PISTON CONTROL FOR AN INTEGRAL COMPRESSION WIND TURBINE

SIMON WOOLHEAD, MEng.

Thesis submitted to The University of Nottingham

for the degree of Doctor of Philosophy

JULY 2015

Abstract

This thesis concerns an analysis of an Integral Compressed Air Wind Turbine (ICWT), in which energy is extracted from a slow-moving renewable source through the use of compressed air. This concept is particularly applicable to large offshore wind turbines, and can be readily integrated with compressed air energy storage methods. The ICWT has a very large rotor with free pistons travelling within the rotor blades, inducting and compressing air to high pressures at each end of the stroke. The compressed air can be stored and expanded when the energy is required, solving the intermittency issue of wind energy. By gathering energy along the rotor blades, rather than at the hub, it also avoids the very high torques associated with extremely large turbines.

This thesis investigates optimal control strategies for ICWTs. Firstly, an initial model of the system using coupled ordinary differential equations (ODEs) is constructed to simulate a single piston pair of an ICWT system. This framework utilises several 'modes' which the system passes through in the course of each stroke, with movement between modes controlled by simple algorithms. Calculations of potential and required energy are developed to allow basic control of the valve timings.

The simulation is then extended to include thermal modelling of the walls of the compression tube, using orthonormal polynomials. A long-duration instance of the model is used to identify steady-state values for the orthonormal parameters, which demonstrates that the wall temperatures would remain within 15 K of the ambient

temperature.

One possible solution to the high temperatures caused by the near-adiabatic conditions of the compression is added to the model; namely, the injection of water droplets to the cylinder at the start of the compression stage. A method to efficiently simulate a phase transition in MATLAB is developed and implemented, allowing an analysis of the optimum mass balance of water to be injected to reduce the exhausted air temperature. An appendix examines several of the assumptions built into the model, in particular the rigidity of the components and variations in the rotational velocity of the rotor due to Coriolis and gravitational forces.

Two valve control schemes are developed and implemented into the model; firstly, a simple proportional and derivative controller, which acts according to a rule dictating a gradual reduction in the energy surplus. This option proves to be limited in the degree to which it can avoid wasting compressed air. A second scheme, involving a simple version of sliding-mode control with two controllers operating at different timescales, is instead developed and shown to be significantly more effective at improving the system efficiency.

Finally, an optimisation study is carried out on the 'kick' stage, in which stored compressed air is used to accelerate the piston before compression. A large dataset of simulations allows for the specification of a set of optimum parameters based on a balance between power extraction from the rotating frame and net power generation.

Acknowledgements

I would like to thank my supervisor, Prof. Seamus Garvey, for his guidance, expertise, and above all his patience. I would also like to thank my internal assessor, Dr. David Hann, for his vital feedback at my progression assessments. The assistance of everyone from the Structural Integrity and Dynamics Group at everything from LaTeX to Matlab has been invaluable, in particular the advice of Dr. Andrew Pimm, James Buck, Kai Wah Liew and Dr. Rupesh Patel and my associate supervisor, Prof. Atanas Popov.

I would also like to thank my parents and all my family and friends for their constant help and encouragement, which has been hugely appreciated. Finally, I owe the existence of this thesis to my truly incredible wife Leeanne, without whose boundless support throughout the last four years it would never have happened. Thank you.

Contents

List of Figures

Nomenclature

Acronyms and initialisms

EU	European Union
NREL	National Renewable Energy Laboratory, USA
FPEC	Free-Piston Energy Converter
CAES	Compressed Air Energy Storage
PID	Proportional, Integral, Derivative
VSCS	Variable Structure Control System
SMC	Sliding Mode Control
ODE	Ordinary Differential Equation
FSAL	First Same As Last
NDF	Numerical Differentiation Function
SSQ	Sum of Squares of Differences
SVD	Singular Value Decomposition
KERS	Kinetic Energy Recovery System
PHES	Pumped Hydroelectric Energy Storage
HVDC	High-Voltage Direct Current
DSM	Demand-Side Management
ICWT	Integral Compressed Air Wind Turbine
TDC	Top Dead Centre

BDC	Bottom Dead Centre
GPE	Gravitational Potential Energy
CPE	Centrifugal Potential Energy
LP, HP	Low-pressure and high-pressure
CoM	Centre of Mass
GSONP	Gram-Schmidt Orthonormalisation Process

Symbols and mathematical notation

$\mathbf{y}$	State vector
t	Time
θ	Angular position of compression tube, in radians
h	Position of piston along compression tube
m	Mass
m_P	Piston mass
T	Temperature
L	Length
L_{B}	Length of blade
L_{PE}	Half length of piston
L_{TE}	Distance from rotor axis to end of compression tube
t	Thickness
k	Valve constants
p	Pressure
r	Pressure ratio
p_{targ}	Target pressure
A	Area
F	Force
F_μ	Friction force
P	Polynomial

c_p	Specific heat capacity at constant pressure
c_v	Specific heat capacity at constant volume
$L_{f\to g}$	Specific heat of vaporisation of water
E	Energy
E_{pot}	Potential energy
E_{req}	Required energy
E_{WD}	Work done
$\dot{E}$	Power
ψ	Control parameter
v	Wind speed
c_{PR}	Rotor power coefficient
λ	Tip-speed ratio
k_{th}	Thermal conductivity

Common subscripts and modifiers

0	Instantaneous value of the variable in question
i	The current element
a	Airmass
wf	Liquid water
wg	Steam
m	Mixture of air, steam and water
comp	The compression stage
exh	The exhaust stage
lock	The locked stage
kick	The 'kick' stage
atm	Atmospheric
CT	The compression tube
PE	Piston end

TE	Compression tube end
1, 2	Identifiers referring to the respective compression chambers (see Figure 3.5)
α, β	Identifiers referring to opposed pistons (see Figure 3.5)

Dot notation, as in $\dot{h}$ and $\ddot{\theta}$, is used to indicate derivatives with respect to time. Matrices and vectors are typeset in bold, as $\mathbf{A}$ and $\mathbf{y}$.

Chapter 1

Introduction

Global climate change is rapidly becoming the largest single issue facing world civilisation in the 21st century, with extremely widespread impacts including agricultural production crashes, increased flooding, loss of water security, desertification, wildfires, and diseases. Anthropogenic greenhouse gas emissions are now considered to be responsible for most of the global average temperature increase over the last 50 years, and the effects of continued emissions are very likely to be more significant than those in the second half of the 20th century. [1]

To stave off the worst-case scenarios, limiting the global temperature rise to 2 K is needed, but this requires for a reduction in carbon dioxide emissions of up to 85% from 2000 to 2050. The ongoing use of fossil fuels is a major part of the issue, contributing 57% of all anthropogenic greenhouse gas emissions in 2004. This is therefore driving an enormous degree of research and investment into ways to reduce these emissions, particularly renewable energy sources. [2]

The European Union (EU) has set targets of both reducing greenhouse gas emissions by 20% and increasing the proportion of energy consumption from renewable sources to 20% by 2020. The latter target is likely to be extremely challenging for the UK,

and given the immaturity of other technologies, will require an enormous investment in wind energy.[3]

5% of the UK's electricity was generated from wind in 2013[4]. Onshore wind in the UK is showing clear signs of reaching saturation, but offshore wind remains relatively immature. Investment has so far focussed on installing conventional turbines offshore in shallow water, with more installations expected in the future[5]. However, wind energy is expensive, and additionally the system costs associated with its intermittency and dispatchability problems promise to add more than 5% to the operating cost of the entire electrical system in the UK[6]. In total, meeting the UK's emissions requirements is likely to cost €2.5 billion per year[7].

This thesis is an exploration of an alternative design of wind turbine, which is intended to avoid some of the system costs of wind energy. To fully explain this concept, we must first consider the science underlying the design of wind turbines.

1.1 Wind rotor theory

The first publication of wind rotor theory was in 1920 by Prof. Albert Betz[8]. By considering the reduction in linear wind velocity before and after the rotor in terms of both kinetic energy and momentum, Betz formulated an expression for the rotor power coefficient, c_{PR}, which is the fraction of the available kinetic energy extracted by the rotor:

$$c_{\mathrm{PR}} = \frac{1}{2}\left(1 - \left(\frac{v_2}{v_1}\right)^2\right)\left(1 + \frac{v_2}{v_1}\right) \tag{1.1}$$

where v_1 is the wind speed upstream of the turbine and v_2 is the downstream speed. By differentiating and then solving this expression, we obtain the Lanchester-Betz limit of $c_{\mathrm{PR}} = \frac{16}{27}$, when the air leaving the turbine has $\frac{1}{3}$ of its initial velocity[9, p.43] [10, p.83]. This maximum is shown in Figure 1.1.

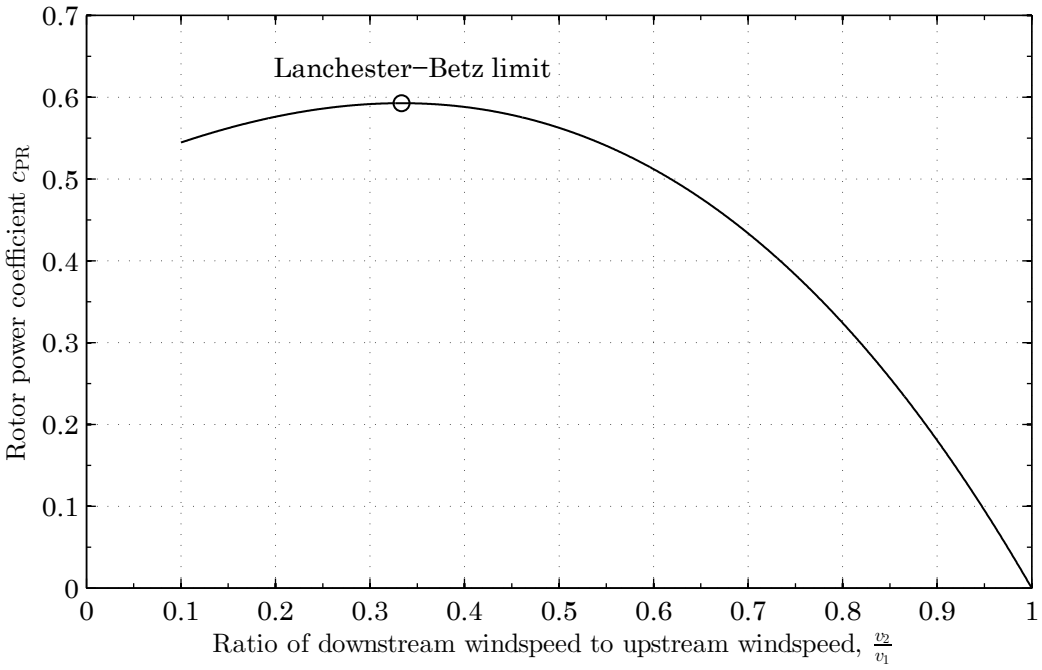


Figure 1.1: The Lanchester-Betz limit to rotor power coefficient

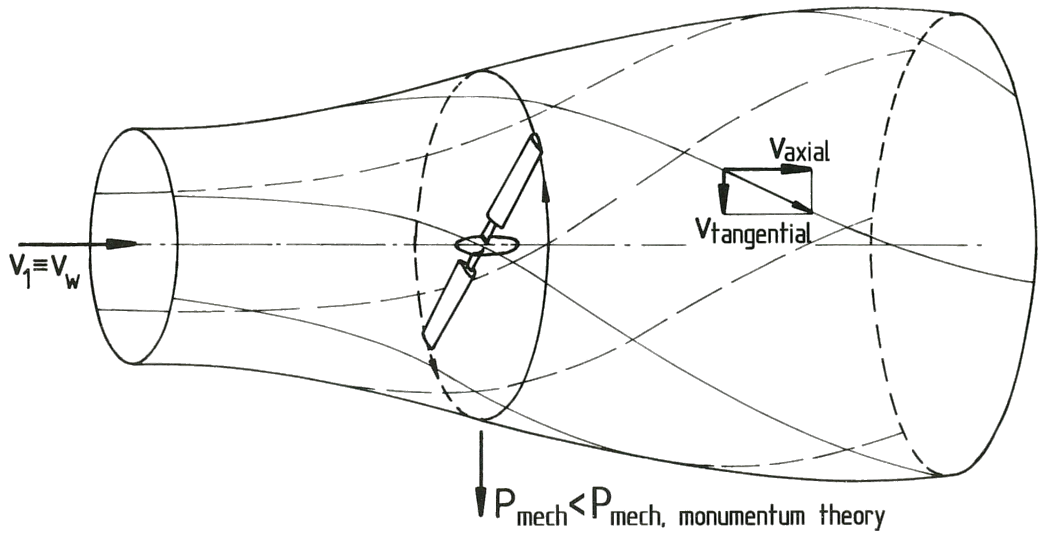


Figure 1.2: Glauert's extended momentum theory with rotating rotor wake, taken from Hau and Von Renouard [10, p.91]

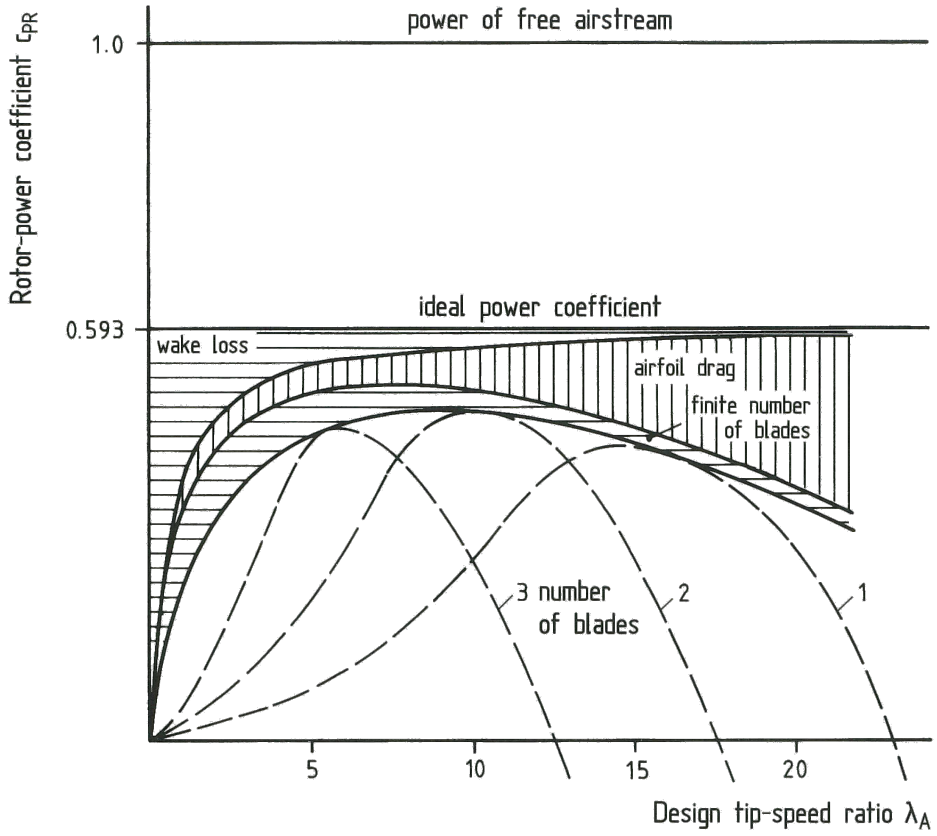


Figure 1.3: Rotor power coefficient and tip-speed ratio with varying blade count, taken from Hau and Von Renouard [10, p.95]

Later work, in particular by Glauert, developed 'wing theory', which showed that the actual power coefficient of a single-rotor turbine is limited further by the rotational momentum imparted to the airflow by the rotor, known as 'swirl', shown in Figure 1.2. This is characterised using the tip-speed ratio, λ, which is the ratio of the tangential speed of the blade tip to the wind speed:

$$\lambda = \frac{\dot{\theta} L_{\mathrm{B}}}{v_{\mathrm{W}}} \tag{1.2}$$

where L_{B} is blade length, v_{W} is wind speed, and $\dot{\theta}$ is the rotational speed of the rotor.

Rotors with differing numbers of blades have different optimum tip-speed ratios,

as shown in Figure 1.3. For the standard 'Danish' 3-bladed upwind design, a λ of around 7 is a good balance between minimum swirl and maximum energy extraction.[10, p.91–105]

1.2 Large-diameter wind turbines

Since the optimum tip-speed ratio for a turbine design is independent of rotor diameter, it means that larger wind turbines in the same wind speeds must turn more slowly. For a 2.5 MW wind turbine, $\dot{\theta} = 2.6$ rad/s[10, p.351], which is difficult to reconcile with the very high rotational speeds of around 260 rad/s required by most power station electrical generators[10, p.306]. There are broadly two approaches to dealing with this problem, which is a focus of much current research.

1.2.1 Gearboxes

Most large turbines installed today use gearboxes with ratios from 1:50 to 1:100, combined with relatively conventional generators[10, p.306]. Due to the large ratios required, wind turbine gearboxes generally have multiple stages, which are some mix of planetary and parallel shaft stages. Planetary stages are lighter, have lower losses, and are capable of ratios of up to 1:12 per stage, but their higher costs mean they are less common in turbines below 500 kW[10, 350–356].

This complexity increases cost; the gearbox represents around 10% of the total capital cost of an offshore turbine, compared to 8% for the entire turbine electrical system and 7% for the rotor blades[11, p.9]. Additionally, the tough operating conditions and difficulty of maintenance and replacement resulted in gearboxes causing the most downtime per failure of any turbine component, according to two separate studies of offshore turbines[9, p.663].

1.2.2 Direct-drive machines

An alternative to gearboxes is the use of low-speed generators coupled directly to the rotor. Synchronous machines are used, combined with frequency modulation equipment to harmonise the output frequency for the grid [9, p.367]. The resulting drivetrain is mechanically simpler, although more electrically complex.

However, the generators required are larger diameter, heavier, and require more cooling than those used with gearboxes. The use of either hand-winding on the large number of poles in electrically excited generators, or very high strength permanent magnets, also increases manufacturing complexity. [10, p.418–420]

1.3 Scaling effects

The problem of tip-speed ratio with large wind turbines may be generalised by considering scaling laws.

Since tip-speed ratio is constant, the rotational speed $\dot{\theta}$ at rated conditions is inversely proportional to turbine diameter D.

Rotor power is given by [10, p.102]:

$$\dot{E} = c_{\mathrm{PR}} \frac{\rho_a}{2} v_{\mathrm{W}}^3 \pi \left(\frac{D}{2} \right)^2 \tag{1.3}$$

where c_{PR} is rotor power coefficient, ρ_a is the air density, and v_{W} is the rated wind speed. Since all the parameters except blade length are constant with increasing turbine size, the power output (and thus the costs of the generator and other electrical systems) is proportional to D^2 [9, p.327]. Other component costs which are proportional to D^2 might include surface treatments and covers.

Dividing power by rotor speed, we find that shaft torque increases in proportion

to D^3. As gearbox costs are approximately proportional to the input torque, we take gearbox cost as proportional to D^3. Mass and volume also rise with D^3, as do several costs which depend on those factors such as tower, bearings, and foundation cost [9, p.327–329].

A very detailed cost model was built for the US National Renewable Energy Laboratory in 2006 by Fingersh et al. [12]. The NREL model was based on 7 WindPACT US Department of Energy studies into various turbine subsystems, providing very accurate data on scaling proportions. It found costs scaled in proportion to $D^{1.8446}$ for the generator, D^2 for electrical connections, $D^{2.498}$ for gearbox, $D^{1.2}$ for the foundations, and D^3 for the blade material and $D^{2.5}$ for the labour involved in the blade manufacture [12].

To generalise, we can model the total cost of the turbine as a polynomial in D with positive coefficients. By dividing through by D^2, we obtain a function for cost per unit energy [9, p.328], [13]. One such function, normalised to 100 at $D = 60$ m, was calculated by Burton et al. [9, p.327] and is shown in Figure 1.4 in black. The minimum cost per unit energy was found to be located at $D = 43.6$ m; the NREL model found that the minimum cost per unit energy is at a rotor diameter slightly larger than 70 m [9, p.331]. Both of these cost models are for onshore turbines, which are substantially smaller than the offshore turbines currently under investigation, but no cost model has yet been formulated for offshore machines.

1.4 The integral compression wind turbine

A 2010 concept by Garvey [14] called the Integral Compression Wind Turbine (ICWT) aims to define a turbine with an alternative cost curve, with a minimum cost per unit energy which is both lower than that for conventional designs, and also located at a significantly larger diameter. A purely hypothetical cost curve for such a design

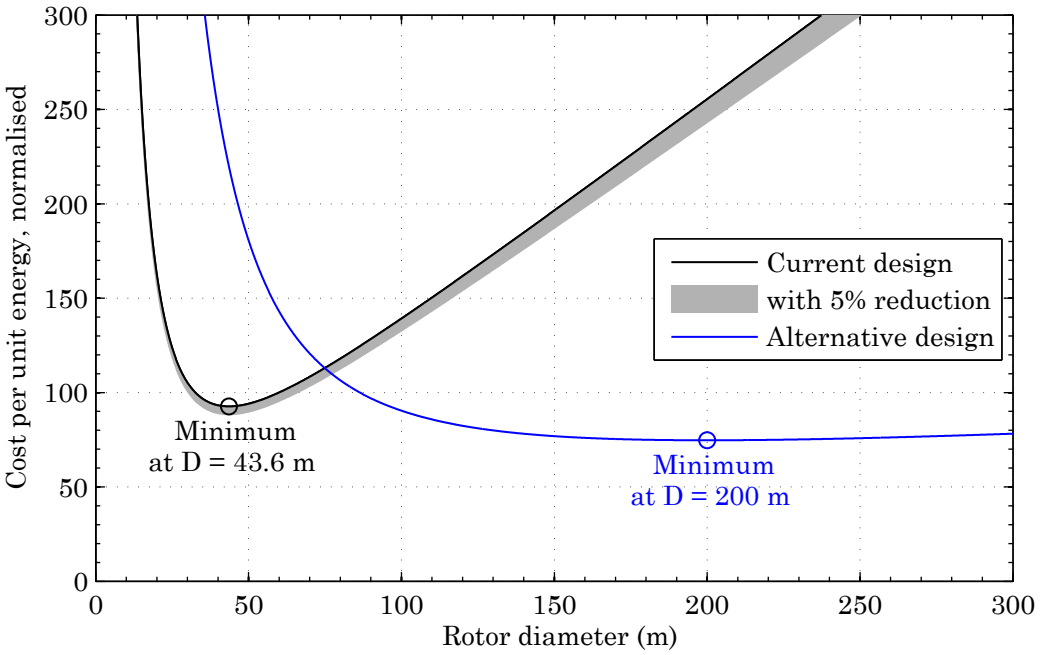


Figure 1.4: Simplified turbine cost model from Burton et al. [9, p.327], with hypothetical alternative design

is shown in Figure 1.4 in blue.

The ICWT concept is designed specifically to avoid the use of low-speed shafts and very high torques and bending moments which characterise existing very large turbines. Instead, it aims to extract the energy directly at the blades themselves, without transmitting the power through a shaft [15, p.127].

The ICWT uses a very large rotor with hollow compression tubes within the blades, shown in Figure 1.5. Pistons travelling freely under the action of gravity fall down the tubes, with a carefully-controlled valve system using their kinetic and gravitational potential energy to intake, compress and exhaust air into a high-pressure manifold. The compressed air is then piped down to sea level, where it is either sent straight to high efficiency expansion turbines to extract the energy or piped to storage balloons on the seabed. The rotor is held up on a tetrahedral frame with cross-bracing, anchored so that it yaws freely with changes in wind direction. [14]

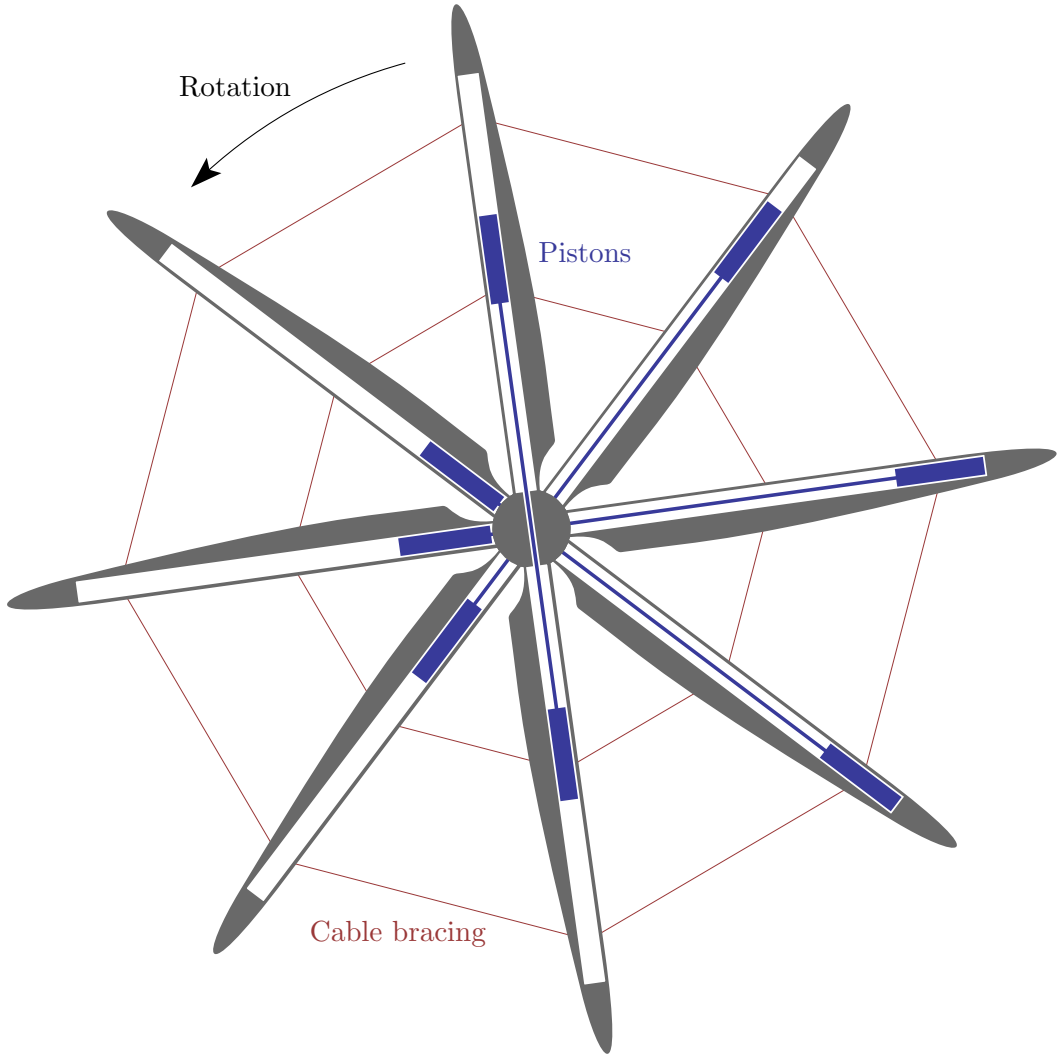


Figure 1.5: Overall view of the ICWT rotor

A second key advantage of the ICWT concept is its compatibility for energy storage. As outlined in section 2.2, renewable power has a significant problem with intermittency due to the uncontrollable nature of the various energy sources involved. Compressed air energy storage (CAES) is attracting a great deal of attention for its suitability in dealing with variable power plant, in particular wind [16]. By converting renewable energy directly to a suitable storage medium, the ICWT design is inherently more controllable than conventional wind turbines.

1.4.1 The ICWT stroke

Between strokes, the pistons will reside at the end of the compression tubes, 'locked' there by the centrifugal force. Valves to the atmosphere are held open to allow the free movement of the piston. When the blade reaches a preset angle, the piston is pushed away from the end of the tube using compressed air, driving it against centrifugal force; this is referred to as the 'kick'.

Once the piston is moving at a preset speed, the valve delivering the compressed air is closed, and the high pressure air behind the piston expands, continuing to accelerate it down from the end of the tube. Once the pressure behind the piston reaches atmospheric pressure, valves are opened to allow the free movement of air into the tube above the piston as it falls, described as the 'freefall' stage.

Next, the tube below the piston is sealed, and the air begins to compress as the piston descends. Once it reaches the target pressure, a valve allows it to exhaust from the tube. Finally, the piston comes to a rest at the end of its stroke. Any excess compressed air which was not exhausted is dumped to the atmosphere.

For every full rotation of the ICWT rotor, the piston in each blade will make two strokes: one towards the hub of the rotor, and one towards the blade tip. Both strokes are shown in Figure 1.7, along with the labelled stages of the stroke.

The piston velocity follows the profile shown in Figure 1.8, which is plotted against time; it can be seen that the whole stroke takes around 6 seconds. In terms of position along the tube, Figure 1.9 shows that the majority of the piston's travel is in freefall, with compression starting at a position after the piston has travelled approximately 83% of the tube's length.

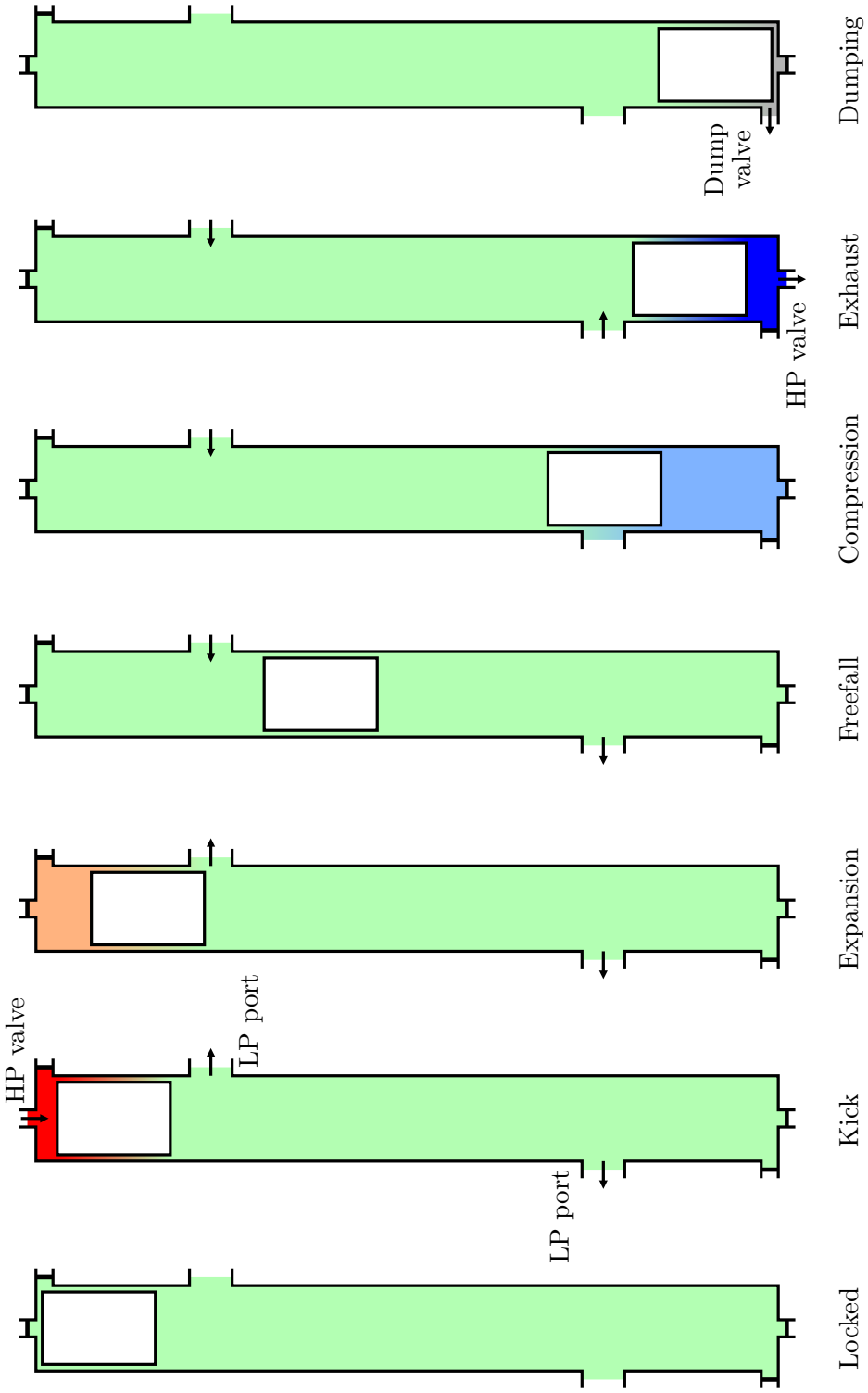


Figure 1.6: The seven stages of the stroke, with valve operation

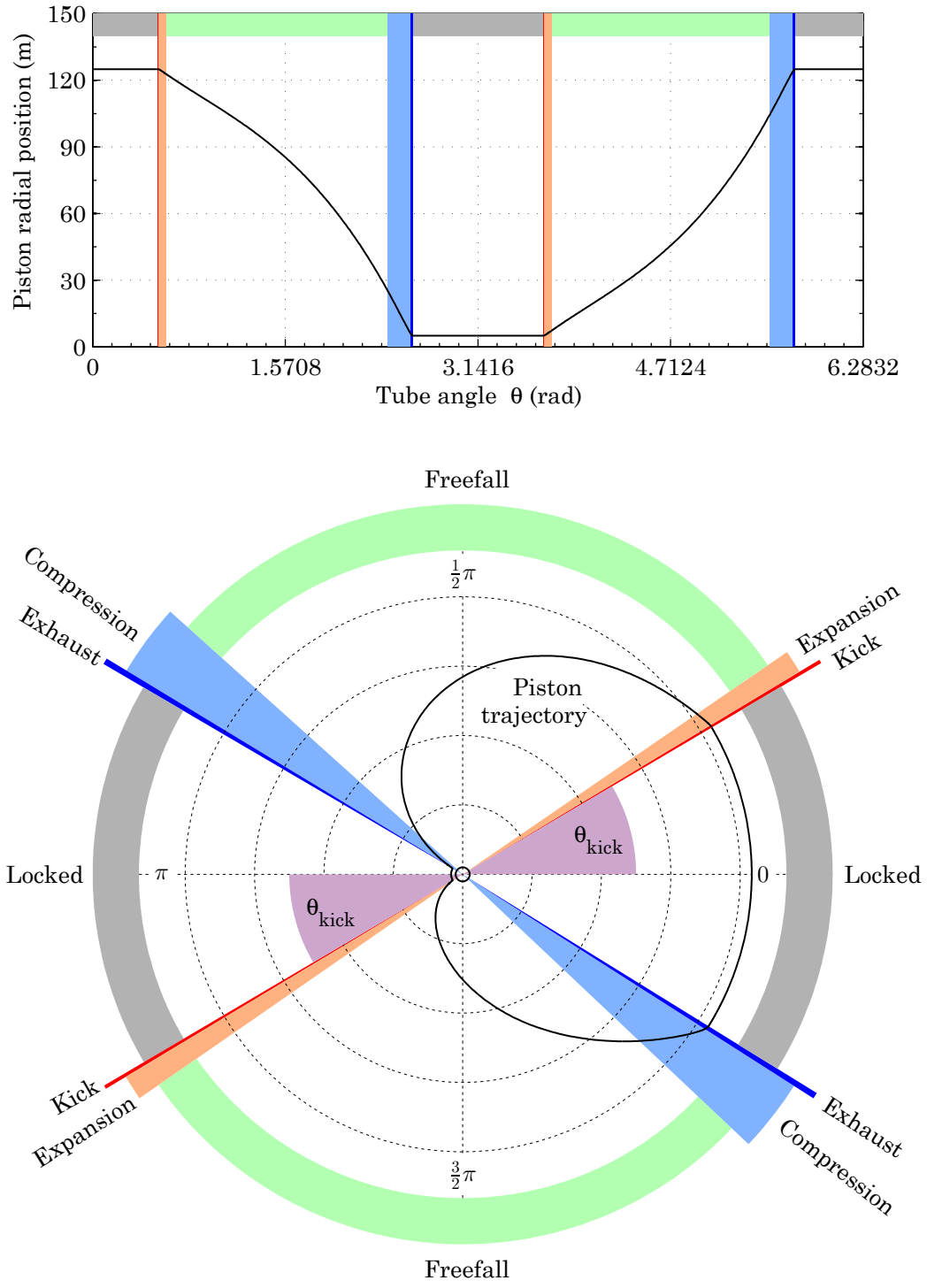


Figure 1.7: Angular diagram showing stages

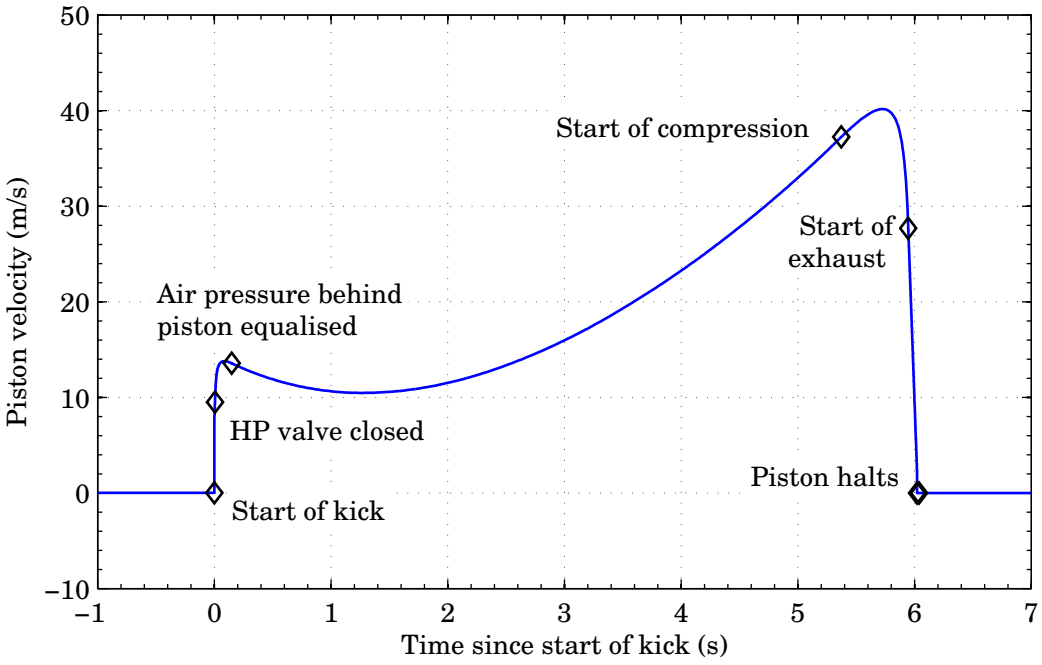


Figure 1.8: Velocity profile of the piston with stages

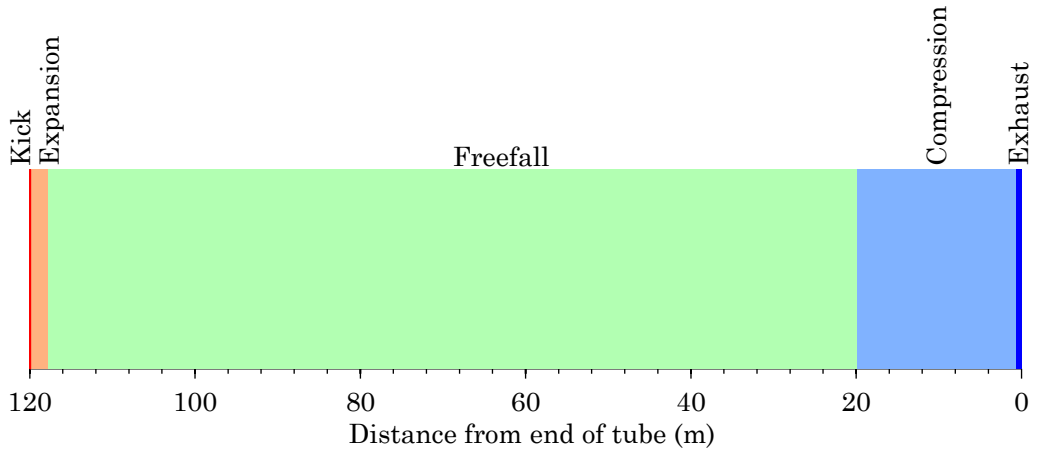


Figure 1.9: Stroke stage plotted against piston position

1.4.2 Conceptual design

The design calls for a very large, multi-bladed rotor, to reduce the rotational speed and thus the centrifugal force seen by the pistons (which must be overcome by the kick in every stroke). By linking opposing pistons, as shown in Figure 1.5, the effective radial positioning of the centre of mass of a piston pair is halved.

To seal around the pistons in travel, the conceptual design calls for labyrinth seals with injected water. This water may be expressed from small orifices in the tube wall and scooped into the piston, and then injected between the labyrinth seal rings during compression.

Each blade contains a long, cylindrical compression tube, running from close to the hub out to near the blade tip. At each end of the tube, there are 3 orifices for air to flow through, as shown in Figure 1.6.

A low-pressure (LP) port, which allows the free movement of air to and from the atmosphere during the freefall stage of the stroke (to keep the air pressure equal to atmospheric pressure on each side of the piston), is required to allow for large flow rates at low pressure differences. Since this is only required at points remote from the end of the tube, however, it can occupy a large space in the compression tube wall. Depending on the operating requirements of a fully-detailed design, it may even not require the capacity to be closed, existing as a permanent hole through which atmospheric air can be inducted. This is shown in Figure 1.6, in which the LP ports are located remote from the ends and not closed during the stroke.

Each end also requires a high-pressure (HP) valve, which allows the exhaust of the HP air after compression, into the HP manifold for conduction away to storage. This valve must actuate at very high speed, so a final design would utilise many small check valves for this purpose, with electronic actuation applied to the check springs. Additionally, this valve is opened to provide the kick, using air from the HP manifold.

A third valve is a simple dump valve, which is used to remove any high-pressure air which remains in the tube after the piston has come to a halt.

The high-pressure air is carried from the valves at the tube ends to a manifold in the rotor hub, and from there to turbines and storage facilities shared across the wind farm.

1.5 Aims and scope

The aim of this project is to construct a detailed simulation of an Integral Compression Wind Turbine, and to propose and investigate valve control approaches.

The objectives are:

1. Create a coupled mechanical, thermal and airflow simulation environment in which behaviour can be fully predicted
2. Investigate the possibility of water-cooling to reduce air temperatures
3. Design and implement a control structure to ensure the efficient operating of the ICWT system

1.6 Layout of the thesis

This thesis comprises eight chapters plus this introduction.

Literature review

The second chapter contains a short overview of the various fields this project spans, including wind turbine technology, energy storage, free-piston energy converters, state-space simulations, and basic control techniques.

System modelling

The third chapter covers the derivation of the basic specification of the ICWT which will be modelled in this thesis, along with the derivation of the core model. It also addresses some of the implementation details of the program code.

Energy calculations

The fourth chapter details the algorithms used to calculate the various energies of the system, divided into the potential energy available and the energy required to compress the air in the cylinder. A method for the prediction of the angle of the compression tube at the end of the stroke is also presented.

Thermal modelling

The fifth chapter is concerned with accurate thermal modelling of the compression tube, and begins by laying out the differential equations which govern the wall temperatures. It then goes on to describe a technique for generating and using orthogonal polynomials with the Gram-Schmidt method, which provide a useful way to model the wall temperature profile. Finally, it addresses the identification of the eventual steady-state temperature profile of the wall through a Newton-Raphson method.

Water cooling

The sixth chapter is about the possibility of injecting water droplets into the cylinder before the compression stage to reduce the exhaust temperature of the air. It outlines a system of ODEs and events to model a mix of air, water and steam in the compression chamber, and describes the effect of implementing it on the model.

Exhaust valve control

The seventh chapter outlines some different control strategies for the exhaust valves in the system, from a relatively simple method involving the surplus energy, to a two-level nested controller system incorporating elements of model predictive control.

System optimisation

The eighth chapter describes the optimisation of the 'kick' stage. Two competing objectives are described, and a large number of simulations are run to investigate the state-space available through control of the kick parameters. A preferred operating surface is identified, and an inter-cycle control technique involving it is outlined.

Conclusions and future work

The ninth chapter reflects on the results found over the course of the work, and describes fruitful future research which may be investigated around ICWTs and the model laid out in this thesis.

Chapter 2

Literature review

In order to provide a suitable overview of the background literature relating to this project, this chapter will cover several disparate topics. We begin by considering the history and development of wind turbines, reviewing the issues surrounding dispatchability and intermittency of wind energy, before moving on to the various forms of energy storage and their respective attributes. Free-piston energy converters, which have some similarity to the ICWT concept, are then discussed, followed by basic valve technology and state-space simulations. Finally, a section considers some control techniques which will be useful concepts in the work.

2.1 Wind turbine technology

Wind power is one of the most significant contributors of renewable electricity to the world's power grids, and is generally considered a mature technology.

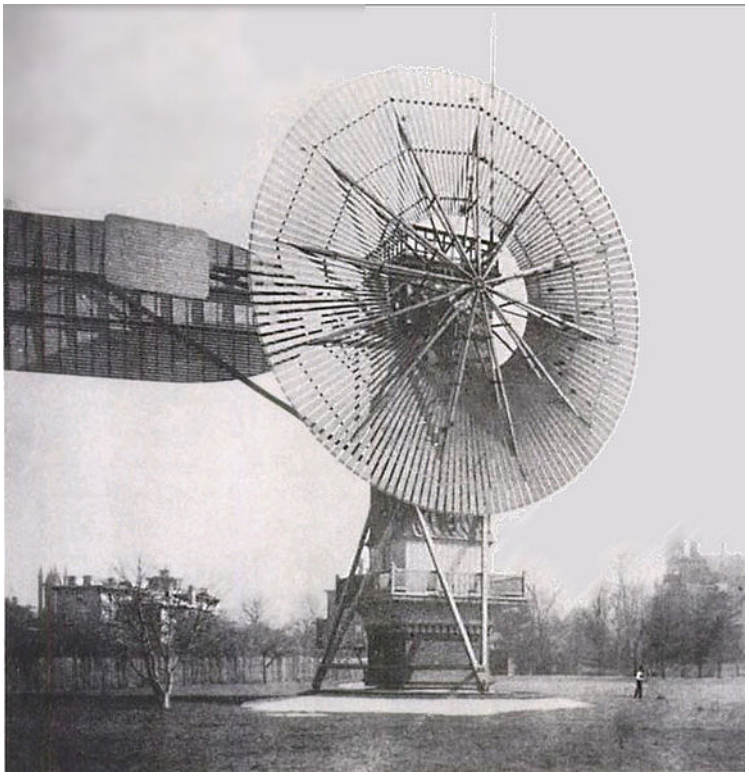


Figure 2.1: The 1888 wind turbine of Charles Brush, taken from Wikimedia Commons[17]

2.1.1 Development to date

In the 1890s, electricity remained a niche pursuit; generators were typically linked directly to the system load, making the intermittency of wind a significant problem right at the outset. Accordingly, in the first few years of their existence, wind turbine systems were closely linked to energy storage.

The first wind-powered electrical generator was constructed by Charles Brush in Ohio in 1888. It owed much to the flour-grinding and water-pumping windmills of the time, with a 144-bladed wheel and a rotating tower structure, and generated just 12 kW of DC power, used to charge batteries for lighting. It used two transmission belts to increase shaft speed by a factor of 50:1, in the process highlighting the first hurdle for turbines to overcome: the conventional multiblade rotors used for raising

water were just too slow for effective electrical generation. [18, p.36]

In 1891, the pioneering Danish researcher Prof. Poul laCour independently constructed a four-bladed turbine, using the dynamo to electrolyse water to produce hydrogen gas, which was then burned for light. It used a four-bladed shuttered rotor to increase speeds and reduce maintenance. His subsequent designs built on this foundation, and by 1908 72 turbines had been constructed, some reaching 35 kW in power. [10, p.24]

Two factors combined to help wind generation in the first quarter of the 20th century. Firstly, the design of smaller rotors was boosted by the interest in development of propellers for fixed-wing aircraft; and secondly, the rise in fuel prices caused by the First World War brought about a key opportunity for renewable energy. Small 2.5 kW generators for the charging of batteries became common in America in the 1930s, and larger 50 kW 'Aeromotor' turbines were produced in Europe by F.L. Smidth. Already, these turbines had begun to conform to what is now called the 'Danish' model of turbine; a two- or three-bladed horizontal-axis rotor, upwind of a gearbox and generator set, all mounted on the top of a narrow tower. [18, p.37–39], [10, p.25–26]

Despite the achievement of the first megawatt-scale turbine, constructed in 1941, development was slow up until the 1973 Suez oil crisis. A sudden rise in the cost of oil, combined with early warnings about an eventual depletion of fossil fuels, spurred a great deal of interest in renewable power, and wind power research programmes were started or accelerated in the USA, Sweden, Denmark and Germany. Turbines both small and large, from 55 kW up to 500 kW, were put into production, and the 1980s saw several experimental turbines in the megawatt scale - in particular the Growian 3 MW, in 1983. Promoted as a key vision of the future of wind energy, the Growian turbine's crippling technical problems warned off development of similarly large machines. Vertical-axis machines hit a similar roadblock in the shape of the 1985 Éole 4 MW project, which was dismantled and the Canadian programme which created it terminated shortly afterwards. [10, p.43–54]

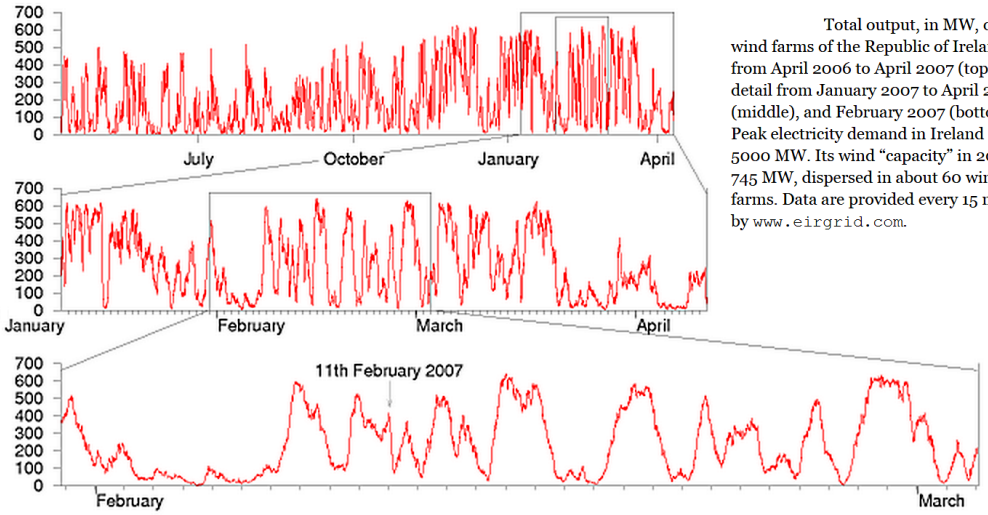


Figure 2.2: Wind farm intermittency, taken from MacKay [20, p.187]

2.1.2 Deployment

From the 1980s onwards, wind power has focussed on wind farms, with tens of turbines onshore or offshore to provide a significant contribution to the power grid; a worldwide total of 94 GW was operating by 2007. Concerns about land costs and public perception have led to the development of offshore wind farms, which are rapidly increasing in installed capacity; the EWEA estimates that 40 GW will be installed in Europe alone by 2030.[19] The UK had 11.2 GW of rated wind capacity at the end of 2013, equivalent to 4.8 GW after discounting for intermittency - 5.7% of the UK's total generating capacity[4, p.29].

2.2 Intermittency and dispatchability problems of renewable energy

Conventional electrical grids have minimal storage capacity; in 2008, the world had 125 GW of electrical energy storage, compared to a generation capacity of 3.9 TW[21]. Electrical grids therefore are closely controlled to keep supply and demand as close

as possible at all times, creating plans which account for not only plant capacity and downtime but also weather forecasting and TV schedules[20, p.196]. In this context, the operating characteristics of the grid's generating capacity are critically important.

Renewable energy sources, in particular wind and wave power, suffer from a significant drawback in this context: the quantity of renewable energy available for harvesting at any one time limits the power which can be generated. In the case of wind energy, this limit intrudes significantly into the normal operation of a wind farm. The capacity factor of a wind turbine is given by:

$$\text{capacity factor} = \frac{\text{Mean power output}}{\text{Rated power}} \tag{2.1}$$

Modern turbines are sufficiently low-maintenance to attain 98% technical availability[10, p.586] (i.e. they are offline due to maintenance or other technical reasons for 2% or less of the time), but wind speeds which are too high or too low keeps the capacity factor of most onshore wind farms around 30%[20, p.33][10, p.587]. Periods of low output, when a wind farm operates at 10% or less of its rated power, may last for as long as several days[20, p.187]. In addition, these lulls can be relatively sudden; changes of 20% of rated capacity in 30 minutes are common[21], and one operator saw a sudden reduction of 87% of power in 6 hours[10, p.664].

A second problem is due to the difficulty of predicting wind conditions, over periods from 10 minutes to 7 days. Although this was a significant historic concern, current techniques combining artificial neural networks with real-time telemetry and complex models are capable of predicting power output 24 hours in advance with errors around 15% of the wind farm's rated capacity. Additionally, projections based on total energy produced rather than hour-by-hour power output tend to be more accurate, since the timing predictions are particularly difficult. This represents an advantage for wind farms with integral or associated energy storage systems, since they are capable of using their storage capacity to smooth out the discrepancies and meet

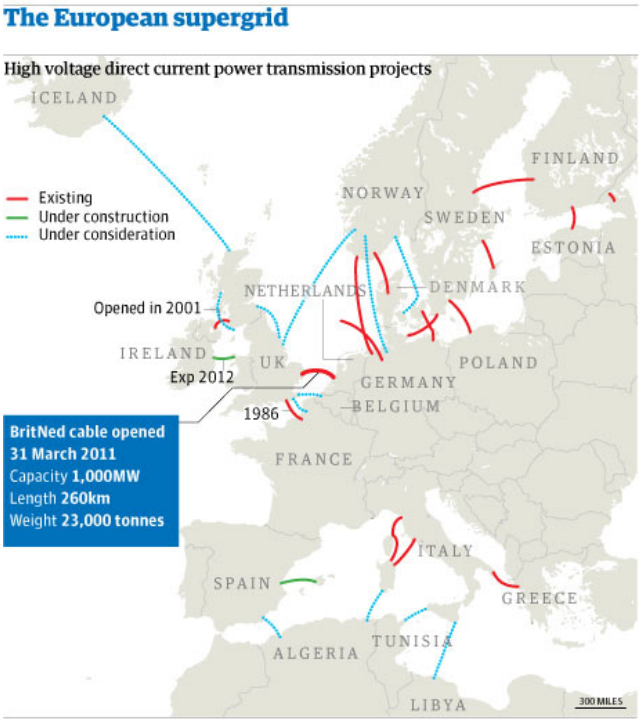


Figure 2.3: HVDC interconnectors in Europe in 2011, taken from the Guardian [24]

contracted power schedules more accurately. [22]

Taken together, this intermittency issue restricts large-scale penetration of wind energy. The Intergovernmental Panel on Climate Change (IPCC) to recommend that power grids contain only 15-20% of renewable capacity to avoid these problems [23, p.41]. Other sources recommend limits of 10% [22], with significant investment in stabilising measures required if that limit is exceeded.

2.2.1 Possible solutions

Interconnectors

The intermittency effect of renewables is reduced with increasing grid size, both geographically and electrically; a network with widely separated wind farms is less likely to see a simultaneous reduction in capacity at several, and reducing the fraction

of the overall grid generation capacity which is represented by any particular wind farm also aids in counteracting variation. Increased high-voltage direct current (HVDC) interconnection between electrical grids can allow for up to an extra 7% of wind penetration, up to 28% without negative effects, by increasing the size of the grid relative to the possible renewable fluctuations [25].

The UK National Grid intends to expand the UK's 4 GW interconnector capacity to 8–9 GW by 2020 [26]. However, this is at best a short-term solution, as interconnected grids will still have strict limits on overall variability, and large weather systems (covering whole regions) are not uncommon.

Load-following plant

At present, most of the variability is managed on the supply-side, by conventional fossil-fuel plant capable of operating in load-following mode (in which its output is scaled according to the current grid requirements) [20, p.186], [6, p.33]. Current reserve capacity in the UK is around 20%, mainly used for dealing with demand peaks [27]; a reserve of up to 10% of installed wind capacity is additionally required [22].

Nuclear power plant, which represented 11.7% of UK generating capacity in 2013 [4, p.29], is less suitable. Although Generation III and III+ reactors are technically capable of load-following, their production costs are primarily due to operating and capital expense rather than fuel, so running at reduced load does not significantly reduce costs. Combined with mechanical fatigue effects, this makes nuclear load-following economically infeasible for the foreseeable future [28]. Therefore, as fossil-fuel plant are decommissioned over the next 5–35 years and replaced by nuclear and renewables [29, p.10], this capability will be eroded.

Demand-side management

Demand-side management (DSM) is another technique. First developed in the 1970s as a response to energy security concerns following the 1973 oil crisis, DSM has attracted significantly more interest in recent years as part of wider 'smart grid' proposals[27]. Methods range from simply paying significant consumers for reductions in energy use, to remote deactivation of consumer appliances such as freezers and heaters during peak times[20, p.196]. However, current cost estimates indicate that additional reserve fossil-fuel generating capacity is less expensive than the roll-out of DSM technology[30].

2.3 Energy storage

Energy storage is thus a critical part of future renewables technology, and is the focus of a great deal of research and development. It is even a legal requirement in some areas; California Assembly Bill 2514 in 2010 requires utilities to include storage of at least 5% of daytime peak power usage by 2020[31].

There are several different methods of energy storage being used at grid-scale today, each with different characteristics and different applications. This section will outline the basic properties of the four most prominent technologies, with a particular view to their suitability for use with wind energy systems.

2.3.1 Flywheels

Flywheels have been used as energy storage devices, particularly for load smoothing, for tens of thousands of years. Original examples in spindles, spinning wheels and wood-turning lathes were used to smooth out variable power supplies, normally harmonically-varying torque from a foot-pedal. Early steam engines used flywheels to

convert from linear oscillations of steam pistons to constant-velocity rotational movement. In the 1970s, flywheels entered the power generation domain as uninterruptible power supplies for critical infrastructure, for which they are now commonplace[32]. Recent developments have seen flywheels used for short-term energy storage in Kinetic Energy Recovery Systems (KERS) in Formula 1 cars[33].

These applications are natural consequences of the key properties of flywheels for energy storage: they offer high short-term storage efficiency, large charge and discharge rates, mechanical simplicity, and no reduction in capacity with time. Modern flywheels, utilising magnetic bearings and partial- or absolute-vacuum chambers to reduce energy losses due to friction, can last for over 20 years[34] and reach turnaround efficiencies of over 85% for storage periods of seconds; however, this reduces to 45% after 24 hours[35], making them unsuitable for the long-term storage needed for wind. They are also expensive, with costs in the region of 300–600 €/kW·h[16].

2.3.2 Electrochemical storage

Batteries are an extremely common method of energy storage at very small scales due to their high energy densities, which range from 20 W·h/kg for lead-acid up to 180 W·h/kg for lithium-ion[16]. However, they have several drawbacks for use at grid-scale. Lead-acid batteries, although the cheapest per unit of energy at around 185 €/kW·h[16], last only 500–1000 cycles while requiring frequent maintenance, ventilation and temperature control[21]. The longest-lasting are nickel-cadmium batteries with 3500 cycles[16], but these exhibit a 'memory effect' in which partial cycling reduces their capacity over time - particularly unsuitable for renewables[21]. Additionally, their use of toxic heavy metals raises serious environmental issues, even aside from the limited quantities available for low-cost extraction[36].

Flow batteries

Flow batteries are a relatively new concept, comprising separate tanks of electrolytic solution which are passed through an electrochemical cell with an ion-selective membrane[35]. Since the storage is separate from the electrode array, flow batteries are more modular and become cheaper with larger energy capacities[16]. Vanadium redox flow batteries cost around 120 €/kW·h[16], and 12 MW·h have been installed at the Sorne Hill wind farm in Ireland - sufficient to smooth out fluctuations of a few minutes, but only enough to make up for a drop in wind power of 65% for one hour[20, p.200].

2.3.3 Pumped hydroelectric energy storage

Pumped hydroelectric energy storage (PHES) involves the connection of two large reservoirs of water, separated by a vertical distance of up to 800 m[37]. At times of excess supply the plant pumps water up to the upper reservoir, storing energy as gravitational potential until there is a supply shortage, when sluice gates open allowing flow through a turbine at the bottom. Some facilities, known as 'pump-back' PHES, produce a net power output due to glacial meltwater or rainfall in the catchment area of the upper reservoir[37].

PHES is the most common form of energy storage on the world's energy grids, making up 3% of the power capacity and 97% of the storage capacity[21]. A particularly strong early driver of development in the UK, US and Japan was nuclear plant, which requires a large power input to perform a 'black start' (in which the reactor is brought online from a fully shut down state) and is also less capable of adapting to changes in demand than fossil-fuel plant (see section 2.2.1). Construction slowed significantly in the 1990s, but recovered in the 2000s as variable renewable energy sources became a larger part of electrical grids worldwide[37].

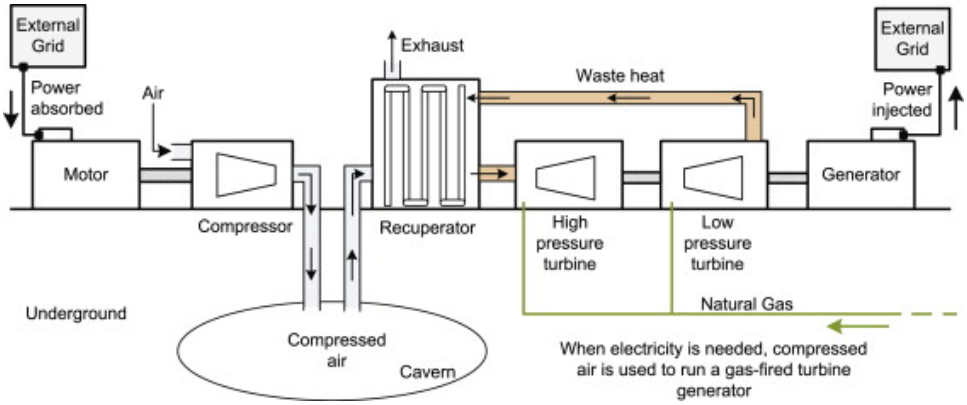


Figure 2.4: Example CAES system, taken from Díaz-González et al. [16]

Although PHES has a relatively low cost of 10–20 or 35–70 €/kW·h [16], a high turnaround efficiency and a long operating lifetime, they are considered relatively unsuitable for use purely to support wind farms. This is primarily because they are uneconomic at the small sizes most renewables require, and they additionally require project lead times of 10 years to construct - significantly longer than most wind farms [21].

2.3.4 Compressed air energy storage

With conventional compressed air energy storage (CAES), a compressor intakes air at 0.1 MPa from the atmosphere and compresses it to around 4–7 MPa for storage [35]. When the energy is required the compressed air is heated, then fed through a series of turbines linked to generators [16]. Finally, a recuperator uses the residual heat after the expansion stage to heat the air coming out of storage.

To avoid the use of expensive pressure vessels, the air may be stored around 600 m underground in disused mines, limestone caverns or underground natural gas storage caves, where the geostatic pressure provides reinforcement [35]. Salt domes, which are mined by dissolving the salt into water that is then piped out under pressure, are particularly suitable for this purpose as the salt is self-sealing under pressure [39, p.284].

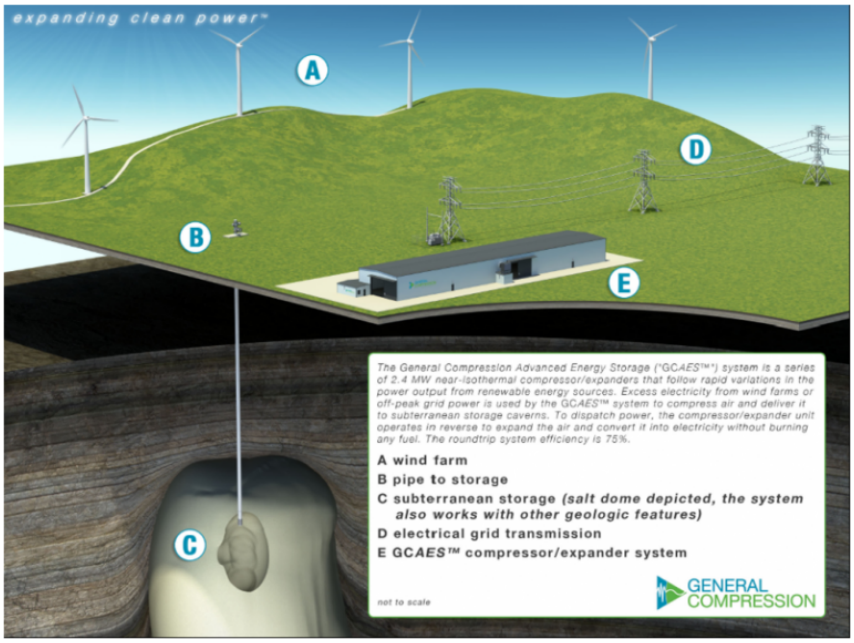


Figure 2.5: General Compression's 'GCAES' system with salt dome storage, taken from Saylor [38]

Another approach is to use hydrostatic pressure, by storing the air in reinforced fabric bags in the sea at depths around 600 m [40].

Thermal considerations

Compressing air increases both heat and pressure, due to the ideal gas law [41, p.157]:

$$pV = mR_{\text{spec}}T \tag{2.2}$$

If the thermal energy was left in the compressed air, the storage vessel would need excellent thermal insulation to avoid losing energy, in addition to the high strength the pressure requires; this is not possible in practical systems.

As reducing carbon emissions was not a priority, early plants simply discarded the thermal energy in the compressed air, replenishing it by burning natural gas before the expansion stage, which is known as 'diabatic' CAES operation [42]. This is

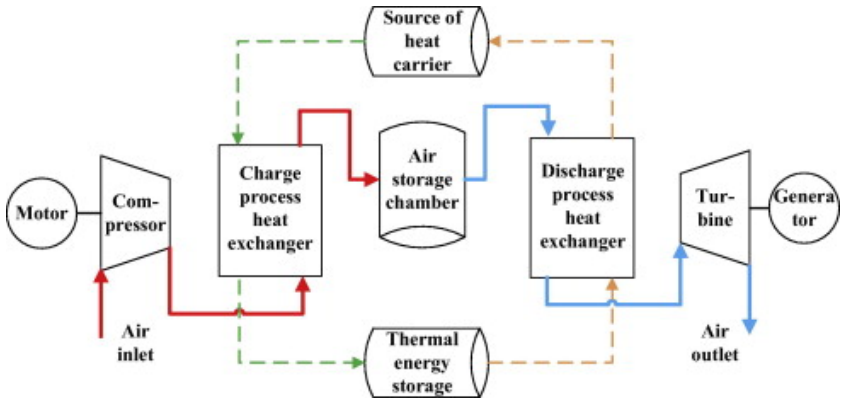


Figure 2.6: The Advanced Adiabatic CAES system, taken from Yang et al.[43]

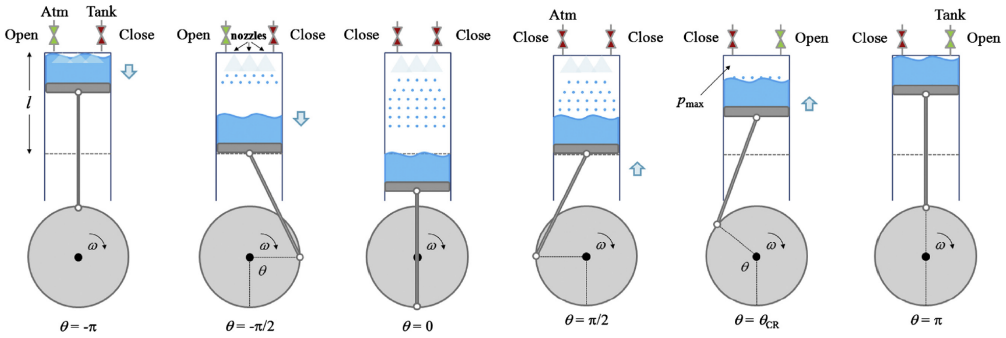


Figure 2.7: Use of a liquid piston and water-mist cooling for CAES, taken from Qin and Loth[45]

relatively inefficient and reliant on fossil fuels, so modern systems instead store the thermal energy in some way.

One approach is to use heat exchangers to remove the heat from the air after compression, as in the Advanced Adiabatic CAES project[42]. The heat is then added back into the compressed air before expansion[44]. This increases storage efficiency, but the high pressure and temperature conditions have caused problems for development of suitable turbines and exchangers to make the system economically viable[43].

Water-mist cooling has also been considered for use with CAES[45], following earlier work which considered its use for railway tunnels[46] and the compression stage of gas turbines[47]. Several companies, including LightSail Energy[48], General

Compression[49] and SustainX[50], use a mechanical piston compressor with a water spray or foam to absorb the heat during compression instead[51], which is then either used for heating of buildings or held in a thermal store[44,52]. This allows both processes to operate close to isothermally, which has the potential for significant efficiency savings; in particular, the low temperature conditions allow for the use of the same unit for compression and expansion[53].

Deployment of CAES

Initial research in the 1970s focussed on the use of CAES for load-levelling, to allow fossil-fuel and nuclear power plant to run continuously at their most efficient rate while the CAES plant absorbed fluctuations in demand[54]. The first CAES installation was built as an adjunct to a nuclear power station in 1978 at Huntorf in Germany, and is capable of providing 290 MW onto the grid for 2 hours with 90% availability. Today, it provides variable supply to the grid for 3 hours per day. The second followed in 1991 at McIntosh, AL, USA at a coal-fired plant, and can produce 110 MW for 26 hours; amongst other improvements, it was the first plant to use a recuperator, reducing fuel consumption by 25%[21].

CAES has one of the lowest costs of any large-scale energy storage system at 3–5 or 10–70 €/kW·h[16], as well as fast start-up times and high power capacities[21]. Additionally, its good long-term storage capabilities and use of standard gas turbines and compressors, allowing for modular and scalable designs, makes it an ideal form of storage for use with renewable energy harvesters. As a result, there has been a significant resurgence in interest in CAES, particular for wind farms. Plants are proposed in Cheshire, UK and California, New York, Texas, and Utah in the United States[21].

The Iowa Stored Energy Park was a 270 MW facility near a large number of wind farms, intended to be finished in 2015[21]. The plant would have used bubbles in

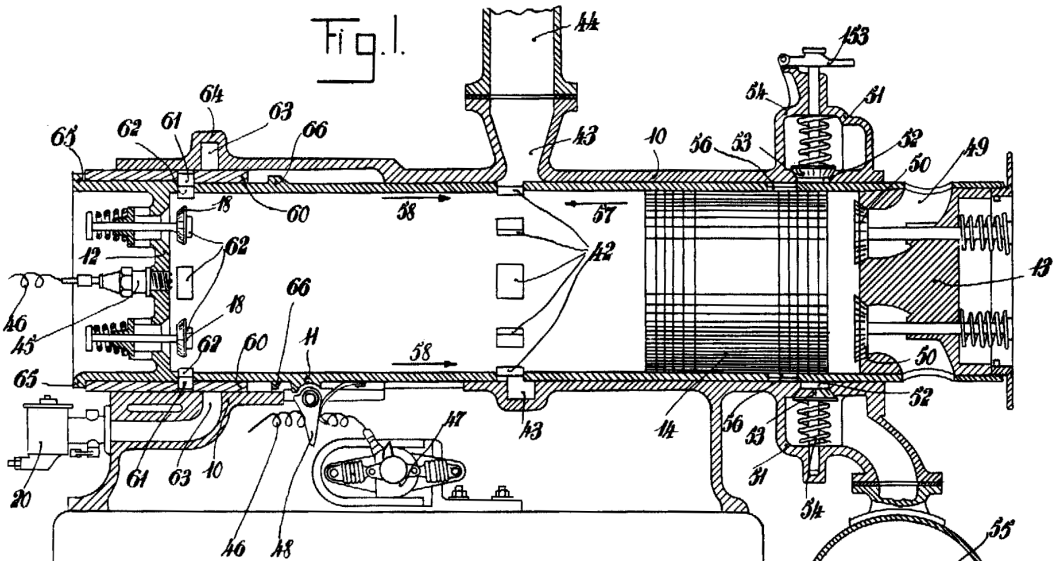


Figure 2.8: Free-piston air compressor illustration, taken from Pescara[56]

underground aquifers to store compressed air, since the displaced water would provide a back-pressure to assist with the exhaust; however, the project was terminated after problems with permeability of the aquifer sandstone proved insurmountable[55, p.76].

2.4 Free-Piston Energy Converters

Free-piston engines were first patented in 1928 by Pescara. His engines comprised a single free piston-head without connecting rod, moving in a single double-ended cylinder, as shown in Figure 2.8. After the valves (part 62) inject fuel into the left-hand end of the cylinder, the spark plug ignites the mixture and forces the piston to the right. The exhaust gases are vented through the ports (42) while the piston is compressing air on the right-hand end of the cylinder. Once it reaches sufficient pressure, the check valve (53) opens and the air is exhausted into the reservoir. The remaining air acts as a spring to 'bounce' the piston back towards the combustion end, compressing the fuel-air mixture ready for the next cycle. The air in the compressor end is replenished through the check valves (50).[56]

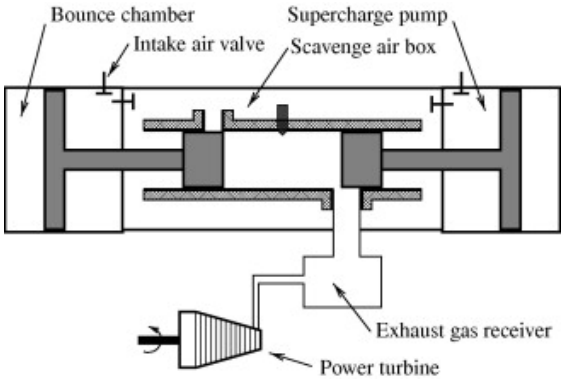


Figure 2.9: Free-piston gas generator illustration, taken from Mikalsen and Roskilly [59]

Subsequent improvements led to a multi-stage design, patented in 1941 [57], and its small size and efficiency was exploited in the German navy during the Second World War as a source of compressed air to launch torpedos [58]. Despite early research into potential applications as gas generators for automotive applications, they failed to become widespread and were effectively abandoned until more recently [59], when their potential for use in hybrid vehicles [60] and mobile robotics [61] saw a resurgence in interest.

Hydraulic free-piston engines use the linear piston motion to directly apply hydraulic force, usually combined with accumulators to store the energy [62]. They have been investigated and developed for use in construction vehicles, where their easy starting (using the stored hydraulic energy), low cost, low weight, and efficiency characteristics provide an advantage over the rotary diesel engines and hydraulic pumps conventionally used [59]. However, compared to rotary engines, they place greater demands on the control system (including the valves) due to significantly higher piston accelerations. High-speed control is thus important to ensure effective operation [63].

2.4.1 FPEC control

Control of piston motion in FPECs has concentrated on optimisation of top- and bottom-dead-centre (TDC and BDC) positions in combustion FPECs, to maximise efficiency and emissions[64]. Work on free-piston compressors by Johansen et al.[65] implemented a three-level control structure: low-level control subsystems effect the timing of compressor, exhaust and air cushion valves, while the timings are set by an intermediate-level piston motion controller, attempting to optimise the TDC and BDC positions to match efficiency and load control parameters input from the top-level supervisory controller.

Tikkanen and Vilenius[66] used a two-level control structure to set the injected fuel mass in a double-acting compressor, aiming to optimise the compression ratio and thus performance. Since actual compressor load could not be controlled, a quasi-load circuit designed to help balance the system was also introduced. Feed-forward control, allowing the structure to response to quickly-varying loads, was required in addition to normal PID control of the fuel injection.

Work by Mikalsen and Roskilly[67] concerned the optimisation of a single-acting free-piston generator, which used a similar control structure to optimise TDC and BDC positions, but additionally aimed to keep frequency roughly constant through a wide range of loads. Fuel injection rate and timing, as well as bounce chamber air mass, were controlled to compensate for large TDC errors caused by load variation. Subsequent work[68] found that basic PID control was unsuitable due to the coupled nature of the load and required TDC position. They recommended the investigation of feedforward and nonlinear control techniques. A further paper[69] used a predictive method: the piston velocity at an instant halfway through compression is used to predict TDC position using a kinetic energy approach using a linear relationship. This resulted in significantly improved performance under varying loads.

The ICWT system is similar to a free-piston engine, but the lack of any combustion

stroke leads its cycles to be quite different. In particular, the lack of ignition timing control (which is the focus for most FPEC research) requires the ICWT to be controlled through valve action alone.

2.5 Valves

Control valves, first seen in the 1930s, today come in a vast range of sizes, functions, features and capabilities. Characterising a valve, in which the input is compared to the flow reduction, is a key concern of control valves, and some classes of valve are difficult or impossible to characterise.

The commonest type of control or throttle valve is the linear-motion globe valve, in which a spherical plug is inserted into a narrow gap (called the seat) to allow or obstruct flow. The plug is moved at right-angles to the flow direction, so the flow path is necessarily nonlinear to go around it and through the seat. This type's flexibility to requirements and simplicity are key virtues, but its relatively large size and weight often count against it.[70]

Butterfly valves, in which a disk rotating provides the regulating element, are also used for throttling. Generally preferred in low-pressure applications or where size or weight is important, a key advantage of butterfly valves is their linear flow path - when fully open, the flow need not make any turns at all, and only the edge profile of the valve obstructs the flow. However, they develop significant torques from the flow when used in high-pressure environments, which introduces significant control difficulties. They are best used as on-off valves, although their high actuation speeds opens the possibility of 'bang-bang' (binary) control methods.[70]

A similar design is the ball valve, in which a solid ball with a bored hole is rotated to align the hole with the flow. These share many advantages with butterfly valves, and also have a linear flow path and a very high capability to resist pressure differences

(due to the large surface area supporting the ball). Characterisable-ball valves, which use half of a full sphere to regulate the flow, are better for control; eccentric plug valves operate on a similar basis, rotating a plug away from a set to avoid restricting the flow path.[70]

One important concern for this project is valve balancing, which determines how the pressure difference across a valve affects the ease of opening or closing the valve. Since the tube-end valves in the rotor will need to operate extremely quickly, of the order of 0.01 seconds, it is very important that the force required to operate them is small.[71]

We also require a very low residence volume in the end of the piston tube, to improve the compression and exhaust efficiency; to this end, the valves chosen will need to be located very close to the tube and be designed to take up very little space in the tube when closed.

2.6 State-space simulations

The modelling of physical systems by the use of differential equations is well understood. This thesis is concerned principally with the construction of a state-space model of nonlinear differential equations.

In constructing a simulation, the first step is to categorise the variables which define the system behaviour into inputs, which are unaffected by the system; and state variables, which represent a particular quantity of the system itself; and outputs, which are the values we are interested in solving for (possibly including one or more state variables)[72, p.62]. These are commonly combined into vectors, here written as the input vector $\mathbf{u}$, the state vector $\mathbf{y}$, and the output vector $\mathbf{q}$.

For example, in the case of a vehicle model, the inputs might include both environmental attributes such as the acceleration due to gravity and the road conditions,

and core system parameters such as the vehicle mass and throttle position. The state variables contain enough information to completely describe the state of the system at any particular time t; depending on the complexity of the simulation, they might be as simple as the vehicle's position and velocity, or they could include details such as the current pitch, roll and yaw angles and angular speeds, the suspension position for each wheel, and the fill level of the fuel tank. The outputs might be the fuel consumption or the velocity.[72, p.2–3]

The minimum number of state variables is equal to the number of independent energy levels in a system. For example, in a simple pendulum, energy may be stored either as gravitational potential energy, or as kinetic energy; therefore, there must be at least two state variables in the model[73, p.86].

2.6.1 Ordinary differential equations

The governing equations of any complex system will generally be expressed as ordinary differential equations, which are simply relations between state and input variables which include their derivatives. For example, we consider a simple mass-spring system in which the only state variable is the displacement of the mass. The acceleration (the second derivative of position with respect to time) is:

$$\frac{\mathrm{d}^2 y}{\mathrm{d}t^2} = -\frac{k}{m} y \tag{2.3}$$

To remove the higher-order $\frac{\mathrm{d}^2\mathbf{y}}{\mathrm{d}t^2}$ term, a substitution is employed[74, p.225]. We add a second state variable, the velocity, to obtain:

$$\begin{aligned} \frac{\mathrm{d}y_1}{\mathrm{d}t} &= y_2 \\ \frac{\mathrm{d}y_2}{\mathrm{d}t} &= -\frac{k}{m} y_1 \end{aligned} \tag{2.4}$$

Using dot notation and matrix form, this is written:

$$\begin{bmatrix} \dot{y_1} \\ \dot{y_2} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & 0 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \tag{2.5}$$

$$\dot{\mathbf{y}} = \mathbf{K}\mathbf{y} \tag{2.6}$$

The general form of this is:[75, p.23–24] [73, p.86]:

$$\dot{\mathbf{y}}(t) = \mathbf{A}(t)\mathbf{y}(t) + \mathbf{B}(t)\mathbf{u}(t) \tag{2.7}$$

$$\mathbf{q}(t) = \mathbf{C}(t)\mathbf{y}(t) + \mathbf{D}(t)\mathbf{u}(t) \tag{2.8}$$

where $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ and $\mathbf{D}$ are matrices.

Some systems are nonlinear, in which the governing ODEs cannot be written as a linear combination of $\mathbf{y}$ and $\mathbf{u}$. An ODE might involve two variables multiplied together, a variable raised to a power, or the derivative of a state variable; for example, the governing equation of a pendulum at an angle θ from the vertical is:

$$\ddot{\theta} = -\sin\theta \tag{2.9}$$

Since $\sin\theta$ is not linear, the matrices cannot be extracted and the general form is instead[73, p.89]:

$$\dot{\mathbf{y}}(t) = f(\mathbf{y}, \mathbf{u}, t) \tag{2.10}$$

$$\mathbf{q}(t) = g(\mathbf{y}, \mathbf{u}, t) \tag{2.11}$$

Most nonlinear differential equations do not have analytical solutions. Their solution is normally via numerical methods.

2.6.2 Numerical methods for nonlinear ODEs

The earliest technique for solving ODEs numerically was formulated by Leonhard Euler in 1768.[72, p.121]

Euler's method

First, rearrange in the form:

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(y,t) \tag{2.12}$$

Next, using Taylor's approximation to a first-order continuous derivative:

$$\left.\frac{\mathrm{d}y}{\mathrm{d}t}\right|_{t_0} = \lim_{\delta t \to 0} \frac{y(t_0+\delta t) - y(t_0)}{\delta t} \tag{2.13}$$

we combine and rearrange, to obtain an equation for the next state in terms of the current state y_0 and time t_0[72, p.121] [75, 226–228].

$$y(t_0+\delta t) \approx y_0 + f(y_0,t_0)\delta t \tag{2.14}$$

In simple terms, the algorithm draws a tangent to the solution curve at the current point, then steps a distance δt along that tangent. If the solution curve is locally 'well-behaved', meaning it has a low second derivative so the gradient is relatively constant across the timestep, then the approximation will be accurate; however it will be very inaccurate if the gradient changes too quickly, known as a 'stiff' problem.[72, p.121–125]

Modified Euler's method

A simple modification is modified Euler, which additionally calculates the gradient at the second point and averages the two gradients to obtain an accurate result [72, p.125]:

$$\begin{aligned} k_1 &= f(y_0, t_0) \\ \tilde{y}(t_0 + \delta t) &= y_0 + k_1 \delta t c \\ k_2 &= f(\tilde{y}(t_0 + \delta t), t_0 + \delta t) \\ y(t_0 + \delta t) &\approx y_0 + \frac{\delta t}{2}(k_1 + k_2) \end{aligned} \tag{2.15}$$

Although requiring twice as many evaluations of the (often computation-expensive) derivative function, this change can reduce the error by more than half for the same set of steps [72, p.126].

Runge-Kutta method

It may be noted that the Euler methods use only the first term of Taylor's approximation to a derivative. By increasing the number of Taylor terms used to model the gradient, we may further increase accuracy. This is effectively fitting the curve with higher-order polynomials instead of straight tangents. This was generalised in 1900 as the Runge-Kutta family of methods, of which the most common is the 4th-order algorithm [72, p.127]:

$$y(t_0 + \delta t) \approx y_0 + \frac{\delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{2.16}$$

$$\begin{aligned} \text{where } k_1 &= f(y_0, t_0) \\ \text{and } k_2 &= f\left(\left(x_0 + \frac{\delta t}{2} k_1\right), \left(t_0 + \frac{\delta t}{2}\right)\right) \\ \text{and } k_3 &= f\left(\left(x_0 + \frac{\delta t}{2} k_2\right), \left(t_0 + \frac{\delta t}{2}\right)\right) \\ \text{and } k_4 &= f\left(\left(x_0 + \delta t k_3\right), \left(t_0 + \delta t\right)\right) \end{aligned}$$

The Dormand-Prince method is a modification of the Runge-Kutta methods for 4th- and 5th-order evaluation. The algorithm, created in 1980[76] and also known as DOPRI, has several advantages; most notable is that it is particularly effective for multiple steps, due to the last function evaluation each step being identical to the first evaluation of the next step, known as First Same As Last (FSAL).

Dormand-Prince is built into MATLAB as the basis for the `ode45` ODE solving function[77]. A 4th-order polynomial interpolant also allows MATLAB to both generate additional points in the output without requiring more function evaluations, and allows the precise location of user-defined events, when some function $g(\mathbf{y}, t)$ is equal to zero.

Adaptive step sizes

One way to improve the accuracy of any of the Runge-Kutta family of methods is simply to reduce the step size, but this leads to a corresponding increase in the number of function evaluations required and thus the computational resources consumed[74, p.229]. To strike the best balance, adaptive step size methods have been developed, which test and retry with smaller steps if the error is too great. This is a standard part of most ODE solvers in MATLAB[77].

A particular advantage is seen with strongly nonlinear systems, which can be stiff in some parts of the state space (requiring very small steps) while being nonstiff elsewhere[72, p.129]. An example of this might be two particles acting under a gravitational attraction; while the particles are distant, the forces on them will be small and the timesteps can be correspondingly large, but as they move closer together the forces increase rapidly demanding smaller timesteps to retain accuracy. An adaptive step size algorithm allows the timestep to scale efficiently to the local requirements.

Numerical Differentiation Formulas

For extremely stiff systems the Runge-Kutta methods are insufficiently stable, requiring timesteps too small for the working precision, and numerical differentiation formula (NDFs) are used instead. First developed by Klopfenstein in the 1970s[78], NDFs work by constructing local Jacobian matrices, which include information from previous steps to allow increased stability when dealing with high stiffness[79].

The MATLAB function `ode15s` is based on an NDF method, and includes the same event-handling capabilities of `ode45`. It is capable of solving considerably stiffer problems than `ode45`, and additionally requires less resources. However, it is slightly less accurate[77].

2.7 Control

2.7.1 PID control

The most basic form of control is proportional. The error (the difference between the current state and the desired value) is calculated, multiplied by a gain value, and fed back into the system. This has a few disadvantages, most notably a non-zero steady-state error, in which the output will reach a point where the disturbance and the control signal balance out; also, in higher-order systems, high proportional gains can cause instability. However, proportional control is extremely simple, relatively stable, and quick, so it forms the main part of most control applications.[73,74,80]

To avoid this steady-state error, an integrator is introduced into the control system. This takes the integral of the error signal and multiplies it by a different gain value, then adds it to the control signal. Integrators 'detect' the small steady-state error by allowing it to build up over time, eventually cancelling it out; since the initial error remains 'stored' in the integrator's state, it does not stop working when the error

reaches zero. However, this can introduce oscillations to the system and instability to the system, especially when combined with a high proportional gain.[73,74,80]

The third component, then, is derivative control. Unlike the others, derivative control in its 'pure' sense depends on knowledge of the future behaviour of the error; in practice, this needs to be estimated based on recent observations. Including the derivative of the error signal allows the control system to 'predict' upcoming error changes; for instance, if the error is heading rapidly towards zero and there is a risk of overshoot, the derivative error signal acts to 'damp' the controller down and reduce any oscillation that might occur. Correctly-tuned derivative control can even render otherwise unstable systems stable.[73,74,80]

Most modern electronic controllers are 'PID' controllers, meaning they include all three components and allow them to be individually adjusted to tune the system. A variety of more exotic controllers also exist, which are generally designed for specialist situations; for instance, fractional-order control involves a modified integrator function, designed to avoid having a single natural frequency for the controller. Many of these more complex controllers have internal dynamics, however; for instance, a simple low-pass filter (to filter out high-frequency noise from the control signal) involves a time delay. This is a key factor in many decisions to use simpler control systems.[74]

One relevant subtype of PID control is PD control, in which the integral term is omitted. This is extremely fast, and it is useful in cases where the intention is to track a moving target. However, the controller is susceptible to high-frequency noise, which will be exaggerated by the derivative component, and since it lacks the integral part it also has a degree of steady-state error.[73,80]

2.7.2 Model Predictive Control

Model Predictive Control (MPC), also known as receding horizon control, first developed in the 1970s, is a method for controlling a process which uses a computational model of the process itself. Detailed information about the current state of the system is fed into the model, which simulates the effect of many changes to the control signal over a short timescale; the first step of the most successful strategy is then implemented, and the controller starts again for the next system state. [81]

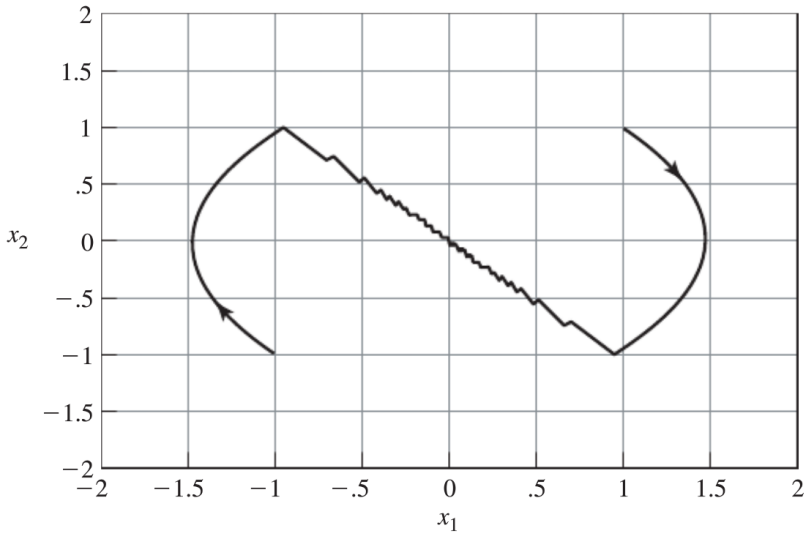
A key advantage of MPC is its ability to deal with significant time delays. The main downside is that nonlinear systems can be prohibitively complex to simulate at the speed required for practical implementations of MPC; however, many systems can be linearised about the operating point, and other systems are modelled using closely-fitted orthonormal bases of polynomials to simplify calculation. [81]

2.7.3 Sliding mode control

For some systems, it is extremely hard to define a stable single control structure which provides sufficient regulation. To assist in solving these cases, variable structure control systems (VSCSs) were developed, based on the work by Emel'yanov and Barbashin in the USSR in the early 1960s [82, p.1].

VSCSs define multiple separate control substructures, and select which to use during operation based on the current operating state. By careful choice of switching function, the system can guarantee that every control substructure is stable for the entire region it is applied.

The most common form of VSCS is sliding mode control (SMC), in which a 'sliding surface' [82, p.5] is first defined (also known as a 'discontinuity' [83, p.320] or 'switching' surface [84, p.11] for obvious reasons). This is a particular relationship between all of the controlled variables, which governs the final system response.



An illustration of chattering in a variable structure sliding mode system consisting of the double integrator and the control law, $u = -\text{sign}(x_1 + x_2)$. Two trajectories starting from two different initial conditions are plotted.

Figure 2.10: Chatter in sliding mode control, taken from Zak [83, p.319]

Next, control substructures are devised such that they always direct the system back towards the sliding surface. Finding suitable substructures is known as the reachability problem; they may take the form of PID controllers or more advanced laws [82, p.41-46]. They are required to exhibit convergence on the sliding surface in finite time, not asymptotic [84, p.29], and they must themselves be stable in their respective regions (though not necessarily outside them) [83, p.324]. Reducing the 'reaching time' taken is particularly important since the system only reaches its robustness goals when on the surface; very high input gains are used to minimise this timespan [82, p.31].

The theoretical result is that the state trajectories of the controlled system quickly converge on the sliding surface for any starting position, and then the closed-loop feedback of the appropriate substructures holds the system onto it. When on the surface the system is operating in the eponymous 'sliding mode', in which its order is reduced to beneficial effect [84, p.158]. In practice, there is a integration timestep problem. The system, rather than being locked onto the sliding surface, oscillates

through it as the switching function repeatedly changes control structure in finite intervals [82, p.32]; in the theoretical case, this occurred at an infinite frequency [84, p.132]. This 'chattering' effect is shown in Figure 2.10.

Caused by the discontinuous nature of the SMC method, chattering causes heat losses, high mechanical wear and control inaccuracy in practical systems. One solution is to define a 'boundary layer' around the sliding surface, within which a different control structure is applied; it is even possible to select control laws such that the system becomes continuous around the surface. Ultimately however, any solution to chattering will necessarily dilute the robustness and efficacy of SMC [84, p.135–140].

Chapter 3

System modelling

This chapter will first outline the process of finding a set of design parameters, which describe the example ICWT system we will investigate. A simple system of coupled ODEs is then outlined and developed into a detailed simulation. Finally, we will cover details of the way the system has been implemented in MATLAB.

3.1 Reference turbine specifications

For an ideal gas in adiabatic compression, we have the relationship [41, p.180]:

$$pV^{\gamma} = k \tag{3.1}$$

We can find the energy required to compress air in terms of r, the ratio of final to initial pressure, by integrating an adiabatic process with respect to pressure, from a pressure p_0 and a volume V_0 up to a pressure $p_0 r$. We also introduce $\rho_{a,0}$, the density

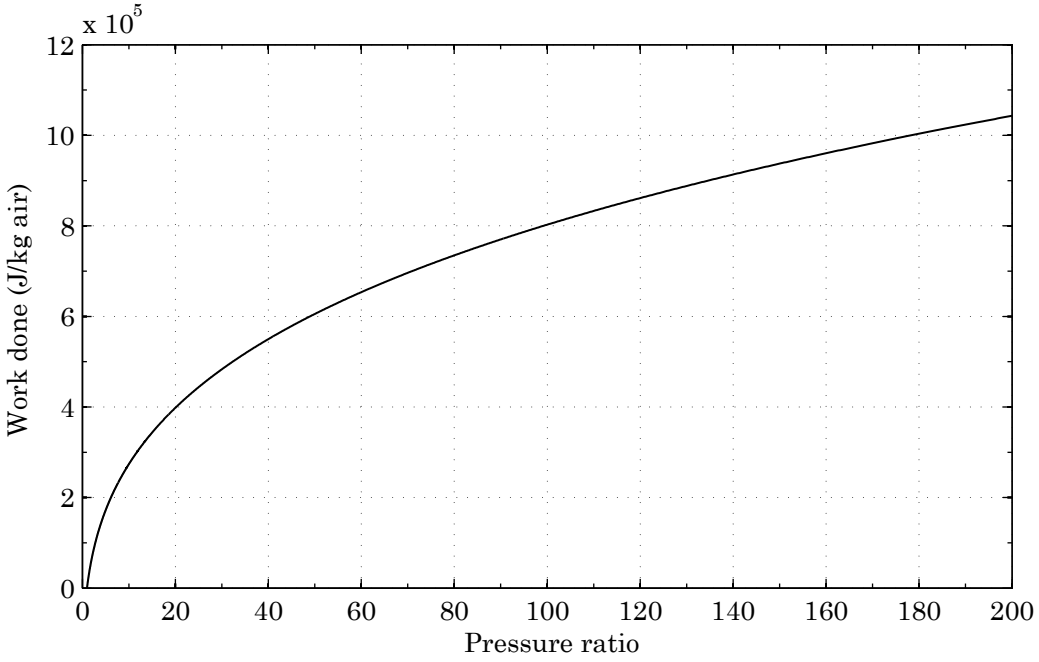


Figure 3.1: Work done in an adiabatic compression process

of air at the initial conditions.

$$E_{\text{comp}} = \int V \, \mathrm{d}p \tag{3.2}$$

$$E_{\text{comp}} = k^{\frac{1}{\gamma}} \int_{p_0}^{p_0 r} p^{\frac{1}{\gamma}} \, \mathrm{d}p$$

$$E_{\text{comp}} = p_0^{\frac{1}{\gamma}} V_0 \left(\frac{\gamma}{\gamma - 1} \right) \left(p_0^{\frac{\gamma-1}{\gamma}} \left(r^{\frac{\gamma-1}{\gamma}} - 1 \right) \right)$$

$$\frac{E_{\text{comp}}}{V_0 \rho_{a,0}} = \frac{p_0}{\rho_{a,0}} \left(\frac{\gamma}{\gamma - 1} \right) \left(r^{\frac{\gamma-1}{\gamma}} - 1 \right) \tag{3.3}$$

If we take $p_0 = 101.325$ kPa[85, p.26], $\rho_{a,0} = 1.207$ kg/m^3 and $\gamma = 1.401$[85, p.16], we can calculate the work done per kg of air for a range of pressure ratios, as shown in Figure 3.1. The curve clearly levels off at high pressures; to strike a good balance between high power density and high containment costs, we set our target $p_{\text{targ}} = 7$ MPa, a pressure commonly used in CAES systems[35] with an energy density (assuming adiabatic expansion) of 696 kJ/kg air.

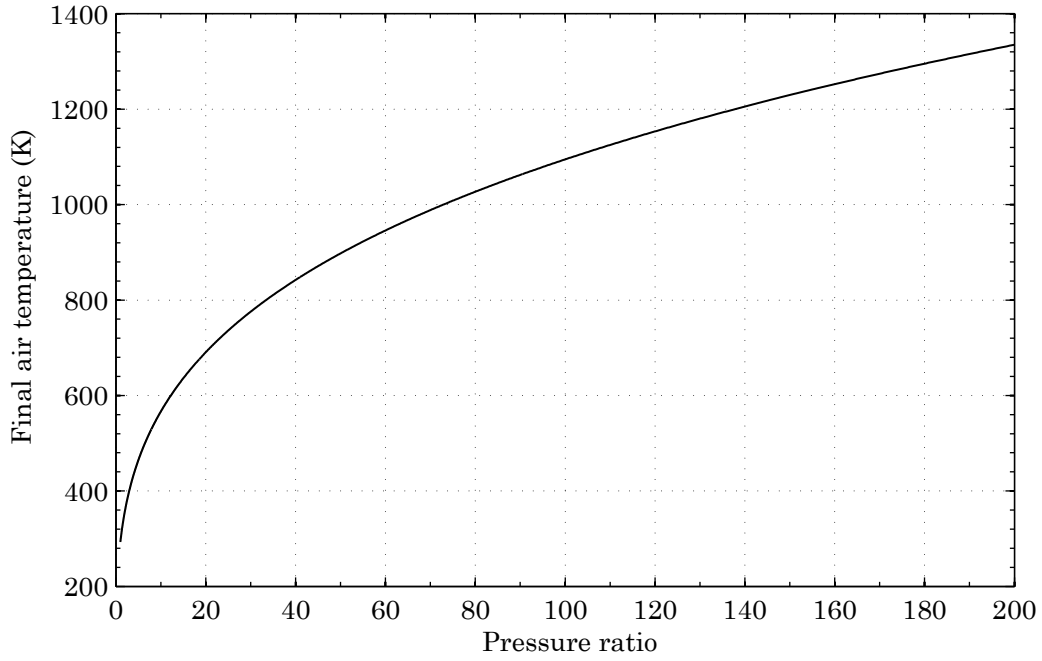


Figure 3.2: Final air temperature for an adiabatic compression process starting at 293 K

The temperature after adiabatic compression is given by [41, p.180]:

$$T = T_0 r^{\frac{\gamma-1}{\gamma}} \tag{3.4}$$

This relationship, for an initial temperature of 293 K, is plotted in Figure 3.2. The temperature after compression to 7 MPa is around 989 K.

Next, we calculate our tip-speed ratio λ using Equation 1.2. For this study, we will take $L_{\mathrm{B}} = 150$ m and $\lambda = 6.3$ at $v_{\mathrm{W}} = 9.5$ m/s, resulting in a rated rotor speed $\dot{\theta}$ of 0.4 rad/s.

If our rotor achieves a normal rotor power coefficient c_{PR} of 70% of the Betz limit, or 0.4148 overall, then at $v_{\mathrm{W}} = 9.5$ m/s we can use Equation 1.3 to calculate that the design rotor will harvest a power of 15.2 MW. Given 4 separate compression tubes and 2 strokes per cycle, this equates to 29.9 MJ per stroke.

The maximum possible energy which could be extracted from a mass m_P falling a distance L_B under an acceleration due to gravity of g is trivially $m_P \cdot h \cdot g$, but our pistons will not fall perfectly vertically nor for the full length of the blade. If we assume we will be able to capture 80% of the theoretical maximum gravitational potential energy, and the pistons have a range of $L_\mathrm{B} - 20 = 130$ m, we find each piston pair will need a mass of around $29.2{\times}10^3$ kg.

To store 29.9 MJ in each stroke at an energy density of 696 kJ/kg air will require the induction of 35.5 m^3 of air per stroke. If we set the compression tube cross-sectional area $A = 1$ m^3, the tube will be able to hold 150 m^3 in total, and induction of the air through ports on the tube walls away from the ends will be easily able to accomplish this.

3.2 Model derivation

3.2.1 Non-rotating model

We begin with a very simple non-rotating model with just 4 state variables; piston position h, piston speed $\dot{h}$, pressure above piston p_1 and pressure below piston p_2, as shown in Figure 3.3.

From the equation of state, we have:

$$m\ddot{h} = A(p_2 - p_1) - mg \tag{3.5}$$

Adding in the piston velocity as the state variable $\dot{h}$, the standard substitution is used to reduce the order of the ODE, with $\dot{h} = z$ and $\ddot{h} = \dot{z}$, obtaining two simple

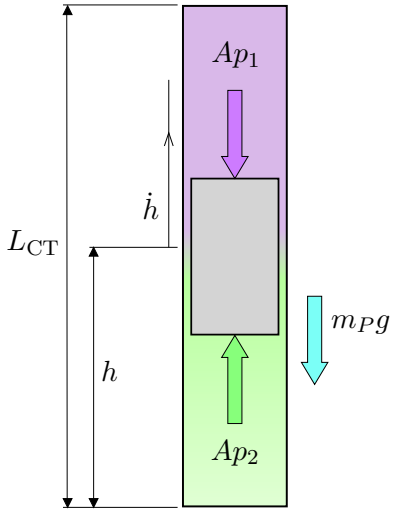


Figure 3.3: Forces and variables in non-rotating model

first-order differential equations:

$$\dot{h} = z \tag{3.6}$$

$$\ddot{h} = \frac{A}{m}(p_2 - p_1) - g \tag{3.7}$$

The compressibility factor of a gas, which how closely its behaviour conforms to the ideal gas law, is given by:

$$Z = \frac{p}{\rho R_{\mathrm{spec}} T} \tag{3.8}$$

Air has a compressibility factor of 0.9997 at 300 K and 101.3 kPa[85, p.16,26], very close to 1, so it will be treated as an ideal gas. We will initially assume that heat transfer to and from the compressed air is negligible, due to the short timescales involved, implying that the compression cycle is adiabatic. This gives[41, p.180]:

$$pV^{\gamma} = k \tag{3.9}$$

This equation is true provided the mass of air in the cylinder is constant. We

will calculate k_1 and k_2 (above and below the piston respectively) from the initial pressures ($p_{1,0}$ and $p_{2,0}$) and initial piston position (h_0, or $(L_{\mathrm{CT}} - h_0)$ in the rimwards direction).

$$k_1 = p_{1,0}\left(\left(L_{\mathrm{CT}} - h_0\right)A\right)^{\gamma} \tag{3.10}$$

$$k_2 = p_{2,0}(h_0 A)^{\gamma} \tag{3.11}$$

Rearranging these and differentiating with respect to time, we obtain:

$$\dot{p}_1 = \frac{k_1}{A^{\gamma}}\gamma\dot{h}(L_{\mathrm{CT}} - h)^{-1-\gamma} \tag{3.12}$$

$$\dot{p}_2 = \frac{k_2}{A^{\gamma}}\gamma\dot{h}h^{-1-\gamma} \tag{3.13}$$

We combine these into vector form:

$$\begin{bmatrix} \dot{h} \\ \ddot{h} \\ \dot{p}_1 \\ \dot{p}_2 \end{bmatrix} = \begin{bmatrix} z \\ \frac{A}{m}(p_2 - p_1) - g \\ \frac{k_1}{A^{\gamma}}\gamma\dot{h}(L_{\mathrm{CT}} - h)^{-1-\gamma} \\ \frac{k_2}{A^{\gamma}}\gamma\dot{h}h^{-1-\gamma} \end{bmatrix} \tag{3.14}$$

$$\dot{\mathbf{y}} = f(\mathbf{y}) \tag{3.15}$$

This form is suitable for use with the ODE solvers built into MATLAB.

3.2.2 Basic rotating model

Adding constant-speed rotation to the previous system is relatively simple. We need only add a state variable representing the current angle of the system from horizontal, θ, adjust the gravitational acceleration to rotate with it, and add in a term due to

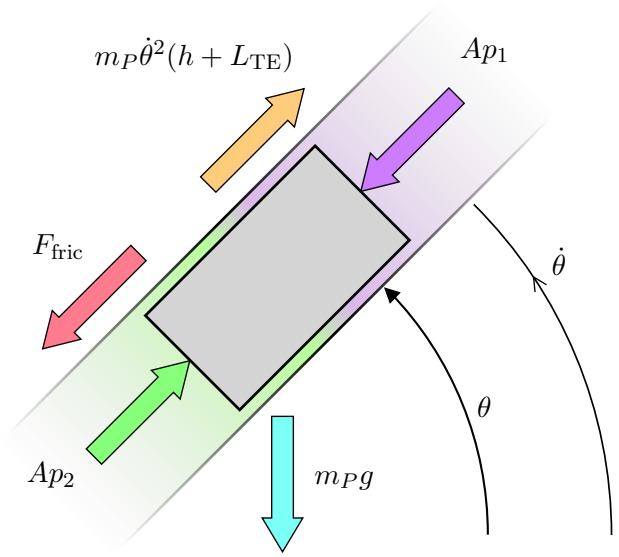


Figure 3.4: Forces and variables in basic rotating model

centripetal acceleration. This makes the acceleration part of the ODE:

$$\ddot{h} = \frac{A}{m_P}(p_2 - p_1) - g\sin\theta + \dot{\theta}^2(h + L_{\text{TE}}) \tag{3.16}$$

We will also add Coulomb friction to the model, which is a constant μ outside a chosen velocity bandwidth and ramps linearly at low velocities inside the limits set by $\dot{h}_{\text{HB}}$, the 'velocity half-bandwidth';

$$\text{Friction force, } F_\mu = \begin{cases} \mu & \text{for } \dot{h} > \dot{h}_{\text{HB}}, \\ \frac{\mu}{\dot{h}_{\text{HB}}}\dot{h} & \text{for } -\dot{h}_{\text{HB}} < \dot{h} < \dot{h}_{\text{HB}}, \\ -\mu & \text{for } \dot{h} < -\dot{h}_{\text{HB}}. \end{cases} \tag{3.17}$$

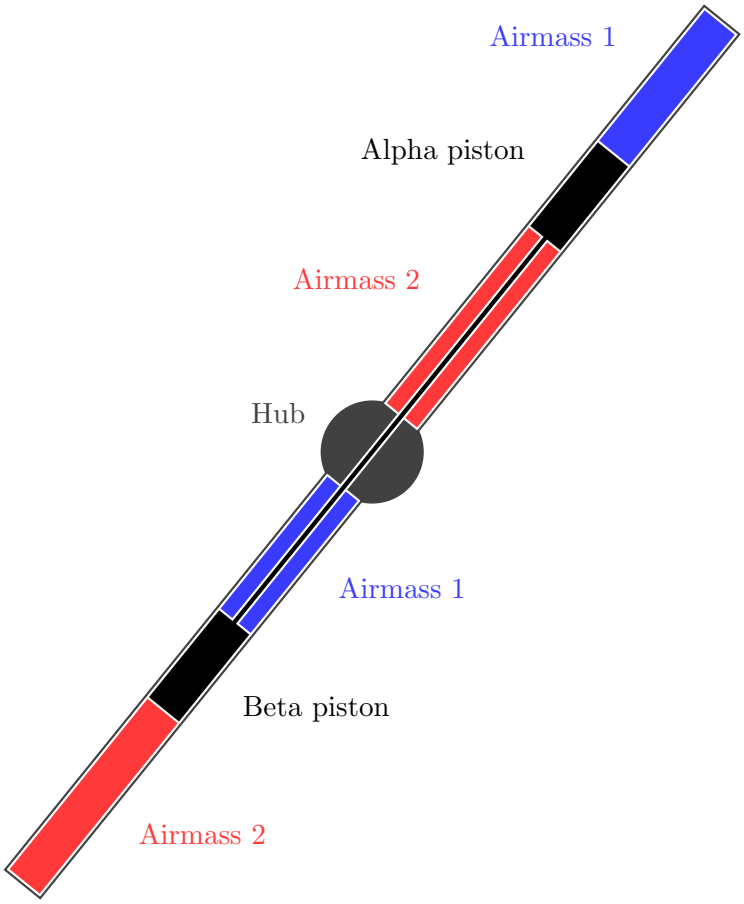


Figure 3.5: Air masses with connected pistons

Including this friction in the acceleration expression, the complete model is thus:

$$\begin{bmatrix} \dot{h} \\ \ddot{h} \\ \dot{p_1} \\ \dot{p_2} \end{bmatrix} = \begin{bmatrix} \dot{h} \\ \frac{A}{m_P}(p_2 - p_1) - g\sin\theta + \dot{\theta}^2(h + L_{\text{TE}}) - \frac{F_{\text{fric}}}{m_P} \\ \frac{k_1}{A^\gamma}\gamma\dot{h}(L_{\text{CT}} - h)^{-1-\gamma} \\ -\frac{k_2}{A^\gamma}\gamma\dot{h}h^{-1-\gamma} \end{bmatrix} \tag{3.18}$$

3.2.3 Connected piston model

To expand the model to two connected opposing pistons, we will assume that the connecting rod is perfectly rigid. Since the pistons are the same distances from the

ends of their tubes, the pressures and masses are all equivalent. The centre of mass, for calculating the centrifugal force, is located halfway between the two pistons.

The pistons are referred to as the 'alpha' piston (with pressures p_1 on its rimward face and p_2 on its hubward face) and 'beta' piston (with pressures p_2 on its rimward face and p_1 on its hubward face). For clarity, directions will be given relative to the α piston, so 'alpha-hubwards' refers to the direction from the alpha piston towards the rotor hub (and is thus equivalent to 'beta-rimwards'). This is shown in Figure 3.5. A new variable of 'airmass height' h_a is introduced, which is given by:

$$\begin{aligned} h_{a,1} &= L_{\mathrm{CT}} - h \\ h_{a,2} &= h \end{aligned} \tag{3.19}$$

3.2.4 Limitations of pressure state variable

This model has a significant problem: it does not account for mass flow of air into or out of the cylinder. To add mass flow to the model, we will need to replace the pressure state variables with temperature state variables.

3.3 Mass-based model

This model has three core state variables for each compression chamber: volume V, mass m, and temperature T. The rate of change of volume is trivially given by the piston velocity $\dot{h}$ and the tube cross-sectional area A.

3.3.1 Mass derivative

Mass flow rate $\dot{m}$ will vary depending on three valve constants;

$k_{\mathrm{m\,in}}$ Inflow valve, allowing air to enter the cylinder whenever the pressure is below atmospheric pressure p_{atm}.

$k_{\mathrm{m\,exh}}$ Exhaust valve, allowing air to leave the cylinder into the HP manifold when the pressure is above the target pressure p_{targ}.

$k_{\mathrm{m\,out}}$ Outflow valve, allowing air to flow out of the cylinder to increase piston speed before it begins compression.

3.3.2 Temperature derivative

In order to add temperature to this model, we consider two components contributing to rate of change of temperature.

Adiabatic compression

Firstly, in the case of adiabatic compression (with no thermal energy transfer), we have the following relation between temperature and specific volume[41, p.180];

$$TV^{\gamma-1}m^{1-\gamma} = \text{constant} \tag{3.20}$$

where γ is the heat capacity ratio of air. Differentiating under the product rule;

$$\begin{aligned}
0 &= V^{\gamma-1}m^{1-\gamma}\left(\dot{T}_{\mathrm{comp}}\right) + Tm^{1-\gamma}\left((\gamma-1)V^{\gamma-2}\dot{V}\right) + TV^{\gamma-1}\left(-(\gamma-1)m^{-\gamma}\dot{m}\right) \\
\dot{T}_{\mathrm{comp}} &= T(\gamma-1)\left(V^{\gamma-1}m^{-\gamma}\dot{m} - m^{1-\gamma}V^{\gamma-2}\dot{V}\right)V^{1-\gamma}m^{\gamma-1} \\
\dot{T}_{\mathrm{comp}} &= T(\gamma-1)\left(V^{1-\gamma+\gamma-1}m^{\gamma-1-\gamma}\dot{m} - m^{\gamma-1+1-\gamma}V^{1-\gamma+\gamma-2}\dot{V}\right) \\
\dot{T}_{\mathrm{comp}} &= T(\gamma-1)\left(\frac{\dot{m}}{m} - \frac{\dot{V}}{V}\right) \\
\dot{T}_{\mathrm{comp}} &= T(\gamma-1)\left(\frac{\dot{m}}{m} - \frac{\dot{h_a}}{h_a}\right)
\end{aligned} \tag{3.21}$$

We now have an expression for $\dot{T}_{\text{comp}}$, the rate of change of temperature due to compression.

Thermal conduction

The second case we will consider is that of a stationary volume of hot compressed air losing heat to the surrounding cylinder. Here, the rate of change of temperature is given by [41, p.508]:

$$\dot{T}_{\text{trans}} = \frac{U h_a (T_{\text{CT}} - T)}{m c_{v,a}} \tag{3.22}$$

Where U is the heat transfer coefficient per unit length of the compression tube surface.

Overall

These two components can be superposed, obtaining;

$$\dot{T} = T(\gamma - 1)\left(\frac{\dot{m}}{m} - \frac{\dot{h_a}}{h_a}\right) + \frac{U\,(T_{\text{CT}} - T)}{m c_{v,a}} \tag{3.23}$$

3.3.3 Pressure function

Volume and mass are state variables, so we will find a relation for p which we can use in all circumstances - including during air flow into or out of the cylinder. We will use the ideal gas law [41, p.157]:

$$p(Ah) = mTR_{\text{spec}} \tag{3.24}$$

and the isentropic compression relation;

$$\frac{T}{T_{\text{atm}}} = \left(\frac{p}{p_{\text{atm}}}\right)^{\frac{\gamma-1}{\gamma}} \tag{3.25}$$

Combining these two relations, we can eliminate the current temperature T and obtain;

$$p = \left(\frac{mT_{\text{atm}}R_{\text{spec}}}{V}\right)^{\gamma} p_{\text{atm}}^{1-\gamma} \tag{3.26}$$

The model can calculate the pressure as an internal variable, updated each timestep based on the current values of state variables m and h.

3.3.4 Complete ODE

We can now detail the complete ODE for this model;

$$
\begin{aligned}
p_1 &= \left(\frac{m_1 T_{\text{atm}} R_{\text{spec}}}{A(L_{\text{CT}} - h)}\right)^{\gamma} p_{\text{atm}}^{1-\gamma} \\
p_2 &= \left(\frac{m_2 T_{\text{atm}} R_{\text{spec}}}{Ah}\right)^{\gamma} p_{\text{atm}}^{1-\gamma} \\
\ddot{h} &= \frac{A}{m_P}(p_2 - p_1) - g\sin\theta + \dot{\theta}^2(h + L_{\text{TE}}) - \frac{F_{\text{fric}}}{m_P} \\
\dot{m_{a,1}} &= \begin{cases}
k_{\text{m in}}(p_{\text{atm}} - p_1) & \text{for } p_1 < p_{\text{atm}}, \\
k_{\text{m exh}}(p_{\text{targ}} - p_1) & \text{for } p_1 > p_{\text{targ}}, \\
k_{\text{m out}}(p_{\text{atm}} - p_1) & \text{if } p_1 > p_{\text{atm}} \text{ and mode is freefall}, \\
0 & \text{otherwise.}
\end{cases} \\
\dot{m_{a,2}} &= \begin{cases}
k_{\text{m in}}(p_{\text{atm}} - p_2) & \text{for } p_2 < p_{\text{atm}}, \\
k_{\text{m exh}}(p_{\text{targ}} - p_2) & \text{for } p_2 > p_{\text{targ}}, \\
k_{\text{m out}}(p_{\text{atm}} - p_2) & \text{if } p_2 > p_{\text{atm}} \text{ and mode is freefall}, \\
0 & \text{otherwise.}
\end{cases}
\end{aligned}
\tag{3.27}
$$

3.3.5 Work done

We describe two distinct methods of tracking work done.

Pressure force method

This method is based on the work done against the pressure forces on the piston.

$$
E_{\text{WD}} = \int F \, \mathrm{d}h \tag{3.28}
$$

In the alpha-rimwards direction;

$$\begin{aligned} E_{\mathrm{WD,p}} &= \int 2A(p_1 - p_2)\,\mathrm{d}h \\ \dot{E}_{\mathrm{WD,p}} &= \frac{\mathrm{d}}{\mathrm{d}h}\left(\int 2A(p_1 - p_2)\,\mathrm{d}h\right)\frac{\mathrm{d}h}{\mathrm{d}t} \\ \dot{E}_{\mathrm{WD,p}} &= 2A(p_1 - p_2)\dot{h} \end{aligned} \tag{3.29}$$

This function is also true if we consider the alpha-hubwards direction, since the sign of $\dot{h}$ changes for positive work and the pressures switch places.

Gravity torque method

This method considers the work done lifting the piston by the rotor. This is simply the weight of the piston pair as a torque, multiplied by the rotor speed:

$$\dot{E}_{\mathrm{WD,g}} = \frac{2h - L_{\mathrm{CT}}}{2}\cos(\theta) m_P g \dot{\theta} \tag{3.30}$$

An example output from the final model is shown in Figure 3.6. It can be seen that the overall gradient, representing the total power of the system, is equal for both state variables. The gap between the two curves is the energy stored in the system itself, as compressed air.

3.4 Structure of the modelling code

3.4.1 Operational modes

The model's behaviour is defined by a series of operational modes. In full, these modes are:

Mode 0 Freefall. Airflow is allowed both into and out of both compression tubes in

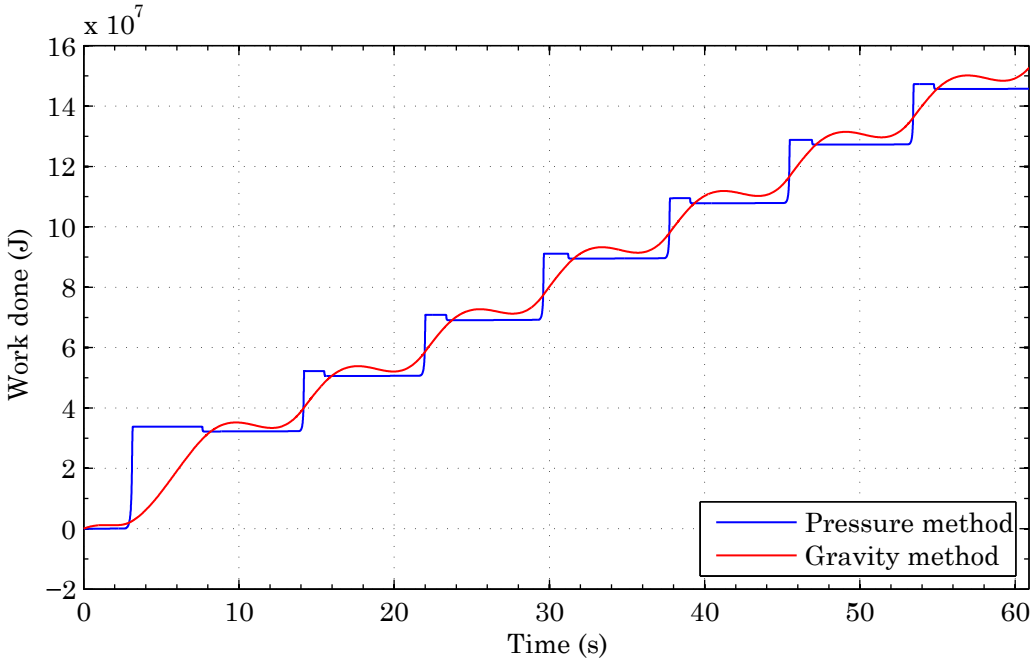


Figure 3.6: The results of two methods to calculate work done

one diametrically opposite pair, to allow the piston pair to build up kinetic energy as it drops. Its position and velocity are closely monitored to calculate both $E_{\text{pot.}}$, the energy currently in the system, and E_{req}, the energy required to compress the remaining air. When $E_{\text{pot.}} = 1.02 \times E_{\text{req}}$, the mode advances.

Mode 1 Compressing. The valve ahead of the piston is closed, allowing the pressure to increase as the piston decelerates. The intake valve behind the piston remains open. Once the pressure in the cylinder reaches the target p_{targ}, the mode advances.

Mode 2 Exhausting. The control algorithms are called for every step taken in this mode, adjusting the constant on the exhaust valve to control how the energy in the system is dissipated. As the compressed air is exhausted to the storage, the piston will slow down further until it stops close to the end of the tube. The mode advances when the piston velocity reaches 0.

Mode 3 Dumping. The piston is locked in position to prevent it moving, and the tiny

remaining amount of high-pressure air in the system is dumped out to the atmosphere. The brakes thus need only to support the weight of the pistons, and not also resist the pressure of this remnant. Once the air pressure has equalised with the atmospheric pressure, the mode advances.

Mode 4 Locked. The system remains locked in position as the tube rotates, until it reaches a set angle at which it will start to kick the piston.

Mode 5 Kicking. The brakes are disengaged and the exhaust valve is opened. The hot compressed air pushes the piston away from the end, giving it the initial energy needed to start falling again. The mode advances once the piston has reached a set velocity.

Mode 6 Kick expansion. The exhaust valve is closed again, and the compressed air in the chamber is allowed to expand as the piston moves away from it. Once the pressure reaches atmospheric pressure, the mode switches back to freefall.

3.4.2 Simulation events

As described in subsection 2.6.2, the numerical methods we will use to solve the simulation work significantly better if the system is non-stiff, i.e. if the derivative functions calculated are smooth and do not change abruptly.

Since our system has very different functions for some state variables depending on the operating mode, we would need to switch between different modes of behaviour based on the mode. However, this would represent a discontinuity in the derivative function, making the problem extremely stiff and potentially preventing any numerical solution.

As a solution, we will use the MATLAB ODE solvers' event location property. A user-defined function is passed the state variable and time on every step evaluation, and calculates a vector of event variables. When one of the event variables changes

sign from one evaluation to the next, MATLAB will locate the exact time and state at that position, and optionally halt the operation of the simulation.

We will set an events function which will detect the conditions for switching between each mode and halt the system when they are detected. Code in the container function will then update a `sys.mode` variable to reflect the new value, concatenate the output results, and then restart the simulation with the halting state set as the initial conditions.

By ensuring that any logical tests within the ODE function consider only those variables which are switched outside the ODE, such as `sys.mode` and `sys.dir` (the direction in which compression is occurring), we can avoid the existence of discontinuities and thus keep the model stiffness as low as possible.

3.4.3 Packed state vector

The way the simulation deals with state vectors has been implemented in a very flexible way. MATLAB's ODE solvers require state vectors as inputs (for both the initial state and the output from the ODE), but this is a difficult format to work with. Newly added state variables need to be at the end of the state vector, not in a logical order, or all previous references need to be updated to reflect the new ordering; also, the difficulty of mapping `y(8)` to m_2 while `y(9)` represents T_1 often leads to indexing errors, which can be very difficult to spot while debugging.

The solution to this issue was to use a function called `SW_pack`, which converts a state vector into a structure with fields corresponding to each state variable, and cell arrays for variables which are different for the different ends of the tube. The respective formats are shown in Figure 3.7 and Figure 3.8.

For instance, `model_pack` maps `y(3)` to `y.hdot`, which clearly marks it as the piston velocity $\dot{h}$, and it converts `y(7)` and `y(8)` to `y.T{1}` and `y.T{2}`, which allow

$$\mathtt{y} = \begin{bmatrix} \theta \\ h \\ \dot{h} \\ m_1 \\ m_2 \\ m_{\text{out}} \\ T_1 \\ T_2 \\ T_{\text{PE},1} \\ T_{\text{PE},2} \\ T_{\text{TE},1} \\ T_{\text{TE},2} \\ WD \\ I \\ t_{\text{delay},1} \\ \vdots \\ t_{\text{delay},4} \\ \theta_{\text{OP},1} \\ \vdots \\ \theta_{\text{OP},12} \end{bmatrix}$$

Figure 3.7: Unpacked state vector

$$\begin{aligned} \mathtt{y.theta} &= \theta \\ \mathtt{y.h} &= h \\ \mathtt{y.hdot} &= \dot{h} \\ \mathtt{y.m}\{1:2\} &= m_1, m_2 \\ \mathtt{y.m\_out} &= m_{\text{out}} \\ \mathtt{y.T}\{1:2\} &= T_1, T_2 \\ \mathtt{y.T\_pe}\{1:2\} &= T_{\text{PE},1}, T_{\text{PE},2} \\ \mathtt{y.T\_te}\{1:2\} &= T_{\text{TE},1}, T_{\text{TE},2} \\ \mathtt{y.wd} &= WD \\ \mathtt{y.I} &= I \\ \mathtt{y.tdel} &= \begin{bmatrix} t_{\text{delay},1} \\ \vdots \\ t_{\text{delay},4} \end{bmatrix} \\ \mathtt{y.opthets} &= \begin{bmatrix} \theta_{\text{OP},1} \\ \vdots \\ \theta_{\text{OP},12} \end{bmatrix} \end{aligned}$$

Figure 3.8: Packed 'state structure'

functions within the ODE to loop twice and easily reference whichever monolithic air temperature T_1 or T_2 is appropriate for the calculation. It also stores all of the θ_{OP} values (the coefficients of the orthogonal polynomials, which are used to calculate the current wall temperature profile) in a single vector, `y.opthets`, which allows vector calculations to be used much more easily. The state vector is thus only represented as a vector when passed to MATLAB's solving functions, and converted to a structure immediately afterwards to simplify code. Adding a new state variable only requires the modification of the `SW_pack` function.

Although this packing and unpacking step adds overhead to the function, it is minimised by the MATLAB compiler, which parallelises the operation during the just-in-time acceleration part of its compiler.

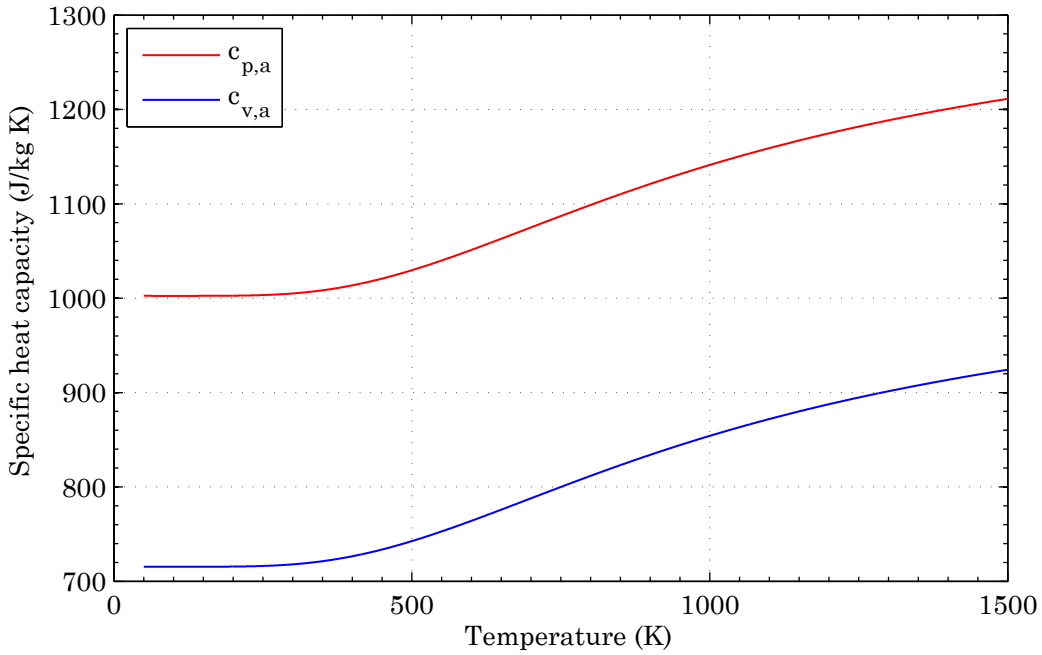


Figure 3.9: Specific heat capacities of air $c_{p,a}$ and $c_{v,a}$, based on Lemmon et al.[86]

3.4.4 Physical properties

For optimum accuracy of the simulation, we will use functions to calculate those physical properties of the system that may vary during operation.

Specific heat capacity of air

To obtain accurate values for the specific heat capacities of air $c_{p,a}$ and $c_{v,a}$ over a range of temperatures, we implement the highly-accurate formulae given by Lemmon et al.[86] in MATLAB. The resulting curves are shown in Figure 3.9.

γ_a, the heat capacity ratio (also known as the adiabatic index), can then be calculated from the two specific heat capacities given by the Lemmon et al.[86] functions:

$$\gamma_a = \frac{c_{p,a}(T)}{c_{v,a}(T)} \tag{3.31}$$

Thermal conductivity of air

For $k_{\text{th,air}}$, the thermal conductivity of the air, we use data from Rogers and Mayhew [85, p.16] imported into MATLAB, as shown in Figure 3.10. Since this will need to be evaluated several times for each timestep, we fit a polynomial to the curve rather than use a MATLAB linear interpolation function such as `interp1`, which would take significantly longer to evaluate. In a test with 10^5 pseudorandom temperature values, the standard linear interpolation function `interp1` took 16.76 s, the faster `interp1q` took 5.09 s, and the polynomial evaluation function `polyval` took 4.08 s, representing a performance boost for the polynomial method of 75.7% over `interp1` and 19.9% over `interp1q`.

A cubic polymonial fitted to the data, as plotted in Figure 3.10, has an error of around 0.5% for the range required.

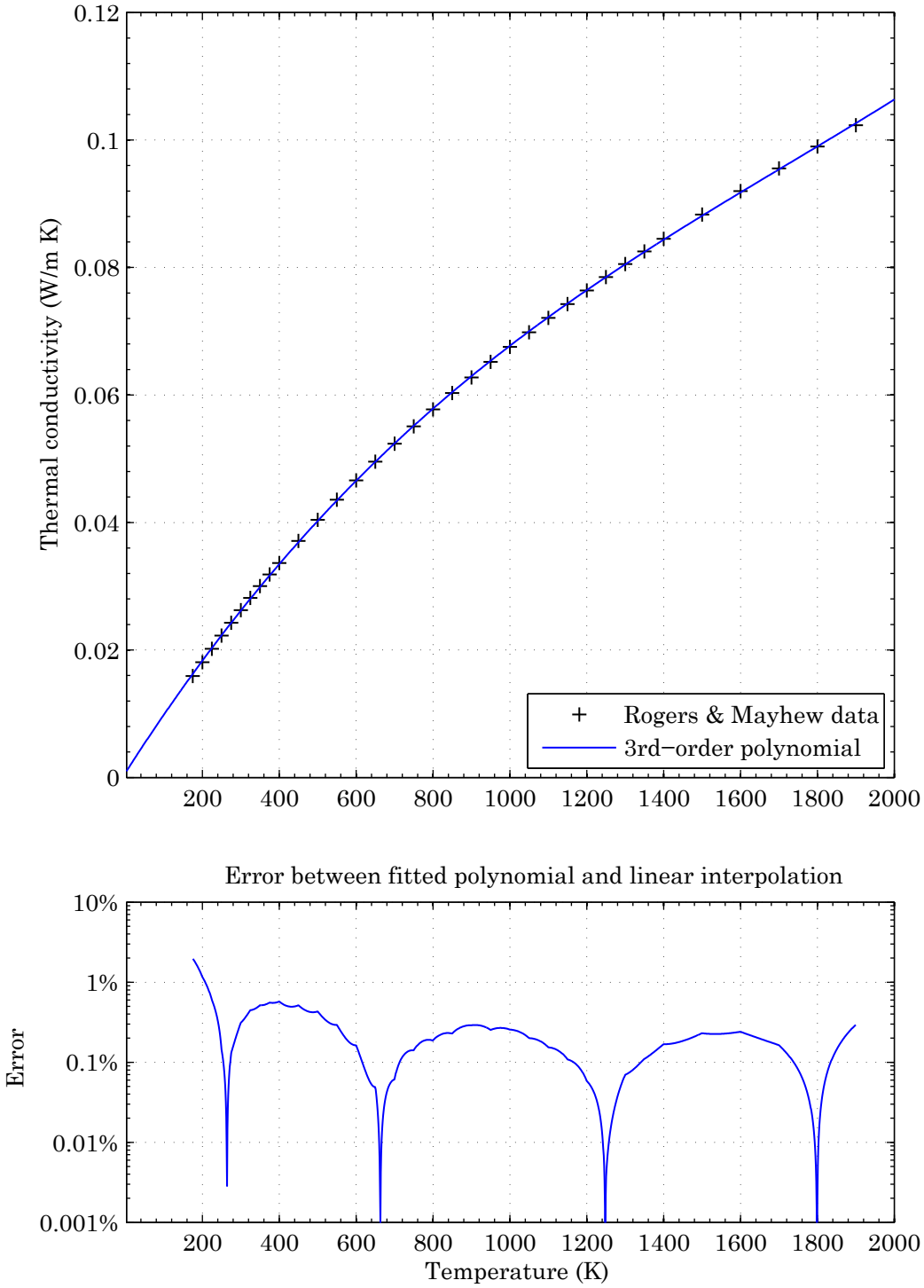


Figure 3.10: Thermal conductivity of air, $k_{\mathrm{th,air}}$

Chapter 4

Energy calculations

Our piston control system will use a constantly-updated estimate of the potential energy in the piston, compared to a calculation of how much energy is required to compress and exhaust the remaining air in the tube, to determine the timing of the start of the compression stage, for optimum system performance. Data from the energy functions is also useful as a performance metric.

4.1 Potential energy in system

First, we calculate the total potential energy of the piston and use it as two variables (in the rimwards and hubwards directions).

4.1.1 Kinetic energy

The total kinetic energy of the piston is calculated simply from its current velocity;

$$E_{\text{pot,kin}} = \frac{1}{2} m_P \dot{h}^2 \tag{4.1}$$

4.1.2 Centrifugal potential energy

The potential energy from the centrifugal acceleration can be found by integrating the centrifugal force over the distance from the piston's current height to the limit of its movement at L_{CT};

$$E_{\mathrm{pot,cent}} = \int mr\dot{\theta}^2 \,\mathrm{d}r \tag{4.2}$$

$$= m\dot{\theta}^2 \int_{h}^{L_{\mathrm{CT}}} \hbar + L_{\mathrm{TE}} \,\mathrm{d}\hbar$$

$$= m\dot{\theta}^2 (L_{\mathrm{CT}} - h) \left(\frac{L_{\mathrm{CT}} + h}{2} + L_{\mathrm{TE}} \right) \tag{4.3}$$

When compressing air in the hubwards direction, this component is unhelpful and will detract from the work done. As a result, it has a different sign depending on which end of the tube is considered.

4.1.3 Diametrically linked pistons

In a system with two diametrically linked pistons, each of height $2L_{\mathrm{PE}}$, the behaviour of the combination in the hubwards and rimwards directions is different;

$$E_{\mathrm{pot,cent}} = m\dot{\theta}^2 \int_{h_0 - \frac{L_{\mathrm{CT}}}{2}}^{\frac{L_{\mathrm{CT}}}{2} - L_{\mathrm{PE}}} r \,\mathrm{d}r$$

$$= m\dot{\theta}^2 \left[\frac{r^2}{2} \right]_{\mathrm{h_0} - \frac{\mathrm{L_{CT}}}{2}}^{\frac{L_{\mathrm{CT}}}{2} - L_{\mathrm{PE}}}$$

$$= \frac{1}{2} m\dot{\theta}^2 \left(h_0 - L_{\mathrm{PE}}\right) \left(L_{\mathrm{CT}} - L_{\mathrm{PE}} - h_0\right) \tag{4.4}$$

This expression (a negative quadratic with roots at $h_0 = L_{\mathrm{PE}}$ and $h_0 = L_{\mathrm{CT}} - L_{\mathrm{PE}}$) can be used when considering either direction for compression.

4.1.4 Gravitational potential energy

GPE needs to consider the distance to the hub for the upper part of the revolution ($0 < \theta < \pi$) and the distance to the rim for the lower part of the revolution ($\pi < \theta < 2\pi$). This is complicated by the change of sign of $sin(\theta)$.

$$E_{\text{pot,gr}} = \begin{cases} mgh\sin\theta & \text{if } \sin\theta > 0, \\ mg(h - L_{\text{CT}})\sin\theta & \text{if } \sin\theta < 0. \end{cases} \tag{4.5}$$

This function does not work, however, since θ will change during the time it takes the piston to reach the end of the tube. A more accurate function assumes linear deceleration, obtaining:

$$\ddot{h} = at + b \tag{4.6}$$

$$\dot{h} = a\frac{t^2}{2} + bt + c \tag{4.7}$$

$$h = a\frac{t^3}{6} + b\frac{t^2}{2} + ct + d \tag{4.8}$$

We let t_{lock} be the time when the piston reaches the end and is locked into place, obtaining the boundary conditions:

$$t = 0,\ h = h_0,\ \dot{h} = \dot{h}_0,\ \theta = \theta_0 \tag{4.9}$$

$$t = t_{\text{lock}},\ h = h_{\text{lock}},\ \dot{h} = 0,\ \theta = \theta_{\text{lock}} \tag{4.10}$$

$$t_{\text{lock}} = \frac{\theta_{\text{lock}} - \theta_0}{\dot{\theta}} \tag{4.11}$$

We use these boundary conditions to obtain the constants:

$$a = \frac{12}{t_{\text{lock}}^3}\left(h_0 - h_{\text{lock}} + \frac{t_{\text{lock}}}{2}\dot{h}_0\right) \tag{4.12}$$

$$b = \frac{6}{t_{\text{lock}}^2}\left(h_{\text{lock}} - h_0 - \frac{2t_{\text{lock}}}{3}\dot{h}_0\right) \tag{4.13}$$

$$c = \dot{h}_0 \tag{4.14}$$

$$d = h_0 \tag{4.15}$$

We now integrate to find the potential energy. The minus sign is required either due to the direction of $\dot{h}$ or the sign of $\sin\theta$, depending on the current radial position.

$$\begin{aligned}
E_{\text{pot,gr}} &= -\int_0^{t_{\text{lock}}} mg\dot{h}\sin\theta\,\mathrm{d}t \\
&= -mg\int_0^{t_{\text{lock}}} \left(\frac{a}{2}t^2 + bt + c\right)\sin\theta\,\mathrm{d}t \\
&= -mg\left[\left(\frac{a}{2}t^2 + bt + c\right)\left(\frac{-\cos\theta}{\dot{\theta}}\right) - (at+b)\left(\frac{-\sin\theta}{\dot{\theta}^2}\right) + a\left(\frac{\cos\theta}{\dot{\theta}^3}\right)\right]_0^{t_{\text{lock}}} \\
&= -mg\left(\begin{array}{r} a\left(t_{\text{lock}}\dfrac{\sin\theta_{\text{lock}}}{\dot{\theta}^2} + \dfrac{\cos\theta_{\text{lock}}}{\dot{\theta}^3} - \dfrac{\cos\theta_0}{\dot{\theta}^3}\right) \\ + b\left(\dfrac{\sin\theta_{\text{lock}} - \sin\theta_0}{\dot{\theta}^2}\right) + c\left(\dfrac{\cos\theta_0}{\dot{\theta}}\right) \end{array}\right)
\end{aligned}$$

Substituting in expressions for a, b and c;

$$\begin{aligned}
E_{\text{pot,gr}} = {} & mg\frac{12}{t_{\text{lock}}^3}\left(\frac{t_{\text{lock}}}{2}\dot{h}_0 + h_0 - h_{\text{lock}}\right)\left(\frac{\cos\theta_0}{\dot{\theta}^3} - t_{\text{lock}}\frac{\sin\theta_{\text{lock}}}{\dot{\theta}^2} - \frac{\cos\theta_{\text{lock}}}{\dot{\theta}^3}\right) \\
& + mg\frac{6}{t_{\text{lock}}^2}\left(h_{\text{lock}} - h_0 - \frac{2t_{\text{lock}}}{3}\dot{h}_0\right)\left(\frac{\sin\theta_0 - \sin\theta_{\text{lock}}}{\dot{\theta}^2}\right) \\
& - mg\dot{h}_0\left(\frac{\cos\theta_0}{\dot{\theta}}\right)
\end{aligned} \tag{4.16}$$

Substituting in $t_{\text{lock}} = \left(\frac{\theta_{\text{lock}}-\theta_0}{\dot{\theta}}\right)$;

$$\begin{aligned} E_{\text{pot,gr}} = mg\frac{12}{(\theta_{\text{lock}}-\theta_0)^3}&\left(\frac{\theta_{\text{lock}}-\theta_0}{2\dot{\theta}}\dot{h}_0 + h_0 - h_{\text{lock}}\right) \\ &\times (\cos\theta_0 - (\theta_{\text{lock}}-\theta_0)\sin\theta_{\text{lock}} - \cos\theta_{\text{lock}}) \\ &+ mg\frac{6}{(\theta_{\text{lock}}-\theta_0)^2}\left(h_{\text{lock}} - h_0 - \frac{2(\theta_{\text{lock}}-\theta_0)}{3\dot{\theta}}\dot{h}_0\right)(\sin\theta_0 - \sin\theta_{\text{lock}}) \\ &- mg\dot{h}_0\left(\frac{\cos\theta_0}{\dot{\theta}}\right) \end{aligned} \tag{4.17}$$

The values of h_{lock} and θ_{lock} vary depending on the direction in which we are considering compression. For a set of two diametrically-connected pistons each of length $2L_{\text{PE}}$, we replace h_0 with $h_0 - \frac{L_{\text{CT}}}{2}$, the position of the centre of gravity of the piston pair; the gravitational potential energy available for compressing air in the rimwards direction (relative to the α piston) is given by setting $h_{\text{lock}} = \frac{L_{\text{CT}}}{2} - L_{\text{PE}}$, while in the hubwards direction we have $h_{\text{lock}} = L_{\text{PE}} - \frac{L_{\text{CT}}}{2}$. The angle of the piston at the end a compression stroke must obviously be the same in either case, so we pick a suitable θ_{lock} for the hubwards stroke and use $\theta_{\text{lock}} + \pi$ for the rimwards stroke.

However, this method both requires us to choose some appropriate value for θ_{lock} and also generates asymptotes in the function where $\theta_0 = \theta_{\text{lock}}$.

An alternative to using a constant is to use a function to estimate t_{lock}, and use that to obtain a θ_{lock} which is always slightly ahead of the current value. If we assume that centrifugal force is absent and that the tube will not rotate significantly during the descent of the piston, we can use the standard equations of motion;

$$\begin{aligned} t = \frac{v - v_0}{a},\ v^2 = v_0^2 + 2as \\ t = \frac{\sqrt{v_0^2 + 2as} - v_0}{a} \end{aligned} \tag{4.18}$$

In the rimwards case, we have;

$$\begin{aligned} v_0 &= \dot{h}_0 \\ s &= (L_{\mathrm{CT}} - h_0 - L_{\mathrm{PE}}) \\ a &= g \sin\theta_0 \\ t &= \frac{\sqrt{\dot{h}_0^2 + 2(g\sin\theta_0)(L_{\mathrm{CT}} - h_0 - L_{\mathrm{PE}})} - \dot{h}_0}{g\sin\theta_0} \end{aligned} \tag{4.19}$$

In the hubwards case, we have;

$$\begin{aligned} v_0 &= -\dot{h}_0 \\ s &= h_0 - L_{\mathrm{PE}} \\ a &= g \sin\theta_0 \\ t &= \frac{\sqrt{\dot{h}_0^2 + 2(g\sin\theta_0)(h_0 - L_{\mathrm{PE}})} + \dot{h}_0}{g\sin\theta_0} \end{aligned} \tag{4.20}$$

We can use these functions to obtain estimates for t_{lock}, which we then substitute into $\theta_{\mathrm{lock}} = \theta_0 + t_{\mathrm{lock}}\dot{\theta}$ to obtain an estimate for θ_{lock}.

Figure 4.1 shows the predicted values of θ_{lock} over a short simulation, with the start of compression and the end of the exhaust both marked with points. Figure 4.2 shows the trajectory of the error in the prediction over the time before the piston stopped for several cycles; the maximum error is 0.35 rad, 2 rad before the end of the stroke, which reduces to around 0.1 rad by the position in which the compression starts. This is more than accurate enough for our purposes.

4.1.5 Total potential energy

We can combine the above three expressions into two variables representing potential energy. $E_{\mathrm{pot},1}$ is the energy available for compressing air rimwards of piston α and

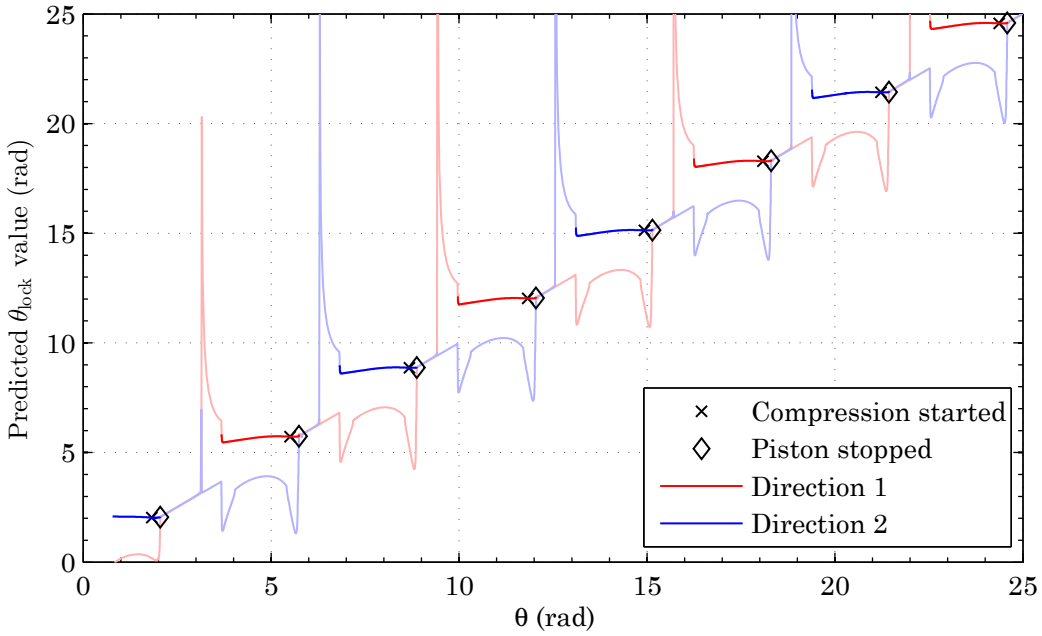


Figure 4.1: Predicted values of θ_{lock}

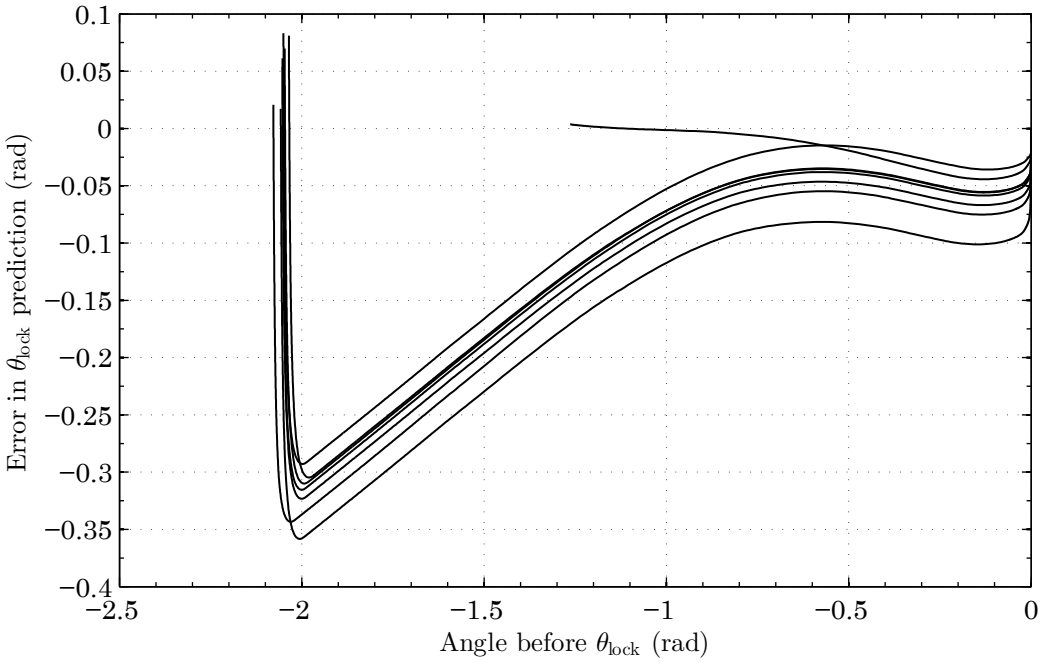


Figure 4.2: Error in θ_{lock} predictions

hubwards of piston β;

$$E_{\text{pot},1} = \frac{1}{2}m\dot{h}^2 + \frac{1}{2}m\dot{\theta}^2 h(L_{\text{CT}} - h) + E_{\text{pot,gr},1} \tag{4.21}$$

$E_{\text{pot},2}$ is the energy available for compressing air hubwards of piston α and rimwards of piston β;

$$E_{\text{pot},2} = \frac{1}{2}m\dot{h}^2 + \frac{1}{2}m\dot{\theta}^2 h(L_{\text{CT}} - h) + E_{\text{pot,gr},2} \tag{4.22}$$

We reduce computation time by calculating only one of these at a time, depending on the current direction of movement of the piston pair.

4.2 Energy required

The model must continuously calculate the energy required to compress and exhaust the air currently in the tube. Before the valve is closed, this air will be at atmospheric pressure.

4.2.1 Energy to compress air

The air must be compressed from the current pressure in the tube up to the target air pressure of the HP manifold. The energy required to do this is given by;

$$E_{\text{comp}} = \int p \, \mathrm{d}V$$

Since we know that $V = hA$ and $p = kV^{-\gamma}$ [41, p.180], we can substitute;

$$E_{\text{comp}} = kA^{1-\gamma} \int_{h_1}^{h_2} h^{-\gamma} \, \mathrm{d}h$$

We can find the limiting piston positions by considering the ratio between the target pressure p_{targ} and the current pressure, p_1 or p_2 depending on direction. The current length of the volume to be compressed may be $(L_{\text{CT}} - h)$ for alpha-rimwards compression or h for alpha-hubwards compression; we will denote the current pressure as p_0 and the distance from the current position to the end of the stroke as h_0 to obtain a general expression.

$$\begin{aligned} E_{\text{comp}} &= kA^{1-\gamma} \int_{h_0\left(\frac{p_0}{p_{\text{targ}}}\right)^{\frac{1}{\gamma}}}^{h_0} h^{-\gamma}\,\mathrm{d}h \\ &= (p_0 A^{\gamma} h^{\gamma})\,\frac{A^{1-\gamma}}{1-\gamma}\left[h_0^{1-\gamma} - h_0^{1-\gamma}\left(\frac{p_0}{p_{\text{targ}}}\right)^{\frac{1-\gamma}{\gamma}}\right] \\ &= \frac{p_0 h_0 A}{1-\gamma}\left(1 - \left(\frac{p_0}{p_{\text{targ}}}\right)^{\frac{1-\gamma}{\gamma}}\right) \end{aligned} \tag{4.23}$$

4.2.2 Energy to exhaust compressed air

The compressed volume is given by;

$$V_{\text{exh}} = h_0 A \sqrt[\gamma]{\frac{p_0}{p_{\text{targ}}}} \tag{4.24}$$

The work done exhausting this volume at constant pressure p_{targ} is given by;

$$\begin{aligned} E_{\text{exh}} &= \int p\,\mathrm{d}V \\ &= \int_{V_{\text{exh}}}^{0} p_{\text{targ}}\,\mathrm{d}V \\ &= h_0 A p_{\text{targ}} \sqrt[\gamma]{\frac{p_0}{p_{\text{targ}}}} \end{aligned} \tag{4.25}$$

4.2.3 Work done by atmosphere

There will be a negative contribution to the energy required, from the work done by atmospheric pressure behind the piston. We find;

$$\begin{aligned} E_{\text{atmos}} &= \int p \, \mathrm{d}V \\ &= \int_0^{h_0 A} p_{\text{atm}} \, \mathrm{d}V \\ E_{\text{atmos}} &= -h_0 A p_{\text{atm}} \end{aligned} \tag{4.26}$$

4.2.4 Energy to overcome friction

Our model includes only a constant Coulomb friction force, F_μ, so the energy required is simply;

$$E_{\text{fric}} = h_0 F_\mu \tag{4.27}$$

4.2.5 Total energy required

We can now write an expression for the total energy required to compress and exhaust the air in the tube, in terms of the tube area, the starting and target pressures, and the distance the piston needs to move;

$$E_{\text{req}} = h_0 \left(\frac{p_0 A}{1-\gamma} \left(1 - \left(\frac{p_0}{p_{\text{targ}}} \right)^{\frac{1-\gamma}{\gamma}} \right) + A p_{\text{targ}} \sqrt[\gamma]{\frac{p_0}{p_{\text{targ}}}} + F_\mu - A p_{\text{atm}} \right) \tag{4.28}$$

This is built into the function `SW_control_energy`, which is called by the events function `SW_events` to be calculated for each timestep and compared to the potential energies in the system to determine the correct point at which to switch modes.

4.3 Conclusions

This chapter outlined both a set of functions and a process by which the various potential energies in the system may be calculated. Their implementation into the model results in Figure 4.3, which shows how the various components change as the piston moves through the stroke. The functions are of use not only in the decision of when the compression stage should start, but also for the calculation of work done, which provides a metric for the system performance, which will be discussed further in chapter 8.

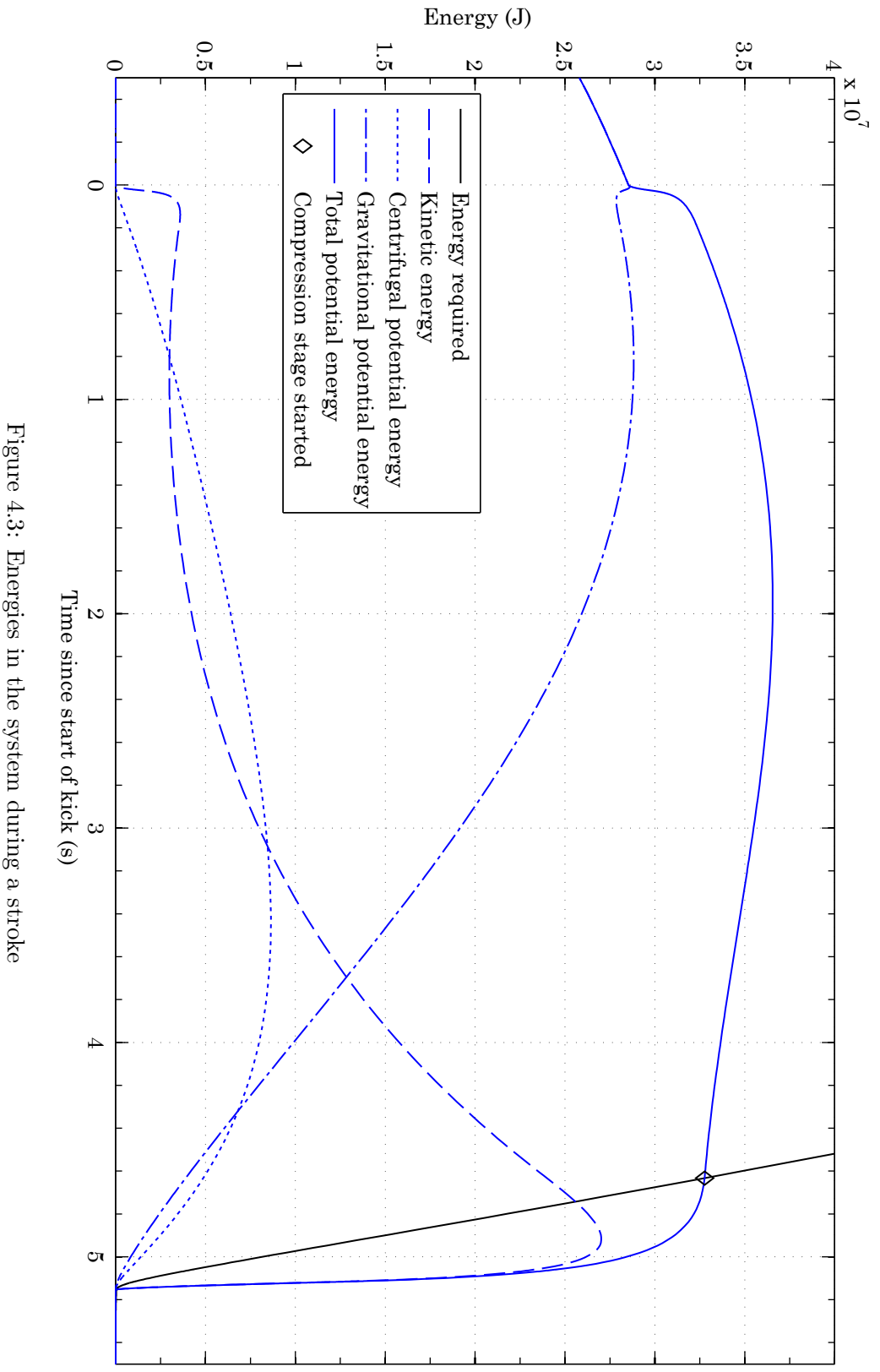


Figure 4.3: Energies in the system during a stroke

Chapter 5

Thermal modelling

As described in the previous chapters, the model assumes the compression to be fully adiabatic, involving no heat transfer to or from the compression cylinder and the piston. To increase the accuracy, this chapter will outline the process and results of adding a thermal model of the compression tube to the overall model.

Since the air compression occurs at the ends of the tube, and the particularly high temperatures associated with the target pressure are also at the extreme ends, we will design a thermal model which possesses great accuracy at the ends.

5.1 Wall temperatures

We now introduce new properties Q, the rate of thermal energy transfer per unit area; $(\rho_{\mathrm{CT}} c_{p,\mathrm{CT}})$, the thermal mass of the wall per unit volume; $k_{\mathrm{th,CT}}$, the internal thermal conductivity of the tube wall per unit area; U, a heat transfer coefficient; r, the radius of the cylinder; r_{t}, the radial thickness of the wall; and z, a spatial coordinate for the wall. We consider the heat transfer acting over a vertical section of cylinder Δz and assume that $r >> r_{\mathrm{t}}$.

5.1.1 Wall temperature ODE

We begin by considering the ordinary differential equation which governs the wall temperature. This is made up of two components: one due to heat transfer within the wall, and one due to heat transfer to or from the air mass. We assume the cylinder is thermally isolated from its surroundings.

$$\frac{\partial T_{\mathrm{CT}}}{\partial t}(z,t) = \left.\frac{\partial T_{\mathrm{CT}}}{\partial t}\right|_{\mathrm{int}} + \left.\frac{\partial T_{\mathrm{CT}}}{\partial t}\right|_{\mathrm{surf}} \tag{5.1}$$

Conduction internal to the wall

If we consider a thin element of the cylinder, we find that the thermal transfer along the cylinder is simply the second-order partial derivative of the temperature with respect to height, multiplied by the rate of heat flow and divided by the thermal mass of the element.

$$\begin{aligned}\left.\frac{\partial T_{\mathrm{CT}}}{\partial t}\right|_{\mathrm{int}} &= \frac{k_{\mathrm{th,CT}}\ (2\pi\, r\, r_{\mathrm{t}})\ \frac{\delta^2 T_{\mathrm{CT}}}{\delta z^2}\ \Delta z}{2\pi\, r\, r_{\mathrm{t}}\, \Delta z\, (\rho c_p)_{\mathrm{CT}}} \\ \left.\frac{\partial T_{\mathrm{CT}}}{\partial t}\right|_{\mathrm{int}} &= \frac{k_{\mathrm{th,CT}}}{(\rho c_p)_{\mathrm{CT}}}\ \frac{\delta^2 T_{\mathrm{CT}}}{\delta z^2}\end{aligned} \tag{5.2}$$

This will allow us to model the diffusion of heat along the length of the cylinder.

Conduction at wall surface

Here, we consider the heat transfer acting over an area equal to the circumference multiplied by Δh;

$$\begin{aligned}\left.\frac{\partial T_{\mathrm{CT}}}{\partial t}\right|_{\mathrm{surf}} &= \frac{Q\ (2\pi\, r\, \Delta z)}{2\pi\, r\, \Delta z\, (\rho c_p)_{\mathrm{CT}}} \\ \left.\frac{\partial T_{\mathrm{CT}}}{\partial t}\right|_{\mathrm{surf}} &= \frac{Q}{(\rho c_p)_{\mathrm{CT}}}\end{aligned} \tag{5.3}$$

The rate of thermal energy transfer per unit area of wall inner surface, Q, is given by [41, p.508];

$$Q = U \times \Delta T \tag{5.4}$$

where we will take the heat transfer coefficient per length of tube U to be given by considering the resistance of nominal thicknesses of wall, t_{CT}, and air, t_{air}.

$$\frac{1}{U} = \frac{t_{\mathrm{CT}}}{k_{\mathrm{th,CT}}} + \frac{t_{\mathrm{air}}}{k_{\mathrm{th,air}}} \tag{5.5}$$

We can simply take $t_{\mathrm{CT}} = \frac{r_{\mathrm{t}}}{2}$, half the radial thickness of the wall. $k_{th,\mathrm{air}}$ is obtained by interpolation of data from Rogers and Mayhew [85, p.16] as shown in section 3.4.4. We assume the thickness of the air boundary layer to be $t_{\mathrm{air}} = 0.003$ mm, as detailed in appendix D.

Total ODE

Combining these two expressions, we obtain;

$$\frac{\partial T_{\mathrm{CT}}}{\partial t}(z,t) = \frac{k_{\mathrm{th,CT}}}{\rho c_p}\frac{\partial^2 T_{\mathrm{CT}}}{\partial z^2} + \frac{Q}{\rho c_p}\left(T - T_{\mathrm{CT}}\right) \tag{5.6}$$

In order to simplify the process of finding $\frac{\partial^2 T_{\mathrm{CT}}}{\partial z^2}$, we will model temperature in the wall as a linear combination of orthogonal polynomials.

5.2 Orthogonal Polynomials

We first define a weighted inner product, for two polynomials $f(z)$ and $g(z)$:

$$\langle f, g\rangle = \int_a^b f \cdot g \cdot w(z)\,\mathrm{d}z \tag{5.7}$$

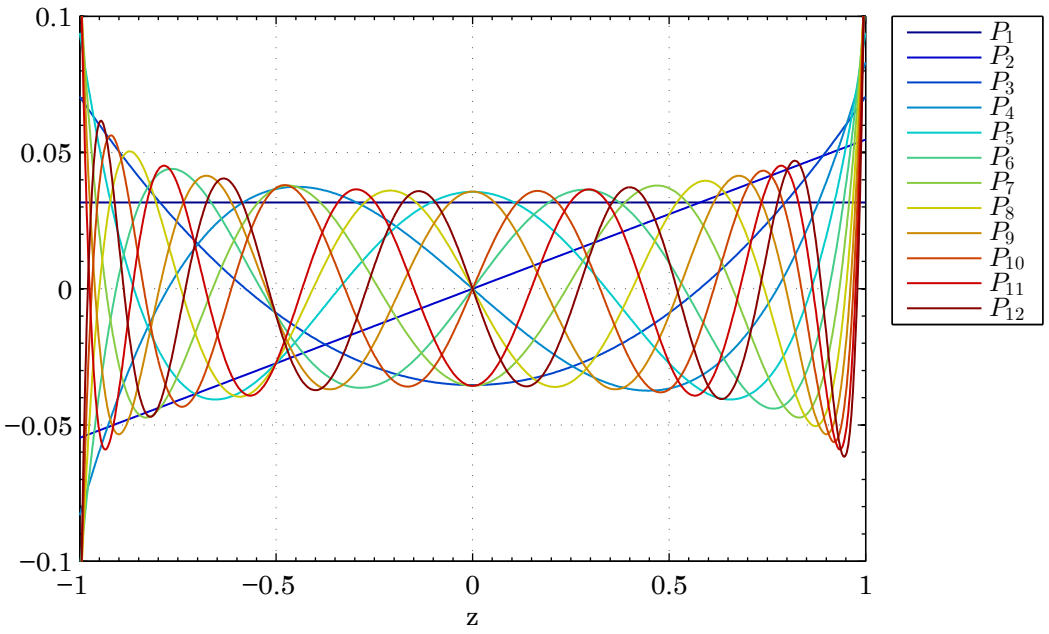


Figure 5.1: Legendre set of orthogonal polynomials

where $w(z)$ is a weighting function which is positive for all z values, and a and b are the limits of the region of interest. If f and g are 'orthogonal', this means that $\langle f, g \rangle = 0$. If we take $w(z) = 1$, $a = -1$ and $b = 1$, we obtain a set of polynomials referred to as the Legendre polynomials, shown in Figure 5.1.

5.2.1 Gram-Schmidt orthonormalisation

By convention, our first polynomial is a constant. We let the second polynomial take the form of a straight line, and find it using the definition of orthogonality;

$$P_0 = 1 \tag{5.8}$$

$$P_1 = z + c$$

$$\langle P_0, P_1 \rangle = 0$$

$$\int_0^{L_{\mathrm{CT}}} 1 \times (z + c) \times \left(1 - \frac{z}{L_{\mathrm{CT}}}\right) \mathrm{d}z = 0$$

$$c = -\frac{L_{\mathrm{CT}}}{3} \tag{5.9}$$

We can now find as many orthogonal polynomials as required using a recurrence relation derived from the Gram-Schmidt orthonormalisation process (GSONP);

$$P_{\mathrm{i}+1} = \left[z - \frac{\langle z, P_i, P_i \rangle}{\langle P_i, P_i \rangle}\right] P_i - \left[\frac{\langle P_i, P_i \rangle}{\langle P_{\mathrm{i}-1}, P_{\mathrm{i}-1} \rangle}\right] P_{\mathrm{i}-1} \tag{5.10}$$

This delivers a series of polynomials, each of the order of its index (i.e. P_4 is a 4th-order polynomial).

5.2.2 Analytical method

Initially, a method of generating the polynomials using analytical methods was tried. Here, we carry out operations on the polynomial coefficients to multiply and integrate the polynomials. This method led to a build up of significant orthonormalisation errors after a relatively low number of polynomials were generated, due to the numerical instability of the algorithm.

5.2.3 Numerical method

An alternative technique was adopted whereby the polynomials are represented as vectors of coordinates, m elements long. First, we define a basis of z-values:

$$\mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ \vdots \\ z_m \end{bmatrix} = \begin{bmatrix} a \\ a + \frac{b-a}{m} \\ a + 2\frac{b-a}{m} \\ \vdots \\ b \end{bmatrix} \tag{5.11}$$

We similarly define the weight function $w(z)$ as a vector of values $\mathbf{w}$ from w_1 to w_m, and the ith polynomial P_i as a vector $\mathbf{P}_i$ with terms $P_{i,1}$ up to $P_{i,m}$. The inner product of two vectors is now:

$$\langle \mathbf{P}_1, \mathbf{P}_2 \rangle = \sum_{i=1}^{m} P_{1,i} \cdot P_{2,i} \cdot w_i \tag{5.12}$$

We can now carry out GSONP as before. This method results in a significant increase in accuracy when generating larger numbers of polynomials, as shown in Figure 5.2.

5.2.4 Derivatives

Recalling that we require a function for $\frac{\partial^2 T_{\mathrm{CT}}}{\partial z^2}$ for Equation 5.6, we will obtain the derivatives to the vectors using fitted polynomials. As we know the order of each of the m polynomials and have n points to fit, this will be very accurate provided $n \gg m$. We will take $n = 1000$ and $m = 12$ to provide adequate detail.

Obtaining derivatives is then simply a case of analytically differentiating the polynomials by operations on their coefficients.

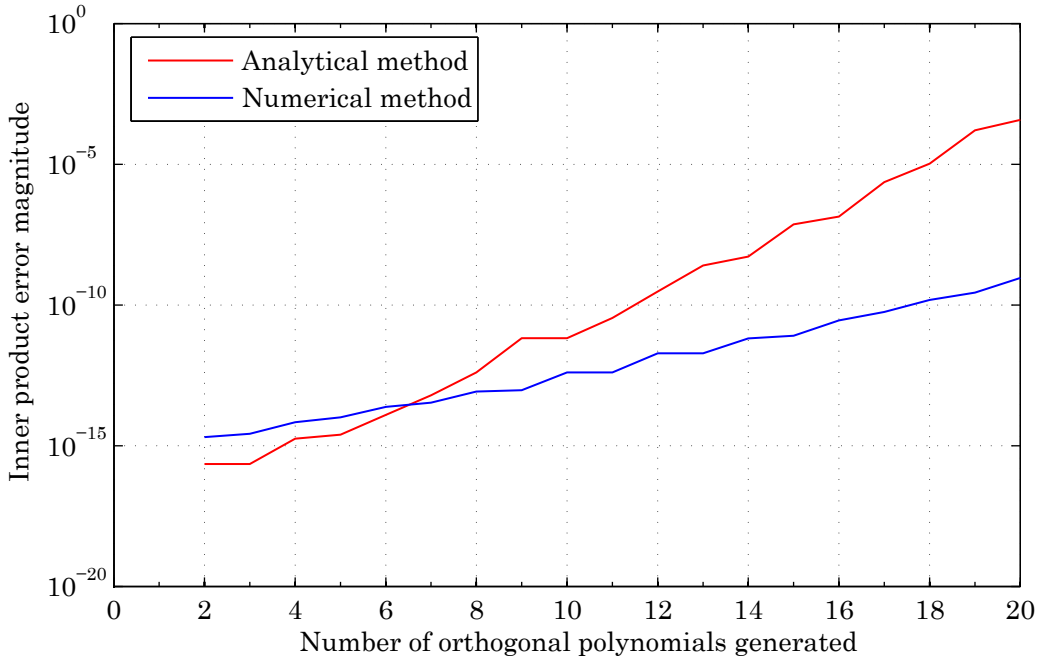


Figure 5.2: Accuracy of two GSONP methods at generating the Legendre polynomials

5.2.5 Weighting

In our case, we will weight the polynomials to be more accurate closer to the ends of the tube. This is better for this system because the majority of heat flow will occur into and out of the ends of the tube. This is done in two ways.

Firstly, we will set the weighting curve to be five times larger at the tube ends than at the middle, using a 4th-order polynomial with centring and scaling. We take:

$$w(z) = 4\left(\frac{2z}{L_{\mathrm{CT}}} - 1\right)^4 + 1 \tag{5.13}$$

$$a = 0$$

$$b = L_{\mathrm{CT}}$$

This weighting function is shown in Figure 5.3.

Secondly, we will generate more points at the ends than in the middle, by interpolating

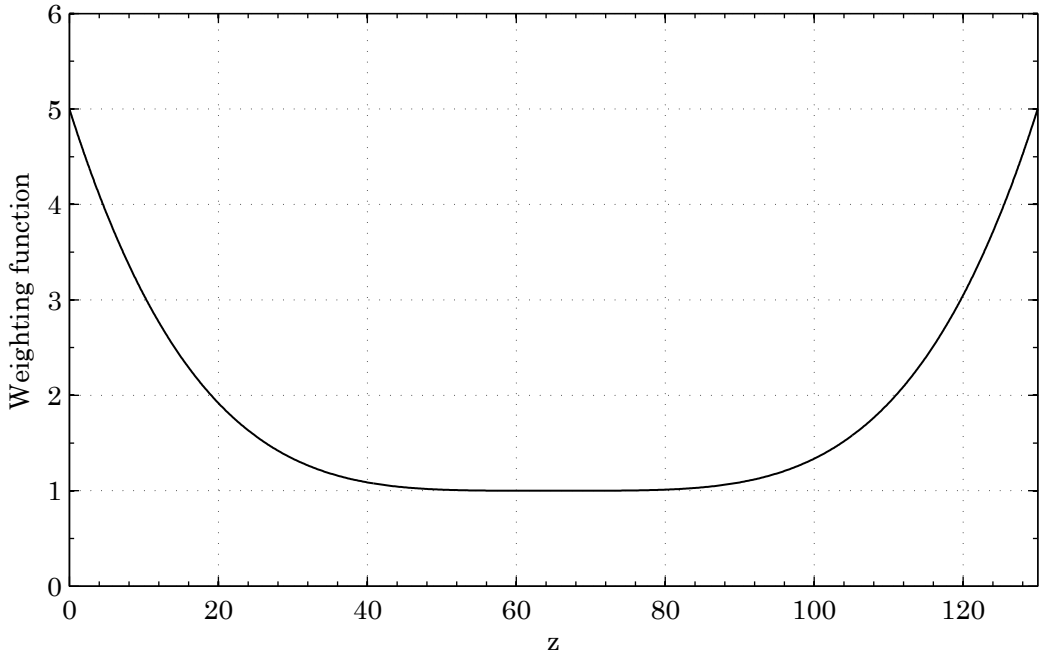


Figure 5.3: Weighting curve for inner products

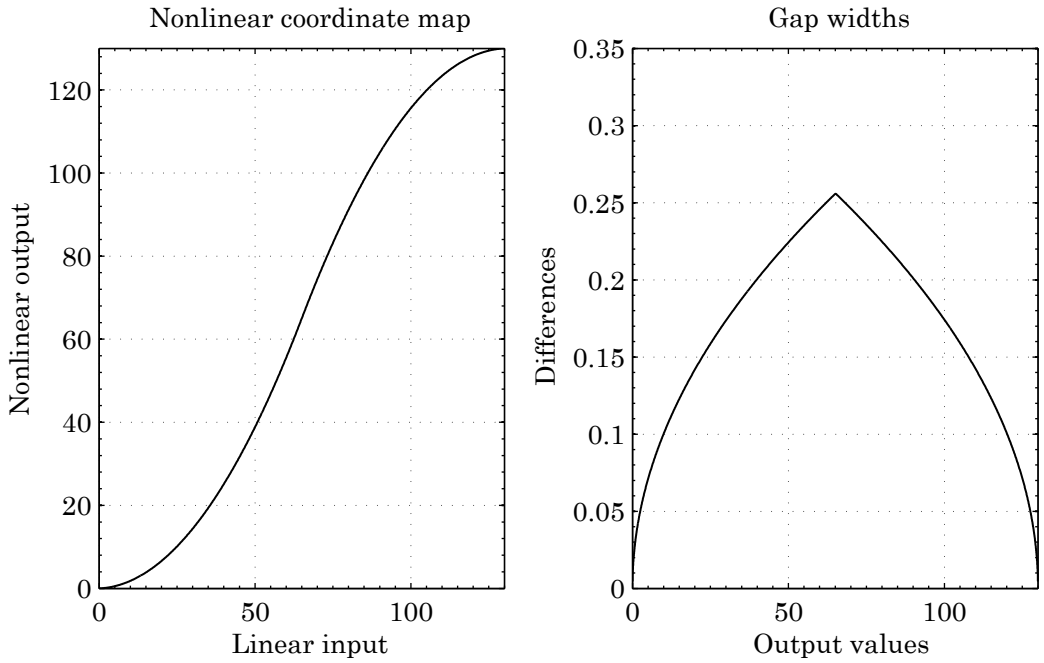


Figure 5.4: Generating points with nonlinear spacing

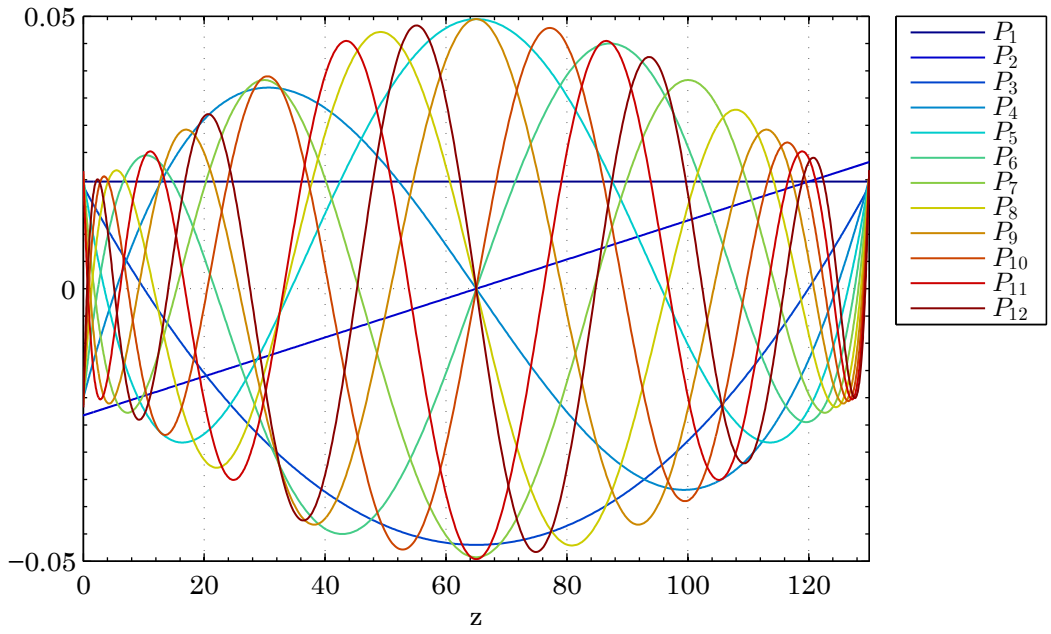


Figure 5.5: Set of orthogonal polynomials, weighted towards the ends of the compression tube

a linearly-spaced set of points on a quadratic curve, as shown in Figure 5.4. Since we will generate the polynomials in the form of vectors of coordinates, the increased point density at the ends will weight the inner product function towards the ends. Figure 5.5 shows a set of orthogonal polynomials generated using the weighting method outlined above.

5.2.6 Projection matrices

The orthogonal basis of polynomials described above allows us to calculate $\frac{\partial T_{\mathrm{CT}}}{\partial t}(z,t)$ at any point along the wall. In order to 'close the circle' and formulate a usable ODE to model the wall temperatures, we need an expression which dictates how the coefficients of those polynomials change with respect to time.

First, we give each polynomial P_i a state-variable coefficient θ_i, so the wall tempera-

ture at a point z is given by;

$$T_w(h) = \sum_{i=0}^{n} \theta_i P_i(z) \tag{5.14}$$

The problem is now to calculate the derivative, $\dot{\theta}$, of this vector of n orthogonal polynomial coefficients. We can do this by evaluating $\dot{T}$ at m different values of z (calling this vector $\dot{\mathbf{T}}_{\text{full}}$), and using a projection matrix $\mathbf{A}$ to convert to the (far smaller) vector $\dot{\theta}$. $\mathbf{A}$ can be constructed quite simply; each row corresponds to one of the m values of z, and each column corresponds to one of the n orthogonal polynomials P_1 to P_n.

$$\mathbf{A} \equiv \begin{bmatrix} P_1(z_1) & P_2(z_1) & \dots & P_n(z_1) \\ P_1(z_2) & P_2(z_2) & \dots & P_n(z_2) \\ \vdots & \vdots & \ddots & \vdots \\ P_1(z_m) & P_2(z_m) & \dots & P_n(z_m) \end{bmatrix} \tag{5.15}$$

The projection is thus:

$$\dot{\mathbf{T}}_{\text{full}} \simeq \mathbf{A}\,\dot{\theta} \tag{5.16}$$

Using the Moore-Penrose pseudoinverse for non-square matrices:

$$\left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T\,\dot{\mathbf{T}}_{\text{full}} \simeq \dot{\theta} \tag{5.17}$$

$\mathbf{A}$ can be calculated before the simulation is run, along with $\left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T$, so the computational load during the simulation is smaller.

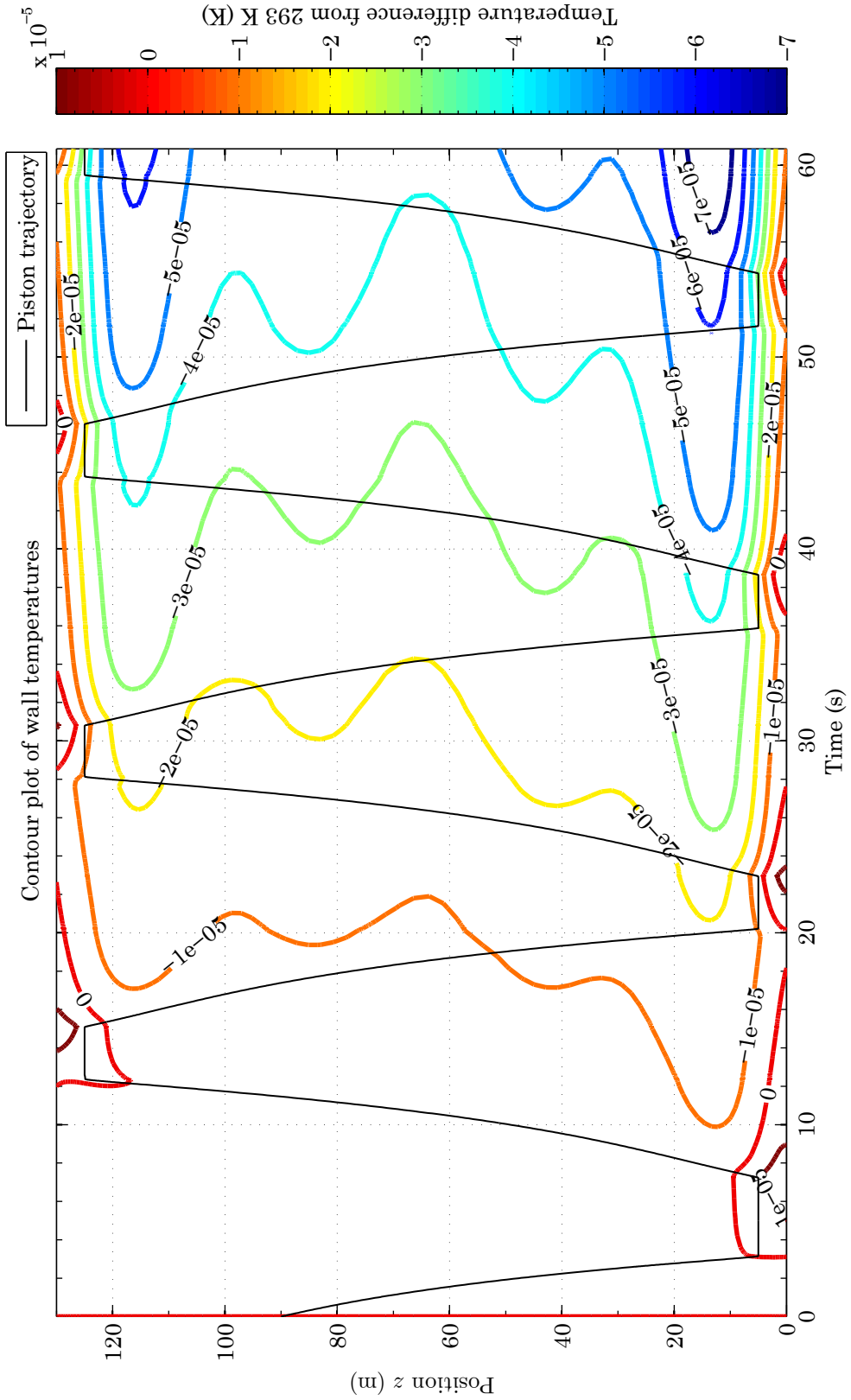


Figure 5.6: Wall temperature contours, starting from uniform 293 K

5.3 Implementation

Implementing this in the model, the output can be viewed as a 'temperature surface'; the total sum across all the orthogonal polynomials, plotted against position and time. The results of running the model for 1 minute of simulated time, with the wall temperature initialised to a uniform 293 K, are shown as a contour plot in Figure 5.6. It can be seen that, firstly, the general pattern is that the wall temperature reduces in the centre, while the tube ends remain the same temperature or slightly warmer. The visible cycle of the piston trajectory $h(t)$ (shown as a black line) with rises in T_{CT} at the tube ends demonstrates the heat flow into the wall; at the very end, a net heat flow into the wall due to the adiabatic compression heat, while the rest of the cyclinder is cooled by the expanding kicks, particularly around 10 m from the end.

It is clear that the distribution of wall temperatures will reach a steady state after some initial duration.

5.4 Finding steady-state wall temperatures

We use singular value decomposition (SVD) to look at the matrix of orthogonal polynomial coefficients output from a long-term simulation, $\mathbf{\Theta}$. The number of non-zero elements singular values will tell us how many functions are linearly independent.

Firstly, we make the assumption that each orthogonal polynomial coefficient $\theta_i(t)$ is a linear combination of decaying exponential terms, as:

$$\theta_1(t) = a_1 + b_1 e^{-\tau_b t} + c_1 e^{-\tau_c t} + d_1 e^{-\tau_d t} + \dots$$
$$\theta_2(t) = a_2 + b_2 e^{-\tau_b t} + c_2 e^{-\tau_c t} + d_2 e^{-\tau_d t} + \dots$$

If we make an initial guess at a set of values for τ, the decay constants, we can use a

relatively simple Moore-Penrose pseudo-inverse to calculate the coefficients for each orthogonal polynomial;

$$\begin{bmatrix} \theta_1(t_1) & \theta_2(t_1) & \dots & \theta_n(t_1) \\ \theta_1(t_2) & \theta_2(t_2) & \dots & \theta_n(t_2) \\ \theta_1(t_3) & \theta_2(t_3) & \dots & \theta_n(t_3) \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix} = \begin{bmatrix} 1 & e^{-\tau_b t_1} & e^{-\tau_c t_1} & e^{-\tau_d t_1} \\ 1 & e^{-\tau_b t_2} & e^{-\tau_c t_2} & e^{-\tau_d t_2} \\ 1 & e^{-\tau_b t_3} & e^{-\tau_c t_3} & e^{-\tau_d t_3} \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} a_1 & a_2 & \dots & a_n \\ b_1 & b_2 & \dots & b_n \\ c_1 & c_2 & \dots & c_n \\ d_1 & d_2 & \dots & d_n \end{bmatrix} \tag{5.18}$$

$$\mathbf{\Theta} = \mathbf{FA}$$

$$\mathbf{A} = (\mathbf{F}^T\mathbf{F})^{-1}\mathbf{F}^T\mathbf{\Theta} \tag{5.19}$$

We can then construct a matrix of 'projected' θ values - a version of Θ based on the $\mathbf{A}$ we have calculated.

$$\mathbf{\Theta}_{\text{proj}} = \mathbf{FA} \tag{5.20}$$

From the differences between $\mathbf{\Theta}$ and $\mathbf{\Theta}_{\text{proj}}$ we obtain the sum of squares of differences (SSQ), which we can use as a measure of how appropriate the τ values we picked were.

$$\text{SSQ} = \sum_i \sum_j \left(\mathbf{\Theta}_{\text{proj}_{ij}} - \mathbf{\Theta}_{ij}\right)^2 \tag{5.21}$$

However, this still requires us to manually tune the τ values.

5.4.1 Newton-Raphson optimisation process

We know that we are looking for a minimum value of the SSQ, where:

$$\frac{\partial \text{SSQ}}{\partial \tau_i} = 0 \; \forall \, i \tag{5.22}$$

To find this minimum, we form up two arrays of derivatives. Firstly, a vector of first-derivatives;

$$\mathbf{d} = \begin{bmatrix} \frac{\partial \mathrm{SSQ}}{\partial \tau_1} \\ \frac{\partial \mathrm{SSQ}}{\partial \tau_2} \\ \vdots \\ \frac{\partial \mathrm{SSQ}}{\partial \tau_n} \end{bmatrix} \tag{5.23}$$

Although it is possible in this case to find the derivatives analytically, it is simpler to use a simple numerical derivative. Given a function which calculates the SSQ from a given set of τ values, `getSSQ`, we have:

$$\frac{\partial \mathrm{SSQ}}{\partial \tau_i} = \frac{1}{2\epsilon_i} \left(\texttt{getSSQ}(\tau_{i+}) - \texttt{getSSQ}(\tau_{i-}) \right) \tag{5.24}$$

where τ_{i+} is equal to τ with a small perturbation ϵ_i added to the ith value, and τ_{i-} has the perturbation subtracted. ϵ_i is normally found as $0.001 \times \tau_i$.

Next, we form the Hessian matrix of second-order partial derivatives, $\mathbf{H}$, which is given by:

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 \mathrm{SSQ}}{\partial \tau_1 \partial \tau_1} & \frac{\partial^2 \mathrm{SSQ}}{\partial \tau_1 \partial \tau_2} & \cdots & \frac{\partial^2 \mathrm{SSQ}}{\partial \tau_1 \partial \tau_n} \\ \frac{\partial^2 \mathrm{SSQ}}{\partial \tau_2 \partial \tau_1} & \frac{\partial^2 \mathrm{SSQ}}{\partial \tau_2 \partial \tau_2} & \cdots & \frac{\partial^2 \mathrm{SSQ}}{\partial \tau_2 \partial \tau_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 \mathrm{SSQ}}{\partial \tau_n \partial \tau_1} & \frac{\partial^2 \mathrm{SSQ}}{\partial \tau_n \partial \tau_2} & \cdots & \frac{\partial^2 \mathrm{SSQ}}{\partial \tau_n \partial \tau_n} \end{bmatrix} \tag{5.25}$$

To find these second-order derivatives, we use the same simple method outlined above. We obtain perturbed τ vectors τ_{i-} & τ_{i+} and τ_{j-} & τ_{j+} (with perturbations to the elements τ_i and τ_j respectively), and we combine them into τ vectors such as $\tau_{i-,j+}$, in which the ith element has been perturbed downwards and the jth element

upwards. Then, we calculate the derivatives twice:

$$\begin{aligned}\left|\frac{\partial \mathrm{SSQ}}{\partial \tau_i}\right|_{j+} &= \frac{1}{2\epsilon_i}\left(\texttt{getSSQ}(\tau_{i+,j+}) - \texttt{getSSQ}(\tau_{i-,j+})\right)\\ \left|\frac{\partial \mathrm{SSQ}}{\partial \tau_i}\right|_{j-} &= \frac{1}{2\epsilon_i}\left(\texttt{getSSQ}(\tau_{i+,j-}) - \texttt{getSSQ}(\tau_{i-,j-})\right)\end{aligned} \tag{5.26}$$

$$\begin{aligned}\frac{\partial^2 \mathrm{SSQ}}{\partial \tau_i \partial \tau_j} &= \frac{1}{2\epsilon_j}\left(\left|\frac{\partial \mathrm{SSQ}}{\partial \tau_i}\right|_{j+} - \left|\frac{\partial \mathrm{SSQ}}{\partial \tau_i}\right|_{j-}\right)\\ &= \frac{1}{4\epsilon_i\epsilon_j}\begin{pmatrix}\texttt{getSSQ}(\tau_{i+,j+}) - \texttt{getSSQ}(\tau_{i-,j+})\\ \qquad - \texttt{getSSQ}(\tau_{i+,j-}) + \texttt{getSSQ}(\tau_{i-,j-})\end{pmatrix}\end{aligned} \tag{5.27}$$

The symmetry of $\mathbf{H}$ (from Clairaut's theorum, which states that successive partial derivations are commutative) reduces the number of calculations required to obtain $\mathbf{H}$.

Once we have $\mathbf{d}$ and $\mathbf{H}$, we can calculate an updated τ, τ', according to standard Newton-Raphson:

$$\tau' = \tau - \mathbf{H}^{-1}\mathbf{d} \tag{5.28}$$

Modification to improve stability

We will further modify the Newton-Raphson optimisation algorithm to improve its stability, by reducing the size of the step between adjacent τ values. A crude approach would be:

$$\tau' = \tau - (1-\zeta)\mathbf{H}^{-1}\mathbf{d} \tag{5.29}$$

where $\zeta = 0.95$ initially, and is further reduced as $\zeta' = 0.95\zeta$ on each step.

However, on some steps, it might be desirable to reduce ζ by more or less than 0.95,

so we will adopt a different method. Firstly, we compute a new value for τ, with a reduction factor of ξ:

$$\tau' = \tau - \xi \mathbf{H}^{-1}\mathbf{d} \tag{5.30}$$

Then, we use the new τ value τ' to recalculate the Hessian matrix, $\mathbf{H}'$. This is multiplied by the original correction:

$$\mathbf{d}' = \mathbf{H}' \left(\mathbf{H}^{-1}\mathbf{d}\right) \tag{5.31}$$

Finally, we find the greatest difference between the derivative vectors $\mathbf{d}$ and $\mathbf{d}'$ when scaled relative to the corresponding entry in $\mathbf{d}$;

$$\Delta = \texttt{max}\left(\frac{\mathbf{d}'_i - \mathbf{d}_i}{\mathbf{d}_i}\right) \tag{5.32}$$

If $\Delta > 0.01$, then we take $\xi' = \frac{\xi}{2}$ and start over. Once we have a suitably small Δ, we accept that step and move on.

5.4.2 Final steady state

The resulting constant parts of the coefficient matrix (the values a_1, a_2 etc.) are the values of θ which define the final steady-state wall temperature. A simulation which ran for 48 hours of simulated time found the steady-state temperature profile to be

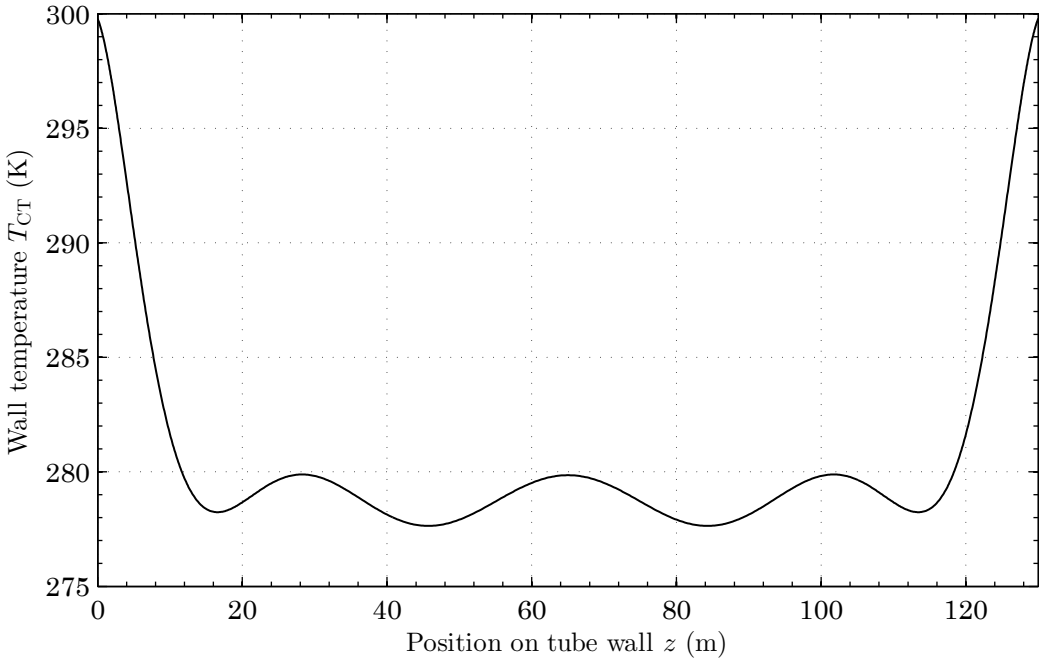


Figure 5.7: Steady-state T_{wall} profile

given by:

$$\theta_i = \begin{cases} 1.469 \times 10^4 & \text{for } i = 1, \\ 375.1 & \text{for } i = 3, \\ 214.1 & \text{for } i = 5, \\ 59.27 & \text{for } i = 7, \\ -25.13 & \text{for } i = 9, \\ -46.77 & \text{for } i = 11, \\ 0 & \text{for even } i. \end{cases} \tag{5.33}$$

The zero value on the even polynomials is due to their asymmetry; as the temperature profile is fundamentally symmetric, the polynomials with odd orders (z, z^3, z^5 &c.) make no net contribution to the steady-state temperature profile.

The resulting steady-state profile is shown in Figure 5.7.

5.5 Conclusions

This chapter has outlined a method and set of equations to govern both heat flow within the tube walls, and heat flow onto and off the walls. The model is now capable of accurately representing the non-adiabatic nature of the compression process in detail, which gives greater confidence in its results.

Chapter 6

Water cooling

This chapter covers an investigation into the proposition of spraying liquid water into the compression tube as small droplets. This is intended to reduce the maximum temperature seen, as the air mass' thermal energy is partially consumed by the larger heat capacity and phase change energy of the water. This is similar to the approach taken by the CAES companies LightSail Energy[48], General Compression[49] and SustainX[50].

We introduce a new subscript, m, which indicates that the term is related to the mixture of air, steam and water.

6.1 Thermal properties

We require several new parameters which give the thermal properties of water, both liquid and steam.

6.1.1 Saturation temperature and pressure

Firstly, we introduce the saturation temperature of water, T_{sat} (also known as the boiling point), which is dependent on pressure. Data from Rogers and Mayhew [85, p.3–5] is imported into MATLAB, as shown in Figure 6.1. As in section 3.4.4, we fit a polynomial curve to the data to increase speed compared to interpolating. A cubic polymonial fitted to the data, as plotted in Figure 6.1, has an error of around 0.1% for the range required.

6.1.2 Latent heat of evaporation

The model also needs an accurate value for the latent heat of evaporation, which is the energy required for the phase transition of water from liquid to gas, denoted here as $L_{f\to g}$, which is dependent on pressure. This is given as a set of points by Rogers and Mayhew [85, p.3–5], shown in Figure 6.2. A 9th-order polynomial, also plotted in Figure 6.2, has an error (relative to linear interpolation) of less than 1% until pressure reaches 10^7 Pa, which is above our target pressure.

6.1.3 Specific heat capacity of water

For accurate operation, the simulation needs to be able to evaluate the heat capacities at constant volume or pressure of liquid water, $c_{v,wf}$ and $c_{p,wf}$, and steam, $c_{v,wg}$ and $c_{p,wg}$, respectively.

Data from Rogers and Mayhew [85, p.10] is imported into MATLAB, and 4th-order polynomials are fitted to allow fast evaluation of both $c_{p,wg}$ and $c_{p,wf}$ as functions of temperature, as shown in Figure 6.3. Since liquid water is effectively incompressible, $c_{v,wf} = c_{p,wf}$, so the same polynomial can be used for both; for clarity, derivations will retain them as separate terms, and another function created for $c_{v,wf}$. The

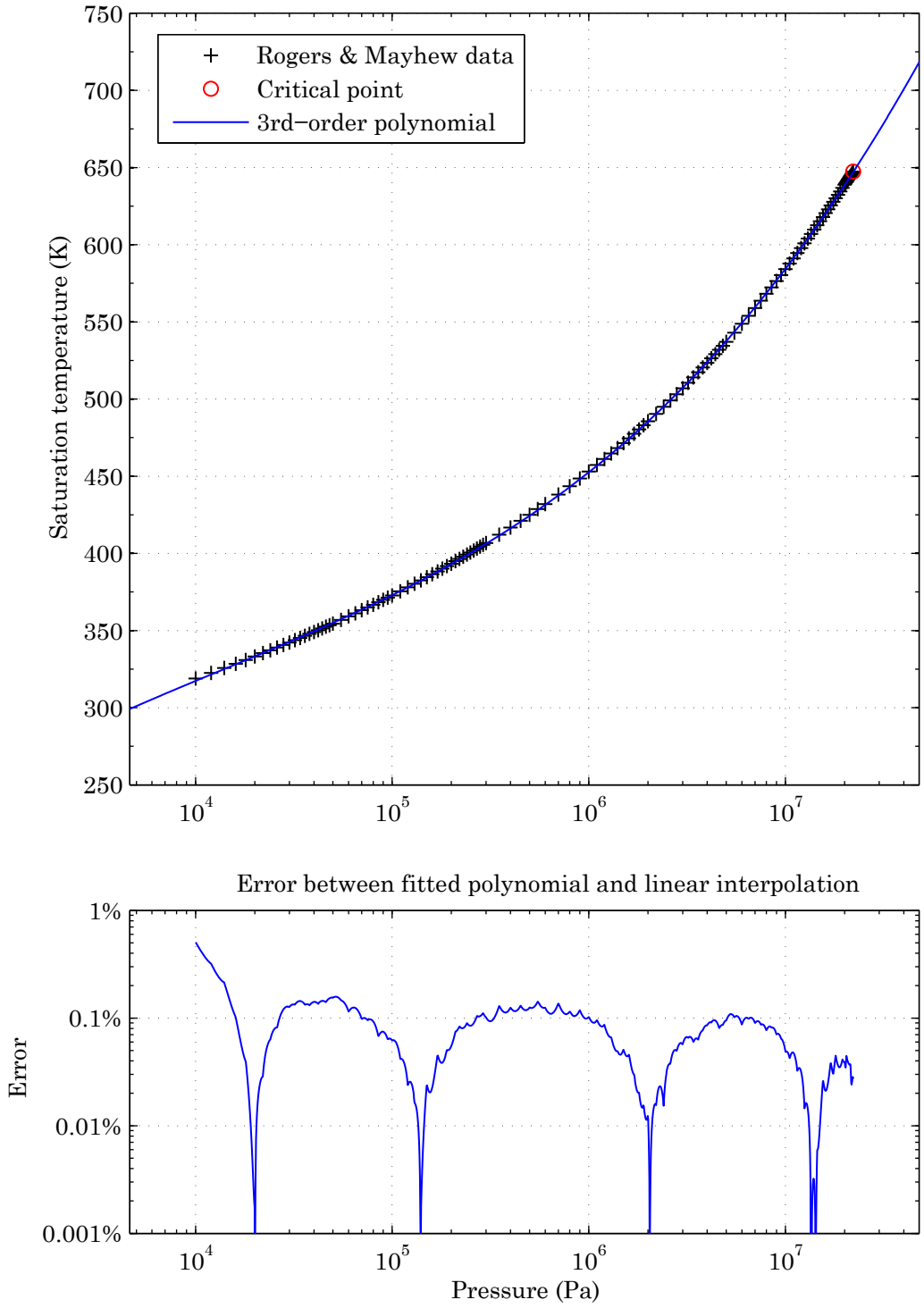


Figure 6.1: Saturation temperature of water, T_{sat}

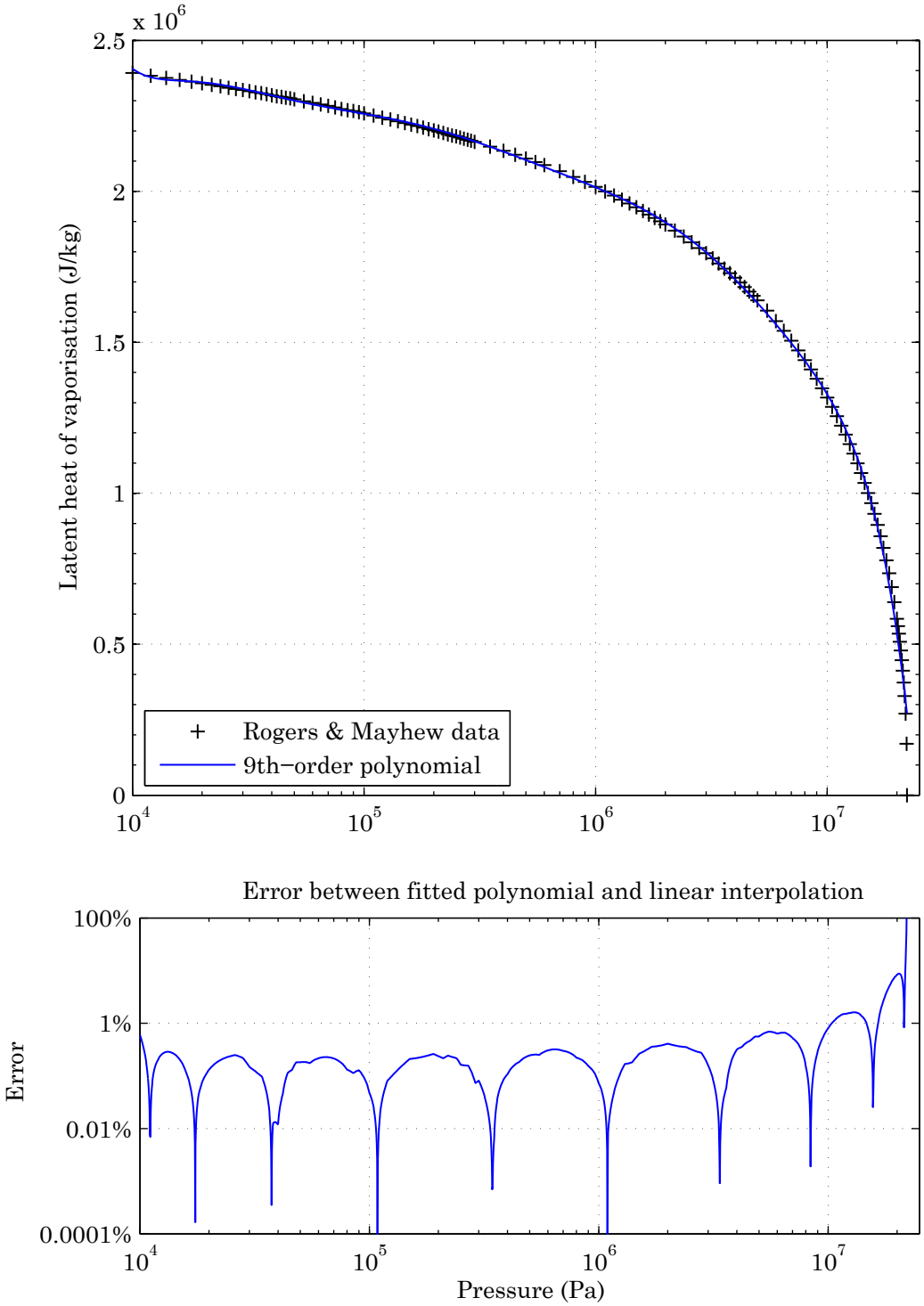


Figure 6.2: Latent heat of vaporisation of water, $L_{f \to g}$

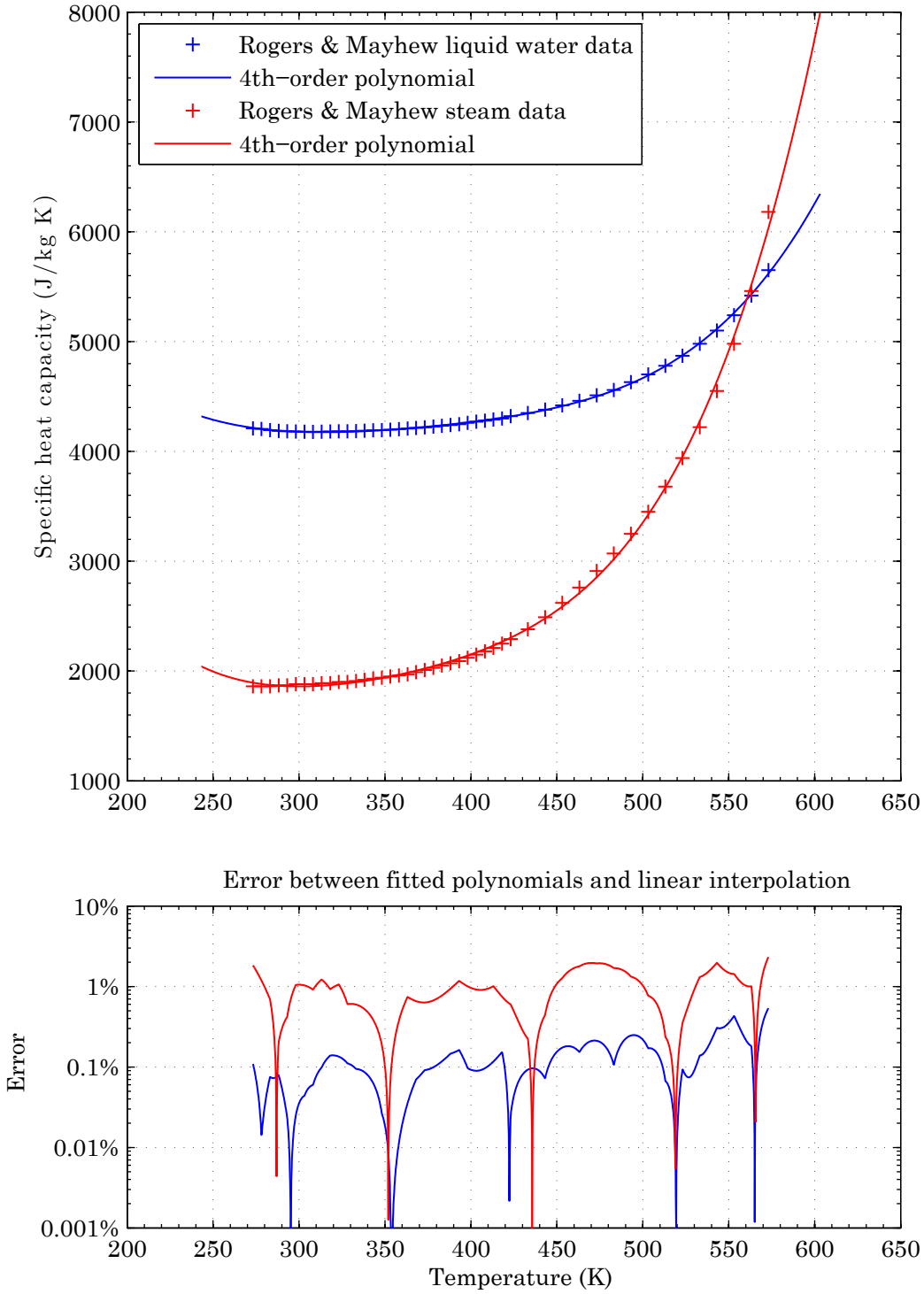


Figure 6.3: Specific heat capacities of water, $c_{p,wg}$ and $c_{p,wf}$

function for $c_{v,wg}$ is simply obtained from $c_{p,wg}$ using the universal gas constant $\bar{R}$ and the molar mass of water M_w:

$$c_{v,wg}(T) = c_{p,wg}(T) - \frac{\bar{R}}{M_w} \tag{6.1}$$

6.2 Additional state variables

Some assumptions are made:

- Water in the tube is either liquid or gas (steam)
- Steam temperature is equal to air temperature, so we replace the airmass temperature T_a with the air-steam mixture temperature T_m, using the subscript $_m$ to denote 'mixture'. This is justified due to the extremely turbulent nature of the flows within the compression tube.

At a minimum, we must add two new state variables.

m_{wf}	Mass of liquid water
m_{wg}	Mass of steam

Each has a corresponding entry in the rate vector $\dot{\mathbf{y}}$, and all three are 'paired' for the two different compression volumes, giving rise to individual state variables $m_{wf,1}$, $m_{wf,2}$, $m_{wg,1}$, and $m_{wg,2}$.

6.3 Flow through valves

We will assume that inducted air is completely dry. To apply this in the simulation, we sum the positive mass flow rates through the three valves (low-pressure to atmosphere,

high-pressure to storage, and dump to atmosphere) and apply that exclusively to $\dot{m_a}$. Then, we sum the negative mass flow rates to form $\dot{m_m}^{-\text{ve}}$, which is then divided between the three masses in the compression tube based on their mass fraction. Effectively, we are assuming perfectly mixed gas is output.

$$\dot{m_a} = \frac{m_a}{m_a + m_{wf} + m_{wg}} \dot{m_m}^{-\text{ve}} \tag{6.2}$$

$$\dot{m_{wf}} = \frac{m_{wf}}{m_a + m_{wf} + m_{wg}} \dot{m_m}^{-\text{ve}} \tag{6.3}$$

$$\dot{m_{wg}} = \frac{m_{wg}}{m_a + m_{wf} + m_{wg}} \dot{m_m}^{-\text{ve}} \tag{6.4}$$

6.4 Pressure and volume

The pressure function must also be altered. We need to work out the total number of moles of air and water vapour, using their molar masses M_a and M_w, then use the universal gas constant to calculate the total pressure.

$$p = \left(\frac{m_a}{M_a} + \frac{m_{wg}}{M_w} \right) \frac{\bar{R} \cdot T_m}{V} \tag{6.5}$$

All evaluations of the volume are adjusted to take into account the volume occupied by the liquid water. We assume its density to be constant.

$$V' = Ah_a - \frac{m_{wf}}{\rho_{wf}} \tag{6.6}$$

6.5 Two-temperature model

Initially, two temperature state variables per compression chamber were used:

T_m	Temperature of air-steam mixture
T_{wf}	Temperature of liquid water

Since T_a has been replaced with T_m, we must modify the functions for $\dot{T_a}$ similarly. We alter the thermal inertia:

$$\text{Thermal inertia for the mixture} = m_a c_{v,a}(T_m) + m_{wg} c_{v,wg}(T_m) \tag{6.7}$$

where $c_{v,a}(T_m)$ is the specific heat capacity at constant volume of air at T_m. This inertia is used throughout the calculations for the various parts of $\dot{T_m}$.

The heat capacity ratio, γ, must also be altered to reflect the effect of the steam, using an average of the air and steam c_v and c_p values weighted by their respective masses:

$$\begin{aligned} c_{p,m} &= \frac{c_{p,a} m_a + c_{p,wg} m_{wg}}{m_a + m_{wg}} \\ c_{v,m} &= \frac{c_{v,a} m_a + c_{v,wg} m_{wg}}{m_a + m_{wg}} \\ \gamma_m &= \frac{c_{p,m}}{c_{v,m}} = \frac{c_{p,a} m_a + c_{p,wg} m_{wg}}{c_{v,a} m_a + c_{v,wg} m_{wg}} \end{aligned} \tag{6.8}$$

6.5.1 Evaporation ODEs

To find the rates of change of these state variables, firstly we consider the rate of heat transfer into the liquid water, $\dot{Q}_{a \to wf}$.

$$\dot{Q}_{a \to wf} = k_{\text{evap}}(T_a - T_{\text{sat}}) m_{wf} \tag{6.9}$$

where T_a is the temperature of the air, T_{sat} is the saturation temperature of water at the current air pressure, and k_{evap} is a 'rate of evaporation' constant. m_{wf} is included so that the rate scales with the volume of water, which is used as a proxy for the available surface area for heat transfer to occur across. The effect of this energy transfer is added to the expression for $\dot{T_m}$ from Equation 3.23 using the new

thermal inertia:

$$\dot{T_m}' = \dot{T_m} + \frac{-\dot{Q}_{a\to wf}}{m_a c_{v,a} + m_{wg} c_{v,wg}} \tag{6.10}$$

Next, we formulate an expression for the fraction of $\dot{Q}_{a\to wf}$ which is being used for vaporising the water.

Assuming that the temperature of the liquid water particles follows a Gaussian probability density function (PDF), with $\mu = T_{wf}$ and $\sigma = 5$ K, we obtain the distributions shown in Figure 6.4:

$$\text{PDF}\left(T_i\right) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{\left(T_i - T_{wf}\right)^2}{2\sigma^2}} \tag{6.11}$$

$$\text{CDF}\left(T_i\right) = \frac{1}{2}\left(1 + \text{erf}\left(\frac{T_i - T_{wf}}{\sqrt{2\sigma^2}}\right)\right) \tag{6.12}$$

We assume that the energy passing into the water $\dot{Q}_{a\to wf}$ is split proportionally between water particles which are above the saturation temperature T_{sat} (and will use the energy for the phase transition) and those below T_{sat} (which will increase in temperature instead). This proportion is obtained by evaluating the CDF at T_{sat}.

The rate of vaporisation is thus simply:

$$\dot{m_{wf}}_{\text{evap}} = \frac{\left(1 - \text{CDF}\left(T_{\text{sat}}\right)\right) \cdot \dot{Q}_{a\to wf}}{L_{f\to g}(T_m)} \tag{6.13}$$

$$\dot{m_{wg}}_{\text{evap}} = -\dot{m_{wf}}_{\text{evap}} \tag{6.14}$$

where $L_{f\to g}(T_m)$ is the latent heat of vaporisation of water at T_m. The rate of change of temperature of the liquid water is similarly uncomplicated:

$$\dot{T_{wf}} = \frac{\text{CDF}\left(T_{\text{sat}}\right)\dot{Q}_{a\to wf}}{m_{wf} \cdot c_{v,wf}(T_{wf})} \tag{6.15}$$

where $c_{v,wf}(T_{wf})$ is the specific heat capacity at constant volume of liquid water, at

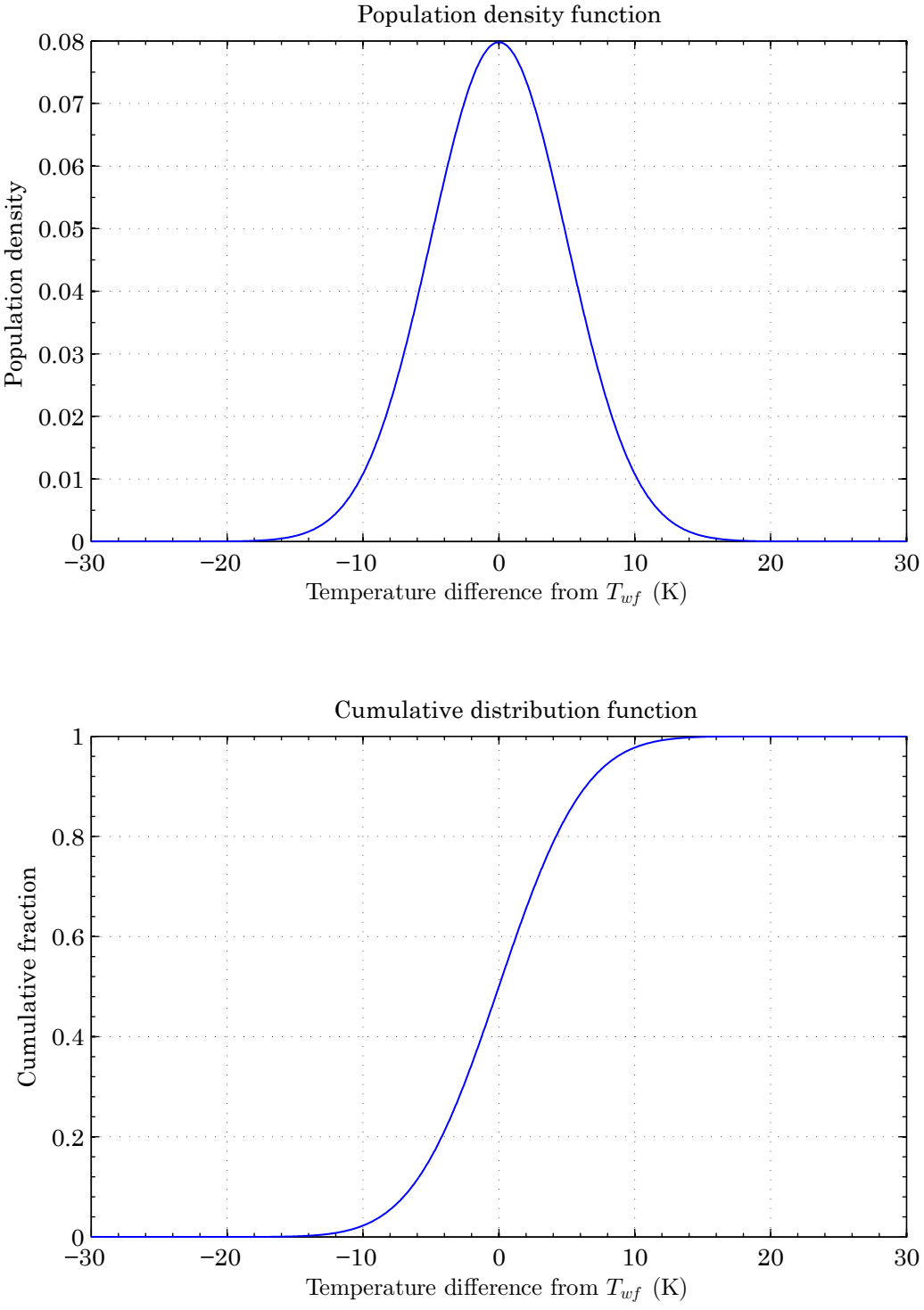


Figure 6.4: Particle temperature distribution from two-temperature wet model

the current temperature T_{wf}.

6.5.2 Energy required

Adding water also alters the energy required by the system. At present, the energy calculation in the model considers three components: to compress the dry air, to exhaust the dry air, and to defeat the small Coulomb friction. We will model the 'wet' compression as a stage of adiabatic compression followed by a stage of isothermal compression.

Adiabatic stage

Unlike the dry air energy required calculation, which assumed adiabatic compression from p_0 up to p_{targ}, the first stage of wet compression will run from $T_{a,0}$ to an estimated final temperature $T_{a,\max}$. We calculate the γ value for the mixture, γ_m, by assuming that all the water injected will evaporate instantly, so the mass of water is used as the mass of steam:

$$\gamma_m = \frac{c_{p,a} m_a + c_{p,wg} m_{wf}}{c_{v,a} m_a + c_{v,wg} m_{wf}} \tag{6.16}$$

Using the standard adiabatic relationships, we find that the piston position in the limits of the integral will be from h_0 to $h_0 \left(\frac{T_{a,0}}{T_{a,\max}}\right)^{\frac{1}{\gamma-1}}$, and use the substitution $pV = k_{\text{adi}}$, setting the constant using the initial conditions (pressure at p_0, position

at h_0).

$$\begin{aligned}
E_{\text{comp,adi}} &= \int p \,\mathrm{d}V \\
&= \int (k_{\text{adi}} A^{-\gamma} h^{-\gamma}) \,\mathrm{d}(Ah) \\
&= (p_0 A^\gamma h_0^\gamma)\, A^{1-\gamma} \int_{h_0\left(\frac{T_{a,0}}{T_{a,\max}}\right)^{\frac{1}{\gamma-1}}}^{h_0} h^{-\gamma} \,\mathrm{d}h \\
&= p_0 A h_0^\gamma \left[\frac{h^{1-\gamma}}{1-\gamma}\right]_{h_0\left(\frac{T_{a,0}}{T_{a,\max}}\right)^{\frac{1}{\gamma-1}}}^{h_0} \\
&= \frac{p_0 A h_0^\gamma}{1-\gamma} \left[h_0^{1-\gamma} - h_0^{1-\gamma} \left(\frac{T_{a,0}}{T_{a,\max}}\right)^{\frac{1-\gamma}{\gamma-1}}\right] \\
&= \frac{p_0 A h_0}{1-\gamma} \left(1 - \frac{T_{a,\max}}{T_{a,0}}\right)
\end{aligned} \tag{6.17}$$

Isothermal stage

This is followed by an isothermal process, in which $pV = k_{\text{iso}}$. We set the constant using the conditions at the end, where the pressure is p_{targ} and the piston position is $\frac{p_0 h_0}{p_{\text{targ}}} \left(\frac{T_{a,\max}}{T_{a,0}}\right)$. The energy in this stage is:

$$\begin{aligned}
E_{\text{comp,iso}} &= \int p \,\mathrm{d}V \\
&= \int (k_{\text{iso}} A^{-1} h^{-1}) \,\mathrm{d}(Ah) \\
&= \left(p_0 \left(\frac{T_{a,\max}}{T_{a,0}}\right)^{\frac{\gamma}{\gamma-1}} A h_0 \left(\frac{T_{a,\max}}{T_{a,0}}\right)^{\frac{1}{1-\gamma}}\right) \int_{\frac{p_0 h_0}{p_{\text{targ}}}\left(\frac{T_{a,\max}}{T_{a,0}}\right)}^{h_0\left(\frac{T_{a,0}}{T_{a,\max}}\right)^{\frac{1}{\gamma-1}}} h^{-1} \,\mathrm{d}h \\
&= p_0 A h_0 \left(\frac{T_{a,\max}}{T_{a,0}}\right) \left(\log_e \left(h_0 \left(\frac{T_{a,0}}{T_{a,\max}}\right)^{\frac{1}{\gamma-1}}\right) - \log_e \left(\frac{p_0 h_0}{p_{\text{targ}}} \left(\frac{T_{a,\max}}{T_{a,0}}\right)\right)\right) \\
&= p_0 A h_0 \left(\frac{T_{a,\max}}{T_{a,0}}\right) \log_e \left(\frac{p_{\text{targ}}}{p_0} \left(\frac{T_{a,\max}}{T_{a,0}}\right)^{\frac{\gamma}{1-\gamma}}\right)
\end{aligned} \tag{6.18}$$

Exhaust stage

During the exhaust stage, the work done is simply the final pressure multiplied by the tube area and the distance moved during exhaust (from the final piston position);

$$E_{\text{exh}} = Ah_0 p_0 \frac{T_{a,\max}}{T_{a,0}} \tag{6.19}$$

Total energy required

Adding these components to those which are the same as for dry air (energy to overcome friction, minus that done by the atmosphere) we obtain the overall expression:

$$\begin{aligned} E_{\text{req}} = \frac{p_0 Ah_0}{1-\gamma}\left(1 - \frac{T_{a,\max}}{T_{a,0}}\right) + p_0 Ah_0 \left(\frac{T_{a,\max}}{T_{a,0}}\right) \log_e \left(\frac{p_0}{p_{\text{targ}}}\left(\frac{T_{a,\max}}{T_{a,0}}\right)^{\frac{\gamma}{\gamma-1}}\right) \\ + p_0 Ah_0 \frac{T_{a,\max}}{T_{a,0}} + h_0 F_\mu - Ah_0 p_{\text{atm}} \end{aligned} \tag{6.20}$$

6.5.3 Stiffness problems

Implementing this two-temperature model, we run into a significant problem. The model becomes extremely stiff, requiring timesteps much smaller than is practically possible to solve. To investigate the cause of this instability, we will examine the time constant of the liquid water temperature state variable, through a simple water droplet model.

Water droplet test calculation

We consider a solid sphere of water, and look at the diffusion of heat into the droplet from the air surrounding it. The heat passing through a concentric spherical surface,

with a radius r and thermal conductivity k is given by;

$$Q(r) = -\frac{\partial T}{\partial r} \cdot k \cdot \left(4\pi r^2\right) \tag{6.21}$$

We can form the partial differential equation by considering the net heat going into a hollow-spherical element of thickness Δr;

$$(c_v\rho)\frac{\partial T}{\partial t}\left(4\pi r^2 \Delta r\right) = -\left|\frac{\partial T}{\partial r} k\left(4\pi r^2\right)\right|_r + \left|\frac{\partial T}{\partial r} k\left(4\pi r^2\right)\right|_{r+\Delta r} \tag{6.22}$$

$$\frac{c_v\rho}{k}\frac{\partial T}{\partial t}\left(r^2 \cdot \Delta r\right) = \left|\frac{\partial T}{\partial r} r^2\right|_{r+\Delta r} - \left|\frac{\partial T}{\partial r} r^2\right|_r \tag{6.23}$$

The right hand side is a derivative with respect to r, which obtains;

$$\frac{c_v\rho}{k}\frac{\partial T}{\partial t} r^2 = \frac{\partial^2 T}{\partial r^2} r^2 + 2r\frac{\partial T}{\partial r} \tag{6.24}$$

$$\frac{\partial T}{\partial t} = \frac{k}{c_v\rho}\left(\frac{\partial^2 T}{\partial r^2} + \frac{\partial T}{\partial r}\frac{2}{r}\right) \tag{6.25}$$

For water at 293 K, we take $k = 0.609$ W/m·K, $c_v = 3.951$ kJ/kg·K and $\rho = 998.3$ kg/m^3.

The initial temperature profile is a spherical droplet at a uniform 293 K, with the outer surface at 393 K. This is represented numerically as a large vector of temperature values, where the numerical derivatives at each point are defined as:

$$\left|\frac{\partial T}{\partial r}\right|_i = \frac{T_{i+1} - T_{i-1}}{r_{i+1} - r_{i-1}} \tag{6.26}$$

$$\left|\frac{\partial^2 T}{\partial r^2}\right|_i = \frac{T_{i+1} - 2T_i + T_{i-1}}{(r_{i+1} - r_i)(r_i - r_{i-1})} \tag{6.27}$$

with the end conditions:

$$\left|\frac{\partial T}{\partial r}\right|_{r_1} = \left|\frac{\partial T}{\partial r}\right|_{r_2} \text{ and } \left|\frac{\partial^2 T}{\partial r^2}\right|_{r_1} = \left|\frac{\partial^2 T}{\partial r^2}\right|_{r_2} \tag{6.28}$$

$$\left|\frac{\partial T}{\partial r}\right|_{r_{\text{end}}} = \left|\frac{\partial^2 T}{\partial r^2}\right|_{r_{\text{end}}} = 0 \tag{6.29}$$

If we substitute these into the PDE, we obtain;

$$\left|\frac{\partial T}{\partial t}\right|_{r_i} = \frac{k}{c_v \rho}\left(\frac{T_{i+1} - 2T_i + T_{i-1}}{(r_{i+1} - r_i)(r_i - r_{i-1})} + \frac{T_{i+1} - T_{i-1}}{r_{i+1} - r_{i-1}}\frac{2}{r_i}\right) \tag{6.30}$$

$$\begin{aligned}\left|\frac{\partial T}{\partial t}\right|_{r_i} = {} & \frac{k}{c_v \rho}\left[\frac{1}{(r_{i+1} - r_i)(r_i - r_{i-1})} + \frac{2}{r_i(r_{i+1} - r_{i-1})}\right] T_{i+1} \\ & + \frac{k}{c_v \rho}\left[\frac{-2}{(r_{i+1} - r_i)(r_i - r_{i-1})}\right] T_i \\ & + \frac{k}{c_v \rho}\left[\frac{1}{(r_{i+1} - r_i)(r_i - r_{i-1})} - \frac{2}{r_i(r_{i+1} - r_{i-1})}\right] T_{i-1}\end{aligned} \tag{6.31}$$

This allows us to calculate a matrix $\mathbf{D}$ such that;

$$\dot{T}(r) = \mathbf{D} \cdot T(r) \tag{6.32}$$

Modelling this in the MATLAB m-file `thermal_mod_func` with a 10 K temperature difference and a 2×10^4 m diameter droplet gives the profile shown in Figure 6.5. It can be readily seen that after only 1.5×10^{-5} seconds, the centre of the droplet has warmed to within 1 K of the outer edge. This fast time constant leads to an extremely high eigenvalue of the overall ODE system, increasing its stiffness.

Since the time constant is so fast, this variable lends itself extremely well to being removed. We will assume that the temperature changes instantaneously, so only one temperature state variable is needed for the whole mixture. This change (to a simple Differential Algebraic Equation, or DAE) should work to reduce the stiffness of the model and improve its efficiency.

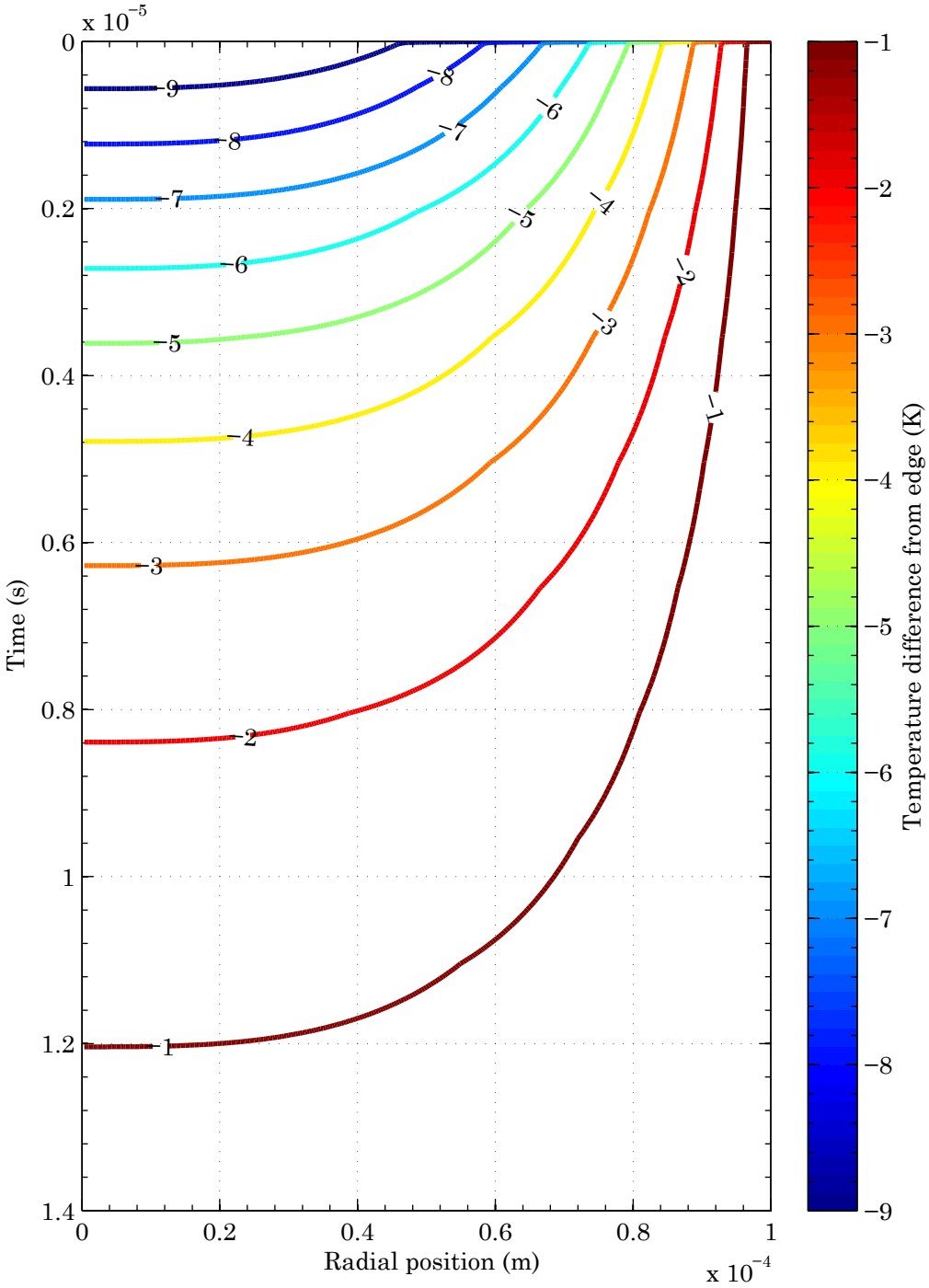


Figure 6.5: Thermal diffusion of 10 K gradient in water droplet

6.6 Single-temperature model

One assumption is added:

- The air, the steam and the liquid water all have the same temperature T_m.

This necessitates redefining γ_m:

$$\gamma_m = \frac{c_{p,a}m_a + c_{p,wg}m_{wg} + c_{p,wf}m_{wf}}{c_{v,a}m_a + c_{v,wg}m_{wg} + c_{v,wf}m_{wf}} \tag{6.33}$$

We also modify the thermal inertia of the mixture to include the effect of liquid water and steam:

$$\text{New thermal inertia} = m_a c_{v,a}(T_m) + m_{wg}c_{v,wg}(T_m) + m_{wf}c_{v,wf}(T_m) \tag{6.34}$$

6.6.1 States

We will assume that each mixture can have three possible states: liquid water, phase transition, and steam. In the liquid state, $m_{wg} = 0$; in the steam state, $m_{wf} = 0$.

In the MATLAB model, the states are tracked by a new field of the `sys` structure, `sys.phase`, which is a cell array of integers; a value of 1, 2 or 3 in the `sys.phase{2}` field shows that mixture 2 is liquid, in transition, or steam respectively. Moving between states is controlled by events which detect when either $m_{wf} = 0$ or $m_{wg} = 0$, in the phase transition, or when $T_m = T_{\text{sat}}$ otherwise.

6.6.2 Evaporation ODEs

During the phase transition, the mixture's temperature and pressure position is always on the saturation curve, $T_{\text{sat}}(p)$. This means that the rate of change of the

mixture temperature is simply the rate of change of the saturation temperature:

$$\dot{T_m} = \dot{T_{\text{sat}}} = \frac{\mathrm{d}T_{\text{sat}}(p)}{\mathrm{d}p} \cdot \dot{p} \tag{6.35}$$

The gradient of the saturation curve, $\frac{\mathrm{d}T_{\text{sat}}(p)}{\mathrm{d}p}$, can be obtained using the fitted polynomial being used to estimate the saturation curve, described in subsection 6.1.1. We calculate the numerical derivative for every line segment of the linear interpolation, then fit a polynomial to that curve; the result of this is shown in Figure 6.6. We obtain an expression for the rate of change of pressure, $\dot{p}$, by differentiating the rearranged ideal gas law with respect to t:

$$p = \left(\frac{m_a}{M_a} + \frac{m_{wg}}{M_w}\right) \frac{\bar{R}T_m}{Ah_a - \frac{m_{wf}}{\rho_w}} \tag{6.36}$$

$$p = \frac{\bar{R}}{A}\left(\frac{m_a T_m}{M_a} + \frac{m_{wg} T_m}{M_w}\right) \frac{1}{h_a} \tag{6.37}$$

$$\dot{p} = \frac{\bar{R}}{A}\left(\left(\frac{\dot{m_a} T_m + m_a \dot{T_m}}{M_a} + \frac{\dot{m_{wg}} T_m + m_{wg} \dot{T_m}}{M_w}\right) h - \left(\frac{m_a}{M_a} + \frac{m_{wg}}{M_w}\right) T_m \dot{h_a}\right) \frac{1}{h_a^2} \tag{6.38}$$

Note that we have assumed that $Ah_a \gg \frac{m_{wf}}{\rho_w}$, which is justified due to the large value of ρ_w. Problematically, this expression contains $\dot{m_{wg}}$ and $\dot{T_m}$ on the right-hand side, which both depend on $\dot{p}$.

Rearranging this is simplified by the introduction of several intermediate variables,

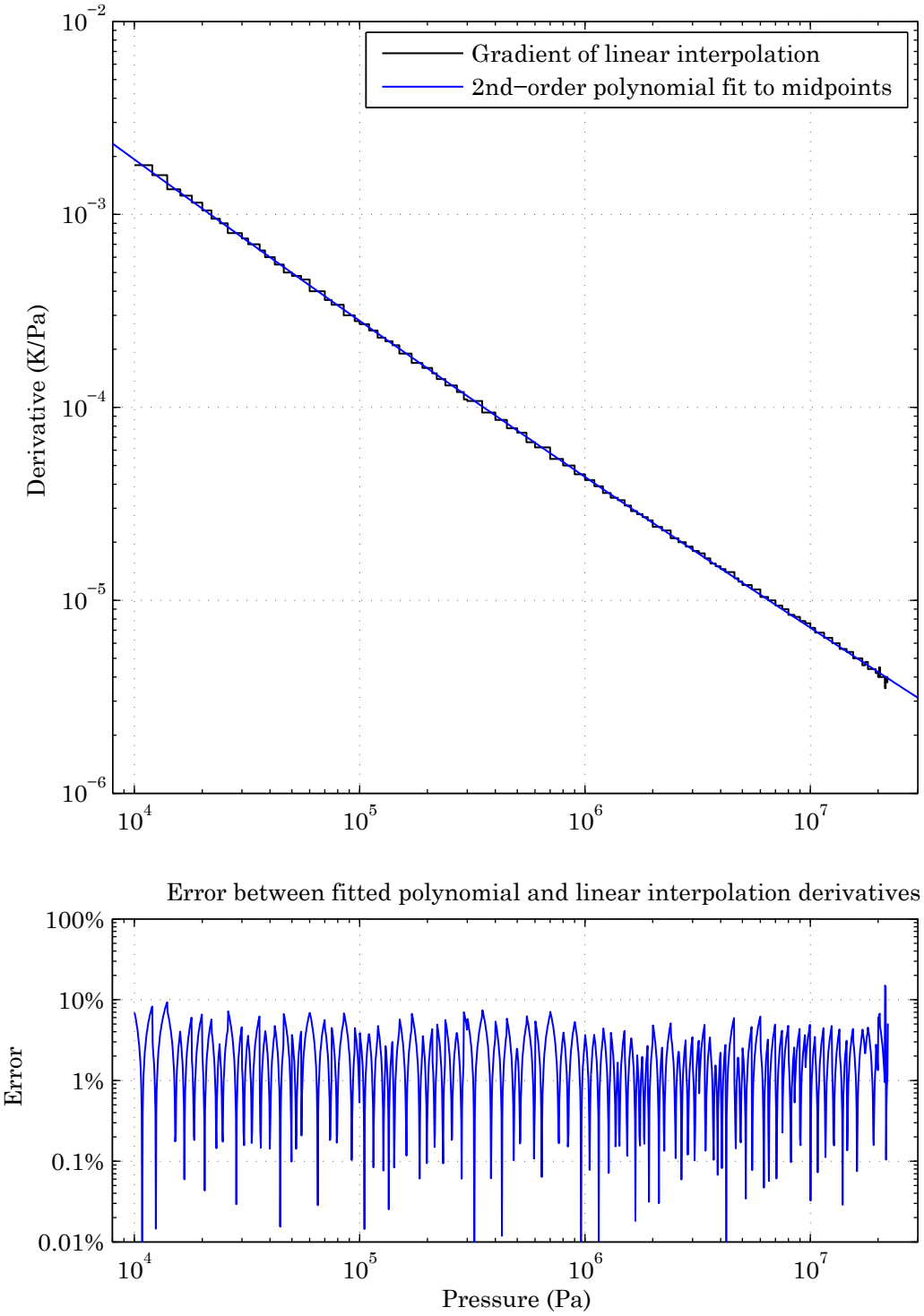


Figure 6.6: Derivative $\frac{\mathrm{d}T_{\mathrm{sat}}}{\mathrm{d}p}$ of water saturation curve

x_1 to x_4:

$$\dot{p} = \left(\frac{m_a}{M_a} + \frac{m_{wg}}{M_w}\right)\frac{\bar{R}}{Ah_a}\dot{T_m} + \left(\frac{\dot{m_a}}{M_a}T_m h_a - \left(\frac{m_a}{M_a} + \frac{m_{wg}}{M_w}\right)T_m\dot{h_a}\right)\frac{\bar{R}}{Ah_a^2} + \frac{T_m\bar{R}}{M_w A h_a}\dot{m_{wg}} \tag{6.39}$$

$$\dot{p} = x_1\dot{T_m} + x_2 + x_3\dot{m_{wg}} \tag{6.40}$$

$$\text{where } x_1 = \left(\frac{m_a}{M_a} + \frac{m_{wg}}{M_w}\right)\frac{\bar{R}}{Ah_a}$$

$$\text{and } x_2 = \left(\frac{\dot{m_a}\bar{R}}{M_a A} - x_1\dot{h_a}\right)\frac{T_m}{h_a}$$

$$\text{and } x_3 = \frac{T_m\bar{R}}{M_w A h_a}$$

Recalling that $\dot{T_m} = \dot{T_{\text{sat}}}$, we obtain:

$$\dot{T_{\text{sat}}} = \frac{\mathrm{d}T_{\text{sat}}(p)}{\mathrm{d}p}\left(x_1\dot{T_{\text{sat}}} + x_2 + x_3\dot{m_{wg}}\right) \tag{6.41}$$

$$\dot{T_{\text{sat}}} = \frac{x_2 + x_3\dot{m_{wg}}}{1 - x_1 x_4}x_4 \tag{6.42}$$

$$\text{where } x_4 = \frac{\mathrm{d}T_{\text{sat}}(p)}{\mathrm{d}p}$$

To find the evaporation rate $\dot{m_{wg}}$, we consider the adiabatic rate of change of temperature $\dot{T_{m\text{adi}}}$, which would be the actual rate $\dot{T_m}$ if a phase transition was not occurring, as in Equation 3.21:

$$\dot{T_{m\text{adi}}} = T_m\left(1 - \gamma_m\right)\left(\frac{\dot{m_a}}{m_a} - \frac{\dot{h_a}}{h_a}\right) \tag{6.43}$$

γ_m here is obtained with Equation 6.33, using the current values of m_{wf} and m_{wg}. The difference between the two rates $\dot{T_{m\text{adi}}}$ and $\dot{T_{\text{sat}}}$, shown in Figure 6.7, is multiplied by the heat capacity of the mixture to give an expression for the power being used for the phase change. Dividing by the latent heat of vaporisation at the current pressure, $L_{f\to g}(p)$, will give the evaporation rate. We also include a component due to the instantaneous flow through valves, $\dot{m_{wg_{\text{vlv}}}}$, as given by Equation 6.3 and

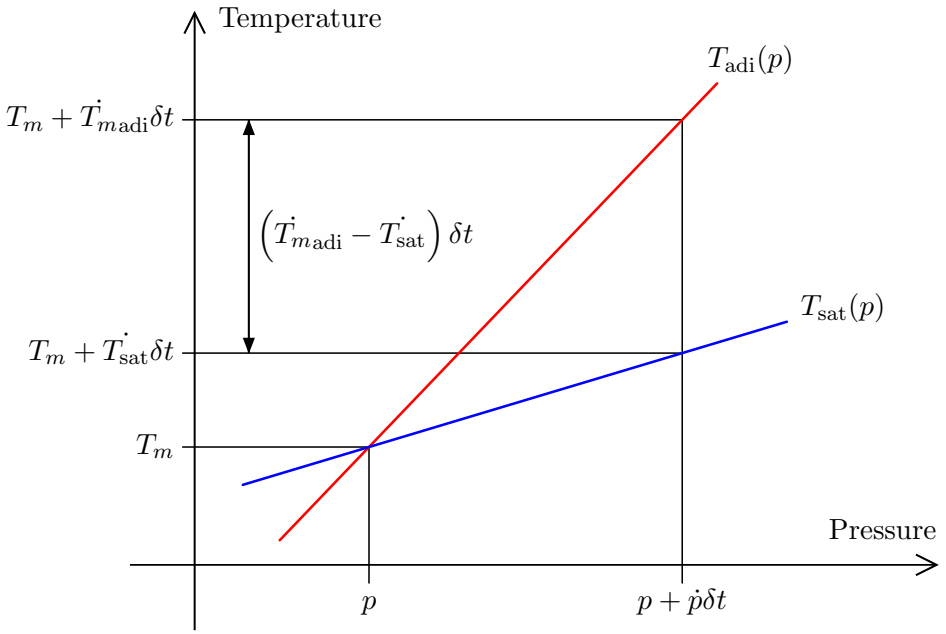


Figure 6.7: Calculation of evaporation power

Equation 6.4.

$$\dot{m_{wg}} = \left(\dot{T}_{m\text{adi}} - \dot{T}_{\text{sat}}\right) x_5 + \dot{m_{wg}}_{\text{vlv}} \tag{6.44}$$

$$\text{where } x_5 = \frac{m_a c_{p,a}(T_m) + m_{wg} c_{p,wg}(T_m) + m_{wf} c_{p,wf}(T_m)}{L_{f\to g}(p)} \tag{6.45}$$

Our final expression is thus:

$$\dot{m_{wg}} = \frac{\dot{T}_{m\text{adi}} x_5 + \dot{m_{wg}}_{\text{vlv}} - \dfrac{x_2 x_4 x_5}{1 - x_1 x_4}}{1 + \dfrac{x_3 x_4 x_5}{1 - x_1 x_4}} \tag{6.46}$$

By symmetry, the mass flow rate of the liquid component is $\dot{m_{wf}} = -\dot{m_{wg}}$. A temperature - pressure plot for this model is shown in Figure 6.8. It can be seen that the trajectory hews extremely closely to the saturation curve during the evaporation stage.

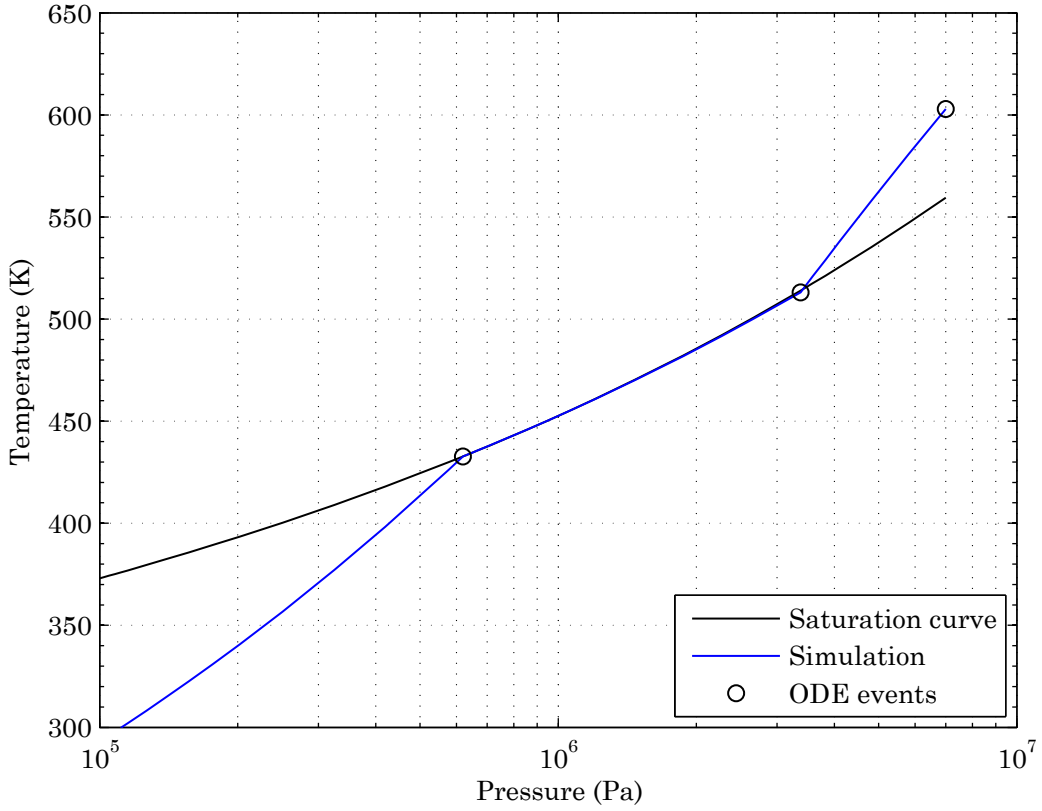


Figure 6.8: Example evaporation curve

6.6.3 Energy required

It is significantly more complicated to attempt an analytical solution to find the energy required to compress the air in this case. Instead, we construct a simple function, `SW_water_energy`, which takes the model's parameters structure `GP` and runs a simple, non-rotating, constant piston velocity model.

The initial conditions set the airmass at ambient temperature and pressure, and use any downward velocity and initial position `h_init` for the piston. The amount of water to be added is a given fraction, `GP.water_add`, of the mass of air in the compression chamber. An internal `events` function detects when the simulation has reached the saturation temperature and when the water mass has fully evaporated,

and switches a `phase` variable accordingly.

The simple model has an internal state variable to track work done, `y.WD`, with the rate given by:

$$\frac{\mathrm{d}}{\mathrm{d}t}\texttt{y.WD} = -\dot{h_m}Ap \tag{6.47}$$

The simulation ends when the airmass reaches the target pressure, p_{targ}. It then sums the compression work done, the energy required to exhaust (based on the final piston height), the work done against Coulomb friction, and the negative work done by atmospheric pressure on the opposite side of the piston:

$$E_{\text{req}} = \texttt{y.WD(end)} + h_{\text{end}}Ap_{\text{targ}} + h_{\text{init}}F_{\mu} - h_{\text{init}}Ap_{\text{atm}} \tag{6.48}$$

Dividing through by the initial airmass height h_{init} obtains an expression for energy required per metre, suitable for use based on the piston position. The energy function `SW_control_energy` is updated to use this method for calculating energy required.

6.6.4 Verification

Unfortunately, practical data was not available to verify this model. However, many of the specific functions used have a basis in the literature.

- The technique for calculating the values of γ and thermal inertia of the mixture, in Equation 6.33 and Equation 6.34, is similar to the approach taken by Kim et al.[47].
- Equation 6.45 uses the phase change energy of the water at the current pressure to calculate the mass flow rate of evaporation, in a similar manner to work by Qin and Loth[45] and Barrow and Pope[46].

- Finally, the use of the temperature derivative to map the pressure change onto the evaporation curve, as shown in Figure 6.7, is supported by work by Fu et al.[87].

6.7 Conclusions

Figure 6.9 shows the levels of liquid water and steam changing at the end of a compression stage, when the air is hot. The upper graph clearly shows the transition onto and off the saturation curve, corresponding with the start and end of the phase transition, along with the condensation when the temperature dropped at the end of the exhaust stroke.

A series of simulations were run to determine the effect of the water mass fraction on the peak temperature. To obtain the target temperature reduction, from around 900 K to 700 K, we use a water mass fraction of 0.06 (i.e. 0.06 kg of water are added at the start of the compression stage for every kg of air in the cylinder). The resulting profile, compared to the system without water-cooling,is shown in Figure 6.10.

From these results, we conclude that water injection is suitable for beneficially reducing the temperature of the exhaust air. Further research is needed to investigate the system efficiency at different levels of water mass fraction, and to more accurately simulate the water droplets with velocity and size considerations taken into account.

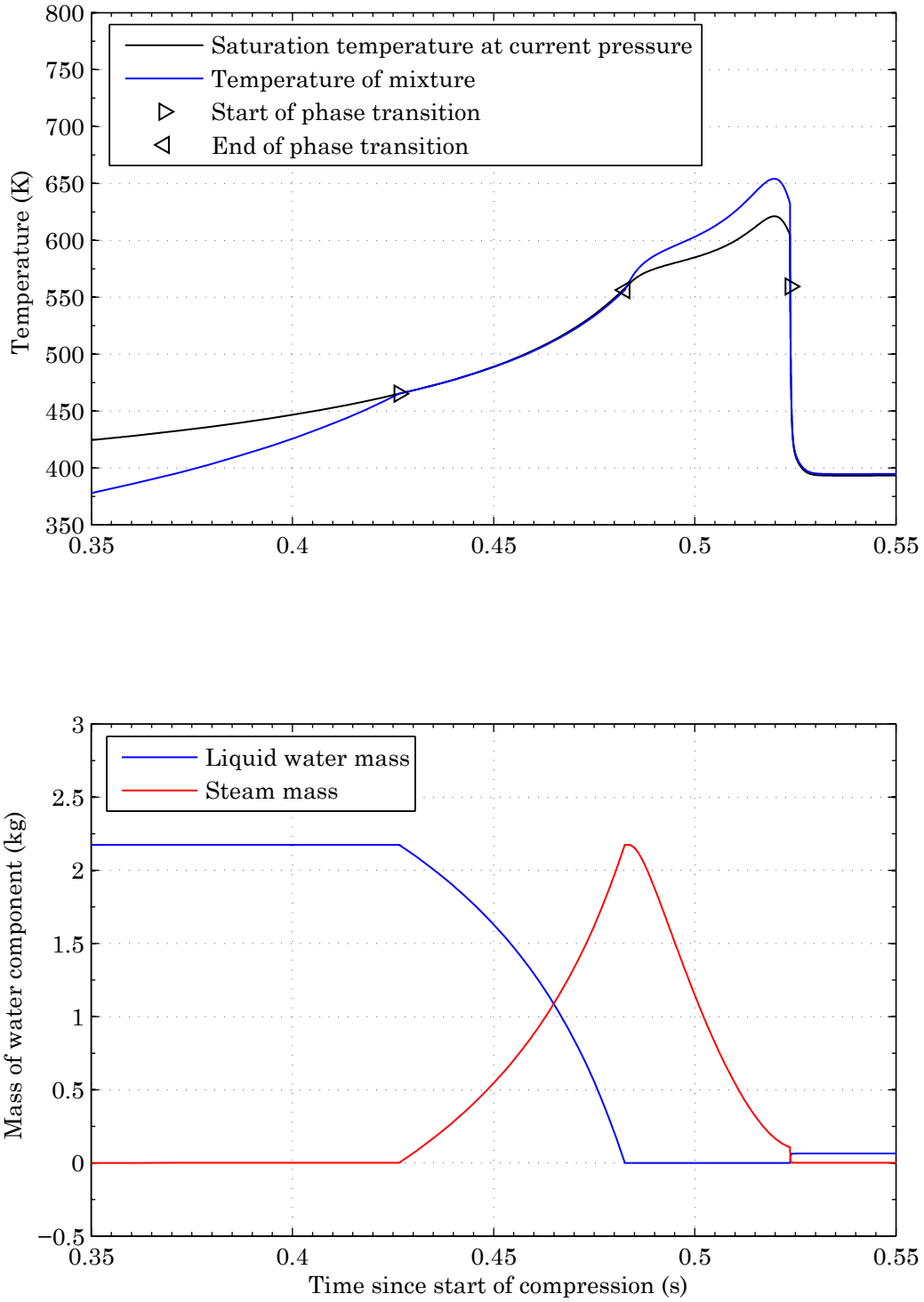


Figure 6.9: Phase transitions of the water during a stroke

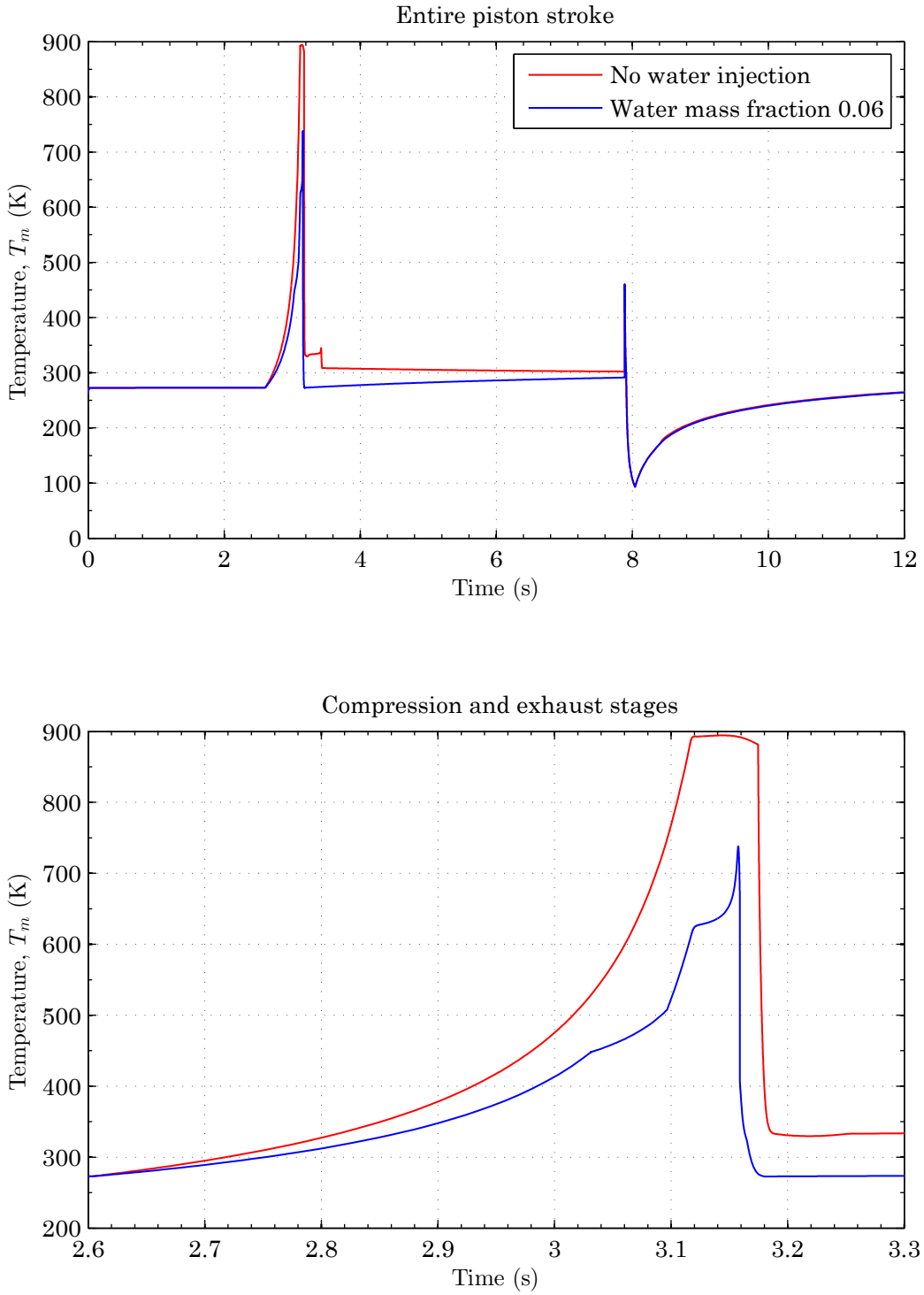


Figure 6.10: The effect of water injection on temperature in the compression chamber

Chapter 7

Exhaust valve control

The control of the exhaust valves in the system is paramount; by controlling the compression and exhaust of the air, as well as the 'kick' to drive the pistons against centrifugal force at the start of the stroke, the intention is that the principle method of control for the whole system should be through the air valves. This involves consideration of both timing, as outlined in other chapters, and throttling.

The control of the exhaust valve is a difficult problem, although it involves few state variables and a single second-order system, because it is extremely nonlinear and operates over a very short timescale. The time interval between the airmass being compressed reaching the target pressure, and that airmass being subsequently exhausted, is very short, around 0.08 seconds; the piston must be stopped with a minimum of wasted energy as close to the end of the compression tube as possible. Additionally, the period over which the exhaust valve acts is not known in advance, prevent the use of control techniques which operate continuously or with known start and end states. In this chapter, two control techniques are investigated.

A simple method is described, which modifies the energy calculation which determines the timing of the start of compression to include an energy surplus. Then, the HP valve is throttled to gradually remove the energy as the exhaust stage continues.

A second method is then shown, which uses a hierarchical twin controller system. The method models the valve as a simple mass-spring-damper system tracking a parameter-defined trajectory. A 'fast' proportional controller keeps the valve constant on the trajectory, while a second 'slow' controller periodically updates the trajectory definition parameters ψ based on simulations of the exhaust stroke.

7.1 Simple method

After the energy calculations (detailed in chapter 4) estimate that the energy required is equal to total potential energy, the system will close the outflow valve and keep the air mass inside the tube constant while it is compressed. This event is located based on estimates of the system's energy, and (particularly for the physical system) this will not be a perfect solution. As a result, the piston may have too much or too little energy, and a control system is required if the system is to operate at peak efficiency and to avoid destructive impacts. To simplify the control needed, all adjustments will take place after the air has reached the target pressure and while it is being exhausted to the reservoir.

In cases where the piston has too much energy, throttling the exhaust valve to raise the pressure in front of the piston will provide an adequate way to prevent contact between the piston and the tube end. This has the added advantage that the surplus energy is removed linearly with the air flow out of the piston, reaching zero at the same time the piston stops.

On the other hand, if the piston has too little energy, the solution is much more difficult; dumping some volume of air to atmosphere is an undesirable solution, due to both noise and inefficiency problems. To illustrate, we consider a case in which the piston has less energy than is required to exhaust the remaining compressed air, so a volume surplus ($A \cdot \delta h$) needs to be dumped. The surplus energy δE_{exh} in this

case is given by:

$$\begin{aligned} \delta E_{\text{exh}} &= A p_{\text{targ}} \delta h \\ \delta h &= \frac{\delta E_{\text{exh}}}{A p_{\text{targ}}} \end{aligned} \tag{7.1}$$

Assuming the compression was perfectly adiabatic, we can find the corresponding initial volume of the air which will be dumped, $A(\delta h)_0$:

$$\left(\frac{A(\delta h)_0}{A\delta h}\right)^{\gamma} = \left(\frac{p_{\text{targ}}}{p_{\text{atm}}}\right) \tag{7.2}$$

$$(\delta h)_0 = \frac{\delta E_{\text{exh}}}{A p_{\text{targ}}} \left(\frac{p_{\text{targ}}}{p_{\text{atm}}}\right)^{\frac{1}{\gamma}} \tag{7.3}$$

Finally, we substitute this into the adiabatic compression energy expression, Equation 4.23 to calculate the energy already expended to compress this surplus volume, $E_{\text{comp,surp}}$;

$$\begin{aligned} \delta E_{\text{comp}} &= (\delta h)_0 \frac{A p_{\text{atm}}}{1-\gamma} \left(1 - \left(\frac{p_{\text{atm}}}{p_{\text{targ}}}\right)^{\frac{1-\gamma}{\gamma}}\right) \\ \delta E_{\text{comp}} &= \frac{\delta E_{\text{exh}}}{A p_{\text{targ}}} \left(\frac{p_{\text{targ}}}{p_{\text{atm}}}\right)^{\frac{1}{\gamma}} \frac{A p_{\text{atm}}}{1-\gamma} \left(1 - \left(\frac{p_{\text{atm}}}{p_{\text{targ}}}\right)^{\frac{1-\gamma}{\gamma}}\right) \\ \frac{\delta E_{\text{comp}}}{\delta E_{\text{exh}}} &= \frac{1}{p_{\text{targ}}(\gamma - 1)} \left(p_{\text{targ}} - p_{\text{atm}} \left(\frac{p_{\text{targ}}}{p_{\text{atm}}}\right)^{\frac{1}{\gamma}}\right) \end{aligned} \tag{7.4}$$

Substituting in our parameters and taking $\gamma = 1.401$, we find that 1.755 J of already-used energy must be wasted for every Joule of surplus. This is very inefficient, so to avoid needing to dump air, the energy required function is adjusted upwards, erring on the side of a higher-energy piston. The control algorithm will then set the valve parameter on the basis of the current energy surplus (recalculated at every step).

7.1.1 Control algorithms

The first step is to obtain a 'reference' HP valve constant $k_{\mathrm{HP,ref}}$. We define the mass flow rate equation;

$$k_{\mathrm{HP,ref}} = \frac{\dot{m}}{\Delta p} \tag{7.5}$$

If $k_{\mathrm{HP,ref}}$ is sufficiently large, little energy is lost, since the pressure difference across the valve $\Delta p \to 0$. To pick a suitable basic value of $k_{\mathrm{HP,ref}}$, we estimate the total amount of energy we are expecting to lose due to the valve throttling, E_{loss}. This is given by:

$$E_{\mathrm{loss}} = V_T \times \Delta p \tag{7.6}$$

$k_{\mathrm{HP,ref}}$ can now be found by considering typical average values of $\dot{m}_a$ and V_T during exhaust for an uncontrolled model.

We form an error signal ϵ based on the proportional difference between the current energy surplus δE and a 'desired' energy surplus δE_{des}, both normalised relative to E_{loss};

$$\epsilon = \frac{\delta E - \delta E_{\mathrm{des}}}{E_{\mathrm{loss}}} \tag{7.7}$$

δE_{des} is included to force the system to gradually remove the energy surplus as the piston moves towards its endpoint; without this factor, the control function would attempt to reduce the energy surplus to zero immediately, risking it overshooting and requiring energy to be wasted to correct it. We define δE_{des} using a ramping function based on the fraction of the exhaust stroke distance the piston has travelled. When the piston has stopped moving at the end of the exhaust stroke, it is locked in place at a position h_{lock}. The piston position recorded at the start of the exhaust

stroke was h_{exh}.

$$\delta E_{\text{des}} = E_{\text{loss}} \times \frac{h_{\text{lock}} - h_a}{h_{\text{lock}} - h_{\text{exh}}} \tag{7.8}$$

The 'working' value of the valve constant, k_{HP}, is calculated from $k_{\text{HP,ref}}$ and the error multiplied by a gain, G;

$$k_{\text{HP}} = k_{\text{HP ref}} \times 10^{-G\epsilon} \tag{7.9}$$

The exponential form of this control algorithm allows it to react suitably to large error signals while preventing it from becoming negative, since a negative valve constant is a physical impossibility.

7.1.2 Derivative control

As described above, the system is based on proportional control only, with no derivative or integral components. However, since the system needs to respond extremely quickly, we now consider a component based on the derivative of the error signal. To obtain an estimate of the derivative in a causal system, we follow standard practice in using a first-order filter. In effect, this is emulating a series RL circuit, in which the voltage across a resistor and an inductor in series is taken as the input, and the voltage across the inductor along is the output. This has the transfer function:

$$\frac{V_{\text{out}}}{V_{\text{in}}} = \frac{Ls}{R + Ls} \tag{7.10}$$

where R is the resistance of the resistor, and L is the inductance of the inductor. For $R \gg L$, this functions as a 'differentiator' - the output signal will approximate LR^{-1} times the rate of change of the input signal.

We model this in our state-space model using a state variable, I (the current in the

inductor), governed by

$$L\frac{\mathrm{d}I}{\mathrm{d}t} = V_{\mathrm{in}}(t) - RI(t) \tag{7.11}$$

$$\dot{I} = \frac{1}{L}\left(V_{\mathrm{in(t)}} - RI(t)\right) \tag{7.12}$$

The derivative is thus approximated by:

$$\dot{V_{\mathrm{in}}} \approx \frac{R}{L}\left(V_{\mathrm{in(t)}} - RI(t)\right) \tag{7.13}$$

Figure 7.1 shows a control block diagram of the control system.

7.1.3 Implementation

Problems were encountered implementing this control system, relating to the instability of the error signals. Our model avoids these issues by saturating at two points; once, applied to the proportional error signal before it is used to calculate the derivative signal; and a second time, after the exponential is taken, to represent the upper limit on the physical valve.

The model performs relatively well; with the control functions set up to attempt to stop the piston at the end of the tube, simulations report the piston stops around 2 mm from the end. However, moving to a new system was required, which would better represent both the time delay on the valve response as well as the desire for a smoother valve throttling trajectory.

7.2 Hierarchical twin controller system

7.2.1 Principles

In order to be as efficient as possible, our control algorithm must have two objectives:

1. The `energycalc` function calculates both the energy required to compress the air in the system, and the potential energy in the system, as outlined in autorefsec:sysenergy.
2. The `10^` function returns the value of 10 raised to the power of the input signal, as laid out in Equation 7.9.
3. The `sat` function saturates the input signal to some set limit using a tan function.

Figure 7.1: Control block diagram for simpler control system

- when piston velocity $\dot{h_a} = 0$, air mass height $h_a = 0$
- when $\dot{h_a} = 0$, the surplus energy $E_\Delta = 0$

We define a simple linear trajectory for the target valve coefficient $k_{\text{HP,targ}}$, defined by two parameters:

$$k_{\text{HP,targ}} = \psi_d(\psi_e - t) \tag{7.14}$$

ψ_d is the decay parameter, which controls the rate at which the valve is closed, and ψ_e is the end parameter, which controls the time at which the valve will be fully closed. These parameters form the interface between two separate controllers, being set by a 'slow' controller and followed by a 'fast' controller.

Fast controller

The fast controller, which updates many times for each update of the slow controller, is a simple proportional and derivative controller. It tries to keep the valve constant k_{HP} on a trajectory defined by $k_{\text{HP,targ}}$, calculated using values of ψ_d and ψ_e it has been passed, using a combination of derivative and proportional control mechanisms.

We also model the valve constant as a simple mass-spring-damper system in its own right, to simulate the time delay between the control signal and the actual valve closure. This a common approach, as seen in work by Hõs et al.[88], Xu et al.[89] and Garcia[90]. The specific damping and stiffness constants are similar to those in Hõs et al.[88], picked to be suitably fast for the control purposes of the system.

Slow controller

The slow controller explores the phase space of ψ values near the current set, determining better parameters using Newton's method of optimisation. It runs

concurrently with the fast controller, sending out an updated set of parameters after each iteration.

Firstly we define a small proportionate perturbation, $\delta = 0.05$. Next, the controller runs five simulations, which all simulate the exhaust stage up until $\dot{h_a} = 0$, with slightly different parameters in each case:

(0,0) with $\psi_d = \psi_{d,0}$ and $\psi_e = \psi_{e,0}$, the current values

(1,0) with $\psi_d = (1 + \delta)\psi_{d,0}$ and $\psi_e = \psi_{e,0}$

(-1,0) with $\psi_d = (1 - \delta)\psi_{d,0}$ and $\psi_e = \psi_{e,0}$

(0,1) with $\psi_d = \psi_{d,0}$ and $\psi_e = (1 + \delta)\psi_{e,0}$

(0,-1) with $\psi_d = \psi_{d,0}$ and $\psi_e = (1 - \delta)\psi_{e,0}$

This quincunx pattern of points is known as a "five-point scheme", often used to calculate finite difference approximations to the first and second derivatives of a data field in two dimensions. We only need the first derivatives, but by taking all five points we can avoid bias in our estimation.

Each simulation produces two outputs, which are the values of the targets h_a and E_Δ when the simulation finished at $\dot{h_a} = 0$. For each of these two targets, we calculate a gradient to obtain an approximation to the partial derivative of each target value with respect to each parameter. So:

$$\frac{\partial h_a}{\partial \psi_d} = \frac{h_{a,(1,0)} - h_{a,(-1,0)}}{2\delta\psi_d} \tag{7.15}$$

Similarly:

$$\frac{\partial h_a}{\partial \psi_e} = \frac{h_{a,(0,1)} - h_{a,(0,-1)}}{2\delta\psi_e} \tag{7.16}$$

$$\frac{\partial E_\Delta}{\partial \psi_d} = \frac{E_{\Delta,(1,0)} - E_{\Delta,(-1,0)}}{2\delta\psi_d} \tag{7.17}$$

$$\frac{\partial E_\Delta}{\partial \psi_e} = \frac{E_{\Delta,(0,1)} - E_{\Delta,(0,-1)}}{2\delta\psi_e} \tag{7.18}$$

$$\begin{bmatrix} \frac{\partial h_a}{\partial \psi_d} & \frac{\partial h_a}{\partial \psi_e} \\ \frac{\partial E_\Delta}{\partial \psi_d} & \frac{\partial E_\Delta}{\partial \psi_e} \end{bmatrix} \begin{bmatrix} \Delta\psi_d \\ \Delta\psi_e \end{bmatrix} = \begin{bmatrix} \Delta h_a \\ \Delta E_\Delta \end{bmatrix} \tag{7.19}$$

The desired change in the targets, on the right-hand side, is that the values should change from those obtained from the current values ($h_{a,(0,0)}$ and $E_{\Delta,(0,0)}$) to zero. By inverting the matrix, we can calculate the required changes in the parameters, $\Delta\psi_d$ and $\Delta\psi_e$, to obtain this change.

$$\begin{bmatrix} \frac{\partial h_a}{\partial \psi_d} & \frac{\partial h_a}{\partial \psi_e} \\ \frac{\partial E_\Delta}{\partial \psi_d} & \frac{\partial E_\Delta}{\partial \psi_e} \end{bmatrix}^{-1} \begin{bmatrix} -h_{a,(0,0)} \\ -E_{\Delta,(0,0)} \end{bmatrix} = \begin{bmatrix} \Delta\psi_d \\ \Delta\psi_e \end{bmatrix} \tag{7.20}$$

The improved values of the parameters are then passed over to the fast controller, and the slow controller starts again from the new values. This whole cycle has taken a time t_δ.

7.2.2 Implementation in model

The MATLAB model constructed has one disadvantage and a few advantages compared to implementing the control mechanism in a real system.

The disadvantage is that, while a real system would be able to run the slow controller in parallel with the fast controller and the system progressing, we are limited to

strictly non-parallel operation; generally, MATLAB can only execute one command at a time. To solve this, we use one of our advantages - that we can 'pause time', by halting execution of the system simulation. If we do this, we can record the current state of the targets h_a and E_Δ and, while 'offline', calculate (by the slow controller algorithm described above) the next pair of values for the parameters ψ_d and ψ_e. This is explained in Figure 7.2.

We must, however, continue the system simulation for a period of time before we stop again and pass our new parameters in for the fast controller, to represent the time the slow controller would take to run. This one-step delay must be observed to avoid 'cheating' by running the slow controller 'instantaneously' (from the point of view of the simulation proper).

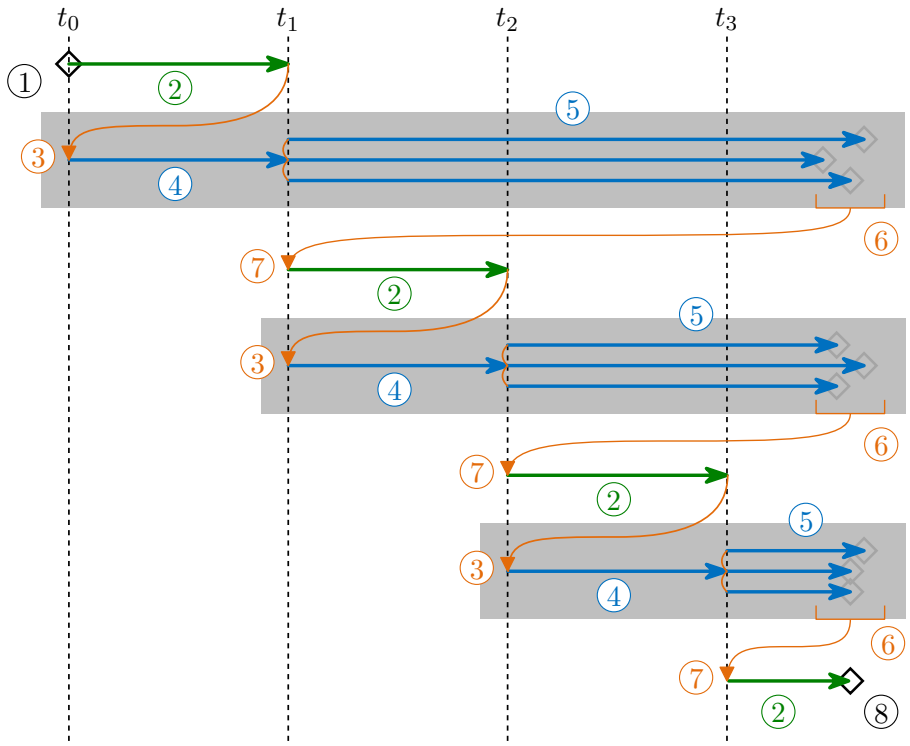
An advantage is that we possess a complete and accurate model of the simulation for the slow controller to use; the simulation itself. To avoid false confidence from this, we must implement two features.

Reduced simulation complexity

The first feature is a flag variable, part of the parameter structure `GP`, which is set to `true` for the simulations run by the slow controller. Inside the ODE function, we now selectively eliminate terms from the ODE, both to increase the speed of the slow controller simulations and to represent the inaccuracy of the sensors. In particular, we remove the terms corresponding to the wall temperatures T_{wall}, the fast controller mass-spring-damper system and the shock absorbers at the tube ends.

Simulated sensors

Secondly, we create a 'Chinese wall' between the control and simulation functions: the function `SW_sensors`, which converts a state vector into a structure of 'sensor



1. The exhaust stage of the 'real' simulation is triggered from the ODE events function. The current state vector is recorded, and the simulation starts with a first-guess at ψ.

2. The 'real' simulation runs for a short period of time, the approximate runtime for the 'slow' controller (on specialised hardware).

3. When the 'real simulation pauses, the recorded state vector is passed to the slow controller function.

4. The slow controller runs a 'fast-forward' simulation, which uses the current value of ψ to predict what the state will be at the time the slow controller finishes.

5. The slow controller then runs several simulations, each with a different value of ψ, for use in the optimisation problem.

6. The results from those simulations are used to return an improved value for ψ.

7. The core script records its latest state vector for the next time, and the 'real' simulation resumes with the improved value for ψ.

8. When the piston halts, the exhaust stage is complete.

Figure 7.2: 'Fast' and 'slow' simulated control scheme

data'. By ensuring the various control functions can only accept state information in this form, we can guarantee that all data to the controller has passed through this function, which can itself then deliberately degrade the results. For example, it would not be possible to sense the air and water masses in the compression tube, so the sensor structure first erases those terms, then reconstructs them using the other variables and the pressure. It would also not be able to detect the compression cylinder wall temperatures, so it overwrites them with constant values which will be assumed. Finally, we could implement algorithms in this function to represent sensor noise and resolution issues. This has not been done in this work, but remains a clear option.

7.2.3 Problems with two-parameter system

When implemented in the model, very large changes in the ψ values were observed for even minor changes. On investigation, the determinant of the matrix on the left-hand side of Equation 7.20, which contains the partial derivative estimates for the objectives, was found to be close to zero. As a result, inverting it was producing extremely large values.

This is due to the objective variables h_a and E_Δ being insufficiently independent. Since they are so closely related, the algorithm is attempting to solve for two variables with one input, resulting in significant errors.

7.2.4 Single-parameter method

Instead, we move to a method with a single parameter, ψ. We require that the valve constant be equal to some constant multiplied by the piston position, defining the

target of the fast controller as:

$$k_{\mathrm{HP,targ}} = \psi \cdot h_a \tag{7.21}$$

We need only run three simulations, denoted as:

(0) with $\psi = \psi_0$, the current value

(-1) with $\psi = (1 - \delta)\psi_0$

(1) with $\psi = (1 + \delta)\psi_0$

The update function becomes:

$$\begin{aligned} \frac{\partial h_a}{\partial \psi}\delta\psi &= -h_{a,(0)} \\ \Delta\psi &= -h_{a,(0)}\frac{2\delta\psi}{h_{a,(1)} - h_{a,(-1)}} \end{aligned} \tag{7.22}$$

An illustrative trajectory of k_{HP} and ψ is shown in Figure 7.3.

7.2.5 Results

Once implemented in the model, we obtain the trajectory shown in Figure 7.4.

7.3 Simulation duration

In order for the twin controller system to operate, the simulations in the slow controller must run in less time than the update interval. The time taken to run the slow controller is the sum of the time taken running the 'catch-up' simulations and the time taken running the 'projection' simulations (given by a triangular number formula). If we intend to run n_1 iterations of the slow controller, each of which runs

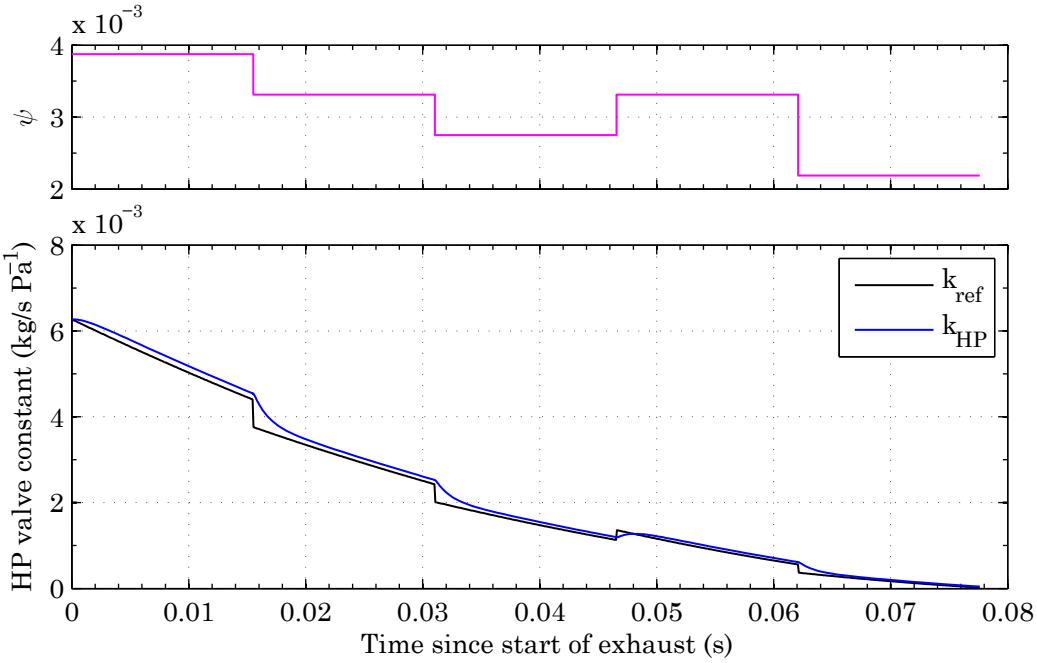


Figure 7.3: Illustrative k_{HP} and ψ trajectory from single-parameter method

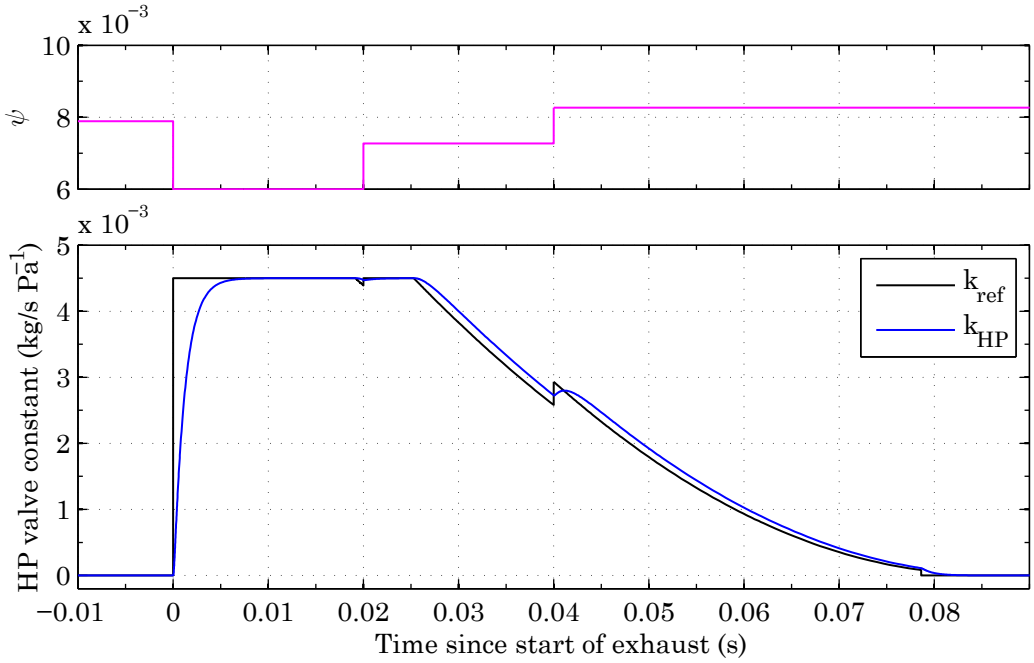


Figure 7.4: Actual k_{HP} and ψ trajectory from single-parameter method

m projection simulations in series, during a total exhaust duration of t_{exh}, the total simulated time is:

$$\begin{aligned} t_{\text{sim}} &= \left(\frac{n}{n+1} + \frac{m}{n+1} \frac{n(n+1)}{2} \right) t_{\text{exh}} \\ \frac{t_{\text{sim}}}{t_{\text{exh}}} &= \frac{n}{n+1} + \frac{mn}{2} \end{aligned} \tag{7.23}$$

We can now define a ratio of controller-simulated time to real time, $\dot{t}$; this is effectively a measure for how quickly the simulation can run. If $\dot{t} = 1$, for instance, this means that the simulation is running in realtime; it takes 1 second to simulate 1 second of operation. If this is larger than the limit defined above, then the simulation is workable;

$$\dot{t} > \frac{n}{n+1} + \frac{mn}{2} \tag{7.24}$$

For our model outlined above, $m = 3$. Specialised parallel-processing hardware with m cores would be able to run the projection simulations concurrently instead of consecutively, effectively reducing m to 1. These limits are shown in Figure 7.5, along with a horizontal dashed line at $\dot{t} = 0.9$, indicating the simulation speed obtained without optimisation on the author's desktop computer*.

For best performance of the exhaust valve, it was found that a minimum of 3 iterations of the slow controller were needed, requiring a simulation speed $\dot{t} > 3.33$ on a parallel-processing machine. It is likely that significant performance gains could be made using specialised control processors, so this factor of 3.7 speed increase is reasonable with further research and development.

*MATLAB R2013b on 64-bit Windows 7, with an Intel E8400 3 GHz processor and 4 GB RAM.

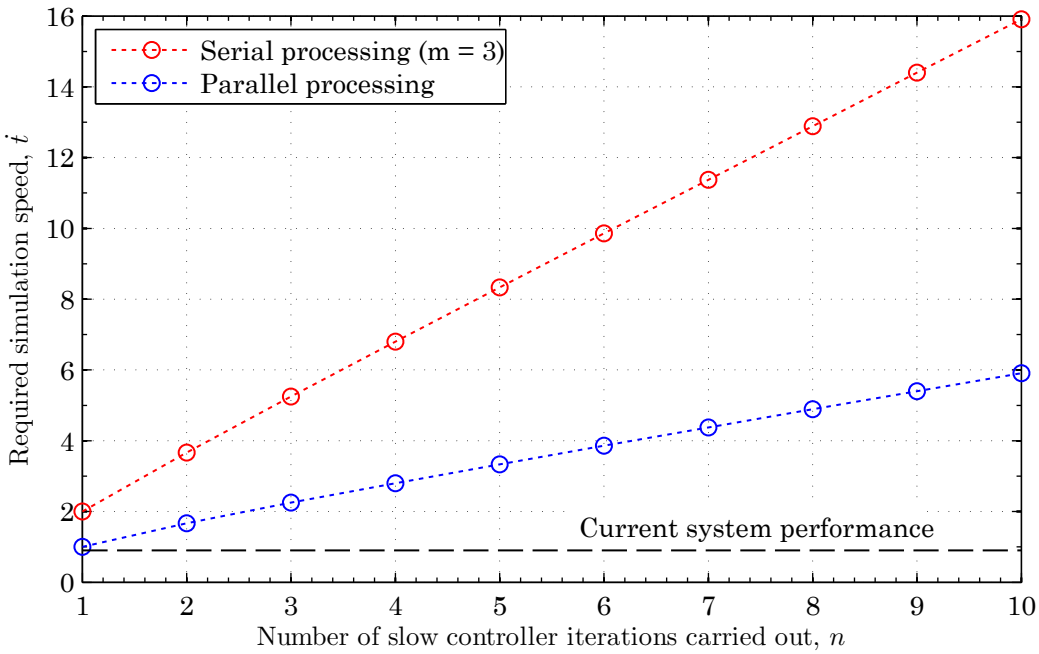


Figure 7.5: Simulation speed required for a variety of simulation counts

Chapter 8

System optimisation

In the final part of the project, we consider the possibility of inter-cycle control, in which we modify the parameters governing the 'kick' stage of the ICWT cycle.

The kick stage comes after the piston pair has been locked at one end of the compression tube following a stroke. When the compression tube reaches a an angle θ_{kick}, the HP valve connecting the compression chamber to the HP manifold is opened and HP air fills the compression chamber, pushing the piston pair towards the middle of the compression tube against centrifugal force. When the piston velocity has reached a velocity $\dot{h}_{\text{kick}}$, the HP valve is closed and the air in the compression tube allowed to expand until it reaches p_{atm}. At that point, the intake valve to the atmosphere is opened, and the departing piston draws in air ready for the next stroke.

There are two reasons to attempt to minimise the air used in the kick. The first is simply that the net air mass flow rate over many cycles is increased if we can avoid spending too much air on the kicking stroke. The second concerns the power fluctuations; the ICWT is producing compressed air on a repeating cycle, which could be viewed as a combination of a constant term and a sine wave. The constant term represents the real power produced, while the peaks and troughs of the sine

wave (due to the intermittent nature of the exhaust and kick stages) is the reactive power in the system. Since the piping and turbomachinery will be sized according to the instantaneous peak power output, we must aim to maximise the real power and minimise the reactive power, by reducing the amount of air passed back in the form of the kick.

A second competing objective is to control the power absorbed by the rotor. This is necessary to allow the ICWT to operate in varying wind conditions, since we must be able to tune the amount of power taken from the wind according to the quantity of power available to avoid overrunning the turbine.

The objectives for this chapter are:

- to quantify the possible performance changes from considerations of operating speed
- To outline a possible long-term control strategy for the model

8.1 Variables & targets

8.1.1 Controlled variables

The kick stage involves two control inputs, which we will alter to effect the desired changes in the targets. The first is the kick angle θ_{kick}, which is the angle at which the control system opens the HP exhaust valve behind the piston and starts to drive it towards the hub. If the kick begins too early in the cycle, while the compression tube is still close to horizontal, a large quantity of high-pressure air will be used before the piston pair reaches the required velocity to get to the end of the tube before the start of the next kick; if the kick comes too late in the cycle, the piston will not have time to fully complete its descent before the compression tube is horizontal again.

The second control input is the piston velocity at which the HP valve closes, $\dot{h}_{\text{kick}}$. High velocities waste compressed air, while too low a value will result in the piston taking too long to complete its stroke, reducing the amount of potential energy available to be converted. A very low $\dot{h}_{\text{kick}}$ might even fail to defeat centrifugal force, preventing the CoG of the piston pair from crossing through the axis.

It is clear that these two inputs are not wholly independent, so our aim will be to define a particular relationship between θ_{kick} and $\dot{h}_{\text{kick}}$. The control system will implement a particular kick parameter pair $[\dot{h}_{\text{kick}}, \theta_{\text{kick}}]$ which obeys that relationship to obtain the desired outcome in the dependent variables.

8.1.2 Dependent variables

Power fraction

A key concern is the 'verticality' of the piston stroke. If the kick stage starts too early or ends too late, or if the piston moves too slowly at the start, the piston will fail to take full advantage of the gravitational potential energy in the system, negatively affecting the ability of the system to absorb power at a given $\dot{\theta}$. We will measure the verticality of the piston stroke in terms of power.

Firstly, we will calculate an expression for the maximum energy that the system could produce. Consider a piston locked at the limit of its movement until the compression tube was vertical, then released to fall the full distance from the edge into the centre instantaneously. The work done in half a cycle is given by:

$$E_{\text{WD,max}} = 2\left(L_{\text{CT}} - L_{\text{TE}}\right) m_P g \tag{8.1}$$

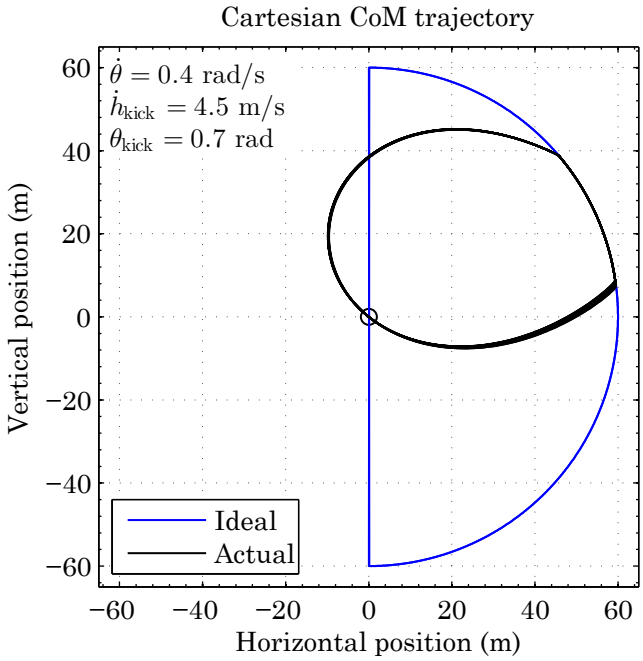


Figure 8.1: Cartesian position of piston pair centre-of-mass

Dividing this by the time taken gives an expression for the maximum average power:

$$\dot{E}_{\text{WD,max}} = 2\left(L_{\text{CT}} - L_{\text{TE}}\right) m_P g \frac{\pi}{\dot{\theta}} \tag{8.2}$$

The simulation model has a state variable which tracks the work done against gravitational torque, governed by Equation 3.30:

$$\dot{E}_{\text{WD}} = \frac{2h - L_{\text{CT}}}{2} \cos(\theta) m_P g \dot{\theta} \tag{8.3}$$

By dividing the average power throughout an individual simulation by the theoretical maximum power obtainable with those parameters, we obtain a normalised power, denoted $\dot{E}_\eta$.

To illustrate this, we run the model for 3 minutes of simulated time, with $\dot{\theta} = 0.4$ rad/s, $\dot{h}_{\text{kick}} = 4.5$ m/s and $\theta_{\text{kick}} = 0.7$ rad. The resulting Cartesian trajectory, in Figure 8.1, shows that the stroke starts off relatively close to the ideal case but

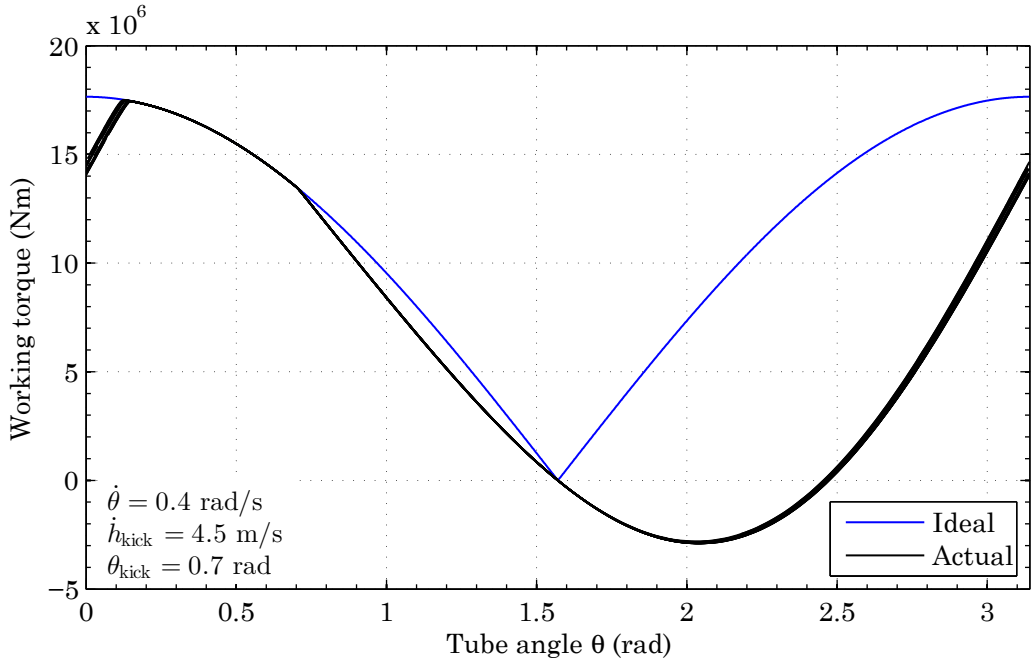


Figure 8.2: Working torque

diverges significantly by the time of the compression and exhaust strokes. It can also be seen that the piston pair CoM is on the left-hand side of the rotor hub for a noticeable period in each stroke, during which time the piston pair is actually adding energy back into the rotor through a positive gravity torque.

The effect of this is more clear in Figure 8.2, which plots the gravitational torque against the modulus of the tube angle with π (to show multiple cycles superimposed). The point discontinuity in the ideal cycle at $\theta = \frac{\pi}{2}$ is a result of the instantaneous stroke in its definition. The work done in a stroke is the integral of the torque with angle; the area between the two curves in the region $\frac{\pi}{2} < \theta < \pi$ is the difference between the average power per piston pair of 2.46 MW and the 4.50 MW maximum at that rotor speed, resulting in $\dot{E}_\eta = 0.55$. Closing this gap will be one objective.

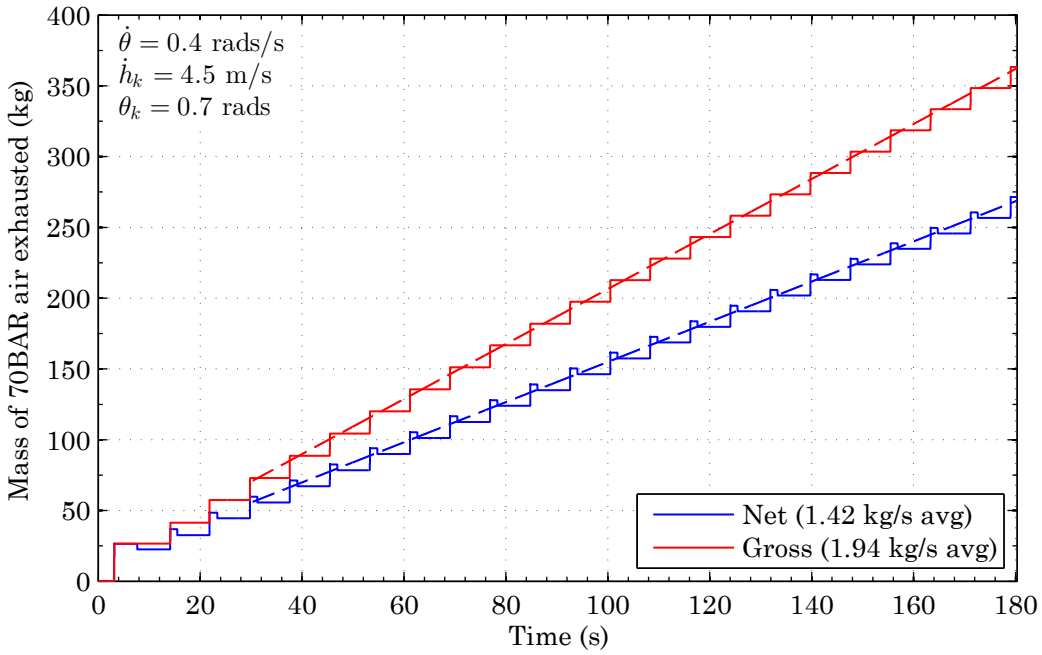


Figure 8.3: HP air exhaust rates

Net rate of air being exhausted

A second objective is to maximise the net rate of air being exhausted, $\dot{m}_{a\text{net}}$. Whether this is being expanded to provide electricity or being stored, it is important that we know the optimum kick parameters to maximise the rate at which pressurised air is being produced. The net rate (which takes account of reverse flow through the HP valve, mostly during the kick stage but also while the valve is closing after the exhaust stage) is trivially obtained by considering the initial and final values of the air exhausted state variable. We also track the gross rate $\dot{m}_{a\text{gross}}$ (which only counts positive outflow), to measure the ratio.

The two exhaust masses are shown in Figure 8.3. The dashed lines are a straight line fitted to the data after the warming-up period; the gradients of these give the rates as $\dot{m}_{a\text{net}} = 1.42$ kg/s and $\dot{m}_{a\text{gross}} = 1.94$ kg/s.

8.1.3 Rotor speed consideration

It is clear that the kick parameter pairs $(\dot{h}_{\text{kick}}, \theta_{\text{kick}})$ which obtain the maximum values of net exhaust rate $\dot{m}_{a\,\text{net}}$ and normalised power $\dot{E}_\eta$ will vary depending on rotor speed, since increasing centrifugal force will require a more energetic kick. The strategy will therefore also need to consider the rotor speed as a variable input.

8.2 Exploratory simulations

The next step is to quantify the field of possible kick parameter values, running the model with each set of parameters to obtain data on the resulting values of the dependent variables. To this end, the model's MATLAB script is converted into a function, which can be run inside a pair of nested loops in a container script to iterate through possible values of each kick parameter in turn. The container script exploits the fully-independent nature of the individual simulations to implement a very basic parallelisation technique, dividing the field into several segments which can each be run by a different computer and subsequently assembled into a single mesh.

We use simulations with a uniform duration of 180 seconds of simulated time, striking a balance between computer time and accuracy. It can be seen in Figure 8.3 that this timespan is enough to overcome the initial warm-up period, allowing time for transients to die down, and provide a sufficient period of regular operation for measuring the dependent variables.

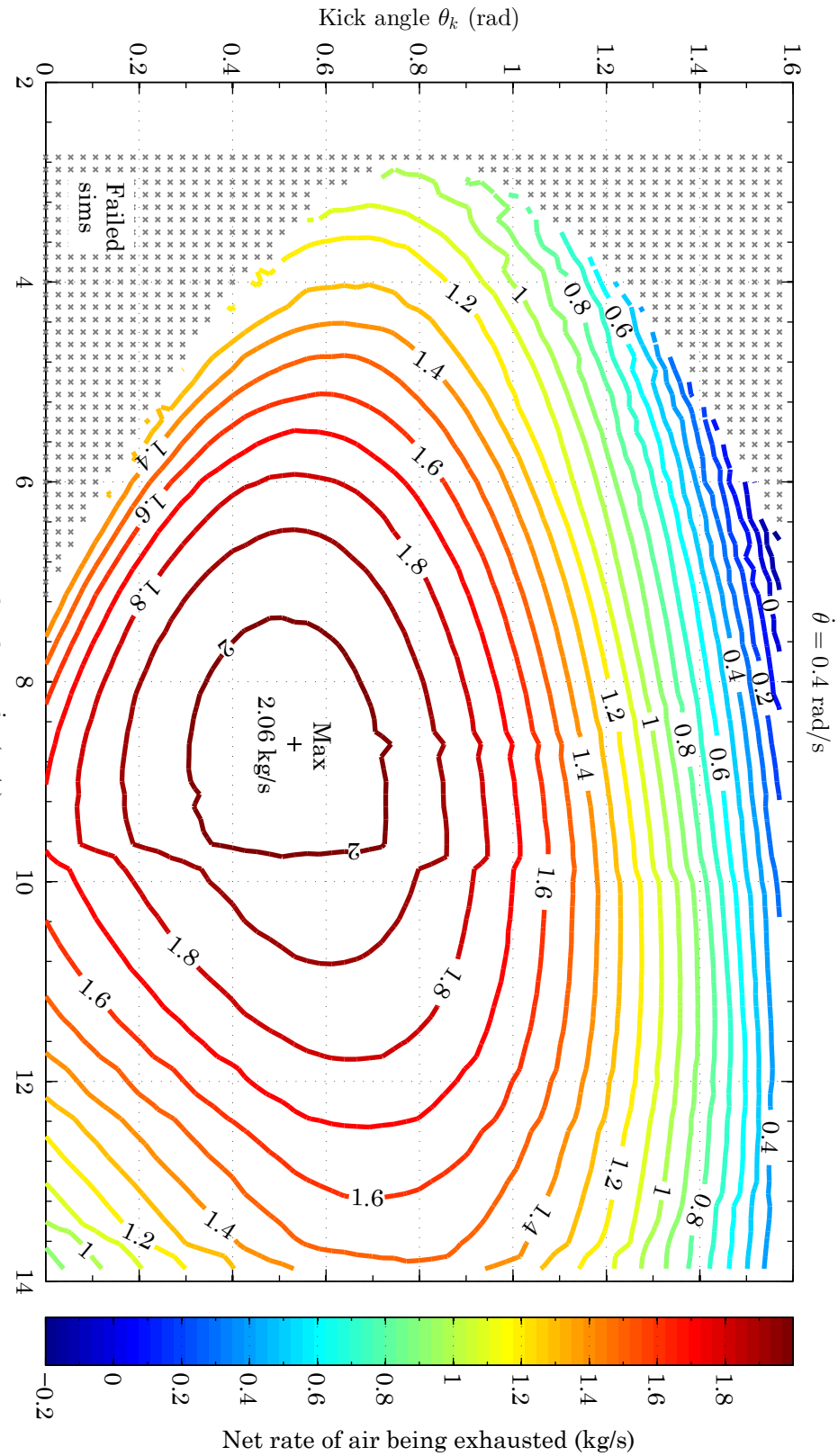


Figure 8.4: Surface of $\dot{m}_{a_{\text{net}}}$ values for $\dot{\theta} = 0.4$ rad/s

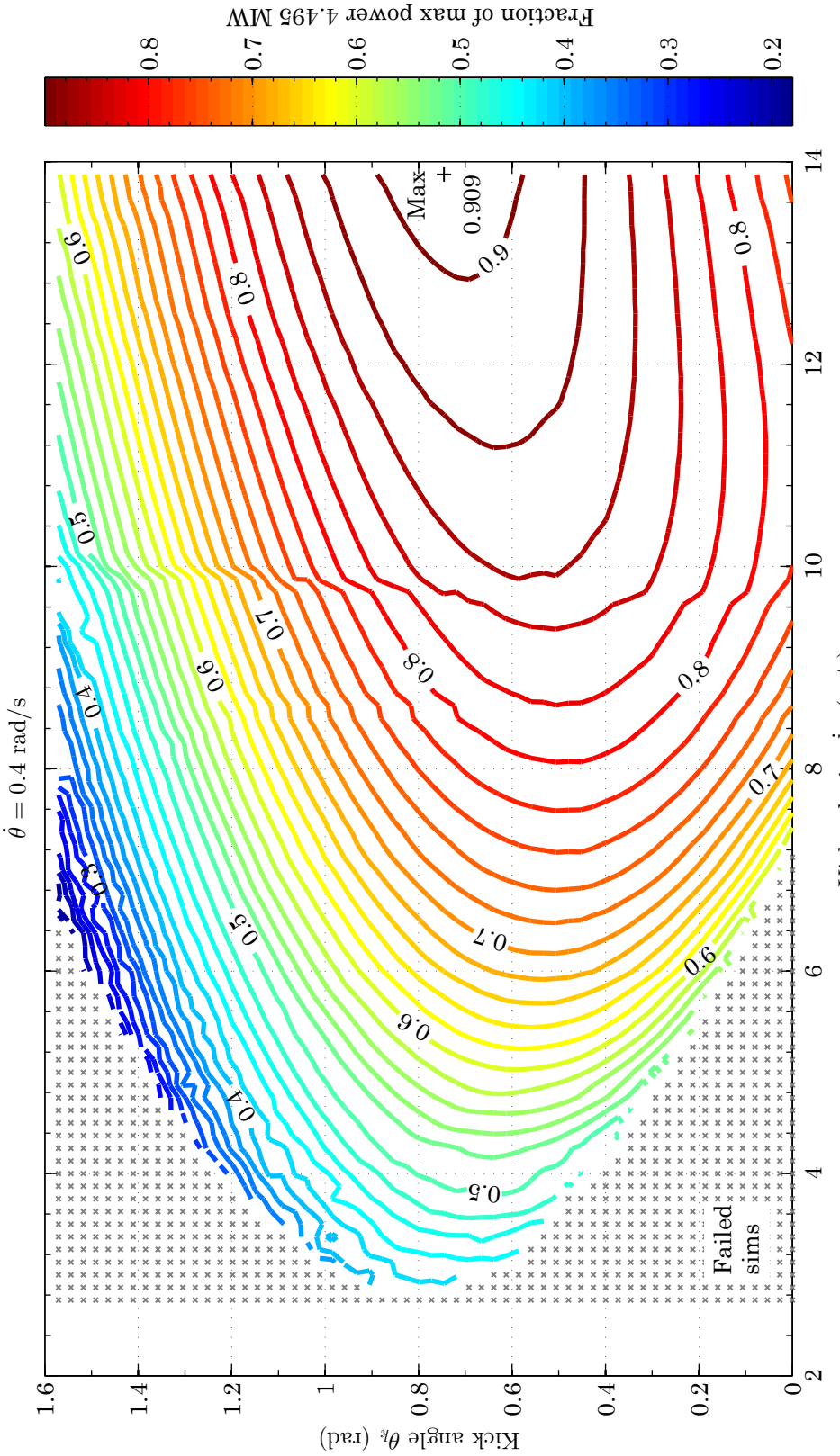


Figure 8.5: Surface of $\dot{E}_\eta$ values for $\dot{\theta} = 0.4$ rad/s

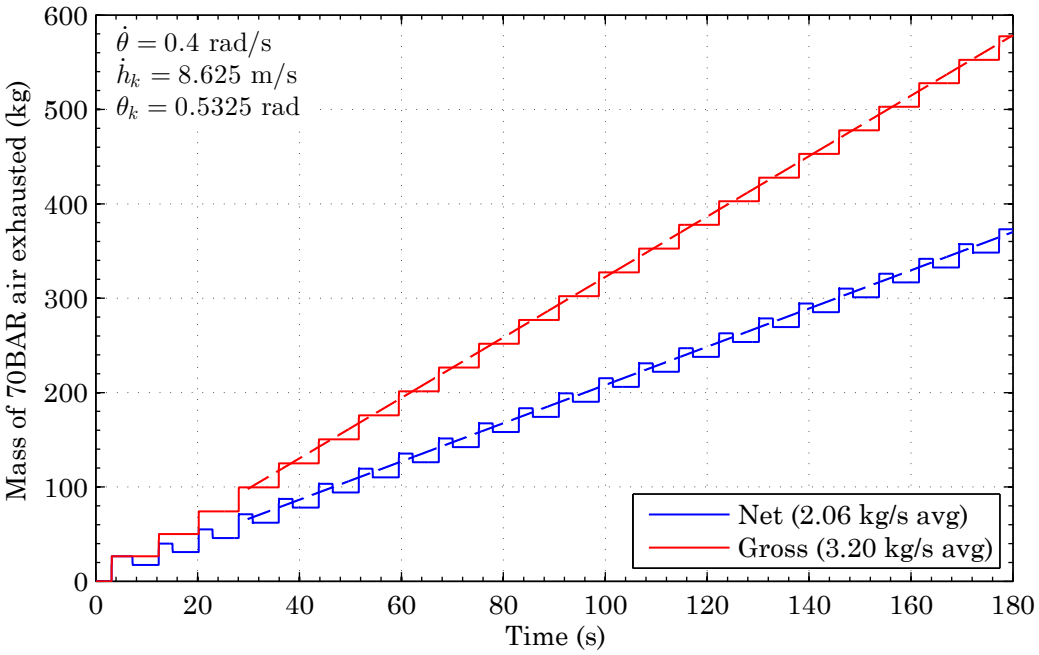


Figure 8.6: CoM trajectory at maximum $\dot{m}_{a\text{net}}$ for $\dot{\theta} = 0.4$ rad/s

8.2.1 Results

Single rotor speed

The first set of results, for rotor speed $\dot{\theta} = 0.4$ rad/s, are shown in Figure 8.4 and Figure 8.5. The grey markers at the left edge of the field are simulations which did not complete, due to the kick being insufficient to overcome the centrifugal force. The maxima for each surface are labelled.

Maxima

The maximum $\dot{m}_{a\text{net}}$ value observed was 2.0602 kg/s, at ($\dot{h}_{\text{kick}} = 8.625$ m/s, $\theta_{\text{kick}} = 0.5325$ rad), as shown in Figure 8.6. The maximum $\dot{E}_\eta$ was 0.9089, at ($\dot{h}_{\text{kick}} = 13.875$ m/s, $\theta_{\text{kick}} = 0.7455$ rad), with a trajectory and gravity torque as shown in Figure 8.7 and Figure 8.8 respectively.

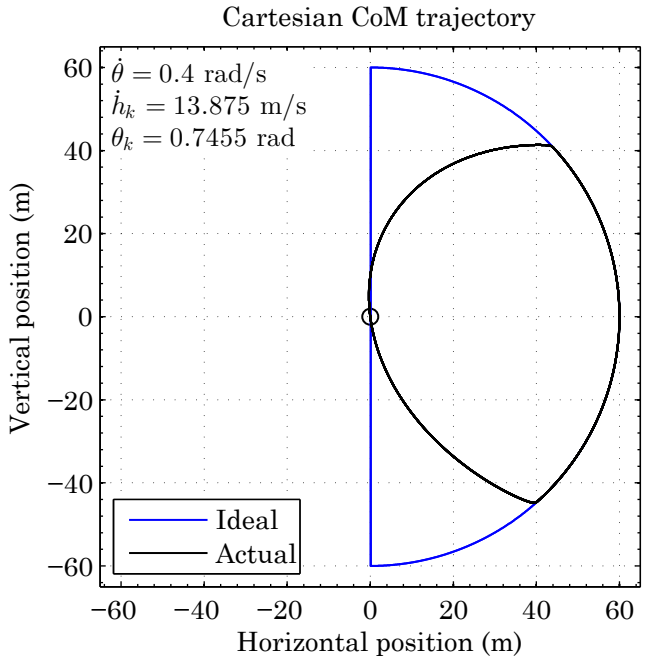


Figure 8.7: CoM trajectory at maximum $\dot{E}_\eta$ for $\dot{\theta} = 0.4$ rad/s

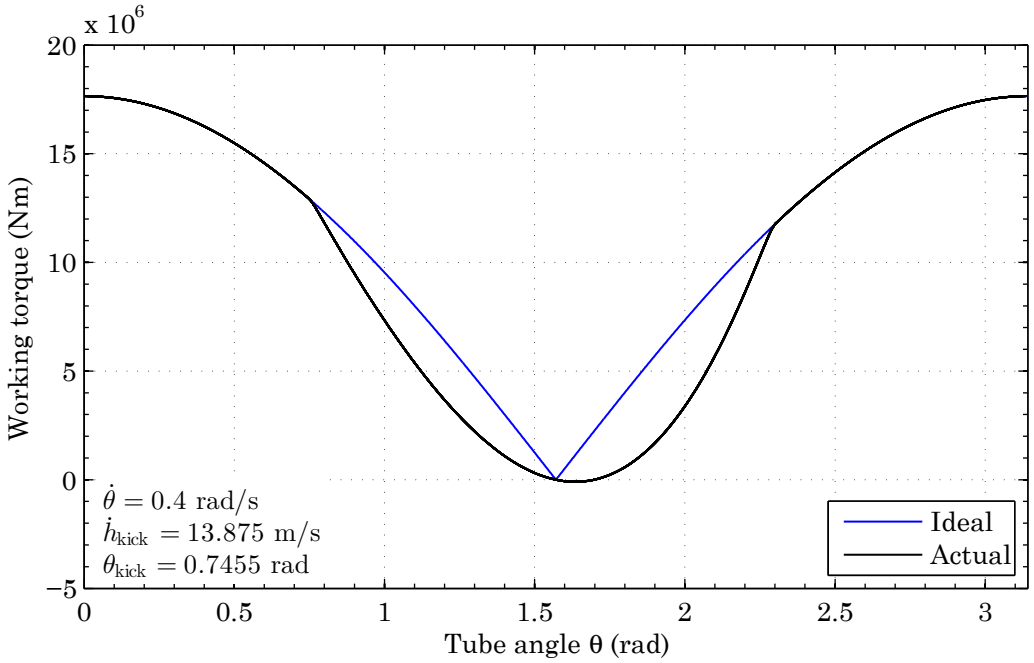


Figure 8.8: Working torque at maximum $\dot{E}_\eta$ for $\dot{\theta} = 0.4$ rad/s

Four rotor speeds

The next step is to extend this data to different rotor speeds. The resulting contour plots are shown in Figure 8.9 and Figure 8.10. The black lines represent the extent of the data gathered. In total, these graphs summarise around 1000 computer-hours of simulation.

It is clear from Figure 8.9 that $\dot{m}_{a\text{net}} < 0$ for $\dot{\theta} = 0.6$ rad/s, so we will discard that set of data and concentrate on the lower three rotor speeds.

8.3 Defining a control surface

We now define a control surface using the local maxima of each surface. First, a quadratic function $f_1(\dot{\theta})$ which returns $\dot{h}_{\text{kick}}$ is obtained from the three points representing the maximum $\dot{m}_{a\text{net}}$ for each rotor speed tested; then, a second quadratic is likewise obtained as $\theta_{\text{kick}} = f_2(\dot{h}_{\text{kick}})$, as shown in Figure 8.11. This set of relations represents one possible solution for the control strategy, defining a pair of kick parameters for every rotor speed to maximise $\dot{m}_{a\text{net}}$. The combined curve is shown in black in Figure 8.12, passing through the maxima at each rotor speed level.

The same technique is repeated on the maximum $\dot{E}_\eta$ points. Each pair of points is then connected with a straight line, representing the affine combination of the two curves, as seen in Figure 8.13.

From inspection of Figure 8.10, it can be seen that the locations of maximum $\dot{E}_\eta$ are always at the edge of the field corresponding to the highest value of $\dot{h}_{\text{kick}}$ investigated. This is due to the definition of $\dot{E}_\eta$; since the ideal cycle assumes a stroke of zero duration, increasing kick velocity will result in the power fraction asymptotically approaching unity. Therefore, we will extend the control surface past the recorded maximum $\dot{E}_\eta$ points.

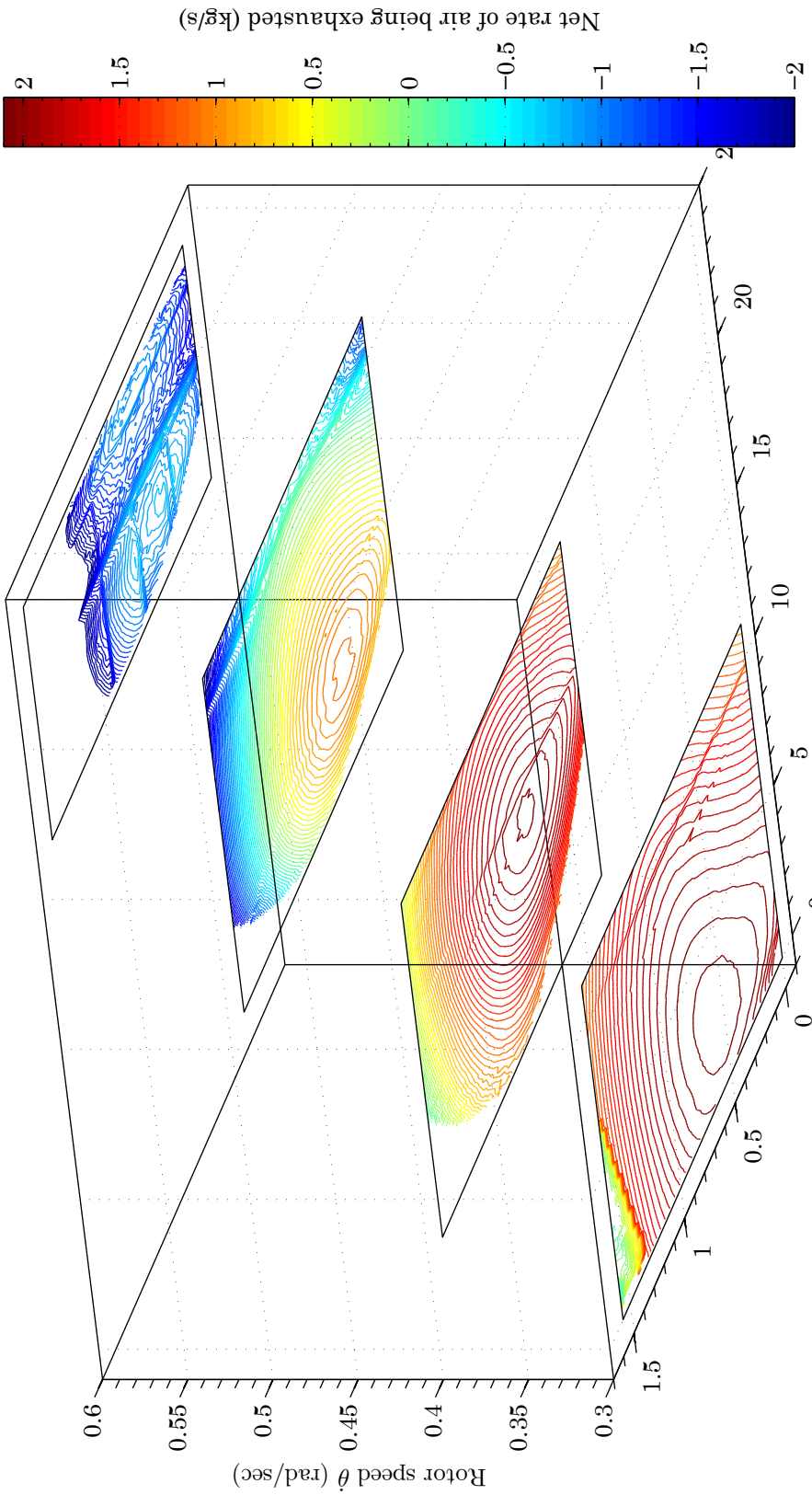


Figure 8.9: Surfaces of $\dot{m}_{a_{\text{net}}}$ values for multiple $\dot{\theta}$ values

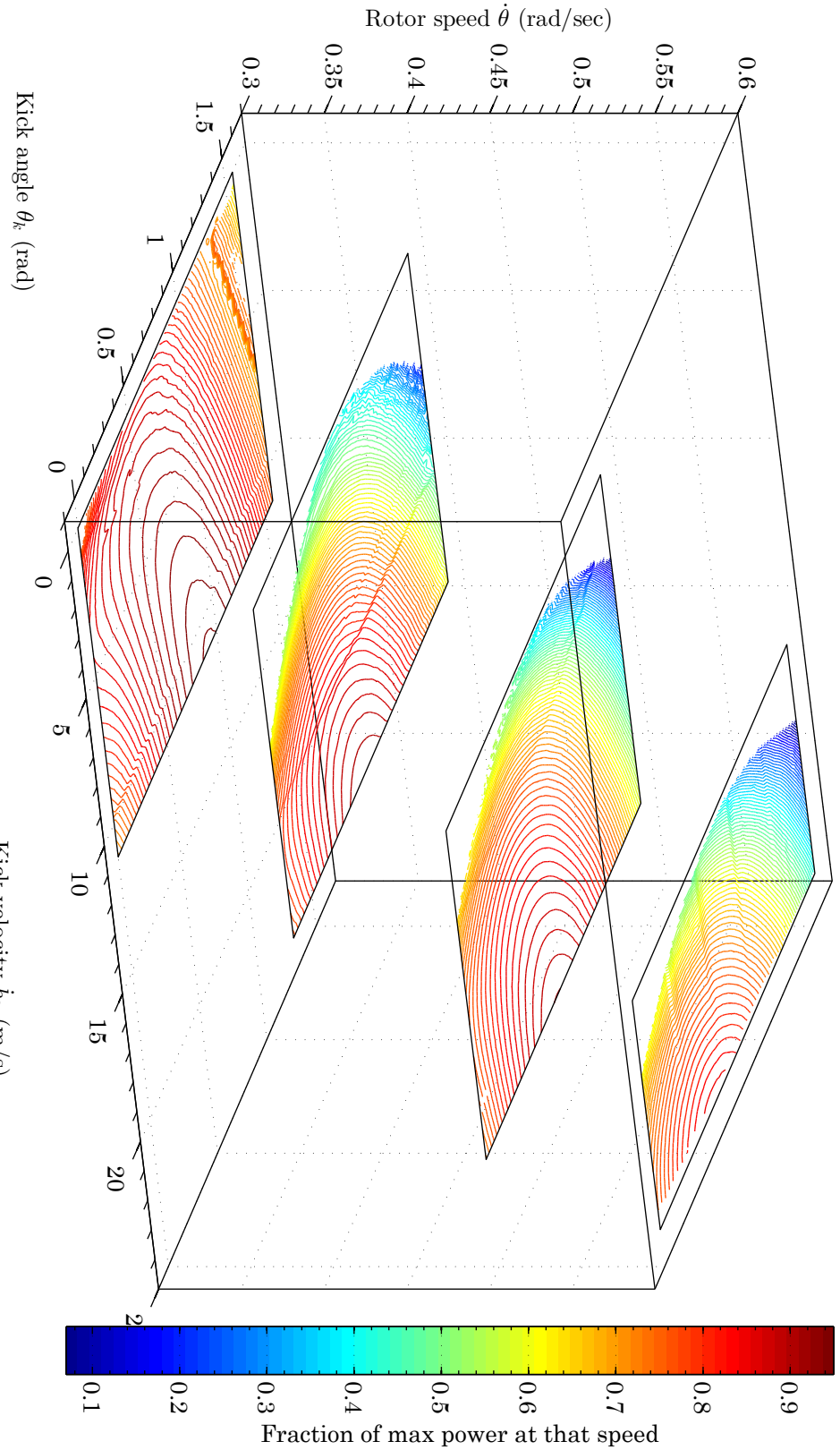


Figure 8.10: Surfaces of $\dot{E}_\eta$ values for multiple $\dot{\theta}$ values

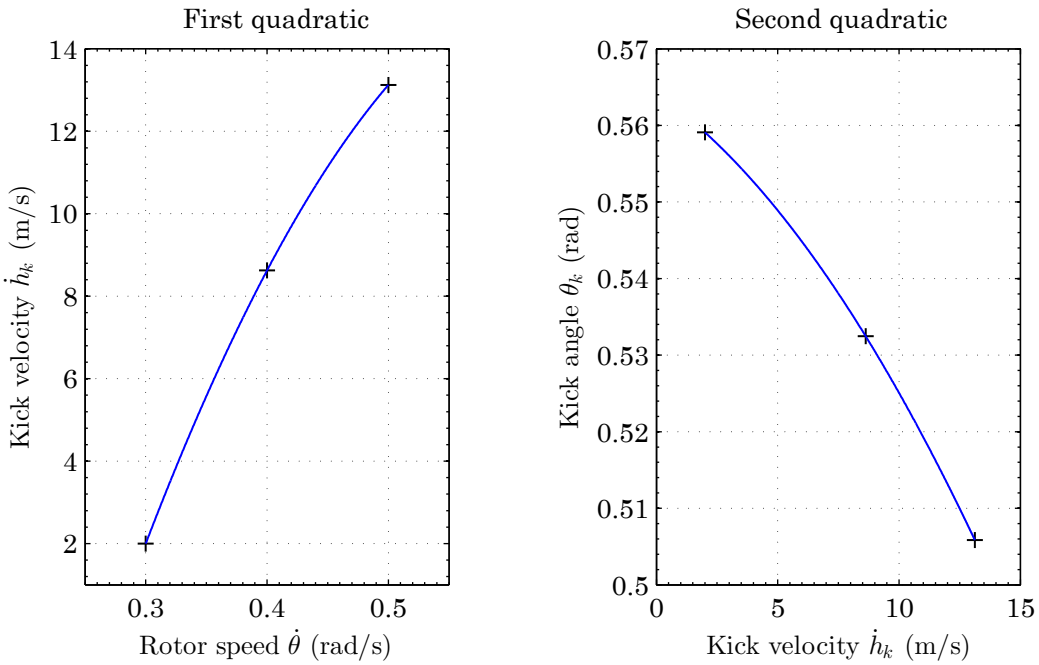


Figure 8.11: Quadratic curves through maximum $\dot{m}_{a\text{net}}$ values

8.3.1 Control surface results

The final step is to explore the values of the dependent variables on the control surface itself. A grid of values is drawn up, as shown in Figure 8.14, and every point is simulated as before, taking a further 300 computer-hours. The contours are then mapped onto the control surface and plotted, as shown in Figure 8.15 and Figure 8.16.

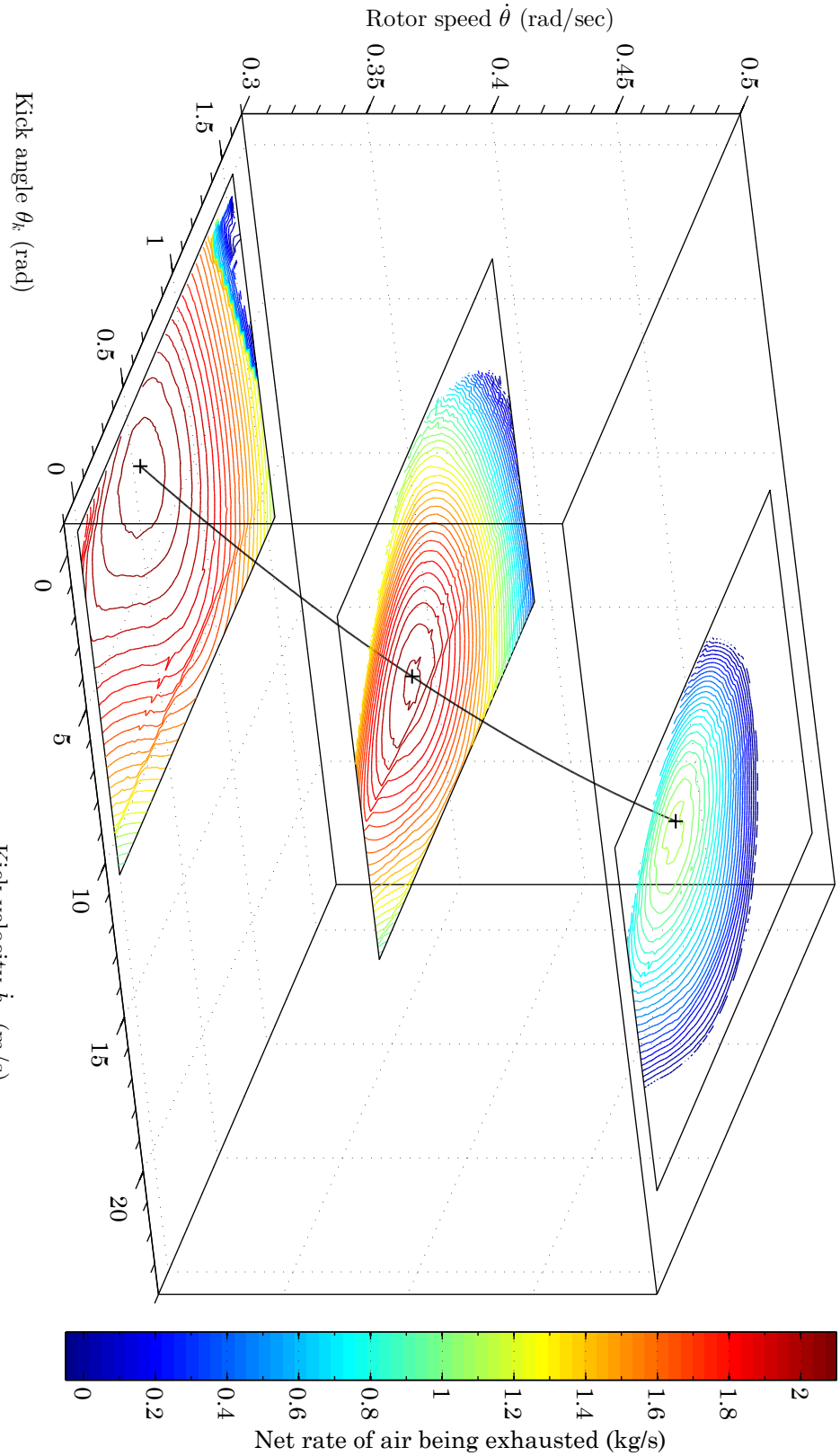


Figure 8.12: 3D curve described by maximum $\dot{m}_{a\,\text{net}}$ values

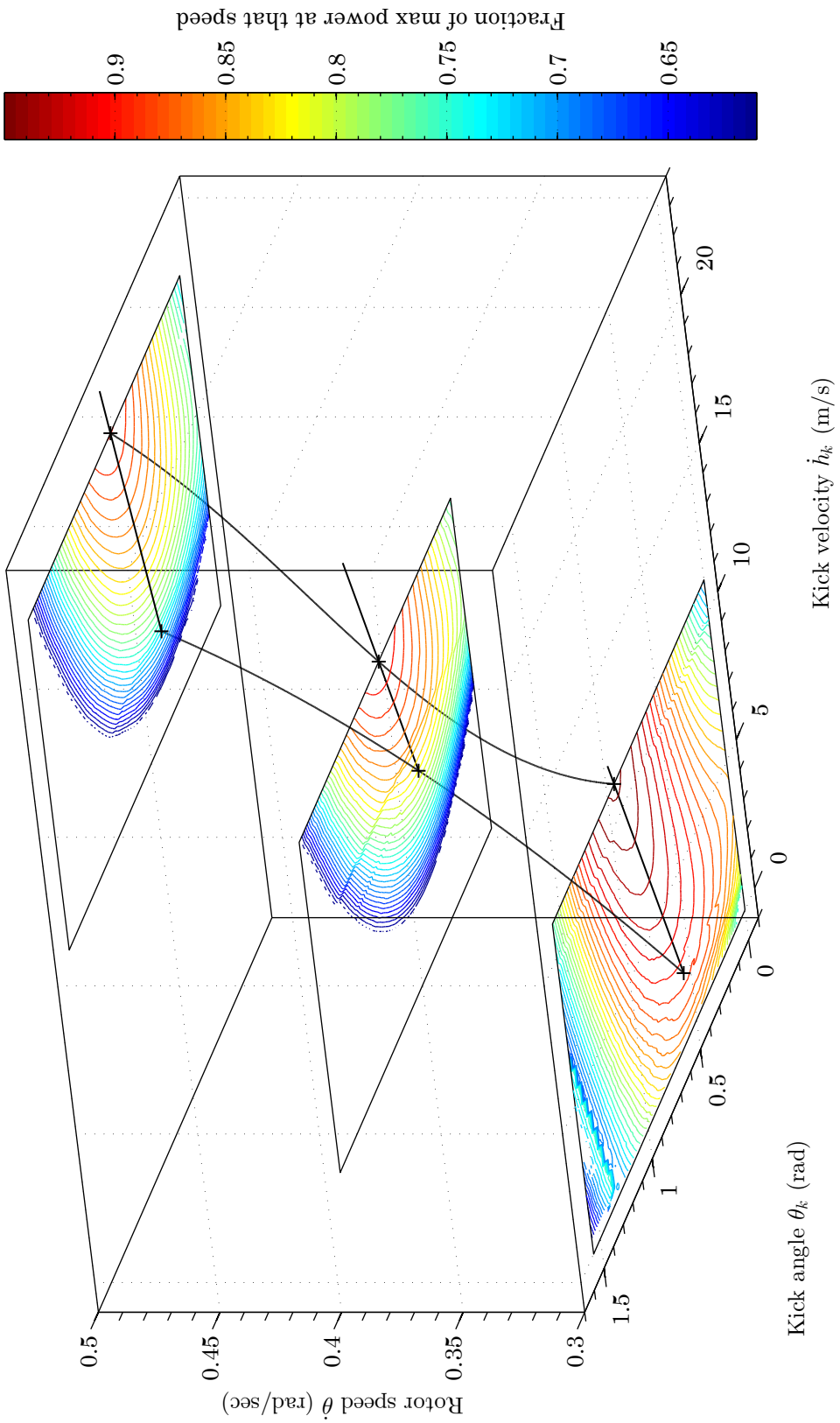


Figure 8.13: Definition of optimum surface

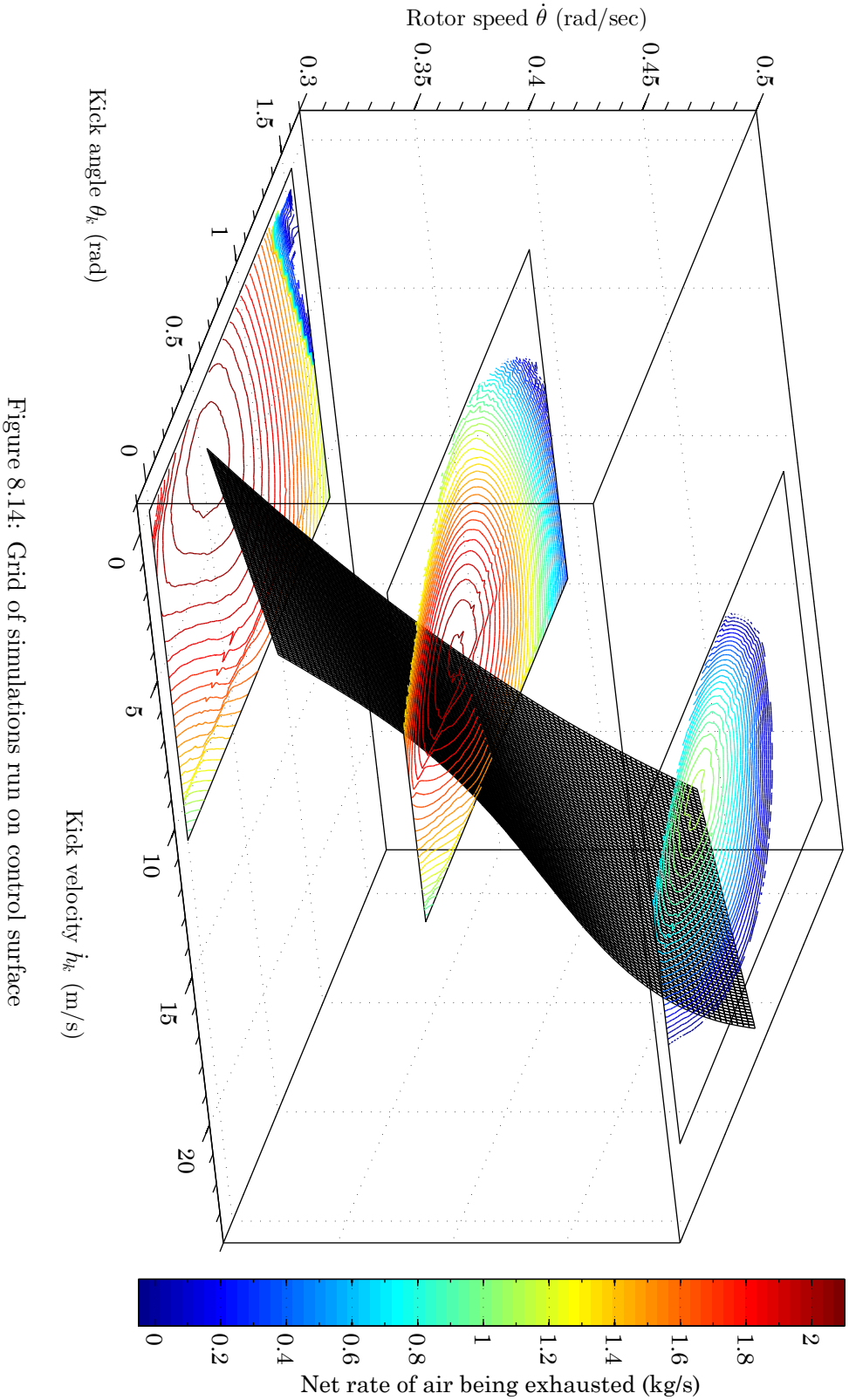


Figure 8.14: Grid of simulations run on control surface

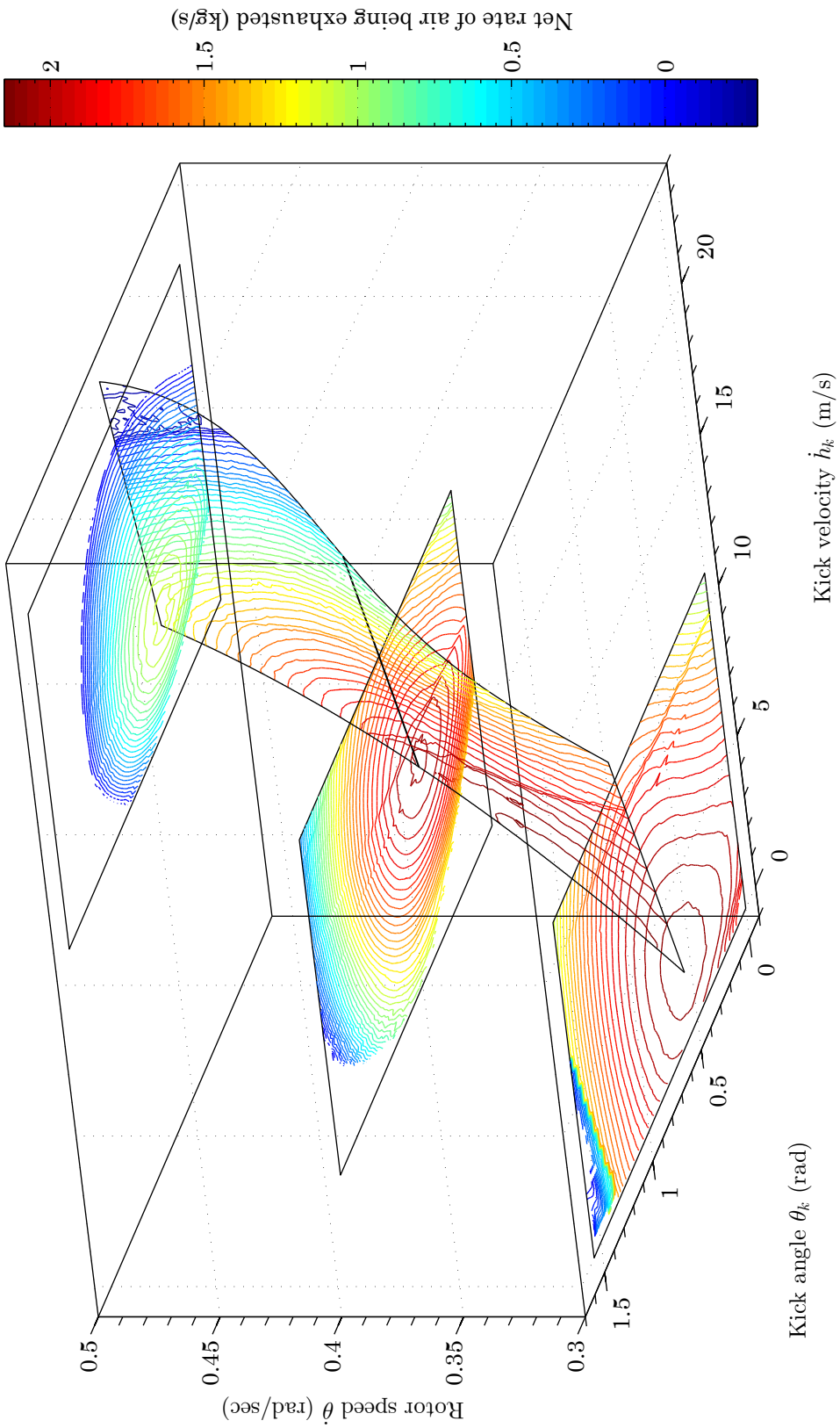


Figure 8.15: Control surface $\dot{m}_{a\text{net}}$ values in 3D

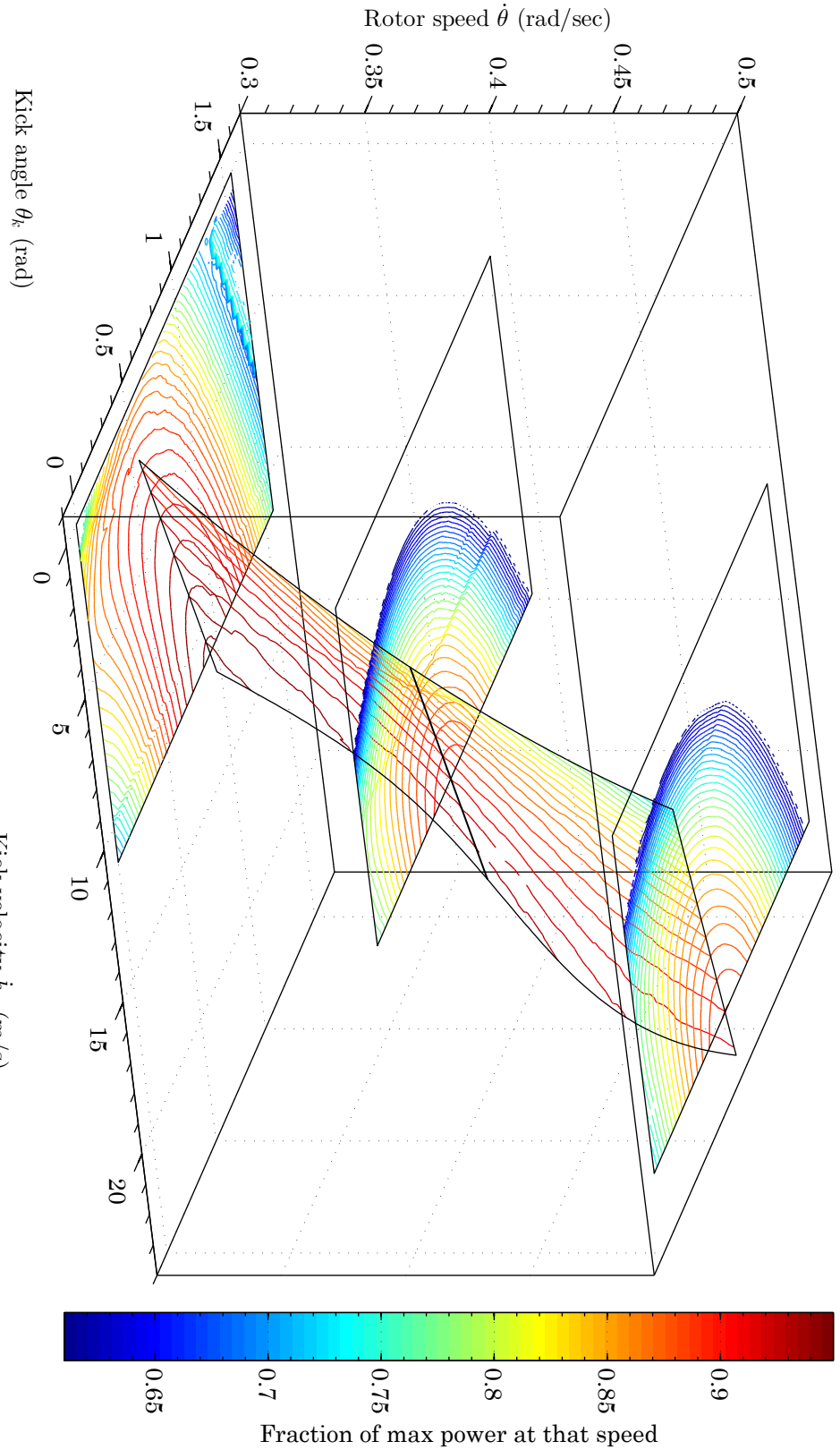


Figure 8.16: Control surface $\dot{E}_\eta$ values in 3D

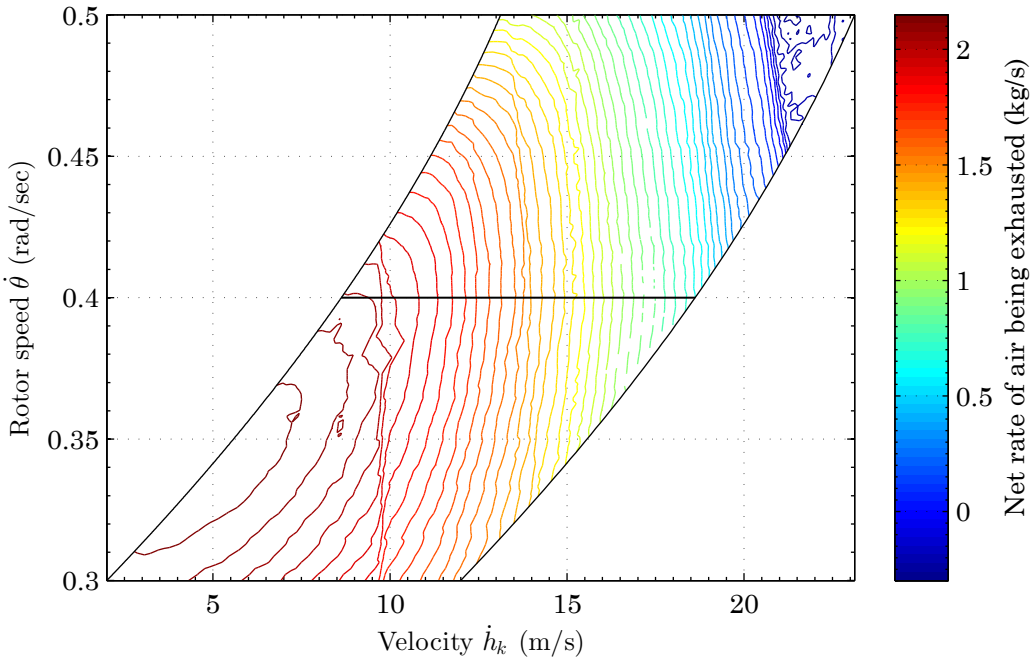


Figure 8.17: Net airflow rate $\dot{m}_{a_{\text{net}}}$ for optimum surface

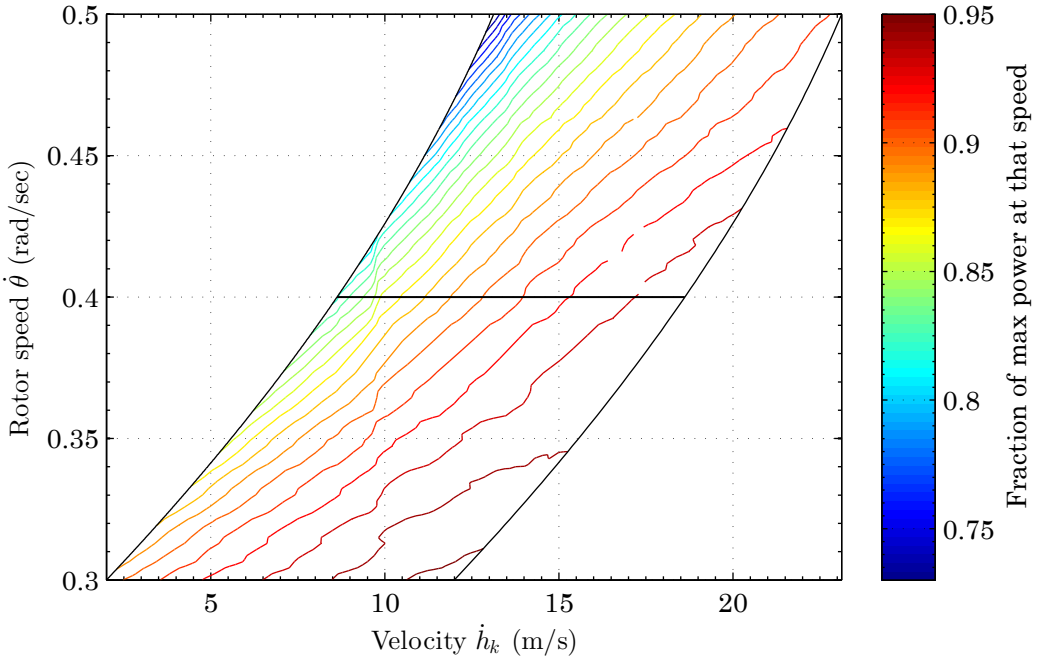


Figure 8.18: Normalised power $\dot{E}_\eta$ for optimum surface

8.4 Conclusions

It can be seen in Figure 8.17 that the overall peak $\dot{m}_{a\text{net}}$ is 2.3 kg/s, obtained at a rotor speed of 0.34 rad/s and a kick velocity of 5 m/s. From the contours, we can also state that $\dot{m}_{a\text{net}}$ is primarily dependent on kick velocity, and is relatively independent of rotor speed for $\dot{h}_{\text{kick}} > 10$ m/s.

While $\dot{m}_{a\text{net}}$ appears to be inversely proportional to $\dot{h}_{\text{kick}}$, obtaining a high normalised power value $\dot{E}_\eta$ is significantly dependent on achieving both a low rotor speed and a high kick velocity; this verifies our assumptions about the 'optimum' kick verticality.

Inter-cycle control, which would vary these parameters to select a particular desired performance, is clearly possible and desirable. A clear future direction of research would be to consider how external variables such as wind speeds, grid electricity prices or storage conditions might require alternative $\dot{m}_{a\text{net}}$ or $\dot{E}_\eta$ outputs, and design a control structure intended to allow the ICWT to alter its settings to seek the new optimum performance.

Finally, we find that it is critically important to achieve low rotor speed to avoid reductions in both $\dot{m}_{a\text{net}}$ and $\dot{E}_\eta$. Peak $\dot{m}_{a\text{net}}$ for the reference turbine is 0.34 rad/s; since this differs from the optimum rotor speed as governed by wind rotor theory, which was set as 0.4 rad/s in Figure 3.1, we can conclude that the reference turbine is below the optimum rotor diameter for the ICWT concept. Larger rotors, turning more slowly, would result in a better fit between the aerodynamic and compression optimum rotor speeds.

Chapter 9

Conclusions and future work

9.1 Contributions of present work

This thesis has presented a detailed simulation of an integral compression wind turbine (ICWT).

Starting from basic non-rotating models, the simulation was extended to include the capacity for mass flow of air both from the atmosphere into the compression cylinder and from the cylinder into a high-pressure manifold for storage. A set of algorithms were derived for calculating both the instantaneous potential energy in the system were formed, partly based on a predicted angle at the end of the exhaust stroke, and also the energy required to compress and exhaust the air below the piston. MATLAB solver events were implemented to switch between different compression stages without discontinuities, allowing the accurate simulation of the model despite its nature as a highly-stiff system of ODEs.

The temperature profile along the compression cylinder wall was then added to the simulation, modelled as a series of orthogonal polynomials generated using the Gram-Schmidt orthogonalisation process. Heat flow into and out of the cylinder

wall was simulated, allowing the wall temperatures and the air mass temperatures to interact as part of the system of ODEs. A steady-state wall temperature profile was found, using a modified Newton-Raphson process in conjunction with the results of a long-term simulation, based on linear combinations of exponential terms.

The equations governing a water-cooling mechanism were derived, simulating the effects of spraying a fine mist of water droplets into the compression cylinder immediately before the compression stroke. Algorithms to track the change of state of the water were developed and implemented in the larger model, showing the reduction in exhaust temperature which could be achieved.

Strategies for exhaust valve control were outlined, starting with a relatively simple exponential proportional control mechanism based on energy surplus. Next, a more complex twin-controller strategy was described, with an inner proportional controller seeking targets set by a model-based control system. Although successful, constraints of computing power prevented it from being physically realisable on present hardware.

Finally, an optimisation study was carried out, looking at the optimum parameters for the turbine's 'kick' stage. Relevant objective variables of net air exhaust rate and rotor power fraction were devised, and a large-scale search space was outlined and investigated over 1000 computer-hours. Using this insight into the behaviour of the system, an optimum operating surface was identified, representing the best trade-off between energy produced and energy harvested; further investigation into the objective variables on that surface allowed a general inter-cycle control technique to be outlined.

9.2 Future work

Several areas of future research are envisaged as progressions of the work in this thesis.

Control modelling and implementation

The implementation of the inter-cycle sliding-mode control system described in Chapter 8 would represent a clear next step, providing the necessary control to ensure efficient operation of a practical ICWT.

Another area of investigation would be the potential for the use of bistable valves and pulse-width modulation to approximate valve constants.

Variable rotor speed

Appendix C calculated that the effect on the rotor of the moving masses was relatively minor and thus assumed constant rotor speed. An improved simulation would certainly model rotor speed as a variable, and additionally model some degree of wind variablility, which would significantly increase verisimilitude. Additionally, the tracking of spanwise forces in the rotor plane would allow a degree of simple finite element analysis to model the in-plane deflections of the rotor blades.

One particular detail of interest, which would be a natural study area for a variable rotor speed model, is the problem of starting an ICWT rotor from stationary, using air from the HP manifold to move the pistons and thus start the rotor turning.

Rotor axis movement

That model could be extended by the addition of five further axes to provide a full three-dimensional mass-spring-damper system modelling translation and rotation of the rotor axis in any direction. Wind forces, tower shadow effects, and yawing & feathering processes could all modelled and their effect on the piston dynamics observed.

Double-piston sim

Another useful addition to the model would be the variables to track the two paired pistons separately, including spring and damping forces between them from the tie rod.

Sensor noise

Adding a simulation of sensor inaccuracy to `SW_sensors` would allow the investigation of the effects of sensor quantisation, noise and bias on the efficient operation of the system.

References

[1] Rajendra K. Pachauri, Andy Reisinger, et al. Climate Change 2007: Synthesis Report. Contribution of working groups I, II and III to the fourth assessment report of the Intergovernmental Panel on Climate Change. Technical report, Intergovernmental Panel on Climate Change, 2007. URL https://www.ipcc.ch/report/ar4/syr/.

[2] Ottmar Edenhofer, Ramón Pichs-Madruga, Youba Sokona, Kristin Seyboth, Susanne Kadner, Timm Zwickel, Patrick Eickemeier, Gerrit Hansen, Steffen Schlömer, Christoph von Stechow, et al. *Renewable Energy Sources and Climate Change Mitigation: Special report of the Intergovernmental Panel on Climate Change*. Cambridge University Press, 2011. URL http://ipcc.ch/report/srren/.

[3] European Union Committee. The EU's target for renewable energy: 20% by 2020. Technical Report 175-I, UK House of Lords, October 2008. URL http://www.publications.parliament.uk/pa/ld200708/ldselect/ldeucom/175/175.pdf.

[4] Electricity: chapter 5, Digest of United Kingdom energy statistics (DUKES). Online, July 2014. URL https://www.gov.uk/government/statistics/electricity-chapter-5-digest-of-united-kingdom-energy-statistics-dukes.

[5] Paraic Higgins and Aoife M. Foley. Review of offshore wind power development in the United Kingdom. In *12th International Conference on Environment and Electrical Engineering (EEEIC)*, pages 589–593. IEEE, 2013. doi: 10.1109/EEEIC.2013.6549584.

[6] Goran Strbac. Quantifying the system costs of additional renewables in 2020. Technical report, ILEX Energy Consulting & Manchester Centre for Electrical Energy, UMIST, October 2002. URL http://webarchive.nationalarchives.gov.uk/+/http://www.berr.gov.uk/files/file21352.pdf.

[7] Richard Slark and Gareth Davies. Compliance costs for meeting the 20% renewable energy target in 2020. Technical report, Pöyry Energy Consulting for the UK Department for Business, Enterprise and Regulatory Reform, March 2008. URL https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/42974/1_20090501132011_e____Compliancecosts.pdf.

[8] Albert Betz. Das maximum der theoretisch möglichen ausnutzung des windes durch windmotoren. *Zeitschrift fr das gesamte Turbinenwesen*, 26:307–309, 1920.

[9] Tony Burton, Nick Jenkins, David Sharpe, and Ervin Bossanyi. *Wind Energy Handbook*. John Wiley & Sons, 2nd edition, 2011. doi: 10.1002/9781119992714.

[10] Erich Hau and Horst Von Renouard. *Wind Turbines: Fundamentals, Technologies, Application, Economics*. Springer, 3rd edition, 2013. doi: 10.1007/978-3-642-27151-9.

[11] Renewables Advisory Board. Value breakdown for the offshore wind sector. Technical report, Department of Energy and Climate Change, UK Government, February 2010. URL `https://www.gov.uk/government/publications/offshore-wind-sector-value-breakdown`.

[12] Lee Jay Fingersh, M Maureen Hand, and Alan S Laxson. Wind turbine design cost and scaling model. Technical Report NREL/TP-500-40566, National Renewable Energy Laboratory, USA, December 2006. URL `http://www.nrel.gov/wind/pdfs/40566.pdf`.

[13] Andrew Cordle. Evaluating the optimal lifetime design of different support structure concepts under the loads of larger turbines. Conference presentation, 'Future Offshore Foundations 2013', October 2013. URL `http://www.windpowermonthlyevents.com/events/future-offshore-wind-foundations-2013/agenda-7/`.

[14] Seamus D. Garvey. Structural capacity and the 20MW wind turbine. *Proceedings of the Institution of Mechanical Engineers, Part A: Journal of Power and Energy*, 224(8):1083–1115, December 2010. doi: 10.1243/09576509JPE973.

[15] Peter Jamieson. *Innovation in wind turbine design*. John Wiley & Sons, 2011. doi: 10.1002/9781119975441.

[16] Francisco Díaz-González, Andreas Sumper, Oriol Gomis-Bellmunt, and Roberto Villafáfila-Robles. A review of energy storage technologies for wind power applications. *Renewable and Sustainable Energy Reviews*, 16(4):2154–2171, 2012. doi: 10.1016/j.rser.2012.01.029.

[17] File:wind turbine 1888 charles brush.jpg. Wikimedia Commons, 2014. URL `http://commons.wikimedia.org/wiki/File:Wind_turbine_1888_Charles_Brush.jpg`. Public domain.

[18] David A. Spera, editor. *Wind Turbine Technology*. ASME Press, 1994.

[19] Christian Kjaer, Bruce Douglas, Raffaella Bianchin, and Elke Zander. Wind energy - the facts. Online, October 2008. URL `http://www.ewea.org/fileadmin/ewea_documents/documents/publications/WETF/1565_ExSum_ENG.pdf`.

[20] David J. C. MacKay. *Sustainable Energy - without the hot air*. UIT Cambridge, 2008. ISBN 978-1-906860-01-1. doi: 10.1119/1.3273852. URL www.withouthotair.com.

[21] Marc Beaudin, Hamidreza Zareipour, Anthony Schellenberglabe, and William Rosehart. Energy storage for mitigating the variability of renewable electricity sources: An updated review. *Energy for Sustainable Development*, 14(4):302–314, 2010. ISSN 0973-0826. doi: 10.1016/j.esd.2010.09.007.

[22] Pavlos S. Georgilakis. Technical challenges associated with the integration of wind power into power systems. *Renewable and Sustainable Energy Reviews*, 12 (3):852–863, 2008. ISSN 1364-0321. doi: 10.1016/j.rser.2006.10.007.

[23] Robert T. Watson, Marufu C. Zinyowera, and Richard H. Moss, editors. *Technologies, Policies and Measures for Mitigating Climate Change*. IPCC, November 1996. URL http://www.ipcc.ch/pdf/technical-papers/paper-I-en.pdf.

[24] Damian Carrington. BritNed power cable boosts hopes for European supergrid. Online, April 2011. URL http://www.theguardian.com/environment/2011/apr/11/uk-netherlands-power-cable-britned.

[25] R.K. Edmunds, T.T. Cockerill, T.J. Foxon, D.B. Ingham, and M. Pourkashanian. Technical benefits of energy storage and electricity interconnections in future British power systems. *Energy*, 70(0):577–587, 2014. ISSN 0360-5442. doi: 10.1016/j.energy.2014.04.041.

[26] National Grid Interconnector Positioning Statement. Online, March 2014. URL http://www2.nationalgrid.com/About-us/European-business-development/Interconnectors/.

[27] Peter Warren. A review of demand-side management policy in the UK. *Renewable and Sustainable Energy Reviews*, 29(0):941–951, 2014. ISSN 1364-0321. doi: 10.1016/j.rser.2013.09.009.

[28] Laurent Pouret, Nigel Buttery, and W.J. Nuttall. Is nuclear power inflexible? *Nuclear Future*, 5(6):333–340, 2009. URL http://www.nuclearinst.com/CoreCode/Admin/ContentManagement/MediaHub/Assets/FileDownload.ashx?fid=74608&pid=13034&loc=en-GB&fd=False.

[29] Electricity Market Reform: policy overview. Online, November 2012. URL https://www.gov.uk/government/publications/electricity-market-reform-policy-overview--2.

[30] Goran Strbac. Demand side management: Benefits and challenges. *Energy Policy*, 36(12):4419–4426, 2008. ISSN 0301-4215. doi: 10.1016/j.enpol.2008.09.030. Foresight Sustainable Energy Management and the Built Environment Project.

[31] Nancy Skinner. CA AB-2514 Energy storage systems. Online, September 2010. URL http://leginfo.legislature.ca.gov/faces/billNavClient.xhtml?bill_id=200920100AB2514.

[32] Bjrn Bolund, Hans Bernhoff, and Mats Leijon. Flywheel energy and power storage systems. *Renewable and Sustainable Energy Reviews*, 11(2):235–258, 2007. ISSN 1364-0321. doi: 10.1016/j.rser.2005.01.004.

[33] Douglas Cross and Chris Brockbank. Mechanical hybrid system comprising a flywheel and CVT for motorsport and mainstream automotive applications. Technical report, SAE Technical Paper, 2009.

[34] R. Sebastin and R. Pea Alzola. Flywheel energy storage systems: Review and simulation for an isolated wind power system. *Renewable and Sustainable Energy Reviews*, 16(9):6803–6813, 2012. ISSN 1364-0321. doi: 10.1016/j.rser.2012.08.008.

[35] H. Ibrahim, A. Ilinca, and J. Perron. Energy storage systems - characteristics and comparisons. *Renewable and Sustainable Energy Reviews*, 12(5):1221–1250, 2008. ISSN 1364-0321. doi: 10.1016/j.rser.2007.01.023.

[36] Cyrus Wadia, Paul Albertus, and Venkat Srinivasan. Resource constraints on the battery energy storage potential for grid and transportation applications. *Journal of Power Sources*, 196(3):1593–1598, 2011. doi: 10.1016/j.jpowsour.2010.08.056.

[37] J.P. Deane, B.P. Ó Gallachóir, and E.J. McKeogh. Techno-economic review of existing and new pumped hydro energy storage plant. *Renewable and Sustainable Energy Reviews*, 14(4):1293–1302, 2010. ISSN 1364-0321. doi: 10.1016/j.rser.2009.11.015.

[38] April Saylor. General Compression looks at energy storage from a different angle. Online, February 2011. URL `http://energy.gov/articles/general-compression-looks-energy-storage-different-angle`.

[39] John Andrews and Nick Jelley. *Energy Science: Principles, Technologies, and Impacts*. Oxford University Press, 2007.

[40] Andrew J. Pimm, Seamus D. Garvey, and R. J. Drew. Shape and cost analysis of pressurized fabric structures for subsea compressed air energy storage. In *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, volume 225, pages 1027–1043. SAGE Publications, May 2011. doi: 10.1177/0954406211399506.

[41] Gordon Frederick Crichton Rogers, Yon Richard Mayhew, and Michael Hollingsworth. *Engineering Thermodynamics: Work and Heat Transfer*. Pearson Education, 4th edition, 1992.

[42] Chris Bullough, Christoph Gatzen, Christoph Jakiel, Martin Koller, Andreas Nowi, and Stefan Zunft. Advanced adiabatic compressed air energy storage for the integration of wind energy. In *Proceedings of the European Wind Energy Conference*, 2004. URL `http://www.ewi.uni-koeln.de/fileadmin/user_upload/Publikationen/Zeitschriften/2004/04_11_23_EWEC_Paper_Final.pdf`.

[43] Ke Yang, Yuan Zhang, Xuemei Li, and Jianzhong Xu. Theoretical evaluation on the impact of heat exchanger in advanced adiabatic compressed air energy storage system. *Energy Conversion and Management*, 86(0):1031–1044, 2014. ISSN 0196-8904. doi: 10.1016/j.enconman.2014.06.062.

[44] Benjamin Bollinger, Patrick Magari, and Troy O. McBride. High-efficiency liquid heat exchange in compressed-gas energy storage systems, May 2012. URL `http://www.google.com/patents/US8171728`.

[45] Chao Qin and Eric Loth. Liquid piston compression efficiency with droplet heat transfer. *Applied Energy*, 114:539–550, October 2014. doi: 10.1016/j.apenergy.2013.10.005.

[46] H. Barrow and C. W. Pope. Droplet evaporation with reference to the effectiveness of water-mist cooling. *Applied energy*, 84(4):404–412, 2007. doi: 10.1016/j.apenergy.2006.09.007.

[47] Kyoung Hoon Kim, Hyung-Jong Ko, and Horacio Perez-Blanco. Analytical modeling of wet compression of gas turbine systems. *Applied Thermal Engineering*, 31(5):834–840, April 2011. doi: 10.1016/j.applthermaleng.2010.11.002.

[48] LightSail website. Online, 2014. URL `http://www.lightsail.com`.

[49] General Compression website. Online, 2014. URL `http://www.generalcompression.com`.

[50] SustainX website. Online, 2014. URL `http://www.sustainx.com`.

[51] Xian Ma, Jingtian Bi, Weili Chen, Zhisen Li, and Tong Jiang. Research on new compressed air energy storage technology. *Energy and Power Engineering*, 5 (4B):22–25, 2013. doi: 10.4236/epe.2013.54B004.

[52] Shimshon Brokman, Isaac Shnaid, and Dan Weiner. Compressed air energy storage method and system, July 1996. URL `http://www.google.com/patents/US5537822`.

[53] Richard Brody. Optimizing grid infrastructure with site-flexible, fuel-free compressed air energy storage. In *IEEE PES T&D Conference and Exposition*, pages 1–4. IEEE, 2014. doi: 10.1109/TDC.2014.6863149.

[54] Albert J. Giramonti, Robert D. Lessard, William A. Blecher, and Edward B. Smith. Conceptual design of compressed air energy storage electric power systems. *Applied Energy*, 4(4):231–249, 1978. ISSN 0306-2619. doi: 10.1016/0306-2619(78)90023-5.

[55] Robert H. Schulte, Nicholas Critelli Jr., Kent Holst, and Georgianne Huff. Lessons from Iowa: Development of a 270 Megawatt Compressed Air Energy Storage Project in Midwest Independent System Operator. Technical report, Sandia National Laboratories, Albuquerque, New Mexico, January 2012. URL `http://www.sandia.gov/ess/publications/120388.pdf`.

[56] Raul Pateras Pescara. Motor-compressor apparatus, January 1928. URL http://www.google.com/patents?id=KoM_AAAAEBAJ.

[57] Raul Pateras Pescara. Motor compressor of the free piston type, May 1941. URL http://www.google.com/patents/US2241957.

[58] William T. Toutant. The Worthington-Junkers free piston air compressor. *Journal of the American Society for Naval Engineers*, 64(3):583–594, 1952. ISSN 1559-3584. doi: 10.1111/j.1559-3584.1952.tb02985.x.

[59] R. Mikalsen and A.P. Roskilly. A review of free-piston engine history and applications. *Applied Thermal Engineering*, 27:2339–2352, 2007. doi: 10.1016/j.applthermaleng.2007.03.015. URL http://www.sciencedirect.com/science/article/pii/S1359431107000968.

[60] R. Mikalsen and A.P. Roskilly. Performance simulation of a spark ignited free-piston engine generator. *Applied Thermal Engineering*, 28(14–15):1726–1733, 2008. ISSN 1359-4311. doi: 10.1016/j.applthermaleng.2007.11.015.

[61] E. J. Barth J. Riofrio. A free piston compressor as a pneumatic mobile robot power supply: Design, characterization and experimental operation. *International Journal of Fluid Power*, 8(1):17–28, March 2007. URL http://www.vanderbilt.edu/dces/PDF/papers/journal/Riofrio%20Barth%20FPC%20Int%20J%20of%20Fluid%20Power.pdf.

[62] A. Hibi and T. Ito. Fundamental test results of a hydraulic free piston internal combustion engine. *Proceedings of the Institution of Mechanical Engineers, Part D: Journal of Automobile Engineering*, 218(10):1149–1157, 2004. doi: 10.1177/095440700421801010.

[63] Seppo Tikkanen and Matti Vilenius. On the dynamic characteristics of the hydraulic free piston engine. In *ICMA'98: International conference on machine automation*, pages 193–202. Tampere University of Technology (TUT), 1998. URL http://cat.inist.fr/?aModele=afficheN&cpsidt=1368555.

[64] Tor A. Johansen, Olav Egeland, Erling Aa. Johannessen, and Rolf Kvamsdal. Dynamics and control of a free-piston diesel engine. *Journal of Dynamic Systems, Measurement, and Control*, 125:468–474, September 2003. doi: 10.1115/1.1589035.

[65] Tor Arne Johansen, Olav Egeland, Erling Aa Johannessen, and Rolf Kvamsdal. Free-piston diesel engine timing and control - toward electronic cam- and crankshaft. *IEEE Transactions on Control Systems Technology*, 10(2):177–190, 2002. doi: 10.1109/87.987063.

[66] Seppo Tikkanen and Matti Vilenius. Control of dual hydraulic free piston engine. *International journal of vehicle autonomous systems*, 4(1):3–23, 2006. doi: 10.1504/IJVAS.2006.009305.

[67] R. Mikalsen and A.P. Roskilly. The control of a free-piston engine generator. part 1: Fundamental analyses. *Applied Energy*, 87(4):1273–1280, 2010. ISSN 0306-2619. doi: 10.1016/j.apenergy.2009.06.036.

[68] R. Mikalsen and A.P. Roskilly. The control of a free-piston engine generator. part 2: Engine dynamics and piston motion control. *Applied Energy*, 87(4): 1281–1287, 2010. ISSN 0306-2619. doi: 10.1016/j.apenergy.2009.06.035.

[69] R. Mikalsen, E. Jones, and A.P. Roskilly. Predictive piston motion control in a free-piston internal combustion engine. *Applied Energy*, 87(5):1722–1728, 2010. ISSN 0306-2619. doi: 10.1016/j.apenergy.2009.11.005.

[70] Philip L. Skousen. *Valve Handbook*. McGraw-Hill, 1997.

[71] Heinz P. Bloch. *A Practical Guide to Compressor Technology*. John Wiley & Sons, 2nd edition, 2006.

[72] Bohdan T. Kulakowski, John F. Gardner, and J. Lowen Shearer. *Dynamic Modeling and Control of Engineering Systems*. Cambridge University Press, 3rd edition, 2007. ISBN 978-0-521-86435-0. doi: 10.1017/CBO9780511805417.

[73] Arthur G. O. Mutambara. *Design and Analysis of Control Systems*. CRC Press, 1999.

[74] George Ellis. *Control System Design Guide*. Elsevier Academic Press, 3rd edition, 2004.

[75] Bernard Friedland. *Control System Design - An Introduction to State-Space Methods*. Dover Publications, 1986. ISBN 978-0-486-44278-5. URL `http://app.knovel.com/hotlink/toc/id:kpCSDAISS1/control-system-design/control-system-design`.

[76] J.R. Dormand and P.J. Prince. A family of embedded Runge-Kutta formulae. *Journal of Computational and Applied Mathematics*, 6(1):19–26, 1980. ISSN 0377-0427. doi: 10.1016/0771-050X(80)90013-3.

[77] Lawrence F. Shampine and Mark W. Reichelt. The MATLAB ODE suite. *SIAM Journal on Scientific Computing*, 18(1):1–22, 1997. doi: 10.1137/S1064827594276424. URL `http://www.mathworks.co.uk/help/pdf_doc/otherdocs/ode_suite.pdf`.

[78] R. W. Klopfenstein. Numerical differentiation formulas for stiff systems of ordinary differential equations. *RCA Review*, 32(3):447–462, 1971.

[79] R. Ashino, M. Nagase, and R. Vaillancourt. Behind and beyond the Matlab ODE suite. *Computers & Mathematics with Applications*, 40(45):491–512, 2000. ISSN 0898-1221. doi: 10.1016/S0898-1221(00)00175-9.

[80] C. C. Bissell. *Control Engineering*. Chapman & Hall, 2nd edition, 1994. doi: 10.1007/978-1-4899-7224-8.

[81] Liuping Wang. *Model Predictive Control System Design and Implementation Using MATLAB*. Advances in Industrial Control. Springer, 2008.

[82] Christopher Edwards and Sarah Spurgeon. *Sliding Mode Control: Theory and Applications*. Taylor & Francis, 1998. ISBN 978-0748406012.

[83] Stanislaw H. Zak. *Systems and Control*. Oxford University Press, 2003. ISBN 978-0-19-515011-7. URL `http://app.knovel.com/hotlink/toc/id:kpSC000001/systems-and-control/systems-and-control`.

[84] Vadim Utkin, Jürgen Guldner, and Ma Shijun. *Sliding Mode Control in Electromechanical Systems*. Taylor & Francis, 1999. ISBN 0-7484-0116-4.

[85] G.F.C. Rogers and Y.R. Mayhew. *Thermodynamic and Transport Properties of Fluids*. Blackwell Publishing, fifth edition, 1995.

[86] Eric W. Lemmon, Richard T. Jacobsen, Steven G. Penoncello, and Daniel G. Friend. Thermodynamic properties of air and mixtures of nitrogen, argon, and oxygen from 60 to 2000K at pressures to 2000MPa. *Journal of Physical and Chemical Reference Data*, 29(3):331–385, 2000. doi: 10.1063/1.1285884.

[87] Long Fu, Guoliang Ding, and Chunlu Zhang. Dynamic simulation of air-to-water dual-mode heat pump with screw compressor. *Applied Thermal Engineering*, 23 (13):1629–1645, 2003. doi: 10.1016/S1359-4311(03)00109-1.

[88] C.J. Hõs, A.R. Champneys, K. Paul, and M. McNeely. Dynamic behavior of direct spring loaded pressure relief valves in gas service: Model development, measurements and instability mechanisms. *Journal of Loss Prevention in the Process Industries*, 31(0):70–81, 2014. ISSN 0950-4230. doi: http://dx.doi.org/10.1016/j.jlp.2014.06.005.

[89] Bing Xu, Ruqi Ding, Junhui Zhang, and Qi Su. Modeling and dynamic characteristics analysis on a three-stage fast-response and large-flow directional valve. *Energy Conversion and Management*, 79:187–199, 2014. doi: doi:10.1016/j.enconman.2013.12.013.

[90] Claudio Garcia. Comparison of friction models applied to a control valve. *Control Engineering Practice*, 16(10):1231–1243, 2008. doi: 10.1016/j.conengprac.2008.01.010.

Appendices

A Assumptions made in the model

The following are assumed to be constant and uniform:

- Wind speed
- External air pressure
- Air temperature at intake
- Compression tube's radius and length
- Max Coulomb friction on pistons
- Temperature and pressure of air in the HP manifold
- Behaviour parameters of valves and shock absorbers
- Position and orientation of the rotor's axis of rotation

The following are assumed to be negligible or non-existent:

- Bending of the compression tube, both in- and out-of-plane
- Bending of the tie rod

- Extension of the tie rod
- Torque in the tie rod
- All elastic deformation, except for the shock absorbers in the axial direction only
- All inelastic deformation
- All thermal expansion
- Fatigue of all parts, including thermal and mechanical fatigue
- Time delay or lag on signal and switches
- Leakage through any seal or closed valve, including the seal around the piston
- Any effects due to rotation of the pistons about the tube axis
- Any aerodynamic effects in the tubes - see uniform pressure rule below
- Magnetic or electrostatic forces

The following relationship and rules are assumed:

- All intake air is treated as dry air, with no effects from water vapour or salt deposition
- The temperature and pressure is uniform for each compression chamber
- The locking brakes on the piston are of infinite strength
- All sensors are of infinite resolution, perfect accuracy, and instantaneous reaction time
- The mass flow rate through the valve is assumed to be $\dot{m} = k\Delta p$

B Tie rod dynamic behaviour

The system at present uses several pairs of pistons, with each piston remaining on one side of the central hub but connected to a diametrically opposing piston using a tie rod. This rod, which will transmit only tensile force, reduces the tendency of centrifugal effects to force the pistons to the outer edge of the tube and hold it there.

It is important to understand the forces in the tie rod, both to better understand the design requirements governing it and also to predict the effects of non-synchronous actuation of valves in opposing sides of the hub on each piston.

B.1 Natural frequency

First, the force in the tie rod can be found by considering the centrifugal force on a single piston when the tie rod is centred in the tube;

$$\begin{aligned} F_{\text{cent}} &= \dot{\theta}^2 \left(\frac{L_{\text{CT}} + 2L_{\text{TE}}}{2} \right) m_P \\ &= 0.4^2 \cdot \frac{130 + 2 \cdot 5}{2} \cdot 30 \times 10^3 \\ &= 336 \text{ kN} \end{aligned} \tag{1}$$

We consider two possible materials; a steel at 100 MPa working stress and 200 GPa stiffness, and an aramid composite at 500 MPa working stress with 70 GPa stiffness. We can find the cross-sectional area of the tie rod for each material;

$$\begin{aligned} A &= \frac{F_{\text{cent}}}{\sigma} \\ &= \frac{3.36 \times 10^5}{100 \times 10^6} \\ &= 3360 \text{ mm}^2 \end{aligned} \tag{2}$$

We will use a mass-spring analogy to find the resonant frequency of the system. This considers the longitudinal vibrational mode, in which the first mode represents the pistons oscillating first towards, and then away from each other. From the basic stress-strain relationship and Hooke's law, the analogous spring stiffness is;

$$\begin{aligned} k &= \frac{A \cdot E}{L} \\ &= \frac{3.36 \times 10^{-3} \cdot 200 \times 10^{9}}{70} \\ &= 9.6 \text{ MN/m} \end{aligned} \tag{3}$$

The natural frequency is;

$$\begin{aligned} \omega_n &= \sqrt{\frac{k}{m}} \\ &= \sqrt{\frac{9.6E6}{30 \times 10^3}} \\ &= 2.85 \text{ Hz} \end{aligned} \tag{4}$$

For an aramid with a working stress of 500 MPa and a Young's modulus of 70 GPa, the natural frequency is 0.753 Hz. This means that, on the timescales of the forces in the simulation, the pistons are not rigidly connected, and asymmetric perturbations on the order of 0.1 seconds in duration may not be observed at the other piston at all. The model will need to reflect this by modelling each piston separately.

B.2 Modelling

We build a basic mass-spring model, using a single piston mass and the stiffness worked out above. Under normal use, the force on the tie rod should be a constant - since the tension due to the centrifugal force is constant, and all other forces should apply to the pistons equally. We will model the result of a failure in one set of valves so that the rod is subjected to the full pressure forces from a single piston.

C Rotational speed of the rotor

C.1 Effect of piston forces

During operation of the turbine, the pistons exert two different forces on the rotor: Coriolis forces and gravity forces.

Coriolis forces

Coriolis forces are fictitious forces which appear in a rotating reference frame. They are due to the conservation of rotational momentum; as our pistons move radially while restrained in the compression tube, their rotational inertia (relative to the centre of the rotor) changes. The momentum in this change is transferred to the rotor, which sees a force equal to the derivative of the overall rotor's inertia.

The Coriolis force required to keep a single piston in line along the compression tube, with a velocity (radially in the rotor reference frame) of $\dot{h}$ and a mass of m_P, with rotor rotational speed of $\dot{\theta}$ is given by:

$$F_{Coriolis} = -2m_P\dot{\theta}\dot{h} \tag{5}$$

This force is thus applied to the rotor at a distance of $(h + L_{\mathrm{TE}})$ from the hub, producing the Coriolis torque:

$$\tau_{Coriolis} = -2m_P\dot{\theta}\dot{h}(h + L_{\mathrm{TE}}) \tag{6}$$

This is also the expression given by differentiating the rotational inertia for the piston about the hub, $m(h + L_{\mathrm{TE}})^2$. Since there is no net change in the inertia of the rotor over time, we can conclude that the Coriolis torque will have no net effect; however, the fluctuating torque may cause cyclic variations in the rotational velocity of the

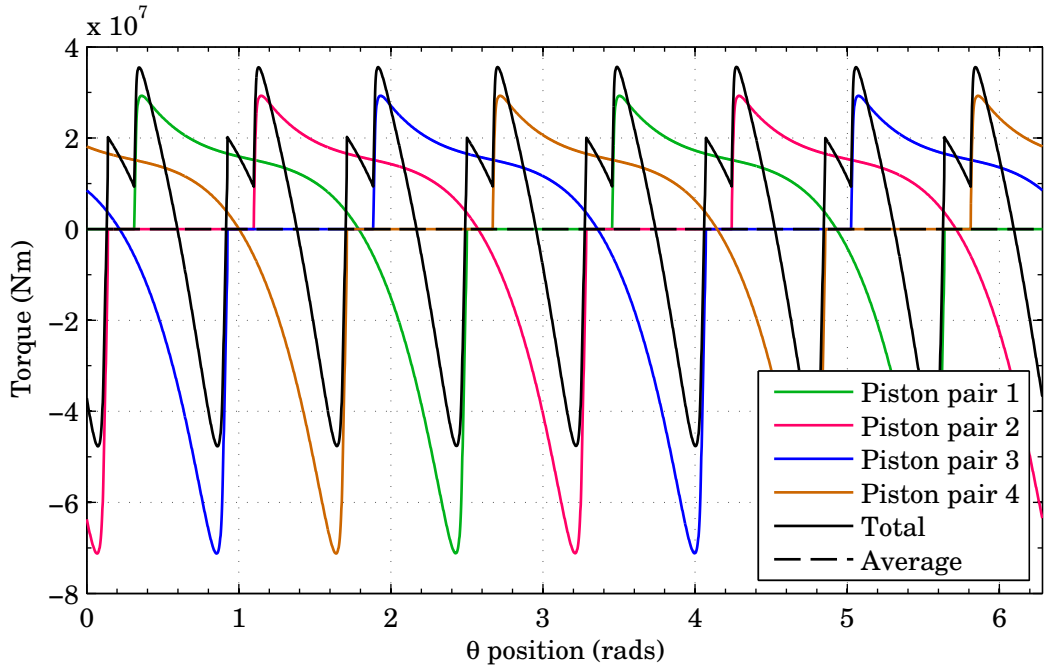


Figure C.1: Rotor torques due to Coriolis forces from piston motion

rotor. Figure C.1 shows the Coriolis torque from eight pistons (arranged in four pairs) around a full revolution of the rotor.

Gravity forces

The pistons will also produce a gravity torque, which is generated by the component of the gravity force on each piston which acts orthogonally to the compression tube. This is simply given by:

$$\tau_{gravity} = -\cos(\theta) m_P g (h + L_{\mathrm{TE}}) \tag{7}$$

Work done against this gravity torque is the means by which the turbine is extracting energy from the wind, so we expect this to average out at some negative value equal to the power of the turbine divided by the rotational speed $\dot{\theta}$. This is shown in Figure C.2.

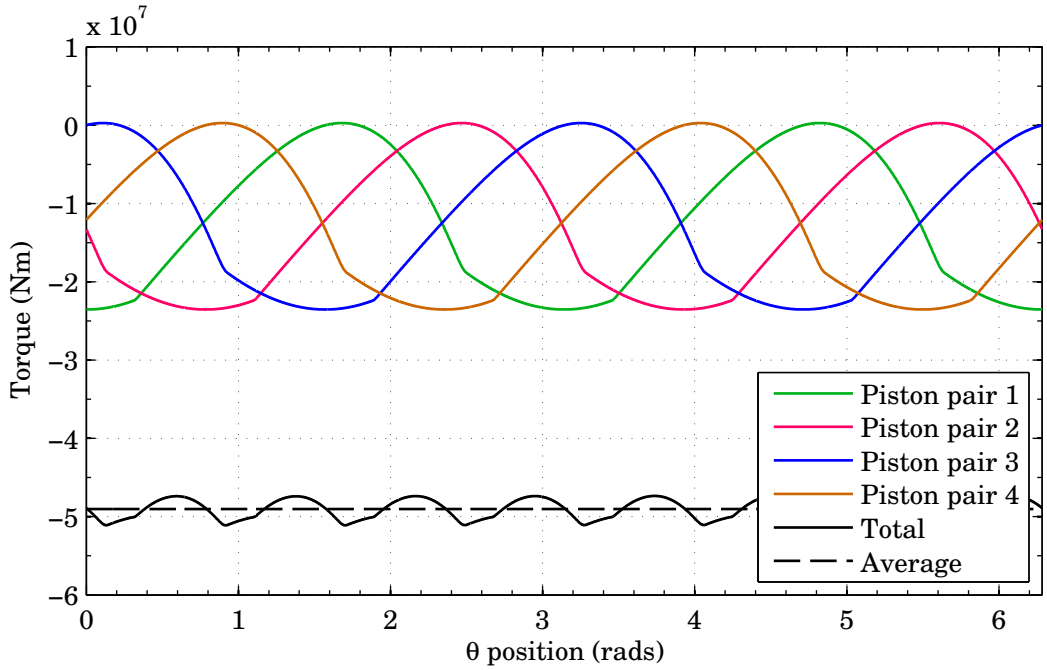


Figure C.2: Rotor torques due to gravity

C.2 Net effect on rotor

We can trivially calculate the rotational inertia of the pistons over the full cycle, which is shown in Figure C.3. We also need to make an estimation of the rotational inertia embodied in the non-compressing part of the rotor; this includes the compression tubes, blade elements, bracing cables, and bearings. Combining these two inertias with the total torque resulting from the Coriolis and gravity forces, we can simulate the variation in rotor angular velocity over a cycle, as shown in Figure C.4. It is clear that any variation from neutral values is minimal

D Heat transfer at walls

In Section 5.1.1, we needed a value for the effective thickness of the boundary layer of air at the surface. To obtain that value, we will construct a simple ODE.

We consider a solid cylinder of air, and look at the diffusion of heat into the air

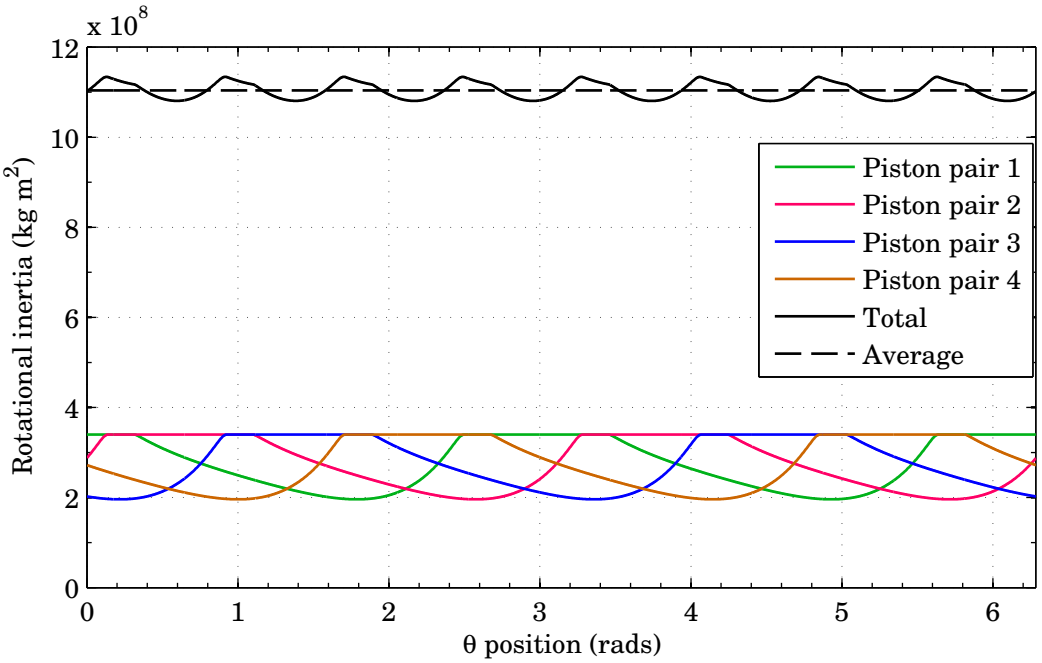


Figure C.3: Rotational inertia due to pistons

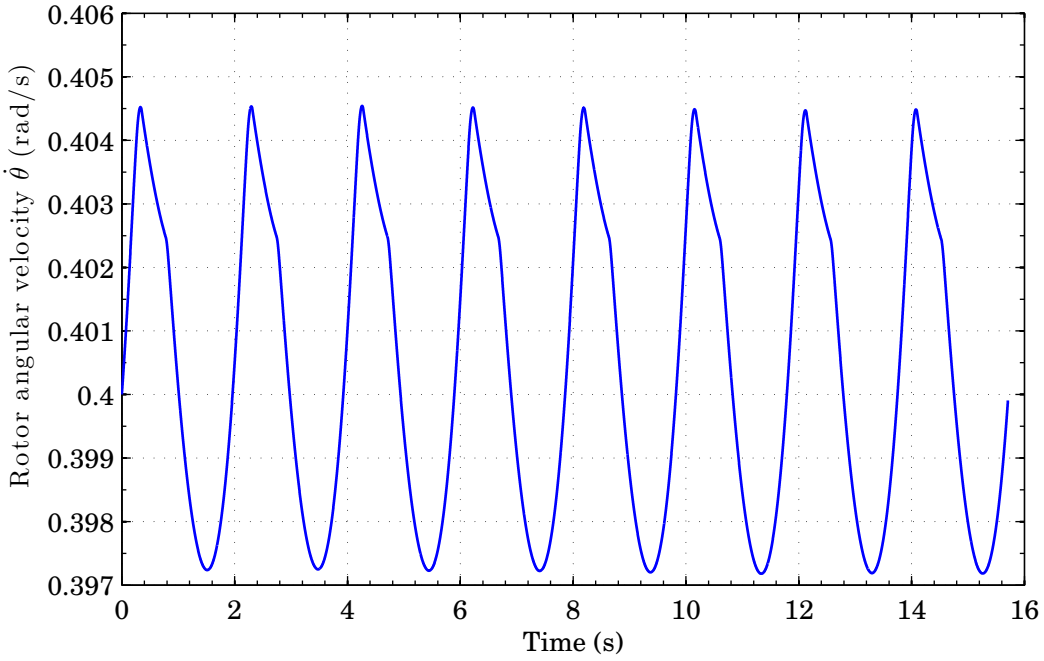


Figure C.4: Simulation of rotor angular velocity over one cycle

from the circumferential wall surrounding it. The heat passing through a cylindrical surface along the tube axis, with a radius r, height z, and thermal conductivity k is given by;

$$Q(r) = -\frac{\partial T}{\partial r} \cdot k \cdot (z \cdot 2\pi r) \tag{8}$$

We can form the partial differential equation by considering the net heat going into a hollow-cylindrical element of thickness Δr;

$$(c_v \rho)\frac{\partial T}{\partial t}\,(z \cdot 2\pi r \cdot \Delta r) = -\left|\frac{\partial T}{\partial r} k\,(z 2\pi r)\right|_r + \left|\frac{\partial T}{\partial r} k\,(z 2\pi r)\right|_{r+\Delta r} \tag{9}$$

Divide through by $k \cdot z \cdot 2\pi$;

$$\frac{c_v \rho}{k}\frac{\partial T}{\partial t}\,(r \cdot \Delta r) = \left|\frac{\partial T}{\partial r} r\right|_{r+\Delta r} - \left|\frac{\partial T}{\partial r} r\right|_r \tag{10}$$

The right hand side is a derivative with respect to r, which obtains;

$$\frac{c_v \rho}{k}\frac{\partial T}{\partial t} r = \frac{\partial^2 T}{\partial r^2} r + \frac{\partial T}{\partial r} \tag{11}$$

$$\frac{\partial T}{\partial t} = \frac{k}{c_v \rho}\left(\frac{\partial^2 T}{\partial r^2} + \frac{\partial T}{\partial r}\frac{1}{r}\right) \tag{12}$$

For air, we take $k = 0.02587$ W/m·K. At 800 K and 7 MPa, $c_v = 8116$ J/kg·K and $\rho = 29.72$ kg/m^3.

The initial temperature profile is a uniform air mass at 600 K, with the wall at 293 K. This is represented numerically as a large vector of temperature values, where the numerical derivatives at each point are defined as:

$$\left|\frac{\partial T}{\partial r}\right|_i = \frac{T_{i+1} - T_{i-1}}{r_{i+1} - r_{i-1}} \tag{13}$$

$$\left|\frac{\partial^2 T}{\partial r^2}\right|_i = \frac{T_{i+1} - 2T_i + T_{i-1}}{(r_{i+1} - r_i)(r_i - r_{i-1})} \tag{14}$$

With the end conditions:

$$\left|\frac{\partial T}{\partial r}\right|_{r_1} = \left|\frac{\partial T}{\partial r}\right|_{r_2} \text{and} \left|\frac{\partial^2 T}{\partial r^2}\right|_{r_1} = \left|\frac{\partial^2 T}{\partial r^2}\right|_{r_2} \tag{15}$$

$$\left|\frac{\partial T}{\partial r}\right|_{r_{\text{end}}} = \left|\frac{\partial^2 T}{\partial r^2}\right|_{r_{\text{end}}} = 0 \tag{16}$$

If we substitute these into the PDE, we obtain;

$$\left|\frac{\partial T}{\partial t}\right|_{r_i} = \frac{k}{c_v \rho}\left(\frac{T_{i+1} - 2T_i + T_{i-1}}{(r_{i+1} - r_i)(r_i - r_{i-1})} + \frac{T_{i+1} - T_{i-1}}{r_{i+1} - r_{i-1}}\frac{1}{r_i}\right) \tag{17}$$

$$\begin{aligned}\left|\frac{\partial T}{\partial t}\right|_{r_i} = {} & \frac{k}{c_v \rho}\left[\frac{1}{(r_{i+1} - r_i)(r_i - r_{i-1})} + \frac{1}{r_i(r_{i+1} - r_{i-1})}\right] T_{i+1} \\ & + \frac{k}{c_v \rho}\left[\frac{-2}{(r_{i+1} - r_i)(r_i - r_{i-1})}\right] T_i \\ & + \frac{k}{c_v \rho}\left[\frac{1}{(r_{i+1} - r_i)(r_i - r_{i-1})} - \frac{1}{r_i(r_{i+1} - r_{i-1})}\right] T_{i-1}\end{aligned} \tag{18}$$

This allows us to calculate a matrix **DT** such that;

$$\dot{T}(r) = \mathbf{DT} \cdot T(r) \tag{19}$$

Modelling this in the m-file `thermal_mod_func` with a 400 K temperature difference and around 1 second of conduction time produces the temperature profile shown in Figure D.1. This shows that only the first 2–3 mm of air will experience a temperature rise. As a result, we will take $t_{\text{air}} = 0.003$ mm.

E Linearising the model about a given state

The MATLAB model has experienced serious problems with stability throughout its lifetime. Extensive work has been done into the issues, including constructing a modified Euler solver to manually solve the ODE.

Having an accurate method of linearising the model is useful to determine its degree

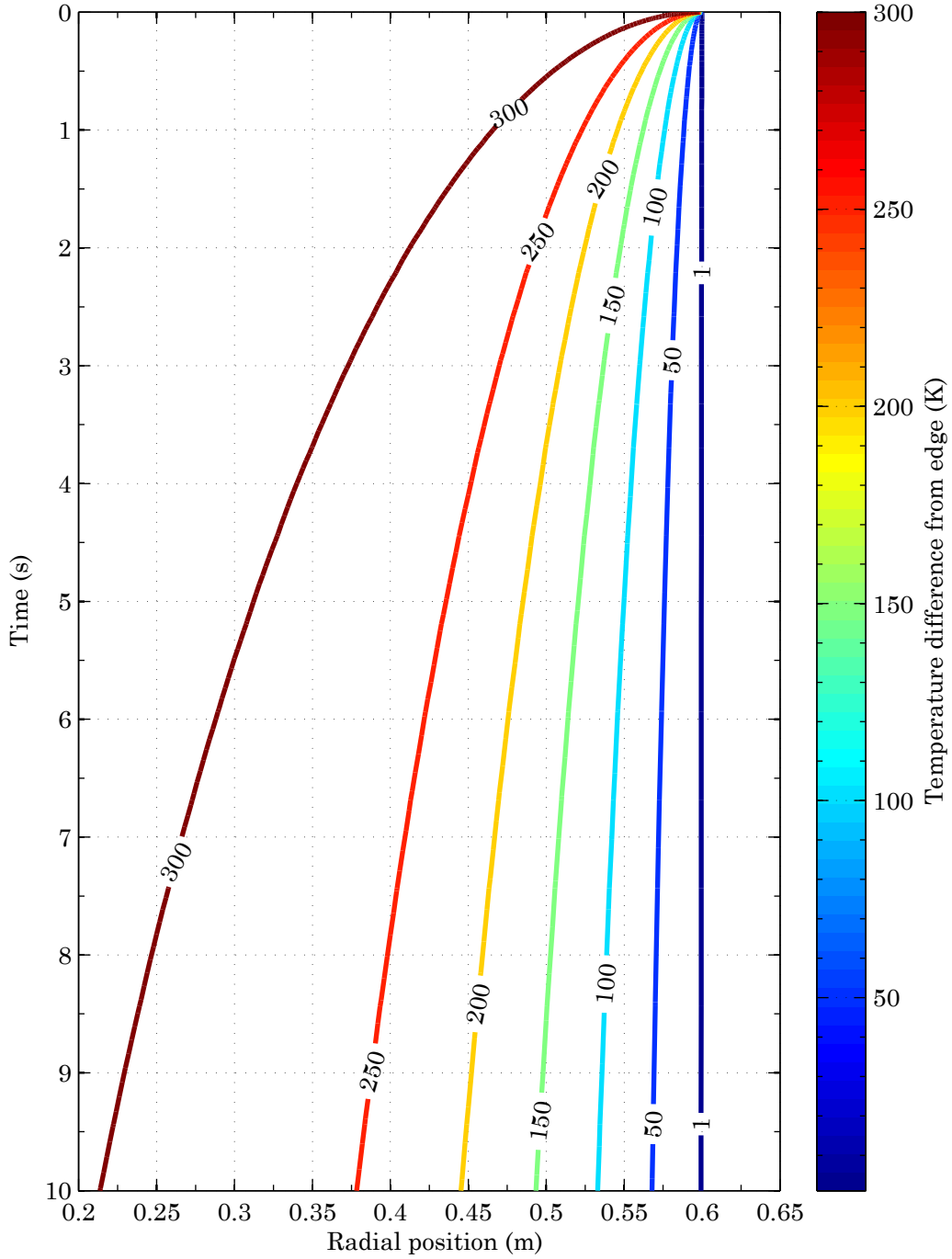


Figure D.1: Basic thermal model showing boundary layer thickness

of stiffness and nonlinearity, which may affect the capability of the MATLAB solvers to solve it. This was accomplished by building a function to calculate the sensitivity of the rate vector to small changes in the state vector. This sensitivity is represented in the matrix **A**, as shown below;

$$\mathbf{A} \equiv \begin{bmatrix} \frac{\mathrm{d}\dot{y}_1}{\mathrm{d}y_1} & \frac{\mathrm{d}\dot{y}_1}{\mathrm{d}y_2} & \cdots & \frac{\mathrm{d}\dot{y}_1}{\mathrm{d}y_n} \\ \frac{\mathrm{d}\dot{y}_2}{\mathrm{d}y_1} & \frac{\mathrm{d}\dot{y}_2}{\mathrm{d}y_2} & \cdots & \frac{\mathrm{d}\dot{y}_2}{\mathrm{d}y_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\mathrm{d}\dot{y}_n}{\mathrm{d}y_1} & \frac{\mathrm{d}\dot{y}_3}{\mathrm{d}y_2} & \cdots & \frac{\mathrm{d}\dot{y}_n}{\mathrm{d}y_n} \end{bmatrix} \tag{20}$$

The element in the ith row and jth column is the derivative of $\dot{y}_i$ (the ith element of the rate vector) with respect to y_j (the jth element of the state vector).

To find each element of **A**, we extend the simple numerical derivative formula from Equation 5.24 to the vector case:

$$\frac{\mathrm{d}\dot{y}_i}{\mathrm{d}y_j} = \frac{\dot{y}_i(y_j + \epsilon) - \dot{y}_i(y_j - \epsilon)}{2\epsilon} \tag{21}$$

The rate vector $\dot{y}$ is calculated twice, from two state vectors; both are based on y_0, the state about which we are linearising the problem, with a small increment ϵ either added or subtracted from the jth element. The difference between the value of the jth element in each rate vector is then divided by 2ϵ, giving an estimate of $\frac{\mathrm{d}\dot{y}_i}{\mathrm{d}y_j}$.

In order to obtain the best possible estimates for every element of **A**, we try the above numerical differentiation for a wide range of logarithmically-spaced ϵ values, then calculate the differences between all these estimates. A log-log plot of these differences versus ϵ shows a clear boundary; below some value of epsilon, the estimates will change randomly as we encounter the accuracy limits of the algorithm, while above this value, the estimates increase with ϵ. We pick the value of epsilon just above this boundary, to minimise the error. An example of the output of `lineariser.m` is

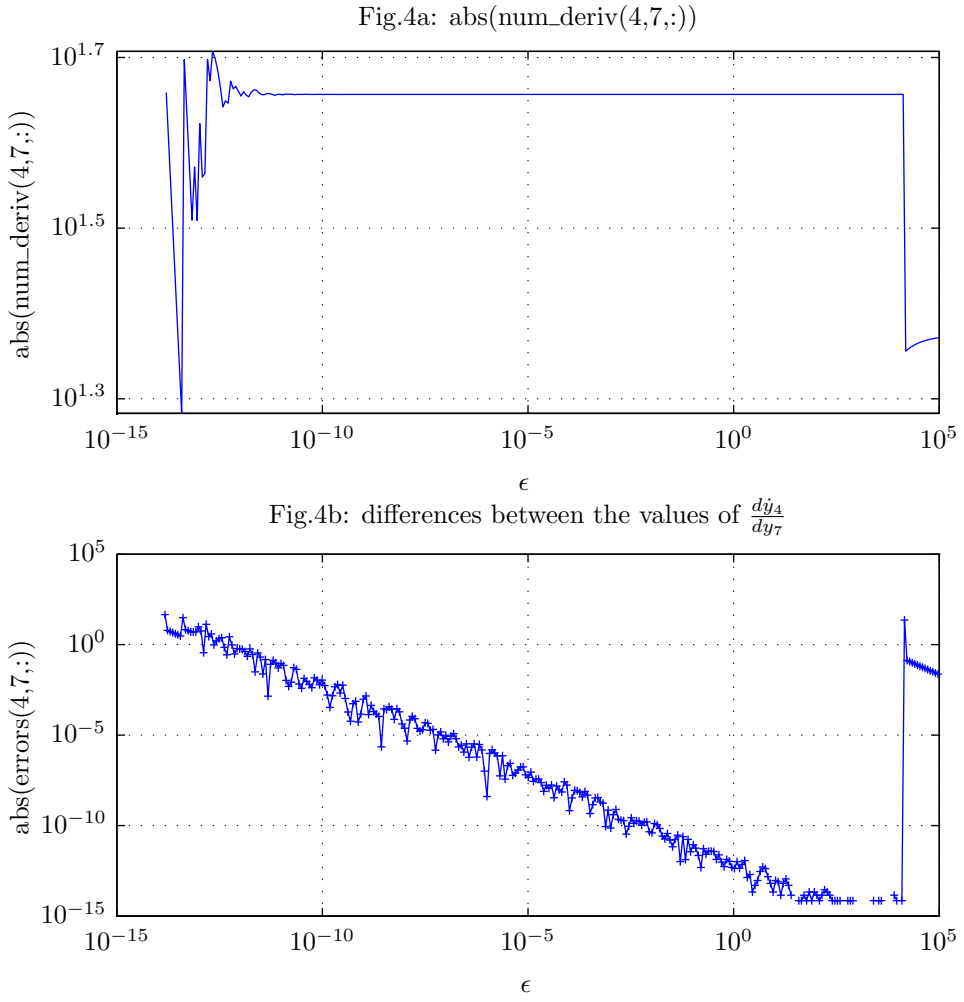


Figure E.1: Sample output graph from linearisation function

shown in Figure E.1.

The linearisation allows the investigation of the stiffness of the model about a particular state. Due to this stiffness, we move away from the Runge-Kutta family of solvers and use the `ode15s` solver based on numerical difference formulae instead.

F MATLAB scripts

F.1 Core model scripts

```
% function out = SW_run( in_omega,in_th,in_hdot ); isFunction = true;
% To make this script into a function, uncomment the line above.
if ~exist('isFunction','var') || isFunction == false
  tic; clc; close all; clear; isFunction = false;
end
dotimestamp, fprintf('Running %s ...\n',mfilename);
warning off verbose; warning off backtrace; % Change warning message
%% Get parameters
GP = SW_getparameters;
dotimestamp, fprintf(' - xxl is %i, ControlGraphs is %i, PlotGraphs is %i.\n',...
  GP.xxl,GP.ControlGraphs,GP.PlotGraphs);
if isFunction % Use input if running as a function
  GP.omega = in_omega;
  GP.C.th_kick = in_th;
  GP.C.kick_hdot = in_hdot;
end
% Overrides
% GP.water_add = 0; % Turn off water injection
GP.runtime = 30;
GP.xxl = 0;

% GP NEVER MODIFIED BELOW THIS LINE
%--------------------------------------------------------------------------
%% Define initial y, y_init
y_init.theta = 0.7861;     % theta, CCW angle from horizontal (rads)
y_init.h = 90;          % h, height (must be suitable for mode 0)
y_init.hdot = -15;       % h dot, velocity
h_a = GP.h_a_fun(y_init.h);
for c1 = 1:2 % Stuff with both ends
  y_init.m_a{c1} = h_a{c1}*GP.A*GP.p_atm/(GP.T_atm*GP.R_univ/GP.Mol.air);
  % Air masses, m_a (based on ideal gas law)
  y_init.T_a{c1} = GP.T_atm-30;                  % Air/steam/water temperatures, T_a
  y_init.m_wg{c1} = 0;                        % Steam masses, m_wg
  y_init.m_wf{c1} = 1e-6;                     % Liquid water masses, m_wf
  y_init.T_te{c1} = (GP.T_kick+GP.T_atm)/2+3; % Temperatures of ends of tube, T_te
  y_init.T_pe{c1} = GP.T_atm+6.2;             % Temperatures of ends of piston, T_pe
  y_init.k_e{c1} = 0;                         % Valve constants, k_e
  y_init.k_e_dot{c1} = 0;                     % Rates of change of valve constants, k_e_dot
end
y_init.m_out.net = 0;       % Net m_out, mass of air in storage (subtracting kick)
y_init.m_out.exh = 0;       % Exhausted m_out, total mass of air exhausted (ignoring kick)
y_init.wd_p = 0;              % E_WD,p, work done by piston vs pressure
% (positive compressing, positive exhausting, negative kicking)
y_init.wd_g = 0;              % E_WD,g, work done by rotor vs gravity

y_init.opthets = zeros(GP.T.npolys,1); % Coefficients of orthpolys
y_init.opthets(1) =  1.469273985e4; % Projected steady state
y_init.opthets(3) = 375.114;
y_init.opthets(5) = 214.0543;
y_init.opthets(7) = 59.27124;
y_init.opthets(9) = -25.13436;
y_init.opthets(11) = -46.76907;

y_init.vecsize = 23 + GP.T.npolys;
y0 = SW_pack(y_init);

%% Initialise sys and counting variables
sys = struct;  % Structure of modes and other persistent settings
sys.dir = 3;            % Current direction of operation
sys.mode = 0;           % Current mode of operation
sys.psi = {0,0};        % Control trajectory parameters
sys.comp_m_a = {0,0};   % Air masses when compressing started (for sensors.m)

% Initial values of controlled variables:
sys.C.Emarg = 0.15;  % Additional fraction of energy needed at compression switch

sys.cyclenum = 1; % What cycle we're on

sys.phase = {1,1}; % Phase of water. 1 = liquid, 2 = phasechange, 3 = steam.
```

```
sys.sim = false;  % Is this current run occurring inside the slow controller?

if GP.debug_msg, SW_debug_message(2);  end % Reset debug message

out.sys(1) = sys;
t0 = 0; out.te = t0; out.yev = y0; out.ie = 0;
numpoints = 0; out.evtdata = [];
out.t = t0; out.yv = y0;
if GP.xxl == -1 % If we're recording full sets of states
  moderefine = 3*ones(1,7); moderefine(5) = 1; % Refine more points unless locked
else
  moderefine = ones(1,7); % Never refine points
end

%% Run simulation
docolumns = @(x) fprintf('    ODETime    Theta/pi    Pts   CTime Event Description\n');
dotimestamp, fprintf('Running simulation for %.1f minutes internal time, using ode15s\n',GP.runtime);
if ~isFunction && GP.WaitBar
  dotimestamp(0), docolumns(1);
  h = waitbar(0,'Running simulation...');
end
eventdur = tic; % Start the event timer
simdur = tic; % Start the simulation timer

while GP.simfail == false  &&  t0 < GP.runtime * 60

  if sys.mode == 2 % If we're exhausting
    tf = t0 + GP.C.slow_runtime; % Run for the duration of the slow controller
  else
    tf = t0 + 1; % Otherwise run for 1 second
  end

  % Escalate stiffness warning to an error (undocumented)
  warning error MATLAB:ode15s:IntegrationTolNotMet; %#ok<CTPCT,WNTAG>
  try % In case of ODE solver error
    % Deactivate ill-conditioned matrix warning
    warning off MATLAB:illConditionedMatrix
    [t,y,te,ye,ie] = ode15s(@(t,y)SW_ode(GP,sys,t,y),[t0,tf],y0, ...
      odeset('Events',@(t,y)SW_events(GP,sys,t,y),'Refine',moderefine(sys.mode+1)) );
    % Reactivate ill-conditioned matrix warning
    warning on MATLAB:illConditionedMatrix

    y = y'; ye = ye'; % Rotate y and ye to be rows (for SW_pack)
    t0 = t(end); % Prepare for next loop of matrix
    y0 = y(:,end);

    % Delete the last status update
    if GP.debug_msg, SW_debug_message(1); end

    if GP.xxl == -1 % If recording all states
      for c1 = 1:length(t)
        out.sys(length(out.t)+c1) = sys;
      end
      out.t = [out.t; t(2:end)];
      out.yv = [out.yv, y(:,2:end)];
    end
    numpoints = numpoints + length(t);

    if numel(ie) > 1 % If multiple events occuring simultaneously
      dotimestamp, fprintf('ERROR - simultaneous events at ODETime %.6f [%s]\n',...
        t0,num2str(ie(:)'));
      GP.simfail = true; % Get out of the while loop
    elseif ~isempty(ie) % If an event has happened
      switch ie  % What event just happened?
        case {1 , 2}
          eventname = sprintf('Dir %i energy matched, compressing', ie);
          sys.dir = ie; sys.mode = 1;
          y0 = SW_pack(y0); % Pack to struct
          sens_temp = SW_sensors(GP,t,y0,sys);
```

```
      sys.comp_m_a = sens_temp.y.m_a; % Remember current airmass value
      y0.m_wf{sys.dir} = GP.water_add * y0.m_a{sys.dir};% Add water
      y0 = SW_pack(y0); % Unpack to vector
    case 3
      eventname = 'Pressure reached, exhausting';
      sys.mode = 2;
      % Set initial value for psi
      sys.psi{sys.dir} = 1e-3;
      % Save current state ready for slow controller
      sens_queued = SW_sensors(GP,t0,SW_pack(y0),sys);
      % Close all figures if control_slow is plotting
      if GP.ControlGraphs, close all; end
    case 4
      eventname = 'Piston stopped, dumping';
      sys.mode = 3;
      if GP.ControlGraphs % Add surface of target value
        targsurf.x = get(gca,'XLim'); targsurf.y = get(gca,'YLim');
        fill3(...
          targsurf.x([1,1,2,2]),targsurf.y([2,1,1,2]),...
          GP.C.targ*[1,1,1,1],'g',...
          'FaceAlpha',0.5,'EdgeAlpha',0.5,'EdgeColor','g');
      end
      y_temp = SW_pack(y0); % Pack to struct temporarily
      h_a_temp = GP.h_a_fun(y_temp.h); % Get h_a at end
      sys.C.Emarg = sys.C.Emarg ... % Update Ereq_margin using proportional control
        + GP.C.Emarg.gain * (h_a_temp{sys.dir} - GP.C.Emarg.h_targ(sys.cyclenum));
    case 5
      eventname = 'Pressure equal, piston at rest';
      sys.mode = 4;
    case 6
      eventname = 'Angle reached, kick started';
      sys.mode = 5;
    case 7
      eventname = 'Velocity reached, kick closed';
      sys.mode = 6;
    case 8
      eventname = 'Pressure equalised, entering freefall';
      sys.dir = 3; % 3 = no dir
      sys.mode = 0;
      sys.cyclenum = sys.cyclenum + 1;
    case { 9 , 10 } % Entered phase transition
      event_end = ie-8; % Which end had the event happen
      eventname = sprintf(' - dir %i started phase transition',event_end);
      sys.phase{event_end} = 2;
    case { 11 , 12 } % Finished phase transition by running out of steam
      event_end = ie-10; % Which end had the event happen
      sys.phase{event_end} = 1; % We're in liquid region
      eventname = sprintf(' - dir %i now in liquid phase',event_end);
    case { 13 , 14 } % Finished phase transition by running out of water
      event_end = ie-12; % Which end had the event happen
      sys.phase{event_end} = 3; % We're in liquid region
      eventname = sprintf(' - dir %i now in steam phase',event_end);
  end
  if GP.xxl <= 0 || ( ie == 8 && mod(sys.cyclenum,GP.xxl) == 0 )
    % If recording all events, or at end of a cycle
    out.ie = [out.ie; ie]; %#ok<*AGROW>
    out.te = [out.te; te]; % Save the state information
    out.yev = [out.yev, ye];
    if GP.xxl ~= -1 % If not recording everything
      out.sys(length(out.te)) = sys;
    end
    out.evtdata = [out.evtdata; numpoints,toc(eventdur),sys.dir,ie];
  end
  if ~isFunction
    dotimestamp, fprintf('%12.6f %10.4f %7i %6.2f   %2i  %s\n', ...
      te, ye(1)/pi, numpoints, toc(eventdur), ...
      ie, eventname );
  end
  eventdur = tic; numpoints = 0; % Reset event timer and points counter
```

```
    elseif sys.mode == 2 % Stopped at the end of a slow controller run
      if ~isFunction
        dotimestamp, ...
          fprintf('%12.6f %10.4f %7i %6.2f    -    - Timestop for control_slow\n', ...
          t0, y0(1)/pi, numpoints, toc(eventdur));
      end
      % Use the queued sens pack from last stop to update sys.psi
      sys.psi{sys.dir} = SW_control_slow(GP,sys,sens_queued,GP.ControlGraphs,true);
      % Now save a sensor pack of the current state to use at the next stop
      sens_queued = SW_sensors(GP,t0,SW_pack(y0),sys);
      % Add new column headings if needed
      if GP.ControlGraphs, dotimestamp(0), docolumns(1); end
      eventdur = tic; numpoints = 0; % Reset event timer and points counter
    end
  catch exception % If we hit an error during the ODE solving ...
    % ... and if that error was the tolerance warning we escalated
    if strcmp(exception.identifier,'MATLAB:ode15s:IntegrationTolNotMet')
      % say so then quit the while loop.
      dotimestamp, fprintf('ERROR - %s\n',exception.identifier);
      GP.simfail = true; % Quit the while loop
    else % If it wasn't the warning we escalated
      rethrow(exception); % Then it did count after all.
    end
  end
  if ~isFunction && GP.WaitBar
    waitbar(toc(simdur)/(GP.runtime*60),h);
  end
end
if ~isFunction && GP.WaitBar
  close(h);
end
dotimestamp, fprintf(' - Simulation ended.\n');
if GP.debug_msg, SW_debug_message(2); end % clear persistents

%% Save data
out.GP = GP;
if ~isFunction % don't save if we're in function mode
  % Specify a filepath *outside* Dropbox (avoid wasting time syncing 100MB files)
  file.name = ['../../../../MATLAB_outputs/',datestr(now,'yyyy-mm-dd_HH.MM'),'.mat'];
  save(file.name,'out');
  file.data = dir(file.name);
  if file.data.bytes > 2e6
    file.size = sprintf('%.3f MB',file.data.bytes/1e6);
  else
    file.size = sprintf('%.0f kB',file.data.bytes/1e3);
  end
  dotimestamp, fprintf(' - Saved structure ''out'' to file %s (%s)\n',...
    file.name,file.size);
end

%% Finish up
SW_postprocess;
dotimestamp, fprintf('%s complete.\n',mfilename);

%% Postpostprocessing
% custom_solver(out,plots,GP,sys); % Call other solvers to look into stability
% lineariser;
```

```
function GP = SW_getparameters
% This function returns the default set of parameters for the model
%  One-off runs are controlled by changing the variables in SW_run

%% Overall parameters
GP.runtime = 3;   % Internal ODEtime to run for in minutes
GP.PlotGraphs = true;    % Plot final graphs?
GP.WaitBar = false; % Plot fancy waitbar?
GP.ControlGraphs = false;    % Plot control_slow graphs?
GP.xxl = -1;    % Sets data recording policy:
% -1 = record all states,
% 0  = record state at every event,
% n  = record state every nth half-cycle (one cycle = both ends compress)

GP.debug_msg = false; % Print live debug message?
% Set which entries in y to display:
GP.debug_vals = {'y.T_a{2}','y.hdot','y.h'};

%% GP - Core parameters
GP.g = 9.80665;           % Accel due to gravity
GP.p_atm = 101325;        % Atmospheric air pressure
GP.p_targ = 70e5;         % Target air pressure
GP.T_atm = 293;           % Atmospheric air temp in Kelvin
GP.A = 1;                 % Xsec area of tube
GP.A_tr = 0.003;          % Xsec area of tie rod.
GP.A_tend = 0.8*GP.A;     % Xsec area of one tube end for heat transfer calcs
GP.L_hubrad = 5;          % Distance from rotation centre to tube base
GP.L_T = 130;             % Length of tube
GP.L_ph = 5;              % Half piston length (symmetric about coordinate)
GP.m_P = 30e3;            % Mass of a piston pair and cable
GP.F_mu = 1000;           % Maximum Coulomb friction on pistons
GP.v_HB = 0.5;            % Velocity half-bandwidth (for friction ramping)
GP.omega = 0.40;          % Rotational speed (in rad per sec)

%% GP - Direction and shock absorber anonymous functions
% Direction functions, to get airmass height and velocities
GP.h_a_fun = @(h) { GP.L_T - GP.L_ph - h ; h - GP.L_ph };
GP.h_a_dot_fun = @(hdot) { -hdot ; hdot };
% Generalised piecewise line definition
%  (evaluates continuous function with negative gradient and saturation at 0)
GP.spline = @(P, x) 0.5*(P.grad*( P.interc - x ) ...
  + sqrt( P.grad^2 * (x - P.interc).^2 + 4*P.rad^2) );
% Shock absorbers at end of tube
% Damping function - will return *damping constant*
GP.shox.D.rad = 100;       % Radius of curvature of rounding on continuous function
GP.shox.D.interc = 0.01;   % h-intercept of linear part of force function
GP.shox.D.grad = 1.5e9;    % Rate of change of damping constant (linear part of fcn)
% Shock absorbers at end of tube
% Spring function - will return *force*
GP.shox.S.rad = 100;       % Radius of curvature of rounding on continuous function
GP.shox.S.interc = 0.005;  % h-intercept of linear part of force function
GP.shox.S.grad = 1.5e9;    % Spring stiffness of linear part
% Total shock absorber force
%  sign(GP.L_T/2 - h) is +ve at low h and -ve at high, so always towards tube middle
%  Both are doubled to reflect 2 pistons
GP.shox.force = @(h,h_dot,h_a) ...
  sign(GP.L_T/2 - h) * 2*GP.spline(GP.shox.S,h_a)... % Spring force
  - h_dot * 2*GP.spline(GP.shox.D,h_a); % Damping force

%% GP - Temperature-related parameters & functions
GP.R_univ = 8.3144621;    % Universal gas constant \bar{R}
GP.Mol.air = 28.9645/1e3; % Molar mass of air
GP.Mol.wat = 18.0155/1e3; % Molar mass of water
GP.rho_water = 958;       % Density of liquid water (at 373K)
% Get various functions for thermal properties like heat capacity, conductivity, boiling point
GP.Tfun = SW_thermal_funcs(false);
% Func for heat capacity ratio of air/steam mix (normally 1.401 for dry air)
GP.gam = @(m_a, m_wg, m_wf, T) ...
  ( GP.Tfun.c_pa(T).*m_a + GP.Tfun.c_pwg(T).*m_wg + GP.Tfun.c_pwf(T).*m_wf ) ./...
```

```
  ( GP.Tfun.c_va(T).*m_a + GP.Tfun.c_vwg(T).*m_wg + GP.Tfun.c_vwf(T).*m_wf );
GP.water_add = 0.08;      % kg of water to add per kg of air at start of compression
GP.T.T_max = 300;         % Assumed maximum temperature (for energy required isothermal part)
GP.T_kick = 500;          % Temperature of air injected during kick (in K)
GP.T.k_wall = 35;         % Thermal conductivity of wall in W/(K*m)
GP.T.rho_wall = 7800;     % Density of wall
GP.T.cp_wall = 0.46e3;    % Volumetric heat capacity of wall
GP.T.pe_t = 0.1;          % Thickness of end-of-piston for heating calcs (m)
GP.T.m_add = 1e-3;        % Small mass added to stabilise T functions at low mass
GP.T.R.t_wall = 0.005;    % Thermal thickness of wall for thermal resistance
GP.T.R.t_air = 0.15;     % Thickness of air boundary layer for thermal resistance
GP.T.R.func = @(T_bl) GP.T.R.t_wall./GP.T.k_wall + GP.T.R.t_air./GP.Tfun.k_air(T_bl);
% Function to get thermal resistance per m^2 at a given boundary layer temp T_bl
GP.T.npolys = 12;         % Number of polynomials used for wall temp
% Get op.Ap, op.Apdd, op.psdinvAp and op.hvals for wall temp modelling
GP.T.op = SW_getorthovectors(GP.T.npolys, [0,GP.L_T], false);

%% GP - Control- & valve-related
GP.dump.k = 1e-3;             % Dump valve constant (kg/s per Pa)
GP.dump.spec.rad = 100;       % Radius of curvature on rounding func
GP.dump.spec.interc = 1e5;    % Set pressure difference where valve starts to dump (Pa)
GP.dump.spec.grad = -1;       % Negative gradient of inclined line (1)
GP.dump.fun = @(dp)  GP.dump.k * GP.spline(GP.dump.spec,dp);% (input in Pa, output in kg/s)

GP.C.k_LP = 0.1;            % LP valve constant in&out (kg/s per Pa)
GP.C.val_sat = 0.0045;      % Saturation limit of HP valve constant (kg/s per Pa)
GP.C.val_stiff = 1e7;       % 'Spring' force on HP valve constant
GP.C.val_damp = 2*2*sqrt(GP.C.val_stiff); % Damping on HP valve constant = 2*C_crit

GP.C.ptb = 0.1;             % Fraction to perturb sys.psi by in slow controller
GP.C.slow_runtime = 0.02;   % Time it takes slow controller to run (s)
GP.C.targ = GP.shox.S.interc; % Target h_a for slow controller

% Get energy required per unit height for set constants
GP.C.Ereq_fun = SW_water_energy(GP,false);

% First-level control: Ereq_margin
GP.C.Emarg.gain = 3e0; % Gain to apply to Ereq_margin control
% Target dump airmass height (in m) as a function of cycle number
GP.C.Emarg.h_targ = @(cycle) GP.shox.D.interc * (5*exp(-0.15*cycle) + 1) ;
% Second-level control: Kick end velocity
% GP.C.kick_hdot.decay = 0.99^(1/20); % Factor to discount updates by (reduce speed)

% Preset: angle when to start kick
GP.C.th_kick = 0.8;   % Angle when to kick piston after rimwards comp (radians)
GP.C.kick_hdot = 7; % Piston velocity when to disengage kick

% System variables
GP.post = false;          % Allow deactivation of debug messages in post
GP.replay = false;        % Tells ODE to access stored control signals
GP.simfail = false;       % Allows neat error cleanup from while loop
```

```
function [ydot, Tdot, accel_cmp] = SW_ode(GP,sys,t,y)
% [ydot, Tdot] = SW_ode(GP,sys,t,y)
ydot = SW_pack(0*y);
y = SW_pack(y);
h_a = GP.h_a_fun(y.h);
h_a_dot = GP.h_a_dot_fun(y.hdot);

p = SW_pressure(GP,y);

%%  POSITION & ACCELERATION
ydot.theta = GP.omega;
ydot.h = y.hdot;
% Pressure accel
accel_cmp(1) = (p{2}-p{1})*(2*GP.A - GP.A_tr)/GP.m_P;
% Gravitational accel
accel_cmp(2) = - GP.g*sin(y.theta);
% Centrifugal accel
accel_cmp(3) = (y.h-GP.L_T/2)*(GP.omega^2);
% Friction accel
fric = -y.hdot*GP.F_mu/GP.v_HB;
fric(fric > +GP.F_mu) = +GP.F_mu;
fric(fric < -GP.F_mu) = -GP.F_mu;
accel_cmp(4) = fric/GP.m_P;
% Shock absorber accel
if ~sys.sim && sys.dir ~= 3
  % Only if we're not inside a control_slow simulation and not in freefall
  accel_cmp(5) = GP.shox.force(y.h,y.hdot,h_a{sys.dir}) / GP.m_P;
end
% Total accel
accel_cmp(6) = sum(accel_cmp);
ydot.hdot = accel_cmp(6);

%%  VALVES & AIR MASS
% Will need sens package both times
sens = SW_sensors(GP,t,y,sys,sys.sim);
for c1 = 1:2 % Once for each direction
  m_dot = zeros(3,1); % Initialise
  % Find out k_ref
  k_ref = SW_control_fast(GP,sys,c1,sens); % Note c1 is passed into control_fast as direction
  if sys.sim % If we're running in a slow-controller simulation
    k_exh = k_ref; % Assume the controller gets what it wants
  else  % Apply k_ref target to simple valve ode
    ydot.k_e{c1} = y.k_e_dot{c1}; % Mass-spring-damper analogy
    ydot.k_e_dot{c1} = GP.C.val_stiff * (k_ref - y.k_e{c1}) ...
      - GP.C.val_damp * y.k_e_dot{c1};
    k_exh = y.k_e{c1}; % Otherwise use the actual value
    k_exh(k_exh<0) = 0; % Use 0 if negative
  end
  % Get total HP mass flow rate
  m_dot(1) = k_exh * (GP.p_targ - p{c1});
  if sys.dir == c1 && sys.mode == 3
    % If dumping in this direction, also have dump valve open
    m_dot(2) = - GP.dump.fun(p{c1} - GP.p_atm);
  end
  if sys.dir ~= c1 || sys.mode == 0
    % If not compressing this way, or in freefall, then allow free movement of air through LP
    %  valve
    m_dot(3) = GP.C.k_LP * (GP.p_atm - p{c1});
  end
  % Find total outwards MFR, i.e. total of negative components only
  m_dot_out = sum(m_dot(m_dot<0));
  % Spread that around the air / liquid water / steam using a weighted average
  m_total = y.m_a{c1} + y.m_wf{c1} + y.m_wg{c1};
  ydot.m_a{c1} = m_dot_out * y.m_a{c1}/m_total;
  ydot.m_wf{c1} = m_dot_out * y.m_wf{c1}/m_total;
  ydot.m_wg{c1} = m_dot_out * y.m_wg{c1}/m_total;
  % Now add total positive MFR to air component only (intake is dry).
  ydot.m_a{c1} = ydot.m_a{c1} + sum(m_dot(m_dot>0));
  % Finally, m_out only tracks HP valve flow (not dump or LP):
```

```
  ydot.m_out.net = ydot.m_out.net ... % Start with current value
    - max(m_dot(1),0)... % add m_dot(1) if it's positive
    - min(m_dot(1),0)*y.m_a{c1}/m_total; % Scale m_dot(1) by air fraction if not
  if ydot.m_out.net > 0 % If positive exhaust
    ydot.m_out.exh = ydot.m_out.net; % Include in exhausted air
  end % (otherwise, ydot.m_out.exh will default to 0)
end

%%  AIRMASS TEMPERATURES
% Initialise Tdot, vector to collect all components of airmass Tdot
Tdot = zeros(5,2);
% >> Do ideal-gas-law based Tdot component
for c1 = 1:2 % Once for each end
  gamma = GP.gam(y.m_a{c1},y.m_wg{c1},y.m_wf{c1},y.T_a{c1});
  Tdot(1,c1) = y.T_a{c1} * ( gamma - 1 ) * ...
    (ydot.m_a{c1}/(y.m_a{c1}+GP.T.m_add) - h_a_dot{c1} ./ h_a{c1});
end
if ~sys.sim % If not in a simulation...
  % >> Wall temperatures
  % Calculate current cylinder temperature profile
  h_index = find(GP.T.op.xvals>y.h,1); % Index of first element of hvals above piston
  T_A = 0*GP.T.op.xvals;
  T_A(h_index:end) = y.T_a{1}; % Rimwards temps
  T_A(1:h_index-1) = y.T_a{2}; % Hubwards temps
  T_ext = GP.T_atm * ones(length(GP.T.op.xvals),1);
  % Calculate wall temperature profiles
  T_w = GP.T.op.Ap * y.opthets;
  d2Twdh2 = GP.T.op.Apdd * y.opthets;
  % Get thermal resistance of boundary layer based on wall temps
  Rth_wall = GP.T.R.func(T_w);
  % Calculate heat transfer on inner and outer surfaces
  Q_insurf = (T_A - T_w) ./ Rth_wall; % Positive *into* wall
  Q_outsurf = (T_ext - T_w) ./ Rth_wall; % Positive *into* wall
  % For wall, combine Tdot_surf with along-wall conduction and get thdot
  Twdot = (-GP.T.k_wall * d2Twdh2 + Q_insurf + Q_outsurf)/(GP.T.rho_wall * GP.T.cp_wall);
  ydot.opthets = GP.T.op.psdinvAp * Twdot;
  % >> Air conduction Tdot components
  % For air, integrate inner surf trans on each side (indexing with piston interval
  % ignored)
  TAdot_int = -Q_insurf .* GP.T.op.xdiffs;
  TAdot_surf(1) = sum( TAdot_int(h_index:end) ); % Rimwards
  TAdot_surf(2) = sum( TAdot_int(1:h_index-1) ); % Hubwards
  for c1 = 1:2 % Loop for each end
    % Find air thermal inertia based on mass and c_v function
    air_thermal_inertia = ( y.m_a{c1} + GP.T.m_add ) * GP.Tfun.c_va(y.T_a{c1})...
      + y.m_wg{c1} * GP.Tfun.c_pwg(y.T_a{c1}); % Includes steam
    % Conduction from wall into air;
    Tdot(2,c1) = TAdot_surf(c1) / air_thermal_inertia;
    % Conduction at relevant end of piston
    Q_piston = GP.A * (y.T_a{c1} - y.T_pe{c1}) ./ GP.T.R.func(y.T_pe{c1});% Q in W
    %   Piston end Tdot
    ydot.T_pe{c1} = Q_piston / (GP.A * GP.T.pe_t * GP.T.rho_wall * GP.T.cp_wall);
    %   Air Tdot component from piston
    Tdot(3,c1) = - Q_piston ./ air_thermal_inertia; % -ve since Q was into piston
    % Conduction at relevant end of tube
    Rth_end = GP.T.R.func(y.T_te{c1});
    Q_end_int = GP.A_tend * (y.T_a{c1} - y.T_te{c1}) ./ Rth_end; % from air side (W)
    Q_end_ext = GP.A_tend * (GP.T_kick - y.T_te{c1}) ./ Rth_end; % from exhaust side (W)
    %   Tube end Tdot
    ydot.T_te{c1} = (Q_end_int + Q_end_ext) / ...
      (GP.A * GP.T.pe_t * GP.T.rho_wall * GP.T.cp_wall);
    %   Air Tdot component from tube end
    Tdot(4,c1) = - Q_end_int ./ air_thermal_inertia;
    % >> Air mixing Tdot component
    if ydot.m_a{c1} > 0 % If air flowing in, then also mix temperatures
      if sys.mode == 4 % If kicking, base it on GP.T_kick
        mixingdT = GP.T_kick - y.T_a{c1};
      else % If in freefall, base it on GP.T_atm
        mixingdT = GP.T_atm - y.T_a{c1};
```

```
      end  % Now add in mix based on ratio of mass to mfr
      Tdot(5,c1) = mixingdT * ydot.m_a{c1} / (y.m_a{c1}+GP.T.m_add);
    end
  end
end

for c1 = 1:2
  % >> Sum all components so far for ydot.T_a
  ydot.T_a{c1} = sum(Tdot(:,c1));
  %% Phase transition
  if sys.phase{c1} == 2 % If currently changing phase ...
    % Components of complicated pdot, Tdot_sat and m_wg algebra tomfoolery.
    x_1 = (y.m_a{c1}/GP.Mol.air + y.m_wg{c1}/GP.Mol.wat)...
      * GP.R_univ / (GP.A*h_a{c1});
    x_2 = ( (ydot.m_a{c1}/GP.Mol.air) * (GP.R_univ / GP.A) - x_1*h_a_dot{c1} )...
      * y.T_a{c1} / h_a{c1};
    x_3 = GP.R_univ*y.T_a{c1} / (GP.Mol.wat*GP.A*h_a{c1});
    x_4 = GP.Tfun.dT_sat(p{c1});
    x_5 = (GP.Tfun.c_pa(y.T_a{c1}).*y.m_a{c1} ...
      + GP.Tfun.c_pwg(y.T_a{c1}).*y.m_wg{c1} ...
      + GP.Tfun.c_pwf(y.T_a{c1}).*y.m_wf{c1}) ...
      / GP.Tfun.L_water(p{c1});
    % Formula for m_wg
    ydot.m_wg{c1} = (ydot.T_a{c1}*x_5+ydot.m_wg{c1}-x_2*x_4*x_5/(1-x_1*x_4))...
      / (1 + x_3*x_4*x_5/(1-x_1*x_4));
    % Formula for Tdot_sat
    ydot.T_a{c1} = x_4*(x_2 + x_3*ydot.m_wg{c1})/(1-x_1*x_4);
    % Symmetry
    ydot.m_wf{c1} = - ydot.m_wg{c1};
  end
end

%%  WORK DONE
% Work done against pressure
ydot.wd_p = (2*GP.A-GP.A_tr)*(p{1} - p{2})*y.hdot;
% Work done against gravity
total_grav_torque = -cos(y.theta)*(GP.m_P/2)*GP.g*( 2*y.h - GP.L_T);
% Negative to get work done vs gravity torque
ydot.wd_g = -total_grav_torque*GP.omega;

%% Finalise
if ~GP.post && GP.debug_msg
  SW_debug_message(false, GP, t, y, Tdot, gamma)
end
ydot = SW_pack(ydot);
end
```

```
function p = SW_pressure(GP,ys)
% SW_pressure takes a structure ys as its input then outputs a cell array.
%   If the structure contains vectors of states, then the cell array contains vectors
%   of pressures at the corresponding states; if the structure contains a single state,
%   the cell array will merely contain single variables.
% Based on gas law (not relative to T_atm) with molar calculation for air and steam.
%  Includes adjustment for volume taken up by liquid water.
p = cell(2,1);
h_a = GP.h_a_fun(ys.h);
for c1 = 1:2
  p{c1} = GP.R_univ * (ys.m_a{c1}/GP.Mol.air + ys.m_wg{c1}/GP.Mol.wat)...
    .*ys.T_a{c1}./( GP.A*h_a{c1} - ys.m_wf{c1}/GP.rho_water);
end

end
```

```
function k_targ = SW_control_fast(GP,sys,dir,sens)
if  sys.mode == 2 && sys.dir == dir % If exhausting in this dir, use trajectory
  h_a = GP.h_a_fun(sens.y.h); % Correct for direction and half piston height
  k_targ = sys.psi{sys.dir} * h_a{sys.dir};
  % Saturation:
  k_targ(k_targ<0) = 0; % Use 0 if negative
  k_targ(k_targ>GP.C.val_sat) = GP.C.val_sat;% Use GP.C.val_sat if over GP.C.val_sat

elseif sys.mode == 5 && sys.dir == dir % If kicking in this dir, use max
  k_targ = GP.C.val_sat;

else % If not kicking or exhausting, keep exhaust valve closed!
  k_targ = 0;

end
end
```

```
function psi_out = SW_control_slow(GP,sys,sens,ControlGraphs,FastForward)
% sys = SW_control_slow(GP,sys,sens,ControlGraphs,FastForward)
%  Runs five simulations in a quincunx pattern, starting at the current model state,
%  and updates the values for sys.phi(1:2). Inputs include sens, potentially-inaccurate
% 'sensor data package' containing versions of t, y and p.
if GP.ControlGraphs, dotimestamp, fprintf('Running %s ...\n',mfilename); end
% if true, psi_out = sys.psi{sys.dir}; return; end
%% Extract sens
t0 = sens.t;
y0 = SW_pack(sens.y); % Make initial y0 vector from input sensor data state

%% Run sim for duration of slow controller to "fast-forward" it to application point
if FastForward
  % Run sim from now until [GP.C.slow_runtime] seconds in the future
  sol = ode15s(@(t,y)SW_ode(GP,sys,t,y),[t0,t0+GP.C.slow_runtime], y0);

  % Set t0 and y0 to the state at the end of that sim
  t0 = sol.x(end);
  y0 = sol.y(:,end);
  if ControlGraphs
    dotimestamp, fprintf(' - Completed fastforward, initialising simulations ...\n');
  end
end

%% Initialise perturbation array
% Build array of perturbed psi values
psi.arr = sys.psi{sys.dir} * (1 + GP.C.ptb*[0,-1,+1]);

% Initialise the array to hold the values of h_end
h_end = 0*psi.arr;

plots.name = {'baseline';'psi -';'psi +'};
if ControlGraphs
  % Create cell arrays for plotting of graphs
  plots.col = {'k';'r';'b'};
  plots.line = {'-';'-';'-'};

  plots.var = {'h',...
    sprintf('m_a{%i}',sys.dir),...
    sprintf('k_e{%i}',sys.dir)};
  plots.var_num = length(plots.var);
  for c1 = 1:plots.var_num+1
    figure(c1);
  end
end
tmp.sys = sys; % Create a temporary sys variable to use in the simulation
tmp.sys.sim = true; % So that the ODE knows it's in a simulation

%% Solve ODE 3 times
% dotimestamp, fprintf(' - Running quincunx ...\n');
for c1 = 1:3
  tmp.sys.psi{sys.dir} = psi.arr(c1); % Get trial value of psi

  simtime = tic;
  % Run sim using loop values of sys, stopping when piston stops
  sol = ode15s(@(t,y)SW_ode(GP,tmp.sys,t,y),[t0,t0+1], y0, ...
    odeset('Events',@(t,y)SW_slow_events(GP,y),'Refine',1));

  % Pack up y and y-event data
  sol.yes = SW_pack(sol.ye);
  % Save result value
  h_a = GP.h_a_fun(sol.yes.h);
  h_end(c1) = h_a{sys.dir};
  if ControlGraphs
    dotimestamp, fprintf(' - Completed %s sim in %gs; h_end = %g for psi = %g\n',...
      plots.name{c1},toc(simtime),h_end(c1),tmp.sys.psi{sys.dir});
    sol.ys = SW_pack(sol.y);
    % Plot the trajectories for each of the variables in plots.var
    for c2 = 1:plots.var_num
```

```
      figure(c2); hold on; % Switch to appropriate figure
      plotdata = eval(['sol.ys.',plots.var{c2}]);
      plot3(sol.x,t0+0*sol.x,plotdata,...
        [plots.col{c1},plots.line{c2}],...
        'DisplayName',plots.name{c1});
    end
  end
%   if c1==1 && h_end(c1)<GP.C.targ % If baseline sim is good enough
%     if ControlGraphs % Say we're done
%       dotimestamp, fprintf(' - Baseline h_end < GP.C.targ, short-circuiting.\n');
%     end
%     psi_out = sys.psi{sys.dir}; % Use current value
%     return; % Quit.
%   end
end

if ControlGraphs
  plots.var{plots.var_num+1} = 'h';
  for c1 = 1:(plots.var_num+1)
    figure(c1); hold on;
    set(gcf,'Name',plots.var{c1});
    xlabel('Simulation time (s)');
    ylabel('Timestamp (s)');
    zlabel(strrep(plots.var{c1},'_','\_'));
    title(strrep(sprintf('%s - ys.%s',mfilename,plots.var{c1}),'_','\_'));
    grid on; legend('show');
  end
  % On the last plot:
  plot3(psi.arr([2,1,3]),t0+0*psi.arr,h_end([2,1,3]),'k+-'); % Plot quincunx
  legend('hide'); xlabel('\psi value');
end

%% Do Newton-Raphson and update sys
% Gradient
grad = (h_end(3)-h_end(2)) / (psi.arr(3) - psi.arr(2));
% Multiply grad inverse by des. change in outputs to get req. change in psi
psi_out = sys.psi{sys.dir} + (GP.C.targ - h_end(1)) /grad ;

if ControlGraphs
  %% Add answer (assumed and actual) to plots
  figure(1+plots.var_num); hold on; % Go to h plot
  % Get actual h of answer
  tmp.sys.psi{sys.dir} = psi_out;
  sol = ode15s(@(t,y)SW_ode(GP,tmp.sys,t,y),[t0,t0+1], y0, ...
    odeset('Events',@(t,y)SW_slow_events(GP,y),'Refine',1));
  sol.yes = SW_pack(sol.ye);
  final_h_a = GP.h_a_fun(sol.yes.h); % Save height value
  % Plot current-assumed-actual
  plot3([sys.psi{sys.dir},psi_out,psi_out],[t0,t0,t0],[h_end(1),GP.C.targ,final_h_a{sys.dir}],'b-o');
  %% Output answer to window
  dotimestamp, fprintf(...
    ' - Altered k_e multiplier @ t = %gs from %g to %g\n',...
    t0,sys.psi{sys.dir},psi_out);
  dotimestamp, fprintf(' - %s complete.\n',mfilename);
end
end

function [value,isterminal,direction] = SW_slow_events(GP,y)
% [value,isterminal,direction] = SW_slow_events(y)
%  Subfunction to tell control_slow's sims to stop when the piston does
y = SW_pack(y);
% Stop if piston stops
value(1) = y.hdot;
% Stop if piston hits end
h_a = GP.h_a_fun(y.h);
value(2) = h_a{1} * h_a{2};
% Always stop & don't care about direction
isterminal(1:2) = 1; direction(1:2) = 0;
end
```

```
function [value,isterminal,direction] = SW_events(GP,sys,t,y)
y = SW_pack(y);
p = SW_pressure(GP,y);

% Set default values of output variables
value= ones(14,1); % Never trigger
isterminal = value; % Always stop when triggered
direction = 0*value; % Never care about direction

switch sys.mode % Direction-independant mode selector
  case 0 % Freefall
    sens = SW_sensors(GP,t,y,sys);
    if sens.y.hdot > 0, lookdir = 1; % If rimwards...
    else lookdir = 2; end % If hubwards ...
    E = SW_control_energy(GP,lookdir,sens);
    result = E(5) - E(1)*(1+sys.C.Emarg);
    if y.hdot > 0
      value(1) = result;
    elseif y.hdot < 0
      value(2) = result;
    end
  case 1  % Compressing
    value(3) = p{sys.dir} - GP.p_targ;
  case 2  % Exhausting
    value(4) = y.hdot;
  case 3 % Dumping, awaiting piston stop
    value(5) = sin(y.theta +0.0001 - GP.C.th_kick);%y.hdot; %p{sys.dir} - GP.p_atm;
  case 4  % Resting, awaiting angle
    value(6) = sin(y.theta - GP.C.th_kick);
  case 5  % Kick engaged
    value(7) = abs(y.hdot) - GP.C.kick_hdot;
  case 6  % Expanding after kick
    value(8) = p{sys.dir} - GP.p_atm;
end

if GP.water_add > 0
  for c1 = 1:2 % For each direction
    if sys.phase{c1} == 2 % If currently boiling
      event_end = c1 + [10,12]; % Set both events [11,13] or [12,14] depending on end
      value(event_end(1)) = y.m_wg{c1}; % Lower-numbered event if out of steam
      value(event_end(2)) = y.m_wf{c1}; % Higher-numbered event if out of water
      direction(event_end) = -1; % Only interested in downwards-pointing events
    else % If not currently boiling
      event_end = 8+c1; % Then set event 7+c1
      % Watch for hitting boiling point
      value(event_end) = y.T_a{c1} - GP.Tfun.T_sat(p{c1});
      if sys.phase{c1} == 3 % If we're (nominally) in steam region
        direction(event_end) = -1; % Only interested in downwards-trending crossings
      else % If we're (nominally) in liquid region
        direction(event_end) = +1; % Only interested in upwards-pointing crossings
      end
    end
  end
end

value = real(value);
end
```

```
function [E,th_E] = SW_control_energy( GP,dir,sens )
% [E,err] = SW_control_energy( GP,sys,sens ) calculates the potential and required
%  energies based on the current state and pressure, which are passed in in the 'sensor
%  data package' structure sens.
%
%  It will change which direction it's considering based on
%  internal switches, so inputs don't need to be filtered.
%
%  Outputs are:
%    E           Vector of energies:
%       E(1)        Energy required
%       E(2)        Kinetic energy
%       E(3)        Centrifugal PE
%       E(4)        Gravitational PE
%       E(5)        Total PE
%    th_E        Predicted angle when piston reaches end, for post

%% Switch direction depending on direction of interest
if dir == 1 % If rimwards...
  th_0 = sens.y.theta + pi;
else
  th_0 = sens.y.theta;
end
% Get airmass height and piston velocity
h_a = GP.h_a_fun(sens.y.h);
h_0 = h_a{dir};
h_a_dot = GP.h_a_dot_fun(sens.y.hdot);
hdot_0 = h_a_dot{dir};
% Get appropriate pressure and air temperature
p_0 = sens.p{dir};
T_a0 = sens.y.T_a{dir};
m_a0 = sens.y.m_a{dir};

E = zeros(5,1); th_E = 0;

%% Energy required
E(1) = 2 * GP.C.Ereq_fun(h_0); % 2* because double-ended
%% Kinetic & centrifugal potential energies
E(2) = 0.5*GP.m_P*(hdot_0^2);
E(3) = 0.5*GP.m_P*(GP.omega^2) * h_0 * (GP.L_T - 2*GP.L_ph - h_0);
%% Gravitational potential energy
if sin(th_0) ~= 0 % Prevent NaN errors from th_E = -Inf etc.
  % Predict the final angle of the tube
  th_E = th_0 + GP.omega * ( ...
    hdot_0 + sqrt( (hdot_0^2)+2*GP.g*sin(th_0)*h_0 )  )...
    / ( GP.g*sin(th_0) );
  % Is speed more than 3* required?
  % Cap the velocity to prevent it assuming it overshoots in the GPE calc
  limit = - 3*h_0*GP.omega/(th_E-th_0);
  hdot_0(hdot_0 > limit) = limit;
  % Calculate GPE
  E(4) = (GP.m_P*GP.g*12/((th_E-th_0)^3))...
    *((th_E-th_0)*hdot_0/(2*GP.omega)+h_0)...
    *(cos(th_0)-(th_E-th_0)*sin(th_E)-cos(th_E))...
    + ...
    (GP.m_P*GP.g*6/((th_E-th_0)^2))...
    *(-h_0-2*(th_E-th_0)*hdot_0/(3*GP.omega))...
    *(sin(th_0)-sin(th_E))...
    - ...
    GP.m_P*GP.g*hdot_0*cos(th_0)/GP.omega;
end
%% Total potential energy
E(5) = sum(E(2:4));
E = real(E);
if dir == 1
  th_E = th_E - pi; % Correct th_E for export
end

end
```

```
dotimestamp, fprintf('Running %s ...\n',mfilename);
%% xxl switching
if GP.xxl ~= -1; % if we only recorded states at events, use that as the dataset
  out.t = out.te;
  out.yv = out.yev;
end
out.yv = real(out.yv);
out.ys = SW_pack(out.yv);
out.yes = SW_pack(out.yev);

%% Initialising
dotimestamp, fprintf(' - Stepping through output data ...\n');
clear pp;
if ~isFunction || GP.PlotGraphs
  for c1 = 1:2
    pp.E{c1} = zeros(5,length(out.t));
    pp.th_E{c1} = 0*out.t;
    pp.k_targ{c1} = 0*out.t;
  end
  pp.yvdot = 0*out.yv;
  pp.Tdot = zeros(5,2,length(out.t));
  pp.accel_cmp = zeros(6,length(out.t));
  pp.sysvec.dir = 0*out.t;
  pp.sysvec.mode = 0*out.t;
  pp.sysvec.psi{1} = 0*out.t;
  pp.sysvec.psi{2} = 0*out.t;
  pp.sysvec.C.Emarg = 0*out.t;
  pp.sysvec.C.kick_hdot = 0*out.t;
  pp.sysvec.cyclenum = 0*out.t;
  pp.sysvec.phase{1} = 0*out.t;
  pp.sysvec.phase{2} = 0*out.t;

  GP.post = true; % just in case anything needs to know we're in post

  %% Step through for non-vectorised values
  pp.p = SW_pressure(GP,out.ys); % get all pressures at once
  if GP.WaitBar, h = waitbar(0,'Stepping through ys ...'); end
  for c1 = 1:length(out.t)
    sys = out.sys(c1); % look up the 'current' value of sys
    pp.sysvec.dir(c1) = sys.dir; % de-vectorise out.sys.dir, .mode, .psi, .phase and .Ereq
    pp.sysvec.mode(c1) = sys.mode;
    pp.sysvec.psi{1}(c1) = sys.psi{1};
    pp.sysvec.psi{2}(c1) = sys.psi{2};
    pp.sysvec.phase{1}(c1) = sys.phase{1};
    pp.sysvec.phase{2}(c1) = sys.phase{2};
    pp.sysvec.C.Emarg(c1) = sys.C.Emarg;
    pp.sysvec.cyclenum(c1) = sys.cyclenum;
    sens = SW_sensors( GP, out.t(c1), SW_pack(out.yv(:,c1)), sys );% pass through sensors
    for c2 = 1:2 % get e, th_e and k_targ for both ends
      [pp.E{c2}(:,c1),pp.th_E{c2}(c1)]... % get energies and th_e
        = SW_control_energy(GP,c2,sens);
      pp.k_targ{c2}(c1) = SW_control_fast(GP,sys,c2,sens);
    end % get rates for all the state variables
    [pp.yvdot(:,c1),pp.Tdot(:,:,c1),pp.accel_cmp(:,c1)]...
      = SW_ode(GP,sys,out.t(c1),out.yv(:,c1));
    if GP.WaitBar, waitbar(c1/length(out.t),h); end
  end
  if GP.WaitBar, close(h); end
  pp.ysdot = SW_pack(pp.yvdot);

  %% Vectorised processes
  dotimestamp, fprintf(' - Running vectorised postprocesses ...\n');
  h_a = GP.h_a_fun(out.ys.h);
  for c1 = 1:2
    pp.th_E{c1} = real(pp.th_E{c1}); % reify th_E
    % energy surplus
    pp.dE{c1} = pp.E{c1}(5,:) - pp.E{c1}(1,:);
    % valve area
    pp.valve_area{c1} = -pp.ysdot.m_a{c1} ./ ...
```

```
      sqrt(2.*(pp.p{c1}-GP.p_targ).*out.ys.m_a{c1}./h_a{c1});
    pp.valve_area{c1} = real(pp.valve_area{c1});
    % specific volume & gamma values
    pp.spec_vol{c1} = h_a{c1}*GP.A ./ out.ys.m_a{c1};
    pp.gam.tv{c1} = 1 + log(out.ys.T_a{c1}./GP.T_atm) ./ ...
      log((GP.T_atm*(GP.R_univ/GP.Mol.air)/GP.p_atm)./pp.spec_vol{c1});
    pp.gam.tp{c1} = ( 1 - log(out.ys.T_a{c1}./GP.T_atm) ./ ...
      log(pp.p{c1}./GP.p_atm) ).^-1;
    pp.gam.pv{c1} = log(pp.p{c1}./GP.p_atm) ./ ...
      log((GP.T_atm*(GP.R_univ/GP.Mol.air)/GP.p_atm)./pp.spec_vol{c1});
  end
  % Rotor-affecting variables: coriolis forces, piston weight forces...
  pp.piston{1}.rad = out.ys.h + GP.L_hubrad; % Distance from hub to alpha piston CoG
  pp.piston{2}.rad = GP.L_T + GP.L_hubrad - out.ys.h; % Distance from hub to beta piston CoG

  pp.piston{1}.sign = 1; pp.piston{2}.sign = -1; % Reflects turnwise change for beta piston
  pp.rotor.torq.cori = 0; % Initialise cumulatives
  pp.rotor.torq.grav = 0;
  pp.rotor.torq.net = 0;
  for c1 = 1:2
    % Coriolis force
    pp.piston{c1}.corio = -2*(GP.m_P/2)*GP.omega.*out.ys.hdot * pp.piston{c1}.sign;
    % Torque due to coriolis force
    pp.piston{c1}.torq.cori = pp.piston{c1}.corio .* pp.piston{c1}.rad;
    pp.rotor.torq.cori = pp.rotor.torq.cori + pp.piston{c1}.torq.cori;% Cumulative
    % Torque due to gravity
    pp.piston{c1}.torq.grav = -cos(out.ys.theta).*(GP.m_P/2)*GP.g.*pp.piston{c1}.rad..
      * pp.piston{c1}.sign;
    pp.rotor.torq.grav = pp.rotor.torq.grav + pp.piston{c1}.torq.grav;% Cumulative
    % Net torque
    pp.piston{c1}.torq.net = pp.piston{c1}.torq.cori + pp.piston{c1}.torq.grav;
    pp.rotor.torq.net = pp.rotor.torq.net + pp.piston{c1}.torq.net;% Cumulative
  end
  % Rotational inertia due to pistons
  pp.rotor.inertia = (GP.m_P/2) * (pp.piston{1}.rad.^2 + pp.piston{2}.rad.^2);
end

%% One-off calculations
if out.t(end)>150 % If we have more than 150s of data
  % Only base the linefits on the last 150s of data
  pp.power.index = find(out.t<out.t(end)-150,1,'last');
else % Otherwise use all of it
  pp.power.index = 1;
end
% Work done vs pressure
pp.power.wd_p = polyfit(out.t(pp.power.index:end),out.ys.wd_p(pp.power.index:end)',1);% linear
pp.power.wd_p = pp.power.wd_p(1); % only want first coefficient
% Work done vs gravity
pp.power.wd_g = polyfit(out.t(pp.power.index:end),out.ys.wd_g(pp.power.index:end)',1);% linear
pp.power.wd_g = pp.power.wd_g(1); % only want first coefficient
% Air exhausted
pp.power.m.net = polyfit(out.t(pp.power.index:end),out.ys.m_out.net(pp.power.index:end)',1);
pp.power.m.net = pp.power.m.net(1);
pp.power.m.exh = polyfit(out.t(pp.power.index:end),out.ys.m_out.exh(pp.power.index:end)',1);
pp.power.m.exh = pp.power.m.exh(1);

%% Finished - pass to plotgraphs if needed
dotimestamp, fprintf(' - %s complete.\n',mfilename);
if isFunction
  out.pp = pp; % Store that
end
if GP.PlotGraphs
  SW_plotgraphs;
end
```

```
dotimestamp, fprintf('Running %s ...\n',mfilename);
%% Preparation
clear plots;
if ~exist('file','var') || ~isstruct(file) || ~isfield(file,'name')
  % If there's no filename for this run
  file.name = '[not saved]';
end
set(0,'DefaultFigureWindowStyle','docked');
set(0,'DefaultFigureColor','w');
set(0,'defaultAxesXGrid','on');
set(0,'defaultAxesYGrid','on');
set(0,'defaultAxesZGrid','on');

%% Create common surtitle for all plots
% variables to include
plots.NoTitles = false; % Set to true to exclude titles
if plots.NoTitles
  plots.autotitle = @(str) []; % Never add a title
else
  % Variables to include
  plots.subtitlevars = {'GP.omega','GP.C.th_kick','GP.C.kick_hdot','file.name'};
  % Build subtitle accordingly
  plots.subtitle = [];
  for c1 = 1:length(plots.subtitlevars)
    if ischar(eval(plots.subtitlevars{c1}))
      plots.subtitletype = '%s';
    else
      plots.subtitletype = '%g';
    end
    plots.subtitle = sprintf('%s%s = %s,  ',...
      plots.subtitle,plots.subtitlevars{c1},...
      sprintf(plots.subtitletype,eval(plots.subtitlevars{c1})));
  end
  plots.subtitle = [plots.subtitle(1:end-3),10]; % delete last comma and add a newline
  % Create anonymous function including underscore fixer
  plots.autotitle = @(str) title(strrep([plots.subtitle, str],'_','\_'));

  % Alternate fancy title for thesis - no actual title
  plots.autotitle = @(str) title(str);
end
plots.floatstr = @(loc) SW_cornertext(...
  ['$\dot{\theta} = ',sprintf('%g',GP.omega),'$ rad/s',10,...
  '$\dot{h}_\mathrm{kick} = ',sprintf('%g',GP.C.kick_hdot),'$ m/s',10,...
  '$\theta_\mathrm{kick} = ',sprintf('%g',GP.C.th_kick),'$ rad'],...
  loc, true);

%% Set common x-axis
plots.axes = [];

if true % true = plots use time as dependant variable, false = plots use angle
  plots.xvar = out.t;%-out.te(6); %#ok<*UNRCH>
  plots.xlab = 'Time (s)';% since event 6';
  plots.xlims = [0,Inf];%[out.te(find(out.ie==4,1,'first')),out.te(find(out.ie==6,1,'first'))];
  % plots.xevents = out.te;
else
  plots.xvar = mod(out.ys.theta,pi);
  plots.xlab = '\theta (rad)';
  plots.xlims = [out.ys.theta(1) out.ys.theta(end)];
  % plots.xevents = out.ye.theta;
end

plots.xlimsind{1} = find(plots.xvar>=plots.xlims(1), 1);
plots.xlimsind{2} = find(plots.xvar<=plots.xlims(2), 1,'last');

plots.xsub = plots.xlimsind{1}:plots.xlimsind{2}; % X-points we're going to plot
if length(plots.xsub)>750 % If we're going to try and plot more than 750 points
  plots.xsub = round(linspace(plots.xsub(1),plots.xsub(end),750));
end
plots.opsub = round(linspace(1,length(GP.T.op.xvals),100)); % Only plot 100 points
```

```
pp.Tsurf = GP.T.op.Ap(plots.opsub,:) * out.ys.opthets(:,plots.xsub);% max 100×750
pp.Tsurf_diff = pp.Tsurf - GP.T.op.Ap(plots.opsub,:) ...
  * repmat(out.ys.opthets(:,1),1,750); % max 100×750
plots.dircol{1} = 'r'; plots.dircol{2} = 'b';

%% Tdot components
% plots.Tdot.sz = size(pp.Tdot,1);
% plots.Tdot.cols = get_colour_spec(plots.Tdot.sz);
% plots.Tdot.names = {'Ideal gas'; 'Wall conduction'; ...
%   'From piston end'; 'From tube end';'From air mixing'};
% for c1 = 1:2
%   figure('Name',['Tdot in dir ',num2str(c1)]); hold on;
%   for c2 = 1:plots.Tdot.sz
%     plots.Tdot.v = pp.Tdot(c2,c1,:);
%     plots.Tdot.v = plots.Tdot.v(:);
%     plot(plots.xvar,plots.Tdot.v,...
%       'Color',plots.Tdot.cols(c2,:),...
%       'DisplayName',plots.Tdot.names{c2});
%   end
%   h = plot(plots.xvar,pp.ysdot.T{c1},'k');
%   set(h,'DisplayName','Total');
%   plots.autotitle(['Tdot output in dir ',num2str(c1),' (1=rimwards)']);
%   ylabel('Tdot (K per second)');
%   legend('toggle','Location','Best')
%   SW_plotgraphs_tweaks;
%
%   for c2 = 1:plots.Tdot.sz
%     figure('Name',sprintf('dir%i Tdot, %s',c1,plots.Tdot.names{c2}));  hold on;
%     plots.Tdot.v = pp.Tdot(c2,c1,:);
%     plots.Tdot.v = plots.Tdot.v(:);
%     plot(plots.xvar,plots.Tdot.v,plots.dircol{c1});
%     plots.autotitle(sprintf(...
%       '"%s" component of Tdot in the %i direction',plots.Tdot.names{c2},c1));
%     ylabel('Tdot (K per second)');
%     SW_plotgraphs_tweaks;
%   end
% end

%% Position, velocity and forces
figure('Name','Position and velocity'); hold on;
subplot(2,1,1), plot(plots.xvar,out.ys.h,'k'); hold on;
plot(plots.xvar([1,end]),GP.L_ph*[1,1],'k--');
plot(plots.xvar([1,end]),(GP.L_T-GP.L_ph)*[1,1],'k--');
plots.autotitle('Position and velocity'); ylabel('h (m)');
SW_plotgraphs_tweaks;
subplot(2,1,2), plot(plots.xvar,out.ys.hdot,'m');
plots.autotitle('Position and velocity'); ylabel('h-dot (m/s)');
SW_plotgraphs_tweaks;

plots.title = 'Force components';
figure('Name',plots.title); hold on;
plots.force.names = {'Pressure','Gravity','Centrifugal',...
  'Friction','Shock absorber','Net'};
plots.force.cols = get_colour_spec(length(plots.force.names)-1);
plots.force.cols = [ plots.force.cols; [0,0,0] ];% Add black for the last
for c1 = 1:length(plots.force.names)
  plot(plots.xvar,pp.accel_cmp(c1,:)*GP.m_P,...
    'Color',plots.force.cols(c1,:),...
    'DisplayName',plots.force.names{c1}); hold on;
end
plots.autotitle(plots.title);
ylabel('Magnitude (N)');
legend('show');
SW_plotgraphs_tweaks;

%% Polar plots
plots.title = 'Cartesian CoM trajectory';
figure('Name',plots.title); hold on;
```

```
c1 = [linspace(0,pi/2,1e3),linspace(3*pi/2,2*pi,1e3)]';
[plots.pol.rim.x, plots.pol.rim.y] = pol2cart(c1, GP.L_T/2-GP.L_hubrad);
plots.pol.ideal.r = c1*0 + (GP.L_T/2 - GP.L_hubrad);
[plots.pol.ideal.x, plots.pol.ideal.y] = pol2cart(c1,plots.pol.ideal.r);
[plots.pol.x, plots.pol.y] = pol2cart(out.ys.theta(1e4:end),out.ys.h(1e4:end)-GP.L_T/2);
h = plot(0,0,'ko'); % Plot a centre
SW_nolegend(h);
% hLine = plot(plots.pol.rim.x, plots.pol.rim.y,'k');
% set(get(get(hLine,'Annotation'),'LegendInformation'),...
%   'IconDisplayStyle','off'); % Hide that from the legend
plot(plots.pol.ideal.x, plots.pol.ideal.y,'b',...
  'DisplayName','Ideal');
plot(plots.pol.x, plots.pol.y,'k',...
  'DisplayName','Actual');
plots.autotitle(plots.title); axis equal; legend('Location','SouthWest');
set(gca,'XLim',[-1,1]*GP.L_T/2,'YLim',[-1,1]*GP.L_T/2);
xlabel('Horizontal position (m)');
ylabel('Vertical position (m)');
plots.floatstr('TL');

plots.title = 'Polar plot of gravity torque';
figure('Name',plots.title);
% subplot(3,1,1:2);
hold on;
c1 = linspace(0,pi,2e3)'; c1(end) = [];
plots.pol.torq.ideal.r = - cos(c1).*GP.g*GP.m_P.*(GP.L_T/2 - GP.L_hubrad);
% Gravity torque
plots.pol.torq.actual.r = -cos(out.ys.theta).*(out.ys.h-GP.L_T/2)*GP.g*GP.m_P;
[plots.pol.torq.ideal.x, plots.pol.torq.ideal.y] = ...
  pol2cart(c1,plots.pol.torq.ideal.r);
plots.pol.torq.actual.th = mod(out.ys.theta(1e4:end)+pi/2,pi)-pi/2;
[plots.pol.torq.actual.x, plots.pol.torq.actual.y] = ...
  pol2cart(plots.pol.torq.actual.th,plots.pol.torq.actual.r(1e4:end));
h = plot(0,0,'ko'); % Plot a centre
SW_nolegend(h);
plot(plots.pol.torq.ideal.x, plots.pol.torq.ideal.y,'b',...
  'DisplayName','Ideal');
plot(plots.pol.torq.actual.x, plots.pol.torq.actual.y,'k',...
  'DisplayName','Actual');
axis equal; legend show;
set(gca,'XLim',[-20,5]*1e6,'YLim',[-10,10]*1e6);
plots.autotitle(plots.title);
xlabel('Torque (Nm)'); ylabel('Torque (Nm)');
plots.floatstr('BR');

plots.title = 'Gravity torque';
figure('Name',plots.title);
% subplot(3,1,3);
hold on;
% plot(out.ys.theta, interp1q(c1,plots.pol.torq.ideal.r,mod(out.ys.theta,2*pi)'),'b',...
%   'DisplayName','Ideal');
% plot(out.ys.theta, plots.pol.torq.actual.r,'k',...
%   'DisplayName','Actual')
% Mod version - one loop with multiple traces
plot(mod(c1,pi),-abs(plots.pol.torq.ideal.r),'b',...
  'DisplayName','Ideal');
plots.pol.torq.actual.th2 = mod(out.ys.theta(1e4:end),pi);
plots.pol.torq.actual.dth = [diff(plots.pol.torq.actual.th2),1];
plots.pol.torq.actual.th2(plots.pol.torq.actual.dth<0)=NaN;
plot(plots.pol.torq.actual.th2,plots.pol.torq.actual.r(1e4:end),'k',...
  'DisplayName','Actual')
xlabel('Tube angle  \theta (rad)'); ylabel('Total gravity torque (Nm)');
legend('location','NorthEast');
set(gca,'XLim',[0,pi]);...
%   'XTick',pi*(-0.5:0.25:1.5),...
%   'XTickLabel',{'-pi/2','-pi/4','0','pi/4','pi/2'});
% plots.autotitle(plots.title);
plots.floatstr('TL');
```

```
%% Pressure, temperature & density
figure('Name','Air pressures'); hold on;
plot(plots.xvar,GP.p_targ+0*plots.xvar,'k--');
for c1 = 1:2
  plot(plots.xvar,pp.p{c1},plots.dircol{c1});
end
plots.autotitle('Air pressures'); ylabel('Pressure (Pa)');
legend('p_{targ}','dir 1','dir 2');
set(gca,'YScale','log');
SW_plotgraphs_tweaks;

figure('Name','Air temperatures'); hold on;
% plot(plots.xvar,GP.Tfun.T_sat(pp.p{2}),'k--','DisplayName','T_{sat}(p_2)');
for c1 = 1:2
  plot(plots.xvar,out.ys.T_a{c1},plots.dircol{c1},...
    'DisplayName',['dir ',num2str(c1)]);
end
plots.autotitle('Air/steam/water temperatures'); ylabel('Temperature (K)');
legend('show');
SW_plotgraphs_tweaks;

figure('Name','Air densities'); hold on;
for c1 = 1:2
  plot(plots.xvar,pp.spec_vol{c1}.^-1,plots.dircol{c1});
end
legend('dir 1','dir 2');
plots.autotitle('Air densities'); ylabel('Density (kg/m^3)');
SW_plotgraphs_tweaks;

%% Water
figure('Name','Water masses'); hold on;
for c1 = 1:2
  plot(plots.xvar,out.ys.m_wf{c1},['-',plots.dircol{c1}],...
    'DisplayName',sprintf('m_{wf} in dir %i',c1));
  plot(plots.xvar,out.ys.m_wg{c1},['--',plots.dircol{c1}],...
    'DisplayName',sprintf('m_{wg} in dir %i',c1));
end
plots.autotitle('Water masses'); ylabel('Mass of water (kg)');
legend('show');
SW_plotgraphs_tweaks;

%% Air mass, mass flow rate & volumetric flow rate
figure('Name','Air masses'); hold on;
for c1 = 1:2
  plot(plots.xvar,out.ys.m_a{c1},plots.dircol{c1});
end
plot(plots.xvar,out.ys.m_out.net,'m');
legend('dir 1','dir 2','Net exhausted',...
  'Location','NorthWest');
ylabel('m (kg)');
plots.autotitle('Air masses');
if max(out.ys.m_out.net) > 2.5*max(out.ys.m_a{1})
  set(gca,'Ylim',[0, 2.5*max(out.ys.m_a{1})]);
end
SW_plotgraphs_tweaks;

for c1 = 1:2
  figure('Name',sprintf('Air mass & mass flow rate - dir %i',c1)); hold on;
  subplot(2,1,1), plot(plots.xvar,out.ys.m_a{c1},plots.dircol{c1});
  plots.autotitle(sprintf('dir %i air mass',c1));
  ylabel('Air mass (kg)'); SW_plotgraphs_tweaks;
  subplot(2,1,2), plot(plots.xvar,pp.ysdot.m_a{c1},plots.dircol{c1});
  title(sprintf('dir %i air mass flow rate',c1));
  ylabel('Air mass flow rate (kg/s)'); SW_plotgraphs_tweaks;

  figure('Name',sprintf('Air volume & volumetric flow rate - dir %i',c1)); hold on;
  subplot(2,1,1), plot(plots.xvar,out.ys.m_a{c1}.*pp.spec_vol{c1},plots.dircol{c1});
  plots.autotitle(sprintf('dir %i air volume',c1));
  ylabel('Air volume (m^3)'); SW_plotgraphs_tweaks;
```

```
  subplot(2,1,2), plot(plots.xvar,pp.ysdot.m_a{c1}.*pp.spec_vol{c1},plots.dircol{c1});
  title(sprintf('dir %i air volumetric flow rate',c1));
  ylabel('Air volumetric flow rate (m^3/s)'); SW_plotgraphs_tweaks;
end

%% Theta_E predictions
plots.title = 'thetalock_predictions';
figure('Name',plots.title); hold on;
plots.dircol_rgb = [1,0,0;0,0,1]; % Set up red and blue
plots.dircol_rgb_lite = 1 - (1-plots.dircol_rgb)*0.3; % Make lighter
plots.th_E = cell(2,1);
for c1 = 1:2
  plots.th_E{c1}.theta = [];
  plots.th_E{c1}.th_E = [];
end
ind.stop = find(out.ie==4); % Find all the stopping times
for c1 = 1:length(ind.stop) % For each of those
  % Find the immediately preceeding switch into freefall
  ind.freefall = find(out.ie(1:ind.stop(c1))==7,1,'last');
  if ~isempty(ind.freefall) % If that worked
    ind.t1 = find(out.t==out.te(ind.freefall),1); % Find the index in out.t
  else
    ind.t1 = 1; % Otherwise go from 1
  end
  % Find the stopping entry in out.t
  ind.t2 = find(out.t==out.te(ind.stop(c1)));
  % Find out which direction it was in
  ind.dir = pp.sysvec.dir(ind.t2-1);
  % Build up the vectors
  plots.th_E{ind.dir}.theta = [plots.th_E{ind.dir}.theta;...
    NaN;out.ys.theta(ind.t1:ind.t2)'];
  plots.th_E{ind.dir}.th_E = [plots.th_E{ind.dir}.th_E;...
    NaN;pp.th_E{ind.dir}(ind.t1:ind.t2)];
  % And the error signal
  plots.th_E2(c1).theta = out.ys.theta(ind.t1:ind.t2)'-out.ys.theta(ind.t2);
  plots.th_E2(c1).th_E = pp.th_E{ind.dir}(ind.t1:ind.t2)-pp.th_E{ind.dir}(ind.t2);
end
for c1 = 1:2
  h = plot(out.ys.theta,pp.th_E{c1},... % Plot the whole lines
    'Color',plots.dircol_rgb_lite(c1,:)); hold on; % ... faded
  SW_nolegend(h);
end
plot(...
  [out.yes.theta(out.ie==1),out.yes.theta(out.ie==2)]',...
  [interp1q(out.ys.theta',pp.th_E{1},out.yes.theta(out.ie==1)');...
  interp1q(out.ys.theta',pp.th_E{2},out.yes.theta(out.ie==2)')],'kx',...
  'DisplayName','Compression started'); % Plot the compression started points
plot(out.yes.theta(out.ie==4),out.yes.theta(out.ie==4),'kd',...
  'DisplayName','Piston stopped'); % Plot the piston stopped points
for c1 = 1:2
  plot(plots.th_E{c1}.theta,plots.th_E{c1}.th_E,... % Plot the relevant bits
    'Color',plots.dircol_rgb(c1,:),... % ... not faded
    'DisplayName',sprintf('Direction %i',c1)); hold on;
end
title(plots.title); legend('show','Location','SouthEast'); grid on;
ylabel('Predicted $\theta_\mathrm{lock}$ value (rad)','interpreter','latex');
xlabel('\theta (rad)');
plots.xlimsgot = get(gca,'Xlim');
set(gca,'Ylim', plots.xlimsgot);

% Alternative method (plotting just error signals)
plots.title = 'thetalock_errors';
figure('Name',plots.title); hold on;
for c1 = 1:length(plots.th_E2)
  plot(plots.th_E2(c1).theta,plots.th_E2(c1).th_E,'k');
end
title(plots.title); title([]);
ylabel('Error in $\theta_\mathrm{lock}$ prediction (rad)','interpreter','latex');
xlabel('Angle before $\theta_\mathrm{lock}$ (rad)','interpreter','latex');
```

```
%% Energies and energy surplus
plots.E.names = {'Energy required','Kinetic PE',...
  'Centrifugal PE','Gravitational PE','Total PE'};
plots.E.style = {'k','b--',...
  'b:','b-.','b-'};
for c1 = 1:2
  plots.title = sprintf('Energies in dir %i',c1);
  figure('Name',plots.title); hold on;
  for c2 = 1:5
    plot(plots.xvar,pp.E{c1}(c2,:),plots.E.style{c2},...
      'DisplayName',plots.E.names{c2});
  end
  %   plot(plots.xvar,out.ys.wd,'m',...
  %     'DisplayName','Work done');
  plots.autotitle(plots.title);
  ylim([0,Inf]);
  ylabel('Energy (J)');
  legend('show',...
    'Location','NorthWest')
  % if max(out.ys.wd) > 2.5*max(pp.E(5,:))
  %     set(gca,'Ylim',[0, 2.5*max(pp.E(5,:))]);
  % end
  SW_plotgraphs_tweaks;

  plots.title = sprintf('Surplus energy in dir %i',c1);
  figure('Name',plots.title); hold on;
  plot(plots.xvar(pp.sysvec.dir==c1),pp.dE{c1}(pp.sysvec.dir==c1),'k-');
  plots.autotitle(plots.title);  ylabel('Energy surplus');
  SW_plotgraphs_tweaks;
end

%% Power and kg-per-s fitted lines
figure('Name','Work done'); hold on;
if out.t(end)>150 % If we have more than 150s of data
  % Only base the linefits on the last 150s of data
  plots.pow.index = find(out.t<out.t(end)-150,1,'last');
else % Otherwise use all of it
  plots.pow.index = 1;
end
% Linear polyfits:
plots.pow.polys.m.net = polyfit(plots.xvar(plots.pow.index:end), ...
  out.ys.m_out.net(plots.pow.index:end)',1); % Net airmass
plots.pow.polys.m.exh = polyfit(plots.xvar(plots.pow.index:end), ...
  out.ys.m_out.exh(plots.pow.index:end)',1); % Exh airmass
plots.pow.polys.wd_p = polyfit(plots.xvar(plots.pow.index:end), ...
  out.ys.wd_p(plots.pow.index:end)',1); % Work done
plots.pow.polys.wd_g = polyfit(plots.xvar(plots.pow.index:end), ...
  out.ys.wd_g(plots.pow.index:end)',1); % Work done

% Work done
hold on;
plot(plots.xvar,out.ys.wd_p,'b','DisplayName','Pressure method');
h = plot(plots.xvar(plots.pow.index:end),polyval(plots.pow.polys.wd_p, ...
  plots.xvar(plots.pow.index:end)),'b--');
SW_nolegend(h);
plot(plots.xvar,out.ys.wd_g,'r','DisplayName','Gravity method');
h = plot(plots.xvar(plots.pow.index:end),polyval(plots.pow.polys.wd_g, ...
  plots.xvar(plots.pow.index:end)),'r--');
SW_nolegend(h);
plots.autotitle(...
  sprintf('Work done with linefits (pressure = %g MW, gravity = %g MW)'...
  ,pp.power.wd_p/1e6,pp.power.wd_g/1e6));
ylabel('Work done (J)'); legend('Location','SouthEast');
SW_plotgraphs_tweaks;
plots.floatstr('TL');

% Air exhausted
figure('Name','Air exhaust rates'); hold on;
```

```
plot(plots.xvar,out.ys.m_out.net,'b',...
  'DisplayName',sprintf('Net (%g kg/s avg)',pp.power.m.net));
h = plot(plots.xvar(plots.pow.index:end),polyval(plots.pow.polys.m.net,...
  plots.xvar(plots.pow.index:end)),'b--');
SW_nolegend(h);
plot(plots.xvar,out.ys.m_out.exh,'r',...
  'DisplayName',sprintf('Gross (%g kg/s avg)',pp.power.m.exh));
h = plot(plots.xvar(plots.pow.index:end),polyval(plots.pow.polys.m.exh,...
  plots.xvar(plots.pow.index:end)),'r--');
SW_nolegend(h);
plots.autotitle(sprintf(...
  'Air exhausted linefits (net rate = %g kg/s, exh rate = %g kg/s, ratio = %g)',...
  pp.power.m.net,pp.power.m.exh, pp.power.m.exh/pp.power.m.net));
ylabel(sprintf('Mass of %g MPa air exhausted (kg)',GP.p_targ/1e6));
legend('show','Location','SouthEast');
SW_plotgraphs_tweaks;
plots.floatstr('TL');

%% Ereq_margin tuning
figure('Name','Ereq_margin tuning'); hold on;
subplot(2,1,1), plot(plots.xvar,pp.sysvec.C.Emarg,'k');
ylabel('Value'); plots.autotitle('Ereq_margin value');
SW_plotgraphs_tweaks;
plots.dump.times = out.te(out.ie==4); % Find all start-of-dump event times
for c1 = 1:length(plots.dump.times)
  index = find(out.t==plots.dump.times(c1));
  plots.dump.xvar(c1) = plots.xvar(index);
  plots.dump.h_a_temp = GP.h_a_fun(out.ys.h(index));
  plots.dump.h_a(c1) = min(plots.dump.h_a_temp{1}, plots.dump.h_a_temp{2});
end
subplot(2,1,2), hold on;
plot(plots.xvar,GP.C.Emarg.h_targ(pp.sysvec.cyclenum),'m','DisplayName','Target height');
plot(plots.dump.xvar,plots.dump.h_a,'k','DisplayName','Actual height');
ylabel('Height of airmass at start of dump (m)'); title('Air mass height');
legend('show','Location','NorthEast'); SW_plotgraphs_tweaks;

%% Valve constants
for c1=1:2
  plots.title = sprintf('dir %i valve constant',c1);
  figure('Name',plots.title);
  subplot(4,1,1),
  plot(plots.xvar,pp.sysvec.psi{c1},'m');
  ylabel('\psi');
  SW_plotgraphs_tweaks;
  plots.autotitle(plots.title);
  subplot(4,1,2:4),
  plot(plots.xvar,pp.k_targ{c1},'k',plots.xvar,out.ys.k_e{c1},plots.dircol{c1});
  legend('k_{ref}','k_{HP}');
  ylabel('HP valve constant (kg/s per Pa)')
  SW_plotgraphs_tweaks;
end

%% Valve area
figure('Name','Valve area'); hold on;
for c1 = 1:2
  plot(plots.xvar,pp.valve_area{c1},plots.dircol{c1});
end
legend('dir 1','dir 2');
plots.autotitle('HP valve xsec area');
ylabel('HP valve area (m^2)');
SW_plotgraphs_tweaks;

%% Gamma values & T versus p
figure('Name','Gamma values')
for c1=1:2
  subplot(3,1,1), hold on, plot(plots.xvar,pp.gam.tv{c1},plots.dircol{c1});
end
title('T & v based gamma');
SW_plotgraphs_tweaks;
```

```
for c1=1:2
  subplot(3,1,2), hold on, plot(plots.xvar,pp.gam.tp{c1},plots.dircol{c1});
end
title('T & p based gamma');
SW_plotgraphs_tweaks;
for c1=1:2
  subplot(3,1,3), hold on, plot(plots.xvar,pp.gam.pv{c1},plots.dircol{c1});
end
title('p & v based gamma');
SW_plotgraphs_tweaks;

figure('Name','T versus p plots'); hold on;
plots.p_scale = logspace(0.9*log10(GP.p_atm),1.1*log10(GP.p_targ),300)';
plot(plots.p_scale,GP.Tfun.T_sat(plots.p_scale),'k--');
for c1 = 1:2
  plot(pp.p{c1},out.ys.T_a{c1},plots.dircol{c1});
end
set(gca,'XScale','log');
legend({'T_{sat}','dir 1','dir 2'},'Location','NorthWest');
plots.autotitle('T vs p'); ylabel('T (K)'); xlabel('p (Pa)'); grid on;

%% Tube- and piston-end temperatures
figure('Name','Tube-end temperatures'); hold on;
for c1 = 1:2
  plot(plots.xvar,out.ys.T_te{c1}-out.ys.T_te{1}(1),plots.dircol{c1});
end
legend('dir 1','dir 2');
plots.autotitle('Relative tube-end temperatures');
ylabel(sprintf('Temperature difference from %gK starting temp (K)',out.ys.T_te{1}(1)));
SW_plotgraphs_tweaks;

figure('Name','Piston-end temperatures'); hold on;
for c1 = 1:2
  plot(plots.xvar,out.ys.T_pe{c1}-out.ys.T_pe{1}(1),plots.dircol{c1});
end
legend('dir 1','dir 2');
plots.autotitle('Relative piston-end temperatures');
ylabel(sprintf('Temperature difference from %gK starting temp (K)',out.ys.T_pe{1}(1)));
SW_plotgraphs_tweaks;

%% Wall temperatures
dotimestamp, fprintf(' - Plotting temperature figures ...\n');
figure('Name','T_wall diff contours')
[conts,h] = contour3(plots.xvar(plots.xsub),GP.T.op.xvals(plots.opsub),...
  pp.Tsurf_diff); hold on; SW_nolegend(h);
set(h,'LineWidth',2);
h2 = clabel(conts,h,'LabelSpacing',300);
% h2 = clabel(conts,h,'manual');
SW_nolegend(h2);
plot(plots.xvar,out.ys.h,'k','DisplayName','Piston trajectory');
ylabel('Position $z$ (m)','interpreter','latex');
zlabel('Temperature (K)');
plots.autotitle('Contour plot of wall temperatures');
h = colorbar;
ylabel(h,sprintf('Temperature difference from %g K (K)',GP.T_atm));
view([0,90]);
SW_plotgraphs_tweaks;

figure('Name','T_wall diff mesh')
mesh(plots.xvar(plots.xsub),GP.T.op.xvals(plots.opsub),pp.Tsurf_diff);
ylabel('Position along tube (m)');
zlabel(sprintf('Temperature difference from %g K (K)',GP.T_atm));
plots.autotitle('Mesh plot of wall temperature changes');
view(52,23); %shading interp;
SW_plotgraphs_tweaks;

figure('Name','Wall temperature mesh')
mesh( plots.xvar(plots.xsub), GP.T.op.xvals(plots.opsub), pp.Tsurf-GP.T_atm );
view(52,23); %shading interp;
```

```
set(gca,'ylim',[0,GP.L_T]);
ylabel('Position along tube (m)');
zlabel(sprintf('Temperature difference from ambient of %g K (K)',GP.T_atm));
plots.autotitle('Mesh plot of wall temperatures');
SW_plotgraphs_tweaks;

% for c1 = 1:GP.T.npolys
%     figure('Name',sprintf('Theta %i vs time',c1));
%     plot(plots.xvar,out.ys.opthets(c1,:),'k');
%     plots.autotitle(sprintf('Coefficient of the %i orthogonal polynomial',c1));
%     SW_plotgraphs_tweaks;
% end

% figure('Name','Wall temperature meshes')
%  subplot(2,1,1), mesh( plots.xvar(plots.xsub), GP.T.op.xvals(plots.opsub), pp.Tsurf );
% view(-110,29);
% ylabel('Position along tube (m)');
% zlabel('Temperature(K)');
% SW_plotgraphs_tweaks;
%  subplot(2,1,2), mesh( plots.xvar(plots.xsub), GP.T.op.xvals(plots.opsub), pp.Tsurf-GP.T_atm );
% view(-110,29);
% ylabel('Position along tube (m)');
% zlabel(sprintf('Temperature difference from ambient of %gK (K)',GP.T_atm));
% SW_plotgraphs_tweaks;

%% Ending
linkprop(plots.axes,'Xlim'); % Ensure all plots have the same x-axis limits
set(gca,'Xlim',plots.xlims); % and set those limits to plots.xlims
dotimestamp, fprintf(' - %s complete.\n',mfilename);
```

```
function yout = SW_pack( yin )
% yout = SW_pack( yin )
% if yin is a row vector, SW_pack will return a structure yout which contains fields
%              (which may be themselves cell arrays or scalars) for each of the state
%              variables in yin.
% if yin is a matrix, SW_pack will return the same structure, but every variable
%              will be a vector (and every cell array will contain vectors not
%              scalars) representing the columns.
% if yin is a structure, SW_pack will return a matrix yout which contains all the
%              fields in yin, sorted into the appropriate order to be transformed back.
if isstruct(yin)
  yout = zeros(yin.vecsize,length(yin.theta));
  yout(1,:) = yin.theta;
  yout(2,:) = yin.h;
  yout(3,:) = yin.hdot;
  yout(4,:) = yin.m_a{1};
  yout(5,:) = yin.m_a{2};
  yout(6,:) = yin.m_wf{1};
  yout(7,:) = yin.m_wf{2};
  yout(8,:) = yin.m_wg{1};
  yout(9,:) = yin.m_wg{2};
  yout(10,:) = yin.m_out.net;
  yout(11,:) = yin.m_out.exh;
  yout(12,:) = yin.T_a{1};
  yout(13,:) = yin.T_a{2};
  yout(14,:) = yin.T_pe{1};
  yout(15,:) = yin.T_pe{2};
  yout(16,:) = yin.T_te{1};
  yout(17,:) = yin.T_te{2};
  yout(18,:) = yin.wd_p;
  yout(19,:) = yin.wd_g;
  yout(20,:) = yin.k_e{1};
  yout(21,:) = yin.k_e{2};
  yout(22,:) = yin.k_e_dot{1};
  yout(23,:) = yin.k_e_dot{2};
  yout(24:end,:) = yin.opthets;
else
  yout.theta = yin(1,:);
  yout.h = yin(2,:);
  yout.hdot = yin(3,:);
  yout.m_a{1} = yin(4,:);
  yout.m_a{2} = yin(5,:);
  yout.m_wf{1} = yin(6,:);
  yout.m_wf{2} = yin(7,:);
  yout.m_wg{1} = yin(8,:);
  yout.m_wg{2} = yin(9,:);
  yout.m_out.net = yin(10,:);
  yout.m_out.exh = yin(11,:);
  yout.T_a{1} = yin(12,:);
  yout.T_a{2} = yin(13,:);
  yout.T_pe{1} = yin(14,:);
  yout.T_pe{2} = yin(15,:);
  yout.T_te{1} = yin(16,:);
  yout.T_te{2} = yin(17,:);
  yout.wd_p = yin(18,:);
  yout.wd_g = yin(19,:);
  yout.k_e{1} = yin(20,:);
  yout.k_e{2} = yin(21,:);
  yout.k_e_dot{1} = yin(22,:);
  yout.k_e_dot{2} = yin(23,:);
  yout.opthets = yin(24:end,:);
  yout.vecsize = size(yin,1);
end
```

```
function sens = SW_sensors( GP, t, y, sys, perfect )
% sens = SW_sensors( GP, t, y, sys, perfect )
%  Accepts an input structure y with all the usual fields (as defined in SW_pack), and
%  outputs a structure sens which contains a set of simulated sensor data.
%
%  For instance, y.m is not passed through to sens.y.m, since it is not possible
%  to measure airmass with a sensor; instead, SW_sensors calls SW_pressure and uses the
%  pressure values (along with other components of y) to calculate an approximation to
%  the airmass, stored in sens.y.m. Some components are rounded to some finite
%  resolution or modified by the addition of noise.
%
%  If perfect = true, t, y and p are just passed-through to sens without the addition
%  of noise or other alterations. This is for use inside SW_control_fast only. If
%  omitted, perfect is assumed to be false.

%% Get pressure and pass everything through by default
p = SW_pressure(GP,y); % This is the real pressure
sens.p = p;

sens.y = y;

sens.t = t;

if nargin == 4 || ~perfect
  %% Wipe fields we aren't allowed to know
  % We need to estimate these later
  for c1 = 1:2
    sens.y.m_a{c1} = NaN;
    sens.y.m_wf{c1} = NaN;
    sens.y.m_wg{c1} = NaN;
  end

  % These don't need to be estimated
  sens.y.m_out.net = 0;
  sens.y.m_out.exh = 0;
  sens.y.wd = 0;

  % Overwrite wall temperature functions with initial conditions
  sens.y.opthets = zeros(GP.T.npolys,1); % Coefficients of orthpolys
  sens.y.opthets(1) =  1.469273985e4; % Projected steady state
  sens.y.opthets(3) = 375.114;
  sens.y.opthets(5) = 214.0543;
  sens.y.opthets(7) = 59.27124;
  sens.y.opthets(9) = -25.13436;
  sens.y.opthets(11) = -46.76907;
  for c1 = 1:2
    % Temperatures of ends of tube, T_te
    sens.y.T_te{c1} = (GP.T_kick+GP.T_atm)/2+3;
    % Temperatures of ends of piston, T_pe
    sens.y.T_pe{c1} = GP.T_atm + 6.2;
  end

  %% Add resolution & noise details


  %% Calculate the air and water masses
  % based on the sens.y and p_sens APPROXIMATIONS
  % (formula from SW_pressure, rearranged)
  h_a = GP.h_a_fun(sens.y.h);
  for c1 = 1:2 % Set freefall conditions for both ends by default
    % Assume no steam and no liquid water
    sens.y.m_wg{c1} = 0;
    sens.y.m_wf{c1} = 0;
    sens.y.m_a{c1} = (sens.p{c1} * GP.A*h_a{c1})...
      ./ (sens.y.T_a{c1} * GP.R_univ/GP.Mol.air);
  end
  if sys.mode == 1 % If compressing ...
    sens.y.m_a = sys.comp_m_a; % use logged airmass instead
    % Get mass of steam from pressure (assume no volume change due to liquid water)
```

```
    sens.y.m_wg{sys.dir} = GP.Mol.wat*(sens.p{sys.dir}*GP.A*h_a{sys.dir}...
      ./(sens.y.T_a{sys.dir}*GP.R_univ) - sys.comp_m_a{sys.dir}/GP.Mol.air);
    % Get mass of liquid water from ratio
    sens.y.m_wf{sys.dir} = GP.water_add * sys.comp_m_a{sys.dir} - sens.y.m_wg{sys.dir};
  elseif sys.mode == 2 % If exhausting ...
    % Get air mass by assuming ratio is constant (assume no liquid water)
    sens.y.m_a{sys.dir} = (sens.p{sys.dir}*GP.A*h_a{sys.dir}) ./ ...
      (sens.y.T_a{sys.dir}*GP.R_univ * (GP.water_add/GP.Mol.wat + 1/GP.Mol.air) );
    % Get steam mass from ratio
    sens.y.m_wg{sys.dir} = GP.water_add * sens.y.m_a{sys.dir};
  end
end

end
```

```
   function ydivs = SW_sigmoid( ndiv, rat, lims, withplot )
% --- sigmoid ---
% ydivs = sigmoid( ndiv, rat, lims, withplot )
% This subdivides the interval [lims] into [ndiv] sub-intervals
% with the central ones being (roughly) <rat> times as big as the first.
% <ydivs> contains exactly ndiv entries (woo).
%   withplot = true creates a figure showing the result and tictoc.
% Version 1. Based on bender.m by SDG, modified 22.07.2011 by SW
% ------------------------------------------------------------------

% Start by thinking about the (much easier) interval [0 1]
% --- Polynomial is a*x^2 + b*x. ---
% dydx( 0) =    1-a/2
% dydx(+1) =  a+1-a/2 = 1+a/2
% The ratio of the latter / the former = <rat>
% Hence 1+a/2 = rat * (1-a/2)
% a = 2*(rat-1)/(1+rat)
a = 2*(rat-1)/(1+rat);
b = 1-a/2;

xdivs = linspace(0,1,ndiv)'; % Generate a linear input

fliplength = ceil(ndiv/2); % Work on the first half including midway
ydivs = a*xdivs(1:fliplength).^2 + b*xdivs(1:fliplength);

% Flip ydivs and glue it together again
ydivs = [ydivs;(1-flipud(ydivs(1:(end - mod(ndiv,2)))))];
% If ndiv is odd, then we need to ignore the central point;
% mod(ndiv,2) = 1 if ndivs odd, 0 if even

% Scale it to fit lims
xdivs = xdivs*diff(lims) + lims(1);
ydivs = ydivs*diff(lims) + lims(1);

if withplot
    figure('Name','Sigmoid.m')
    subplot(1,2,1), plot(xdivs,ydivs,'k');
    xlabel('Linear input'), ylabel('Nonlinear output');
    title('Nonlinear coordinate map'); grid on; xlim(lims); ylim(lims);
    subplot(1,2,2), plot(ydivs(2:end),diff(ydivs),'k');
    xlabel('Output values'), ylabel('Differences');
    title('Gap widths'); grid on; xlim(lims);
end

end
```

```
function [ov,maxerror] = SW_getorthovectors( nvectors, lims, withplot, weight )
%ov = SW_orthovectors( npolys, lims, withplot )
% This is a function to generate orthogonal vectors. Inputs are:
%    npolys     How many vectors to generate
%    lims       The range over which the vectors need to be orthonormal.
%                   Needs to be set up as [lower, upper] !
%    withplot   Boolean - if true, func will plot graphs of orthopolys and errors.
%    weight     Boolean - if true, func will use weighting functions
%
% Outputs a structure ov containing six parts:
%    xvals      A vector of the x co-ords
%    xdiffs     A vector of the x diffs *at* each point (avg of upper and lower).
%    Ap         A matrix of the orthonormal vectors.
%    Apd        As Ap, but the derivative values.
%    Apdd       As Ap, but the second derivative values.
%                 ( Ap, Apd and Apdd can be used to evaluate the sum of the vectors by
%                   multiplying with a vector of coefficients theta. )
%    psdinvAp   The pseudoinverse of Ap (since Ap is not normally square).
% Also outputs the maximum error magnitude.
%
% 2012-02-02
dotimestamp, fprintf('Running %s ...\n',mfilename);
npoints = 1000;
if ~exist('weight','var')
  weight=true;
end

%% Define x and the weighting function
if weight
  gapratio = 60;
  xvals = SW_sigmoid(npoints,gapratio,lims,withplot);
  xbasis = SW_sigmoid(npoints,gapratio,[-1,+1],false);

  % Establish weighting vector
  order = 4; wtpolylims = [1,5];
  weight = xbasis.^order; % Make curve
  weight = diff(wtpolylims)*weight/max(weight) + wtpolylims(1);% Scale curve

  % Plot weight function
  if withplot
    figure('Name','orthovector_weights');
    plot(xvals,weight,'k'); xlabel('z'); ylabel('Weighting function');
  end
else
  xvals = linspace(lims(1),lims(2),npoints)';
  xbasis = linspace(-1,1,npoints)';
  weight = 1;
end

%% Generate Vandermonde matrix
V = zeros(npoints,nvectors);
for c1 = 1:nvectors
  V(:,c1) = xbasis.^(c1 - 1);
end

%% Find the largest vector and move it to the front
norms = zeros(nvectors,1);
for c1 = 1:nvectors
  norms(c1) = norm(V(:,c1));
end
[~, index] = sort(norms,'descend');
V = V(:,index);

%% Perform modified Gram-Schmidt
% Define inner product function
innerprod = @(a,b) sum(weight .* a .* b);
A = V;
for c1 = 1:nvectors
  for c2 = 1:c1-1 % for each of the vectors already orthogonalised
```

```
    % Subtract projection
    A(:,c1) = A(:,c1) - innerprod(A(:,c2),A(:,c1))*A(:,c2);
  end
  % Normalise
  A(:,c1) = A(:,c1) ./ sqrt( innerprod(A(:,c1),A(:,c1)) );
end

%% Calculate properties
% Create anon function to get nice differences
equidiff = @(vec) 0.5*([vec(2)-vec(1); diff(vec)] + [diff(vec); vec(end)-vec(end-1)]);

ov.xvals = xvals;
ov.xdiffs = equidiff(xvals);
ov.Ap = A;
ov.psdinvAp = (ov.Ap.' * ov.Ap) \ ov.Ap.';

dotimestamp, fprintf(' - Calculating derivatives with polyfit() & polyder() ...\n');
scaling = 2/diff(lims); % Scaling factor since using xbasis for diffs
for c1 = 2:nvectors
  dpoly = polyder(polyfit(xbasis,A(:,c1),c1-1));
  Apd(:,c1) = polyval(dpoly,xbasis) .* scaling;
  Apdd(:,c1) = polyval(polyder(dpoly),xbasis) .* scaling^2;
end

ov.Apd = Apd;
ov.Apdd = Apdd;

%% Calculate errors
if withplot || nargout > 1
  accuracy = zeros(nvectors);
  for c1 = 1:nvectors
    for c2 = 1:c1
      accuracy(c1,c2) = innerprod(A(:,c1),A(:,c2));
      accuracy(c2,c1) = accuracy(c1,c2); % Symmetry
    end
  end
  % Fix so that 0 everywhere means correct
  accuracy = accuracy - eye(nvectors);
  % Calculate errors
  errors = sqrt(real(accuracy).^2+imag(accuracy).^2);
  maxerror = max(errors(:));
end
%% Plot orthovectors
if withplot
  dotimestamp, fprintf(' - Plotting orthovectors ...\n');
  % Get colour data
  colours = get_colour_spec(nvectors);

  figure('Name','Orthovectors')
  clf; hold on;
  %   subplot(3,1,1:2);
  hold on;
  % Plot the lines with colours worked out above
  for c1 = 1:nvectors
    h = plot(xvals,A(:,c1),...
      'Color',colours(c1,:),...
      'DisplayName',['$P_{',num2str(c1),'}$']);
  end
  title(['Orthonormal basis of polynomials',10,...
    '(normalised, no coefficients)']);
  xlabel('z');
  grid on; legend('toggle');
  xlim(lims); h = legend('Location','NorthEastOutside');
  set(h,'Interpreter','LaTeX');

  % Plot the weighting function
  %   subplot(3,1,3), plot(xvals,weight,'k-');
  %   title('Weighting function used for GSONP'); xlim(lims); grid on;
```

```
  %% Plot accuracy
  figure('Name','Orthovector errors');
  set(gcf,'WindowStyle','docked'); clf; hold on;
  mesh(errors,'FaceColor','none');
  grid on; shading interp;
  title(['Inner product errors',10, 'max = ',...
    num2str(max(real(accuracy(:)))),', min = ',...
    num2str(min(real(accuracy(:))))]);
  zlabel('Complex magnitude of error'); colorbar; view(-17,35);
  xlabel('First input poly'), ylabel('Second input poly');
end
dotimestamp, fprintf(' - %s complete.\n',mfilename);
```

```
% funcs = SW_thermal_funcs(PlotGraphs)
% This function returns a structure of function handles. All functions use SI units only.
%
% funcs.c_pa(T) and funcs.c_va(T) return the heat capacity at constant pressure or volume of
%   dry air, by making a call to lemmon.m (originally by SDG).
%
% funcs.k_air(T) can be used to quickly evaluate the thermal conductivity of air within the
%   temperature range 175 - 1900 K, based on data from Rogers & Mayhew steam tables. Outside
%   this range, inaccurate values based on the fitted curve will be returned.
%
% funcs.T_sat(p) and funcs.p_sat(T) return the saturation temperature of water, aka boiling
%   point, and saturation pressure of water respectively. funcs.T_sat is accurate from
%   0.1 - 221.2 BAR, and funcs.p_sat is accurate from 318.95 - 647.3 K. Outside these ranges,
%   inaccurate values based on the fitted curves will be returned. The critical point of
%   water is at (221.1 BAR, 647.3 K).
%
% funcs.dT_sat(p) returns the gradient of the saturation curve (d T_sat / d p_sat) at a given
%   pressure. This uses the derivative of the previous polynomial.
%
% funcs.L_water(p) will return the latent heat of water at pressures in the range
%   0.02 - 30 BAR. Outside this range, inaccurate values based on the fitted curve will be
%   returned.
%
% funcs.c_pwf(T), funcs.c_pwg(T), funcs.c_vwg(T), funcs.c_vwg(T) will return the heat capacity
%   at constant pressure (_p) or constant volume (_v) for liquid water (_f) and steam (_g).
%   These datasets are valid from 273.16 - 573.15 K only. Outside this range, inaccurate
%   values based on the fitted curve will be returned. c_pwf = c_vwf since water is
%   incompressible (included as separate functions for convenience).
%
% If the argument PlotGraphs is included and is true, a figure showing the two functions will
% also be plotted.
%
% Added to model by SW on 2012-03-05, modified to avoid interp on 2013-01-31, modified to add
% T_sat, L_water, c_pwf and c_pwg on 2013-10-07.
function funcs = SW_thermal_funcs(PlotGraphs)
% Some constants
R_univ = 8.3144621;     % Universal gas constant \bar{R} (J/ mol K)
Mol.air = 28.9645/1e3;  % Molar mass of air (kg/mole)
Mol.wat = 18.0155/1e3;  % Molar mass of water (kg/mole)

%% PlotGraphs setup
if ~exist('PlotGraphs','var')
  PlotGraphs = false;
elseif PlotGraphs
  subtitle = [10,'from ',mfilename];
  subtitle = strrep(subtitle,'_','\_'); % Make underscores display properly
  numpoints = 2e3;
  padding = 0.1;
end

%% c_p,a and c_v,a (heat capacities of air)
% code moved to internal_cp_func because cp cannot be returned in one operation, due to the
% way lemmon.m works.
funcs.c_pa = @(T) internal_cp_func(T,R_univ/Mol.air);
funcs.c_va = @(T) funcs.c_pa(T) - R_univ/Mol.air;

if PlotGraphs
  figure('Name','c_p,a and c_v,a');  hold on;
  c_a.T = linspace(50,1500,numpoints);
  c_a.cp = funcs.c_pa(c_a.T);
  c_a.cv = funcs.c_va(c_a.T);
  plot(c_a.T,c_a.cp,'r','DisplayName','c_{pa}');
  plot(c_a.T,c_a.cv,'b','DisplayName','c_{va}');
  grid on; title(['Specific heat capacities of air, from lemmon.m by SDG',subtitle]);
  xlabel('Temperature (K)'), ylabel('Specific heat capacity (J/kg K)');
  legend('show','Location','NorthWest');
end

%% k (thermal conductivity of air)
```

```
% Data from Rogers & Mayhew steam tables, p16.
k.T = [ 175:25:400, 450:50:1400, 1500:100:1900 ]'; % In Kelvin
k.k = [1.593,1.809,2.020,2.227,2.428,...
  2.624,2.816,3.003,3.186,3.365,...
  3.710,4.041,4.357,4.661,4.954,...
  5.236,5.509,5.774,6.030,6.276,...
  6.520,6.754,6.985,7.209,7.427,...
  7.640,7.849,8.054,8.253,8.450,...
  8.831,9.199,9.554,9.899,10.233]' *1e-5... % In kW/mK
  *1e3; % In W/mK

% Fit polynomial (order 3) to data
warning off MATLAB:polyfit:RepeatedPointsOrRescale % Tested, fit is still fine at order 3.
k.poly = polyfit(k.T,k.k,3);
warning on MATLAB:polyfit:RepeatedPointsOrRescale % Reset.
% Create anonymous function
funcs.k_air = @(T) polyval(k.poly, real(T));

if PlotGraphs
  figure('Name','k_air');
  subplot(3,1,1:2), hold on;
  k.pg.T = padded_linspace(k.T(1),k.T(end),padding,numpoints);
  k.pg.k = funcs.k_air(k.pg.T);
  plot(k.T,k.k,'k+', k.pg.T,k.pg.k,'b-');
  title('Thermal conductivity of air');
  grid on; ylabel('Thermal conductivity, k (W/m K)');
  axesgroup = gca;
  subplot(3,1,3), hold on;
  k.pg.interp = interp1q(k.T,k.k,k.pg.T);
  plot(k.pg.T,abs(k.pg.k-k.pg.interp)./k.pg.interp,'b-');
  xlabel('Pressure (Pa)'), ylabel('Error');
  title('Error between fitted polynomial and linear interpolation');
  set(gca,'YScale','log','YMinorGrid','off');
  xlabel('Temperature (K)')
  ylim([1e-5,1e-1]);
  set(gca,'YTick',[1e-5,1e-4,1e-3,1e-2,1e-1]);
  set(gca,'YTickLabel',{'0.001%','0.01%','0.1%','1%','10%'});
  linkprop([axesgroup,gca],{'XLim','XScale','XMinorGrid'});
end

%% T_sat and p_sat (saturation temperature and pressures of water, aka boiling point)
% Saturation pressure (in BAR) (Rogers & Mayhew, p3-5)
T_s.p = [0.1:0.02:0.5, 0.55:0.05:1, 1.1:0.1:3, 3.5:0.5:6, 7:1:20, 22:2:50, 55:5:200,...
  202:2:220, 221.2]' * 1e5; % Convert from BAR to Pa
% Saturation temperature (in K)  (Rogers & Mayhew, p3-5)
T_s.T = [45.8, 49.4,52.6,55.3,57.8,60.1, ...
  62.2,64.1,65.9,67.5,69.1, ...
  70.6,72.0,73.4,74.7,75.9, ...
  77.1,78.2,79.3,80.3,81.3, ...
  83.7,86.0,88.0,90.0,91.8, ...
  93.5,95.2,96.7,98.2,99.6, ...
  102.3,104.8,107.1,109.3,111.4, ...
  113.3,115.2,116.9,118.6,120.2, ...
  121.8,123.3,124.7,126.1,127.4, ...
  128.7,130.0,131.2,132.4,133.5, ...
  138.9,143.6,147.9,151.8,155.5, ...
  158.8,165.0,170.4,175.4,179.9, ...
  184.1,188.0,191.6,195.0,198.3, ...
  201.4,204.3,207.1,209.8,212.4, ...
  217.2,221.8,226.0,230.0,233.8, ...
  237.4,240.9,244.2,247.3,250.3, ...
  253.2,256.0,258.8,261.4,263.9, ...
  269.9,275.6,280.8,285.8,290.5, ...
  295.0,299.2,303.3,307.2,311.0, ...
  314.6,318.0,321.4,324.6,327.8, ...
  330.8,333.8,336.6,339.4,342.1, ...
  344.8,347.3,349.8,352.3,354.6, ...
  357.0,359.2,361.4,363.6,365.7, ...
  366.5,367.4,368.2,369.0,369.8, ...
```

```
  370.6,371.4,372.1,372.9,373.7, 374.15]' +273.15; % Convert from C to K

% Create anonymous functions
T_s.Tpoly = polyfit( log(T_s.p), T_s.T, 3 );
funcs.T_sat = @(p) polyval(T_s.Tpoly, log(p) );
% === Speed test ===
% funcs.T_sat_interp1 = @(p) interp1(T_s.p,T_s.T,p);
% funcs.T_sat_interp1q = @(p) interp1q(T_s.p,T_s.T,p);
% count = 1e5;
% rand_p = rand(count,1)*(log10(221.2)-log10(0.1))+log10(0.1);
% fprintf('\nPoly     : '); tic;
% for c1 = 1:count
%   Tsat = funcs.T_sat(rand_p(c1));
% end
% time(1) = toc; fprintf('%f sec\n',time(1));
% fprintf('Interp1  : '); tic;
% for c1 = 1:count
%   Tsat = funcs.T_sat_interp1(rand_p(c1));
% end
% time(2) = toc; fprintf('%f sec (poly %.1f%% faster)\n',time(2),100*(time(2)-time(1))/time(2));
% fprintf('Interp1q : '); tic;
% for c1 = 1:count
%   Tsat = funcs.T_sat_interp1q(rand_p(c1));
% end
% time(3) = toc; fprintf('%f sec (poly %.1f%% faster)\n',time(3),100*(time(3)-time(1))/time(3));

% Find midpoint values of gradient
T_s.dT.dT = diff(T_s.T)./diff(T_s.p);
T_s.dT.p = T_s.p(1:end-1) + 0.5*diff(T_s.p);
% Polyfit to gradient
T_s.dTpoly = polyfit(log(T_s.dT.p),log(T_s.dT.dT),2);
funcs.dT_sat = @(p) exp(polyval( T_s.dTpoly,log(p) ));

if PlotGraphs
  figure('Name','T_sat');
  subplot(3,1,1:2), hold on;
  T_s.pg.p = exp(padded_linspace(log(T_s.p(1)),log(T_s.p(end)),...
    padding, numpoints));
  T_s.pg.T = funcs.T_sat(T_s.pg.p);
  plot(T_s.p, T_s.T, 'k+', 'DisplayName','Rogers & Mayhew data');
  plot(T_s.p(end),T_s.T(end),'ro','DisplayName','Critical point');
  plot(T_s.pg.p, T_s.pg.T, 'b-','DisplayName','3rd-order polynomial');
  grid on; legend('show','Location','NorthWest');
  title('Saturation temperature of water');
  ylabel('Saturation temperature, $T_\mathrm{sat}$ (K)','Interpreter','latex');
  set(gca,'XScale','log');
  axesgroup = gca;
  subplot(3,1,3), hold on;
  T_s.pg.interp = interp1q(T_s.p,T_s.T,T_s.pg.p);
  plot(T_s.pg.p,abs(T_s.pg.T-T_s.pg.interp)./T_s.pg.interp,'b-');
  xlabel('Pressure (Pa)'), ylabel('Error');
  title('Error between fitted polynomial and linear interpolation');
  set(gca,'YScale','log','YMinorGrid','off'); xlabel('Pressure (Pa)');
  ylim([1e-5,1e-2]);
  set(gca,'YTick',[1e-5,1e-4,1e-3,1e-2]);
  set(gca,'YTickLabel',{'0.001%','0.01%','0.1%','1%'});
  linkprop([axesgroup,gca],{'XLim','XScale','XMinorGrid'});

  figure('Name','dT_sat');
  subplot(3,1,1:2),
  hold on;
  % Construct zero-order hold line
  T_s.pg.dTi.p = T_s.p([1,sort([2:length(T_s.p)-1,2:length(T_s.p)-1]),length(T_s.p)]);
  T_s.pg.dTi.dT = T_s.dT.dT(sort([1:length(T_s.dT.dT),1:length(T_s.dT.dT)]));
  % Interpolated & polydur datasets
  T_s.pg.dTinterp = interp1q(T_s.pg.dTi.p,T_s.pg.dTi.dT,T_s.pg.p);
  T_s.pg.dTpolyder = polyval( polyder(T_s.Tpoly),log(T_s.pg.p) )./ T_s.pg.p;
  plot(T_s.pg.dTi.p,T_s.pg.dTi.dT,'k-','DisplayName','Gradient of linear interpolation');
%   plot(T_s.dT.p,T_s.dT.dT,'m+','DisplayName','Midpoint values');
```

```
%   plot(T_s.pg.p, T_s.pg.dTpolyder,'r-','DisplayName','Polyder solution ');
  plot(T_s.pg.p, funcs.dT_sat(T_s.pg.p), ...
    'b-','DisplayName','2nd-order polynomial fit to midpoints');
  grid on; title('Derivative $\frac{\mathrm{d}T_\mathrm{sat}}{\mathrm{d}p}$ of saturation curve',...
    'Interpreter','latex');
  ylabel('Derivative (K/Pa)');
  set(gca,'XScale','log','XMinorGrid','off','YScale','log','YMinorGrid','off');
  legend('show','Location','NorthEast'); xlim([8e3,3e7]);
  axesgroup = gca;
  subplot(3,1,3), hold on;
%   plot(T_s.pg.p, abs(T_s.pg.dTinterp - T_s.pg.dTpolyder)./T_s.pg.dTinterp, ...
%     'r-', 'DisplayName','Polynomial derivative');
  plot(T_s.pg.p, abs(T_s.pg.dTinterp - funcs.dT_sat(T_s.pg.p))./T_s.pg.dTinterp,...
    'b-', 'DisplayName','Polynomial fitted to midpoints');
  xlabel('Pressure (Pa)'), ylabel('Error');
  title('Error between fitted polynomial and linear interpolation derivatives');
  set(gca,'YScale','log','YMinorGrid','off','XScale','log','XMinorGrid','off');
  xlabel('Pressure (Pa)');
  ylim([1e-4,1]);
  set(gca,'YTick',10.^(-4:1));
  set(gca,'YTickLabel',{'0.01%','0.1%','1%','10%','100%'});
  linkprop([axesgroup,gca],{'XLim','XScale'});
end

%% L_water (latent heat of vaporisation of water)
% Pressure values (Rogers & Mayhew, p3-5)
L.p = T_s.p;
% Latent heat values (in J/kg) (Rogers & Mayhew, p3-5)
L.L = [2392, 2383,2376,2369,2363,2358, 2353,2348,2343,2339,2336, ...
  2332,2328,2325,2322,2318, 2315,2313,2310,2308,2305, 2298,2293,2288,2283,2278, ...
  2273,2269,2266,2262,2258, 2251,2244,2238,2232,2226, 2221,2216,2211,2206,2202, ...
  2198,2193,2189,2185,2182, 2178,2174,2171,2168,2164, 2148,2134,2121,2109,2097, ...
  2087,2067,2048,2031,2015, 2000,1986,1972,1960,1947, 1935,1923,1912,1901,1890, ...
  1870,1850,1831,1812,1795, 1778,1761,1744,1729,1714, 1698,1683,1668,1654,1639, ...
  1605,1570,1538,1505,1473, 1441,1410,1379,1348,1317, 1286,1255,1224,1194,1163, ...
  1131,1099,1067,1034,1001, 967,932,895,858,819, 778,735,689,639,584, 560,535,508,479,447,...
  412,373,328,270,170, 0]' * 1e3; % Convert from kJ/kg to J/kg

% Create anonymous function.
L.poly.p = logspace(log10(L.p(1)),log10(L.p(end)),5e2)';
warning off MATLAB:polyfit:RepeatedPointsOrRescale
L.poly.poly = polyfit( log10(L.poly.p), interp1q(L.p,L.L,L.poly.p), 9 );
warning on MATLAB:polyfit:RepeatedPointsOrRescale % Reset.

funcs.L_water = @(p) polyval(L.poly.poly, log10(p) );

if PlotGraphs
  figure('Name','L_water'); hold on;
  L.pg.p = logspace(log10(L.p(1)), log10(L.p(end)), numpoints)';
  subplot(3,1,1:2), hold on;
  plot(L.p,L.L,'k+','DisplayName','Rogers & Mayhew data');
  L.pg.L_fun = polyval(L.poly.poly,log10(L.pg.p));
  plot(L.pg.p,L.pg.L_fun,'b-',...
    'DisplayName','9th-order polynomial');
  ylimits = get(gca,'YLim'); xlimits = get(gca,'XLim');
  title(['Latent heat of vaporisation of water',subtitle]);
  grid on; ylabel('Latent heat of vaporisation (J/kg)');
  set(gca,'XScale','log'); set(gca,'YLim',ylimits,'XLim',xlimits);
  subplot(3,1,1:2),  legend('show','Location','SouthWest');
  axesgroup = gca;

  subplot(3,1,3), hold on;
  L.pg.interp = interp1q(L.p,L.L,L.pg.p);
  plot(L.pg.p,abs(L.pg.L_fun-L.pg.interp)./L.pg.interp,'b-');
  xlabel('Pressure (Pa)'), ylabel('Error');
  title('Error between fitted polynomial and linear interpolation');
  set(gca,'YScale','log');
  set(gca,'YTick',[1e-6,1e-4,1e-2,1]);
  set(gca,'YTickLabel',{'0.0001%','0.01%','1%','100%'});
```

```
  ylim([1e-6,1]);
  linkprop([axesgroup,gca],{'XLim','XScale'});
end

%% c_p,wf and c_p,wg (heat capacity at constant pressure for water, both liquid & gas)
% c_p values, temperature axis  (Rogers & Mayhew, p10)
c_pw.Tdata = [0.01, 5:5:150, 160:10:300]' +273.15; % Convert from C to K
% c_pwf (heat capacity at constant pressure, for liquid water) (Rogers & Mayhew, p10)
c_pw.f.data = [4.21,4.204,4.193,4.186,4.183, ...
  4.181,4.179,4.178,4.179,4.181, ...
  4.182,4.183,4.185,4.188,4.191, ...
  4.194,4.198,4.203,4.208,4.213, ...
  4.219,4.226,4.233,4.240,4.248, ...
  4.26,4.27,4.28,4.29,4.30, ...
  4.32,4.35,4.38,4.42,4.46, ...
  4.51,4.56,4.63,4.70,4.78, ...
  4.87,4.98,5.10,5.24,5.42, 5.65]' * 1e3; % Convert from kJ/ kg K to J/kg K
% c_pwg (heat capacity at constant pressure, for gaseous water) (Rogers & Mayhew, p10)
c_pw.g.data = [1.86,1.86,1.86,1.87,1.87, ...
  1.88,1.88,1.88,1.89,1.89, ...
  1.90,1.90,1.91,1.92,1.93,...
  1.94,1.95,1.96,1.97,1.99, ...
  2.01,2.03,2.05,2.07,2.09, ...
  2.12,2.15,2.18,2.21,2.25, ...
  2.29,2.38,2.49,2.62,2.76, ...
  2.91,3.07,3.25,3.45,3.68, ...
  3.94,4.22,4.55,4.98,5.46, 6.18]' * 1e3; % Convert from kJ/ kg K to J/kg K

% Define anonymous functions
warning off MATLAB:polyfit:RepeatedPointsOrRescale % Tested, fit is still fine at order 4.
c_pw.f.poly = polyfit(c_pw.Tdata,c_pw.f.data,4);
funcs.c_pwf = @(T) polyval(c_pw.f.poly,T);
c_pw.g.poly = polyfit(c_pw.Tdata,c_pw.g.data,4);
funcs.c_pwg = @(T) polyval(c_pw.g.poly,T);
warning on MATLAB:polyfit:RepeatedPointsOrRescale % Reset.

funcs.c_vwf = funcs.c_pwf; % Liquid water is incompressible!
funcs.c_vwg = @(T) funcs.c_pwg(T) - R_univ/Mol.wat;

if PlotGraphs
  figure('Name','c_pwf and c_pwg');
  subplot(3,1,1:2), hold on;
  c_pw.pg.T = padded_linspace(c_pw.Tdata(1),c_pw.Tdata(end),padding,numpoints);
  c_pw.pg.f = funcs.c_pwf(c_pw.pg.T);
  c_pw.pg.g = funcs.c_pwg(c_pw.pg.T);
  plot(c_pw.Tdata,c_pw.f.data,'b+','DisplayName','Rogers & Mayhew liquid water data');
  plot(c_pw.pg.T,c_pw.pg.f,'b-','DisplayName','4th-order polynomial');
  plot(c_pw.Tdata,c_pw.g.data,'r+','DisplayName','Rogers & Mayhew steam data');
  plot(c_pw.pg.T,c_pw.pg.g,'r-','DisplayName','4th-order polynomial');
  ylabel('Heat capacity, $c_p$ (J/kg K)','interpreter','latex');
  legend('show','Location','NorthWest');
  title('Heat capacities of water at constant pressure, $c_{p,wf}$ and $c_{p,wg}$',...
    'Interpreter','latex');
  axesgroup = gca;
  subplot(3,1,3), hold on;
  c_pw.pg.finterp = interp1q(c_pw.Tdata,c_pw.f.data,c_pw.pg.T);
  c_pw.pg.ginterp = interp1q(c_pw.Tdata,c_pw.g.data,c_pw.pg.T);
  plot(c_pw.pg.T,abs(c_pw.pg.f-c_pw.pg.finterp)./c_pw.pg.finterp,'b-');
  plot(c_pw.pg.T,abs(c_pw.pg.g-c_pw.pg.ginterp)./c_pw.pg.ginterp,'r-');
  xlabel('Pressure (Pa)'), ylabel('Error');
  title('Error between fitted polynomials and linear interpolation');
  set(gca,'YScale','log','YMinorGrid','off');
  xlabel('Temperature (K)')
  ylim([1e-5,1e-1]);
  set(gca,'YTick',[1e-5,1e-4,1e-3,1e-2,1e-1]);
  set(gca,'YTickLabel',{'0.001%','0.01%','0.1%','1%','10%'});
  linkprop([axesgroup,gca],{'XLim','XScale'});
end
```

```
end

%% Function to use T to get cp from lemmon scripts
function cp_out = internal_cp_func(T, Rspec)
T = real(T);
rho = 70e5./(Rspec./T); % 70e5 is arbitrary pressure value (cp is isobaric)
[~,~,~,cp_out] = lemmon(rho,T); % Ignore first 3 outputs
end

%% Function which duplicates linspace, with padding, for the plots
function vector = padded_linspace(start, finish, padding, numpoints)
range = finish - start;
vector = linspace(start-padding*range, finish+padding*range, numpoints)';
end
```

```
% This is a function to calculate the energy required to compress air with a given fraction of
%   liquid water from air pressure up to 70BAR and then exhaust it. It includes the energy used
%   to change phase. It is based on water_energy.m and was built on 2013-09-19 by SW,
function E_req_func = SW_water_energy(GP,PlotGraphs)
dotimestamp, fprintf('Running %s ...\n',mfilename);
% The one constant that isn't in GP that we need:
GP.hdot = -1; % Speed of piston

% Initialise
y_init.h = 100; % Any value will do, it gets divided through at the end
y_init.hdot = GP.hdot;
y_init.T_a = GP.T_atm;
y_init.m_a = y_init.h*GP.A*GP.p_atm/(y_init.T_a*GP.R_univ/GP.Mol.air);
y_init.m_wf = GP.water_add * y_init.m_a;
y_init.m_wg = 0;
y_init.WD = 0; % Work done (integral of instantaneous power)

phase = 1;
y0 = pack(y_init); t0 = 0;
out.t = 0; out.yv = pack(y_init);
out.te = []; out.ie = []; out.ye = [];

% Run ODE in loop
finished = false;
while ~finished
  clear sol;
  sol = ode15s(@(t,y)ode(GP,y,phase),t0+[0,100],y0, ...
    odeset('Events',@(t,y)events(GP,y,phase),'Refine',1)  );

  t0 = sol.x(end);
  y0 = sol.y(:,end);

  out.t = [out.t, sol.x(2:end)]; % Concatenate
  out.yv = [out.yv, sol.y(:,2:end)];

  out.te = [out.te,sol.xe];
  out.ie = [out.ie,sol.ie];
  out.ye = [out.ye,sol.ye];

  if sol.ie == 1 % If tripped this phase's check
    phase = phase + 1;
  else % Otherwise we're done.
    finished = true;
  end
end
out.ys = pack(out.yv);

h_end = out.ys.h(end);
% Total energy required (sim + exhaust - atmo + friction)
E_req = out.ys.WD(end) + h_end*GP.p_targ*GP.A - y_init.h*GP.A*GP.p_atm + y_init.h*GP.F_mu;
% Function form
E_req_func = @(h_a) h_a * E_req/y_init.h;

if PlotGraphs
  dotimestamp, fprintf(' - Plotting graph...\n');
  % Prepare the data
  out.yes = pack(out.ye);
  out.pp.p = pressure(GP,out.ys);
  figure('Name',mfilename); hold on;
  % Plot the three datasets
  plot(out.pp.p,GP.Tfun.T_sat(out.pp.p),'k-','DisplayName','Saturation curve');
  plot(out.pp.p,out.ys.T_a,'b-','DisplayName','Simulation');
  plot(interp1q(out.t',out.pp.p',out.te'),out.yes.T_a,'ko',...
    'DisplayName','ODE events');
  legend('show','Location','SouthEast');
  % Add chrome
  xlabel('Pressure (Pa)'); set(gca,'XScale','log');
  ylabel('Temperature (K)'); grid on; box on;
  plottitle = sprintf('From %s\nGP.water_add = %g, energy required = %g MJ / m air',...
```

```
    mfilename,GP.water_add,E_req/1e6);
  title(strrep(plottitle,'_','\_'));
end
dotimestamp, fprintf(' - %s complete.\n',mfilename);
end

%% ODE function
function ydot = ode(GP, y, phase)
ydot = pack(0*y);
y = pack(y);

p = pressure(GP,y); % Pressure

ydot.h = GP.hdot;
ydot.hdot = 0;
% Get gamma using general formula
gamma = GP.gam(y.m_a, y.m_wg, y.m_wf, y.T_a);
% Get Tdot_adi, adiabatic rate
ydot.T_a =  y.T_a * ( 1-gamma ) * (ydot.m_a/y.m_a + y.hdot/y.h );

if phase == 2 % If in evaporating phase

    %% New method (allows variable ydot.m_wg and correct ydot.T_a_adi)
    % Components of complicated pdot, Tdot_sat and m_wg algebra tomfoolery.
    x_1 = (y.m_a/GP.Mol.air + y.m_wg/GP.Mol.wat)...
      * GP.R_univ / (GP.A*y.h);
    x_2 = ( (ydot.m_a/GP.Mol.air) * (GP.R_univ / GP.A) - x_1*ydot.h )...
      * y.T_a / y.h;
    x_3 = GP.R_univ*y.T_a / (GP.Mol.wat*GP.A*y.h);
    x_4 = GP.Tfun.dT_sat(p);
    x_5 = (GP.Tfun.c_pa(y.T_a).*y.m_a ...
      + GP.Tfun.c_pwg(y.T_a).*y.m_wg ...
      + GP.Tfun.c_pwf(y.T_a).*y.m_wf) ...
      / GP.Tfun.L_water(p);
    % Formula for m_wg
    ydot.m_wg = (ydot.T_a*x_5+ydot.m_wg-x_2*x_4*x_5/(1-x_1*x_4))...
      / (1 + x_3*x_4*x_5/(1-x_1*x_4));
    % Formula for Tdot_sat
    ydot.T_a = x_4*(x_2 + x_3*ydot.m_wg)/(1-x_1*x_4);
  % Symmetry
  ydot.m_wf = - ydot.m_wg;
end
% Work done
ydot.WD = - y.hdot * p * GP.A;

% Repack
ydot = pack(ydot);
end

%% ODE events function
function [value, isterminal, direction] = events(GP,y,phase)
y = pack(y); p = pressure(GP,y);
switch phase
  case 1
    value(1) = GP.Tfun.T_sat(p) - y.T_a;  % Stop if hit saturation temperature
  case 2
    value(1) = y.m_wf; % Stop when no water left
  case 3
    value(1) = 1; % Never stop (rely on value(2) to finish)
end
value(2) = p - GP.p_targ; % Stop if target pressure
isterminal(1:2) = true; % Always stop
direction(1:2) = false; % Never care about direction
end

%% pressure function
function p = pressure(GP,ys)
p = (ys.m_a/GP.Mol.air+ys.m_wg/GP.Mol.wat)*GP.R_univ .*...
  ys.T_a ./( GP.A*ys.h-ys.m_wf/GP.rho_water);
```

```
end

%% pack function
function yout = pack(yin)
if isstruct(yin)
  yout(1,:) = yin.h;
  yout(2,:) = yin.hdot;
  yout(3,:) = yin.m_a;
  yout(4,:) = yin.m_wf;
  yout(5,:) = yin.m_wg;
  yout(6,:) = yin.T_a;
  yout(7,:) = yin.WD;
else
  yout.h = yin(1,:);
  yout.hdot = yin(2,:);
  yout.m_a = yin(3,:);
  yout.m_wf = yin(4,:);
  yout.m_wg = yin(5,:);
  yout.T_a = yin(6,:);
  yout.WD = yin(7,:);
end
end
```

F.2 Ancilliary scripts

```
%thermal_mod_func
%   This function simulates thermal diffusion through a cylinder of air using a
%    point-derived numerical difference model.
%
% It has been extended on 2013-03-02 to do the same calculation for a spherical droplet of
% water.
clear all; close all; clc;

Tfuncs = SW_thermal_funcs(false);

% Properties of each sim
sim(1).name = 'Cylinder of air';
sim(1).constant = 1; % 1 for cylinder, 2 for sphere
sim(1).t.init = 600; % Initial temperature of all points
sim(1).t.contours = sim(1).t.init-[300:50:550,599]; % Temperature contours to plot
sim(1).t.end = 293; % Temperature of node at end
sim(1).radius = [0.2,0.6]; % Range of radii to model (t.end will be at upper limit)
sim(1).tlims = [0,10]; % Timespan for simulation
sim(1).rho = 29.72; % Density
sim(1).cv = Tfuncs.c_va(sim(1).t.init); % Volumetric heat capacity (J/kgK)
sim(1).k = Tfuncs.k_air(sim(1).t.init); % Thermal conductivity

sim(2).name = 'Droplet of water';
sim(2).constant = 2; % 1 for cylinder, 2 for sphere
sim(2).t.init = 293; % Initial temperature of all points
sim(2).t.end = sim(2).t.init+10; % Temperature of node at end
sim(2).t.contours = -9:-1; % Temperature contours to plot
sim(2).radius = [0,1e-4]; % Range of radii to model (t.end will be at upper limit)
sim(2).tlims = [0,1.5e-5]; % Timespan for simulation
sim(2).rho = 998.3; % Density
sim(2).cv = Tfuncs.c_vwf(sim(2).t.init); % Heat capacity (J/kgK)
sim(2).k = 0.596; % Thermal conductivity (W/m^2)

for c1 = 1:length(sim)

  P = sim(c1); % Parameters

  % Set up the arrays
  npoints = 300;
  r = linspace(P.radius(1),P.radius(2),npoints+1);
  r = r(2:end); % Eliminate first point

  T0 = P.t.init + 0*r; % Initial vector of temperatures
  T0(end) = P.t.end; % Set last point

  DT = zeros(npoints); % Square matrix
  const = P.k / (P.cv / P.rho); % Constant part
  for c2 = 2:(npoints-1)
    bit(1) = 1 / ((r(c2+1)-r(c2))*(r(c2)-r(c2-1)));
    bit(2) = P.constant / (r(c2)*(r(c2+1)-r(c2-1)));
    DT(c2,c2-1) = const*( bit(1) + bit(2) );
    DT(c2,c2) = const*( -2*bit(1) );
    DT(c2,c2+1) = const*( bit(1) - bit(2) );
  end
  DT(1,:) = DT(2,:);

  air_thermal_ode = @(t,y) DT*y;

  disp('Running ODE solver...')
  sim(c1).sol = ode15s(air_thermal_ode,P.tlims,T0);

  % Only plot 100 points in each direction
  plots.rng.leng = 100;
  if length(sim(c1).sol.x) > plots.rng.leng
    % Get plotsrng.leng equally-spaced points from time
    plots.rng.t = round(linspace(1,length(sim(c1).sol.x),plots.rng.leng));
  else
    plots.rng.t = 1:length(sim(c1).sol.x);
  end
```

```
  plots.t = sim(c1).sol.x;
  if npoints > plots.rng.leng
    % Get plotsrng.leng equally-spaced points from radius
    plots.rng.r = round(linspace(1,npoints,plots.rng.leng));
  else
    plots.rng.r = 1:npoints;
  end
  plots.r = r(plots.rng.r);

  figure('Name',[sim(c1).name,' mesh'])
  set(gcf,'WindowStyle','docked'); clf;
  mesh(plots.t,plots.r,sim(c1).sol.y(plots.rng.r,plots.rng.t)); hold on; shading interp;
  grid on; view([105, 30]);
  ylabel('Radial position (m)'), xlabel('Time (s)'), zlabel('Temperature (K)');
  title([sim(c1).name,10,'Plot of temperatures with time']);

  figure('Name',[sim(c1).name,' contours'])
  set(gcf,'WindowStyle','docked'); clf;
  mesh(plots.t,plots.r,sim(c1).sol.y(plots.rng.r,plots.rng.t)-sim(c1).t.end,...
    'FaceColor','none','EdgeColor',[0.5,0.5,0.5]); hold on;
  [conts,h] = contour3(sim(c1).sol.x,r,sim(c1).sol.y-sim(c1).t.end,...
    sim(c1).t.contours);...
    set(h,'LineWidth',2); clabel(conts);
  grid on; view([105, 30]); % clabel(conts);
  ylabel('Radial position (m)'), xlabel('Time (s)'), zlabel('Temperature (K)');
  title([sim(c1).name,10,'Contour plot of temperatures with time']);

  sim(c1).eigens = eig(DT);

  figure('Name',[sim(c1).name,' eigenvalues'])
  set(gcf,'WindowStyle','docked'); clf; hold on;
  plot(log(abs(sort(sim(c1).eigens))),'k+-'); grid on;
  ylabel('Eigenvalue'), title([sim(c1).name,10,'Plot of log(abs(sort( eigenvalues of DT ))']);
  disp('Done!');
end
```

```
function exp_fitter
% This function loads a saved data file, runs out.yv through model_pack to get opthets,
% deletes the rest, then tries to fit exponentials to it.
tic; clc; close all;
dotimestamp, fprintf('Running %s ...\n',mfilename);

%% Load data, tidy it up, get opthets
dotimestamp, filename = input(' - Enter the filename to load ''out'' from >> ','s');
load(['outputs/',filename,'.mat']);

if out.GP.xxl ~= -1; %#ok<NODEF>
    out.t = out.te;
    out.yv = out.yev;
end
out.yv = real(out.yv);
out.ys = model_pack(out.yv);

% Save what we want, delete out
in.t = out.t;
THETA = out.ys.opthets';
in.GP = out.GP;
clearvars out;

% Delete the even columns of THETA since they correspond to asymmetric polynomials,
%  no significant contribution to the steady state
THETA(:,2:2:size(THETA,2)) = [];

%% SVD the matrix to get an idea of how many exponentials we have
DS = svd(THETA);
for c1 = 1:length(DS)
    dotimestamp, fprintf(' DS(%i) = %g\n',c1,DS(c1));
end

%% Newton-Raphson tau to get better values
% Assign initial values for tau
tau = [2e-6; 3e-5; 1e-7];
% tau = [1e-3; 3e-4; 1e-5];
perturb = 0.001; % size of perturbations to tau for finding derivatives
THpack.TH = THETA;
THpack.t = in.t; % pack up data for easier passing to subfunction
HESSIAN = get_HESSIAN(tau,perturb,THpack); % Get Hessian of initial tau
xi = 1;
loops = 150;
rec.tau = zeros(length(tau),loops);
rec.xi = zeros(loops,1);
derivs = 0*tau;

for c1 = 1:loops
    rec.tau(:,c1) = tau;
    rec.xi(c1) = xi;

    if mod(c1,floor(loops/10)) == 1 || c1 == loops
        %% Talk about the current values
        [SSSQ,THpredict,SSQ] = get_SSSQ(tau,THpack);
        dotimestamp, fprintf(' - Loop %i, tau = ',c1);
        for c2 = 1:length(tau)
            fprintf('%g, ',tau(c2))
        end
        fprintf('and SSSQ = %g\n',SSSQ);
        if c1 == 1 || c1 == loops % If on first or last loop
            % Plot it
            figure('Name','exp_fitter output');
            for c2 = 1:size(THETA,2)
                subplot(3,2,c2);
                hold on;
                plot(THpack.t,THpack.TH(:,c2),'k','DisplayName','Original \Theta');
                plot(THpack.t,THpredict(:,c2),'b','DisplayName','\Theta_p');
                legend('show','Location','Best'); grid on;
                xlabel('t (s)'); title(sprintf('\\theta %i, SSQ = %g',c2*2-1,SSQ(c2)));
```

```
            end
        end
    end
    %% Calculate the correction to apply
    % Calculate the derivs vector
    for c2 = 1:length(tau) % Which element we're perturbing
        ptb_v = 0*tau;
        ptb_v(c2) = tau(c2)*perturb; % Perturbation amount
        SSSQ(1) = get_SSSQ(tau - ptb_v ,THpack);
        SSSQ(2) = get_SSSQ(tau + ptb_v ,THpack);
        derivs(c2) = (SSSQ(2)-SSSQ(1)) / (2*ptb_v(c2)); % Numerical derivative
    end
    % Calculate the correction
    correc = HESSIAN\derivs;

    %% Try some xi values
    looping = true;
    while looping
        tau_p = tau - xi*correc; % Try the current xi
        HESSIAN = get_HESSIAN( tau_p, perturb,THpack); % get the "next" Hessian
        derivs_p = HESSIAN * correc; % Estimate the next derivs
        if max((derivs_p - derivs)./derivs) > 0.01 % if too divergent
            xi = xi / 2; % Reduce xi and try again
        else % if good enough
            tau = tau_p; % use current tau (and current Hessian, already calc'd)
            looping = false; % stop retrying xi values
        end
    end

end

figure('Name','Tau values')
for c1 = 1:length(tau)
    subplot(length(tau),1,c1),plot(rec.tau(c1,:),'k+-');
    grid on; title(sprintf('tau %i',c1));
end
dotimestamp, fprintf(' - %s completed.\n',mfilename);
end

function [SSSQ, THpredict, SSQ] = get_SSSQ( tau, THpack )
% This function is used by exp_fitter to calculate the sum of sums of squares of errors
% of a particular set of tau-values, when used to predict the THETA matrix. It also
% outputs the predicted \Theta matrix and the sums of squares of errors.

% Calculate the FUNCS matrix, the functions which we're linearly combining;
FUNCS = zeros(length(THpack.t),length(tau)+1);
FUNCS(:,1) = 1; % Constant term
for c1 = 1:length(tau)
    FUNCS(:,c1+1) = exp(-tau(c1).*THpack.t);
end
% Calculate the coefficient matrix with the psudoinverse of FUNCS
COEFF = (FUNCS' * FUNCS) \ FUNCS' * THpack.TH;
% Reconstruct the data and calculate the sum of the differences squared
THpredict = FUNCS * COEFF;
SSQ = sum( (THpack.TH-THpredict).^2 );
SSSQ = sum(SSQ);
end

function HESSIAN = get_HESSIAN( tau, perturb, THpack )
% This function calculates the Hessian matrix for a set of tau-values.
HESSIAN = zeros(length(tau));
for c1 = 1:length(tau) % For every entry in tau
    for c2 = 1:c1 % For every entry in tau up to the current one
        ptb_v{1} = 0*tau; %#ok<*AGROW>
        ptb_v{2} = 0*tau;
        ptb_v{1}(c1) = tau(c1)*perturb; % Perturbation in c2-nd direction
        ptb_v{2}(c2) = tau(c2)*perturb; % Perturbation in c3-rd direction
        SSSQ(1) = get_SSSQ( tau - ptb_v{1} - ptb_v{2} ,THpack);
        SSSQ(2) = get_SSSQ( tau + ptb_v{1} - ptb_v{2} ,THpack);
```

```
        SSSQ(3) = get_SSSQ( tau - ptb_v{1} + ptb_v{2} ,THpack);
        SSSQ(4) = get_SSSQ( tau + ptb_v{1} + ptb_v{2} ,THpack);
        HESSIAN(c1,c2) = ( SSSQ(4) - SSSQ(3) - SSSQ(2) + SSSQ(1) )...
            / (4 * ptb_v{1}(c1) * ptb_v{2}(c2) );
        HESSIAN(c2,c1) = HESSIAN(c1,c2); % symmetric
    end
end

end
```

```
% This script is for use with a modified version of SW_run.
%   The modification is to turn it into a function, as follows:
%         function [out, pp] = SW_run(hdot_kick, th_kick)
%   It also needs to have the inputs inserted into the GP definition, and
%   the 'clear' statement at the start excised.
%   GP.PlotGraphs should be false.
%   GP.runtime should be around 2.
%   GP.xxl should be 0.
%   The internal saving system should be turned off.
%
% Modified on 2014-01-23 by SW.
tic; clear all; clc; close all;
dotimestamp, fprintf('Running %s ...\n',mfilename);

machine_ID = 5;

num_of_machines = 5;

omega = 0.4;

%% Initialise
vert = 60;
horiz = 90;

switch machine_ID
  case 1
    vert_range = 1:16;
  case 2
    vert_range = 17:27;
  case 3
    vert_range = 28:38;
  case 4
    vert_range = 39:49;
  case 5
    vert_range = 50:60;
end

inputs.th_kick = linspace(0, pi/2, vert)';
inputs.kick_vel = 2.75 + 0.125*((1:horiz)-1)';

inputs.th_kick = inputs.th_kick(vert_range);

dotimestamp,fprintf('Running with machine_ID = %i, omega = %g, and vert_range =\n',...
  machine_ID, omega);
disp(vert_range);

outputs.th_end = zeros(length(inputs.th_kick),length(inputs.kick_vel));
outputs.m.exh = outputs.th_end;
outputs.m.net = outputs.th_end;
outputs.wd = outputs.th_end;

%% Run simulations
for c1 = 1:length(inputs.th_kick)
  for c2 = 1:length(inputs.kick_vel)
    dotimestamp, fprintf('- Datetime is %s\n',...
      datestr(now,'yyyy-mm-dd_HH:MM:SS') );
    dotimestamp, fprintf('- Running %i of %i, %i of %i ...\n',...
      c1, length(inputs.th_kick), c2, length(inputs.kick_vel) );

    out = SW_run( omega, inputs.th_kick(c1), inputs.kick_vel(c2) );

    if out.GP.simfail % If the sim failed
      outputs.th_end(c1,c2) = NaN; % Gimme a NaN
      outputs.m.net(c1,c2) = NaN;
      outputs.m.exh(c1,c2) = NaN;
      outputs.wd(c1,c2) = NaN;
    else
      out.yes = SW_pack(out.yev);
      out.ys = SW_pack(out.yv);
```

```
      if out.t(end)>150 % If we have more than 150s of data
        % Only base the linefits on the last 150s of data
        pow.index = find(out.t<out.t(end)-150,1,'last');
      else % Otherwise use all of it
        pow.index = 1;
      end
      % Linear polyfits:
      pow.polys.m.net = polyfit(out.t(pow.index:end), ...
        out.ys.m_out.net(pow.index:end)',1); % Net airmass
      pow.polys.m.exh = polyfit(out.t(pow.index:end), ...
        out.ys.m_out.exh(pow.index:end)',1); % Exh airmass
      pow.polys.E = polyfit(out.t(pow.index:end), ...
        out.ys.wd(pow.index:end)',1); % Work done
      % Save those variables
      outputs.m.net(c1,c2) = pow.polys.m.net(1);
      outputs.m.exh(c1,c2) = pow.polys.m.exh(1);
      outputs.wd(c1,c2) = pow.polys.E(1);

      % Find the last angle the piston stopped
      outputs.th_end(c1,c2) = out.yes.theta(find(out.ie==4,1,'last'));
      % Tidy up
      clear out;
    end
  end
end

num_isnans = sum(isnan(outputs.th_end(:)));
endtime = toc;

save(sprintf('omega_%.1f_machine_ID_%i.mat',omega,machine_ID))

dotimestamp,fprintf('Number of NaNs found = %i\n',num_isnans);

dotimestamp, fprintf('Simulations are complete.\n');
```

```
clc; clear all; close all;

vals = 3:6;

for c1 = vals
  temp = load(sprintf('agg_%i',c1));
  agg{c1} = temp.agg;
end

plots.fields = {'m_net','ratio','power_norm'};

plots.clabel = false;

plots.titlestr.m_net = 'Net rate of air being exhausted (kg/s)';
plots.titlestr.ratio = 'Net airmass rate divided by exhausted rate';
plots.titlestr.power_norm = 'Fraction of max power at that speed';

plots.contours.m_net = -2:0.05:4;
plots.contours.ratio = -2:0.02:1;
plots.contours.power_norm = -1:0.01:2;

% Plot the four-layer simple version first
for c1 = 1:length(plots.fields)
  superagg_plotfun(agg,vals,plots,plots.fields{c1});
  set(gcf,'Name',['ko_simple_',get(gcf,'Name'),'_3D']);
  set(gca,'xlim',[-1,25],'zlim',[0.3,0.6]);
end

% Only interested in the three-layer version from now on
vals = 3:5;

% Initialise
omega = [];
m_net.kick_vel = [];
m_net.th_kick = [];
pow = m_net;
for c1 = vals
  omega = [omega, agg{c1}.omega];
  % Find max m_net location
  [row,col] = find(agg{c1}.m_net == max(agg{c1}.m_net(:)));
  m_net.kick_vel = [m_net.kick_vel, agg{c1}.kick_vel(col)];
  m_net.th_kick = [m_net.th_kick, agg{c1}.th_kick(row)];
  % Find max power_norm location
  [row,col] = find(agg{c1}.power_norm == max(agg{c1}.power_norm(:)));
  pow.kick_vel = [pow.kick_vel, agg{c1}.kick_vel(col)];
  pow.th_kick = [pow.th_kick, agg{c1}.th_kick(row)];
end

% Omega set for polyval polyfit
omega_set = linspace(omega(1),omega(end),1e3);

% Max net air out
% Quadratic to get th_kick for given kick_vel
m_net.th_poly = polyfit(m_net.kick_vel,m_net.th_kick,2);
% Quadratic to pick best kick_vel for a given omega
m_net.kick_poly = polyfit(omega,m_net.kick_vel,2);
% kick_vel data for curve
m_net.kick_vel_set = polyval(m_net.kick_poly,omega_set);
% th_kick data for curve
m_net.th_kick_set = polyval(m_net.th_poly,m_net.kick_vel_set);
figure('Name','ko_quadratics');
subplot(1,2,1), hold on;
plot(omega,m_net.kick_vel,'k+');
plot(omega_set,m_net.kick_vel_set,'b-');
xlabel('Rotor speed $\dot{\theta}$ (rad/s)','Interpreter','LaTeX');
ylabel('Kick velocity $\dot{h}_k$ (m/s)','Interpreter','LaTeX');
title('First quadratic');

subplot(1,2,2), hold on;
```

```
plot(m_net.kick_vel,m_net.th_kick,'k+');
plot(m_net.kick_vel_set,m_net.th_kick_set,'b-');
ylabel('Kick angle $\theta_k$ (rad)','Interpreter','LaTeX');
xlabel('Kick velocity $\dot{h}_k$ (m/s)','Interpreter','LaTeX');
title('Second quadratic');

% Max power extraction
% Quadratic to get th_kick for given kick_vel
pow.th_poly = polyfit(pow.kick_vel,pow.th_kick,2);
% Quadratic to pick best kick_vel for a given omega
pow.kick_poly = polyfit(omega,pow.kick_vel,2);
% kick_vel data for curve
pow.kick_vel_set = polyval(pow.kick_poly,omega_set);
% th_kick data for curve
pow.th_kick_set = polyval(pow.th_poly,pow.kick_vel_set);
% Hypotenuse length
hypot = 10;
% Angles from m_net peak to pow peak
angles = atan(( pow.th_kick - m_net.th_kick )...
  ./ ( pow.kick_vel - m_net.kick_vel ) );
% Positions of right edges
rightedge.kick_vel = m_net.kick_vel + hypot*cos(angles);
rightedge.th_kick = m_net.th_kick + hypot*sin(angles);

% Remove low-scoring values (to even out color plotting)
for c1 = vals
  agg{c1}.m_net(agg{c1}.m_net < -0.1) = NaN;
  agg{c1}.ratio(agg{c1}.ratio < 0) = NaN;
  agg{c1}.power_norm(agg{c1}.power_norm<0.6) = NaN;
end

for c1 = 1:length(plots.fields)
  superagg_plotfun(agg,vals,plots,plots.fields{c1});
  set(gcf,'Name',['ko_lines_',get(gcf,'Name'),'_3D']);
  plot3(m_net.kick_vel,m_net.th_kick,omega,'k+');
  plot3(m_net.kick_vel_set,m_net.th_kick_set,omega_set,'k-');
  if c1>1
    plot3(pow.kick_vel_set,pow.th_kick_set,omega_set,'k-');
    plot3(pow.kick_vel,pow.th_kick,omega,'k+');
    for c2 = 1:3
      plot3([m_net.kick_vel(c2),rightedge.kick_vel(c2)],...
        [m_net.th_kick(c2),rightedge.th_kick(c2)],omega(c2)*[1,1],'k-')
    end
  end
end

% Grid for vis
grid_height = 94;
grid_width = 80;
omega_grid = linspace(omega(1),omega(end),grid_height)';
m_net.kick_vel_grid = polyval(m_net.kick_poly,omega_grid);
m_net.th_kick_grid = polyval(m_net.th_poly,m_net.kick_vel_grid);
pow.kick_vel_grid = polyval(pow.kick_poly,omega_grid);
pow.th_kick_grid = polyval(pow.th_poly,pow.kick_vel_grid);

angles = atan(( pow.th_kick_grid - m_net.th_kick_grid )...
  ./ ( pow.kick_vel_grid - m_net.kick_vel_grid ) );
rightedge.kick_vel_grid = m_net.kick_vel_grid + hypot*cos(angles);
rightedge.th_kick_grid = m_net.th_kick_grid + hypot*sin(angles);

grid.kick_vel = [];
grid.th_kick =[];
for c1 = 1:length(omega_grid)
  grid.kick_vel = [grid.kick_vel; linspace(...
    m_net.kick_vel_grid(c1), rightedge.kick_vel_grid(c1), grid_width)];
  grid.th_kick = [grid.th_kick; linspace(...
    m_net.th_kick_grid(c1), rightedge.th_kick_grid(c1), grid_width)];
end
grid.omega = omega_grid * ones(1,grid_width);
```

```
for c1 = 1:length(plots.fields)
  superagg_plotfun(agg,vals,plots,plots.fields{c1});
  set(gcf,'Name',['ko_grid_',get(gcf,'Name'),'_3D']);
  mesh(grid.kick_vel,grid.th_kick,grid.omega,...
    grid.omega*0+agg{vals(1)}.(plots.fields{c1})(1),...
    'FaceColor','none','EdgeColor',[0,0,0]); hold on;
end

% Get control surface data
temp = load('agg_control.mat'); control = temp.agg;
% Interpolator to get th_kick values from omega and kick_vel values
interpolator = scatteredInterpolant(control.kick_vel(:),control.omega(:),control.th_kick(:));


for c1 = 1:length(plots.fields)
  superagg_plotfun(agg,vals,plots,plots.fields{c1});
  set(gcf,'Name',['ko_contsurf_',get(gcf,'Name'),'_3D']);
  [C,h] = contour3(control.kick_vel,control.omega,control.(plots.fields{c1}),...
    plots.contours.(plots.fields{c1}));
  for c2 = 1:length(h)
    set(h(c2),'ZData',get(h(c2),'YData')); % set Z (omega) to Y
    set(h(c2),'YData',... % Look up appropriate values for Y (th_kick) using interpolant
      interpolator(get(h(c2),'XData'),get(h(c2),'ZData')) );
  end
  if plots.clabel
    h2 = clabel(C,plots.contours.(plots.fields{c1})(end:-3:1));
    for c2 = 1:length(h2)
      try
        pos = get(h2(c2),'Position');
        set(h2(c2),'Position',[pos(1),interpolator(pos(1),pos(2)),pos(2)]);
      catch
        set(h2(c2),'ZData',get(h2(c2),'YData'));
        set(h2(c2),'YData',interpolator(get(h2(c2),'XData'),get(h2(c2),'ZData')));
      end
    end
  end
  % Draw box around control surface
  patch(...
    [m_net.kick_vel_set';rightedge.kick_vel_grid(end:-1:1)],...
    [m_net.th_kick_set';rightedge.th_kick_grid(end:-1:1)],...
    [omega_set';omega_grid(end:-1:1)],...
    'k','FaceColor','none','EdgeColor','k');
  % Add in midline (top and bottom are covered by patch)
  plot3([m_net.kick_vel(2),rightedge.kick_vel(2)],...
    [m_net.th_kick(2),rightedge.th_kick(2)],omega(2)*[1,1],'k-')
end

for c1 = 1:length(plots.fields)
  figure; hold on;
  set(gcf,'Name',['ko_contsurf_',plots.fields{c1},'_2D']);
  [C,h] = contour(control.kick_vel,control.omega,control.(plots.fields{c1}),...
    plots.contours.(plots.fields{c1}));
  % clabel stuff needs to be added for this one
  % Draw box around control surface
  patch(...
    [m_net.kick_vel_set';rightedge.kick_vel_grid(end:-1:1)],...
    [omega_set';omega_grid(end:-1:1)],...
    'k','FaceColor','none','EdgeColor','k');
  % Add in midline
  plot([m_net.kick_vel(2),rightedge.kick_vel(2)],...
    omega(2)*[1,1],'k-')
  % Add in optimum points
  xlabel('Velocity $\dot{h}_k$ (m/s)','interpreter','latex');
  ylabel('Rotor speed $\dot{\theta}$ (rad/sec)','interpreter','latex');
  c = colorbar;
  ylabel(c,plots.titlestr.(plots.fields{c1}));
end
```

```
function superagg_plotfun(agg,vals,plots,field)
figure('Name',field);
axes; hold on;
view([-28,22]);

for c1 = vals
  % Create contours
  [C,h] = contour3(agg{c1}.kick_vel,agg{c1}.th_kick,agg{c1}.(field),...
    plots.contours.(field));
  % Move contours into position on plane
  for c2 = 1:length(h)
    set(h(c2),'ZData',get(h(c2),'ZData')*0+agg{c1}.omega);
  end
  if plots.clabel
    % Label contours
    h2 = clabel(C,plots.contours.(field)(end:-3:1));
    for c2 = 1:length(h2)
      try
        % Move text contourlabels
        pos = get(h2(c2),'Position');
        set(h2(c2),'Position',[pos(1:2),agg{c1}.omega]);
      catch
        % Move + contourlabel markers
        set(h2(c2),'ZData',agg{c1}.omega);
      end
    end
  end
  % Draw flat box at data limits for this speed
  patch(agg{c1}.kick_vel([1,end,end,1]),agg{c1}.th_kick([1,1,end,end]),...
    agg{c1}.omega*[1,1,1,1],'k','FaceColor','none','EdgeColor','k');
end
xlabel('Kick velocity $\dot{h}_k$ (m/s)','interpreter','latex');
ylabel('Kick angle $\theta_k$ (rad)','interpreter','latex');
zlabel('Rotor speed $\dot{\theta}$ (rad/sec)','interpreter','latex');
box on;
set(gca,'xlim',[-1,24],'ylim',[-0.1,1.7],'zlim',[0.3,0.5],...
  'ztick',0.3:0.05:0.6);

% title(plots.titlestr.(field));
c = colorbar;
ylabel(c,plots.titlestr.(field));

% Plot backside box
%   axis auto;
%   axis manual;
%   xlim = get(gca,'xlim'); ylim = get(gca,'ylim'); zlim = get(gca,'zlim');
%   patch(xlim([2,2,2,2]),ylim([1,1,2,2]),zlim([1,2,2,1]),...
%     'k','FaceColor','none','EdgeColor','k');
%   patch(xlim([1,1,2,2]),ylim([2,2,2,2]),zlim([1,2,2,1]),...
%     'k','FaceColor','none','EdgeColor','k');
end
```