07[1], r07[2])
    stlstr += ajstl.twoface(r07[0], r07[1], r07[2], r08[0], r08[1], r08[2])
    stlstr += ajstl.twoface(r08[0], r08[1], r08[2], r09[0], r09[1], r09[2])
    stlstr += ajstl.twoface(r09[0], r09[1], r09[2], r1[0], r1[1], r1[2])
    stlstr += ajstl.oneface(r1[0][::-1], r1[1][::-1], r1[2][::-1])

    # rotate blade by 90 degrees
    r015[0], r015[1] = rotatecw(r015[0], r015[1], 90)
    r02[0], r02[1] = rotatecw(r02[0], r02[1], 90)
    r025[0], r025[1] = rotatecw(r025[0], r025[1], 90)
    r03[0], r03[1] = rotatecw(r03[0], r03[1], 90)
    r04[0], r04[1] = rotatecw(r04[0], r04[1], 90)
    r05[0], r05[1] = rotatecw(r05[0], r05[1], 90)
    r06[0], r06[1] = rotatecw(r06[0], r06[1], 90)
    r07[0], r07[1] = rotatecw(r07[0], r07[1], 90)
    r08[0], r08[1] = rotatecw(r08[0], r08[1], 90)
    r09[0], r09[1] = rotatecw(r09[0], r09[1], 90)
    r1[0], r1[1] = rotatecw(r1[0], r1[1], 90)
    # create the fourth blade
    stlstr += ajstl.oneface(r02[0], r02[1], r02[2])
    stlstr += ajstl.twoface(r02[0], r02[1], r02[2], r025[0], r025[1], r025[2])
    stlstr += ajstl.twoface(r025[0], r025[1], r025[2], r03[0], r03[1], r03[2])
    stlstr += ajstl.twoface(r03[0], r03[1], r03[2], r04[0], r04[1], r04[2])
    stlstr += ajstl.twoface(r04[0], r04[1], r04[2], r05[0], r05[1], r05[2])
    stlstr += ajstl.twoface(r05[0], r05[1], r05[2], r06[0], r06[1], r06[2])
    stlstr += ajstl.twoface(r06[0], r06[1], r06[2], r07[0], r07[1], r07[2])
    stlstr += ajstl.twoface(r07[0], r07[1], r07[2], r08[0], r08[1], r08[2])
    stlstr += ajstl.twoface(r08[0], r08[1], r08[2], r09[0], r09[1], r09[2])
    stlstr += ajstl.twoface(r09[0], r09[1], r09[2], r1[0], r1[1], r1[2])
    stlstr += ajstl.oneface(r1[0][::-1], r1[1][::-1], r1[2][::-1])

    # find maxz and minz
    maxz = max([max(r02[2]), max(r025[2]), max(r03[2]), max(r04[2]),
                max(r05[2]), max(r06[2]), max(r07[2]), max(r08[2]),
                max(r09[2]), max(r1[2])])
    minz = min([min(r02[2]), min(r025[2]), min(r03[2]), min(r04[2]),
                min(r05[2]), min(r06[2]), min(r07[2]), min(r08[2]),
                min(r09[2]), min(r1[2])])
    hubhalflength = max([abs(maxz), abs(minz)])

    # create hub
    hubrad = 0.1*D
    tempx = [hubrad]
    tempy = [0]
    cylxs = [tempx[0]]
    cylys = [tempy[0]]
    cylz1s = [hubhalflength]
    cylz2s = [-hubhalflength]
    for i in range(360):
        tempx, tempy = rotatecw(tempx, tempy, -1)
        cylxs.append(tempx[0])
        cylys.append(tempy[0])
        cylz1s.append(hubhalflength)
        cylz2s.append(-hubhalflength)
    stlstr += ajstl.oneface(cylxs, cylys, cylz1s)
    stlstr += ajstl.twoface(cylxs[:], cylys[:], cylz1s[:],
                            cylxs[:], cylys[:], cylz2s[:])
    stlstr += ajstl.oneface(cylxs[::-1], cylys[::-1], cylz2s[::-1])

    # write the stl file
    ajstl.writestl("b470", stlstr)


def outputnbladestl(section, chord, thickness, pitch, D, rake, n):
    """
␣␣␣␣Creates␣a␣file␣blades.stl␣containing␣the␣geometry␣information
␣␣␣␣for␣an␣n-bladed␣propeller␣as␣made␣by␣bladegen.
␣␣␣␣"""

    import ajstl

    # get co-ordinates
    r015, r02, r025, r03, r04, r05, r06, r07, r08, r09, r1 =\
        bladegen(section, chord, thickness, pitch, D, rake)

    # create an empty facetstring
    stlstr = ""

    # calculate number of degrees to rotate
    rotdeg = 360.0/n

    # create n blades
    for i in range(n):
        # add blade to stl string
        stlstr += ajstl.oneface(r015[0], r015[1], r015[2])
        stlstr += ajstl.twoface(r015[0], r015[1], r015[2],
                                r02[0], r02[1], r02[2])
        stlstr += ajstl.twoface(r02[0], r02[1], r02[2],
                                r025[0], r025[1], r025[2])
        stlstr += ajstl.twoface(r025[0], r025[1], r025[2],
                                r03[0], r03[1], r03[2])
        stlstr += ajstl.twoface(r03[0], r03[1], r03[2], r04[0], r04[1], r04[2])
        stlstr += ajstl.twoface(r04[0], r04[1], r04[2], r05[0], r05[1], r05[2])
        stlstr += ajstl.twoface(r05[0], r05[1], r05[2], r06[0], r06[1], r06[2])
        stlstr += ajstl.twoface(r06[0], r06[1], r06[2], r07[0], r07[1], r07[2])
        stlstr += ajstl.twoface(r07[0], r07[1], r07[2], r08[0], r08[1], r08[2])
        stlstr += ajstl.twoface(r08[0], r08[1], r08[2], r09[0], r09[1], r09[2])
        stlstr += ajstl.twoface(r09[0], r09[1], r09[2], r1[0], r1[1], r1[2])
        stlstr += ajstl.oneface(r1[0][::-1], r1[1][::-1], r1[2][::-1])

        # rotate blade
        r015[0], r015[1] = rotatecw(r015[0], r015[1], rotdeg)
        r02[0], r02[1] = rotatecw(r02[0], r02[1], rotdeg)
        r025[0], r025[1] = rotatecw(r025[0], r025[1], rotdeg)
        r03[0], r03[1] = rotatecw(r03[0], r03[1], rotdeg)
        r04[0], r04[1] = rotatecw(r04[0], r04[1], rotdeg)
        r05[0], r05[1] = rotatecw(r05[0], r05[1], rotdeg)
        r06[0], r06[1] = rotatecw(r06[0], r06[1], rotdeg)
        r07[0], r07[1] = rotatecw(r07[0], r07[1], rotdeg)
        r08[0], r08[1] = rotatecw(r08[0], r08[1], rotdeg)
        r09[0], r09[1] = rotatecw(r09[0], r09[1], rotdeg)
        r1[0], r1[1] = rotatecw(r1[0], r1[1], rotdeg)

    # write the stl file
    ajstl.writestl("blades", stlstr)
```

Appendix B

Stereolithographic Format Generation and Writing Program

This is a collection of Python functions written to take co-ordinate data and automatically convert the data into stereolithographic (.stl file) format such that it can be read as geometry by snappyHexMesh.

```
”””
ajstl.py

A␣collection␣of␣functions␣for␣the␣writing␣of␣stereolithographic
geometry␣files.␣␣Output␣is␣in␣ASCII␣format␣.stl␣files.

(c)␣copyright␣Aleksander␣Dubas␣2012−2013
”””

# import relevant functions from numpy
from numpy import append, array, cross, radians
from numpy import sin, cos, arange, ones, sqrt, linspace


def writestl(name, facetstring):
    ”””
␣␣␣␣Writes␣a␣solid,␣file␣is␣named␣with␣name.stl,
␣␣␣␣solid␣is␣named␣with␣name.␣and␣defined␣by␣facetstring.

␣␣␣␣Parameters
␣␣␣␣−−−−−−−−−−
␣␣␣␣name:␣␣␣string
␣␣␣␣␣␣␣␣Name␣of␣the␣solid,␣to␣be␣saved␣as␣<name.stl>
␣␣␣␣facetstring:␣␣␣␣string
␣␣␣␣␣␣␣␣String␣detailing␣the␣facets␣to␣make␣the␣stl␣with.

␣␣␣␣Returns
␣␣␣␣−−−−−−−
␣␣␣␣None

␣␣␣␣Example
␣␣␣␣−−−−−−−
␣␣␣␣>>>␣writestl(”mycube”,cube([0,␣0,␣0],␣1))
␣␣␣␣”””
    from os import fsync
    fout = open(name+”.stl”, 'w')
    fout.write(”solid␣”+name+”\n”)
    fout.write(facetstring)
    fout.write(”endsolid␣”+name+”\n”)
    fout.flush()
    fsync(fout.fileno())
    fout.close()
    return None


def cube(centre, size):
    ”””
␣␣␣␣Takes␣input␣of␣a␣centre␣(iterable␣of␣length␣3)
␣␣␣␣and␣side␣length␣size␣(float)
␣␣␣␣and␣outputs␣a␣string␣detailing␣the␣triangles
␣␣␣␣that␣make␣up␣the␣cube,␣ready␣for␣input␣into␣.stl␣file.
```

```
    Parameters
    ----------
    centre: iterable
        A length 3 iterable giving the 3D co-ordinate of the cube centre.
    size:   number
        A number giving how long the sides of the cube should be.

    Returns
    -------
    outstr: string
        A 'facetstring' detailing the triangles for the cube.

    Example
    -------
    >>> cube([0, 0, 0], 1)
    """

    # create output string
    outstr = ""

    # design co-ordinates
    hs = 0.5*size

    # Cube looks like this:
    #    G---H
    #   /|  /|
    # C---D  |
    # |  | | |      y   z
    # |  E-|-F      |  /
    # |/   |/       |/
    # A---B         o--x
    Ax = centre[0]-hs
    Ay = centre[1]-hs
    Az = centre[2]-hs

    Bx = centre[0]+hs
    By = centre[1]-hs
    Bz = centre[2]-hs

    Cx = centre[0]-hs
    Cy = centre[1]+hs
    Cz = centre[2]-hs

    Dx = centre[0]+hs
    Dy = centre[1]+hs
    Dz = centre[2]-hs

    Ex = centre[0]-hs
    Ey = centre[1]-hs
    Ez = centre[2]+hs

    Fx = centre[0]+hs
    Fy = centre[1]-hs
    Fz = centre[2]+hs

    Gx = centre[0]-hs
    Gy = centre[1]+hs
    Gz = centre[2]+hs

    Hx = centre[0]+hs
    Hy = centre[1]+hs
    Hz = centre[2]+hs

    # create triangles
    # face 1 ABCD => ABC BCD; normal = 0 0 -1
    outstr += "facet normal 0 0 -1\n"
    outstr += "outer loop\n"
    outstr += "vertex "+str(Ax)+" "+str(Ay)+" "+str(Az)+"\n"
    outstr += "vertex "+str(Bx)+" "+str(By)+" "+str(Bz)+"\n"
    outstr += "vertex "+str(Cx)+" "+str(Cy)+" "+str(Cz)+"\n"
    outstr += "endloop\n"
    outstr += "endfacet\n"

    outstr += "facet normal 0 0 -1\n"
    outstr += "outer loop\n"
    outstr += "vertex "+str(Bx)+" "+str(By)+" "+str(Bz)+"\n"
    outstr += "vertex "+str(Cx)+" "+str(Cy)+" "+str(Cz)+"\n"
    outstr += "vertex "+str(Dx)+" "+str(Dy)+" "+str(Dz)+"\n"
    outstr += "endloop\n"
    outstr += "endfacet\n"

    # face 2 ABEF => ABE BEF; normal = 0 -1 0
    outstr += "facet normal 0 -1 0\n"
    outstr += "outer loop\n"
    outstr += "vertex "+str(Ax)+" "+str(Ay)+" "+str(Az)+"\n"
    outstr += "vertex "+str(Bx)+" "+str(By)+" "+str(Bz)+"\n"
    outstr += "vertex "+str(Ex)+" "+str(Ey)+" "+str(Ez)+"\n"
    outstr += "endloop\n"
    outstr += "endfacet\n"

    outstr += "facet normal 0 -1 0\n"
    outstr += "outer loop\n"
    outstr += "vertex␣"+str(Bx)+"␣"+str(By)+"␣"+str(Bz)+"\n"
    outstr += "vertex␣"+str(Ex)+"␣"+str(Ey)+"␣"+str(Ez)+"\n"
    outstr += "vertex␣"+str(Fx)+"␣"+str(Fy)+"␣"+str(Fz)+"\n"
    outstr += "endloop\n"
    outstr += "endfacet\n"

    # face 3 BDFH => BDF DFH; normal = 1 0 0
    outstr += "facet␣normal␣1␣0␣0\n"
    outstr += "outer␣loop\n"
    outstr += "vertex␣"+str(Bx)+"␣"+str(By)+"␣"+str(Bz)+"\n"
    outstr += "vertex␣"+str(Dx)+"␣"+str(Dy)+"␣"+str(Dz)+"\n"
    outstr += "vertex␣"+str(Fx)+"␣"+str(Fy)+"␣"+str(Fz)+"\n"
    outstr += "endloop\n"
    outstr += "endfacet\n"

    outstr += "facet␣normal␣1␣0␣0\n"
    outstr += "outer␣loop\n"
    outstr += "vertex␣"+str(Dx)+"␣"+str(Dy)+"␣"+str(Dz)+"\n"
    outstr += "vertex␣"+str(Fx)+"␣"+str(Fy)+"␣"+str(Fz)+"\n"
    outstr += "vertex␣"+str(Hx)+"␣"+str(Hy)+"␣"+str(Hz)+"\n"
    outstr += "endloop\n"
    outstr += "endfacet\n"

    # face 4 CDGH => CDG DGH; normal = 0 1 0
    outstr += "facet␣normal␣0␣1␣0\n"
    outstr += "outer␣loop\n"
    outstr += "vertex␣"+str(Cx)+"␣"+str(Cy)+"␣"+str(Cz)+"\n"
    outstr += "vertex␣"+str(Dx)+"␣"+str(Dy)+"␣"+str(Dz)+"\n"
    outstr += "vertex␣"+str(Gx)+"␣"+str(Gy)+"␣"+str(Gz)+"\n"
    outstr += "endloop\n"
    outstr += "endfacet\n"

    outstr += "facet␣normal␣0␣1␣0\n"
    outstr += "outer␣loop\n"
    outstr += "vertex␣"+str(Dx)+"␣"+str(Dy)+"␣"+str(Dz)+"\n"
    outstr += "vertex␣"+str(Gx)+"␣"+str(Gy)+"␣"+str(Gz)+"\n"
    outstr += "vertex␣"+str(Hx)+"␣"+str(Hy)+"␣"+str(Hz)+"\n"
    outstr += "endloop\n"
    outstr += "endfacet\n"

    # face 5 ACEG => ACE CEG; normal = -1 0 0
    outstr += "facet␣normal␣-1␣0␣0\n"
    outstr += "outer␣loop\n"
    outstr += "vertex␣"+str(Ax)+"␣"+str(Ay)+"␣"+str(Az)+"\n"
    outstr += "vertex␣"+str(Cx)+"␣"+str(Cy)+"␣"+str(Cz)+"\n"
    outstr += "vertex␣"+str(Ex)+"␣"+str(Ey)+"␣"+str(Ez)+"\n"
    outstr += "endloop\n"
    outstr += "endfacet\n"

    outstr += "facet␣normal␣-1␣0␣0\n"
    outstr += "outer␣loop\n"
    outstr += "vertex␣"+str(Cx)+"␣"+str(Cy)+"␣"+str(Cz)+"\n"
    outstr += "vertex␣"+str(Ex)+"␣"+str(Ey)+"␣"+str(Ez)+"\n"
    outstr += "vertex␣"+str(Gx)+"␣"+str(Gy)+"␣"+str(Gz)+"\n"
    outstr += "endloop\n"
    outstr += "endfacet\n"

    # face 6 EFGH => EFG FGH; normal = 0 0 1
    outstr += "facet␣normal␣0␣0␣1\n"
    outstr += "outer␣loop\n"
    outstr += "vertex␣"+str(Ex)+"␣"+str(Ey)+"␣"+str(Ez)+"\n"
    outstr += "vertex␣"+str(Fx)+"␣"+str(Fy)+"␣"+str(Fz)+"\n"
    outstr += "vertex␣"+str(Gx)+"␣"+str(Gy)+"␣"+str(Gz)+"\n"
    outstr += "endloop\n"
    outstr += "endfacet\n"

    outstr += "facet␣normal␣0␣0␣1\n"
    outstr += "outer␣loop\n"
    outstr += "vertex␣"+str(Fx)+"␣"+str(Fy)+"␣"+str(Fz)+"\n"
    outstr += "vertex␣"+str(Gx)+"␣"+str(Gy)+"␣"+str(Gz)+"\n"
    outstr += "vertex␣"+str(Hx)+"␣"+str(Hy)+"␣"+str(Hz)+"\n"
    outstr += "endloop\n"
    outstr += "endfacet\n"

    return outstr


def oneface(xs, ys, zs):
    """
␣␣␣␣Returns␣a␣facetstring␣for␣a␣face␣defined␣by␣a␣series␣of␣points
␣␣␣␣with␣the␣normal␣defined␣in␣a␣right-hand␣rule␣manner.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣xs:␣list␣of␣numbers
␣␣␣␣␣␣␣␣A␣list␣of␣number␣giving␣the␣x␣co-ordinates␣of␣the␣face.
␣␣␣␣ys:␣list␣of␣numbers
␣␣␣␣␣␣␣␣A␣list␣of␣number␣giving␣the␣y␣co-ordinates␣of␣the␣face.
␣␣␣␣zs:␣list␣of␣numbers
␣␣␣␣␣␣␣␣A␣list␣of␣number␣giving␣the␣z␣co-ordinates␣of␣the␣face.

␣␣␣␣Returns
```

```
␣␣␣␣−−−−−−−
␣␣␣␣outstr:␣string
␣␣␣␣␣␣␣␣The␣'facetstring'␣detailing␣the␣triangles␣for␣the␣face.

␣␣␣␣Example
␣␣␣␣−−−−−−−
␣␣␣␣>>>␣oneface([0,␣1,␣1,␣0],␣[0,␣0,␣1,␣1],␣[0,␣0,␣0,␣0])
␣␣␣␣"""
    # sanity checks
    if len(xs) != len(ys) or len(ys) != len(zs):
        print("Error␣in␣function␣oneface:" +
              "␣too␣many␣of␣one␣co−ordinate␣in␣input!")
        return ""

    # check whether series of points is closed, and open if it is
    if xs[0] == xs[−1] and ys[0] == ys[−1] and zs[0] == zs[−1]:
        xs = xs[:−1]
        ys = ys[:−1]
        zs = zs[:−1]

    # define top and bottom co−ordinates
    xtops = xs[:len(xs)/2]
    xbots = xs[len(xs)/2:][::−1]
    ytops = ys[:len(xs)/2]
    ybots = ys[len(xs)/2:][::−1]
    ztops = zs[:len(xs)/2]
    zbots = zs[len(xs)/2:][::−1]

    # define array end reference integer
    xtl = len(xtops) − 1

    # make output string
    outstr = ""

    for i in range(xtl):
        # Make square ABCD:
        # B−−−A <− tops
        # |2\1|
        # D−−−C <− bots
        A = array([xtops[i], ytops[i], ztops[i]])
        B = array([xtops[i+1], ytops[i+1], ztops[i+1]])
        C = array([xbots[i], ybots[i], zbots[i]])
        D = array([xbots[i+1], ybots[i+1], zbots[i+1]])

        # Find normal of triangle 1
        n1 = cross(B−A, C−A)
        # Find normal of triangle 2
        n2 = cross(B−C, D−C)

        # Write triangles to outstr
        outstr += "facet␣normal␣"+str(n1[0])+"␣"+str(n1[1])+"␣"+str(n1[2])+"\n"
        outstr += "outer␣loop\n"
        outstr += "vertex␣"+str(A[0])+"␣"+str(A[1])+"␣"+str(A[2])+"\n"
        outstr += "vertex␣"+str(B[0])+"␣"+str(B[1])+"␣"+str(B[2])+"\n"
        outstr += "vertex␣"+str(C[0])+"␣"+str(C[1])+"␣"+str(C[2])+"\n"
        outstr += "endloop\n"
        outstr += "endfacet\n"

        outstr += "facet␣normal␣"+str(n2[0])+"␣"+str(n2[1])+"␣"+str(n2[2])+"\n"
        outstr += "outer␣loop\n"
        outstr += "vertex␣"+str(C[0])+"␣"+str(C[1])+"␣"+str(C[2])+"\n"
        outstr += "vertex␣"+str(B[0])+"␣"+str(B[1])+"␣"+str(B[2])+"\n"
        outstr += "vertex␣"+str(D[0])+"␣"+str(D[1])+"␣"+str(D[2])+"\n"
        outstr += "endloop\n"
        outstr += "endfacet\n"

    # special case if len(xbots)>len(xtops) i.e. odd number of co−ords
    if len(xbots) > len(xtops):
        # Make last triangle ABC
        #   A   <− tops
        #  / \
        # B−−−C <− bots
        A = array([xtops[xtl], ytops[xtl], ztops[xtl]])
        B = array([xbots[xtl+1], ybots[xtl+1], zbots[xtl+1]])
        C = array([xbots[xtl], ybots[xtl], zbots[xtl]])

        # Find normal
        n1 = cross(B−A, C−A)

        # Add final triangle to outstr
        outstr += "facet␣normal␣"+str(n1[0])+"␣"+str(n1[1])+"␣"+str(n1[2])+"\n"
        outstr += "outer␣loop\n"
        outstr += "vertex␣"+str(A[0])+"␣"+str(A[1])+"␣"+str(A[2])+"\n"
        outstr += "vertex␣"+str(B[0])+"␣"+str(B[1])+"␣"+str(B[2])+"\n"
        outstr += "vertex␣"+str(C[0])+"␣"+str(C[1])+"␣"+str(C[2])+"\n"
        outstr += "endloop\n"
        outstr += "endfacet\n"

    return outstr


def twoface(f1xs, f1ys, f1zs, f2xs, f2ys, f2zs):
        """
␣␣␣␣Returns␣a␣facetstring␣for␣a␣surface␣lofted␣between␣two␣faces␣defined␣by
    ␣␣␣␣two␣equal␣length␣series␣of␣points␣with␣the␣normal␣defined
    ␣␣␣␣in␣a␣right−hand␣rule␣manner.
    ␣␣␣␣For␣the␣facet␣normals␣to␣point␣outwards␣from␣the␣lofted␣surface,
    ␣␣␣␣both␣faces␣f1␣and␣f2␣should␣be␣written␣in␣an␣anti−clockwise␣order
    ␣␣␣␣when␣viewed␣down␣the␣axis␣from␣f1␣to␣f2.

    ␣␣␣␣Parameters
    ␣␣␣␣−−−−−−−−−−
    ␣␣␣␣f1xs:␣␣␣list␣of␣numbers
        ␣␣␣␣␣␣␣␣A␣list␣of␣x␣co−ordinates␣for␣the␣first␣face.
    ␣␣␣␣f1ys:␣␣␣list␣of␣numbers
        ␣␣␣␣␣␣␣␣A␣list␣of␣y␣co−ordinates␣for␣the␣first␣face.
    ␣␣␣␣f1zs:␣␣␣list␣of␣numbers
        ␣␣␣␣␣␣␣␣A␣list␣of␣z␣co−ordinates␣for␣the␣first␣face.
    ␣␣␣␣f2xs:␣␣␣list␣of␣numbers
        ␣␣␣␣␣␣␣␣A␣list␣of␣x␣co−ordinates␣for␣the␣second␣face.
    ␣␣␣␣f2ys:␣␣␣list␣of␣numbers
        ␣␣␣␣␣␣␣␣A␣list␣of␣y␣co−ordinates␣for␣the␣second␣face.
    ␣␣␣␣f2zs:␣␣␣list␣of␣numbers
        ␣␣␣␣␣␣␣␣A␣list␣of␣z␣co−ordinates␣for␣the␣second␣face.

        ␣␣␣␣Returns
        ␣␣␣␣−−−−−−−
    ␣␣␣␣outstr:␣string
        ␣␣␣␣␣␣␣␣A␣'facetstring'␣detailing␣the␣triangles
        ␣␣␣␣␣␣␣␣that␣loft␣between␣the␣two␣faces.
        ␣␣␣␣"""

    # sanity checks
    if len(f1xs) != len(f1ys) or len(f1ys) != len(f1zs) \
       or len(f1zs) != len(f2xs) or len(f2xs) != len(f2ys) \
       or len(f2ys) != len(f2zs):
        print("Error␣in␣function␣twoface:" +
              "too␣many␣of␣one␣co−ordinate␣in␣input!")
        print(str(len(f1xs))+"␣"+str(len(f1ys))+"␣"+str(len(f1zs))+"␣" +
              str(len(f2xs))+"␣"+str(len(f2ys))+"␣"+str(len(f2zs)))
        return ""

    # check whether series of points is open, and close if it is
    if f1xs[0] != f1xs[−1] or f1ys[0] != f1ys[−1] or f1zs[0] != f1zs[−1] \
       or f2xs[0] != f2xs[−1] or f2ys[0] != f2ys[−1] or f2zs[0] != f2zs[−1]:
        try:
            f1xs.append(f1xs[0])
            f1ys.append(f1ys[0])
            f1zs.append(f1zs[0])
            f2xs.append(f2xs[0])
            f2ys.append(f2ys[0])
            f2zs.append(f2zs[0])
        except AttributeError:
            f1xs = append(f1xs, f1xs[0])
            f1ys = append(f1ys, f1ys[0])
            f1zs = append(f1zs, f1zs[0])
            f2xs = append(f2xs, f2xs[0])
            f2ys = append(f2ys, f2ys[0])
            f2zs = append(f2zs, f2zs[0])

    # define top and bottom co−ordinates
    xtops = f2xs[:]
    xbots = f1xs[:]
    ytops = f2ys[:]
    ybots = f1ys[:]
    ztops = f2zs[:]
    zbots = f1zs[:]

    # define array end reference integer
    xtl = len(xtops) − 1

    # make output string
    outstr = ""

    for i in range(xtl):
        # Make square ABCD:
        # B−−−A <− tops
        # |2\1|
        # D−−−C <− bots
        A = array([xtops[i], ytops[i], ztops[i]])
        B = array([xtops[i+1], ytops[i+1], ztops[i+1]])
        C = array([xbots[i], ybots[i], zbots[i]])
        D = array([xbots[i+1], ybots[i+1], zbots[i+1]])

        # Find normal of triangle 1
        n1 = cross(B−A, C−A)
        # Find normal of triangle 2
        n2 = cross(B−C, D−C)

        # Write triangles to outstr
        outstr += "facet␣normal␣"+str(n1[0])+"␣"+str(n1[1])+"␣"+str(n1[2])+"\n"
        outstr += "outer␣loop\n"
        outstr += "vertex␣"+str(A[0])+"␣"+str(A[1])+"␣"+str(A[2])+"\n"
        outstr += "vertex␣"+str(B[0])+"␣"+str(B[1])+"␣"+str(B[2])+"\n"
        outstr += "vertex "+str(C[0])+" "+str(C[1])+" "+str(C[2])+"\n"
        outstr += "endloop\n"
        outstr += "endfacet\n"

        outstr += "facet normal "+str(n2[0])+" "+str(n2[1])+" "+str(n2[2])+"\n"
        outstr += "outer loop\n"
        outstr += "vertex "+str(C[0])+" "+str(C[1])+" "+str(C[2])+"\n"
        outstr += "vertex "+str(B[0])+" "+str(B[1])+" "+str(B[2])+"\n"
        outstr += "vertex "+str(D[0])+" "+str(D[1])+" "+str(D[2])+"\n"
        outstr += "endloop\n"
        outstr += "endfacet\n"

    return outstr


def hubZc(D, L):
    """
    Writes a facetstring for a propeller hub of diameter D and length L,
    centered at the origin, and axially oriented in the 'z' direction.
    Hub has flat ends -- thus a plain cylinder.

    Parameters
    ----------
    D:  number
        Diameter of the hub.
    L:  number
        Length of the hub.

    Returns
    -------
    outstr: string
        A 'facetstring' detailing the triangles
        that make up the hub surface.
    """

    # make output string
    outstr = ""

    # create central cylinder co-ordinates
    r = 0.5*D
    cylxs = r*cos(radians(arange(361)))
    cylys = r*sin(radians(arange(361)))
    cylz1s = 0.5*L*ones(361)
    cylz2s = -0.5*L*ones(361)

    # create cylinder
    outstr += oneface(cylxs[:], cylys[:], cylz1s[:])
    outstr += twoface(cylxs[:], cylys[:], cylz1s[:],
                      cylxs[:], cylys[:], cylz2s[:])
    outstr += oneface(cylxs[::-1], cylys[::-1], cylz2s[::-1])

    return outstr


def hubZs(D, L):
    """
    Writes a facetstring for a propeller hub of diameter D and length L,
    centered at the origin, and axially oriented in the 'z' direction.
    Hub has spherical ends.

    Parameters
    ----------
    D:  number
        Diameter of the hub.
    L:  number
        Length of the hub.

    Returns
    -------
    outstr: string
        A 'facetstring' detailing the triangles
        that make up the hub surface.
    """

    # make output string
    outstr = ""

    # create central cylinder co-ordinates
    r = 0.5*D
    cylxs = r*cos(radians(arange(361)))
    cylys = r*sin(radians(arange(361)))
    cylz1s = 0.5*L*ones(361)
    cylz2s = -0.5*L*ones(361)

    # create hub with spherical ends
    outstr += oneface(cos(radians(89))*cylxs[:],
                      cos(radians(89))*cylys[:],
                      cylz1s[:]+r*sin(radians(89)))
    for i in range(89, 0, -1):
        outstr += twoface(cos(radians(i))*cylxs[:],
                          cos(radians(i))*cylys[:],
                          cylz1s[:]+r*sin(radians(i)),
                              cos(radians(i-1))*cylxs[:],
                              cos(radians(i-1))*cylys[:],
                              cylz1s[:]+r*sin(radians(i-1)))
    outstr += twoface(cylxs[:], cylys[:], cylz1s[:],
                      cylxs[:], cylys[:], cylz2s[:])
    for i in range(0, 89, 1):
        outstr += twoface(cos(radians(i))*cylxs[:],
                          cos(radians(i))*cylys[:],
                          cylz2s[:]-r*sin(radians(i)),
                          cos(radians(i+1))*cylxs[:],
                          cos(radians(i+1))*cylys[:],
                          cylz2s[:]-r*sin(radians(i+1)))
    outstr += oneface(cos(radians(89))*cylxs[:],
                      cos(radians(89))*cylys[:],
                      cylz2s[:]-r*sin(radians(89)))

    return outstr


def rimZ(iD, oD, L):
    """
    Creates a rim or annulus about the origin in the Z direction
    with inner diameter iD, outer diameter oD and length L.
    See also rimZflat and rimZcut.

    Parameters
    ----------
    iD: number
        Inner diameter of the rim.
    oD: number
        Outer diameter of the rim.
    L:  number
        Length of the rim excluding end profiling.

    Returns
    -------
    outstr: string
        A 'facetstring' detailing the triangles
        that make up the rim surface.
    """

    # make output string
    outstr = ""

    # create co-ordinate lists
    rs = []
    zs = []

    # calculate some useful variables
    curvrad = 0.25*(oD - iD)
    curvcen = 0.25*(oD + iD)
    zcoord = 0.5*L

    # create cylinder co-ordinates
    rs.append(0.5*oD)
    zs.append(-zcoord)
    rs.append(0.5*oD)
    zs.append(zcoord)
    for i in range(1, 180):
        rs.append(curvcen+(curvrad*cos(radians(i))))
        zs.append(zcoord+(curvrad*sin(radians(i))))
    rs.append(0.5*iD)
    zs.append(zcoord)
    rs.append(0.5*iD)
    zs.append(-zcoord)
    for i in range(1, 180):
        rs.append(curvcen-(curvrad*cos(radians(i))))
        zs.append(-zcoord-(curvrad*sin(radians(i))))
    rs.append(0.5*oD)
    zs.append(-zcoord)

    # use revolveZ to create output string
    outstr += revolveZ(rs, zs)

    return outstr


def rimZflat(iD, oD, L):
    """
    Creates a rim or annulus about the origin in the Z direction
    with inner diameter iD, outer diameter oD and length L.
    Resulting rim is flat sided rather than having semicircular end profiles.

    Parameters
    ----------
    iD: number
        Inner diameter of the rim.
    oD: number
        Outer diameter of the rim.
    L:  number
        Length of the rim.
```

```
␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣outstr:␣string
␣␣␣␣␣␣␣␣A␣'facetstring'␣detailing␣the␣triangles
␣␣␣␣␣␣␣␣that␣make␣up␣the␣rim␣surface.
␣␣␣␣"""

    # make output string
    outstr = ""

    # create co-ordinate lists
    rs = []
    zs = []

    # calculate some useful variables
    zcoord = 0.5*L

    # create cylinder co-ordinates
    rs.append(0.5*oD)
    zs.append(-zcoord)
    rs.append(0.5*oD)
    zs.append(zcoord)
    rs.append(0.5*iD)
    zs.append(zcoord)
    rs.append(0.5*iD)
    zs.append(-zcoord)
    rs.append(0.5*oD)
    zs.append(-zcoord)

    # use revolveZ to create output string
    outstr += revolveZ(rs, zs)

    return outstr


def rimZcut(iD, oD, L, L2):
    """
␣␣␣␣Creates␣a␣rim␣or␣annulus␣about␣the␣origin␣in␣the␣Z␣direction
␣␣␣␣with␣inner␣diameter␣iD,␣outer␣diameter␣oD␣and␣length␣L.
␣␣␣␣Also␣has␣a␣cutout␣for␣a␣second␣interior␣rim␣with␣length␣L2.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣iD:␣number
␣␣␣␣␣␣␣␣Inner␣diameter␣of␣the␣rim.
␣␣␣␣oD:␣number
␣␣␣␣␣␣␣␣Outer␣diameter␣of␣the␣rim.
␣␣␣␣L:␣␣number
␣␣␣␣␣␣␣␣Length␣of␣the␣rim␣excluding␣end␣profiling.
␣␣␣␣L2:␣number
␣␣␣␣␣␣␣␣Length␣of␣the␣cutout␣section.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣outstr:␣string
␣␣␣␣␣␣␣␣A␣'facetstring'␣detailing␣the␣triangles
␣␣␣␣␣␣␣␣that␣make␣up␣the␣rim␣surface.
␣␣␣␣"""

    # make output string
    outstr = ""

    # create co-ordinate lists
    rs = []
    zs = []

    # calculate some useful variables
    curvrad = 0.25*(oD - iD)
    curvcen = 0.25*(oD + iD)
    zcoord = 0.5*L

    # create cylinder co-ordinates
    rs.append(0.5*oD)
    zs.append(-zcoord)
    rs.append(0.5*oD)
    zs.append(zcoord)
    for i in range(1, 180):
        rs.append(curvcen+(curvrad*cos(radians(i))))
        zs.append(zcoord+(curvrad*sin(radians(i))))
    rs.append(0.5*iD)
    zs.append(zcoord)
    # add in cut
    rs.append(0.5*iD)
    zs.append(0.5*L2)
    rs.append(curvcen)
    zs.append(0.5*L2)
    rs.append(curvcen)
    zs.append(-0.5*L2)
    rs.append(0.5*iD)
    zs.append(-0.5*L2)
    # continue round profile
    rs.append(0.5*iD)
    zs.append(-zcoord)
    for i in range(1, 180):
        rs.append(curvcen-(curvrad*cos(radians(i))))
        zs.append(-zcoord-(curvrad*sin(radians(i))))
    rs.append(0.5*oD)
    zs.append(-zcoord)

    # use revolveZ to create output string
    outstr += revolveZ(rs, zs)

    return outstr


def rimZcut2(iD, oD, L, L2, el, rb):
    """
    Creates a rim or annulus about the origin in the Z direction
    with inner diameter iD, outer diameter oD and length L.
    Also has a cutout for a second interior rim with length L2.
    Additional parameters el and rb shape the curvature of the duct end
    with extra length and radial bias respectively.

    Parameters
    ----------
    iD: number
        Inner diameter of the rim.
    oD: number
        Outer diameter of the rim.
    L:  number
        Length of the rim excluding end profiling.
    L2: number
        Length of the cutout section.
    el: number
        Extra length to be added to the rim profile.
    rb: number
        Radial bias of end profile between 0 and 1,
        0 denoting a complete inwards bias of end profile,
        1 denoting a complete outwards bias of end profile.

    Returns
    -------
    outstr: string
        A 'facetstring' detailing the triangles
        that make up the rim surface.
    """
    # this function should be similar rimZcut if el = 0.25*(oD-iD) and rb = 0.5

    # make output string
    outstr = ""

    # create co-ordinate lists
    rs = []
    zs = []

    # calculate some useful variables
    curvradu = 0.5*(oD-iD)*(1-rb)
    curvradl = 0.5*(oD-iD)*rb
    curvcen = (0.5*iD)+curvradl
    zcoord = 0.5*L

    # create cylinder co-ordinates
    rs.append(0.5*oD)
    zs.append(-zcoord)
    rs.append(0.5*oD)
    zs.append(zcoord)
    for i in range(1, 50):
        tempx = i/50.0
        rs.append(curvcen+curvradu*(sqrt(1-tempx**2)))
        zs.append(zcoord+el*tempx)
    for i in range(1, 49):
        tempx = (50-i)/50.0
        rs.append(curvcen-curvradl*(sqrt(1-tempx**2)))
        zs.append(zcoord+el*tempx)
    # old circular ends
    #for i in range(1,180):
    #    rs.append(curvcen+(curvrad*cos(radians(i))))
    #    zs.append(zcoord+(curvrad*sin(radians(i))))
    rs.append(0.5*iD)
    zs.append(zcoord)
    # add in cut
    rs.append(0.5*iD)
    zs.append(0.5*L2)
    rs.append(0.25*(iD+oD))  # old curvcen
    zs.append(0.5*L2)
    rs.append(0.25*(iD+oD))  # old curvcen
    zs.append(-0.5*L2)
    rs.append(0.5*iD)
    zs.append(-0.5*L2)
    # continue round profile
    rs.append(0.5*iD)
    zs.append(-zcoord)
    for i in range(1, 50):
        tempx = i/50.0
        rs.append(curvcen-curvradl*(sqrt(1-tempx**2)))
        zs.append(-zcoord-el*tempx)
    for i in range(1, 49):
        tempx = (50-i)/50.0
        rs.append(curvcen+curvradu*(sqrt(1-tempx**2)))
        zs.append(-zcoord-el*tempx)
    # old circular ends
    #for i in range(1,180):
    #    rs.append(curvcen-(curvrad*cos(radians(i))))
    #    zs.append(-zcoord-(curvrad*sin(radians(i))))
    rs.append(0.5*oD)
    zs.append(-zcoord)

    # use revolveZ to create output string
    outstr += revolveZ(rs, zs)

    return outstr


def rimZcut3(iD, oD, L, L2, el, rb):
    """
␣␣␣␣Creates␣a␣rim␣or␣annulus␣about␣the␣origin␣in␣the␣Z␣direction
␣␣␣␣with␣inner␣diameter␣iD,␣outer␣diameter␣oD␣and␣length␣L.
␣␣␣␣Also␣has␣a␣cutout␣for␣a␣second␣interior␣rim␣with␣length␣L2.
␣␣␣␣Additional␣parameters␣el␣and␣rb␣shape␣the␣curvature␣of␣the␣duct␣end
␣␣␣␣with␣extra␣length␣and␣radial␣bias␣respectively␣-␣using␣a␣Bezier␣spline.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣iD:␣number
␣␣␣␣␣␣␣␣Inner␣diameter␣of␣the␣rim.
␣␣␣␣oD:␣number
␣␣␣␣␣␣␣␣Outer␣diameter␣of␣the␣rim.
␣␣␣␣L:␣␣number
␣␣␣␣␣␣␣␣Length␣of␣the␣rim␣excluding␣end␣profiling.
␣␣␣␣L2:␣number
␣␣␣␣␣␣␣␣Length␣of␣the␣cutout␣section.
␣␣␣␣el:␣number
␣␣␣␣␣␣␣␣Extra␣length␣to␣be␣added␣to␣the␣rim␣profile.
␣␣␣␣rb:␣number
␣␣␣␣␣␣␣␣Radial␣bias␣of␣end␣profile␣between␣0␣and␣1,
␣␣␣␣␣␣␣␣0␣denoting␣a␣complete␣inwards␣bias␣of␣end␣profile,
␣␣␣␣␣␣␣␣1␣denoting␣a␣complete␣outwards␣bias␣of␣end␣profile.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣outstr:␣string
␣␣␣␣␣␣␣␣A␣'facetstring'␣detailing␣the␣triangles
␣␣␣␣␣␣␣␣that␣make␣up␣the␣rim␣surface.
␣␣␣␣"""
    # this function should be similar rimZcut if el = 0.25*(oD-iD) and rb = 0.5

    # make output string
    outstr = ""

    # create co-ordinate lists
    rs = []
    zs = []

    # calculate some useful variables
    curvcen = (0.5*iD)+0.5*(oD-iD)*rb
    zcoord = 0.5*L

    # calculate co-ordinates for Bezier spline control points
    # - front:
    fr0 = 0.5*oD
    fz0 = zcoord
    fr1 = curvcen
    fz1 = zcoord+el
    fr2 = 0.5*iD
    fz2 = zcoord
    # - back:
    br0 = 0.5*iD
    bz0 = -zcoord
    br1 = curvcen
    bz1 = -zcoord-el
    br2 = 0.5*oD
    bz2 = -zcoord

    # create t parameter array and slice off end points
    # to prevent duplication of t==0,1 points.
    ts = linspace(0, 1, 101)[1:-1]

    # create cylinder co-ordinates
    rs.append(0.5*oD)
    zs.append(-zcoord)
    rs.append(0.5*oD)
    zs.append(zcoord)
    for t in ts:
        rs.append((fr0*(1-t)*(1-t))+(2*fr1*t*(1-t))+(fr2*t*t))
        zs.append((fz0*(1-t)*(1-t))+(2*fz1*t*(1-t))+(fz2*t*t))
    # old elliptic profile
    #for i in range(1,50):
    #    tempx = i/50.0
    #    rs.append(curvcen+curvradu*(sqrt(1-tempx**2)))
    #    zs.append(zcoord+el*tempx)
    #for i in range(1,49):
    #    tempx = (50-i)/50.0
    #    rs.append(curvcen-curvradl*(sqrt(1-tempx**2)))
    #    zs.append(zcoord+el*tempx)
    rs.append(0.5*iD)
    zs.append(zcoord)
    # add in cut
    rs.append(0.5*iD)
    zs.append(0.5*L2)
    rs.append(0.25*(iD+oD))  # old curvcen
    zs.append(0.5*L2)
    rs.append(0.25*(iD+oD))  # old curvcen
    zs.append(-0.5*L2)
    rs.append(0.5*iD)
    zs.append(-0.5*L2)
    # continue round profile
    rs.append(0.5*iD)
    zs.append(-zcoord)
    for t in ts:
        rs.append((br0*(1-t)*(1-t))+(2*br1*t*(1-t))+(br2*t*t))
        zs.append((bz0*(1-t)*(1-t))+(2*bz1*t*(1-t))+(bz2*t*t))
    # old elliptic profile
    #for i in range(1,50):
    #    tempx = i/50.0
    #    rs.append(curvcen-curvradl*(sqrt(1-tempx**2)))
    #    zs.append(-zcoord-el*tempx)
    #for i in range(1,49):
    #    tempx = (50-i)/50.0
    #    rs.append(curvcen+curvradu*(sqrt(1-tempx**2)))
    #    zs.append(-zcoord-el*tempx)
    rs.append(0.5*oD)
    zs.append(-zcoord)

    # use revolveZ to create output string
    outstr += revolveZ(rs, zs)

    return outstr


def rimBlock(D, dz):
    """
    Creates a cylindrical surface designed to block off the rim gap
    in a rim driven thruster to simplify the flow.

    Parameters
    ----------
    D:  number
        Diameter of cylindrical surface block.
    dz: number
        Half length of the cylindrical surface in the axial (Z) direction.

    Returns
    -------
    outstr: string
        A 'facetstring' detailing the triangles
        that make up the cylindrical blocking surface.
    """

    # make output string
    outstr = ""

    # create central cylinder co-ordinates
    r = 0.5*D
    cylxs = r*cos(radians(arange(361)))
    cylys = r*sin(radians(arange(361)))
    cylz1s = dz*ones(361)
    cylz2s = -dz*ones(361)

    outstr += twoface(cylxs, cylys, cylz1s, cylxs, cylys, cylz2s)

    return outstr


def rimBlock100mm():
    """
    Creates a rim block for 100mm thruster.

    Parameters
    ----------
    None

    Returns
    -------
    outstr: string
        A 'facetstring' detailing the triangles
        that make up the cylindrical blocking surface.
    """
    writestl("rimBlock100", rimBlock(0.1001, 0.017))
    return


def revolveZ(rs, zs):
    """
    Creates a revolved surface around the Z axis,
    based on a section defined by co-ordinates rs and zs
    in the radial and axial direction respectively.

    Parameters
    ----------
    rs: list of numbers
        A list of radial co-ordinates to be revolved.
    zs: list of numbers
        A list of axial co-ordinates to be revolved,
        should be the same length as rs.

    Returns
    -------
    outstr: string
        A 'facetstring' detailing the triangles
        that make up the revolved surface.
    """

    # make output string
    outstr = ""

    # make initial co-ordinates
    rsarray = array(rs)
    zsarray = array(zs)

    # do revolution
    for i in range(360):
        outstr += twoface(rsarray*cos(radians(i)), rsarray*sin(radians(i)),
                          zsarray, rsarray*cos(radians(i+1)),
                          rsarray*sin(radians(i+1)), zsarray)

    # return output string
    return outstr


def rotatestlZ(filename, angle):
    """
    Rotates the surface in filename by angle degrees
    about the Z axis and overwrites it.

    Parameters
    ----------
    filename:   string
        Name of the .stl file to be rotated.
    angle:  number
        Angle through which to rotate the .stl file in degrees.

    Returns
    -------
    None
        if successful.
    """
    from math import atan2
    fin = open(filename, 'r')
    lines = fin.readlines()
    fin.close()

    outstr = ""

    for line in lines:
        words = line.split()
        if len(words) == 5:
            # calculate rotation
            xcoord = float(words[2])
            ycoord = float(words[3])
            tA = atan2(ycoord, xcoord)
            tB = tA - radians(angle)
            r = (xcoord**2 + ycoord**2)**0.5
            newx = r*cos(tB)
            newy = r*sin(tB)
            # add string
            outstr += words[0] + " " + words[1] + " "
            outstr += str(newx) + " " + str(newy) + " " + words[4] + "\n"
        elif len(words) == 4:
            # calculate rotation
            xcoord = float(words[1])
            ycoord = float(words[2])
            tA = atan2(ycoord, xcoord)
            tB = tA - radians(angle)
            r = (xcoord**2 + ycoord**2)**0.5
            newx = r*cos(tB)
            newy = r*sin(tB)
            # add string
            outstr += words[0] + " "
            outstr += str(newx) + " " + str(newy) + " " + words[3] + "\n"
        else:
                outstr += line

        # write stl
        fout = open(filename, 'w')
        fout.write(outstr)
        fout.close()

        return None


def scalestl(filename, factor):
    """
    Scales the surface in filename by multiply all vertex co-ordinates by
    factor and then overwrites it.

    Parameters
    ----------
    filename:   string
        Name of the .stl file to be scaled.
    factor:  number
        Factor by which to scale the .stl file.

    Returns
    -------
    None
        if successful.
    """
    fin = open(filename, 'r')
    lines = fin.readlines()
    fin.close()

    outstr = ""

    for line in lines:
        words = line.split()
        if len(words) == 4:
            # calculate rotation
            xcoord = float(words[1])
            ycoord = float(words[2])
            zcoord = float(words[3])
            newx = xcoord*factor
            newy = ycoord*factor
            newz = zcoord*factor
            # add string
            outstr += "      " + words[0] + " "
            outstr += str(newx) + " " + str(newy) + " " + str(newz) + "\n"
        else:
            outstr += line

    # write stl
    fout = open(filename, 'w')
    fout.write(outstr)
    fout.close()

    return None
```

Appendix C

OpenFOAM Automation Functions

This is a collection of Python functions written to automate the writing and execution of OpenFOAM cases, as well as post process the data in such a way that the functions could be incorporated into a single objective function in an automated design optimisation study.

```
”””
ajopenfoam.py

A␣set␣of␣OpenFOAM␣case␣generation␣and␣execution␣functions.

(c)␣copyright␣Aleksander␣Dubas␣2011−2013
”””
import os


def makeCase(casename, overwrite=False):
    ”””
␣␣␣␣Makes␣a␣case␣with␣casename␣and␣the␣basic␣directory␣structure␣for␣an
␣␣␣␣OpenFOAM␣case.␣␣Changes␣present␣directory␣to␣within␣the␣case.

␣␣␣␣Parameters
␣␣␣␣−−−−−−−−−−
␣␣␣␣casename:␣␣␣string
␣␣␣␣␣␣␣␣Name␣of␣the␣case␣directory.
␣␣␣␣overwrite:␣␣boolean␣␣␣␣␣(default␣False)
␣␣␣␣␣␣␣␣Overwrite␣the␣case␣if␣it␣is␣pre−existing.

␣␣␣␣Returns
␣␣␣␣−−−−−−−
␣␣␣␣None
␣␣␣␣␣␣␣␣If␣successful.
␣␣␣␣”””

    # change directory to $FOAM_RUN directory
    os.chdir(os.path.expandvars(”$FOAM_RUN”))

    try:
        # make the main case directory
        os.mkdir(casename)
        # change to case directory
        os.chdir(casename)
    except:
        if overwrite:
            # try changing to directory and emptying it
            try:
                os.chdir(casename)
            except:
                raise IOError(”makeCase:\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Cannot␣make␣directory␣or␣change␣to␣it!”)
            # paranoia check, make sure we're not deleting anything important!
            if os.getcwd().split(”/”)[−1] == casename:
                os.system(”rm␣−r␣*”)
            else:
                raise IOError(”makeCase:\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Tried␣to␣overwrite,␣but␣not␣sure␣in␣correct␣location.”)
        if not overwrite:
            raise IOError(”makeCase:␣Case␣already␣exists.”)

    # make necessary directories for the case
    os.mkdir(”system”)
    os.mkdir("constant")
    os.mkdir("0")
    os.mkdir("constant/polyMesh")
    os.mkdir("constant/triSurface")

    return None


def makeDomain(xmin, ymin, zmin, xmax, ymax, zmax, base, sf=1):
    """
␣␣␣␣Creates␣a␣blockMesh␣dictionary␣to␣create␣a␣domain
␣␣␣␣stretching␣from␣xmin,ymin,zmin␣to␣xmax,ymax,zmax
␣␣␣␣with␣a␣base␣length␣of␣base.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣xmin:␣␣␣number
␣␣␣␣␣␣␣␣Minimum␣x␣of␣domain.
␣␣␣␣ymin:␣␣␣number
␣␣␣␣␣␣␣␣Minimum␣y␣of␣domain.
␣␣␣␣zmin:␣␣␣number
␣␣␣␣␣␣␣␣Minimum␣z␣of␣domain.
␣␣␣␣xmax:␣␣␣number
␣␣␣␣␣␣␣␣Maximum␣x␣of␣domain.
␣␣␣␣ymax:␣␣␣number
␣␣␣␣␣␣␣␣Maximum␣y␣of␣domain.
␣␣␣␣zmax:␣␣␣number
␣␣␣␣␣␣␣␣Maximum␣z␣of␣domain.
␣␣␣␣base:␣␣␣number
␣␣␣␣␣␣␣␣Approximate␣cube␣side␣length␣of␣base␣mesh␣cells.
␣␣␣␣sf:␣␣␣␣␣number␣␣(default␣1)
␣␣␣␣␣␣␣␣Scaling␣factor␣for␣base␣mesh.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣None
␣␣␣␣␣␣␣␣If␣successful.
␣␣␣␣"""

    fout = file("constant/polyMesh/blockMeshDict", "w")

    # blank space to make code a bit neater
    sp = "␣"

    # work out number of cells in each direction, based on base length.
    # this should lead to approximately cubic cells
    xcells = str(int((xmax-xmin)/base))
    ycells = str(int((ymax-ymin)/base))
    zcells = str(int((zmax-zmin)/base))

    # convert all inputs into strings ready for output
    xmin = str(xmin)
    ymin = str(ymin)
    zmin = str(zmin)
    xmax = str(xmax)
    ymax = str(ymax)
    zmax = str(zmax)
    sf = str(sf)

    # write file header
    fout.write("FoamFile\n{\n")
    fout.write("␣␣version␣␣␣␣␣2.0;\n")
    fout.write("␣␣format␣␣␣␣␣␣ascii;\n")
    fout.write("␣␣class␣␣␣␣␣␣␣dictionary;\n")
    fout.write("␣␣object␣␣␣␣␣␣blockMeshDict;\n")
    fout.write("␣␣location␣␣␣␣\"constant/polyMesh\";\n}\n\n")

    # write scaling factor
    fout.write("convertToMeters␣" + sf + ";\n\n")

    # write vertices
    fout.write("vertices\n(\n")
    fout.write("␣␣("+xmin+sp+ymin+sp+zmin+")\n")
    fout.write("␣␣("+xmax+sp+ymin+sp+zmin+")\n")
    fout.write("␣␣("+xmax+sp+ymax+sp+zmin+")\n")
    fout.write("␣␣("+xmin+sp+ymax+sp+zmin+")\n")
    fout.write("␣␣("+xmin+sp+ymin+sp+zmax+")\n")
    fout.write("␣␣("+xmax+sp+ymin+sp+zmax+")\n")
    fout.write("␣␣("+xmax+sp+ymax+sp+zmax+")\n")
    fout.write("␣␣("+xmin+sp+ymax+sp+zmax+")\n")
    fout.write(");\n\n")

    # write blocks
    fout.write("blocks\n(\n")
    fout.write("␣␣hex␣(0␣1␣2␣3␣4␣5␣6␣7)\n")
    fout.write("␣␣("+xcells+"␣"+ycells+"␣"+zcells+")\n")
    fout.write("␣␣simpleGrading␣(1␣1␣1)\n")
    fout.write(");\n\n")

    # write edges (empty for square domain)
    fout.write("edges\n(\n);\n\n")
    # write patches
    fout.write("patches\n(\n")
    fout.write("␣␣patch␣xmin\n␣␣(␣(0␣4␣7␣3)␣)\n")
    fout.write("␣␣patch␣xmax\n␣␣(␣(2␣6␣5␣1)␣)\n")
    fout.write("␣␣patch␣ymin\n␣␣(␣(1␣5␣4␣0)␣)\n")
    fout.write("␣␣patch␣ymax\n␣␣(␣(3␣7␣6␣2)␣)\n")
    fout.write("␣␣patch␣zmin\n␣␣(␣(0␣3␣2␣1)␣)\n")
    fout.write("␣␣patch␣zmax\n␣␣(␣(4␣5␣6␣7)␣)\n);\n\n")

    # close output
    fout.flush()
    os.fsync(fout.fileno())
    fout.close()

    return None


def makeAnnulus_methodA(r, r1, r2, l1, l2, n, AR=2, dg=10, verbose=False):
    """
␣␣␣␣Creates␣an␣annulus␣blockMeshDict␣using␣method␣A.
␣␣␣␣Leaves␣interfacing␣between␣axial␣and␣radial␣gaps␣up␣to␣blockMesh.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣r:␣␣number
␣␣␣␣␣␣␣␣The␣inner␣radius␣of␣end␣plates.
␣␣␣␣r1:␣number
␣␣␣␣␣␣␣␣The␣inner␣radius␣of␣annulus.
␣␣␣␣r2:␣number
␣␣␣␣␣␣␣␣The␣outer␣radius␣of␣annulus.
␣␣␣␣l1:␣number
␣␣␣␣␣␣␣␣The␣half␣length␣to␣inner␣end.
␣␣␣␣l2:␣number
␣␣␣␣␣␣␣␣The␣half␣length␣to␣outer␣end.
␣␣␣␣n:␣␣number
␣␣␣␣␣␣␣␣Number␣of␣cells␣in␣wall-normal␣direction.
␣␣␣␣AR:␣number␣␣(default␣2)
␣␣␣␣␣␣␣␣Aspect␣ratio␣in␣wall␣tangential␣direction.
␣␣␣␣dg:␣number␣␣(default␣10)
␣␣␣␣␣␣␣␣Number␣of␣degrees␣of␣rotation␣to␣be␣simulated.
␣␣␣␣verbose:␣␣␣␣boolean␣␣␣␣␣(default␣False)
␣␣␣␣␣␣␣␣Switch␣to␣turn␣on␣verbose␣output.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣None
␣␣␣␣␣␣␣␣If␣successful.
␣␣␣␣"""
    from math import sin, cos, radians, pi

    # visualisation of domain
    #     y
    #     |
    # z<-x
    #          ---------------      r2
    #        |    ---------    |    |
    #        |   |          |   |    r1
    #        |   |          |   |    |
    #       l2-l1----|----l1-l2   r

    fout = file("constant/polyMesh/blockMeshDict", "w")

    # calculate temporary sin and cos variables
    sd = sin(radians(0.5 * dg))
    cd = cos(radians(0.5 * dg))

    # calculate number of cells in non-normal directions
    cpx = int(2 * r1 * pi * (dg / 360.0) / (AR * (r2 - r1) / float(n)))
    cpy = int(n)
    cpz = int(2 * l1 / (AR * (r2 - r1) / float(n)))
    apx = int(2 * r1 * pi * (dg / 360.0) / (AR * (l2 - l1) / float(n)))
    apy = int((r2 - r) / (AR * (l2 - l1) / float(n)))
    apz = int(n)
    if verbose:
        print("Total␣cells␣=␣" + str(cpx * cpy * cpz + 2 * apx * apy * apz))

    # write file header
    fout.write("FoamFile\n{\n")
    fout.write("␣␣version␣␣␣␣␣2.0;\n")
    fout.write("␣␣format␣␣␣␣␣␣ascii;\n")
    fout.write("␣␣class␣␣␣␣␣␣␣dictionary;\n")
    fout.write("␣␣object␣␣␣␣␣␣blockMeshDict;\n")
    fout.write("␣␣location␣␣␣␣\"constant/polyMesh\";\n}\n\n")

    # write scaling factor
    fout.write("convertToMeters␣1;\n\n")

    # write vertices
    fout.write("vertices\n(\n")
    # central part
    fout.write("␣␣("+str(-sd*r1)+"␣"+str(cd*r1)+"␣"+str(l1)+")\n")
    fout.write("␣␣("+str(sd*r1)+"␣"+str(cd*r1)+"␣"+str(l1)+")\n")
    fout.write("␣␣("+str(sd*r1)+"␣"+str(cd*r1)+"␣"+str(-l1)+")\n")
    fout.write("␣␣("+str(-sd*r1)+"␣"+str(cd*r1)+"␣"+str(-l1)+")\n")
    fout.write("␣␣("+str(-sd*r2)+"␣"+str(cd*r2)+"␣"+str(l1)+")\n")
    fout.write("␣␣("+str(sd*r2)+"␣"+str(cd*r2)+"␣"+str(l1)+")\n")
    fout.write("␣␣("+str(sd*r2)+"␣"+str(cd*r2)+"␣"+str(-l1)+")\n")
    fout.write("␣␣("+str(-sd*r2)+"␣"+str(cd*r2)+"␣"+str(-l1)+")\n")
    # front part
    fout.write("␣␣("+str(-sd*r)+"␣"+str(cd*r)+"␣"+str(l2)+")\n")
    fout.write("␣␣("+str(sd*r)+"␣"+str(cd*r)+"␣"+str(l2)+")\n")
    fout.write("␣␣("+str(sd*r)+"␣"+str(cd*r)+"␣"+str(l1)+")\n")
    fout.write("␣␣("+str(-sd*r)+"␣"+str(cd*r)+"␣"+str(l1)+")\n")
    fout.write("␣␣("+str(-sd*r2)+"␣"+str(cd*r2)+"␣"+str(l2)+")\n")
    fout.write("␣␣("+str(sd*r2)+"␣"+str(cd*r2)+"␣"+str(l2)+")\n")
    fout.write("␣␣("+str(sd*r2)+"␣"+str(cd*r2)+"␣"+str(l1)+")\n")
    fout.write("␣␣("+str(-sd*r2)+"␣"+str(cd*r2)+"␣"+str(l1)+")\n")
    # rear part
    fout.write("␣␣("+str(-sd*r)+"␣"+str(cd*r)+"␣"+str(-l1)+")\n")
    fout.write("␣␣("+str(sd*r)+"␣"+str(cd*r)+"␣"+str(-l1)+")\n")
    fout.write("␣␣("+str(sd*r)+"␣"+str(cd*r)+"␣"+str(l2)+")\n")
    fout.write("␣␣("+str(-sd*r)+"␣"+str(cd*r)+"␣"+str(l2)+")\n")
    fout.write("␣␣("+str(-sd*r2)+"␣"+str(cd*r2)+"␣"+str(-l1)+")\n")
    fout.write("␣␣("+str(sd*r2)+"␣"+str(cd*r2)+"␣"+str(-l1)+")\n")
    fout.write("␣␣("+str(sd*r2)+"␣"+str(cd*r2)+"␣"+str(l2)+")\n")
    fout.write("␣␣("+str(-sd*r2)+"␣"+str(cd*r2)+"␣"+str(l2)+")\n")
    fout.write(");\n\n")

    # write edges
    fout.write("edges\n(\n")
    # central part
    fout.write("␣␣arc␣␣0␣␣1␣(0␣"+str(r1)+"␣"+str(l1)+")\n")
    fout.write("␣␣arc␣␣3␣␣2␣(0␣"+str(r1)+"␣"+str(-l1)+")\n")
    fout.write("␣␣arc␣␣4␣␣5␣(0␣"+str(r2)+"␣"+str(l1)+")\n")
    fout.write("␣␣arc␣␣7␣␣6␣(0␣"+str(r2)+"␣"+str(-l1)+")\n")
    # front part
    fout.write("␣␣arc␣␣8␣␣9␣(0␣"+str(r)+"␣"+str(l2)+")\n")
    fout.write("␣␣arc␣11␣10␣(0␣"+str(r)+"␣"+str(l1)+")\n")
    fout.write("␣␣arc␣12␣13␣(0␣"+str(r2)+"␣"+str(l2)+")\n")
    fout.write("␣␣arc␣15␣14␣(0␣"+str(r2)+"␣"+str(l1)+")\n")
    # rear part
    fout.write("␣␣arc␣16␣17␣(0␣"+str(r)+"␣"+str(-l2)+")\n")
    fout.write("␣␣arc␣19␣18␣(0␣"+str(r)+"␣"+str(-l1)+")\n")
    fout.write("␣␣arc␣20␣21␣(0␣"+str(r2)+"␣"+str(-l2)+")\n")
    fout.write("␣␣arc␣23␣22␣(0␣"+str(r2)+"␣"+str(-l1)+")\n")
    fout.write(");\n\n")

    # write blocks
    fout.write("blocks\n(\n")
    # central part
    fout.write("␣␣hex␣(␣0␣␣1␣␣2␣␣3␣␣4␣␣5␣␣6␣␣7)\n")
    fout.write("␣␣("+str(cpx)+"␣"+str(cpz)+"␣"+str(cpy)+")\n")
    fout.write("␣␣simpleGrading␣␣(1␣1␣1)\n")
    # front part
    fout.write("␣␣hex␣(␣8␣␣9␣10␣11␣12␣13␣14␣15)\n")
    fout.write("␣␣("+str(apx)+"␣"+str(apz)+"␣"+str(apy)+")\n")
    fout.write("␣␣simpleGrading␣␣(1␣1␣1)\n")
    # rear part
    fout.write("␣␣hex␣(16␣17␣18␣19␣20␣21␣22␣23)\n")
    fout.write("␣␣("+str(apx)+"␣"+str(apz)+"␣"+str(apy)+")\n")
    fout.write("␣␣simpleGrading␣␣(1␣1␣1)\n")
    fout.write(");\n\n")

    # write boundary
    # boundary naming - cp = central part, fp = front part, rp = rear part
    # - f = front (z+), r = rear (z-), i = inner (y-), o = outer (y+)
    # - c1 = cyclic1 (x+), c2 = cyclic2 (x-)
    fout.write("boundary\n(\n")
    # central part
    fout.write("␣␣cpr\n␣␣{\n␣␣␣␣type␣patch;\n")
    fout.write("␣␣␣␣faces␣(␣(␣2␣␣3␣␣7␣␣6)␣);\n␣␣}\n")
    fout.write("␣␣cpf\n␣␣{\n␣␣␣␣type␣patch;\n")
    fout.write("␣␣␣␣faces␣(␣(␣0␣␣1␣␣5␣␣4)␣);\n␣␣}\n")
    fout.write("␣␣cpi\n␣␣{\n␣␣␣␣type␣wall;\n")
    fout.write("␣␣␣␣faces␣(␣(␣0␣␣3␣␣2␣␣1)␣);\n␣␣}\n")
    fout.write("␣␣cpo\n␣␣{\n␣␣␣␣type␣wall;\n")
    fout.write("␣␣␣␣faces␣(␣(␣4␣␣5␣␣6␣␣7)␣);\n␣␣}\n")
    fout.write("␣␣cpc1\n␣␣{\n␣␣␣␣type␣cyclic;\n")
    fout.write("␣␣␣␣neighbourPatch␣cpc2;\n")
    fout.write("␣␣␣␣faces␣(␣(␣1␣␣2␣␣6␣␣5)␣);\n␣␣}\n")
    fout.write("␣␣cpc2\n␣␣{\n␣␣␣␣type␣cyclic;\n")
    fout.write("␣␣␣␣neighbourPatch␣cpc1;\n")
    fout.write("␣␣␣␣faces␣(␣(␣0␣␣4␣␣7␣␣3)␣);\n␣␣}\n")
    # front part
    fout.write("␣␣fpr\n␣␣{\n␣␣␣␣type␣wall;\n")
    fout.write("␣␣␣␣faces␣(␣(10␣11␣15␣14)␣);\n␣␣}\n")
    fout.write("␣␣fpf\n␣␣{\n␣␣␣␣type␣wall;\n")
    fout.write("␣␣␣␣faces␣(␣(␣8␣␣9␣13␣12)␣);\n␣␣}\n")
    fout.write("␣␣fpi\n␣␣{\n␣␣␣␣type␣patch;\n")
    fout.write("␣␣␣␣faces␣(␣(␣8␣11␣10␣␣9)␣);\n␣␣}\n")
    fout.write("␣␣fpo\n␣␣{\n␣␣␣␣type␣wall;\n")
    fout.write("␣␣␣␣faces␣(␣(12␣13␣14␣15)␣);\n␣␣}\n")
    fout.write("␣␣fpc1\n␣␣{\n␣␣␣␣type␣cyclic;\n")
    fout.write("␣␣␣␣neighbourPatch␣fpc2;\n")
```

```
    fout.write("␣␣␣␣faces␣(␣(␣9␣10␣14␣13)␣);\n␣␣}\n")
    fout.write("␣␣fpc2\n␣␣{\n␣␣␣␣type␣cyclic;\n")
    fout.write("␣␣␣␣neighbourPatch␣fpc1;\n")
    fout.write("␣␣␣␣faces␣(␣(␣8␣12␣15␣11)␣);\n␣␣}\n")
    # rear part
    fout.write("␣␣rpr\n␣␣{\n␣␣␣␣type␣wall;\n")
    fout.write("␣␣␣␣faces␣(␣(18␣19␣23␣22)␣);\n␣␣}\n")
    fout.write("␣␣rpf\n␣␣{\n␣␣␣␣type␣wall;\n")
    fout.write("␣␣␣␣faces␣(␣(16␣17␣21␣20)␣);\n␣␣}\n")
    fout.write("␣␣rpi\n␣␣{\n␣␣␣␣type␣patch;\n")
    fout.write("␣␣␣␣faces␣(␣(16␣19␣18␣17)␣);\n␣␣}\n")
    fout.write("␣␣rpo\n␣␣{\n␣␣␣␣type␣wall;\n")
    fout.write("␣␣␣␣faces␣(␣(20␣21␣22␣23)␣);\n␣␣}\n")
    fout.write("␣␣rpc1\n␣␣{\n␣␣␣␣type␣cyclic;\n")
    fout.write("␣␣␣␣neighbourPatch␣rpc2;\n")
    fout.write("␣␣␣␣faces␣(␣(17␣18␣22␣21)␣);\n␣␣}\n")
    fout.write("␣␣rpc2\n␣␣{\n␣␣␣␣type␣cyclic;\n")
    fout.write("␣␣␣␣neighbourPatch␣rpc1;\n")
    fout.write("␣␣␣␣faces␣(␣(16␣20␣23␣19)␣);\n␣␣}\n")
    fout.write(");\n\n")

    # write mergePatchPairs
    fout.write("mergePatchPairs\n(\n")
    fout.write("␣␣(fpr␣cpf)\n␣␣(rpf␣cpr)\n);\n\n")

    fout.flush()
    os.fsync(fout.fileno())
    fout.close()

    return None


def makeAnnulus_methodB(r, r1, r2, l1, l2, n, AR=2, dg=10, verbose=False):
    """
␣␣␣␣Creates␣an␣annulus␣blockMeshDict␣using␣method␣B.
␣␣␣␣Manually␣handles␣interface␣between␣parts␣by␣generating␣5␣separate␣regions.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣r:␣␣number
␣␣␣␣␣␣␣␣The␣inner␣radius␣of␣end␣plates.
␣␣␣␣r1:␣number
␣␣␣␣␣␣␣␣The␣inner␣radius␣of␣annulus.
␣␣␣␣r2:␣number
␣␣␣␣␣␣␣␣The␣outer␣radius␣of␣annulus.
␣␣␣␣l1:␣number
␣␣␣␣␣␣␣␣The␣half␣length␣to␣inner␣end.
␣␣␣␣l2:␣number
␣␣␣␣␣␣␣␣The␣half␣length␣to␣outer␣end.
␣␣␣␣n:␣␣number
␣␣␣␣␣␣␣␣Number␣of␣cells␣in␣wall-normal␣direction.
␣␣␣␣AR:␣number␣␣(default␣2)
␣␣␣␣␣␣␣␣Aspect␣ratio␣in␣wall␣tangential␣direction.
␣␣␣␣dg:␣number␣␣(default␣10)
␣␣␣␣␣␣␣␣Number␣of␣degrees␣of␣rotation␣to␣be␣simulated.
␣␣␣␣verbose:␣␣␣␣boolean␣␣␣␣␣(default␣False)
␣␣␣␣␣␣␣␣Switch␣to␣turn␣on␣verbose␣output.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣None
␣␣␣␣␣␣␣␣If␣successful.
␣␣␣␣"""
    from math import sin, cos, radians, pi

    # visualisation of domain
    #     y
    #     |
    # z<-x
    #          ---------------     r2
    #         |   ---------   |    |
    #         |   |        |   |    r1
    #         |   |        |   |    |
    #        l2-l1----|----l1-l2   r

    fout = file("constant/polyMesh/blockMeshDict", "w")

    # calculate temporary sin and cos variables
    sd = sin(radians(0.5 * dg))
    cd = cos(radians(0.5 * dg))

    # calculate number of cells in non-normal directions
    cpx = int(2 * r1 * pi * (dg / 360.0) / (AR * (r2 - r1) / float(n)))
    cpy = int(n)
    cpz = int(2 * l1 / (AR * (r2 - r1) / float(n)))
    apx = int(2 * r1 * pi * (dg / 360.0) / (AR * (l2 - l1) / float(n)))
    apy = int((r1 - r) / (AR * (l2 - l1) / float(n)))
    apz = int(n)
    if verbose:
        print("Total␣cells␣=␣" + str(cpx * cpy * cpz +
                                    2 * apx * apy * apz +
                                    2 * apx * cpy * apz))
```

```
# write file header
fout.write("FoamFile\n{\n")
fout.write("␣␣version␣␣␣␣␣2.0;\n")
fout.write("␣␣format␣␣␣␣␣␣ascii;\n")
fout.write("␣␣class␣␣␣␣␣␣␣dictionary;\n")
fout.write("␣␣object␣␣␣␣␣␣blockMeshDict;\n")
fout.write("␣␣location␣␣␣␣\"constant/polyMesh\";\n}\n\n")

# write scaling factor
fout.write("convertToMeters␣1;\n\n")

# write vertices
fout.write("vertices\n(\n")
# central part
fout.write("␣␣("+str(-sd*r1)+"␣"+str(cd*r1)+"␣"+str(l1)+")\n")
fout.write("␣␣("+str(sd*r1)+"␣"+str(cd*r1)+"␣"+str(l1)+")\n")
fout.write("␣␣("+str(sd*r1)+"␣"+str(cd*r1)+"␣"+str(-l1)+")\n")
fout.write("␣␣("+str(-sd*r1)+"␣"+str(cd*r1)+"␣"+str(-l1)+")\n")
fout.write("␣␣("+str(-sd*r2)+"␣"+str(cd*r2)+"␣"+str(l1)+")\n")
fout.write("␣␣("+str(sd*r2)+"␣"+str(cd*r2)+"␣"+str(l1)+")\n")
fout.write("␣␣("+str(sd*r2)+"␣"+str(cd*r2)+"␣"+str(-l1)+")\n")
fout.write("␣␣("+str(-sd*r2)+"␣"+str(cd*r2)+"␣"+str(-l1)+")\n")
# upper front part
fout.write("␣␣("+str(-sd*r1)+"␣"+str(cd*r1)+"␣"+str(l2)+")\n")
fout.write("␣␣("+str(sd*r1)+"␣"+str(cd*r1)+"␣"+str(l2)+")\n")
fout.write("␣␣("+str(sd*r1)+"␣"+str(cd*r1)+"␣"+str(l1)+")\n")
fout.write("␣␣("+str(-sd*r1)+"␣"+str(cd*r1)+"␣"+str(l1)+")\n")
fout.write("␣␣("+str(-sd*r2)+"␣"+str(cd*r2)+"␣"+str(l2)+")\n")
fout.write("␣␣("+str(sd*r2)+"␣"+str(cd*r2)+"␣"+str(l2)+")\n")
fout.write("␣␣("+str(sd*r2)+"␣"+str(cd*r2)+"␣"+str(l1)+")\n")
fout.write("␣␣("+str(-sd*r2)+"␣"+str(cd*r2)+"␣"+str(l1)+")\n")
# lower front part
fout.write("␣␣("+str(-sd*r)+"␣"+str(cd*r)+"␣"+str(l2)+")\n")
fout.write("␣␣("+str(sd*r)+"␣"+str(cd*r)+"␣"+str(l2)+")\n")
fout.write("␣␣("+str(sd*r)+"␣"+str(cd*r)+"␣"+str(l1)+")\n")
fout.write("␣␣("+str(-sd*r)+"␣"+str(cd*r)+"␣"+str(l1)+")\n")
fout.write("␣␣("+str(-sd*r1)+"␣"+str(cd*r1)+"␣"+str(l2)+")\n")
fout.write("␣␣("+str(sd*r1)+"␣"+str(cd*r1)+"␣"+str(l2)+")\n")
fout.write("␣␣("+str(sd*r1)+"␣"+str(cd*r1)+"␣"+str(l1)+")\n")
fout.write("␣␣("+str(-sd*r1)+"␣"+str(cd*r1)+"␣"+str(l1)+")\n")
# upper rear part
fout.write("␣␣("+str(-sd*r1)+"␣"+str(cd*r1)+"␣"+str(-l1)+")\n")
fout.write("␣␣("+str(sd*r1)+"␣"+str(cd*r1)+"␣"+str(-l1)+")\n")
fout.write("␣␣("+str(sd*r1)+"␣"+str(cd*r1)+"␣"+str(-l2)+")\n")
fout.write("␣␣("+str(-sd*r1)+"␣"+str(cd*r1)+"␣"+str(-l2)+")\n")
fout.write("␣␣("+str(-sd*r2)+"␣"+str(cd*r2)+"␣"+str(-l1)+")\n")
fout.write("␣␣("+str(sd*r2)+"␣"+str(cd*r2)+"␣"+str(-l1)+")\n")
fout.write("␣␣("+str(sd*r2)+"␣"+str(cd*r2)+"␣"+str(-l2)+")\n")
fout.write("␣␣("+str(-sd*r2)+"␣"+str(cd*r2)+"␣"+str(-l2)+")\n")
# lower rear part
fout.write("␣␣("+str(-sd*r)+"␣"+str(cd*r)+"␣"+str(-l1)+")\n")
fout.write("␣␣("+str(sd*r)+"␣"+str(cd*r)+"␣"+str(-l1)+")\n")
fout.write("␣␣("+str(sd*r)+"␣"+str(cd*r)+"␣"+str(-l2)+")\n")
fout.write("␣␣("+str(-sd*r)+"␣"+str(cd*r)+"␣"+str(-l2)+")\n")
fout.write("␣␣("+str(-sd*r1)+"␣"+str(cd*r1)+"␣"+str(-l1)+")\n")
fout.write("␣␣("+str(sd*r1)+"␣"+str(cd*r1)+"␣"+str(-l1)+")\n")
fout.write("␣␣("+str(sd*r1)+"␣"+str(cd*r1)+"␣"+str(-l2)+")\n")
fout.write("␣␣("+str(-sd*r1)+"␣"+str(cd*r1)+"␣"+str(-l2)+")\n")
fout.write(");\n\n")

# write edges
fout.write("edges\n(\n")
# central part
fout.write("␣␣arc␣␣0␣␣1␣(0␣"+str(r1)+"␣"+str(l1)+")\n")
fout.write("␣␣arc␣␣3␣␣2␣(0␣"+str(r1)+"␣"+str(-l1)+")\n")
fout.write("␣␣arc␣␣4␣␣5␣(0␣"+str(r2)+"␣"+str(l1)+")\n")
fout.write("␣␣arc␣␣7␣␣6␣(0␣"+str(r2)+"␣"+str(-l1)+")\n")
# upper front part
fout.write("␣␣arc␣␣8␣␣9␣(0␣"+str(r1)+"␣"+str(l2)+")\n")
fout.write("␣␣arc␣11␣10␣(0␣"+str(r1)+"␣"+str(l1)+")\n")
fout.write("␣␣arc␣12␣13␣(0␣"+str(r2)+"␣"+str(l2)+")\n")
fout.write("␣␣arc␣15␣14␣(0␣"+str(r2)+"␣"+str(l1)+")\n")
# lower front part
fout.write("␣␣arc␣16␣17␣(0␣"+str(r)+"␣"+str(l2)+")\n")
fout.write("␣␣arc␣19␣18␣(0␣"+str(r)+"␣"+str(l1)+")\n")
fout.write("␣␣arc␣20␣21␣(0␣"+str(r1)+"␣"+str(l2)+")\n")
fout.write("␣␣arc␣23␣22␣(0␣"+str(r1)+"␣"+str(l1)+")\n")
# upper rear part
fout.write("␣␣arc␣24␣25␣(0␣"+str(r1)+"␣"+str(-l1)+")\n")
fout.write("␣␣arc␣27␣26␣(0␣"+str(r1)+"␣"+str(-l2)+")\n")
fout.write("␣␣arc␣28␣29␣(0␣"+str(r2)+"␣"+str(-l1)+")\n")
fout.write("␣␣arc␣31␣30␣(0␣"+str(r2)+"␣"+str(-l2)+")\n")
# lower rear part
fout.write("␣␣arc␣32␣33␣(0␣"+str(r)+"␣"+str(-l1)+")\n")
fout.write("␣␣arc␣35␣34␣(0␣"+str(r)+"␣"+str(-l2)+")\n")
fout.write("␣␣arc␣36␣37␣(0␣"+str(r1)+"␣"+str(-l1)+")\n")
fout.write("␣␣arc␣39␣38␣(0␣"+str(r1)+"␣"+str(-l2)+")\n")
fout.write(");\n\n")

# write blocks
fout.write("blocks\n(\n")
# central part
fout.write("  hex ( 0  1  2  3  4  5  6  7)\n")
fout.write("  (" + str(cpx) + " " + str(cpz) + " " + str(cpy) + ")\n")
fout.write("  simpleGrading  (1 1 1)\n")
# upper front part
fout.write("  hex ( 8  9 10 11 12 13 14 15)\n")
fout.write("  (" + str(apx) + " " + str(apz) + " " + str(cpy) + ")\n")
fout.write("  simpleGrading  (1 1 1)\n")
# lower front part
fout.write("  hex (16 17 18 19 20 21 22 23)\n")
fout.write("  (" + str(apx) + " " + str(apz) + " " + str(apy) + ")\n")
fout.write("  simpleGrading  (1 1 1)\n")
# upper rear part
fout.write("  hex (24 25 26 27 28 29 30 31)\n")
fout.write("  (" + str(apx) + " " + str(apz) + " " + str(cpy) + ")\n")
fout.write("  simpleGrading  (1 1 1)\n")
# lower rear part
fout.write("  hex (32 33 34 35 36 37 38 39)\n")
fout.write("  (" + str(apx) + " " + str(apz) + " " + str(apy) + ")\n")
fout.write("  simpleGrading  (1 1 1)\n")
fout.write(");\n\n")

# write boundary
# boundary naming - cp = central part, fp = front part, rp = rear part
# - f = front (z+), r = rear (z-), i = inner (y-), o = outer (y+)
# - c1 = cyclic1 (x+), c2 = cyclic2 (x-)
# - u = upper, l = lower
fout.write("boundary\n(\n")
# central part
fout.write("  cpr\n  {\n    type patch;\n")
fout.write("    faces ( ( 2  3  7  6) );\n  }\n")
fout.write("  cpf\n  {\n    type patch;\n")
fout.write("    faces ( ( 0  1  5  4) );\n  }\n")
fout.write("  cpi\n  {\n    type wall;\n")
fout.write("    faces ( ( 0  3  2  1) );\n  }\n")
fout.write("  cpo\n  {\n    type wall;\n")
fout.write("    faces ( ( 4  5  6  7) );\n  }\n")
fout.write("  cpc1\n  {\n    type cyclic;\n")
fout.write("    neighbourPatch cpc2;\n")
fout.write("    faces ( ( 1  2  6  5) );\n  }\n")
fout.write("  cpc2\n  {\n    type cyclic;\n")
fout.write("    neighbourPatch cpc1;\n")
fout.write("    faces ( ( 0  4  7  3) );\n  }\n")
# upper front part
fout.write("  ufpr\n  {\n    type patch;\n")
fout.write("    faces ( (10 11 15 14) );\n  }\n")
fout.write("  ufpf\n  {\n    type wall;\n")
fout.write("    faces ( ( 8  9 13 12) );\n  }\n")
fout.write("  ufpi\n  {\n    type patch;\n")
fout.write("    faces ( ( 8 11 10  9) );\n  }\n")
fout.write("  ufpo\n  {\n    type wall;\n")
fout.write("    faces ( (12 13 14 15) );\n  }\n")
fout.write("  ufpc1\n  {\n    type cyclic;\n")
fout.write("    neighbourPatch ufpc2;\n")
fout.write("    faces ( ( 9 10 14 13) );\n  }\n")
fout.write("  ufpc2\n  {\n    type cyclic;\n")
fout.write("    neighbourPatch ufpc1;\n")
fout.write("    faces ( ( 8 12 15 11) );\n  }\n")
# lower front part
fout.write("  lfpr\n  {\n    type wall;\n")
fout.write("    faces ( (18 19 23 22) );\n  }\n")
fout.write("  lfpf\n  {\n    type wall;\n")
fout.write("    faces ( (16 17 21 20) );\n  }\n")
fout.write("  lfpi\n  {\n    type patch;\n")
fout.write("    faces ( (16 19 18 17) );\n  }\n")
fout.write("  lfpo\n  {\n    type patch;\n")
fout.write("    faces ( (20 21 22 23) );\n  }\n")
fout.write("  lfpc1\n  {\n    type cyclic;\n")
fout.write("    neighbourPatch lfpc2;\n")
fout.write("    faces ( (17 18 22 21) );\n  }\n")
fout.write("  lfpc2\n  {\n    type cyclic;\n")
fout.write("    neighbourPatch lfpc1;\n")
fout.write("    faces ( (16 20 23 19) );\n  }\n")
# upper rear part
fout.write("  urpr\n  {\n    type wall;\n")
fout.write("    faces ( (26 27 31 30) );\n  }\n")
fout.write("  urpf\n  {\n    type patch;\n")
fout.write("    faces ( (24 25 29 28) );\n  }\n")
fout.write("  urpi\n  {\n    type patch;\n")
fout.write("    faces ( (24 27 26 25) );\n  }\n")
fout.write("  urpo\n  {\n    type wall;\n")
fout.write("    faces ( (28 29 30 31) );\n  }\n")
fout.write("  urpc1\n  {\n    type cyclic;\n")
fout.write("    neighbourPatch urpc2;\n")
fout.write("    faces ( (25 26 30 29) );\n  }\n")
fout.write("  urpc2\n  {\n    type cyclic;\n")
fout.write("    neighbourPatch urpc1;\n")
fout.write("    faces ( (24 28 31 27) );\n  }\n")
# lower rear part
fout.write("  lrpr\n  {\n    type wall;\n")
fout.write("    faces ( (34 35 39 38) );\n  }\n")
fout.write("  lrpf\n  {\n    type wall;\n")
```

```
    fout.write("␣␣␣␣faces␣(␣(32␣33␣37␣36)␣);\n␣␣}\n")
    fout.write("␣␣lrpi\n␣␣{\n␣␣␣␣type␣patch;\n")
    fout.write("␣␣␣␣faces␣(␣(32␣35␣34␣33)␣);\n␣␣}\n")
    fout.write("␣␣lrpo\n␣␣{\n␣␣␣␣type␣patch;\n")
    fout.write("␣␣␣␣faces␣(␣(36␣37␣38␣39)␣);\n␣␣}\n")
    fout.write("␣␣lrpc1\n␣␣{\n␣␣␣␣type␣cyclic;\n")
    fout.write("␣␣␣␣neighbourPatch␣lrpc2;\n")
    fout.write("␣␣␣␣faces␣(␣(33␣34␣38␣37)␣);\n␣␣}\n")
    fout.write("␣␣lrpc2\n␣␣{\n␣␣␣␣type␣cyclic;\n")
    fout.write("␣␣␣␣neighbourPatch␣lrpc1;\n")
    fout.write("␣␣␣␣faces␣(␣(32␣36␣39␣35)␣);\n␣␣}\n")
    fout.write(");\n\n")

    # write mergePatchPairs
    fout.write("mergePatchPairs\n(\n")
    fout.write("␣␣(cpf␣ufpr)\n␣␣(cpr␣urpf)\n")
    fout.write("␣␣(ufpi␣lfpo)\n␣␣(urpi␣lrpo)\n")
    fout.write(");\n\n")

    fout.flush()
    os.fsync(fout.fileno())
    fout.close()

    return None


def makeAnnulus_methodC(r, r1, r2, l1, l2, n, AR=2, dg=10, verbose=False):
    """
␣␣␣␣Creates␣an␣annulus␣blockMeshDict␣using␣method␣C.
␣␣␣␣Manually␣handles␣interface␣between␣parts␣by␣generating␣5␣separate␣regions.
␣␣␣␣Adapted␣from␣method␣B␣to␣not␣use␣mergePatchPairs

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣r:␣␣number
␣␣␣␣␣␣␣␣The␣inner␣radius␣of␣end␣plates.
␣␣␣␣r1:␣number
␣␣␣␣␣␣␣␣The␣inner␣radius␣of␣annulus.
␣␣␣␣r2:␣number
␣␣␣␣␣␣␣␣The␣outer␣radius␣of␣annulus.
␣␣␣␣l1:␣number
␣␣␣␣␣␣␣␣The␣half␣length␣to␣inner␣end.
␣␣␣␣l2:␣number
␣␣␣␣␣␣␣␣The␣half␣length␣to␣outer␣end.
␣␣␣␣n:␣␣number
␣␣␣␣␣␣␣␣Number␣of␣cells␣in␣wall-normal␣direction.
␣␣␣␣AR:␣number␣␣(default␣2)
␣␣␣␣␣␣␣␣Aspect␣ratio␣in␣wall␣tangential␣direction.
␣␣␣␣dg:␣number␣␣(default␣10)
␣␣␣␣␣␣␣␣Number␣of␣degrees␣of␣rotation␣to␣be␣simulated.
␣␣␣␣verbose:␣␣␣␣boolean␣␣␣␣␣(default␣False)
␣␣␣␣␣␣␣␣Switch␣to␣turn␣on␣verbose␣output.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣None
␣␣␣␣␣␣␣␣If␣successful.
␣␣␣␣"""
    from math import sin, cos, radians, pi

    # visualisation of domain
    #    y
    #    |
    # z<-x
    #         ---------------     r2
    #        |    ---------    |    |
    #        |   |         |   |    r1
    #        |   |         |   |    |
    #       l2-l1----|----l1-l2   r

    fout = file("constant/polyMesh/blockMeshDict", "w")

    # calculate temporary sin and cos variables
    sd = sin(radians(0.5 * dg))
    cd = cos(radians(0.5 * dg))

    # calculate number of cells in non-normal directions
    cpx = int(2 * r1 * pi * (dg / 360.0) / (AR * (r2 - r1) / float(n)))
    cpy = int(n)
    cpz = int(2 * l1 / (AR * (r2 - r1) / float(n)))
    apx = int(2 * r1 * pi * (dg / 360.0) / (AR * (l2 - l1) / float(n)))
    apy = int((r1 - r) / (AR * (l2 - l1) / float(n)))
    apz = int(n)
    if verbose:
        print("Total␣cells␣=␣" + str(cpx * cpy * cpz +
                                     2 * apx * apy * apz +
                                     2 * apx * cpy * apz))

    # write file header
    fout.write("FoamFile\n{\n")
    fout.write("␣␣version␣␣␣␣␣2.0;\n")
    fout.write("␣␣format␣␣␣␣␣␣ascii;\n")
fout.write("␣␣class␣␣␣␣␣␣␣dictionary;\n")
fout.write("␣␣object␣␣␣␣␣␣blockMeshDict;\n")
fout.write("␣␣location␣␣␣␣\"constant/polyMesh\";\n}\n\n")

# write scaling factor
fout.write("convertToMeters␣1;\n\n")

# write vertices
fout.write("vertices\n(\n")
# central part
fout.write("␣␣("+str(-sd*r1)+"␣"+str(cd*r1)+"␣"+str(l1)+")\n")
fout.write("␣␣("+str(sd*r1)+"␣"+str(cd*r1)+"␣"+str(l1)+")\n")
fout.write("␣␣("+str(sd*r1)+"␣"+str(cd*r1)+"␣"+str(-l1)+")\n")
fout.write("␣␣("+str(-sd*r1)+"␣"+str(cd*r1)+"␣"+str(-l1)+")\n")
fout.write("␣␣("+str(-sd*r2)+"␣"+str(cd*r2)+"␣"+str(l1)+")\n")
fout.write("␣␣("+str(sd*r2)+"␣"+str(cd*r2)+"␣"+str(l1)+")\n")
fout.write("␣␣("+str(sd*r2)+"␣"+str(cd*r2)+"␣"+str(-l1)+")\n")
fout.write("␣␣("+str(-sd*r2)+"␣"+str(cd*r2)+"␣"+str(-l1)+")\n")
# upper front part
fout.write("␣␣("+str(-sd*r1)+"␣"+str(cd*r1)+"␣"+str(l2)+")\n")
fout.write("␣␣("+str(sd*r1)+"␣"+str(cd*r1)+"␣"+str(l2)+")\n")
#fout.write("  ("+str(sd*r1)+" "+str(cd*r1)+" "+str(l1)+")\n")
#fout.write("  ("+str(-sd*r1)+" "+str(cd*r1)+" "+str(l1)+")\n")
fout.write("␣␣("+str(-sd*r2)+"␣"+str(cd*r2)+"␣"+str(l2)+")\n")
fout.write("␣␣("+str(sd*r2)+"␣"+str(cd*r2)+"␣"+str(l2)+")\n")
#fout.write("  ("+str(sd*r2)+" "+str(cd*r2)+" "+str(l1)+")\n")
#fout.write("  ("+str(-sd*r2)+" "+str(cd*r2)+" "+str(l1)+")\n")
# lower front part
fout.write("␣␣("+str(-sd*r)+"␣"+str(cd*r)+"␣"+str(l2)+")\n")
fout.write("␣␣("+str(sd*r)+"␣"+str(cd*r)+"␣"+str(l2)+")\n")
fout.write("␣␣("+str(sd*r)+"␣"+str(cd*r)+"␣"+str(l1)+")\n")
fout.write("␣␣("+str(-sd*r)+"␣"+str(cd*r)+"␣"+str(l1)+")\n")
#fout.write("  ("+str(-sd*r1)+" "+str(cd*r1)+" "+str(l2)+")\n")
#fout.write("  ("+str(sd*r1)+" "+str(cd*r1)+" "+str(l2)+")\n")
#fout.write("  ("+str(sd*r1)+" "+str(cd*r1)+" "+str(l1)+")\n")
#fout.write("  ("+str(-sd*r1)+" "+str(cd*r1)+" "+str(l1)+")\n")
# upper rear part
#fout.write("  ("+str(-sd*r1)+" "+str(cd*r1)+" "+str(-l1)+")\n")
#fout.write("  ("+str(sd*r1)+" "+str(cd*r1)+" "+str(-l1)+")\n")
fout.write("␣␣("+str(sd*r1)+"␣"+str(cd*r1)+"␣"+str(-l2)+")\n")
fout.write("␣␣("+str(-sd*r1)+"␣"+str(cd*r1)+"␣"+str(-l2)+")\n")
#fout.write("  ("+str(-sd*r2)+" "+str(cd*r2)+" "+str(-l1)+")\n")
#fout.write("  ("+str(sd*r2)+" "+str(cd*r2)+" "+str(-l1)+")\n")
fout.write("␣␣("+str(sd*r2)+"␣"+str(cd*r2)+"␣"+str(-l2)+")\n")
fout.write("␣␣("+str(-sd*r2)+"␣"+str(cd*r2)+"␣"+str(-l2)+")\n")
# lower rear part
fout.write("␣␣("+str(-sd*r)+"␣"+str(cd*r)+"␣"+str(-l1)+")\n")
fout.write("␣␣("+str(sd*r)+"␣"+str(cd*r)+"␣"+str(-l1)+")\n")
fout.write("␣␣("+str(sd*r)+"␣"+str(cd*r)+"␣"+str(-l2)+")\n")
fout.write("␣␣("+str(-sd*r)+"␣"+str(cd*r)+"␣"+str(-l2)+")\n")
#fout.write("  ("+str(-sd*r1)+" "+str(cd*r1)+" "+str(-l1)+")\n")
#fout.write("  ("+str(sd*r1)+" "+str(cd*r1)+" "+str(-l1)+")\n")
#fout.write("  ("+str(sd*r1)+" "+str(cd*r1)+" "+str(-l2)+")\n")
#fout.write("  ("+str(-sd*r1)+" "+str(cd*r1)+" "+str(-l2)+")\n")
fout.write(");\n\n")

# write edges
fout.write("edges\n(\n")
# central part
fout.write("␣␣arc␣␣0␣␣1␣(0␣"+str(r1)+"␣"+str(l1)+")\n")
fout.write("␣␣arc␣␣3␣␣2␣(0␣"+str(r1)+"␣"+str(-l1)+")\n")
fout.write("␣␣arc␣␣4␣␣5␣(0␣"+str(r2)+"␣"+str(l1)+")\n")
fout.write("␣␣arc␣␣7␣␣6␣(0␣"+str(r2)+"␣"+str(-l1)+")\n")
# upper front part
fout.write("␣␣arc␣␣8␣␣9␣(0␣"+str(r1)+"␣"+str(l2)+")\n")
fout.write("␣␣arc␣10␣11␣(0␣"+str(r2)+"␣"+str(l2)+")\n")
# lower front part
fout.write("␣␣arc␣12␣13␣(0␣"+str(r)+"␣"+str(l2)+")\n")
fout.write("␣␣arc␣15␣14␣(0␣"+str(r)+"␣"+str(l1)+")\n")
# upper rear part
fout.write("␣␣arc␣17␣16␣(0␣"+str(r1)+"␣"+str(-l2)+")\n")
fout.write("␣␣arc␣19␣18␣(0␣"+str(r2)+"␣"+str(-l2)+")\n")
# lower rear part
fout.write("␣␣arc␣20␣21␣(0␣"+str(r)+"␣"+str(-l1)+")\n")
fout.write("␣␣arc␣23␣22␣(0␣"+str(r)+"␣"+str(-l2)+")\n")
fout.write(");\n\n")

# write blocks
fout.write("blocks\n(\n")
# central part
fout.write("␣␣hex␣(␣0␣␣1␣␣2␣␣3␣␣4␣␣5␣␣6␣␣7)\n")
fout.write("␣␣(" + str(cpx) + "␣" + str(cpz) + "␣" + str(cpy) + ")\n")
fout.write("␣␣simpleGrading␣␣(1␣1␣1)\n")
# upper front part
fout.write("␣␣hex␣(␣8␣␣9␣␣1␣␣0␣10␣11␣␣5␣␣4)\n")
fout.write("␣␣(" + str(cpx) + "␣" + str(apz) + "␣" + str(cpy) + ")\n")
fout.write("␣␣simpleGrading␣␣(1␣1␣1)\n")
# lower front part
fout.write("␣␣hex␣(12␣13␣14␣15␣␣8␣␣9␣␣1␣␣0)\n")
fout.write("␣␣(" + str(cpx) + "␣" + str(apz) + "␣" + str(apy) + ")\n")
fout.write("␣␣simpleGrading␣␣(1␣1␣1)\n")
# upper rear part
```

```
fout.write("␣␣hex␣(␣3␣␣2␣16␣17␣␣7␣␣6␣18␣19)\n")
fout.write("␣␣(" + str(cpx) + "␣" + str(apz) + "␣" + str(cpy) + ")\n")
fout.write("␣␣simpleGrading␣␣(1␣1␣1)\n")
# lower rear part
fout.write("␣␣hex␣(20␣21␣22␣23␣␣3␣␣2␣16␣17)\n")
fout.write("␣␣(" + str(cpx) + "␣" + str(apz) + "␣" + str(apy) + ")\n")
fout.write("␣␣simpleGrading␣␣(1␣1␣1)\n")
fout.write(");\n\n")

# write boundary
# boundary naming - cp = central part, fp = front part, rp = rear part
# - f = front (z+), r = rear (z-), i = inner (y-), o = outer (y+)
# - c1 = cyclic1 (x+), c2 = cyclic2 (x-)
# - u = upper, l = lower
fout.write("boundary\n(\n")
# central part
#fout.write("  cpr\n  {\n    type patch;\n")
#fout.write("    faces ( ( 2  3  7  6) );\n  }\n")
#fout.write("  cpf\n  {\n    type patch;\n")
#fout.write("    faces ( ( 0  1  5  4) );\n  }\n")
fout.write("␣␣cpi\n␣␣{\n␣␣␣␣type␣wall;\n")
fout.write("␣␣␣␣faces␣(␣(␣0␣␣3␣␣2␣␣1)␣);\n␣␣}\n")
fout.write("␣␣cpo\n␣␣{\n␣␣␣␣type␣wall;\n")
fout.write("␣␣␣␣faces␣(␣(␣4␣␣5␣␣6␣␣7)␣);\n␣␣}\n")
fout.write("␣␣cpc1\n␣␣{\n␣␣␣␣type␣cyclic;\n")
fout.write("␣␣␣␣neighbourPatch␣cpc2;\n")
fout.write("␣␣␣␣faces␣(␣(␣1␣␣2␣␣6␣␣5)␣);\n␣␣}\n")
fout.write("␣␣cpc2\n␣␣{\n␣␣␣␣type␣cyclic;\n")
fout.write("␣␣␣␣neighbourPatch␣cpc1;\n")
fout.write("␣␣␣␣faces␣(␣(␣0␣␣4␣␣7␣␣3)␣);\n␣␣}\n")
# upper front part
#fout.write("  ufpr\n  {\n    type patch;\n")
#fout.write("    faces ( ( 1  0  4  5) );\n  }\n")
fout.write("␣␣ufpf\n␣␣{\n␣␣␣␣type␣wall;\n")
fout.write("␣␣␣␣faces␣(␣(␣8␣␣9␣11␣10)␣);\n␣␣}\n")
#fout.write("  ufpi\n  {\n    type patch;\n")
#fout.write("    faces ( ( 8  0  1  9) );\n  }\n")
fout.write("␣␣ufpo\n␣␣{\n␣␣␣␣type␣wall;\n")
fout.write("␣␣␣␣faces␣(␣(10␣11␣␣5␣␣4)␣);\n␣␣}\n")
fout.write("␣␣ufpc1\n␣␣{\n␣␣␣␣type␣cyclic;\n")
fout.write("␣␣␣␣neighbourPatch␣ufpc2;\n")
fout.write("␣␣␣␣faces␣(␣(␣9␣␣1␣␣5␣11)␣);\n␣␣}\n")
fout.write("␣␣ufpc2\n␣␣{\n␣␣␣␣type␣cyclic;\n")
fout.write("␣␣␣␣neighbourPatch␣ufpc1;\n")
fout.write("␣␣␣␣faces␣(␣(␣8␣10␣␣4␣␣0)␣);\n␣␣}\n")
# lower front part
fout.write("␣␣lfpr\n␣␣{\n␣␣␣␣type␣wall;\n")
fout.write("␣␣␣␣faces␣(␣(14␣15␣␣0␣␣1)␣);\n␣␣}\n")
fout.write("␣␣lfpf\n␣␣{\n␣␣␣␣type␣wall;\n")
fout.write("␣␣␣␣faces␣(␣(12␣13␣␣9␣␣8)␣);\n␣␣}\n")
fout.write("␣␣lfpi\n␣␣{\n␣␣␣␣type␣patch;\n")
fout.write("␣␣␣␣faces␣(␣(12␣15␣14␣13)␣);\n␣␣}\n")
#fout.write("  lfpo\n  {\n    type patch;\n")
#fout.write("    faces ( ( 8  9  1  0) );\n  }\n")
fout.write("␣␣lfpc1\n␣␣{\n␣␣␣␣type␣cyclic;\n")
fout.write("␣␣␣␣neighbourPatch␣lfpc2;\n")
fout.write("␣␣␣␣faces␣(␣(13␣14␣␣1␣␣9)␣);\n␣␣}\n")
fout.write("␣␣lfpc2\n␣␣{\n␣␣␣␣type␣cyclic;\n")
fout.write("␣␣␣␣neighbourPatch␣lfpc1;\n")
fout.write("␣␣␣␣faces␣(␣(12␣␣8␣␣0␣15)␣);\n␣␣}\n")
# upper rear part
fout.write("␣␣urpr\n␣␣{\n␣␣␣␣type␣wall;\n")
fout.write("␣␣␣␣faces␣(␣(16␣17␣19␣18)␣);\n␣␣}\n")
#fout.write("  urpf\n  {\n    type patch;\n")
#fout.write("    faces ( ( 3  2  6  7) );\n  }\n")
#fout.write("  urpi\n  {\n    type patch;\n")
#fout.write("    faces ( ( 3 17 16  2) );\n  }\n")
fout.write("␣␣urpo\n␣␣{\n␣␣␣␣type␣wall;\n")
fout.write("␣␣␣␣faces␣(␣(␣7␣␣6␣18␣19)␣);\n␣␣}\n")
fout.write("␣␣urpc1\n␣␣{\n␣␣␣␣type␣cyclic;\n")
fout.write("␣␣␣␣neighbourPatch␣urpc2;\n")
fout.write("␣␣␣␣faces␣(␣(␣2␣16␣18␣␣6)␣);\n␣␣}\n")
fout.write("␣␣urpc2\n␣␣{\n␣␣␣␣type␣cyclic;\n")
fout.write("␣␣␣␣neighbourPatch␣urpc1;\n")
fout.write("␣␣␣␣faces␣(␣(␣3␣␣7␣19␣17)␣);\n␣␣}\n")
# lower rear part
fout.write("␣␣lrpr\n␣␣{\n␣␣␣␣type␣wall;\n")
fout.write("␣␣␣␣faces␣(␣(22␣23␣17␣16)␣);\n␣␣}\n")
fout.write("␣␣lrpf\n␣␣{\n␣␣␣␣type␣wall;\n")
fout.write("␣␣␣␣faces␣(␣(20␣21␣␣2␣␣3)␣);\n␣␣}\n")
fout.write("␣␣lrpi\n␣␣{\n␣␣␣␣type␣patch;\n")
fout.write("␣␣␣␣faces␣(␣(20␣23␣22␣21)␣);\n␣␣}\n")
#fout.write("  lrpo\n  {\n    type patch;\n")
#fout.write("    faces ( ( 3  2 16 17) );\n  }\n")
fout.write("␣␣lrpc1\n␣␣{\n␣␣␣␣type␣cyclic;\n")
fout.write("␣␣␣␣neighbourPatch␣lrpc2;\n")
fout.write("␣␣␣␣faces␣(␣(21␣22␣16␣␣2)␣);\n␣␣}\n")
fout.write("␣␣lrpc2\n␣␣{\n␣␣␣␣type␣cyclic;\n")
fout.write("␣␣␣␣neighbourPatch␣lrpc1;\n")
fout.write("␣␣␣␣faces␣(␣(20␣␣3␣17␣23)␣);\n␣␣}\n")
fout.write(");\n\n")
```

```
    # write mergePatchPairs - not needed in method C
    #fout.write("mergePatchPairs\n(\n")
    #fout.write("  (cpf ufpr)\n  (cpr urpf)\n")
    #fout.write("  (ufpi lfpo)\n  (urpi lrpo)\n")
    #fout.write(");\n\n")

    fout.flush()
    os.fsync(fout.fileno())
    fout.close()

    return None


def controlDict(ap, et, dt, wi=0):
    """
␣␣␣␣Creates␣a␣control␣dictionary␣in␣the␣case/system␣directory.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣ap:␣string
␣␣␣␣␣␣␣␣Application␣to␣solve␣with␣(application).
␣␣␣␣et:␣number
␣␣␣␣␣␣␣␣End␣time␣of␣the␣solution␣(endTime).
␣␣␣␣dt:␣number
␣␣␣␣␣␣␣␣Desired␣time␣step␣(deltaT).
␣␣␣␣wi:␣number␣␣(default␣write␣at␣et)
␣␣␣␣␣␣␣␣How␣often␣to␣write␣a␣save␣(writeInterval)
␣␣␣␣␣␣␣␣in␣number␣of␣time␣steps␣(timeSteps).

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣None
␣␣␣␣␣␣␣␣If␣successful.
␣␣␣␣"""

    fout = file("system/controlDict", "w")

    # write file header
    fout.write("FoamFile\n{\n")
    fout.write("␣␣version␣␣␣␣␣2.0;\n")
    fout.write("␣␣format␣␣␣␣␣␣ascii;\n")
    fout.write("␣␣class␣␣␣␣␣␣␣dictionary;\n")
    fout.write("␣␣object␣␣␣␣␣␣controlDict;\n")
    fout.write("␣␣location␣␣␣␣\"system\";\n}\n\n")

    # calculate write interval if none is specified.
    if wi == 0:
        wi = int(float(et)/float(dt))
    # convert numbers to strings
    et = str(et)
    dt = str(dt)
    wi = str(wi)

    # write controlDict
    fout.write("application␣␣␣␣␣␣␣␣"+ap+";\n\n")
    fout.write("startFrom␣␣␣␣␣␣␣␣␣␣latestTime;\n\n")
    fout.write("startTime␣␣␣␣␣␣␣␣␣␣0;\n\n")
    fout.write("stopAt␣␣␣␣␣␣␣␣␣␣␣␣␣endTime;\n\n")
    fout.write("endTime␣␣␣␣␣␣␣␣␣␣␣␣"+et+";\n\n")
    fout.write("deltaT␣␣␣␣␣␣␣␣␣␣␣␣␣"+dt+";\n\n")
    fout.write("writeControl␣␣␣␣␣␣␣timeStep;\n\n")
    fout.write("writeInterval␣␣␣␣␣␣"+wi+";\n\n")
    fout.write("purgeWrite␣␣␣␣␣␣␣␣␣0;\n\n")
    fout.write("writeFormat␣␣␣␣␣␣␣␣ascii;\n\n")
    fout.write("writePrecision␣␣␣␣␣6;\n\n")
    fout.write("writeCompression␣␣␣uncompressed;\n\n")
    fout.write("timeFormat␣␣␣␣␣␣␣␣␣general;\n\n")
    fout.write("timePrecision␣␣␣␣␣␣6;\n\n")
    fout.write("runTimeModifiable␣␣yes;\n\n")

    # close output
    fout.flush()
    os.fsync(fout.fileno())
    fout.close()

    return None


def forceFuncObj(name, patches, rho=0, cofr=[0, 0, 0], oi=1):
    """
␣␣␣␣Appends␣a␣force␣calculation␣to␣the␣control␣dictionary
␣␣␣␣in␣the␣case/system␣directory.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣name:␣␣␣string␣or␣list␣of␣strings
␣␣␣␣␣␣␣␣Name␣of␣the␣forces␣calculation,␣adds␣multiple␣calculations␣if␣a␣list.
␣␣␣␣patches:␣␣␣␣string␣or␣list␣of␣strings
␣␣␣␣␣␣␣␣Name␣of␣patches␣to␣include,␣separated␣by␣a␣space.
␣␣␣␣␣␣␣␣If␣given␣a␣list␣then␣each␣set␣of␣patches␣should␣correspond␣to␣the␣name
␣␣␣␣␣␣␣␣given␣in␣the␣name␣parameter.
␣␣␣␣␣␣␣␣i.e.␣patches[0]␣will␣be␣combined␣under␣name[0]
```

```
␣␣␣␣rho:␣␣␣␣number
␣␣␣␣␣␣␣␣Value␣of␣the␣density,␣default␣=␣0␣for␣compressible␣flow.
␣␣␣␣cofr:␣␣␣list␣of␣numbers
␣␣␣␣␣␣␣␣Centre␣of␣rotation␣co-ordinates␣for␣moment␣(torque)␣calculation.
␣␣␣␣oi:␣number
␣␣␣␣␣␣␣␣Output␣interval␣in␣timeSteps.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣None
␣␣␣␣␣␣␣␣If␣successful.
␣␣␣␣"""

    fout = file("system/controlDict", "a")

    # convert numerical inputs into strings
    rho = str(rho)
    cofrs = "("+str(cofr[0])+"␣"+str(cofr[1])+"␣"+str(cofr[2])+")"
    oi = str(int(oi))

    # write functions to controlDict if not list
    if type(name) != list:
        fout.write("functions\n(\n" + name +
                   "\n{\ntype␣forces;\nfunctionObjectLibs␣" +
                   "(\"libforces.so\");\n")
        fout.write("patches␣(" + patches + ");\n")
        if rho == "0":
            fout.write("rhoName␣rho;\nrhoInf␣1.0;\n")
        if rho != "0":
            fout.write("rhoName␣rhoInf;\nrhoInf␣" + rho + ";\n")
        fout.write("CofR␣" + cofrs +
                   ";\noutputControl␣timeStep;\noutputInterval␣1;\n}\n);\n\n")

    # write functions to controlDict if names are a list
    if type(name) == list:
        fout.write("functions\n(\n")
        for i in range(len(name)):
            fout.write(name[i] + "\n{\ntype␣forces;\nfunctionObjectLibs␣" +
                       "(\"libforces.so\");\n")
            fout.write("patches␣("+patches[i]+");\n")
            if rho == "0":
                fout.write("rhoName␣rho;\nrhoInf␣1.0;\n")
            if rho != "0":
                fout.write("rhoName␣rhoInf;\nrhoInf␣" + rho + ";\n")
            fout.write("CofR␣" + cofrs +
                       ";\noutputControl␣timeStep;\noutputInterval␣1;\n}\n")
        fout.write(");\n\n")

    # close output
    fout.flush()
    os.fsync(fout.fileno())
    fout.close()

    return None


def decomposeParDict(nsubs=1, method="simple", n=[1, 1, 1], delta=0.001):
    """
␣␣␣␣Creates␣a␣parallel␣decomposition␣dictionary␣in␣the␣case/system␣directory.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣nsubs:␣␣integer
␣␣␣␣␣␣␣␣Number␣of␣subdomains␣(numberOfSubdomains),
␣␣␣␣␣␣␣␣i.e.␣number␣of␣cores␣to␣use,
␣␣␣␣␣␣␣␣should␣be␣equal␣to␣nx*ny*nz.
␣␣␣␣method:␣string
␣␣␣␣␣␣␣␣Decomposition␣method,␣choice␣of␣simple␣or␣hierarchical.
␣␣␣␣n:␣␣list␣of␣ints
␣␣␣␣␣␣␣␣List␣of␣splits␣between␣x,␣y␣and␣z␣directions.
␣␣␣␣delta:␣␣number
␣␣␣␣␣␣␣␣Delta␣value.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣None
␣␣␣␣␣␣␣␣If␣successful.
␣␣␣␣"""

    fout = file("system/decomposeParDict", "w")

    # sanity check
    if abs(n[0]*n[1]*n[2] - nsubs) > 1e-4:
        print "Error␣in␣ajopenfoam.decomposeParDict:"
        print "␣␣Number␣of␣subdomains␣does␣not␣match␣total␣from␣n!"
        return -1

    # convert numerical inputs into strings
    nsubs = str(nsubs)
    nstr = "("+str(int(n[0]))+"␣"+str(int(n[1]))+"␣"+str(int(n[2]))+")"
    delta = str(delta)
    # make sure spellings are correct
    if method[0] == "s" or method[0] == "S":
        method = "simple"
    if method[0] == "h" or method[0] == "H":
        method = "hierarchical"

    # write file header
    fout.write("FoamFile\n{\n")
    fout.write("␣␣version␣␣2.0;\n")
    fout.write("␣␣format␣␣␣ascii;\n")
    fout.write("␣␣class␣␣␣␣dictionary;\n")
    fout.write("␣␣object␣␣␣decomposeParDict;\n")
    fout.write("␣␣location␣\"system\";\n")
    fout.write("}\n\n")

    # write decomposeParDict
    fout.write("numberOfSubdomains␣" + nsubs + ";\n\n")
    fout.write("method␣" + method + ";\n\n")
    fout.write("simpleCoeffs\n{\nn␣" + nstr + ";\ndelta␣" + delta + ";\n}\n\n")
    fout.write("hierarchicalCoeffs\n{\nn␣" + nstr + ";\ndelta␣" + delta +
               ";\norder␣xyz;\n}\n\n")

    # close output
    fout.flush()
    os.fsync(fout.fileno())
    fout.close()

    return None


def fvSchemes(ddt="steadyState", grad="Gauss", gradi="linear", div="Gauss",
              divi="linear", lap="Gauss", lapi="linear", lapsn="corrected",
              inter="linear", sng="corrected"):
    """
␣␣␣␣Creates␣a␣finite␣volume␣schemes␣dictionary␣in␣the␣case/system␣directory.
␣␣␣␣Only␣sets␣default␣parameters␣to␣be␣solution␣neutral,
␣␣␣␣any␣further␣tweaking␣should␣be␣done␣manually.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣ddt:␣␣␣␣string␣␣(default␣"steadyState")
␣␣␣␣␣␣␣␣Time␣discretisation␣scheme␣(ddtSchemes).
␣␣␣␣grad:␣␣␣string␣␣(default␣"Gauss")
␣␣␣␣␣␣␣␣Gradient␣calculation␣scheme␣(gradSchemes).
␣␣␣␣gradi:␣␣string␣␣(default␣"linear")
␣␣␣␣␣␣␣␣Gradient␣calculation␣interpolation␣scheme␣(gradSchemes).
␣␣␣␣div:␣␣␣␣string␣␣(default␣"Gauss")
␣␣␣␣␣␣␣␣Divergence␣calculation␣gradient␣scheme␣(divSchemes).
␣␣␣␣divi:␣␣␣string␣␣(default␣"linear")
␣␣␣␣␣␣␣␣Divergence␣calculation␣interpolation␣scheme␣(divSchemes).
␣␣␣␣lap:␣␣␣␣string␣␣(default␣"Gauss")
␣␣␣␣␣␣␣␣Laplacian␣calculation␣gradient␣scheme␣(laplacianSchemes).
␣␣␣␣lapi:␣␣␣string␣␣(default␣"linear")
␣␣␣␣␣␣␣␣Laplacian␣calculation␣interpolation␣scheme␣(laplacianSchemes).
␣␣␣␣lapsn:␣␣string␣␣(default␣"corrected")
␣␣␣␣␣␣␣␣Laplacian␣calculation␣surface␣normal␣scheme␣(laplacianSchemes).
␣␣␣␣inter:␣␣string␣␣(default␣"linear")
␣␣␣␣␣␣␣␣Interpolation␣scheme␣(interpolationSchemes).
␣␣␣␣sng:␣␣␣␣string␣␣(default␣"corrected")
␣␣␣␣␣␣␣␣Surface␣normal␣gradient␣scheme␣(snGradSchemes).

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣None
␣␣␣␣␣␣␣␣If␣successful.
␣␣␣␣"""

    fout = file("system/fvSchemes", "w")

    # write file header
    fout.write("FoamFile\n{\n")
    fout.write("␣␣version␣␣2.0;\n")
    fout.write("␣␣format␣␣␣ascii;\n")
    fout.write("␣␣class␣␣␣␣dictionary;\n")
    fout.write("␣␣object␣␣␣fvSchemes;\n")
    fout.write("␣␣location␣\"system\";\n")
    fout.write("}\n\n")

    # write ddt
    fout.write("ddtSchemes\n{\n␣␣default␣␣" + ddt + ";\n}\n\n")

    # write grad
    fout.write("gradSchemes\n{\n␣␣default␣␣" + grad + "␣" + gradi + ";\n}\n\n")

    # write div
    fout.write("divSchemes\n{\n␣␣default␣␣" + div + "␣" + divi + ";\n}\n\n")

    # write laplacian
    fout.write("laplacianSchemes\n{\n␣␣default␣␣" + lap + "␣" + lapi + "␣"
               + lapsn + ";\n}\n\n")

    # write interpolation
    fout.write("interpolationSchemes\n{\n␣␣default␣␣" + inter + ";\n}\n\n")

    # write snGrad
    fout.write("snGradSchemes\n{\n␣␣default␣␣" + sng + ";\n}\n\n")

    # write flux required
    fout.write("fluxRequired\n{\n␣␣default␣␣no;\n␣␣p;\n}\n\n")

    # close output
    fout.flush()
    os.fsync(fout.fileno())
    fout.close()

    return None


def fvSolution(gamg=True, abstol=1e-8, reltol=0.01, ncor=2, nonorth=0,
               prefcell=0, prefval=0):
    """
␣␣␣␣Creates␣a␣finite␣volume␣solution␣dictionary␣in␣the␣case/system␣directory.
␣␣␣␣To␣be␣solution␣neutral,␣makes␣some␣assumptions,
␣␣␣␣dictionary␣may␣need␣further␣modification␣after␣creation.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣gamg:␣␣␣boolean␣(default␣True)
␣␣␣␣␣␣␣␣Use␣GAMG␣for␣pressure␣if␣True,␣DICPCG␣if␣False.
␣␣␣␣abstol:␣number␣␣(default␣1e-8)
␣␣␣␣␣␣␣␣Absolute␣tolerance␣for␣all␣residuals.
␣␣␣␣reltol:␣number␣␣(default␣0.01)
␣␣␣␣␣␣␣␣Relative␣tolerance␣per␣timestep␣for␣all␣residuals.
␣␣␣␣ncor:␣␣␣integer␣(default␣2)
␣␣␣␣␣␣␣␣nCorrectors␣for␣PISO.
␣␣␣␣nonorth:␣␣␣␣integer␣(default␣0)
␣␣␣␣␣␣␣␣nNonOrthogonalCorrectors␣for␣PISO␣and␣SIMPLE.
␣␣␣␣prefcell:␣␣␣integer␣(default␣0)
␣␣␣␣␣␣␣␣pRefCell␣for␣PISO␣and␣SIMPLE.
␣␣␣␣prefval:␣␣␣␣number␣␣(default␣0)
␣␣␣␣␣␣␣␣pRefValue␣for␣PISO.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣None
␣␣␣␣␣␣␣␣If␣successful.
␣␣␣␣"""

    fout = file("system/fvSolution", "w")

    # convert numerical inputs into strings
    at = str(abstol)
    rt = str(reltol)
    nc = str(ncor)
    no = str(nonorth)
    pc = str(prefcell)
    pv = str(prefval)

    # write file header
    fout.write("FoamFile\n{\n")
    fout.write("␣␣version␣␣2.0;\n")
    fout.write("␣␣format␣␣␣ascii;\n")
    fout.write("␣␣class␣␣␣␣dictionary;\n")
    fout.write("␣␣object␣␣␣fvSolution;\n")
    fout.write("␣␣location␣\"system\";\n")
    fout.write("}\n\n")

    # write solvers
    fout.write("solvers\n{\n␣␣p\n␣␣{\n")
    if gamg:
        fout.write("␣␣␣␣solver␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣GAMG;\n")
        fout.write("␣␣␣␣smoother␣␣␣␣␣␣␣␣␣␣␣␣␣␣GaussSeidel;\n")
        fout.write("␣␣␣␣cacheAgglomeration␣␣␣␣on;\n")
        fout.write("␣␣␣␣agglomerator␣␣␣␣␣␣␣␣␣␣faceAreaPair;\n")
        fout.write("␣␣␣␣nCellsInCoarsestLevel␣100;\n")
        fout.write("␣␣␣␣mergeLevels␣␣␣␣␣␣␣␣␣␣␣1;\n")
        fout.write("␣␣␣␣tolerance␣␣␣␣␣␣␣␣␣␣␣␣␣"+at+";\n")
        fout.write("␣␣␣␣relTol␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"+rt+";\n")
    else:
        fout.write("␣␣␣␣solver␣␣␣␣␣␣␣␣␣PCG;\n")
        fout.write("␣␣␣␣preconditioner␣DIC;\n")
        fout.write("␣␣␣␣tolerance␣␣␣␣␣␣"+at+";\n")
        fout.write("␣␣␣␣relTol␣␣␣␣␣␣␣␣␣"+rt+";\n")
    fout.write("␣␣}\n\n")
    # U
    fout.write("␣␣U\n␣␣{\n")
    fout.write("␣␣␣␣solver␣␣␣␣␣␣␣␣␣PBiCG;\n")
    fout.write("␣␣␣␣preconditioner␣DILU;\n")
    fout.write("␣␣␣␣tolerance␣␣␣␣␣␣"+at+";\n")
    fout.write("␣␣␣␣relTol␣␣␣␣␣␣␣␣␣"+rt+";\n")
    fout.write("␣␣}\n\n")
    # k
    fout.write("␣␣k\n␣␣{\n")
    fout.write("␣␣␣␣solver␣␣␣␣␣␣␣␣␣PBiCG;\n")
    fout.write("␣␣␣␣preconditioner␣DILU;\n")
    fout.write("␣␣␣␣tolerance␣␣␣␣␣␣"+at+";\n")
    fout.write("␣␣␣␣relTol␣␣␣␣␣␣␣␣␣"+rt+";\n")
    fout.write("␣␣}\n\n")
    # epsilon
    fout.write("␣␣epsilon\n␣␣{\n")
    fout.write("␣␣␣␣solver␣␣␣␣␣␣␣␣␣PBiCG;\n")
    fout.write("␣␣␣␣preconditioner␣DILU;\n")
    fout.write("␣␣␣␣tolerance␣␣␣␣␣␣"+at+";\n")
    fout.write("␣␣␣␣relTol␣␣␣␣␣␣␣␣␣"+rt+";\n")
    fout.write("␣␣}\n\n")
    # omega
    fout.write("␣␣omega\n␣␣{\n")
    fout.write("␣␣␣␣solver␣␣␣␣␣␣␣␣␣PBiCG;\n")
    fout.write("␣␣␣␣preconditioner␣DILU;\n")
    fout.write("␣␣␣␣tolerance␣␣␣␣␣␣"+at+";\n")
    fout.write("␣␣␣␣relTol␣␣␣␣␣␣␣␣␣"+rt+";\n")
    fout.write("␣␣}\n\n")
    fout.write("}\n\n")

    # write PISO
    fout.write("PISO\n{\n")
    fout.write("␣␣nCorrectors␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"+nc+";\n")
    fout.write("␣␣nNonOrthogonalCorrectors␣"+no+";\n")
    fout.write("␣␣pRefCell␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"+pc+";\n")
    fout.write("␣␣pRefValue␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"+pv+";\n")
    fout.write("}\n\n")

    # write SIMPLE
    fout.write("SIMPLE\n{\n")
    fout.write("␣␣nNonOrtogonalCorrectors␣"+no+";\n")
    fout.write("␣␣pRefCell␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"+pc+";\n")
    fout.write("␣␣pRefValue␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"+pv+";\n")
    fout.write("}\n\n")

    # write relaxationFactors
    fout.write("relaxationFactors\n{\n")
    fout.write("␣␣p␣␣␣␣␣␣␣␣0.3;\n")
    fout.write("␣␣U␣␣␣␣␣␣␣␣0.7;\n")
    fout.write("␣␣k␣␣␣␣␣␣␣␣0.7;\n")
    fout.write("␣␣epsilon␣␣0.7;\n")
    fout.write("␣␣omega␣␣␣␣0.7;\n")
    fout.write("}\n\n")

    # close output
    fout.flush()
    os.fsync(fout.fileno())
    fout.close()

    return None


def snappyHexMeshDict(stls, lvls, loc,
                      mainpars=["true", "true", "true", "1e-06", "0"],
                      castpars=["2000000", "2000000", "5", "1", "30"],
                      snappars=["3", "1.0", "30", "5"],
                      lay1pars=["3", "true", "1.2", "0.3", "0.1"],
                      lay2pars=["1", "90", "5", "1", "3", "10", "0.5", "0.3",
                                "130", "0", "30", "15"],
                      mq=["60", "20", "5", "80", "0.5", "1e-15", "-1",
                          "0.02", "0.001", "0.02", "0.02", "-1", "4", "0.85"],
                      mqr=["90", "30", "8", "120", "0.5", "1e-30", "-1",
                           "0.01", "0.0001", "0.01", "0.01", "-1", "6", "0.8"],
                      MRFrotor=False, features=False):
    """
␣␣␣␣Creates␣a␣snappyHexMesh␣dictionary␣in␣the␣case/system␣directory.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣stls:␣␣␣list␣of␣strings
␣␣␣␣␣␣␣␣List␣of␣strings␣naming␣required␣.stl␣files.
␣␣␣␣lvls:␣␣␣list␣of␣strings
␣␣␣␣␣␣␣␣List␣of␣strings␣giving␣min␣and␣max␣\"(min␣max)\"
␣␣␣␣␣␣␣␣surface␣refinement␣levels␣for␣each␣.stl␣surface.
␣␣␣␣loc:␣␣␣␣list␣of␣numbers
␣␣␣␣␣␣␣␣Co-ordinates␣of␣a␣location␣within␣the␣meshed␣region.

␣␣␣␣Optional␣Parameters
␣␣␣␣-------------------
␣␣␣␣mainpars:␣␣␣list␣of␣strings
␣␣␣␣␣␣␣␣List␣of␣strings␣for␣the␣main␣parameters␣in␣the␣following␣order:
␣␣␣␣␣␣␣␣castellatedMesh,snap,addLayers,mergeTolerance,debug.
␣␣␣␣castpars:␣␣␣list␣of␣string
␣␣␣␣␣␣␣␣List␣of␣strings␣for␣the␣castellated␣mesh␣parameters:
␣␣␣␣␣␣␣␣maxLocalCells,maxGlobalCells,minRefinementCells,
␣␣␣␣␣␣␣␣nCellsBetweenLevels,resolveFeatureAngle.
␣␣␣␣snappars:␣␣␣list␣of␣strings
␣␣␣␣␣␣␣␣List␣of␣strings␣for␣the␣mesh␣snapping␣parameters:
␣␣␣␣␣␣␣␣nSmoothPatch,tolerance,nSolveIter,nRelaxIter.
␣␣␣␣lay1pars:␣␣␣list␣of␣strings
␣␣␣␣␣␣␣␣List␣of␣strings␣for␣layer␣addition␣parameters:
␣␣␣␣␣␣␣␣nSurfaceLayers,relativeSizes,expansionRatio,
        finalLayerThickness, minThickness.
    lay2pars:   list of strings
        List of strings for other layer addition parameters:
        nGrow, featureAngle, nRelaxIter, nSmoothSurfaceNormals,
        nSmoothNormals, nSmoothThickness, maxFaceThicknessRatio,
        maxThicknessToMedialRatio, minMedianAxisAngle,
        nBufferCellsNoExtrude, nLayerIter, nRelaxedIter
    mq:    list of strings
        List of strings for mesh quality controls:
        maxNonOrtho, maxBoundarySkewness, maxInternalSkewness,
        maxConcave, minFlatness, minVol, minArea, minTwist,
        minDeterminant, minFaceWeight, minVolRatio, minTriangleTwist,
        nSmoothScale, errorReduction.
    mqr:    list of strings
        List of strings for relaxed mesh quality controls:
        maxNonOrtho, maxBoundarySkewness, maxInternalSkewness,
        maxConcave, minFlatness, minVol, minArea, minTwist,
        minDeterminant, minFaceWeight, minVolRatio, minTriangleTwist,
        nSmoothScale, errorReduction.
    MRFrotor:   list of [list of numbers, list of numbers, number]
        Add optional cylinder zone called rotor, defined with a list of
        three parameters, one (list of nums) giving the first
        co-ordinate of the cylinder, one (list of nums) giving the
        second co-ordinate of the cylinder and the third the radius
        of the cylinder of the rotating region.
    features:   integer
        When not False, uses the stls to create eMesh files
        via the surfaceFeatureExtract application and then uses
        feature snapping to improve the mesh surface capture
        with features number of iterations.

    Returns
    -------
    None
        If successful.
    """

    fout = file("system/snappyHexMeshDict", "w")

    # convert numerical inputs into strings
    locstr = "("+str(loc[0])+" "+str(loc[1])+" "+str(loc[2])+")"
    if MRFrotor:
        cyl1 = "(" + str(MRFrotor[0][0]) + \
               " " + str(MRFrotor[0][1]) + \
               " " + str(MRFrotor[0][2]) + ")"
        cyl2 = "(" + str(MRFrotor[1][0]) + \
               " " + str(MRFrotor[1][1]) + \
               " " + str(MRFrotor[1][2]) + ")"
        rad = str(MRFrotor[2])

    # write file header
    fout.write("FoamFile\n{\n")
    fout.write("  version     2.0;\n")
    fout.write("  format      ascii;\n")
    fout.write("  class       dictionary;\n")
    fout.write("  object      snappyHexMeshDict;\n")
    fout.write("  location    \"system\";\n}\n\n")

    # write main entries
    fout.write("castellatedMesh " + mainpars[0] + ";\n")
    fout.write("snap            " + mainpars[1] + ";\n")
    fout.write("addLayers       " + mainpars[2] + ";\n")
    fout.write("mergeTolerance  " + mainpars[3] + ";\n")
    fout.write("debug           " + mainpars[4] + ";\n\n")

    # write geometry sub-dictionary
    fout.write("geometry\n{")
    for stl in stls:
        fout.write("\n  " + stl + "\n  {\n  type triSurfaceMesh;\n  }\n")
    if MRFrotor:
        fout.write("  rotor\n  {\n  type   searchableCylinder;\n")
        fout.write("  point1 " + cyl1 + ";\n  point2 " + cyl2 + ";\n  radius "
                   + rad + ";\n  }\n")
    fout.write("}\n\n")

    # write castellatedMesh sub-dictionary
    fout.write("castellatedMeshControls\n{\n")
    fout.write("  locationInMesh       " + locstr + ";\n")
    fout.write("  maxLocalCells        " + castpars[0] + ";\n")
    fout.write("  maxGlobalCells       " + castpars[1] + ";\n")
    fout.write("  minRefinementCells   " + castpars[2] + ";\n")
    fout.write("  nCellsBetweenLevels  " + castpars[3] + ";\n")
    fout.write("  resolveFeatureAngle  " + castpars[4] + ";\n")
    fout.write("  features\n  (\n")
    if features:
        for stl in stls:
            fout.write("  { file \"" + stl[:-4] + ".eMesh\"; level 0; }\n")
    fout.write("  );\n")
    fout.write("  refinementSurfaces\n  {")
    for i in range(len(stls)):
        fout.write("\n    " + stls[i] + "\n    {\n      level " + lvls[i] +
                   ";\n    }\n")
```

```
if MRFrotor:
    fout.write("␣␣␣␣rotor\n␣␣␣␣{\n␣␣␣␣␣␣level␣(0␣0);\n" +
               "␣␣␣␣␣␣faceZone␣rotor;\n" +
               "␣␣␣␣␣␣cellZone␣rotor;\n␣␣␣␣␣␣zoneInside␣true;\n␣␣␣␣}\n")
fout.write("␣␣}\n")
fout.write("␣␣allowFreeStandingZoneFaces␣␣false;\n")
fout.write("␣␣refinementRegions\n␣␣{\n␣␣}\n}\n\n")

# write snapControls sub-dictionary
fout.write("snapControls\n{\n")
fout.write("␣␣nSmoothPatch␣␣" + snappars[0] + ";\n")
fout.write("␣␣tolerance␣␣␣␣␣" + snappars[1] + ";\n")
fout.write("␣␣nSolveIter␣␣␣␣" + snappars[2] + ";\n")
fout.write("␣␣nRelaxIter␣␣␣␣" + snappars[3] + ";\n")
if features:
    fout.write("␣␣nFeatureSnapIter␣"+str(features)+";\n")
fout.write("}\n\n")

# write addLayers sub-dictionary part 1
fout.write("addLayersControls\n{\n␣␣layers\n␣␣{\n")
for stl in stls:
    fout.write("␣␣␣␣" + stl + "␣" + stl[:-4] +
               "\n␣␣␣␣{␣␣␣␣␣␣nSurfaceLayers␣␣" + lay1pars[0] +
               ";\n␣␣␣␣}\n")
fout.write("␣␣}\n")
fout.write("␣␣relativeSizes␣␣␣␣␣␣␣␣" + lay1pars[1] + ";\n")
fout.write("␣␣expansionRatio␣␣␣␣␣␣␣" + lay1pars[2] + ";\n")
fout.write("␣␣finalLayerThickness␣␣" + lay1pars[3] + ";\n")
fout.write("␣␣minThickness␣␣␣␣␣␣␣␣␣" + lay1pars[4] + ";\n")

# write addLayers sub-dictionary part 2
fout.write("␣␣nGrow␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣" + lay2pars[0] + ";\n")
fout.write("␣␣featureAngle␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣" + lay2pars[1] + ";\n")
fout.write("␣␣nRelaxIter␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣" + lay2pars[2] + ";\n")
fout.write("␣␣nSmoothSurfaceNormals␣␣␣␣␣␣" + lay2pars[3] + ";\n")
fout.write("␣␣nSmoothNormals␣␣␣␣␣␣␣␣␣␣␣␣␣" + lay2pars[4] + ";\n")
fout.write("␣␣nSmoothThickness␣␣␣␣␣␣␣␣␣␣␣" + lay2pars[5] + ";\n")
fout.write("␣␣maxFaceThicknessRatio␣␣␣␣␣␣" + lay2pars[6] + ";\n")
fout.write("␣␣maxThicknessToMedialRatio␣␣" + lay2pars[7] + ";\n")
fout.write("␣␣minMedianAxisAngle␣␣␣␣␣␣␣␣␣" + lay2pars[8] + ";\n")
fout.write("␣␣nBufferCellsNoExtrude␣␣␣␣␣␣" + lay2pars[9] + ";\n")
fout.write("␣␣nLayerIter␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣" + lay2pars[10] + ";\n")
fout.write("␣␣nRelaxedIter␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣" + lay2pars[11] + ";\n}\n\n")

# write mesh quality controls
fout.write("meshQualityControls\n{\n")
fout.write("␣␣maxNonOrtho␣␣␣␣␣␣␣␣␣␣" + mq[0] + ";\n")
fout.write("␣␣maxBoundarySkewness␣␣" + mq[1] + ";\n")
fout.write("␣␣maxInternalSkewness␣␣" + mq[2] + ";\n")
fout.write("␣␣maxConcave␣␣␣␣␣␣␣␣␣␣␣" + mq[3] + ";\n")
fout.write("␣␣minFlatness␣␣␣␣␣␣␣␣␣␣" + mq[4] + ";\n")
fout.write("␣␣minVol␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣" + mq[5] + ";\n")
fout.write("␣␣minArea␣␣␣␣␣␣␣␣␣␣␣␣␣␣" + mq[6] + ";\n")
fout.write("␣␣minTwist␣␣␣␣␣␣␣␣␣␣␣␣␣" + mq[7] + ";\n")
fout.write("␣␣minDeterminant␣␣␣␣␣␣␣" + mq[8] + ";\n")
fout.write("␣␣minFaceWeight␣␣␣␣␣␣␣␣" + mq[9] + ";\n")
fout.write("␣␣minVolRatio␣␣␣␣␣␣␣␣␣␣" + mq[10] + ";\n")
fout.write("␣␣minTriangleTwist␣␣␣␣␣" + mq[11] + ";\n")
fout.write("␣␣nSmoothScale␣␣␣␣␣␣␣␣␣" + mq[12] + ";\n")
fout.write("␣␣errorReduction␣␣␣␣␣␣␣" + mq[13] + ";\n")
fout.write("␣␣minTetQuality␣␣␣␣␣␣␣␣1e-30;\n")

# write relaxed mesh quality controls
fout.write("␣␣relaxed\n␣␣{\n")
fout.write("␣␣maxNonOrtho␣␣␣␣␣␣␣␣␣␣" + mqr[0] + ";\n")
fout.write("␣␣maxBoundarySkewness␣␣" + mqr[1] + ";\n")
fout.write("␣␣maxInternalSkewness␣␣" + mqr[2] + ";\n")
fout.write("␣␣maxConcave␣␣␣␣␣␣␣␣␣␣␣" + mqr[3] + ";\n")
fout.write("␣␣minFlatness␣␣␣␣␣␣␣␣␣␣" + mqr[4] + ";\n")
fout.write("␣␣minVol␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣" + mqr[5] + ";\n")
fout.write("␣␣minArea␣␣␣␣␣␣␣␣␣␣␣␣␣␣" + mqr[6] + ";\n")
fout.write("␣␣minTwist␣␣␣␣␣␣␣␣␣␣␣␣␣" + mqr[7] + ";\n")
fout.write("␣␣minDeterminant␣␣␣␣␣␣␣" + mqr[8] + ";\n")
fout.write("␣␣minFaceWeight␣␣␣␣␣␣␣␣" + mqr[9] + ";\n")
fout.write("␣␣minVolRatio␣␣␣␣␣␣␣␣␣␣" + mqr[10] + ";\n")
fout.write("␣␣minTriangleTwist␣␣␣␣␣" + mqr[11] + ";\n")
fout.write("␣␣nSmoothScale␣␣␣␣␣␣␣␣␣" + mqr[12] + ";\n")
fout.write("␣␣errorReduction␣␣␣␣␣␣␣" + mqr[13] + ";\n")
fout.write("␣␣minTetQuality␣␣␣␣␣␣␣␣1e-30;\n")
fout.write("␣␣}\n}\n\n")

# close output
fout.flush()
os.fsync(fout.fileno())
fout.close()

# if features is enabled, create eMesh files
if features:
    for stl in stls:
        run("surfaceFeatureExtract␣-includedAngle␣150␣constant/triSurface/"
            + stl + "␣" + stl[:-4], silent=True)
    return None


def MRFZones(region, rpm, origin=[0, 0, 0], axis=[0, 0, 1], nrp=""):
    """
    Creates a MRF (multiple reference frame) zones dictionary
    in the case/constant directory.

    Parameters
    ----------
    region: string
        Name of the rotating region.
    rpm:    number
        Speed of rotation in rpm.
    origin: list of numbers (default [0, 0, 0])
        Centre of rotation.
    axis:   list of numbers (default [0, 0, 1])
        Axis about which the region rotates.
    nrp:    string  (default "")
        nonRotatingPatches separated by a space.

    Returns
    -------
    None
        If successful.
    """
    from math import pi

    fout = file("constant/MRFZones", "w")

    # convert rpm into rad/s
    omega = str(pi*rpm/30.0)

    # convert numerical inputs into strings
    originstr = "("+str(origin[0])+" "+str(origin[1])+" "+str(origin[2])+")"
    axisstr = "("+str(axis[0])+" "+str(axis[1])+" "+str(axis[2])+")"

    # write file header
    fout.write("FoamFile\n{\n")
    fout.write("  version   2.0;\n")
    fout.write("  format    ascii;\n")
    fout.write("  class     dictionary;\n")
    fout.write("  object    MRFZones;\n")
    fout.write("  location  \"constant\";\n}\n\n")

    # write remainder of file
    fout.write("1\n(\n  "+region+"\n  {\n")
    fout.write("    nonRotatingPatches  ("+nrp+");\n\n")
    fout.write("    origin  origin [0 1 0 0 0 0 0]  "+originstr+";\n")
    fout.write("    axis    axis   [0 0 0 0 0 0 0]  "+axisstr+";\n")
    fout.write("    omega   omega [0 0 -1 0 0 0 0]  "+omega+";\n  }\n)\n\n")

    # close output
    fout.flush()
    os.fsync(fout.fileno())
    fout.close()

    return None


def RASProperties(model, turb=True, coef=True):
    """
    Creates a RAS (Reynolds averaged simulation) properties dictionary
    in the case/constant directory.

    Parameters
    ----------
    model:  string
        Turbulence model to use.
    turb:   boolean
        Turbulence on or off.
    coef:   boolean
        Print coefficients?

    Returns
    -------
    None
        If successful.
    """

    fout = file("constant/RASProperties", "w")

    # write file header
    fout.write("FoamFile\n{\n")
    fout.write("  version   2.0;\n")
    fout.write("  format    ascii;\n")
    fout.write("  class     dictionary;\n")
    fout.write("  object    RASProperties;\n")
    fout.write("  location  \"constant\";\n}\n\n")

    # write data
    fout.write("RASModel     "+model+";\n\n")
    if turb:
        fout.write("turbulence   on;\n\n")
    else:
        fout.write("turbulence   off;\n\n")
    if coef:
        fout.write("printCoeffs  on;\n\n")
    else:
        fout.write("printCoeffs  off;\n\n")

    # close output
    fout.flush()
    os.fsync(fout.fileno())
    fout.close()

    return None


def transportProperties(nu=1.307e-6):
    """
    Creates a transport properties dictionary in the case/constant directory.
    Currently defaults to a Newtonian transport model and takes one input
    of dynamic viscosity, defaulting to that of sea water.

    Parameters
    ----------
    nu: number  (default 1.307e-6)
        Kinematic viscosity in m^2/s.

    Returns
    -------
    None
        If successful.
    """

    fout = file("constant/transportProperties", "w")

    # write file header
    fout.write("FoamFile\n{\n")
    fout.write("  version   2.0;\n")
    fout.write("  format    ascii;\n")
    fout.write("  class     dictionary;\n")
    fout.write("  object    transportProperties;\n")
    fout.write("  location  \"constant\";\n}\n\n")

    # write Newtonian transport
    fout.write("transportModel         Newtonian;\n\n")
    fout.write("nu nu [0 2 -1 0 0 0 0] "+str(nu)+";\n\n")

    # close output
    fout.flush()
    os.fsync(fout.fileno())
    fout.close()

    return None


def makeFieldsiKO(Vinit, turbulentIntensity=0.1, turbulentViscosityRatio=10,
                  wallFunctions=True, inlet_pressure=0.1, rot_omega=314.159):
    """
    Creates field files for an incompressible k-omega simulation,
    patches are defined based on their type
    in the constant/polyMesh/boundary file.
    Patches named inlet and outlet
    will be treated as inlet and outlet boundaries.

    Parameters
    ----------
    Vinit:  list of numbers
        Initial/inlet velocity.
    turbulentIntensity: number  (default 0.1)
        Turbulent intensity.
    turbulentViscosityRaio: number  (default 10)
        Turbulent viscosity ratio nut/nu.
    wallFunctions:  boolean (default True)
        Use wall functions?
    inlet_pressure: number  (default 0.1)
        Inlet pressure for pinlet.
    rot_omega:  number  (default 314.159)
        Omega for rotXYZ patches.

    Returns
    -------
    None
        If successful.
    """
    from numpy import sqrt

    # convert numerical inputs into strings
    if isinstance(Vinit, str):
        Vinitstr = Vinit
        Vinit = [float(Vinitstr.split()[0][1:]), float(Vinitstr.split()[1]),
                 float(Vinitstr.split()[2][:-1])]
```

```
else:
    Vinitstr = "("+str(Vinit[0])+"␣"+str(Vinit[1])+"␣"+str(Vinit[2])+")"

# if turbulent intensity is specified as percentage convert into decimal
if turbulentIntensity > 1:
    turbulentIntensity = turbulentIntensity/100.0

# read in boundary file
fin = file("constant/polyMesh/boundary", "r")
boundarywords = fin.read().split()
fin.close()

# extract patches dictionary
patches = {}
for i in range(len(boundarywords)-3):
    if boundarywords[i+1] == "{" and boundarywords[i+2] == "type":
        patches[boundarywords[i]] = boundarywords[i+3]

# read in viscosity from transportProperties dictionary
try:
    fin2 = file("constant/transportProperties", "r")
# this is in case it's a parallel directory
except IOError:
    fin2 = file("../constant/transportProperties", "r")
transportwords = fin2.read().split()
fin2.close()
for i in range(len(transportwords)):
    if transportwords[i] == "0]":
        nu = float(transportwords[i+1][:-1])

# calculate derived quantities
Vmag = sqrt(Vinit[0]*Vinit[0]+Vinit[1]*Vinit[1]+Vinit[2]*Vinit[2])
k = (turbulentIntensity*Vmag)**2
tkestr = str(k)
tvr = turbulentViscosityRatio/100.0
omega = max([0.00001, k/float(tvr*nu)])
omegastr = str(omega)

# open output files
fp = file("0/p", "w")
fU = file("0/U", "w")
fk = file("0/k", "w")
fo = file("0/omega", "w")
fn = file("0/nut", "w")

# write headers
# pressure
fp.write("FoamFile\n{\n")
fp.write("␣␣version␣␣␣2.0;\n")
fp.write("␣␣format␣␣␣␣ascii;\n")
fp.write("␣␣class␣␣␣␣␣volScalarField;\n")
fp.write("␣␣object␣␣␣␣p;\n")
fp.write("␣␣location␣␣\"0\";\n}\n\n")
# velocity
fU.write("FoamFile\n{\n")
fU.write("␣␣version␣␣␣2.0;\n")
fU.write("␣␣format␣␣␣␣ascii;\n")
fU.write("␣␣class␣␣␣␣␣volVectorField;\n")
fU.write("␣␣object␣␣␣␣U;\n")
fU.write("␣␣location␣␣\"0\";\n}\n\n")
# turbulent kinetic energy
fk.write("FoamFile\n{\n")
fk.write("␣␣version␣␣␣2.0;\n")
fk.write("␣␣format␣␣␣␣ascii;\n")
fk.write("␣␣class␣␣␣␣␣volScalarField;\n")
fk.write("␣␣object␣␣␣␣k;\n")
fk.write("␣␣location␣␣\"0\";\n}\n\n")
# specific turbulent kinetic energy dissipation
fo.write("FoamFile\n{\n")
fo.write("␣␣version␣␣␣2.0;\n")
fo.write("␣␣format␣␣␣␣ascii;\n")
fo.write("␣␣class␣␣␣␣␣volScalarField;\n")
fo.write("␣␣object␣␣␣␣omega;\n")
fo.write("␣␣location␣␣\"0\";\n}\n\n")
# turbulent viscosity
fn.write("FoamFile\n{\n")
fn.write("␣␣version␣␣␣2.0;\n")
fn.write("␣␣format␣␣␣␣ascii;\n")
fn.write("␣␣class␣␣␣␣␣volScalarField;\n")
fn.write("␣␣object␣␣␣␣nut;\n")
fn.write("␣␣location␣␣\"0\";\n}\n\n")

# write initial conditions
# pressure
if "pinlet" in patches:
    fp.write("dimensions␣␣␣␣␣[0␣2␣-2␣0␣0␣0␣0];\n\n")
    fp.write("internalField␣␣uniform␣{};\n\n".format(0.5*inlet_pressure))
else:
    fp.write("dimensions␣␣␣␣␣[0␣2␣-2␣0␣0␣0␣0];\n\n")
    fp.write("internalField␣␣uniform␣0;\n\n")
# velocity
fU.write("dimensions␣␣␣␣␣[0␣1␣-1␣0␣0␣0␣0];\n\n")
fU.write("internalField  uniform "+Vinitstr+";\n\n")
# turbulent kinetic energy
fk.write("dimensions     [0 2 -2 0 0 0 0];\n\n")
fk.write("internalField  uniform "+tkestr+";\n\n")
# specific turbulent kinetic energy dissipation
fo.write("dimensions     [0 0 -1 0 0 0 0];\n\n")
fo.write("internalField  uniform "+omegastr+";\n\n")
# turbulent viscosity
fn.write("dimensions     [0 2 -1 0 0 0 0];\n\n")
fn.write("internalField  uniform 0;\n\n")

# write boundary conditions
# begin writing boundaryFields
fp.write("boundaryField\n{\n")
fU.write("boundaryField\n{\n")
fk.write("boundaryField\n{\n")
fo.write("boundaryField\n{\n")
fn.write("boundaryField\n{\n")
for patch in patches:
    # write patch common
    fp.write("  "+patch+"\n  {\n")
    fU.write("  "+patch+"\n  {\n")
    fk.write("  "+patch+"\n  {\n")
    fo.write("  "+patch+"\n  {\n")
    fn.write("  "+patch+"\n  {\n")
    if patch == "inlet":
        # inlet boundary
        fp.write("    type   zeroGradient;\n")
        fp.write("    value  uniform 0;\n")
        fU.write("    type   fixedValue;\n")
        fU.write("    value  uniform "+Vinitstr+";\n")
        fk.write("    type   fixedValue;\n")
        fk.write("    value  uniform "+tkestr+";\n")
        fo.write("    type   fixedValue;\n")
        fo.write("    value  uniform "+omegastr+";\n")
        fn.write("    type   calculated;\n")
        fn.write("    value  uniform 0;\n")
    elif patch == "pinlet":
        # pressure inlet boundary
        fp.write("    type   fixedValue;\n")
        fp.write("    value  uniform {};\n".format(inlet_pressure))
        fU.write("    type   pressureInletVelocity;\n")
        fU.write("    value  uniform "+Vinitstr+";\n")
        fk.write("    type   fixedValue;\n")
        fk.write("    value  uniform "+tkestr+";\n")
        fo.write("    type   fixedValue;\n")
        fo.write("    value  uniform "+omegastr+";\n")
        fn.write("    type   calculated;\n")
        fn.write("    value  uniform 0;\n")
    elif patch[:4] == "rotX":
        # rotating wall about X boundary
        fp.write("    type   zeroGradient;\n")
        fp.write("    value  uniform 0;\n")
        fU.write("    type   rotatingWallVelocity;\n")
        fU.write("    origin (0 0 0);\n")
        fU.write("    axis   (1 0 0);\n")
        fU.write("    omega  constant {};\n".format(rot_omega))
        if wallFunctions:
            fk.write("    type   kqRWallFunction;\n")
            fk.write("    value  uniform "+tkestr+";\n")
            fo.write("    type   omegaWallFunction;\n")
            fo.write("    value  uniform "+omegastr+";\n")
            fn.write("    type   nutkWallFunction;\n")
            fn.write("    value  uniform 0;\n")
        if not wallFunctions:
            fk.write("    type   fixedValue;\n")  # TBC
            fk.write("    value  uniform 0;\n")  # TBC
            fo.write("    type   fixedValue;\n")  # TBC
            fo.write("    value  uniform 0;\n")  # TBC
            fn.write("    type   calculated;\n")  # TBC
            fn.write("    value  uniform 0;\n")  # TBC
    elif patch[:4] == "rotY":
        # rotating wall about Y boundary
        fp.write("    type   zeroGradient;\n")
        fp.write("    value  uniform 0;\n")
        fU.write("    type   rotatingWallVelocity;\n")
        fU.write("    origin (0 0 0);\n")
        fU.write("    axis   (0 1 0);\n")
        fU.write("    omega  constant {};\n".format(rot_omega))
        if wallFunctions:
            fk.write("    type   kqRWallFunction;\n")
            fk.write("    value  uniform "+tkestr+";\n")
            fo.write("    type   omegaWallFunction;\n")
            fo.write("    value  uniform "+omegastr+";\n")
            fn.write("    type   nutkWallFunction;\n")
            fn.write("    value  uniform 0;\n")
        if not wallFunctions:
            fk.write("    type   fixedValue;\n")  # TBC
            fk.write("    value  uniform 0;\n")  # TBC
            fo.write("    type   fixedValue;\n")  # TBC
            fo.write("    value  uniform 0;\n")  # TBC
            fn.write("    type   calculated;\n")  # TBC
            fn.write("␣␣␣␣value␣␣uniform␣0;\n")  # TBC
    elif patch[:4] == "rotZ":
        # rotating wall about X boundary
        fp.write("␣␣␣␣type␣␣␣zeroGradient;\n")
        fp.write("␣␣␣␣value␣␣uniform␣0;\n")
        fU.write("␣␣␣␣type␣␣␣rotatingWallVelocity;\n")
        fU.write("␣␣␣␣origin␣(0␣0␣0);\n")
        fU.write("␣␣␣␣axis␣␣␣(0␣0␣1);\n")
        fU.write("␣␣␣␣omega␣␣constant␣{};\n".format(rot_omega))
        if wallFunctions:
            fk.write("␣␣␣␣type␣␣␣kqRWallFunction;\n")
            fk.write("␣␣␣␣value␣␣uniform␣"+tkestr+";\n")
            fo.write("␣␣␣␣type␣␣␣omegaWallFunction;\n")
            fo.write("␣␣␣␣value␣␣uniform␣"+omegastr+";\n")
            fn.write("␣␣␣␣type␣␣␣nutkWallFunction;\n")
            fn.write("␣␣␣␣value␣␣uniform␣0;\n")
        if not wallFunctions:
            fk.write("␣␣␣␣type␣␣␣fixedValue;\n")  # TBC
            fk.write("␣␣␣␣value␣␣uniform␣0;\n")  # TBC
            fo.write("␣␣␣␣type␣␣␣fixedValue;\n")  # TBC
            fo.write("␣␣␣␣value␣␣uniform␣0;\n")  # TBC
            fn.write("␣␣␣␣type␣␣␣calculated;\n")  # TBC
            fn.write("␣␣␣␣value␣␣uniform␣0;\n")  # TBC
    elif patch == "outlet":
        # outlet boundary
        fp.write("␣␣␣␣type␣␣␣fixedValue;\n")
        fp.write("␣␣␣␣value␣␣uniform␣0;\n")
        fU.write("␣␣␣␣type␣␣␣zeroGradient;\n")
        fU.write("␣␣␣␣value␣␣uniform␣"+Vinitstr+";\n")
        fk.write("␣␣␣␣type␣␣␣zeroGradient;\n")
        fk.write("␣␣␣␣value␣␣uniform␣"+tkestr+";\n")
        fo.write("␣␣␣␣type␣␣␣zeroGradient;\n")
        fo.write("␣␣␣␣value␣␣uniform␣"+omegastr+";\n")
        fn.write("␣␣␣␣type␣␣␣zeroGradient;\n")
        fn.write("␣␣␣␣value␣␣uniform␣0;\n")
    elif patches[patch] == "wall;":
        # no slip wall boundary - using wall functions
        fp.write("␣␣␣␣type␣␣␣zeroGradient;\n")
        fp.write("␣␣␣␣value␣␣uniform␣0;\n")
        fU.write("␣␣␣␣type␣␣␣fixedValue;\n")
        fU.write("␣␣␣␣value␣␣uniform␣(0␣0␣0);\n")
        if wallFunctions:
            fk.write("␣␣␣␣type␣␣␣kqRWallFunction;\n")
            fk.write("␣␣␣␣value␣␣uniform␣"+tkestr+";\n")
            fo.write("␣␣␣␣type␣␣␣omegaWallFunction;\n")
            fo.write("␣␣␣␣value␣␣uniform␣"+omegastr+";\n")
            fn.write("␣␣␣␣type␣␣␣nutkWallFunction;\n")
            fn.write("␣␣␣␣value␣␣uniform␣0;\n")
        if not wallFunctions:
            fk.write("␣␣␣␣type␣␣␣fixedValue;\n")  # TBC
            fk.write("␣␣␣␣value␣␣uniform␣0;\n")  # TBC
            fo.write("␣␣␣␣type␣␣␣fixedValue;\n")  # TBC
            fo.write("␣␣␣␣value␣␣uniform␣0;\n")  # TBC
            fn.write("␣␣␣␣type␣␣␣calculated;\n")  # TBC
            fn.write("␣␣␣␣value␣␣uniform␣0;\n")  # TBC
    elif patches[patch] == "symmetryPlane;":
        # symmetry boundary
        fp.write("␣␣␣␣type␣␣␣symmetryPlane;\n")
        fp.write("␣␣␣␣value␣␣uniform␣0;\n")
        fU.write("␣␣␣␣type␣␣␣symmetryPlane;\n")
        fU.write("␣␣␣␣value␣␣uniform␣"+Vinitstr+";\n")
        fk.write("␣␣␣␣type␣␣␣symmetryPlane;\n")
        fk.write("␣␣␣␣value␣␣uniform␣"+tkestr+";\n")
        fo.write("␣␣␣␣type␣␣␣symmetryPlane;\n")
        fo.write("␣␣␣␣value␣␣uniform␣"+omegastr+";\n")
        fn.write("␣␣␣␣type␣␣␣symmetryPlane;\n")
        fn.write("␣␣␣␣value␣␣uniform␣0;\n")
    elif patches[patch] == "processor;":
        # processor boundary (for parallel)
        fp.write("␣␣␣␣type␣␣␣processor;\n")
        fp.write("␣␣␣␣value␣␣uniform␣0;\n")
        fU.write("␣␣␣␣type␣␣␣processor;\n")
        fU.write("␣␣␣␣value␣␣uniform␣"+Vinitstr+";\n")
        fk.write("␣␣␣␣type␣␣␣processor;\n")
        fk.write("␣␣␣␣value␣␣uniform␣"+tkestr+";\n")
        fo.write("␣␣␣␣type␣␣␣processor;\n")
        fo.write("␣␣␣␣value␣␣uniform␣"+omegastr+";\n")
        fn.write("␣␣␣␣type␣␣␣processor;\n")
        fn.write("␣␣␣␣value␣␣uniform␣0;\n")
    elif patches[patch] == "cyclicAMI;":
        # processor boundary (for parallel)
        fp.write("␣␣␣␣type␣␣␣cyclicAMI;\n")
        fp.write("␣␣␣␣value␣␣uniform␣0;\n")
        fU.write("␣␣␣␣type␣␣␣cyclicAMI;\n")
        fU.write("␣␣␣␣value␣␣uniform␣"+Vinitstr+";\n")
        fk.write("␣␣␣␣type␣␣␣cyclicAMI;\n")
        fk.write("␣␣␣␣value␣␣uniform␣"+tkestr+";\n")
        fo.write("␣␣␣␣type␣␣␣cyclicAMI;\n")
        fo.write("␣␣␣␣value␣␣uniform␣"+omegastr+";\n")
        fn.write("␣␣␣␣type␣␣␣cyclicAMI;\n")
        fn.write("␣␣␣␣value␣␣uniform␣0;\n")
    elif patches[patch] == "cyclic;":
```

```
            # processor boundary (for parallel)
            fp.write("␣␣␣␣type␣␣␣cyclic;\n")
            fp.write("␣␣␣␣value␣␣uniform␣0;\n")
            fU.write("␣␣␣␣type␣␣␣cyclic;\n")
            fU.write("␣␣␣␣value␣␣uniform␣"+Vinitstr+";\n")
            fk.write("␣␣␣␣type␣␣␣cyclic;\n")
            fk.write("␣␣␣␣value␣␣uniform␣"+tkestr+";\n")
            fo.write("␣␣␣␣type␣␣␣cyclic;\n")
            fo.write("␣␣␣␣value␣␣uniform␣"+omegastr+";\n")
            fn.write("␣␣␣␣type␣␣␣cyclic;\n")
            fn.write("␣␣␣␣value␣␣uniform␣0;\n")
        elif patches[patch] == "empty;":
            # empty patch
            fp.write("␣␣␣␣type␣␣␣empty;\n")
            fU.write("␣␣␣␣type␣␣␣empty;\n")
            fk.write("␣␣␣␣type␣␣␣empty;\n")
            fo.write("␣␣␣␣type␣␣␣empty;\n")
            fn.write("␣␣␣␣type␣␣␣empty;\n")
        else:
            # generic boundary type
            fp.write("␣␣␣␣type␣␣␣zeroGradient;\n")
            fp.write("␣␣␣␣value␣␣uniform␣0;\n")
            fU.write("␣␣␣␣type␣␣␣fixedValue;\n")
            fU.write("␣␣␣␣value␣␣uniform␣"+Vinitstr+";\n")
            fk.write("␣␣␣␣type␣␣␣fixedValue;\n")
            fk.write("␣␣␣␣value␣␣uniform␣"+tkestr+";\n")
            fo.write("␣␣␣␣type␣␣␣fixedValue;\n")
            fo.write("␣␣␣␣value␣␣uniform␣"+omegastr+";\n")
            fn.write("␣␣␣␣type␣␣␣calculated;\n")
            fn.write("␣␣␣␣value␣␣uniform␣0;\n")
        # finish patch
        fp.write("␣␣}\n")
        fU.write("␣␣}\n")
        fk.write("␣␣}\n")
        fo.write("␣␣}\n")
        fn.write("␣␣}\n")
    # end writing boundaryFields
    fp.write("}\n")
    fU.write("}\n")
    fk.write("}\n")
    fo.write("}\n")
    fn.write("}\n")

    # close output files
    fp.flush()
    os.fsync(fp.fileno())
    fp.close()
    fU.flush()
    os.fsync(fU.fileno())
    fU.close()
    fk.flush()
    os.fsync(fk.fileno())
    fk.close()
    fo.flush()
    os.fsync(fo.fileno())
    fo.close()
    fn.flush()
    os.fsync(fn.fileno())
    fn.close()

    return None


def renamePatch(name, nname):
    """
␣␣␣␣Renames␣a␣patch␣from␣name␣to␣nname␣in␣the␣constant/polyMesh/boundary␣file.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣name:␣␣␣string
␣␣␣␣␣␣␣␣Name␣of␣the␣patch␣to␣be␣renamed.
␣␣␣␣nname:␣␣string
␣␣␣␣␣␣␣␣New␣name␣for␣patch.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣None
␣␣␣␣␣␣␣␣If␣successful.
␣␣␣␣"""
    import re

    # read the file into a string
    fin = file("constant/polyMesh/boundary", "r")
    filestr = fin.read()
    fin.close()

    # make regular expression
    namepattern = re.compile(r"\s*"+name+r"\s*\{\s*type\s*\w+;")
    reobj = namepattern.search(filestr)
    if reobj:
        namepattern2 = re.compile(r"\w+\s*\{")
        newstr = namepattern2.sub(nname+"\n{", reobj.group())
    else:
        return "Patch not found"

    # create output string
    outstr = filestr[:reobj.start()]+newstr+filestr[reobj.end():]

    # write output string
    fout = file("constant/polyMesh/boundary", "w")
    fout.write(outstr)
    fout.flush()
    os.fsync(fout.fileno())
    fout.close()

    return None


def deletePatch(name):
    """
    Deletes the patch called name from constant/polyMesh/boundary file.

    Parameters
    ----------
    name:   string
        Name of the patch to be deleted.

    Returns
    -------
    None
        If successful.
    """
    import re

    # read the file into a string
    fin = file("constant/polyMesh/boundary", "r")
    filestr = fin.read()
    fin.close()

    # make patch regular expression
    namepattern = re.compile(r"\s*"+name+r"\s*\{\s*type\s*\w+;[\s\w]*?\}")
    namereobj = namepattern.search(filestr)
    if namereobj:
        numpattern = re.compile(r"\n\d+\n{")
        numreobj = numpattern.search(filestr)
        if numreobj:
            newnum = "\n"+str(int(numreobj.group())-1)+"\n{"
        else:
            return "Number of patches not found"
    else:
        return "Patch not found"

    # create output string
    outstr = filestr[:numreobj.start()] + newnum +\
        filestr[numreobj.end():namereobj.start()] +\
        filestr[namereobj.end():]

    # write output string
    fout = file("constant/polyMesh/boundary", "w")
    fout.write(outstr)
    fout.flush()
    os.fsync(fout.fileno())
    fout.close()

    return None


def changePatchType(name, ptype):
    """
    Changes the type of patch name to type.

    Parameters
    ----------
    name:   string
        Name of the patch to be changed.
    ptype:  string
        Type to change the patch to.

    Returns
    -------
    None
        If successful.
    """
    import re

    # check for ; on ptype and add if not present
    if ptype[-1] != ";":
        ptype = ptype + ";"

    # read the file into a string
    fin = file("constant/polyMesh/boundary", "r")
    filestr = fin.read()
    fin.close()
    # make regular expression for patch and type
    patchpattern = re.compile(r"\s*"+name+r"\s*\{\s*type\s*\w+;")
    reobj = patchpattern.search(filestr)
    if reobj:
        typepattern = re.compile(r"\w+;")
        newstr = typepattern.sub(ptype, reobj.group())
    else:
        return "Patch␣not␣found"

    # create output string
    outstr = filestr[:reobj.start()]+newstr+filestr[reobj.end():]

    # write output string
    fout = file("constant/polyMesh/boundary", "w")
    fout.write(outstr)
    fout.flush()
    os.fsync(fout.fileno())
    fout.close()

    return None


def forces(name, time="0"):
    """
␣␣␣␣Extract␣a␣list␣of␣the␣forces␣under␣name␣and␣returns␣them.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣name:␣␣␣string
␣␣␣␣␣␣␣␣Name␣of␣force␣set␣to␣extract␣from.
␣␣␣␣time:␣␣␣string
␣␣␣␣␣␣␣␣Beginning␣time␣of␣extraction.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣T:␣␣list␣of␣numbers
␣␣␣␣␣␣␣␣Time␣values.
␣␣␣␣Fxp:␣␣␣␣list␣of␣numbers
␣␣␣␣␣␣␣␣Pressure␣force␣in␣x␣direction.
␣␣␣␣Fyp:␣␣␣␣list␣of␣numbers
␣␣␣␣␣␣␣␣Pressure␣force␣in␣y␣direction.
␣␣␣␣Fzp:␣␣␣␣list␣of␣numbers
␣␣␣␣␣␣␣␣Pressure␣force␣in␣z␣direction.
␣␣␣␣Fxv:␣␣␣␣list␣of␣numbers
␣␣␣␣␣␣␣␣Viscous␣force␣in␣x␣direction.
␣␣␣␣Fyv:␣␣␣␣list␣of␣numbers
␣␣␣␣␣␣␣␣Viscous␣force␣in␣y␣direction.
␣␣␣␣Fzv:␣␣␣␣list␣of␣numbers
␣␣␣␣␣␣␣␣Viscous␣force␣in␣z␣direction.
␣␣␣␣Mxp:␣␣␣␣list␣of␣numbers
␣␣␣␣␣␣␣␣Pressure␣moment␣about␣x␣axis.
␣␣␣␣Myp:␣␣␣␣list␣of␣numbers
␣␣␣␣␣␣␣␣Pressure␣moment␣about␣y␣axis.
␣␣␣␣Mzp:␣␣␣␣list␣of␣numbers
␣␣␣␣␣␣␣␣Pressure␣moment␣about␣z␣axis.
␣␣␣␣Mxv:␣␣␣␣list␣of␣numbers
␣␣␣␣␣␣␣␣Viscous␣moment␣about␣x␣axis.
␣␣␣␣Myv:␣␣␣␣list␣of␣numbers
␣␣␣␣␣␣␣␣Viscous␣moment␣about␣y␣axis.
␣␣␣␣Mzv:␣␣␣␣list␣of␣numbers
␣␣␣␣␣␣␣␣Viscous␣moment␣about␣z␣axis.
␣␣␣␣"""

    # make filename
    filename = name + "/" + time + "/forces.dat"

    # read in file
    fin = file(filename, "r")
    flines = fin.readlines()
    fin.close()

    # create empty lists
    T = []
    Fxp = []
    Fyp = []
    Fzp = []
    Fxv = []
    Fyv = []
    Fzv = []
    Mxp = []
    Myp = []
    Mzp = []
    Mxv = []
    Myv = []
    Mzv = []

    # populate lists
    for fline in flines:
        flist = fline.split()
        if flist[0] != "#":
            T.append(float(flist[0]))
            Fxp.append(float(flist[1][3:]))
            Fyp.append(float(flist[2]))
            Fzp.append(float(flist[3][:-1]))
            Fxv.append(float(flist[4][1:]))
            Fyv.append(float(flist[5]))
            Fzv.append(float(flist[6][:-2]))
            Mxp.append(float(flist[7][2:]))
            Myp.append(float(flist[8]))
            Mzp.append(float(flist[9][:-1]))
            Mxv.append(float(flist[10][1:]))
            Myv.append(float(flist[11]))
            Mzv.append(float(flist[12][:-3]))

    # convert to array
    #T = array(T)
    #Fxp = array(Fxp)
    #Fyp = array(Fyp)
    #Fzp = array(Fzp)
    #Fxv = array(Fxv)
    #Fyv = array(Fyv)
    #Fzv = array(Fzv)
    #Mxp = array(Mxp)
    #Myp = array(Myp)
    #Mzp = array(Mzp)
    #Mxv = array(Mxv)
    #Myv = array(Myv)
    #Mzv = array(Mzv)

    # return arrays
    return T, Fxp, Fyp, Fzp, Fxv, Fyv, Fzv, Mxp, Myp, Mzp, Mxv, Myv, Mzv


def residuals(name):
    """
    Extracts the residuals from a log file of name.
    Returns the residuals as a dictionary where d[key] = list
    and the keys are the solution variables.

    Parameters
    ----------
    name:   string
        Name of the log file to extract residuals from.

    Returns
    -------
    d:  dictionary
        Dictionary of residuals, where the keys are solution variables
        and the values are chronological lists of residuals.
    """
    # create dictionary
    d = {'Time': [], 'local_cont': [], 'global_cont': [], 'cum_cont': []}

    # read in file
    fin = open(name, 'r')
    flines = fin.readlines()
    fin.close()

    # read each line and deal with each case
    for line in flines:
        words = line.split()
        if len(words) < 3:
            pass
        elif words[0] == 'Time' and words[1] == '=':
            d['Time'].append(float(words[2]))
        elif words[0] == 'time' and words[1] == 'step'\
            and words[2] == 'continuity' and words[3] == 'errors':
                d['local_cont'].append(float(words[8][:-1]))
                d['global_cont'].append(float(words[11][:-1]))
                d['cum_cont'].append(float(words[14][:-1]))
        elif words[1] == 'Solving' and words[2] == 'for':
            # if d.has_key(words[3][:-1]+'_init'):
            if words[3][:-1]+'_init' in d:
                d[words[3][:-1]+'_init'].append(float(words[7][:-1]))
                d[words[3][:-1]+'_final'].append(float(words[11][:-1]))
            else:
                d[words[3][:-1]+'_init'] = []
                d[words[3][:-1]+'_final'] = []
                d[words[3][:-1]+'_init'].append(float(words[7][:-1]))
                d[words[3][:-1]+'_final'].append(float(words[11][:-1]))
    return d


def addAMI():
    """
    Adds an arbitrary mesh interface to a mesh.

    Parameters
    ----------

    Returns
    -------
    None
        If successful.
    """

    return None


def snapSTL(stl, patch=None):
    """
    Snaps an stl to the corresponding patch generated by snappyHexMesh,
    or to 'patch' if supplied.  snapEdge must be installed and compiled.

    Parameters
    ----------
    stl:    string
        Name of stl file to snap.
    patch:  string
        Optional patch to snap to this stl.

    Returns
    -------
    None
        If successful.
    """

    # open file to write snapEdgeDict
    fout = open("constant/snapEdgeDict", 'w')

    # write file header
    fout.write("FoamFile\n{\n")
    fout.write("  version   2.0;\n")
    fout.write("  format    ascii;\n")
    fout.write("  class     dictionary;\n")
    fout.write("  object    snapEdgeDict;\n")
    fout.write("  location  \"constant\";\n}\n\n")

    # write dictionary contents
    fout.write("snapEdgeDict\n{\n\n")
    # snapPatches
    fout.write("  snapPatches\n  (\n    ")
    if patch:
        fout.write(patch+"\n")
    else:
        fout.write(stl+" "+stl[:-4]+"\n")
    fout.write("  );\n\n")
    # snapZones
    fout.write("  snapZones\n  (\n  );\n\n")
    # stlFileNames
    fout.write("  stlFileNames\n  (\n    ")
    fout.write(stl+"\n")
    fout.write("  );\n\n")
    # fitting parameters
    fout.write("  tolerance        1.9;\n")
    fout.write("  relaxation       0.1;\n")
    fout.write("  nIterations      15;\n")
    fout.write("  includeInterior  yes;\n")
    fout.write("  featureAngle     30;\n")
    fout.write("  excludeEdgeAngle 60;\n")
    fout.write("  parallelAngle    50;\n")
    fout.write("  fitFactor        1.0e-5;\n")
    fout.write("\n}\n")

    # close file
    fout.flush()
    os.fsync(fout.fileno())
    fout.close()

    # run snapEdge
    run("snapEdge -overwrite", True)

    # option to remove
    # os.system("rm 0/ccx 0/ccy 0/ccz 0/cellLevel 0/pointLevel 0/meshPhi")

    return None


def run(command, silent=False):
    """
    Executes a command in a shell, automatically writing to a log.

    Parameters
    ----------
    command:    str
        Line to execute.
    silent: bool (default False)
        Whether to suppress output to stdout,
        will still write to a log even when True.

    Returns
    -------
    None
        If successful.
    """
```

```
if silent:
    os.system(command+" >> log."+command.split()[0])
else:
    os.system(command+" | tee -a log."+command.split()[0])

return None
```

Appendix D

Optimisation Functions

This is a collection of Python functions written for design search and optimisation applications, including surrogate modelling methods and a number of genetic algorithms, that are not provided in the `scipy.optimize` 'standard' optimisation library for Python.

```python
"""
ajopt.py

A collection of search and optimisation functions written in Python.

(c) copyright Aleksander Dubas 2011-2013
"""
import numpy as np
import numpy.linalg as npla

# storage for variables for wrapper functions.
_optdic = {}


# modelling classes
class Surrogate():
    """
    Base class for a surrogate model.
    """
    def __init__(self):
        self.xs = []
        self.ys = []

    def loaddata(self, filename):
        """
        Loads data into the surrogate model,
        uses the same data structure as the savedata function.
        Use of an absolute path is recommended.
        """
        fin = open(filename, 'r')
        lines = fin.readlines()
        fin.close()
        for line in lines:
            parts = line.split()
            self.xs.append([])
            for i in range(len(parts)):
                if i < len(parts)-1:
                    self.xs[-1].append(float(parts[i]))
                else:
                    self.ys.append(float(parts[i]))

        # convert xs to array
        self.xs = np.array(self.xs)
        return None

    def savedata(self, filename):
        """
        Saves data from a surrogate model,
        in the structure:
        xs[0][0] xs[0][1] xs[0][2] ... xs[0][-2] xs[0][-1] ys[0]\n
        xs[1][0] xs[1][1] xs[1][2] ... xs[1][-2] xs[1][-1] ys[1]\n
        Use of an absolute path is recommended.
        """
        fout = open(filename, 'w')
        for i in range(len(self.xs)):
            for x in self.xs[i]:
                fout.write(str(x)+" ")
            try:
                fout.write(str(self.ys[i])+"\n")
            except IndexError:
                fout.write("\n")
        fout.close()

        return None

    def infill(self, f):
        """
________Evaluates_any_unevaluated_points_in_self.xs_array_using_f.
________"""
        xlen = len(self.xs)
        ylen = len(self.ys)
        if xlen == ylen:
            return None
        self.ys = np.hstack((self.ys, np.zeros(xlen-ylen)))
        for i in range(ylen, xlen):
            self.ys[i] = f(self.xs[i])
        return None


class GaussianRBF(Surrogate):
    """
____Constructs_a_surrogate_model_using_the_Radial_Basis_Function_in_SciPy.
____Uses_a_Gaussian_distribution.
____"""
    def build(self):
        self.name = "GaussianRBF"
        import scipy.interpolate.rbf as RBFmodule

        # creating packing list for passing to RBF
        packinglist = []
        for i in range(len(self.xs[0])):
            packinglist.append(self.xs[:, i])
        packinglist.append(self.ys)
        self.rbf = RBFmodule.Rbf(*tuple(packinglist))
        self.rbf.function = "gaussian"

    def f(self, xs):
        return self.rbf(*tuple(xs))


class MultiQuadricRBF(Surrogate):
    """
____Constructs_a_surrogate_model_using_the_Radial_Basis_Function_in_SciPy.
____Uses_a_MultiQuadric_distribution.
____"""
    def build(self):
        self.name = "MultiQuadricRBF"
        import scipy.interpolate.rbf as RBFmodule

        # creating packing list for passing to RBF
        packinglist = []
        for i in range(len(self.xs[0])):
            packinglist.append(self.xs[:, i])
        packinglist.append(self.ys)
        self.rbf = RBFmodule.Rbf(*tuple(packinglist))
        self.rbf.function = "multiquadric"

    def f(self, xs):
        return self.rbf(*tuple(xs))


class InverseRBF(Surrogate):
    """
____Constructs_a_surrogate_model_using_the_Radial_Basis_Function_in_SciPy.
____Uses_an_Inverse_distribution.
____"""
    def build(self):
        self.name = "InverseRBF"
        import scipy.interpolate.rbf as RBFmodule

        # creating packing list for passing to RBF
        packinglist = []
        for i in range(len(self.xs[0])):
            packinglist.append(self.xs[:, i])
        packinglist.append(self.ys)
        self.rbf = RBFmodule.Rbf(*tuple(packinglist))
        self.rbf.function = "inverse"

    def f(self, xs):
        return self.rbf(*tuple(xs))


class LinearRBF(Surrogate):
    """
____Constructs_a_surrogate_model_using_the_Radial_Basis_Function_in_SciPy.
____Uses_a_Linear_distribution.
____"""
    def build(self):
        self.name = "LinearRBF"
        import scipy.interpolate.rbf as RBFmodule
        # creating packing list for passing to RBF
        packinglist = []
        for i in range(len(self.xs[0])):
            packinglist.append(self.xs[:, i])
        packinglist.append(self.ys)
        self.rbf = RBFmodule.Rbf(*tuple(packinglist))
        self.rbf.function = "linear"

    def f(self, xs):
        return self.rbf(*tuple(xs))


class CubicRBF(Surrogate):
    """
␣␣␣␣Constructs␣a␣surrogate␣model␣using␣the␣Radial␣Basis␣Function␣in␣SciPy.
␣␣␣␣Uses␣a␣Cubic␣distribution.
␣␣␣␣"""
    def build(self):
        self.name = "CubicRBF"
        import scipy.interpolate.rbf as RBFmodule

        # creating packing list for passing to RBF
        packinglist = []
        for i in range(len(self.xs[0])):
            packinglist.append(self.xs[:, i])
        packinglist.append(self.ys)
        self.rbf = RBFmodule.Rbf(*tuple(packinglist))
        self.rbf.function = "cubic"

    def f(self, xs):
        return self.rbf(*tuple(xs))


class QuinticRBF(Surrogate):
    """
␣␣␣␣Constructs␣a␣surrogate␣model␣using␣the␣Radial␣Basis␣Function␣in␣SciPy.
␣␣␣␣Uses␣a␣Quintic␣distribution.
␣␣␣␣"""
    def build(self):
        self.name = "QuinticRBF"
        import scipy.interpolate.rbf as RBFmodule

        # creating packing list for passing to RBF
        packinglist = []
        for i in range(len(self.xs[0])):
            packinglist.append(self.xs[:, i])
        packinglist.append(self.ys)
        self.rbf = RBFmodule.Rbf(*tuple(packinglist))
        self.rbf.function = "quintic"

    def f(self, xs):
        return self.rbf(*tuple(xs))


class ThinPlateRBF(Surrogate):
    """
␣␣␣␣Constructs␣a␣surrogate␣model␣using␣the␣Radial␣Basis␣Function␣in␣SciPy.
␣␣␣␣Uses␣a␣Thin␣Plate␣distribution.
␣␣␣␣"""
    def build(self):
        self.name = "ThinPlateRBF"
        import scipy.interpolate.rbf as RBFmodule

        # creating packing list for passing to RBF
        packinglist = []
        for i in range(len(self.xs[0])):
            packinglist.append(self.xs[:, i])
        packinglist.append(self.ys)
        self.rbf = RBFmodule.Rbf(*tuple(packinglist))
        self.rbf.function = "thin_plate"

    def f(self, xs):
        return self.rbf(*tuple(xs))


class MPSM(Surrogate):
    """
␣␣␣␣MPS␣Method␣derived␣surrogate␣model␣class,
␣␣␣␣known␣as␣MultiPoint␣Surrogate␣Model,
␣␣␣␣using␣2nd␣order␣Runge−Kutta␣integration␣to␣derive␣the␣surface.
␣␣␣␣"""
    def __init__(self):
        self.xs = []
        self.ys = []
        self.r_e = 0.25

    def setre(self, bounds=False):
        """
␣␣␣␣␣␣␣␣Sets␣the␣effective␣radius␣to␣half␣the␣maximum␣distance,
␣␣␣␣␣␣␣␣with␣optional␣specification␣of␣the␣bounds.
␣␣␣␣␣␣␣␣"""
        total = 0
        k = len(self.xs[0])
        if bounds:
            for bound in bounds:
                total += (bound[1] - bound[0])**2
            self.r_e = 0.5*total**0.5
        else:
            for i in range(k):
                ith_slice = self.xs[:, i]
                total += (max(ith_slice)-min(ith_slice))**2
            self.r_e = 0.5*total**0.5
        return

    def build(self):
        """
␣␣␣␣␣␣␣␣Finds␣the␣relevant␣parameters␣for␣the␣surrogate␣model.
␣␣␣␣␣␣␣␣"""
        self.name = "MPSM"
        n = len(self.ys)
        k = len(self.xs[0])
        self.grads = np.zeros([n, k])
        for i in range(n):
            ni = 0
            total = np.zeros(k)
            for j in range(n):
                if not i == j:
                    r = npla.norm(self.xs[j]-self.xs[i])
                    if r > 0:
                        w = self.weight(r)
                        ni += w
                        scal = w*(self.ys[j]-self.ys[i])/(r**2)
                        total += scal*(self.xs[j]-self.xs[i])
                    else:
                        w = 0
                        ni += 0
                        scal = 0
                        total += 0
            self.grads[i, :] = (k/float(ni))*total
        return None

    def weight(self, r):
        """
␣␣␣␣␣␣␣␣Weight␣function.
␣␣␣␣␣␣␣␣"""
        return max(self.r_e/r - 1, 0)

    def f(self, xs):
        """
␣␣␣␣␣␣␣␣Evaluates␣the␣surrogate␣model␣at␣xs.
␣␣␣␣␣␣␣␣"""
        n = len(self.ys)
        k = len(self.xs[0])
        # find closest x
        curi = 0
        curlen = npla.norm(xs-self.xs[0])
        for i in range(1, n):
            nextlen = npla.norm(xs-self.xs[i])
            if nextlen < curlen:
                curi = i
                curlen = nextlen
        # extrapolate based on gradients
        dx = xs - self.xs[curi]
        newy = self.ys[curi]+np.dot(self.grads[curi], dx)

        # find gradient at new point
        ni = 0
        newgrad = np.zeros(k)
        for j in range(n):
            if npla.norm(self.xs[j]-xs) != 0:
                w = self.weight(npla.norm(self.xs[j]-xs))
                ni += w
                scal = w*(self.ys[j]-newy)/(npla.norm(self.xs[j]-xs)**2)
                newgrad += scal*(self.xs[j]-xs)
        newgrad *= (k/float(ni))

        # return the final point based on average gradient
        finalgrad = 0.5*(newgrad+self.grads[curi])
        return self.ys[curi]+np.dot(finalgrad, dx)


class MPSM_2PE(MPSM):
    """
␣␣␣␣Variation␣of␣the␣MultiPoint␣Surrogate␣Model␣to␣use␣an␣average␣evaluation
␣␣␣␣of␣2␣points␣using␣Eulerian␣integration.
␣␣␣␣"""
    def f(self, xs):
        """
␣␣␣␣␣␣␣␣Evaluates␣the␣surrogate␣model␣at␣xs.
␣␣␣␣␣␣␣␣"""
        n = len(self.ys)
        # find closest xs
        curi = 0
        cur2i = 0
        curlen = npla.norm(xs-self.xs[0])
        for i in range(1, n):
            nextlen = npla.norm(xs-self.xs[i])
            if nextlen < curlen:
                cur2i = curi
                curi = i
                curlen = nextlen
        # extrapolate based on gradients
        dx = xs - self.xs[curi]
        dx2 = xs - self.xs[cur2i]
        y1 = self.ys[curi] + np.dot(self.grads[curi], dx)
        y2 = self.ys[cur2i] + np.dot(self.grads[cur2i], dx2)
        return 0.5*(y1+y2)


class MPSM_2PEB(MPSM):
    """
␣␣␣␣Variation␣of␣the␣MultiPoint␣Surrogate␣Model␣to␣use␣a␣blended␣evaluation
␣␣␣␣of␣2␣points␣using␣Eulerian␣integration.
␣␣␣␣"""
    def f(self, xs):
        """
␣␣␣␣␣␣␣␣Evaluates␣the␣surrogate␣model␣at␣xs.
␣␣␣␣␣␣␣␣"""
        n = len(self.ys)

        # find closest xs
        curi = 0
        cur2i = 0
        curlen = npla.norm(xs-self.xs[0])
        for i in range(1, n):
            nextlen = npla.norm(xs-self.xs[i])
            if nextlen < curlen:
                cur2i = curi
                curi = i
                curlen = nextlen
        # extrapolate based on gradients
        dx = xs - self.xs[curi]
        dx2 = xs - self.xs[cur2i]
        r1 = npla.norm(dx)
        r2 = npla.norm(dx)
        y1 = self.ys[curi] + np.dot(self.grads[curi], dx)
        y2 = self.ys[cur2i] + np.dot(self.grads[cur2i], dx2)
        if r1 < 1e-8:
            return y1
        if r2 < 1e-8:
            return y2
        return ((r2*y1)/(r1+r2)) + ((r1*y2)/(r1+r2))


class MPSM_APB(MPSM):
    """
␣␣␣␣Variation␣of␣the␣MultiPoint␣Surrogate␣Model␣to␣use␣a␣weighted␣evaluation
␣␣␣␣of␣all␣points␣using␣Eulerian␣integration.
␣␣␣␣"""
    def __init__(self):
        self.xs = []
        self.ys = []
        self.r_e = 0.25
        self.laf = 5

    def f(self, xs):
        """
␣␣␣␣␣␣␣␣Evaluates␣the␣surrogate␣model␣at␣xs.
␣␣␣␣␣␣␣␣"""
        n = len(self.ys)
        k = len(self.xs[0])

        y = 0

        # initialise arrays
        dxs = np.zeros([n, k])
        rs = np.zeros([n])
        for i in range(n):
            dxs[i] = xs - self.xs[i]
            rs[i] = npla.norm(dxs[i])
        # renormalise rs so that closer points have higher weighting
        # use self.laf to amplify local effect
        rmax = max(rs)
        rs = (rmax - rs)**self.laf
        # optimisation to do sum once and use faster multiply instead of divide
        inversersum = 1.0/sum(rs)
        for i in range(n):
            y += rs[i]*inversersum*(self.ys[i] + np.dot(self.grads[i], dxs[i]))
        return y


class MPSM_LPB(MPSM):
    """
␣␣␣␣Variation␣of␣the␣MultiPoint␣Surrogate␣Model␣to␣use␣a␣weighted␣evaluation
␣␣␣␣of␣local␣points␣using␣Eulerian␣integration.
␣␣␣␣"""
```

```
    def __init__(self):
        self.xs = []
        self.ys = []
        self.r_e = 0.25
        self.laf = 5

    def f(self, xs):
        """
        Evaluates the surrogate model at xs.
        """
        n = len(self.ys)

        ys = []
        rs = []

        # create y array
        for i in range(n):
            dx = xs - self.xs[i]
            r = npla.norm(dx)
            if r < self.r_e:
                ys.append(self.ys[i] + np.dot(self.grads[i], dx))
                rs.append(r)
        # invert rs and convert ys
        rs = (self.r_e-np.array(rs))**self.laf
        ys = np.array(ys)
        return np.dot(rs, ys)/sum(rs)


class Kriging(Surrogate):
    """
    A Kriging surrogate model, based entirely on:
    Forrester et al - Engineering Design via Surrogate Modelling (Wiley, 2008).
    """
    def __init__(self):
        self.xs = []
        self.ys = []
        self.gapops = 20
        self.gagens = 100
        self.lntlb = -7  # e^-7 ~= 10^-3
        self.lntub = 5   # e^5 ~= 10^2 as per book

    def build(self):
        self.name = "Kriging"
        # extract size parameters
        k = len(self.xs[0])
        # define genetic algorithm bounds for Theta to be used in building
        Kbounds = []
        for i in range(k):
            Kbounds.append((self.lntlb, self.lntub))
            # made bounds a variable so they can be changed
        # run ga search of likelihood
        self.Theta, self.MinNegLnLikelihood, _ = \
            ga3(lambda x: self.likelihood(x)[0], Kbounds,
                self.gapops, self.gagens)
        # put Cholesky factorisation of Psi into namespace
        self.NegLnLike, self.Psi, self.U = self.likelihood(self.Theta)
        return None

    def likelihood(self, thetas):
        # initialise theta, n, one, eps
        theta = np.e**np.array(thetas)
        n = len(self.ys)
        one = np.ones([n])
        eps = 1000*np.spacing(1)
        # pre-allocate memory
        Psi = np.zeros([n, n])
        # build upper half of the correlation matrix
        for i in range(n):
            for j in range(i+1, n):
                Psi[i, j] = np.exp(-sum(theta*(self.xs[i]-self.xs[j])**2))
        # add upper and lower halves and diagonal of ones
        # plus a small number to reduce ill conditioning
        Psi = Psi + Psi.T + np.eye(n) + np.eye(n)*eps

        # cholesky factorisation
        # added try/except block to capture error and implement penalty
        try:
            U = npla.cholesky(Psi).T
        except npla.LinAlgError:
            return 1000, Psi, np.zeros([n, n])
        # Forrester et al. have a penalty here if ill-conditioned
        # but this is not implemented in numpy.linalg.cholesky

        # Sum lns of diagonal to find ln(det(Psi))
        LnDetPsi = 2*sum(np.log(np.abs(np.diag(U))))

        # use back-substitution of Cholesky instead of inverse
        mu = np.dot(one, npla.solve(U, npla.solve(U.conj().T, self.ys))) /\
            np.dot(one, npla.solve(U, npla.solve(U.conj().T, one)))
        SigmaSqr = (np.dot(self.ys-mu,
                           npla.solve(U, npla.solve(U.conj().T, self.ys-mu))) /
                    float(n))
        NegLnLike = -1*(-(0.5*n)*np.log(SigmaSqr)-0.5*LnDetPsi)
        return NegLnLike, Psi, U

    def f(self, xs):
        # initialise theta
        theta = np.e**np.array(self.Theta)
        # calculate number of sample points
        n = len(self.ys)
        # create vector of ones
        one = np.ones([n])
        # calculate mu
        mu = np.dot(one, npla.solve(self.U,
                                    npla.solve(self.U.conj().T, self.ys))) /\
            np.dot(one, npla.solve(self.U, npla.solve(self.U.conj().T, one)))
        # initialise psi
        psi = np.ones([n])
        # fill psi vector
        for i in range(n):
            psi[i] = np.exp(-sum(theta*np.abs(self.xs[i]-xs)**2))
        return mu+np.dot(psi.conj().T,
                         npla.solve(self.U,
                                    npla.solve(self.U.conj().T, self.ys-mu)))

    def lb(self, xs):
        # initialise theta
        theta = np.e**np.array(self.Theta)
        # intialise A
        if not hasattr(self, "A"):
            self.A = 2
        # calculate number of sample points
        n = len(self.ys)
        # create vector of ones
        one = np.ones([n])
        # calculate mu
        mu = np.dot(one, npla.solve(self.U,
                                    npla.solve(self.U.conj().T, self.ys))) /\
            np.dot(one, npla.solve(self.U, npla.solve(self.U.conj().T, one)))
        # calculate sigma^2
        SigmaSqr = np.dot((self.ys-mu),
                          npla.solve(self.U, npla.solve(self.U.conj().T,
                                                        self.ys-mu)))/float(n)
        # initialise psi
        psi = np.ones([n])
        # fill psi vector
        for i in range(n):
            psi[i] = np.exp(-sum(theta*np.abs(self.xs[i]-xs)**2))
        # calculate prediction
        f = mu + np.dot(psi.conj().T,
                        npla.solve(self.U,
                                   npla.solve(self.U.conj().T, self.ys-mu)))
        # error
        SSqr = SigmaSqr*(1-np.dot(psi,
                                  npla.solve(self.U,
                                             npla.solve(self.U.conj().T,
                                                        psi))))
        # lower bound
        return f - self.A * np.sqrt(SSqr)

    def ei(self, xs):
        # define the error function as it's missing from python
        def erf(x):
            # save the sign of x
            sign = 1 if x >= 0 else -1
            x = np.abs(x)

            # constants
            a1 = 0.254829592
            a2 = -0.284496736
            a3 = 1.421413741
            a4 = -1.453152027
            a5 = 1.061405429
            p = 0.3275911

            # A&S formula 7.1.26
            t = 1.0/(1.0 + p*x)
            y = 1.0 - (((((a5*t + a4)*t) + a3)*t + a2)*t + a1)*t*np.exp(-x*x)
            return sign*y  # erf(-x) = -erf(x)
        # initialise theta
        theta = np.e**np.array(self.Theta)
        # intialise A
        if not hasattr(self, "A"):
            self.A = 2
        # calculate number of sample points
        n = len(self.ys)
        # create vector of ones
        one = np.ones([n])
        # calculate mu
        mu = np.dot(one, npla.solve(self.U,
                                    npla.solve(self.U.conj().T, self.ys))) /\
            np.dot(one, npla.solve(self.U,
                                   npla.solve(self.U.conj().T, one)))
        # calculate sigma^2
        SigmaSqr = np.dot((self.ys-mu),
                          npla.solve(self.U,
                                     npla.solve(self.U.conj().T,
                                                self.ys-mu)))/float(n)
        # initialise psi
        psi = np.ones([n])
        # fill psi vector
        for i in range(n):
            psi[i] = np.exp(-sum(theta*np.abs(self.xs[i]-xs)**2))
        # calculate prediction
        f = mu + np.dot(psi.conj().T,
                        npla.solve(self.U,
                                   npla.solve(self.U.conj().T, self.ys-mu)))
        y_hat = f
        # error
        SSqr = SigmaSqr*(1-np.dot(psi,
                                  npla.solve(self.U,
                                             npla.solve(self.U.conj().T,
                                                        psi))))
        # find best so far:
        y_min = min(self.ys)
        # expected improvement
        if SSqr == 0:
            return 0
        else:
            ei_term1 = (y_min-y_hat) *\
                (0.5+0.5*erf((1/np.sqrt(2)) *
                 ((y_min-y_hat)/np.sqrt(np.abs(SSqr)))))
            ei_term2 = np.sqrt(np.abs(SSqr)) *\
                (1/np.sqrt(2*np.pi))*np.exp(-0.5*((y_min-y_hat)**2/SSqr))
            return ei_term1 + ei_term2


class fastKriging(Surrogate):
    """
    A Kriging surrogate model, based entirely on:
    Forrester et al - Engineering Design via Surrogate Modelling (Wiley, 2008).
    including a few optimisation changes to reduce build and evaluation time.
    """
    def __init__(self):
        self.xs = []
        self.ys = []
        self.gapops = 20
        self.gagens = 100
        self.lntlb = -7  # e^-7 ~= 10^-3
        self.lntub = 5   # e^5 ~= 10^2 as per book

    def build(self):
        self.name = "Kriging"
        # extract size parameters
        k = len(self.xs[0])
        n = len(self.ys)
        self.overn = 1/float(n)
        # define genetic algorithm bounds for Theta to be used in building
        Kbounds = []
        for i in range(k):
            Kbounds.append((self.lntlb, self.lntub))
            # made bounds a variable so they can be changed
        # run ga search of likelihood
        self.Theta, self.MinNegLnLikelihood, _ = \
            ga3cc(lambda x: self.likelihood(x)[0], Kbounds,
                  self.gapops, self.gagens/20)
        # put Cholesky factorisation of Psi into namespace
        self.NegLnLike, self.Psi, self.U = self.likelihood(self.Theta)
        return None

    def likelihood(self, thetas):
        # initialise theta, n, one, eps
        theta = np.e**np.array(thetas)
        n = len(self.ys)
        one = np.ones([n])
        eps = 1000*np.spacing(1)
        # pre-allocate memory
        Psi = np.zeros([n, n])
        # build upper half of the correlation matrix
        for i in range(n):
            for j in range(i+1, n):
                Psi[i, j] = np.exp(-sum(theta*(self.xs[i]-self.xs[j])**2))
        # add upper and lower halves and diagonal of ones
        # plus a small number to reduce ill conditioning
        Psi = Psi + Psi.T + np.eye(n) * (1+eps)

        # cholesky factorisation
        # added try/except block to capture error and implement penalty
        try:
            U = npla.cholesky(Psi).T
        except npla.LinAlgError:
            return 1000, Psi, np.zeros([n, n])
        # Forrester et al. have a penalty here if ill-conditioned
        # but this is not implemented in numpy.linalg.cholesky

        # Sum lns of diagonal to find ln(det(Psi))
        LnDetPsi = 2*sum(np.log(np.abs(np.diag(U))))

        # use back-substitution of Cholesky instead of inverse
        mu = np.dot(one, npla.solve(U, npla.solve(U.T, self.ys))) /\
            np.dot(one, npla.solve(U, npla.solve(U.T, one)))
        ysMuTemp = self.ys - mu  # only calculate this once
        SigmaSqr = (np.dot(ysMuTemp,
                           npla.solve(U,
                                      npla.solve(U.T, ysMuTemp)))*self.overn)
        NegLnLike = -1*(-(0.5*n)*np.log(SigmaSqr)-0.5*LnDetPsi)
        return NegLnLike, Psi, U

    def f(self, xs):
        # initialise theta
        theta = np.e**np.array(self.Theta)
        # calculate number of sample points
        n = len(self.ys)
        # create vector of ones
        one = np.ones([n])
        # calculate mu
        mu = np.dot(one, npla.solve(self.U, npla.solve(self.U.T, self.ys))) /\
            np.dot(one, npla.solve(self.U, npla.solve(self.U.T, one)))
        # initialise psi
        psi = np.ones([n])
        # fill psi vector
        for i in range(n):
            psi[i] = np.exp(-sum(theta*np.abs(self.xs[i]-xs)**2))
        return mu+np.dot(psi.T,
                         npla.solve(self.U, npla.solve(self.U.T, self.ys-mu)))

    def lb(self, xs):
        # initialise theta
        theta = np.e**np.array(self.Theta)
        # intialise A
        if not hasattr(self, "A"):
            self.A = 2
        # calculate number of sample points
        n = len(self.ys)
        # create vector of ones
        one = np.ones([n])
        # calculate mu
        mu = np.dot(one, npla.solve(self.U, npla.solve(self.U.T, self.ys))) /\
            np.dot(one, npla.solve(self.U, npla.solve(self.U.T, one)))
        # calculate sigma^2
        ysMuTemp = self.ys - mu  # only calculate this once
        SigmaSqr = np.dot(ysMuTemp,
                          npla.solve(self.U,
                                     npla.solve(self.U.T,
                                                ysMuTemp)))*self.overn
        # initialise psi
        psi = np.ones([n])
        # fill psi vector
        for i in range(n):
            psi[i] = np.exp(-sum(theta*np.abs(self.xs[i]-xs)**2))
        # calculate prediction
        f = mu + np.dot(psi.T, npla.solve(self.U,
                                          npla.solve(self.U.T, ysMuTemp)))
        # error
        SSqr = SigmaSqr*(1-np.dot(psi,
                                  npla.solve(self.U,
                                             npla.solve(self.U.T, psi))))
        # lower bound
        return f - self.A * np.sqrt(SSqr)

    def ei(self, xs):
        # define the error function as it's missing from python
        def erf(x):
            # save the sign of x
            sign = 1 if x >= 0 else -1
            x = np.abs(x)

            # constants
            a1 = 0.254829592
            a2 = -0.284496736
            a3 = 1.421413741
            a4 = -1.453152027
            a5 = 1.061405429
            p = 0.3275911

            # A&S formula 7.1.26
            t = 1.0/(1.0 + p*x)
            y = 1.0 - (((((a5*t + a4)*t) + a3)*t + a2)*t + a1)*t*np.exp(-x*x)
            return sign*y  # erf(-x) = -erf(x)
        # initialise theta
        theta = np.e**np.array(self.Theta)
        # intialise A
        if not hasattr(self, "A"):
            self.A = 2
        # calculate number of sample points
        n = len(self.ys)
        # create vector of ones
        one = np.ones([n])
        # calculate mu
        mu = np.dot(one, npla.solve(self.U, npla.solve(self.U.T, self.ys))) /\
            np.dot(one, npla.solve(self.U, npla.solve(self.U.T, one)))
        # calculate sigma^2
        ysMuTemp = self.ys - mu  # only calculate this once
        SigmaSqr = np.dot(ysMuTemp,
                          npla.solve(self.U,
                                     npla.solve(self.U.T,
                                                ysMuTemp)))*self.overn
        # initialise psi
        psi = np.ones([n])
        # fill psi vector
        for i in range(n):
            psi[i] = np.exp(-sum(theta*np.abs(self.xs[i]-xs)**2))
        # calculate prediction
        f = mu+np.dot(psi.T, npla.solve(self.U,
                                        npla.solve(self.U.T, ysMuTemp)))
        y_hat = f
        # error
        SSqr = SigmaSqr*(1-np.dot(psi,
                                  npla.solve(self.U,
                                             npla.solve(self.U.T, psi))))
        # find best so far:
        y_min = min(self.ys)
        # expected improvement
        if SSqr == 0:
            return 0
        else:
            sqrtAbsSSqr = np.sqrt(np.abs(SSqr))  # only calculate this once
            yDiff = y_min - y_hat  # only calculate this once
            ei_term1 = yDiff *\
                (0.5+0.5*erf((0.70710678)*(yDiff/sqrtAbsSSqr)))
            ei_term2 = sqrtAbsSSqr *\
                (0.39894228)*np.exp(-0.5*(yDiff**2/SSqr))
            return ei_term1 + ei_term2


# optimisation algorithms
def ga1(f, chroms, pops=20, gens=100, mutationrate=0.6):
    """
␣␣␣␣A␣genetic␣algorithm␣for␣finding␣the␣maximum␣of␣f.␣␣Uses␣a␣roulette␣wheel
␣␣␣␣selection␣method␣and␣both␣crossover␣and␣mutation␣to␣introduce␣variation.
␣␣␣␣Elite␣selection␣is␣also␣used␣to␣preserve␣the␣best␣individual␣found␣so␣far.
␣␣␣␣Default␣population␣size␣is␣20␣and␣number␣of␣generations␣is␣100.
␣␣␣␣The␣function␣f␣should␣be␣such␣that␣it␣accepts␣one␣input␣of␣a␣list␣that␣is
␣␣␣␣the␣chromosome␣of␣the␣individual␣and␣returns␣the␣'fitness'.
␣␣␣␣The␣input␣parameter␣chroms␣should␣be␣a␣list␣of␣function␣input␣ranges␣in␣the
␣␣␣␣form:␣[(0,1),(0,1)]␣if␣the␣function␣has␣2␣inputs,␣both␣constrained␣between
␣␣␣␣0␣and␣1.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣f:␣␣function
␣␣␣␣␣␣␣␣Function␣to␣be␣maximised,␣that␣takes␣a␣single␣argument,
␣␣␣␣␣␣␣␣which␣may␣be␣iterable␣for␣multi-dimensional␣functions.
␣␣␣␣chroms:␣list␣of␣two-tuples
␣␣␣␣␣␣␣␣A␣data␣structure␣that␣defines␣the␣number␣of␣dimensions␣by␣len(chroms)
␣␣␣␣␣␣␣␣and␣the␣lower␣and␣upper␣bounds␣of␣each␣dimension.
␣␣␣␣pops:␣␣␣number
␣␣␣␣␣␣␣␣Number␣of␣individuals␣in␣the␣population,␣default␣is␣20.
␣␣␣␣gens:␣␣␣number
␣␣␣␣␣␣␣␣Number␣of␣generations␣of␣populations,␣default␣is␣100.
␣␣␣␣mutationrate:␣␣␣number
␣␣␣␣␣␣␣␣Rate␣of␣mutation␣expressed␣in␣the␣range␣(0,␣1),␣default␣is␣0.6.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣indiout:␣␣␣␣list␣of␣numbers
␣␣␣␣␣␣␣␣The␣'chromosome'␣of␣the␣fittest␣individual␣found.
␣␣␣␣fitness:␣␣␣␣number
␣␣␣␣␣␣␣␣The␣fitness␣of␣the␣output␣individual.␣␣i.e.␣f(indiout).
␣␣␣␣history:␣␣␣␣list␣of␣numbers
␣␣␣␣␣␣␣␣The␣optimisation␣history,␣taking␣the␣maximum␣fitness␣in␣each␣generation
␣␣␣␣␣␣␣␣and␣thus␣returning␣a␣list␣of␣length␣gens.
␣␣␣␣"""
    import random

    #set-up history
    hist = []

    # generate initial generation
    parents = []
    for i in range(pops):
        indi = []
        for chrom in chroms:
            indi.append(random.uniform(chrom[0], chrom[1]))
        parents.append(indi)

    # calculate fitnesses
    fits = []
    for parent in parents:
        fits.append(f(parent))
    hist.append(max(fits))

    # begin main loop over generations
    for gen in range(gens-1):
        children = []

        # elite selection
        elites = 0
        elitemax = max(fits)
        for i in range(len(fits)):
            if fits[i] == elitemax and elites < 2:
                children.append(parents[i])

        # select remaining population
        while len(children) < pops:

            # roulette wheel selection
            p1r = random.random()
            p2r = random.random()
            # hack to prevent failure if parents aren't found
            p1i = 0
            p2i = 1
            sofar = 0
            total = sum(fits)
            for i in range(len(fits)):
                if p1r > sofar/float(total):
                    p1i = i
                if p2r > sofar/float(total):
                    p2i = i
                sofar += fits[i]

            # crossover
            cp = int(random.random()*len(chroms))
            child = parents[p1i][:cp]+parents[p2i][cp:]

            # mutation
            if random.random() < mutationrate:
                mp = int(random.random()*len(chroms))
                child[mp] = random.uniform(chroms[mp][0], chroms[mp][1])

            # add to population
            children.append(child)

        # progress one generation and recalculate fitness
        parents = children
        for i in range(pops):
            fits[i] = f(parents[i])

        # store history
        hist.append(max(fits))

    # find best of final generation
    for i in range(pops):
        if fits[i] == hist[-1]:
            indiout = parents[i]

    return indiout, hist[-1], hist


def ga2(f, chroms, pops=20, gens=100, mutationrate=0.6, offset=1, seed=False):
    """
    A genetic algorithm for finding the maximum of f.  Uses a roulette wheel
    selection method and both crossover and mutation to introduce variation.
    Elite selection is also used to preserve the best individual found so far.
    Default population size is 20 and number of generations is 100.
    The function f should be such that it accepts one input of a list that is
    the chromosome of the individual and returns the 'fitness'.
    The input parameter chroms should be a list of function input ranges in the
    form: [(0,1),(0,1)] if the function has 2 inputs, both constrained between
    0 and 1.
    Adapted from ga1 to include an offset so that:
        a) the function is minimised instead of maximised.
        b) negative values of fitness are suitable.
    Also limited positive values such that offset-f(x) is capped at 0.001,
    thus keeping a small chance for all individuals to be selected.
    Consqeuently the offset should be set to the maximum (+ve) value expected.
    Added the option to 'seed' the initial population with the placement
    of a predefined individual.


    Parameters
    ----------
    f:  function
        Function to be minimised, that takes a single argument,
        which may be iterable for multi-dimensional functions.
    chroms: list of two-tuples
        A data structure that defines the number of dimensions by len(chroms)
        and the lower and upper bounds of each dimension.
    pops:   number
        Number of individuals in the population, default is 20.
    gens:   number
        Number of generations of populations, default is 100.
    mutationrate:   number
        Rate of mutation expressed in the range (0, 1), default is 0.6.
    offset: number
        Amount by which to offset fitness to enable minimisation.
    seed:   list of numbers
        A seed individual to include in the first population.

    Returns
    -------
    indiout:    list of numbers
        The 'chromosome' of the fittest individual found.
    fitness:    number
        The fitness of the output individual.  i.e. f(indiout).
    history:    list of numbers
        The optimisation history, taking the maximum fitness in each generation
        and thus returning a list of length gens.
    """
    import random

    #set-up history
    hist = []

    # generate initial generation
    parents = []
    if seed:
        parents.append(seed)
        for i in range(pops-1):
            indi = []
            for chrom in chroms:
                indi.append(random.uniform(chrom[0], chrom[1]))
            parents.append(indi)
    else:
        for i in range(pops):
            indi = []
            for chrom in chroms:
                indi.append(random.uniform(chrom[0], chrom[1]))
            parents.append(indi)

    # calculate fitnesses
    fits = []
    for parent in parents:
        fits.append(max(offset-f(parent), 0.001))
    hist.append(offset - max(fits))

    # begin main loop over generations
    for gen in range(gens-1):
        children = []

        # elite selection
        elites = 0
        elitemax = max(fits)
        for i in range(len(fits)):
            if fits[i] == elitemax and elites < 2:
                children.append(parents[i])

        # select remaining population
        while len(children) < pops:

            # roulette wheel selection
            p1r = random.random()
            p2r = random.random()
            sofar = 0
            total = sum(fits)
            for i in range(len(fits)):
                if p1r > sofar/float(total):
                    p1i = i
                if p2r > sofar/float(total):
                    p2i = i
                sofar += fits[i]

            # crossover
            cp = int(random.random()*len(chroms))
            child = parents[p1i][:cp]+parents[p2i][cp:]

            # mutation
            if random.random() < mutationrate:
                mp = int(random.random()*len(chroms))
                child[mp] = random.uniform(chroms[mp][0], chroms[mp][1])

            # add to population
            children.append(child)

        # progress one generation and recalculate fitness
        parents = children
        for i in range(pops):
            fits[i] = max(offset-f(parents[i]), 0.001)

        # store history
        hist.append(offset-max(fits))

    # find best of final generation
    for i in range(pops):
        if offset - fits[i] == hist[-1]:
            indiout = parents[i]

    return indiout, hist[-1], hist


def ga3(f, chroms,
        pops=20, gens=100, tournamentSize=0.4, mutationrate=0.6, seed=False):
    """
    A genetic algorithm for finding the minimum of f.  Uses a tournament
    selection method and both crossover and mutation to introduce variation.
    Elite selection is also used to preserve the best individual found so far.
    Default population size is 20 and number of generations is 100.
    The default tournament size is 40 percent of the population.
    The function f should be such that it accepts one input of a list that is
    the chromosome of the individual and returns the 'fitness'.
    The input parameter chroms should be a list of function input ranges in the
    form: [(0,1),(0,1)] if the function has 2 inputs, both constrained between
    0 and 1.
    Includes the option to 'seed' the initial population with the placement
    of a predefined individual.

    Parameters
    ----------
    f:  function
        Function to be minimised, that takes a single argument,
        which may be iterable for multi-dimensional functions.
    chroms: list of two-tuples
        A data structure that defines the number of dimensions by len(chroms)
        and the lower and upper bounds of each dimension.
    pops:   number
        Number of individuals in the population, default is 20.
    gens:   number
        Number of generations of populations, default is 100.
    tournamentSize: number
        Size of tournament as a proportion of the population, default is 0.4.
    mutationrate:   number
        Rate of mutation expressed in the range (0, 1), default is 0.6.
    seed:   list of numbers
        A seed individual to include in the first population.

    Returns
    -------
    indiout:    list of numbers
        The 'chromosome' of the fittest individual found.
    fitness:    number
        The fitness of the output individual.  i.e. f(indiout).
    history:    list of numbers
        The optimisation history, taking the maximum fitness in each generation
        and thus returning a list of length gens.
    """
    import random

    # set-up history
    hist = []

    # set up tournament size as integer
    nTournament = int(tournamentSize*pops)

    # generate initial generation
    parents = []
    if seed:
        parents.append(seed)
        for i in range(pops-1):
            indi = []
            for chrom in chroms:
                indi.append(random.uniform(chrom[0], chrom[1]))
            parents.append(indi)
    else:
        for i in range(pops):
            indi = []
            for chrom in chroms:
                indi.append(random.uniform(chrom[0], chrom[1]))
            parents.append(indi)

    # calculate fitnesses
    fits = []
    for parent in parents:
        fits.append(f(parent))
    hist.append(min(fits))

    # begin main loop over generations
    for gen in range(gens-1):
        children = []

        # elite selection
        elites = 0
        elitemin = min(fits)
        for i in range(len(fits)):
            if fits[i] == elitemin and elites < 2:
                children.append(parents[i])
                elites += 1
        # select remaining population
        while len(children) < pops:

            # tournament selection
            tournis = random.sample(range(pops), nTournament)
            p1i = tournis[0]
            p2i = tournis[0]
            # pick the two best parents in the tournament
            for tourni in tournis[1:]:
                if fits[tourni] < fits[p1i]:
                    p1i = tourni
                elif fits[tourni] < fits[p2i]:
                    p2i = tourni

            # crossover
            cp = int(random.random()*len(chroms))
            child = parents[p1i][:cp]+parents[p2i][cp:]

            # mutation
            if random.random() < mutationrate:
                mp = int(random.random()*len(chroms))
                child[mp] = random.uniform(chroms[mp][0], chroms[mp][1])

            # add to population
            children.append(child)

        # progress one generation and recalculate fitness
        parents = children
        for i in range(pops):
            fits[i] = f(parents[i])

        # store history
        hist.append(min(fits))

    # find best of final generation
    for i in range(pops):
        if fits[i] == hist[-1]:
            indiout = parents[i]

    return indiout, hist[-1], hist


def ga3cc(f, chroms,
          pops=20, cgens=5, tournamentSize=0.4, mutationrate=0.6, seed=False):
    """
␣␣␣␣A␣genetic␣algorithm␣for␣finding␣the␣minimum␣of␣f.␣␣Uses␣a␣tournament
␣␣␣␣selection␣method␣and␣both␣crossover␣and␣mutation␣to␣introduce␣variation.
␣␣␣␣Elite␣selection␣is␣also␣used␣to␣preserve␣the␣best␣individual␣found␣so␣far.
␣␣␣␣Default␣population␣size␣is␣20␣and␣will␣continue␣iterating␣until
␣␣␣␣no␣improvement␣is␣seen␣for␣cgens␣generations.
␣␣␣␣The␣default␣tournament␣size␣is␣40␣percent␣of␣the␣population.
␣␣␣␣The␣function␣f␣should␣be␣such␣that␣it␣accepts␣one␣input␣of␣a␣list␣that␣is
␣␣␣␣the␣chromosome␣of␣the␣individual␣and␣returns␣the␣'fitness'.
␣␣␣␣The␣input␣parameter␣chroms␣should␣be␣a␣list␣of␣function␣input␣ranges␣in␣the
␣␣␣␣form:␣[(0,1),(0,1)]␣if␣the␣function␣has␣2␣inputs,␣both␣constrained␣between
␣␣␣␣0␣and␣1.
␣␣␣␣Includes␣the␣option␣to␣'seed'␣the␣initial␣population␣with␣the␣placement
␣␣␣␣of␣a␣predefined␣individual.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣f:␣␣function
␣␣␣␣␣␣␣␣Function␣to␣be␣minimised,␣that␣takes␣a␣single␣argument,
␣␣␣␣␣␣␣␣which␣may␣be␣iterable␣for␣multi-dimensional␣functions.
␣␣␣␣chroms:␣list␣of␣two-tuples
␣␣␣␣␣␣␣␣A␣data␣structure␣that␣defines␣the␣number␣of␣dimensions␣by␣len(chroms)
␣␣␣␣␣␣␣␣and␣the␣lower␣and␣upper␣bounds␣of␣each␣dimension.
␣␣␣␣pops:␣␣␣number
␣␣␣␣␣␣␣␣Number␣of␣individuals␣in␣the␣population,␣default␣is␣20.
␣␣␣␣cgens:␣␣number
␣␣␣␣␣␣␣␣Number␣of␣generations␣of␣no␣improvement␣before␣considered␣converged.
␣␣␣␣␣␣␣␣Default␣is␣5.
␣␣␣␣tournamentSize:␣number
␣␣␣␣␣␣␣␣Size␣of␣tournament␣as␣a␣proportion␣of␣the␣population,␣default␣is␣0.4.
␣␣␣␣mutationrate:␣␣␣number
␣␣␣␣␣␣␣␣Rate␣of␣mutation␣expressed␣in␣the␣range␣(0,␣1),␣default␣is␣0.6.
␣␣␣␣seed:␣␣␣list␣of␣numbers
␣␣␣␣␣␣␣␣A␣seed␣individual␣to␣include␣in␣the␣first␣population.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣indiout:␣␣␣␣list␣of␣numbers
␣␣␣␣␣␣␣␣The␣'chromosome'␣of␣the␣fittest␣individual␣found.
␣␣␣␣fitness:␣␣␣␣number
␣␣␣␣␣␣␣␣The␣fitness␣of␣the␣output␣individual.␣␣i.e.␣f(indiout).
␣␣␣␣history:␣␣␣␣list␣of␣numbers
␣␣␣␣␣␣␣␣The␣optimisation␣history,␣taking␣the␣maximum␣fitness␣in␣each␣generation
␣␣␣␣␣␣␣␣and␣thus␣returning␣a␣list␣of␣length␣gens.
␣␣␣␣"""
    import random
```

```
# set-up history
hist = []

# set up tournament size as integer
nTournament = int(tournamentSize*pops)

# generate initial generation
parents = []
if seed:
    parents.append(seed)
    for i in range(pops-1):
        indi = []
        for chrom in chroms:
            indi.append(random.uniform(chrom[0], chrom[1]))
        parents.append(indi)
else:
    for i in range(pops):
        indi = []
        for chrom in chroms:
            indi.append(random.uniform(chrom[0], chrom[1]))
        parents.append(indi)

# calculate fitnesses
fits = []
for parent in parents:
    fits.append(f(parent))
hist.append(min(fits))

# set convergence counter to 0
convcount = 0

# begin main loop over generations
while convcount < cgens:
    # increment counter
    convcount += 1

    children = []

    # elite selection
    elites = 0
    elitemin = min(fits)
    for i in range(len(fits)):
        if fits[i] == elitemin and elites < 2:
            children.append(parents[i])
            elites += 1

    # select remaining population
    while len(children) < pops:

        # tournament selection
        tournis = random.sample(range(pops), nTournament)
        p1i = tournis[0]
        p2i = tournis[0]
        # pick the two best parents in the tournament
        for tourni in tournis[1:]:
            if fits[tourni] < fits[p1i]:
                p1i = tourni
            elif fits[tourni] < fits[p2i]:
                p2i = tourni

        # crossover
        cp = int(random.random()*len(chroms))
        child = parents[p1i][:cp]+parents[p2i][cp:]

        # mutation
        if random.random() < mutationrate:
            mp = int(random.random()*len(chroms))
            child[mp] = random.uniform(chroms[mp][0], chroms[mp][1])

        # add to population
        children.append(child)

    # progress one generation and recalculate fitness
    parents = children
    for i in range(pops):
        fits[i] = f(parents[i])

    # check for convergence
    currentBest = min(fits)
    if currentBest < hist[-1]:
        convcount = -1

    # store history
    hist.append(currentBest)

# find best of final generation
for i in range(pops):
    if fits[i] == hist[-1]:
        indiout = parents[i]

return indiout, hist[-1], hist
```

```
def ga3bacc(f, chroms, pops=20, cgens=5,
            tournamentSize=0.4, mutationrate=0.6, seed=False):
    """
    A genetic algorithm for finding the minimum of f.
    Designed for objective functions accepting boolean array as an input.
    Uses a tournament selection method and both crossover and mutation to
    introduce variation.
    Elite selection is also used to preserve the best individual found so far.
    Default population size is 20 and will continue iterating until
    no improvement is seen for cgens generations.
    The default tournament size is 40 percent of the population.
    The function f should be such that it accepts one input of a list that is
    the chromosome of the individual and returns the 'fitness'.
    The input parameter chroms should be the length of boolean array that
    f takes as input.
    Includes the option to 'seed' the initial population with the placement
    of a predefined individual.


    Parameters
    ----------
    f:  function
        Function to be minimised, that takes a single argument,
        which is an array of booleans.
    chroms: number
        The length of boolean array that f accepts.
    pops:   number
        Number of individuals in the population, default is 20.
    cgens:  number
        Number of generations of no improvement before considered converged.
        Default is 5.
    tournamentSize: number
        Size of tournament as a proportion of the population, default is 0.4.
    mutationrate:   number
        Rate of mutation expressed in the range (0, 1), default is 0.6.
    seed:   list of numbers
        A seed individual to include in the first population.

    Returns
    -------
    indiout:    list of numbers
        The 'chromosome' of the fittest individual found.
    fitness:    number
        The fitness of the output individual.  i.e. f(indiout).
    history:    list of numbers
        The optimisation history, taking the maximum fitness in each generation
        and thus returning a list of length gens.
    """
    import random

    # set-up history
    hist = []

    # set up tournament size as integer
    nTournament = int(tournamentSize*pops)

    # generate initial generation
    parents = np.zeros([pops, chroms], dtype=bool)
    if seed:
        parents[0] = seed
        for i in xrange(1, pops):
            for j in xrange(chroms):
                parents[i, j] = random.randint(0, 1)
    else:
        for i in xrange(pops):
            for j in xrange(chroms):
                parents[i, j] = random.randint(0, 1)

    # calculate fitnesses
    fits = np.zeros(pops)
    for i in xrange(pops):
        fits[i] = f(parents[i])
    hist.append(min(fits))

    # set convergence counter to 0
    convcount = 0

    # begin main loop over generations
    while convcount < cgens:
        # increment counter
        convcount += 1

        children = []

        # elite selection
        elites = 0
        elitemin = min(fits)
        for i in range(len(fits)):
            if fits[i] == elitemin and elites < 2:
                children.append(parents[i])
                elites += 1
        # select remaining population
        while len(children) < pops:

            # tournament selection
            tournis = random.sample(range(pops), nTournament)
            p1i = tournis[0]
            p2i = tournis[0]
            # pick the two best parents in the tournament
            for tourni in tournis[1:]:
                if fits[tourni] < fits[p1i]:
                    p1i = tourni
                elif fits[tourni] < fits[p2i]:
                    p2i = tourni

            # crossover
            cp = int(random.random()*chroms)
            child = np.hstack((parents[p1i][:cp], parents[p2i][cp:]))

            # mutation
            if random.random() < mutationrate:
                mp = int(random.random()*chroms)
                child[mp] = bool(random.randint(0, 1))

            # add to population
            children.append(child)

        # progress one generation and recalculate fitness
        parents = np.array(children, dtype=bool)
        for i in range(pops):
            fits[i] = f(parents[i])

        # check for convergence
        currentBest = min(fits)
        if currentBest < hist[-1]:
            convcount = -1

        # store history
        hist.append(currentBest)

    # find best of final generation
    for i in range(pops):
        if fits[i] == hist[-1]:
            indiout = parents[i]

    return indiout, hist[-1], hist


def derivativen(f, x, xn, eps=1e-6):
    """
␣␣␣␣Calculates␣the␣derivative␣of␣f(x)␣with␣respect␣to␣the␣xnth␣dimension␣of␣x.
␣␣␣␣Uses␣the␣central␣difference␣method␣with␣a␣step␣size␣of␣eps,␣default␣1e-6.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣f:␣␣function
␣␣␣␣␣␣␣␣Function␣to␣calculate␣the␣derivative␣of.
␣␣␣␣x:␣␣list␣or␣array
␣␣␣␣␣␣␣␣Location␣at␣which␣to␣calculate␣the␣derivative.
␣␣␣␣xn:␣integer
␣␣␣␣␣␣␣␣The␣dimension␣on␣which␣to␣calculate␣the␣(partial)␣derivative.
␣␣␣␣eps:␣␣␣␣number␣␣(default␣1e-6)
␣␣␣␣␣␣␣␣The␣step␣size␣to␣use␣in␣the␣central␣difference␣method.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣dfdxn:␣␣number
␣␣␣␣␣␣␣␣Partial␣derivative␣of␣f␣with␣respect␣to␣the␣xnth␣dimension␣at␣x.
␣␣␣␣"""
    # create copies of x
    xplus = x[:]
    xminus = x[:]
    # augment copies of x
    xplus[xn] += eps
    xminus[xn] -= eps
    # calculate and return derivative
    return (f(xplus)-f(xminus))/(2*eps)


def derivative2n(f, x, xn, eps=1e-6, d1f=derivativen):
    """
␣␣␣␣Calculates␣the␣second␣derivative␣of␣f(x)␣with␣respect␣to␣the␣xnth
␣␣␣␣dimension␣of␣x,␣using␣a␣central␣difference␣method␣with␣a␣step␣size␣of␣eps,
␣␣␣␣calculating␣the␣first␣derivative␣using␣the␣function␣d1f,

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣f:␣␣function
␣␣␣␣␣␣␣␣Function␣to␣calculate␣the␣derivative␣of.
␣␣␣␣x:␣␣list␣or␣array
␣␣␣␣␣␣␣␣Location␣at␣which␣to␣calculate␣the␣derivative.
␣␣␣␣xn:␣integer
```

```
␣␣␣␣␣␣␣␣The␣dimension␣on␣which␣to␣calculate␣the␣(partial)␣derivative.
␣␣␣␣eps:␣␣␣␣number␣␣(default␣1e-6)
␣␣␣␣␣␣␣␣The␣step␣size␣to␣use␣in␣the␣central␣difference␣method.
␣␣␣␣d1f:␣␣␣␣function␣␣␣␣(default␣derivativen)
␣␣␣␣␣␣␣␣Function␣to␣calculate␣the␣first␣dervivative.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣d2fdxn2:␣␣number
␣␣␣␣␣␣␣␣Second␣partial␣derivative␣of␣f␣with␣respect␣to␣the␣xnth␣dimension␣at␣x.
␣␣␣␣"""
    # create copies of x
    xplus = x[:]
    xminus = x[:]
    # augment copies of x
    xplus[xn] += eps
    xminus[xn] -= eps
    # calculate and return derivative
    return (d1f(f, xplus, xn) - d1f(f, xminus, xn)) / (2*eps)


def multiNewton(f, initial, limits, sweeps=3):
    """
␣␣␣␣Optimises␣a␣multidimensional␣function␣by␣sequentially␣finding␣linear
␣␣␣␣optima␣in␣each␣dimension.
␣␣␣␣N.B.␣function␣f␣is␣evaluated␣a␣total␣of␣sweeps*dimensions*6␣times.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣f:␣␣function
␣␣␣␣␣␣␣␣Function␣to␣be␣optimised␣(minimised).
␣␣␣␣initial:␣␣␣␣list␣or␣array
␣␣␣␣␣␣␣␣Initial␣value␣to␣begin␣the␣optimisation␣at.
␣␣␣␣limits:␣list␣of␣two-tuples
␣␣␣␣␣␣␣␣Lists␣the␣lower␣and␣upper␣bound␣of␣each␣dimension␣in␣a␣tuple.
␣␣␣␣sweeps:␣integer␣(default␣3)
␣␣␣␣␣␣␣␣Number␣of␣times␣to␣sweep␣each␣dimension.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣x:␣␣list␣or␣array
␣␣␣␣␣␣␣␣Location␣of␣optimum␣value.
␣␣␣␣f(x):␣␣␣number
␣␣␣␣␣␣␣␣Optimum␣value.
␣␣␣␣hist:␣␣␣list
␣␣␣␣␣␣␣␣Optimisation␣history.
␣␣␣␣"""

    # define a function to check whether a proposed x value is within limits
    def checklimits(x, limits):
        """Function␣to␣check␣whether␣x␣is␣within␣limits.
␣␣␣␣␣␣␣␣-␣uses␣a␣recursive␣algorithm."""
        if len(x) == 0:
            return True
        if x[0] >= limits[0][0] and x[0] <= limits[0][1]:
            return checklimits(x[1:], limits[1:])
        return False

    # check initial value is within these limits
    if not checklimits(initial, limits):
        print "Error␣in␣function␣multiNewton,␣initial␣value␣not␣within␣limits"
        return -1

    hist = [initial[:]]
    x = initial[:]
    # perform stated number of sweeps
    for i in range(sweeps):
        # loop over all dimensions
        for j in range(len(initial)):
                # this section could be optimised to evaluate f only 3 times
                x[j] -= derivativen(f, x, j) / derivative2n(f, x, j)
                hist.append(x[:])
    return x, f(x), hist


# sample plan space filling metrics
def sampleplan_mean_distance(sampleplan):
    """
␣␣␣␣Returns␣the␣mean␣distance␣between␣points␣in␣a␣sample␣plan.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣sampleplan:␣n*k␣array
␣␣␣␣␣␣␣␣Sample␣plan␣to␣calculate␣the␣mean␣distance␣of.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣mean_distance:␣␣number
␣␣␣␣␣␣␣␣Mean␣distance␣between␣points␣in␣the␣sample␣plan.
␣␣␣␣"""
    mean_distance = 0
    n = len(sampleplan)
    total_measured = 0
    for i in range(n-1):
        for j in range(i + 1, n):
            mean_distance += npla.norm(sampleplan[i] - sampleplan[j])
            total_measured += 1
    mean_distance /= total_measured
    return mean_distance


def morris_mitchell_phi(sampleplan, q=2, euclidean=True):
    """
    Calculates the sampling plan quality criterion of Morris and Mitchell.
    From Forrester et al. Chapter 1.

    Parameters
    ----------
    sampleplan: 2d array
        An n by k array of the sample plan.  Where n is the number of points
        and k is the number of dimensions.
    q:  number (default 2)
        Exponent used in the calculation of the metric.
    euclidean:  bool (default True)
        Whether to use the Euclidean distance metric or rectangular.

    Returns
    -------
    phiq:   number
        Sampling plan space-fillingness metric.
    """
    # number of points in sampling plan
    n = len(sampleplan)

    # compute the distances between all pairs of points
    d = np.zeros(n*(n-1)/2.0)
    for i in range(n-1):
        for j in range(i+1, n):
            # d[(i-1)*n-(i-1)*i/2+j-i] is the original matlab here
            if euclidean:
                d[(i)*n-(i)*(i+1)/2+j-i-1] = npla.norm(sampleplan[i] -
                                                       sampleplan[j])
            if not euclidean:
                d[(i)*n-(i)*(i+1)/2+j-i-1] = npla.norm(sampleplan[i] -
                                                       sampleplan[j], 1)

    # remove multiple occurrences
    dd = np.unique(d)

    # preallocate memory for J
    J = np.zeros(len(dd))

    # generate multiplicity array
    for i in range(len(dd)):
        # J[i] = sum(ismember(d, dd[i])) is the original matlab here
        J[i] = sum(map(lambda x: x == dd[i], d))

    # the sampling plan quality criterion
    phiq = sum(J*(dd**(-q)))**(1.0/q)

    return phiq


# sampling plans
def randlh(k, n, edges=False):
    """
    Returns an random latin hypercube with k dimensions
    and n points in a structure xs[n][k].
    All dimensions are normalised between 0 and 1.

    Parameters
    ----------
    k:  number
        Number of dimensions.
    n:  number
        Number of points in the latin hypercube.
    edges:  bool (default False)
        Whether or not to use edge points at 0 and 1.

    Returns
    -------
    samplexs:   2d array
        An n by k array of sample points in the given space.

    Example
    -------
    >>> randlh(2, 2)
    [[0.25, 0.25], [0.75, 0.75]]
    """
    from random import randint

    samplexs = np.zeros([n, k])

    # create k by n dimensional sampling list - to be popped at random.
    popper = []
    for i in range(k):
        popper.append(range(n))

    # create latin hypercube
    for i in range(n):
        for j in range(k):
            samplexs[i, j] = popper[j].pop(randint(0, len(popper[j]) - 1))
            # and normalise to 1
            if edges:
                samplexs[i, j] /= float(n - 1)
            elif not edges:
                samplexs[i, j] = (samplexs[i, j] + 0.5) / float(n)

    return samplexs


def bestlh(k, n, n_hypercubes=50, edges=False,
           space_fillingness=morris_mitchell_phi):
    """
    Generates a number of random latin hypercubes
    and picks the best one based on maximum space fillingness.

    Parameters
    ----------
    k:  integer
        Number of dimensions.
    n:  integer
        Number of points in the latin hypercube.
    n_hypercubes:   integer (default 50)
        Number of hypercubes to generate to pick the best one.
    edges:  bool (default False)
        Whether or not to use edge points at 0 and 1.
    space_fillingness:  function    (default morris_mitchell_phi)
        Function that defines the space fillingness of a sample plan.
        This is the objective that is maximised.

    Returns
    -------
    samplexs:   2d array
        An n by k array of sample points in the given space.
    """
    currentxs = randlh(k, n, edges)
    newxs = randlh(k, n, edges)
    if space_fillingness(newxs) > space_fillingness(currentxs):
        currentxs = newxs[:]
    for i in range(n_hypercubes - 2):
        newxs = randlh(k, n, edges)
        if space_fillingness(newxs) > space_fillingness(currentxs):
            currentxs = newxs[:]

    return currentxs


def randsampleplan(k, n):
    """
    Returns a random sample plan.
    All dimensions are normalised between 0 and 1.

    Parameters
    ----------
    k:  integer
        Number of dimensions.
    n:  integer
        Number of points.

    Returns
    -------
    samplexs:   2d array
        An n by k array of sample points in the given space.
    """
    from random import random
    # previous code (not preallocating memory)
    #samplexs = []
    #for i in range(n):
    #     samplexs.append([])
    #     for j in range(k):
    #         samplexs[-1].append(random())
    #samplexs = np.array(samplexs)
    samplexs = np.zeros([n, k])
    for i in range(n):
        for j in range(k):
            samplexs[i, j] = random()
    return samplexs


def bestrandplan(k, n, n_plans=50,
                 space_fillingness=sampleplan_mean_distance):
    """
    Generates a number of random sample plans
    and picks the best one based on maximum space fillingness.
    Parameters
    ----------
    k:  integer
        Number of dimensions.
    n:  integer
        Number of points.
    n_plans:   integer (default 50)
        Number of random plans to generate to pick the best one.
    space_fillingness:  function    (default sampleplan_mean_distance)
        Function that defines the space fillingness of a sample plan.
        This is the objective that is maximised.

    Returns
    -------
    samplexs:   2d array
        An n by k array of sample points in the given space.
    """
    currentxs = randsampleplan(k, n)
    newxs = randsampleplan(k, n)
    if space_fillingness(newxs) > space_fillingness(currentxs):
        currentxs = newxs[:]
    for i in range(n_plans - 2):
        newxs = randsampleplan(k, n)
        if space_fillingness(newxs) > space_fillingness(currentxs):
            currentxs = newxs[:]

    return currentxs


def full2dsampleplan(n):
    """
    Returns a full sample plan for 2 dimensions with n points per dimension.
    Note this is a total of n^2 points.
    All dimensions are normalised between 0 and 1.

    Parameters
    ----------
    n:  integer
        Number of points per dimension.

    Returns
    -------
    samplexs:   2d array
        An n*n by 2 array of sample points in the given space.
    """
    samplexs = np.zeros([n**2, 2])
    values = np.linspace(0, 1, n)
    for i in range(n**2):
        samplexs[i, 0] = values[i//n]
        samplexs[i, 1] = values[i % n]
    return samplexs


def full_factoral_sampleplan(k, n_per_dim):
    """
    Returns a full factoral sample plan for k dimensions
    with n_per_dim points in each dimension.
    N.B. this is a total of n_per_dim**k points.
    All dimensions are normalised between 0 and 1.

    Parameters
    ----------
    k:  number
        Number of dimensions.
    n_per_dim:  number
        Number of points per dimension.

    Returns
    -------
    samplexs:   2d array
        Sample plan of xs.

    Example
    -------
    >>> full_factoral_sampleplan(2, 2)
    [[0, 0], [0, 1], [1, 0], [1, 1]]
    """
    totalxs = n_per_dim**k
    samplexs = np.zeros([totalxs, k])
    current_array = np.zeros(k)
    interval = 1.0/(n_per_dim - 1)
    for i in range(totalxs):
        samplexs[i] = current_array[:]
        for j in range(k):
            if current_array[-(j+1)] + interval <= 1.0:
                current_array[-(j+1)] += interval
                for l in range(j):
                    current_array[-(l+1)] = 0.0
                break
            else:
                pass
    return samplexs


# test functions and other utilities
def normdims(k):
    """
    Returns a list of tuples to give limits for k normalised dimensions.
    i.e. [(0,1)]*k

    Parameters
    ----------
    k:  integer
        Number of dimensions.

    Returns
    -------
    dims:   2d array
        An 2 by k array of normalised bounds i.e. [(0,1)]*k.
    """
    return [(0, 1)]*k


def onevar(xs):
    """
    Single variable test function.
    from Forrester, Sobester, Keane
    - Engineering Design via Surrogate Modelling.

    Parameters
    ----------
    xs: 1d array
        Value of x.

    Returns
    -------
    y:  number
        (6*x-2)**2 * np.sin(12*x-4).
    """
    x = xs[0]
    return (6*x-2)**2 * np.sin(12*x-4)


def branin(xs):
    """
    Two variable test function.
    from Forrester, Sobester, Keane
    - Engineering Design via Surrogate Modelling.

    Parameters
    ----------
    xs: 2 by 1d array
        Value of x.

    Returns
    -------
    y:  number
        Value of the branin function at x.
    """
    # convert 0,1 limits to x1<-[-5,10],x2<-[0,15]
    x1 = 15*xs[0]-5
    x2 = 15*xs[1]

    return (x2 - (5.1*x2)/(4*np.pi*np.pi) + (5*x1)/(np.pi) - 6)**2 +\
        10*((1 - 1/(8*np.pi))*np.cos(x1)+1) + 5*x1


def branin10(xs):
    """
    Ten variable test function.
    Linear summation of five two-dimensional branin functions
    from Forrester, Sobester, Keane
    - Engineering Design via Surrogate Modelling.

    Parameters
    ----------
    xs: 10 by 1d array
        Value of x.

    Returns
    -------
    y:  number
        Value of the branin10 function at x.
    """
    # convert 0,1 limits to x1<-[-5,10],x2<-[0,15]
    x1 = 15*xs[0]-5
    x2 = 15*xs[1]
    x3 = 15*xs[2]-5
    x4 = 15*xs[3]
    x5 = 15*xs[4]-5
    x6 = 15*xs[5]
    x7 = 15*xs[6]-5
    x8 = 15*xs[7]
    x9 = 15*xs[8]-5
    x10 = 15*xs[9]

    # create empty answer variable
    ans = 0.0

    ans += (x2 - (5.1*x2)/(4*np.pi*np.pi) + (5*x1)/(np.pi) - 6)**2 +\
        10*((1 - 1/(8*np.pi))*np.cos(x1)+1) + 5*x1
    ans += (x4 - (5.1*x4)/(4*np.pi*np.pi) + (5*x3)/(np.pi) - 6)**2 +\
        10*((1 - 1/(8*np.pi))*np.cos(x3)+1) + 5*x3
    ans += (x6 - (5.1*x6)/(4*np.pi*np.pi) + (5*x5)/(np.pi) - 6)**2 +\
        10*((1 - 1/(8*np.pi))*np.cos(x5)+1) + 5*x5
    ans += (x8 - (5.1*x8)/(4*np.pi*np.pi) + (5*x7)/(np.pi) - 6)**2 +\
        10*((1 - 1/(8*np.pi))*np.cos(x7)+1) + 5*x7
    ans += (x10 - (5.1*x10)/(4*np.pi*np.pi) + (5*x9)/(np.pi) - 6)**2 +\
        10*((1 - 1/(8*np.pi))*np.cos(x9)+1) + 5*x9
    return ans

_branin_noise_stddev = 10


def branin_noisy(xs):
    """
    Two variable test function.
    from Forrester, Sobester, Keane
    - Engineering Design via Surrogate Modelling.

    Parameters
    ----------
    xs: 2 by 1d array
        Value of x.

    Returns
    -------
    y:  number
        Value of the branin function at x with added noise.
    """
    from random import normalvariate
    # convert 0,1 limits to x1<-[-5,10],x2<-[0,15]
    x1 = 15*xs[0]-5
    x2 = 15*xs[1]

    ans = (x2 - (5.1*x2)/(4*np.pi*np.pi) + (5*x1)/(np.pi) - 6)**2 +\
        10*((1 - 1/(8*np.pi))*np.cos(x1)+1) + 5*x1
    return normalvariate(ans, _branin_noise_stddev)


def branin10_noisy(xs):
    """
    Ten variable test function.
    Linear summation of five two-dimensional branin functions
    from Forrester, Sobester, Keane
    - Engineering Design via Surrogate Modelling.

    Parameters
    ----------
    xs: 10 by 1d array
        Value of x.

    Returns
    -------
    y:  number
        Value of the branin10 function at x with added noise.
    """
    from random import normalvariate
    # convert 0,1 limits to x1<-[-5,10],x2<-[0,15]
    x1 = 15*xs[0]-5
    x2 = 15*xs[1]
    x3 = 15*xs[2]-5
    x4 = 15*xs[3]
    x5 = 15*xs[4]-5
    x6 = 15*xs[5]
    x7 = 15*xs[6]-5
    x8 = 15*xs[7]
    x9 = 15*xs[8]-5
    x10 = 15*xs[9]

    # create empty answer variable
    ans = 0.0

    ans += (x2 - (5.1*x2)/(4*np.pi*np.pi) + (5*x1)/(np.pi) - 6)**2 +\
        10*((1 - 1/(8*np.pi))*np.cos(x1)+1) + 5*x1
    ans += (x4 - (5.1*x4)/(4*np.pi*np.pi) + (5*x3)/(np.pi) - 6)**2 +\
        10*((1 - 1/(8*np.pi))*np.cos(x3)+1) + 5*x3
    ans += (x6 - (5.1*x6)/(4*np.pi*np.pi) + (5*x5)/(np.pi) - 6)**2 +\
        10*((1 - 1/(8*np.pi))*np.cos(x5)+1) + 5*x5
    ans += (x8 - (5.1*x8)/(4*np.pi*np.pi) + (5*x7)/(np.pi) - 6)**2 +\
        10*((1 - 1/(8*np.pi))*np.cos(x7)+1) + 5*x7
    ans += (x10 - (5.1*x10)/(4*np.pi*np.pi) + (5*x9)/(np.pi) - 6)**2 +\
        10*((1 - 1/(8*np.pi))*np.cos(x9)+1) + 5*x9
    return normalvariate(ans, _branin_noise_stddev)
```

```
def gatest1(xs):
    """
␣␣␣␣Function␣to␣test␣whether␣ga␣is␣working,
␣␣␣␣takes␣an␣input␣of␣a␣list␣of␣length␣8,
␣␣␣␣rounds␣the␣floating␣point␣values␣to␣integers,
␣␣␣␣and␣returns␣the␣sum␣of␣the␣list.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣xs:␣list␣or␣array
␣␣␣␣␣␣␣␣Value␣of␣x.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣y:␣␣number
␣␣␣␣␣␣␣␣sum(round(xs))
␣␣␣␣"""
    xs = map(round, xs)
    return sum(xs)


def gatest2(xs):
    """
␣␣␣␣A␣second␣test␣function␣for␣a␣ga,
␣␣␣␣returns␣the␣proximity␣to␣0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣xs:␣list␣or␣array␣of␣length␣8
␣␣␣␣␣␣␣␣Value␣of␣x.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣y:␣␣number
␣␣␣␣␣␣␣␣proximity␣to␣0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5.
␣␣␣␣"""
    return 8.0 - sum(map(lambda x: abs(x-0.5), xs))

gatestcr = normdims(8)


def popsweep1():
    """
␣␣␣␣Saves␣a␣figure␣of␣a␣population␣size␣sweep
␣␣␣␣of␣ga1␣using␣gatest2.
␣␣␣␣"""
    import pylab
    for k in range(10, 101, 10):
        x, y, hist = ga1(gatest2, gatestcr, pops=k)
        pylab.plot(hist, label=str(k))
    pylab.legend()
    pylab.savefig("popsweep1.png")
    pylab.clf()


def popsweep2():
    """
␣␣␣␣Saves␣a␣figure␣of␣a␣population␣size␣sweep
␣␣␣␣of␣ga1␣using␣gatest2.
␣␣␣␣"""
    import pylab
    aveys = []
    for k in range(1, 101, 3):
        ys = []
        for j in range(20):
            x, y, hist = ga1(gatest2, gatestcr, pops=k)
            ys.append(y)
        aveys.append(sum(ys)/float(len(ys)))
    pylab.plot(aveys, label="Average␣final␣fitness")
    pylab.legend()
    pylab.savefig("popsweep2.png")
    pylab.clf()


def gensweep1():
    """
␣␣␣␣Saves␣a␣figure␣of␣a␣generation␣number␣sweep
␣␣␣␣of␣ga1␣using␣gatest2.
␣␣␣␣"""
    import pylab
    aveys = []
    for k in range(1, 101, 2):
        ys = []
        for j in range(50):
            x, y, hist = ga1(gatest2, gatestcr, gens=k)
            ys.append(y)
        aveys.append(sum(ys)/float(len(ys)))
    pylab.plot(aveys, label="Average␣final␣fitness")
    pylab.legend()
    pylab.savefig("gensweep1.png")
    pylab.clf()
```

```
def mrsweep1():
    """
␣␣␣␣Saves␣a␣figure␣of␣a␣mutation␣rate␣sweep
␣␣␣␣of␣ga1␣using␣gatest2.
␣␣␣␣"""
    import pylab
    aveys = []
    for k in range(1, 100, 2):
        ys = []
        for j in range(50):
            x, y, hist = ga1(gatest2, gatestcr, mutationrate=k*0.01)
            ys.append(y)
        aveys.append(sum(ys)/float(len(ys)))
    pylab.plot(aveys, label="Average␣final␣fitness")
    pylab.legend()
    pylab.savefig("mrsweep1.png")
    pylab.clf()


def mntest(xs):
    """
␣␣␣␣A␣simple␣test␣function␣for␣the␣multiNewton␣optimisation␣method,
␣␣␣␣returns␣the␣proximity␣to␣0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣xs:␣list␣or␣array
␣␣␣␣␣␣␣␣Value␣of␣x.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣y:␣␣number
␣␣␣␣␣␣␣␣proximity␣to␣0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5.
␣␣␣␣"""
    return 4.0 + sum(map(lambda x: abs(x-0.5)**2, xs))

mntestinitial = [0.1, 0.2, 0.3, 0.4, 0.6, 0.7, 0.8, 0.9]
mntestlimits = normdims(8)


def mntest2(xs):
    """
␣␣␣␣A␣two␣dimensional␣test␣for␣the␣multiNewton␣function,
␣␣␣␣such␣that␣the␣progress␣of␣results␣can␣be␣plotted␣on␣a␣graph.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣xs:␣list␣or␣array
␣␣␣␣␣␣␣␣Value␣of␣x.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣y:␣␣number
␣␣␣␣␣␣␣␣proximity␣to␣0.5,0.5.
␣␣␣␣"""
    return ((xs[0]-0.5)**4+(xs[1]-0.5)**2)

mntest2limits = normdims(2)


def vis2d(f):
    """
␣␣␣␣Visualises␣a␣2d␣function␣with␣limits␣[(0,1),(0,1)].

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣f:␣␣function
␣␣␣␣␣␣␣␣Two-dimensional␣function␣to␣be␣visualised.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣None
␣␣␣␣"""
    import pylab
    xx, yy = pylab.meshgrid(pylab.linspace(0, 1, 51), pylab.linspace(0, 1, 51))
    zz = pylab.zeros([51, 51])
    for i in range(51):
        for j in range(51):
            zz[i, j] = f([xx[i, j], yy[i, j]])
    pylab.contourf(xx, yy, zz, 100)
    pylab.colorbar()
    pylab.show()
    pylab.clf()


def vis2d_slice(f, base, dims):
    """
␣␣␣␣Visualises␣a␣2d␣slice␣of␣function␣with␣limits␣[(0,1),*len(base)].
␣␣␣␣slicing␣across␣the␣dimensions␣indexed␣by␣the␣list␣dims,
␣␣␣␣keeping␣all␣other␣variables␣fixed␣at␣base␣values.
```

```
    Parameters
    ----------
    f:  function
        Function to be visualised.
    base:   list or array
        baseline or pivot point around which to take a two-dimensional slice.
    dims:   list of length 2
        The numerical value of the dimensions to be sliced.

    Returns
    -------
    None
    """
    import pylab
    xx, yy = pylab.meshgrid(pylab.linspace(0, 1, 51), pylab.linspace(0, 1, 51))
    zz = pylab.zeros([51, 51])
    for i in range(51):
        for j in range(51):
            x_eval = base[:]
            x_eval[dims[0]] = xx[i, j]
            x_eval[dims[1]] = yy[i, j]
            zz[i, j] = f(x_eval)
    pylab.contourf(xx, yy, zz, 100)
    pylab.colorbar()
    pylab.show()
    pylab.clf()


def save2d_slice(f, base, dims, xlabel, ylabel, title, filename):
    """
    Visualises a 2d slice of function with limits [(0,1),*len(base)].
    slicing across the dimensions indexed by the list dims,
    keeping all other variables fixed at base values.

    Parameters
    ----------
    f:  function
        Function to be visualised.
    base:   list or array
        baseline or pivot point around which to take a two-dimensional slice.
    dims:   list of length 2
        The numerical value of the dimensions to be sliced.
    xlabel: string
        Label for the x-axis.
    ylabel: string
        Label for the y-axis.
    title:  string
        Title for the plot.
    filename:   string
        Filename to save the plot as.

    Returns
    -------
    None
    """
    import pylab
    xx, yy = pylab.meshgrid(pylab.linspace(0, 1, 51), pylab.linspace(0, 1, 51))
    zz = pylab.zeros([51, 51])
    for i in range(51):
        for j in range(51):
            x_eval = base[:]
            x_eval[dims[0]] = xx[i, j]
            x_eval[dims[1]] = yy[i, j]
            zz[i, j] = f(x_eval)
    pylab.contourf(xx, yy, zz, 100)
    pylab.colorbar()
    pylab.xlabel(xlabel)
    pylab.ylabel(ylabel)
    pylab.title(title)
    pylab.savefig(filename)
    pylab.clf()


def comp2d(model, actual, n=51):
    """
    Compares a model of an actual 2d function and returns the average
    RMS error over n^2 points.

    Parameters
    ----------
    model:  function
        Function for the model.
    actual: function
        Actual function that is being modelled by model.
    n:  integer (default 51)
        Resolution over which to make the comparison.

    Returns
    -------
    error:  number
        Average RMS error.
    """
    xx, yy = np.meshgrid(np.linspace(0, 1, n), np.linspace(0, 1, n))
    zz = np.zeros([n, n])
    for i in range(n):
        for j in range(n):
            zz[i, j] = ((model([xx[i, j], yy[i, j]]) -
                         actual([xx[i, j], yy[i, j]]))**2)**0.5
    return sum(sum(zz))/float(n**2)


# wrapper functions
def exampleRDTwrapper(x):
    """
    A wrapper function to exampleRDT from ajopenfoam.
    For use with ga1.  Takes input x, list of 4 nums describing pitch.
    Returns the efficiency at J = 0.6.
    N.B. This code is deprecated, exampleRDT2wrapper is more accurate.

    Parameters
    ----------
    x:  list of four numbers
        Values of pitch at root, half, 0.7 and tip.

    Returns
    -------
    eta:    number
        Efficiency of RDT with given pitch at J = 0.6.
    """

    from ajopenfoam_simulations import exampleRDT

    # convert to tuple for dictionary use.
    tupledx = (x[0], x[1], x[2], x[3])

    # check if solved before.
    # deprecated: if _optdic.has_key(tupledx):
    if tupledx in _optdic:
        return _optdic[tupledx]

    # solve.
    KT, KQ, eta = exampleRDT(x[0], x[1], x[2], x[3], 0.6)

    # store in optdic for future recall.
    _optdic[tupledx] = eta

    return eta


def exampleRDT2wrapper(x):
    """
    A wrapper function to exampleRDT from ajopenfoam.
    For use with ga1.  Takes input x, list of 4 nums describing pitch.
    Returns the efficiency at J = 0.6.

    Parameters
    ----------
    x:  list of four numbers
        Values of pitch at root, half, 0.7 and tip.

    Returns
    -------
    eta:    number
        Efficiency of RDT with given pitch at J = 0.6.
    """

    from ajopenfoam_simulations import exampleRDT2
    import time

    # convert to tuple for dictionary use.
    tupledx = (x[0], x[1], x[2], x[3])

    # check if solved before.
    # deprecated: if _optdic.has_key(tupledx):
    if tupledx in _optdic:
        return _optdic[tupledx]

    # solve.
    starttime = time.time()
    KT, KQ, eta = \
        exampleRDT2(x[0], x[1], x[2], x[3], 0.6, runsilent=True)
    stoptime = time.time()

    # log
    logout = open("/home/ajd205/ajoptlog.csv", "a")
    logout.write(str(x[0]) + ", " + str(x[1]) + ", " + str(x[2]) + ", " +
                 str(x[3]) + ", " + str(KT) + ", " + str(KQ) + ", " +
                 str(eta) + "\n")
    logout.close()
    print("Solved in " + str(stoptime-starttime) + " seconds " + str(x) +
          ", KT = " + str(KT) + ", KQ = " + str(KQ) + ", eta = " + str(eta))

    # sanity filters
    # is K_T in a reasonable and useful range?
    if KT < 0:
        print("Solution␣discounted␣due␣to␣negative␣thrust!")
        eta = 0
    if KT > 1:
        print("Solution␣discounted␣due␣to␣unphysical␣thrust␣" +
              "(probably␣not␣converged)!")
        eta = 0
    # is K_Q in a reasonable and useful range?
    if KQ < 0 or KQ > 0.5:
        print("Solution␣discounted␣due␣to␣unphysical␣torque␣" +
              "(probably␣not␣converged)!")
        eta = 0

    # store in optdic for future recall.
    _optdic[tupledx] = eta

    return eta


def exampleRDT2wrapper2d(x):
    """
␣␣␣␣A␣wrapper␣function␣to␣exampleRDT␣from␣ajopenfoam.
␣␣␣␣For␣use␣with␣ga1.␣␣Takes␣input␣x,␣list␣of␣2␣nums␣describing␣pitch.
␣␣␣␣Returns␣the␣efficiency␣at␣J␣=␣0.6.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣x:␣␣list␣of␣two␣numbers
␣␣␣␣␣␣␣␣Values␣of␣pitch␣at␣root,␣tip.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣eta:␣␣␣␣number
␣␣␣␣␣␣␣␣Efficiency␣of␣RDT␣with␣given␣pitch␣at␣J␣=␣0.6.
␣␣␣␣"""

    from ajopenfoam_simulations import exampleRDT2
    import time

    # convert to tuple for dictionary use.
    tupledx = (x[0], x[1])

    # check if solved before.
    # deprecated: if _optdic.has_key(tupledx):
    if tupledx in _optdic:
        return _optdic[tupledx]

    # solve.
    starttime = time.time()
    KT, KQ, eta = \
        exampleRDT2(0.5+x[0], 0.5+0.5*(x[0]+x[1]),
                    0.5+x[0]+0.7*(x[1]-x[0]), 0.5+x[1], 0.6,
                    runsilent=True)
    stoptime = time.time()

    # log - not included as hopefully stored in a surrogate model
    #logout = open("/home/ajd205/ajoptlog.csv","a")
    #logout.write(str(x[0])+", "+str(x[1])+", "+str(x[2])+", "+str(x[3])+", "+
    #str(KT)+", "+str(KQ)+", "+str(eta)+"\n")
    #logout.close()
    print("Solved␣in␣" + str(stoptime-starttime) + "␣seconds␣" + str(x) +
          ",␣KT␣=␣" + str(KT) + ",␣KQ␣=␣" + str(KQ) + ",␣eta␣=␣" + str(eta))

    # sanity filters
    # is K_T in a reasonable and useful range?
    if KT < 0:
        print("Solution␣discounted␣due␣to␣negative␣thrust!")
        eta = 0
    if KT > 1:
        print("Solution␣discounted␣due␣to␣unphysical␣thrust␣" +
              "(probably␣not␣converged)!")
        eta = 0
    # is K_Q in a reasonable and useful range?
    if KQ < 0 or KQ > 0.5:
        print("Solution␣discounted␣due␣to␣unphysical␣torque␣" +
              "(probably␣not␣converged)!")
        eta = 0

    # store in optdic for future recall.
    _optdic[tupledx] = eta

    return eta

exampleRDTlimits = [(0.5, 1.5), (0.5, 1.5), (0.5, 1.5), (0.5, 1.5)]
```