THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

Reducing the Global Warming Potential of Coal Mine Ventilation Air by Combustion in a Free-Piston Engine

by

Brendan Twain O'Flaherty

Bachelor of Engineering, Honours IIa

A thesis submitted for the degree of Doctor of Philosophy at The University of Queensland in June 2012

Division of Mechanical Engineering,
School of Mechanical and Mining Engineering,
The University of Queensland,
Australia.

Declaration by Author

This thesis is composed of my original work and contains no material previously published or written by another person except where due reference has been made in the text. I have clearly stated the contribution by others to jointly-authored works that I have included in my thesis.

I have clearly stated the contribution of others to my thesis as a whole, including statistical assistance, survey design, data analysis, significant technical procedures, professional editorial advice, and any other original research work used or reported in my thesis. The content of my thesis is the result of work I have carried out since the commencement of my research higher degree candidature and does not include a substantial part of work that has been submitted to qualify for the award of any other degree or diploma in any university or other tertiary institution. I have clearly stated which parts of my thesis, if any, have been submitted to qualify for another award.

I acknowledge that an electronic copy of my thesis must be lodged with the University Library and, subject to the General Award Rules of The University of Queensland, immediately made available for research and study in accordance with the Copyright Act 1968.

I acknowledge that copyright of all material contained in my thesis resides with the copyright holder(s) of that material.

Statement of Contributions to Jointly Authored Works Contained in the Thesis

No jointly-authored works.

Statement of Contributions by Others to the Thesis as a Whole

No contributions by others.

Statement of Parts of the Thesis Submitted to Qualify for the Award of Another Degree

None.

Published Works by the Author Incorporated into the Thesis

1. O'Flaherty, B.: 2004. Mitigation of Methane from Ventilation Air using a Free-Piston Combustor. Technical Report 2005/04, Department Mechanical Engineering, University of Queensland, Australia

Additional Published Works by the Author Relevant to the Thesis but not Forming Part of it

1. Jacobs, P., Gollan, R., Denman, A., O'Flaherty, B., Potter, D., Petrie-Repar, P., Johnston, I.: 2010. Eilmer3 theory book. Technical Report 2010/09, Department Mechanical Engineering, University of Queensland, Australia

2. Gollan, R., O'Flaherty, B., Jacobs, P., Johnston, I.: 2009. Casbar User's Guide. Technical Report DSTO-GD-0594, Defence Science and Technology Organisation, Edinburgh, South Australia

3. Jacobs, P., Gollan, R., Blyton, P., Bosco, A., Boutamine, D., Brown, L., Buttsworth, D., Chan, W., Chiu, S., Craddock, C., Cook, B., Czapla, J., de Miranda-Ventura, C., Denman, A., Gildfind, D., Goozeé, R., Hess, S., Jacobs, C., Johnston, I., Joshi, O., Kirchhartz, R., McGilvray, M., Mee, D., Montgomery, L., Nap, J.-P., O'Flaherty, B., Petrie-Repar, P., Potter, D., Ramanath, D., Scott, M., Sheikh, U., Stewart, B., Tang, J., Tanimizu, K., van der Laan, P., Vesudevan, J., Wendt, M., Wheatley, V., Window, A., Wojciak, H., Zander, F.: 2008. The Eilmer3 Code: User Guide and Example Book. Technical Report 2008/07, Department Mechanical Engineering, University of Queensland, Australia

Acknowledgements

I gratefully acknowledge the help and guidance of my current supervisors, Assoc. Prof Peter Jacobs and Prof Richard Morgan (UQ, St Lucia), and of my late supervisor Dr Michael Wendt (CSIRO, QCAT), without whom this document would not have been completed. This thesis is dedicated to Michael Wendt who unfortunately passed away in 2005. He formed the initial ideas for this thesis and was a lovely man: enthusiastic, creative and helpful. Dr Peter Jacobs has been very supportive for every year this long PhD has taken me. He has helped me with scope by telling me where to stop (and hindered me with scope by finding new and interesting papers) and was very timely with his guidance. Thanks to Professor Richard Morgan for his very helpful feedback on the completed document. Also to Rowan Gollan for helping me with code, Linux, disillusionment and LaTeX among many, many other things and never asking for anything in return. He is the kind of person who finds collaboration truly rewarding. Thanks to my friends who have helped me keep a semblance of work-life balance. And last but not least, thanks to my family, Marie and Joscelyne, for their unwavering love and support and to Diana Cholewska for her time taken with many proof reads and for making sure I ate during the last few months.

Dr Michael Noel Wendt (1968–2005)

Preface

Originally, this project was to be an experimental investigation into a device for mitigating the global warming potential of coal mine ventilation air. With the passing of Dr Michael Wendt, it became infeasible to continue this approach and the investigation became primarily a numerical one.

Abstract

The increase in the atmospheric mole fraction of greenhouse species since pre-industrial times has forced Earth's atmosphere to higher temperatures. The individual effect of these species on the atmosphere is compared using a global warming potential (GWP) index, which is the cumulative radiative forcing of a species over a given period relative to carbon dioxide (CO_2). After CO_2, the next highest contributing species to global warming is methane (CH_4). Over a hundred-year period, the GWP of CH_4 produces a twenty-five times greater heating effect than CO_2[1].

One such anthropogenic source of CH_4 is that of underground coal mines. For Australia, this source constitutes 6.5% of its greenhouse gas emissions[2]. Of this, almost two-thirds are contained in mine ventilation air at mole fractions typically between 0.3 and 0.7% and flow rates between 150 and 300m^3s^{-1}. The CH_4 is purposefully diluted to ensure safe operation of the mine. If the CH_4 contained in the ventilation air from these mines were converted to CO_2 by combustion, for example, the result would be a 73% reduction in the GWP of ventilation air. The problem that will be addressed in this thesis is: can the CH_4 in ventilation air be oxidised using a self-sustaining process (and without a supplementary fuel or catalyst) in order to reduce its global warming potential?

For selection of a device to employ for this task, consideration was given to both the operating temperature and the thermal efficiency. After a review of potential devices a reciprocating engine concept was selected for investigation, primarily since compression ignition occurs at temperatures too low for nitrogen chemistry to be significant.

Engines are by definition work producing devices, however, the purpose of this device is not principally to produce work but instead to burn ventilation air, converting it to less harmful products. The engine would be applied primarily as an emission control device. However, even without aiming to produce power, the operation of an engine on a <0.7% mole fraction of methane represents a substantial challenge. Diesel engines, for example, may run well at very low global equivalence ratios, but ignition occurred in a region surrounding vaporising droplets where the stoichiometric ratio locally passes through the optimum range for ignition. It is harder to ignite the low equivalence ratios for premixed CH_4 mixtures of uniform composition. At 37MPa, for example, the ignition temperature is about 1330K – much higher than the pre-injection temperatures of about 900K found in even large stationary diesel engines. The main objective of this thesis is to evaluate a conceptual engine design which is capable of achieving combustion of these weak mixtures.

The proposed device makes use of the "free-piston" concept, whereby an unconstrained piston is contained between two in-line, opposing combustors. Oscillation of the piston along

[1] Solomon, S., Qin, D., Manning, M., Chen, Z., Marquis, M., Averyt, K., Tignor, M., H.L., M.: 2007. Contribution of Working Group I to the Fourth Assessment Report of the Intergovernmental Panel on Climate Change. Technical report, Cambridge, United Kingdom and New York, NY, USA

[2] Wendt, M., Mallett, C., Lapszewicz, J., Xue, S., Foulds, G., Mark, R., Sharma, S., Dannell, R., Worrall, R., Balusu, R.: 2000. Methane Capture and Utilisation. Technical Report 723R, CSIRO

the axis of the cylinders drives the thermodynamic cycles in both combustors. A linear electric motor provides the energy to drive the engine with the moving piston representing the driven element. The same motor can also be used to extract energy from the piston if it is available.

The problem of oxidising ventilation air in this way was approached in three parts which form the body of material for this thesis. The first investigated the thermochemical properties of ventilation air itself to find a suitable gas model and kinetic mechanism. The second investigated the modelling of the component processes including piston dynamics, cylinder exhausting, heat-transfer, friction and control models. The third combined these processes into a free-piston engine model which was then used to find the minimum mole fraction of CH_4 required to sustain operation.

The free-piston engine designed for this application had a 0.16m bore, 1.1m cylinder length and 200kg piston mass. The net cycle output was found to be insufficient to overcome the associated losses with the models based on current diesel engine technology and the T4 shock tunnel. The analysis showed that low friction piston seals will be essential to make the concept work. Lower friction seals may be able to be developed since the sealing demands for this engine are not as rigorous as for a normal engine, where the sump oil has to be protected from gas leakage.

If the CH_4 content increases above about 0.9%, the cycles become self sustaining and positive energy can be generated. Alternatively, if about $15\,\mathrm{MJ.kg_{CH_4}}^{-1}$ of work were added, this device could be used to reduce the GWP of ventilation air. In effect, for this application, the engine would have to be partially motored. Multiple engines would be required to process all of the ventilation air.

This thesis has established the basic principles for evaluating the viability of systems for low mole fraction CH_4 conversion by combustion. It sets the groundwork for a family of similar devices and has identified critical issues which need to be addressed to enable the technology. The analysis can be refined as needed by incorporating more detailed physical models of the various processes involved, such as multidimensional flow. Useful generic tools have also been developed which may have other applications.

Keywords

global warming potential, free-piston engine, real gas models, finite-rate chemistry, unsteady heat transfer, mixed friction, control

Australian and New Zealand Standard Research Classifications (ANZSRC)

0913 100%

Contents

Contents vii

List of Figures xi

List of Tables xix

Nomenclature xx

Glossary xxiii

1 Introduction 1

1.1 Anthropogenic sources of atmospheric methane 1
1.2 Reducing the GWP of ventilation air by combustion 3
1.3 The potential for utilisation by conventional methods 3
1.4 Candidate emission control devices 4
1.5 Comparison of candidate devices 5
1.5.1 A potential free-piston engine for emission control 6
1.6 Previous studies of free-piston engines 9
1.7 Scope and contribution of this thesis 10

I Thermodynamics and chemistry of ventilation air 13

2 Gas models 14

2.1 Gas state 14
2.1.1 Evaluation of state 15
2.1.2 Evaluation of calorific values 16
2.1.3 Mixing rules 17
2.2 Entropic relations 18
2.A Chapter end notes 19
2.A.1 General cubic equation of state and parameters 19
2.A.2 General form for some gas variables 19
2.A.3 Evaluation of temperature from internal energy and density in a real gas . 19

3 Finite-rate chemistry **21**
3.1 Chemistry of methane . . . 21
3.1.1 Methane combustion . . . 22
3.2 Reaction modelling . . . 23
3.2.1 Elementary reactions . . . 23
3.2.2 Species production rates . . . 23
3.2.3 Experimental methods for elucidating reaction models . . . 25
3.2.4 Methane decomposition . . . 26
3.2.5 Validation of pressure-dependent rate coefficients . . . 26
3.3 Programming methodology . . . 28
3.4 Perfectly mixed reactors . . . 29
3.5 Methane chemical delays . . . 33
3.5.1 Review of ignition delay experiments . . . 33
3.6 Methane heat release . . . 39
3.6.1 The production of Nitrous Oxides . . . 39
3.7 Mechanism selection summary . . . 40
3.8 The effect of real gas models on finite-rate chemistry . . . 41
3.A Chapter end notes . . . 44
3.A.1 reactor.py . . . 44
3.A.2 simple_reactor.py . . . 46
3.A.3 Hydrogen mechanism . . . 47
3.A.4 DRM19 mechanism . . . 48
3.A.5 Ignition delay definitions . . . 50
3.A.6 Calculation of the equilibrium constant . . . 51
3.A.7 Derivation of the Lindemann-Hinshelwood Form . . . 51
3.A.8 The rate coefficient of Li and Williams . . . 53
3.A.9 The rate coefficient of Smooke and Giovangigli . . . 53
3.A.10 Lua input for a pressure-dependent rate coefficient . . . 53

II Engine component models **55**

4 Engine dynamics **56**
4.1 Dynamics of a free-piston compressor . . . 56
4.2 Otto cycle . . . 60
4.3 Autoignition in a rapid compression machine . . . 62
4.4 Combustion of ventilation air in a free-piston compressor . . . 63
4.4.1 Finite-rate chemistry of a real gas in a free-piston compressor . . . 67
4.5 Dynamics of a dual piston type compressor . . . 69
4.A Chapter end notes . . . 73
4.A.1 test_free_piston_compressor.py . . . 73

5 Engine exhausting **75**
5.1 Exhausting process . . . 75
5.2 Gas dynamics . . . 76
5.2.1 Finite-waves in real gases . . . 77
5.2.2 Discharge coefficient . . . 79
5.3 Exhausting regimes . . . 80
5.4 Valve area . . . 82
5.5 Variable valve timing . . . 83
5.A Chapter end notes . . . 85
5.A.1 Riemann flux solvers . . . 85
5.A.2 Sod's shock tube [64] . . . 86
5.A.3 Full form of conservation of energy . . . 88

6 Engine losses **90**
6.1 Friction . . . 90
6.1.1 Mixed friction coefficient . . . 91
6.2 Heat transfer in a reciprocating engine . . . 94
6.2.1 Heat transfer models . . . 95
6.2.2 Quasi-steady models . . . 96
6.2.3 Unsteady models . . . 97
6.2.4 Boundary layer models . . . 99
6.2.5 Heat transfer in a free-piston compressor with and without finite-rate chemistry . . . 105
6.A Chapter end notes . . . 108
6.A.1 Quasi-steady models . . . 108
6.A.2 Unsteady models . . . 110
6.A.3 Numerical boundary-layer methods . . . 111

7 Engine control **112**
7.1 Control . . . 112
7.1.1 Finding the required electromagnetic force . . . 112
7.1.2 Control of entrained mass of ventilation air . . . 113
7.1.3 PID control of the piston velocity . . . 113
7.2 Engine start-up . . . 114
7.A Chapter end notes . . . 116
7.A.1 Finding the required electromagnetic force to reach a peak temperature . 116
7.A.2 Piston-solenoid dynamics for a linear system . . . 116

III Complete engine model **118**

8 Engine cycle **119**
8.1 Engine program overview 119
8.2 Engine program verification 120
8.3 Parametric study 120

9 Conclusions **129**
9.1 Summary of the reference engine design 129
9.2 Findings 130
9.3 Future work 131

References **133**

A Gas model source code **140**
A.1 Real equations of state 140
A.2 Real thermal behaviour 151
A.3 Pressure-dependent rate coefficient 159

B Engine source code **163**
B.1 Free-piston engine kernel 163
B.2 Free-piston engine tests 166
B.3 Free-piston engine system 188
B.4 Free-piston engine models 220
B.5 Free-piston engine control 230
B.6 Sod's shock tube simulator 233

List of Figures

1.1 Variation of methane concentration in ventilation air for an Australian coal mine. Anonymous data reinterpreted from [6]. The flow rate for this mine varied between 150 to 300m^3s^{-1}. 2

1.2 Compression-ignition free-piston engine. This engine design is similar in geometry to those investigated in this thesis. The long piston and cylinder lengths the cylinder were a consequence of exhausting requirements. Details will be discussed more completely in later sections of this document. 7

1.3 Piston trajectory, gas temperature and methane concentration during one period of a free-piston engine. 8

2.1 Validation of the gas model implementation using isothermal compression of Helium with reinterpreted data from [28]. The Abel-Noble and van der Waals gas models deviate from the ideal gas model. 16

3.1 Pressure-dependent rate coefficient and the Arrhenius asymptotes (Reaction 51, GRI-Mech3.0). Both the Lindemann-Hinshelwood and Troe forms of Reaction 51 were compared to show the effect of the Troe coefficients. Due to limitations in the code (as of revision 672), efficiencies could not be incorporated at this level. This feature will be added in future revisions. 27

3.2 Decomposition of CH_4 (const-T, p) showing equilibrium mole fractions. The species concentrations predicted by the Smooke and Giovangigli reaction do not approach their equilibrium value because a backward rate was specified for this elementary reaction. 28

3.3 Perfectly mixed reactors (homogeneous and adiabatic) 29

3.4 Effect of timestep on the operator-split method. Effects are seen as the timestep becomes larger than the period between ignition delay and combustion delay. . . 31

3.5 Constant-pressure reaction using Hydrogen mechanism, dt_chem = 1×10^{-7}. This is the example provided in Appendix 2 of the Chemkin II manual. 32

3.6 CPFM PMR using DRM19 with mixture $n_{CH_4}, n_{O_2}, n_{N_2}$ = 1,2,7.52, dt_chem=2×10^{-7}. On each of these, Chemkin-II matches well with `Gaspy`, even down to mole fractions of 1×10^{-10}. 32

3.7 Total density versus temperature for ignition delay studies. The boxed region shows conditions for isentropic compression of a thermally-perfect gas from the atmospheric state. Thus we will continue the investigation using the ignition delay data of [36] and [37]. 36

3.8 Reduced induction times for transcribed data using the ignition delay formula of [36]. The data taken from [37] agrees well with this trend. 37

3.9 Ignition delay of experimental results for the lowest and highest density tests by Tsuboi and Wagner compared with the numerical results over the same conditions using four kinetic mechanisms. The slope predicted by the kinetic mechanism is more important than the offset as the former is an indication of global activation energy whereas the latter is a consequence of the ignition delay definition. The mechanisms of Li and Williams and Smooke and Giovangigli were not included in the comparison. The former is solely an ignition mechanism and has non-smooth species production rates. The latter is a flame mechanism that does not attempt to capture ignition delay. 38

3.10 A comparison of heat release for all six mechanisms. As can be seen, all but the ignition delay mechanism of Li and Williams result in equal heat release, albeit at different combustion delays. Tests were performed using a constant-pressure fixed-mass perfectly mixed reactor with ventilation air at initial conditions of T_0=2226 K and p=1 atm. 39

3.11 Greenhouse species for a constant-pressure, fixed-mass, perfectly mixed reactor at 2226 K, 1 atm. 40

3.12 GWP reduction for a constant-pressure, fixed-mass, perfectly mixed reactor at 2226 K, 1 atm. 40

3.13 High pressure decomposition of CH_4 (const-T, p) at T=3000K using different gas models. As can be seen, the van der Waals equation of state retards the rate of decomposition compared to the thermally perfect equation of state. In addition, the equilibrium value attained by Smooke and Giovangigli is incorrect. This is because a backwards rate is specified for this reaction. Thus, only kinetic mechanisms in which the reverse rates are calculated using the equilibrium constant should be used with real gas models. Specifying a reverse rate couples the kinetic mechanism and the thermal model, which is undesirable. The pressure was chosen to exaggerate the difference between thermally-perfect and thermally real models. Peak pressures seen in a free-piston engine are expected to be substantially lower. 42

4.1 Ideal free-piston compressor. 57

4.2 Piston trajectory. Note the sharp deceleration. Initial conditions for this case were m_p=100kg, D=0.2m, L=1.0m filled with thermally-perfect ventilation air at atmospheric conditions. 58

4.3 System energy balance showing components of piston kinetic energy and gas sensible energy for an Otto cycle using air of various gas models starting at the atmospheric state. 58

4.4 Stroke length for free-piston compression of ideal standard air where $m_p = 100$kg and $D = 0.25$m. Numerical results correspond to peak temperatures of 400K to 1800K in 200K increments. 59

4.5 The effect of real gas models on compression of ventilation air. The output here has been normalised as indicated. The analytical solution for an ideal gas is included for contrast. As can be seen, the real gas models require progressively lower piston velocities to reach the same peak temperature, which occurs at practically the same volumetric compression ratio. For example, for compression to 1400K, the piston speed needs to be about 0.8% slower using a van der Waals gas as opposed to a thermally-perfect gas. This effect increases with compression ratio. 60

4.6 The Otto cycle performed using ideal air starting at the atmospheric state and adding 140kJ.kg^{-1} heat at the point of piston reversal. This figure validates the ideal gas implementation and clearly shows the low energy content of ventilation air. This cycle has a volumetric compression ratio of 23.14, with parameters of $m_p = 400$kg, $L_c = 4$m, $D = 0.1$m and $u_{p,0} = 10$m.s^{-1}. 61

4.7 Schlieren photographs of compressed dry air with time in ms from top dead center (reprinted from [48] with permission). The piston remains stationary throughout this sequence. As can be seen, the boundary layer is highly turbulent. The density gradients are the result of spatially nonuniform temperature, since pressure gradients would disappear in a few milliseconds. The dark areas correspond to regions of higher temperature. Test conditions were: initial pressure, 107kPa, initial temperature 338K and compression ratio 12.6. The thermal boundary layer at time $t = 0$ appears to be larger than the cylinder length. 63

4.8 p-ν diagram comparing finite-rate chemistry for DRM19 and GRI-Mech3.0 mechanisms with instantaneous heat addition of 140kJ.kg$_{\text{mix}}^{-1}$ in a free-piston compressor. The gas is thermally-perfect ventilation air at initial conditions of standard temperature and pressure. Engine parameters are $m_p = 400$kg, $u_{p,0} = 11.64$m.s^{-1}, $D = 0.1$m and $L_c = 4.0$m. The compression ratio for these parameters is about 45:1 and the peak temperature (without combustion) is 1200K. Finite-rate chemistry is instrumental in answering the question of whether ventilation air can be combusted in a free-piston compressor since without finite-rate chemistry using a verified kinetic mechanism, the ignition point can only be approximated. It should also be noted that the two mechanisms given here predict different ignition delays, which is why the mechanism selection process was so thorough. The instantaneous heat addition (for the same initial conditions) shows a similar heat release, and thus the enthalpy of combustion can be said to be practically independent of the initial gas state. 64

4.9 Compression to 1200K with and without finite-rate chemistry. Reactions occur throughout the cycle, despite the fact they only release heat above 1200K. Turning the chemistry off at a lower temperature (to lower computational expense) yields incorrect results. Engine parameters are $m_p = 400$kg, $u_{p,0} = 11.64$m.s^{-1}, $D = 0.1$m and $L_c = 4.0$m. The compression ratio for these parameters is 45.15:1 and the peak temperature (without combustion) is $T_p = 1200$K. 65

4.10 Species production and destruction for ventilation air as it is raised to 1200K in a free-piston compressor. System initial conditions are $m_p = 400$kg, $u_{p,0} = 11.63$m.s^{-1}, $D = 0.1$, $L_c = 4.0$m. 66

4.11 Reduction in global warming potential of ventilation air as it is compressed to 1200K. System initial conditions are m_p=400kg, $u_{p,0}$=11.63m.s^{-1}, D=0.1, L_c=4.0m. 66

4.12 p-ν diagram of ventilation air in a free-piston compressor reaching different peak temperatures using DRM19. As can be seen, for parameters $m_p = 100$kg, $D = 0.2$m and $L_c = 5$m, a peak temperature of between 1200 and 1300 K is required for complete combustion. 67

4.13 GWP reduction of ventilation air in a free-piston compressor using a thermally perfect gas and DRM19. Temperatures correspond to peak temperature due to compression only. 68

4.14 The effect of piston mass on combustion yield for a peak temperature of 1100K. This scales linearly with residence time and exponentially with temperature (which is largely dictated by the initial velocity). Note that the piston period is largely dictated by the factor $\left(\frac{A}{m_p L_c}\right)^{1/2}$ which is related to the piston acceleration and the distance travelled. 69

4.15 The Otto cycle performed using air of different gas models, starting at the atmospheric state and adding 140kJ.kg^{-1} heat at the point of piston reversal. The thermally-perfect, Abel-Noble and van der Waals gas models have volumetric compression ratios of 24.5, 23.7 and 23.7 respectively with $m_p = 400$kg, $L_c = 4$m, $D = 0.1$m and $u_{p,0} = 10$m.s^{-1}. 70
4.16 Combustion yield in a free-piston compressor for peak temperatures of 900K-1800K (due to compression only) using DRM19, m_p =100kg, D=0.2 and L_c =5.0m. The largest discrepancy between real gas models occurs for a peak temperature of 1200K when reactions are frozen midway by the expanding piston. At temperatures higher than this, reactions go to completion for all gas models. 71
4.17 A simplified dual piston type compressor. 72
4.18 The components of a simplified free-piston engine. 72

5.1 Sod's shock-tube problem 78
5.2 Expansion of ideal air from the atmospheric state to a range of downstream pressures. This figure validates the implementation and tests for robustness over a large range of pressure ratios. Transport of species and energy have similar profiles. 79
5.3 Expansion of ideal air from the atmospheric state to a range of downstream pressures. 80
5.4 Interpolated and measured discharge coefficients for a four-valve Caterpillar engine [65]. Coefficients are held constant outside the measured range. 81
5.5 Species 'B' is being exhausted with species 'A' using perfect-mixing and perfect-displacement regimes. The initial temperature of all volumes is 298.15K, with the upstream pressure being 2atm and the cylinder and downstream pressures being 1atm. The geometry is $S = \vartheta = 1$. 81
5.6 Free-piston engine valves. 82
5.7 Valve timing logic for the right hand cylinder 83
5.8 A piecewise-constant flow state 85
5.9 An $x - t$ plot of one possible flow state after timestep Δt 85
5.10 Ideal solution to Sod's shock tube, generated using two numerical procedures: one specific to an ideal gas and one which may also be used for a real gas. There is error in both procedures (due to the iteration) which is controlled by tolerances. 88
5.11 Real gas solutions to Sod's shock tube problem. For this test $p_4 = 1{\times}10^7$ Pa, ρ_4 = 10 kg.m^{-3}, $p_1 = 1{\times}10^5$ Pa, $\rho_1 = 10$ kg.m^{-3}. Using the conditions for the classic shock tube problem (that is $\rho_4 = \rho 1 = 1$kg.m^{-3}) resulted in temperatures too low in the region between the shock wave and the contact shock and caused numerical errors. The lower density in this region and the lower total velocity clearly show the effect of the real gas models. Note that the thermally-perfect model does not differ from the ideal solution. 89

6.1 Note the new normalised velocity, which includes the friction factor and ring contact area. For a frictionless compressor, these extra terms go to zero leaving the normalization factor seen previously (for example, in Figure 4.4). This new factor shows that friction may be overcome by increasing the initial piston velocity by a predictable amount. For a simple boundary friction model (where f is constant) this amount is $\left(1 + f\frac{A_r}{A}\right)^{\frac{1}{2}}$. This provides a useful first-pass guess when estimating the additional required piston speed. 92

6.2 A mixed friction coefficient (6.5) with experimental data from [69]. For this function, $b = 0.03$m, $\mu = 0.222$Pa.s and the constants $c_1 = 4.8$, $c_2 = 0.5$ correspond to a parabolic profile. 93

6.3 Reinterpreted data of Annand [33] shown outside of its range of experimental data. Clearly, the coefficient of Annand's model does not fit this data, as it was based instead on experiments performed at Reynold's numbers between 1×10^5 and 1×10^7. 97

6.4 Conduction coefficient of thermally-perfect air. 98

6.5 A comparison between unsteady heat transfer models and the reinterpreted data of Lawton [71]. The discrepancies between heat-transfer models clearly show their specificity to a particular engine. The model of Lawton does not seem to match the data here as well as in his paper. The implementation here is clearly described in case of error. 100

6.6 Nomenclature for the unsteady thermal boundary layer and piston geometry. The idealised boundary layer shown was formed during the exhausting stage. 100

6.7 Boundary layer profile. 104

6.8 Comparison of instantaneous heat transfer with data of [71]. The engine used is a Perkins 98.4mm bore, 127mm stroke, naturally aspirated, four-cylinder, water-cooled diesel of compression ratio 15.6:1. Coefficients were chosen particularly to match with this data. This model was applied using the state equations for a crankshaft engine. 105

6.9 Comparison of the total heat transfer over one stroke between Annand's model with $c_1 = 0.12$ (6.13) and the integral boundary layer model (6.32). The engine parameters are a 98.4mm bore, 127mm stroke, with a compression ratio 15.6:1. This data was generated using a crankshaft engine model. It shows that the total heat transfer predicted by the integral boundary layer model scales relatively well with engine speed. 106

6.10 Comparing the effect of steady and unsteady heat transfer models. Annand's model had a coefficient of $c_1 = 0.0363$ (6.13) and the integral boundary layer model (6.32) was unmodified. The total heat transfer over a stroke for both heat transfer models is about 60kJ. The peak temperature for adiabatic and steady and unsteady heat transfer was 1400K, 1363K and 1323K respectively. Engine parameters were $u_{p,0} = 13.01\text{m.s}^{-1}$, $m_p = 400\text{kg}$, $R = 40$, $D = 0.1\text{m}$ and $T_s = 1400\text{K}$ (without chemistry). Each run takes about 1min on one core of an AMD 1090T Phenom X6 processor. 106

6.11 Comparing the effect of steady and unsteady heat transfer models on finite-rate chemistry. Annand's model again had a coefficient of $c_1 = 0.0363$ (6.13) and the integral boundary layer model (6.32) was unmodified. The effect of the unsteady model was to reduce the total heat transfer over the stroke to 40kJ, even while it lowered the peak temperature at the point of piston reversal and thus retarded combustion. The unsteady model in contrast predicted a total heat transfer of about 72kJ over the stroke, which is greater than without combustion. Engine parameters were $u_{p,0} = 13.01\text{m.s}^{-1}$, $m_p = 400\text{kg}$, $R = 40$, $D = 0.1\text{m}$ and $T_s = 1400\text{K}$ (without chemistry). Each run takes about 130mins on one core of an AMD 1090T Phenom X6 processor. 107

7.1 Control block diagram. The solenoid here is used to adjust the piston velocity by application of an electromagnetic force. 113

7.2 Logic for calculation of the electromagnetic force. 114

7.3 Forced response of the piston during start-up for the reference engine with a target initial velocity of $u_{p,0}$=14.77 m.s^{-1}. The maximum electric motor force applied here is $F_e = 10\,\text{kN}$. Exhausting and combustion is performed during this process. 115

7.4 System dynamics of a spring-mass-damper system coupled to an inductance-capacitance-resistance circuit. This shows the free response of the system given an initial displacement of the mass. 117

8.1 A flowchart of the free-piston engine program showing the operator-split process. 121

8.2 Isentropic engine mass and energy balance. Note that the piston velocity was set to $u_{p,0}$ at the start of each stroke, which can be seen as a step-change in total energy. 122

8.3 Engine mass and energy balance with friction and heat loss. Here, the piston velocity was set to $u_{p,0}$ at the start of the stroke, which can be seen as a step-change in the kinetic energy. This is not obvious in the total system energy because w_f, Q_L and Q_R were set to zero at the same instant such that the losses over one stroke could be quantified. 122

8.4 Initial piston velocity for a range of engines operating at an exhausting efficiency of 100%. To read this plot, first pick a diameter, D, and piston mass, m_p, from the inset key. Then take this line and point style and find the corresponding line-point combination on the plot. Cylinder length, L_c, increases from 1.0m along each line from left-to-right at 0.5m increments. The initial velocity is within a remarkably small range across all engine types indicating that this velocity is favourable for exhausting. 123
8.5 Specific entropy for the same range of engines. To read this plot, first pick a diameter, D, and piston mass, m_p, from the inset key. Then take this line and point style and find the corresponding line-point combination on the plot. Cylinder length, L_c, increases from 1.0m along each line from left-to-right at 0.5m increments. It seems that the specific entropy reduces with increasing cylinder diameter and reducing cylinder length. Piston mass has little effect on the specific entropy, but does affect the compression ratio. The effect of diameter seems to diminish as the diameter increases. Indeed this was the case: the improvement beyond $D = 0.16$m was minimal. 124
8.6 Peak temperature reached for engines where $D = 0.16$m, $m_p = 200$kg and $\eta_{\text{ex}} = 100\%$. 125
8.7 Temperature of the left and right-hand cylinders during start-up of the reference engine. Exhausting and combustion is performed during this process. 125
8.8 Piston trajectory 126
8.9 Methane entrainment and combustion 126
8.10 Species and GWP for a free-piston engine operating on ventilation air. This simulation uses the GRI-Mech3.0 mechanism to show that the generation of N_2O is negligible, even for the peak temperature (without combustion) of about 1400K. As before, the piston velocity was reset at the beginning of each stroke to sustain the engine. Only data for the left-hand cylinder is shown here for clarity. 127
8.11 Work required to operate engine for different mole fractions of CH_4 in air. 128
8.12 Extrapolation was used to find the minimum mole fraction of CH_4 required to sustain operation of the reference engine. From this graph it can seen to be about 0.92%. 128

List of Tables

1.1 Anthropogenic sources of methane [9] 1
1.2 Global warming potential for various greenhouse gases relative to CO_2. Data taken from [8]. 2
1.3 Burner type comparison 4

2.1 Species constants for a real-gas equation of state [35] 19
2.2 Equation of state derivatives 20

3.3 Hydrogen test condition 31
3.4 Methane test condition 32
3.5 Mole fractions used in ignition studies. Values have been derived when not stated explicitly. The range displayed here is based on maximum and minimum CH_4 mole fractions. 35
3.6 Concentrations used in ignition studies. Values have been derived when not stated explicitly. The range displayed here is based on maximum and minimum $[CH_4]$. . 35
3.7 A summary of ignition delay coefficients for the mixtures given in Table 3.6. . . . 35
3.8 Mechanism selection criteria. 41
3.9 Reaction 10, Li and Williams. This seems to overpredict the rate coefficient at low densities when compared to other mechanisms. 53
3.10 CH_4 decomposition model of [40]. 53
3.11 CH_4 decomposition model of [40] in Chemkin-II form. Rate coefficients are of the form of (3.6). 54

4.1 Engine cycle stages. 61

Nomenclature

This thesis brings together several areas of mechanical, heat, fluid and chemical modelling. In order not to depart too drastically from the conventions normally employed in papers on each subject, it was found to be necessary to use the same symbol to denote several different quantities. For example, h denotes both the convection coefficient and enthalpy. It should be clear from the context which quantity is being referred to.

Symbol	Description
	General
A	area, m^2
b	piston ring thickness, m
c_d	discharge coefficient
D	cylinder diameter, m
F	force, N
g	acceleration due to gravity, $m.s^{-2}$
L	length, m
m	mass, kg
r	compression ratio
R	cylinder length to diameter ratio
S	surface area, m^2
t	time, s
u,v,w	velocity components, $m.s^{-1}$
$\mathbf{U}$	conserved quantities vector
w, W	work, $J.kg^{-1}$, J
x,y,z	cartesian co-ordinates, m
ϑ	volume, m^3
	Gasdynamics, chemistry
a	speed of sound, $m.s^{-1}$
A	frequency factor, $s(cm^3mol^{-1})^{\sum_i c_i}$
c_ν, $\bar{c}_\nu$	specific heat at constant density, $J.kg^{-1}K^{-1}$, $J.mol^{-1}K^{-1}$
c_p, $\bar{c}_p$	specific heat at constant pressure, $J.kg^{-1}K^{-1}$, $J.mol^{-1}K^{-1}$

Symbol	Description
$e, \bar{e}, E$	internal energy, J.kg^{-1}, J.mol^{-1}, J
$\frac{E}{R}$	activation energy, K^{-1}
$g, \bar{g}, H$	Gibbs free energy, J.kg^{-1}, J.mol^{-1}, J
$h, \bar{h}, H$	enthalpy, J.kg^{-1}, J.mol^{-1}, J
$h_c, \bar{h}_c, H_c$	heat of combustion, J.kg^{-1}, J.mol^{-1}, J
$h_r, \bar{h}_r, H_r$	enthalpy of reaction, J.kg^{-1}, J.mol^{-1}, J
n	moles, mol
M	molecular mass, kg.mol^{-1}
$[M]$	third body value, mol.m^{-3}
p	pressure, Pa
$\mathbf{Q}$	gas state vector
$R, \mathscr{R}$	gas constant, J.kg^{-1}K^{-1}, J.mol^{-1}K^{-1}
s, S	entropy, J.kg^{-1}K^{-1},J.K^{-1}
T	temperature, K
T_{ig}	ignition temperature, K
T_c	combustion temperature, K
X_i	species i mole fraction
$[X_i]$	species i concentration, mol.m^{-3}
Y_i	species i mass fraction
α	real gas constant
$\gamma = \frac{c_p}{c_\nu}$	isentropic exponent
ν,$\bar{\nu}$	specific volume, m^{3}kg^{-1}, m^{3}mol^{-1}
ν_0,$\bar{\nu}_0$	covolume, m^{3}kg^{-1}, m^{3}mol^{-1}
ω_i	rate of production of species i, mol.s^{-1}
ϕ	stoichiometric ratio
ρ	density, kg.m^{-3}
$\bar{\rho}$	molar density, third body concentration, mol.m^{-3}
θ	total energy, J
	Thermal boundary layers
h	convection coefficient, W.m^{-2}K^{-1}
k	thermal conductivity, W.m^{-1}K^{-1}
$\text{Nu}_x = \frac{hx}{k}$	Nusselt, dimensionless temperature gradient at a surface
q, Q	heat transfer, J.kg^{-1}, J
q'', Q''	heat flux, J.kg^{-1}m^{-2}, J.m^{-2}
$\text{Re}_x = \frac{\rho u x}{\mu}$	Reynolds, ratio of the inertia and viscous forces
$\alpha = \frac{k}{\rho c_p}$	thermal diffusivity, m^{2}s^{-1}

Symbol	Description
δ	boundary layer thickness, m
$\eta = \frac{y}{\delta}$	non-dimensional height
μ	dynamic viscosity, $kg.m^{-1}s^{-1}$
	Subscripts
c	cylinder
d	discharge
e	electromagnetic
f	friction
g	gas
p	piston
pv	poppet valve
r	ring
rv	reed valve
s	stroke
sp	species
va	ventilation air
w	wall
2	momentum
3	thermal
∞	freestream

Glossary

atmospheric residence time	the time taken for an atmospheric species to decay, 1
bimolecular reaction	a reaction between two reagents, 22
chemical delay	the time required for the concentration of a reagent to fall to a specified fraction of its initial concentration [7], 32
code validation	the process of comparing the output of a program to experimental data to check the code is accurate, 28
code verification	the process of comparing the output of a program with a known result to show there are no mistakes, 28
combustion delay	the time interval between the reference state and the dead state of a mixture, 33
combustion temperature	the temperature associated with the point at which reactions go to completion., 3
combustion temperature	the temperature during compression at which combustion goes to completion, 63
discharge coefficient	the ratio of the actual mass flow rate to the inviscid mass flow rate through nozzles of the same geometric area, 79
equivalence ratio	the molar ratio of fuel-to-air normalized by the stoichiometric ratio, 22
exhausting efficiency	the mass of ventilation air used to replace the combustion products, normalised by the mass of the combustion products, 76

forward reaction	a reaction that proceeds from left to right, 23
friction coefficient	the ratio of friction force to applied normal force, 90
ignition delay	for homogeneous combustion, the ignition delay is defined by a characteristic point in the concentration history of an indicative intermediary species, commonly OH, 33
instantaneous radiative forcing	the perturbation to the energy balance of the Earth-atmosphere system, $W.m^{-2}ppm^{-1}$, 1
operator-splitting	a numerical method in which the changes in the state variables due to different coupled processes are evaluated independently, 119
peak temperature	the temperature a gas mixture reaches after adiabatic compression by a free-piston to the point of piston reversal, 57
perfect-displacement	an exhaust model whereby the inlet gas does not mix with the cylinder gas, 80
perfect-mixing	an exhaust model whereby all the inlet gas instantaneously and completely mixes with the cylinder gas, 80
pressure-dependent reaction	a reaction involving one or two reagents, depending on the third body concentration, 22
radiative forcing	the perturbation to the energy balance of the Earth-atmosphere system, $W.m^{-2}ppm^{-1}$, iv
recursive	a function or definition that refers to itself. See, recursive, 1
residual gas fraction	the mole fraction of products that remain after the exhausting stage, 75
reverse reaction	a reaction that proceeds from right to left, 23
third body concentration	the molar density of a gas mixture, 24
third body reaction	a reaction between three reagents, 22
third body value	the sum over all species of the product of concentration and collision efficiency, 24

unimolecular reaction	isomerization or decomposition of a single species to form one or two product species respectively [7], 24
valve delay	the time between port close and valve close during the compression stroke, 83
work delay	the time interval between the initial state and the peak state of the compression process, 25

Chapter 1

Introduction

In April 1990 a letter was published in Nature [8] which aggregated a body of research showing that the combined effect on climate of a number of species could rival or even exceed that of CO_2. In this letter, an index named the global warming potential was used to compare the relative contributions of individual species. Each species has its own value for *instantaneous radiative forcing* which is a function of its absorption bands and concentration. In addition, since different species decay in the atmosphere at different rates (referred to as the *atmospheric residence time*) the GWP is both a species- and time-dependent effect. Using this, it was found that, after CO_2 (contributing 71.5% of the total global emissions), CFCs (at 9.5%) and CH_4 (at 9.2%) were the next highest contributors to global warming.

1.1 Anthropogenic sources of atmospheric methane

Methane-air mixtures in concentrations <1% are common to wetlands, fermenting wastes, agriculture, oil and gas systems and coal mines. Worldwide anthropogenic sources are shown in Table 1.1.

source	**contribution, 10^9kg/yr**
ruminants (particularly cattle)	>50
rice agriculture	
landfills	10–50
coal mines	
biomass burning	
urban areas	
sewage disposal	
natural gas leakages	
industrial sources	<5

Table 1.1: *Anthropogenic sources of methane [9]*

Take the fugitive emissions from underground coal mines. Ventilation air is used to dilute the fugitive methane to low mole fractions to ensure a safe working environment. The result is a mixture of air, methane and trace amounts of other hydrocarbons that is unsteady in both concentration and flow rate (see Figure 1.1). The ventilation air is subsequently exhausted

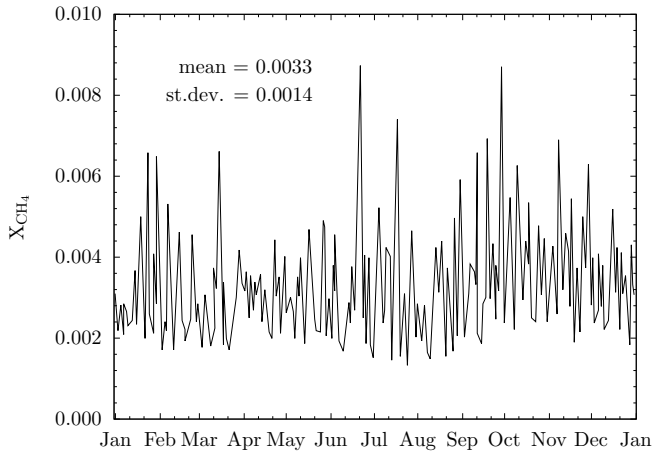


Figure 1.1: *Variation of methane concentration in ventilation air for an Australian coal mine. Anonymous data reinterpreted from [6]. The flow rate for this mine varied between 150 to 300m^{3}s^{-1}.*

through large above-ground nozzles at flow rates between 150 and 300m^{3}s^{-1}.

To quantify the impact of this source, the GWP index of Lashof and Ahuja [8] was used, which combines the effect of the radiative forcing of a molecule with its atmospheric residence time. Table 1.2 shows the global warming potential of a selected number of species.

species, i	**residence time, years**	**instantaneous forcing, W.m^{-2}ppm^{-1}**	**GWP, 1-yr molar basis**
CO_2	230	0.015	1.0
CO	2.1	0.65	1.4
CH_4	14.4	0.65	3.7
N_2O	160	3.8	180

Table 1.2: *Global warming potential for various greenhouse gases relative to CO_2. Data taken from [8].*

The GWP index of a gas mixture may be calculated using

$$\mathrm{GWP}_{\mathrm{mix}} = \sum_i \mathrm{GWP}_i \mathrm{X}_i \ , \tag{1.1}$$

where X_i is the mole fraction and i is a greenhouse species. With this index, mine ventilation air constitutes 4.2% of Australia's greenhouse gas emissions [6].

The challenge presented is how to process the CH_4 content of such a source in order to reduce its GWP. One method to achieve this would be via combustion. According to Table 1.2, the reaction

$$\mathrm{CH_4} + 2\mathrm{O_2} + 7.52\mathrm{N_2} \rightleftharpoons \mathrm{CO_2} + 2\mathrm{H_2O} + 7.52\mathrm{N_2}$$

reduces the GWP of a stoichiometric methane-air mixture by a factor of 3.7. The goal therefore was to reduce the environmental impact of this ventilation air by conversion of the entrained CH_4 to CO_2. This thesis documents the design and analysis of a device to perform this conversion.

1.2 Reducing the GWP of ventilation air by combustion

At the atmospheric state, the extinction limit for a flame is 5.26±0.3% at zero strain-rate [10]. The concentration of methane in ventilation air is well below this limit. However, as the pre-combustion temperature of the mixture increases, the flame extinction limit goes to zero as the combustion enthalpy required to sustain the flame goes to zero. Above this temperature, combustion occurs homogeneously instead of propagating via a flame. This type of combustion (known as oxidation or volumetric combustion) only occurs in very lean methane mixtures and is purely a chemical process. To burn CH_4 therefore, the ventilation air will need to be raised to its *combustion temperature* until reactions have gone to completion.

Coal mine ventilation air is not a pure methane-air mixture. However, hydrocarbon impurities are known to accelerate reactions [11, 12], hence modelling ventilation gas as a methane-air mixture is conservative when considering the combustion delay. For the purposes of modelling, ventilation air was simplified by assuming that

1. the mixture is homogeneous (such that turbulent combustion can be ignored)
2. the concentration of methane in ventilation air is quasi-steady at 0.5% (corresponding to a calorific value of 140kJ.kg_{mix}^{-1}) and
3. the methane concentration is below the flame extinction limit (allowing flame ignition and propagation to be ignored).

With this definition ventilation air has a GWP of 0.0185 which reduces to 0.005 if all the CH_4 is oxidized.

1.3 The potential for utilisation by conventional methods

Two conventional methods to utilise the methane from ventilation air are concentration and storage or supplementation of the ventilation air with additional fuel for use in a power plant. The former approach was found to be uneconomical for methods such as absorption, adsorption and permeability due to the large volumes of ventilation air which need to be processed [6]. For the latter approach, utilisation of ventilation air methane is of secondary importance to the supplementary fuel and, as such, this approach is subject to the efficiency and cost of the power generation method. Conventional technologies for electricity production from methane cost about $0.06 per kW.hr (excluding the cost of storage) which is only economical during peak periods [6]. BHP's Appin and Tower Colliery uses this approach and consumes 15-25% of the mine's ventilation air along with drainage methane in ninety-four 1MW reciprocating engines.

The use of coarse reject coal instead of drainage gas as a supplementary fuel has also been investigated. A pilot plant that used a rotating kiln combustor which exchanged heat with an

indirect-fired turbine was trialled at CSIRO QCAT, Pinjarra Hills in 2004 but combustion of the fuels could not be sustained [13].

Thus, it was decided that an emission control device should be pursued instead.

1.4 Candidate emission control devices

During selection of a device, both the thermal efficiency and the operating temperature was considered. For the device to oxidise ventilation air in a self-sustaining way, losses had to be equal to or less than (on average) the heat released due to combustion. The thermal efficiency of a heat engine is

$$\eta_{\text{th}} = \frac{w_{\text{out}}}{h_c} \tag{1.2}$$

where w_{out} is the work performed and h_c is the enthalpy of combustion. For the candidate device, it is enough to have a thermal efficiency of zero so long as the methane is oxidised.

Oxidation of ventilation air is performed by raising the temperature of the ventilation air to its combustion temperature. For this application, methods of piston compression and heat exchange were investigated. Turbomachinery was considered unsuitable since, without a supplementary fuel, volumetric compression ratios of at least 37 are required for oxidation of ventilation air which are not attainable with current axial-flow compressors. A general comparison of the chosen methods is made in Table 1.3.

combustion via heat exchange	**combustion via piston compression**
work extracted from excess heat by steam turbine or thermoelectric generator [14]	work extracted from piston by an electromagnetic generator, shaft-work or hydraulics [15]
const-p combustion at low pressure	const-ν combustion at high density
heat and viscous losses	heat and frictional losses
higher combustion temperature	lower combustion temperature
5-10% efficiency	up to 56% efficiency

Table 1.3: *Burner type comparison*

A general comparison these devices follows.

Flow-reversal reactors comprise two connected volumes of inert ceramic material as either a pelletized-bed or monolithic heat exchange medium. Flow between the volumes is continuously reversed such that heat is exchanged to and from the ventilation air as it is heated and burned. Pelletized-bed reactors exhibit a high pressure drop compared during heat exchange. Krzysztof *et al.* [16] compared thermal and catalytic combustion of lean methane-air mixtures in both pelletized-bed and monolithic flow-reversal reactors. Peak temperatures of about 1473K and 1073K were required for thermal and catalytic combustion respectively. For the thermal combustion in a monolithic reactor, sustained operation was achieved for a methane mole fraction of 1.83%. Catalytic flow-reversal reactors have

achieved sustained operation at methane mole fractions of 0.22% and slightly lower temperatures [17]. However, the lifespan of the catalysts are reduced by the high temperatures required for homogeneous combustion and the impurities in ventilation air methane. A reliable cost analysis is required before either catalytic or thermal reactors of this type may be considered suitable.

Counterflow heat exchangers recycle heat directly to the incoming fuel-air mixture such that it is brought to the combustion temperature. The flow rate for these devices is typically limited by the flame speed, however, this limit disappears for air preheated to the adiabatic flame temperature [18]. Experiments performed on double-spiral (also known as "Swiss roll") burners show that temperature is virtually constant over a range of flow velocities and extinction does not occur until velocities are two orders of magnitude in excess of normal burning velocities [18]. Unlike the pelletized-bed heat exchangers, these heat exchangers do not suffer from high pressure loss. Pumping losses were always below 10% of the heat released, even with mixtures leaner than 1% by volume [18]. It is envisaged that a double-spiral burner could be started using a stoichiometric mix of methane and air until the heat contained in the device was sufficient for the reaction of ventilation air to be self-sustaining. It is assumed that the operating temperature of these devices is that of the adiabatic flame temperature of a stoichiometric methane-air mixture at atmospheric pressure, that is, 2226 K [7]. It is suggested that thermoelectric generators (which operate at efficiencies between 5-10%) could possibly be used to extract electrical energy from the exhaust stream. It should be noted however that no example of power generation using this method has been found and that they may negate the performance of such a device by impeding the heat exchange process.

Compression ignition engines have potentially the highest efficiencies of all engine types due to their high volumetric compression ratios. A particular diesel engine idles at a mole fraction of about 3%, (17.5% of the stoichiometric fuel-air ratio) and has an indicated efficiency of 56% [19, page 142]. This particular engine has a 13" bore, a volumetric compression ratio of 15 and a mean piston speed of 6.29m.s^{-1}. Typical volumetric compression ratios for automotive two-stroke diesel engines are 17 to 19 with a piston speed of 9.14m.s^{-1} [19, page 461] whereas large marine and stationary engines have lower ratios (14 to 16) and slower piston speeds (7.11m.s^{-1}). Free-piston engines have the potential to operate on even lower concentrations due to the absence of mechanical linkages and sidewall friction, and because the double-acting nature of the piston effectively makes it a "single-stroke" engine.

1.5 Comparison of candidate devices

The adiabatic flame temperature of a stoichiometric CH_4-air mixture at atmospheric pressure is 2226 K [7]. However, the reaction rate of a gas mixture is a function of species concentration and temperature (see §3.5). Thus, the combustion temperature of a compression device is lower

than for a heat exchanger due to the reactions occurring at a higher concentration. Beyond this observation, a potential device must be investigated before the combustion temperature can be known.

Since N_2O has a GWP index of 180, if significant quantities of N_2O were produced during oxidation of ventilation air then the effectiveness of the device would be reduced. Nitrous oxides only form at high temperatures so combustion should be performed at a lower temperature if possible. For a compression ignition device, the combustion temperature of ventilation air is not high enough for N_2O to be produced in significant quantities (see §4.4). For a constant-pressure fixed-mass perfectly-mixed reactor however, significant quantities of N_2O begin to develop for reaction conditions of 2226 K and 1 atm (see §3.6.1). Thus, the primary reason for investigating a free-piston engine over a double-spiral heat exchanger is its lower combustion temperature.

The operation of an engine on ventilation air hinges on the accuracy of the loss models which include exhausting, friction and heat transfer. It is difficult to know *a priori* which of these losses dominate and thus where to concentrate effort. Take, for example, friction. The work over a stroke is dependent on the stroke length, compression ratio, compression ring thickness and friction coefficient. All these variables are unknown. A similar situation is seen for heat transfer and exhausting. Thus, the level of modelling was chosen to be as accurate and scalable for a range of engine geometries (particularly with respect to the heat transfer model) as feasible. In order of importance, the performance of an engine of this type was found to be limited by

1. the exhausting efficiency,
2. heat transfer and
3. compression ring friction.

An energy balance was performed post-simulation to determine the fraction of energy lost by each process and to verify the complete model. The efficacy of this device is determined by these processes.

1.5.1 A potential free-piston engine for emission control

As mentioned in the preface, this project developed from an experimental project on a free-piston engine into a purely numerical study. The engine under investigation comprised two cylinders separated by a free-piston (see Figure 1.2). Due to the lack of a crankshaft the piston trajectory is unlike that of a conventional engine and requires a linear electric motor for both control and extraction of work. Oscillation of the piston along the axis of the cylinders drives the thermodynamic cycles in both combustors (see Figure 1.3).

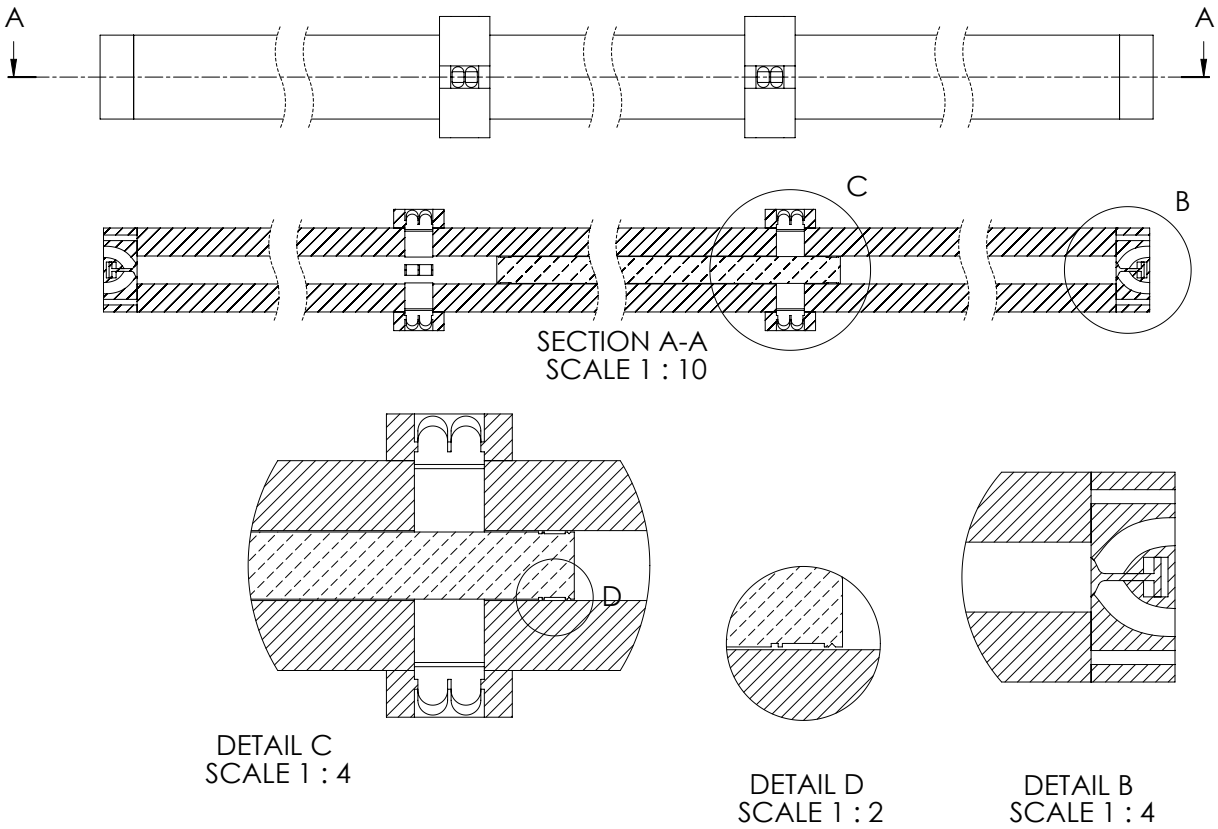


Figure 1.2: *Compression-ignition free-piston engine. This engine design is similar in geometry to those investigated in this thesis. The long piston and cylinder lengths the cylinder were a consequence of exhausting requirements. Details will be discussed more completely in later sections of this document.*

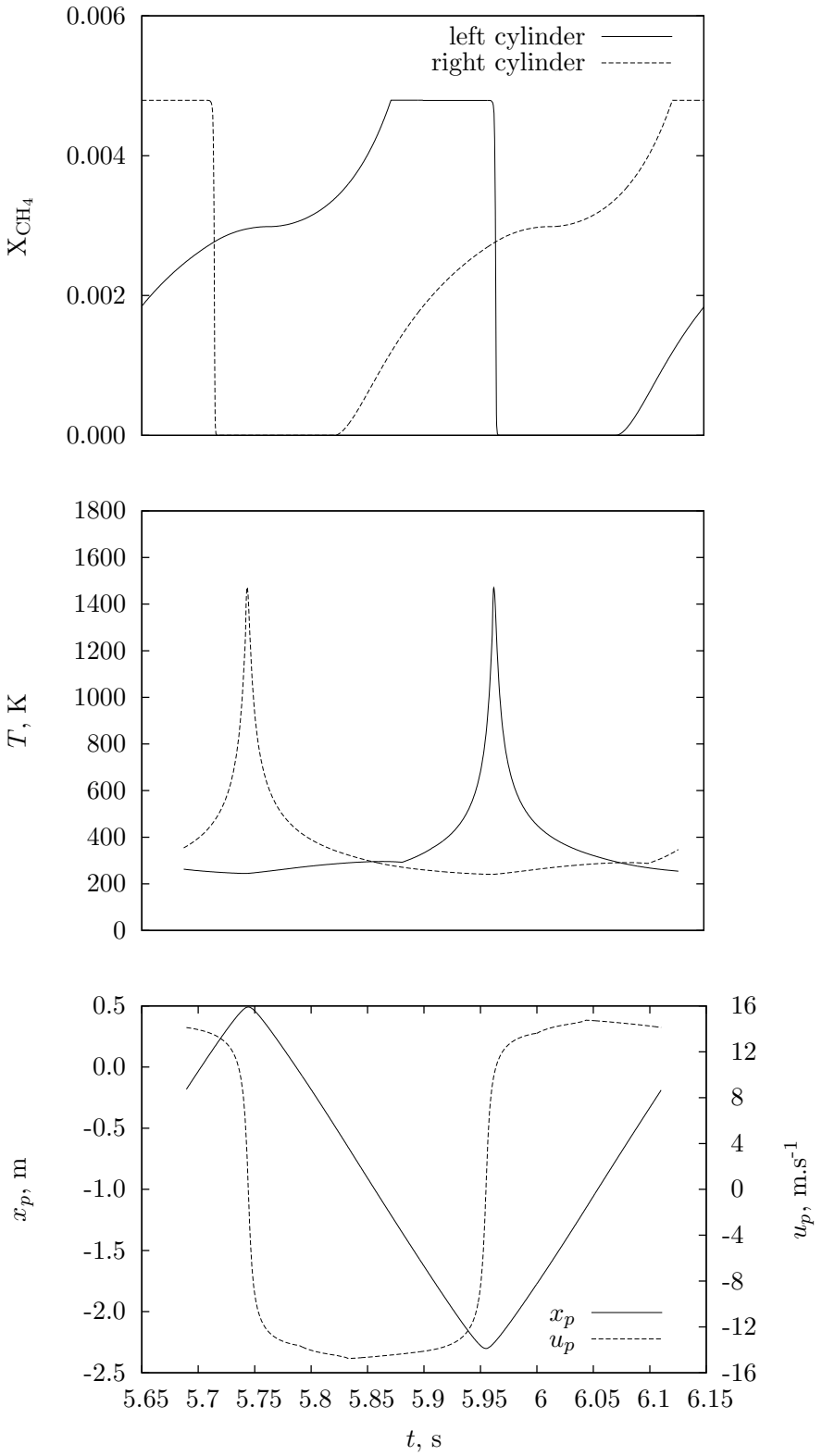


Figure 1.3: *Piston trajectory, gas temperature and methane concentration during one period of a free-piston engine.*

This engine has inherent mechanisms that increases its efficiency above conventional compression ignition engines and might allow it to operate at very low concentrations of methane. These are:

- high compression ratios afforded by the low combustion enthalpy of ventilation air which improves the Otto cycle efficiency
- the short ignition delay of methane, allowing the engine to approach constant volume combustion
- a crankshaft-less design which results in lower sidewall friction, lower heat loss (particularly at the point of piston reversal) and a variable compression ratio
- the power output of a double-acting engine without the sealing problems associated with the connecting-rod (or conrod) [20].

From numerical experiments performed in later chapters, the combustion temperature required by ventilation air in a compression device was found to be about 1330 K (depending on the residence time). This requires volumetric compression ratios of about 37.

Needless to say, this engine design brings technical challenges. Such an engine must operate at its natural frequency [15] which is dependent on the gas properties, the piston mass, the piston surface area and the cylinder length[1]. It requires specialised methods for exhausting (due to the lack of an exhausting stroke as in a four-stroke engine, or a high pressure crankcase as in a traditional two-stroke engine), extraction or addition of work (due to the lack of a crankshaft) and sealing (due to high pressures). It is envisaged that the piston will carry a permanent magnetic field and be driven by one or more solenoids located on the surface of the cylinder. Electrical work would be extracted from the piston by the same mechanism.

1.6 Previous studies of free-piston engines

The free-piston engine design of Figure 1.2 is referred to as a "dual-piston" type due to the opposed piston-cylinder arrangement. It has a coupled intake and compression process, that is, whilst one cylinder is undergoing compression the other is being exhausted (see Figure 1.3). A review of free-piston engine concepts [15] compared single-piston, dual-piston and opposed-piston free-piston engine types. Work was extracted from these engines using hydraulic fluid and control was performed using pause-pulse modulation of the hydraulic fluid flow. The paper advocated that a single-piston engine (which consisted of a single cylinder) was the only type for which control problems had been solved since the presence of a conrod allowed for contact with the hydraulic fluid. However, power electronics have progressed since the time this paper was written and the conclusions reached may no longer apply. Nevertheless, the biggest challenge for the dual-piston engine type remains the control of the volumetric compression ratio.

[1] More correctly, the engine operates at a forced frequency since a force is imparted by the linear electric motor.

Goldsborough and van Blarigan [21] performed a numerical investigation into the thermal efficiency and exhaust emissions of a free-piston engine that utilised homogeneous combustion of hydrogen. A zero-dimensional, thermodynamic model with finite-rate chemistry, empirical scavenging, Woschni's heat transfer model [22] and mixed friction component models were used. They used the model to optimize the thermal efficiency of the engine by varying the fuel-air ratio. Control of the compression ratio and the exhausting processes were found to be critical factors affecting the engine's performance.

Mikalsen and Roskilly [23] designed a compression ignition free-piston engine generator for electric power generation. The engine was of dual piston type, but had a bounce chamber in place of the second combustion cylinder. A single zone combustion model was used along with an ignition delay formula. The quasi-steady heat transfer model of Hohenberg [24] was used along with an assumed mean effective pressure of 120kPa to model friction. The exhausting process was modelled using polytropic expansion of the gas along with a perfect displacement model. The exhausting pressures were modelled using the external turbocharger-supercharger model of [20].

In these studies, fuel mole fraction is a variable quantity and focus is placed on maximizing efficiency or power output of the engine. Because of this, accuracy in the modelling of losses such as heat transfer and friction is not of great concern. Engine heat transfer models in particular still only capture (at best) the average heat transfer. There is substantial literature investigating these component processes. A review of this literature is performed in the subsequent chapters.

1.7 Scope and contribution of this thesis

Unlike previous studies, the purpose of this computational model is to determine the minimum amount of fuel required to sustain operation of this engine. As such, any substantial error in the sub-models directly reflects in the conclusions drawn. Thus, every effort was made to validate the models of friction and heat transfer and to select an appropriate kinetic mechanism for the very lean combustion regime such that, when combined, the resultant model was a close approximation of an actual engine.

Engine modelling can combine many effects including a mixed friction coefficient for the compression ring, turbulent heat transfer, turbulent mixing, finite-rate chemistry, real gas modelling and control. This high level of modelling introduces high levels of uncertainty. The engine model was simplified by limiting the modelling to the following physical processes:

1. finite-rate chemistry
2. piston-gas dynamics
3. unsteady, bulk fluid flow
4. mixed friction and
5. unsteady heat transfer.

The principal contribution of this thesis is the complete modelling of a free-piston engine in the very lean burn regime and its subsequent application to the task of emission control. This went as far as to investigate the effect of real gas models on combustion and, in the case of the heat transfer, resulted in a new transient integral boundary layer model that correctly captured the heat flux inversion about the point of piston reversal (see §6.2.4 on page 99) as well as a novel PID control system.

The investigation was split into three parts:

Part I quantifies the available chemical energy of ventilation air by selection of appropriate thermodynamic and chemical models. This comprised an investigation of gas models in Chapter 2 and the selection of a kinetic mechanism in Chapter 3. Because of the high computational cost of the finite-rate chemistry, various reduced schemes were explored and evaluated for their accuracy in approximating the ignition delay and heat release of ventilation air.

Part II investigates the component models of the engine cycle. The piston-gas dynamics are covered in Chapter 4, cylinder exhausting in Chapter 5, heat transfer and friction models are developed in Chapter 6 and the engine control system is developed in Chapter 7.

Part III brings all the modelling pieces together into a full engine model (Chapter 8) which is used to conduct a parametric study. Based on these results, conclusions were drawn and the question of whether the proposed engine was a suitable device with which to reduce the global warming potential of coal mine ventilation air was answered.

Part I

Thermodynamics and chemistry of ventilation air

Chapter 2

Gas models

Real gases include phenomena such as the change in the number of particles constituting the gas (finite-rate chemistry), change in the number of degrees-of-freedom in the motion of the particles (multiple energy modes) and intermolecular forces due to the proximity between particles (dense gas effects). Real gas models are important for predicting the performance of hypersonic wind tunnels [25] where it has been found the projectile velocity is reduced due to the dominance of intermolecular forces. They are also important for use in shock tube studies [26], particularly those used in the elucidation of reaction rate coefficients.

More recently, it has been suggested that real gas phenomena have an appreciable effect under the conditions seen in modern diesel engines [27]. If the ideal gas model begins to incorrectly predict the state, there will be a corresponding effect on the chemical reaction rates[1]. As was found during this investigation, the high volumetric compression ratios required by this engine do indeed approach the limits of accuracy of the ideal equation of state. Thus, along with the ideal and thermally-perfect gas models, cubic equations of state and real thermal behaviour are investigated here. These models more accurately capture the gas state at high densities and provide better values for use by the finite-rate chemistry module. This was done to quantify the effect of these models on engine operation.

Dense gas effects are governed by the equation of state of the gas model. Real thermal effects are evaluated by the thermal model, using the state provided by the gas model. This is the programming methodology used for the gas models. In this case, the existing framework for the gas models was available, and the Abel-Noble and van der Waals gas models were added. They are included as Listing A.1 on page 140 and A.3 on page 145 respectively. The real thermal behaviour model also built on existing code and is included in Listing A.5 on page 151.

2.1 Gas state

While its motion is subsonic, the gas contained in a piston-compression device is of variable density. The state, $\mathbf{Q}$, of an ideal (that is, thermally and calorifically perfect) gas is completely specified by two calorific values, two dependent state variables and velocity. Using the set

[1] Density (or more precisely, species concentration) affects reaction rates in a polynomial manner whereas temperature affects reaction rates in an exponential manner. These effects will be covered in Section 3.2.1.

$\{R, \gamma, T, p, u\}$, the atmospheric state for dry air may be written

$$\mathbf{Q}_{\text{atm}} = \{R, \gamma, 298.15, 101325, 0\} \tag{2.1}$$

where R is the gas constant and $\gamma = c_p/c_\nu$ is the isentropic exponent. This state is used as a reference state for the evaluation of thermodynamic properties. The gas constant may be calculated

$$R = \frac{\mathscr{R}}{M} \tag{2.2}$$

where M is the molecular mass of the gas mixture.

The dependent state variables are related by the equation of state discussed in §2.1.1. The calorific values of a gas are dependent on this state and may be evaluated using a number of models in §2.1.2. The equations in the following sections apply to a gas mixture (unless otherwise specified), so mixing rules are covered in §2.1.3.

2.1.1 Evaluation of state

There are three types of equation of state, namely virial, analytical and empirical. Of the analytical type, two cubic models (namely, Abel-Noble and van der Waals models) along with the ideal model were selected for study. It is important to note that the cubic models apply only to gases, that is, of a single phase, and again become inaccurate as the gas state approaches the critical point.

The dependent state variables comprise $\{p, T, \nu, \rho\}$. Using the set $\{p, T, \nu\}$, the state equation for an ideal gas may be written

$$p = \frac{RT}{\nu} \tag{2.3}$$

where the gas constant is specific to the mixture.

The state of an Abel-Noble gas (also known as a covolume gas) requires an additional constant, namely the covolume, ν_0, which is the specific volume taken up by the molecules of a gas. Using $\{p, T, \nu, \nu_0\}$, the state equation is

$$p = \frac{RT}{\nu - \nu_0} \qquad \text{for} \quad \nu > \nu_0. \tag{2.4}$$

This model deviates from ideal behaviour as the specific volume of a gas approaches its covolume. The gas becomes incompressible when $\nu = \nu_0$.

The state of a van der Waals gas requires a further constant, α, which is a measure of the intermolecular force

$$p = \frac{RT}{\nu - \nu_0} - \frac{\alpha}{\nu^2} \qquad \text{for} \quad \nu > \nu_0. \tag{2.5}$$

Values of ν_0 and α are given for selected species in Table 2.1, see §2.A.1 on page 19. The Abel-Noble and van der Waals equations are derived from the general form of the cubic equation of state (included in §2.A.1 on page 19).

For validation of the numerical implementation of these equations, Figure 2.1 compares the isothermal compression of Helium with the data of Akin [28]. The thermally perfect gas

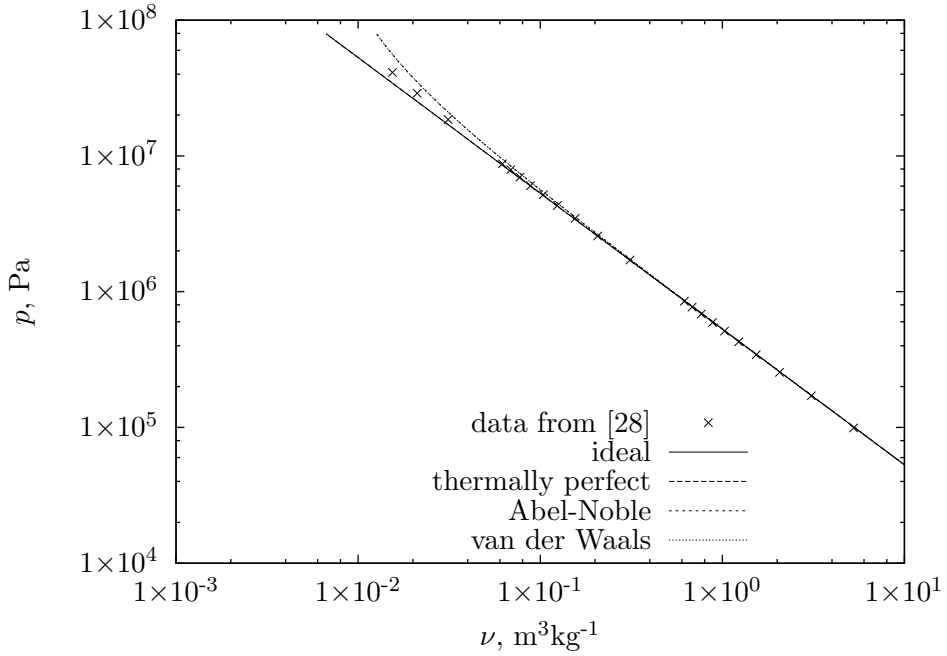


Figure 2.1: *Validation of the gas model implementation using isothermal compression of Helium with reinterpreted data from [28]. The Abel-Noble and van der Waals gas models deviate from the ideal gas model.*

model shows no deviation from ideal behaviour for isothermal compression, whereas both the Abel-Noble and van der Waals gases slightly overestimate pressure.

The effect of these models in a free-piston compressor is covered in §4.2.

2.1.2 Evaluation of calorific values

For all gas models, the definition of enthalpy [29] is

$$h = e + p\nu. \tag{2.6}$$

For a calorifically-perfect gas, the calorific value c_p is assumed to be constant, such that $dh = c_p dT$. This assumption however is only valid over a small temperature range. For a thermally-perfect gas, the specific heat, absolute enthalpy and entropy are temperature dependent properties [29]. Polynomial fits are used and take either the form suggested in [30]

$$\frac{\bar{c}_{p,i}}{\mathscr{R}} = a_1 T^{-2} + a_2 T^{-1} + a_3 + a_4 T + a_5 T^2 + a_6 T^3 + a_7 T^4 \tag{2.7a}$$

$$\frac{\bar{h}_i^0}{\mathscr{R}T} = -a_1 T^{-2} + a_2 T^{-1} \ln T + a_3 + \frac{a_4}{2} T + \frac{a_5}{3} T^2 + \frac{a_6}{4} T^3 + \frac{a_7}{5} T^4 + \frac{a_8}{T} \tag{2.7b}$$

$$\frac{\bar{s}_i^0}{\mathscr{R}} = -\frac{a_1}{2} T^{-2} - a_2 T^{-1} + a_3 \ln T + a_4 T + \frac{a_5}{2} T^2 + \frac{a_6}{3} T^3 + \frac{a_7}{4} T^4 + a_9 \tag{2.7c}$$

or this form with the first two terms omitted as in [31]. Species coefficients were sourced primarily from the Chemkin thermodynamic database [31]. When species data was not available, data from CEA [30] and the JANAF tables [32] were used in that order.

For a thermally real gas, values of de, dh and ds are dependent on the gas state. The general forms of these properties are

$$de = c_\nu dT + \left(T \left.\frac{\partial p}{\partial T}\right|_\nu - p \right) d\nu, \tag{2.8a}$$

$$dh = c_p dT + \left(\nu - T \left.\frac{\partial \nu}{\partial T}\right|_p \right) dp \quad \text{and} \tag{2.8b}$$

$$ds = \frac{1}{T}\left(dh - \nu dp\right) \tag{2.8c}$$

respectively [29]. The pressure dependency of these relations reduces to zero for the ideal equation of state, which is said to be thermally perfect. As such, it is only necessary to use equation (2.8) with the Abel-Noble and van der Waals equations of state. Relevant derivatives for these models are supplied in §2.A.3 on page 19.

Substituting (2.8a) and (2.8b) into (2.6) yields

$$c_\nu = c_p + T \left.\frac{\partial \nu}{\partial T}\right|_p^2 \left.\frac{\partial p}{\partial \nu}\right|_T \tag{2.9}$$

which reduces to the well-known relation $c_v = (c_p - R)$ for an ideal gas.

2.1.3 Mixing rules

For real gas mixtures, Dalton's law of additive pressures has been used as an approximation. The covolume of a gas mixture is a weighted sum of the covolumes of the component species whereas α (in the simplest case) is a geometric sum, *viz.*

$$\bar{\nu}_0 = \sum_{i=1}^{n_{\text{sp}}} \mathrm{X}_i \bar{\nu}_{0,i} \quad \text{and} \tag{2.10a}$$

$$\bar{\alpha} = \left(\sum_{i=1}^{n_{\text{sp}}} \mathrm{X}_i \bar{\alpha}_i^{\frac{1}{2}} \right)^2, \tag{2.10b}$$

where the overbar denotes the property is per unit mole.

The calorific values for a gas mixture are a weighted sum of the calorific value of the component species. An example of this is the specific heat at constant pressure (2.7a),

$$\bar{c}_p = \sum_{i=1}^{n_{\text{sp}}} \left(\mathrm{X}_i \bar{c}_{p,i} \right). \tag{2.11}$$

The molecular mass may be used to convert between mole and mass specific values by

$$c_p = \frac{\bar{c}_p}{M} \tag{2.12}$$

where

$$M = \sum_{i=1}^{n_{\text{sp}}} \mathrm{X}_i M_i = \left(\sum_{i=1}^{n_{\text{sp}}} \frac{Y_i}{M_i} \right)^{-1}. \tag{2.13}$$

The viscosity of a gas mixture may be approximated [33] using

$$\frac{\sum_{i=1}^{n_{\text{sp}}} \left(\mathrm{X}_i \mu_i \sqrt{M_i}\right)}{\sum_{i=1}^{n_{\text{sp}}} \left(\mathrm{X}_i \sqrt{M_i}\right)}. \tag{2.14}$$

This property will be used later for calculating the Reynolds number for heat transfer (see §6.2.2).

2.2 Entropic relations

Entropy is a nonconserved property of a gas. From (2.8c), the entropy generated by an ideal gas undergoing an adiabatic process may be written

$$\left.\begin{aligned} ds &= c_\nu \tfrac{dT}{T} + R\tfrac{d\nu}{\nu} \\ s_2 - s_1 &= c_\nu \ln \tfrac{T_2}{T_1} + R \ln \tfrac{\nu_2}{\nu_1} \end{aligned}\right\} \geq 0. \tag{2.15}$$

For an ideal gas undergoing an isentropic process $ds_{0\to1} = 0$, (2.15) becomes

$$\frac{p_1}{p_0} = \left(\frac{\nu_0}{\nu_1}\right)^{\gamma} \tag{2.16a}$$

$$\frac{T_1}{T_0} = \left(\frac{\nu_0}{\nu_1}\right)^{\gamma-1} \quad . \tag{2.16b}$$

These functions will be used for free-piston compressor models in §4.1 and heat transfer models in §6.2.

Le Châtelier's principle states that when a gas at equilibrium is subjected to a change (for example, in pressure), it will shift in composition in such a way as to minimise the change (for example, by shifting in a direction that produces fewer moles). Gibb's free energy is used to describe the energy of such a gas and is defined (for all gas models [29]) as

$$g = h - Ts \tag{2.17}$$

where h and s may be calculated using (2.8b) and (2.8c) and the appropriate mixing rule. This function is used for the calculation of the equilibrium constant for finite-rate chemistry (see §3.A.6 on page 51).

2.A Chapter end notes

2.A.1 General cubic equation of state and parameters

The general form of the cubic equation of state is

$$p = \frac{\mathscr{R}T}{\bar{\nu} - \bar{\nu}_0} - \frac{\Theta\left(\bar{\nu} - \eta\right)}{\left(\bar{\nu} - \bar{\nu}_0\right)\left(\bar{\nu}^2 + \delta\bar{\nu} + \varepsilon\right)} \quad , \tag{2.18}$$

which reduces to the ideal gas law in the low density limit[2]. The values $\Theta, \eta, \varepsilon, \delta$ and $\bar{\nu}_0$ may be constants (including zero) or functions of temperature or composition depending on the gas model type [34]. For the Abel-Noble and van der Waals models, $\Theta = \alpha$, $\eta = \bar{\nu}_0$ and $\varepsilon = \delta = 0$ and the critical temperature and pressure of a gas $(T_\mathrm{c}, p_\mathrm{c})$ are used to evaluate $\bar{\nu}_0$ and $\bar{\alpha}$.

$$\bar{\nu}_0 = \frac{1}{8}\frac{\mathscr{R}T_c}{p_c} \tag{2.19a}$$

$$\bar{\alpha} = \frac{27}{64}\frac{\left(\mathscr{R}T_c\right)^2}{p_c} \tag{2.19b}$$

The values of $\bar{\nu}_0$ and $\bar{\alpha}$ are given for a selected number of species in Table 2.1.

species	$\bar{\nu}_0$, $\mathbf{m^3mol^{-1}}$	$\bar{\alpha}$, $\mathbf{m^3Jmol^{-2}}$	T_c, K	p_c, Pa
O_2	3.184×10^{-5}	0.1381	154.58	50.43×10^5
N_2	3.864×10^{-5}	0.1368	126.20	33.98×10^5
H_2O	3.051×10^{-5}	0.5542	647.14	220.64×10^5
CH_4	4.306×10^{-5}	0.2303	190.56	45.99×10^5
CO	3.951×10^{-5}	0.1473	132.85	34.94×10^5
CO_2	4.286×10^{-5}	0.3658	304.12	73.74×10^5
He	2.376×10^{-5}	0.00346	5.19	2.27×10^5

Table 2.1: *Species constants for a real-gas equation of state [35]*

2.A.2 General form for some gas variables

Real gas equations-of-state are written using specific-volume instead of density. To find the derivatives with respect to density, we can use the chain rule.

$$\begin{aligned}\frac{\partial}{\partial\rho} &= \frac{\partial}{\partial\nu}\frac{\partial\nu}{\partial\rho} \\ &= -\nu^2\frac{\partial}{\partial\nu}\end{aligned} \tag{2.20}$$

The derivatives for the three gas models considered are shown in Table 2.2.

2.A.3 Evaluation of temperature from internal energy and density in a real gas

There is a coupled relation between the specific heats, internal energy and temperature of a gas. For an Abel-Noble gas, enthalpy is evaluated the same as for a thermally perfect gas. For a

[2]Note the nomenclature here has been changed slightly as $\bar{\nu}_0$ is commonly designated b.

	Ideal	**Abel-Noble**	**van der Waals**
$\left.\frac{\partial T}{\partial p}\right\|_{\nu}$	$\frac{\nu}{R}$	$\frac{(\nu-\nu_0)}{R}$	$\frac{(\nu-\nu_0)}{R}$
$\left.\frac{\partial T}{\partial \nu}\right\|_{p}$	$\frac{p}{R}$	$\frac{p}{R}$	$\frac{1}{R}\left[p-\frac{\alpha}{\nu^3}\left(\nu+2\nu_0\right)\right]$
$\left.\frac{\partial p}{\partial \nu}\right\|_{T}$	$-\frac{RT}{\nu^2}$	$-\frac{RT}{(\nu-\nu_0)^2}$	$-\frac{RT}{(\nu-\nu_0)^2}-\frac{2\alpha}{\nu^3}$
$\left.\frac{\partial \nu}{\partial T}\right\|_{p}$	$\frac{R}{p}$	$\frac{R}{p}$	$R\left[p-\frac{\alpha}{\nu^3}\left(\nu+2\nu_0\right)\right]^{-1}$

Table 2.2: *Equation of state derivatives*

van der Waals gas however,

$$dh = c_p dT + \left\{\nu - RT\left[p + \frac{\alpha}{\nu^3}\left(2\nu_0 - \nu\right)\right]^{-1}\right\} dp. \tag{2.21a}$$

Once h is known, we may evaluate e by combining (2.6) with the state equations for an ideal, Abel-Noble and van der Waals gas,

$$e = h - RT \tag{2.22a}$$

$$e = h - \frac{RT\nu}{(\nu - \nu_0)} \tag{2.22b}$$

$$e = h - \frac{RT\nu}{(\nu - \nu_0)} + \frac{\alpha}{\nu} \tag{2.22c}$$

If e is known, the correct value for temperature may be found by iteration.

Chapter 3

Finite-rate chemistry

This chapter comprises two parts: the first is the selection of an appropriate kinetic mechanism for methane and the second is the investigation into the effect of real gas models on finite-rate chemistry.

Ventilation air burns homogeneously when raised to the adiabatic flame temperature since the methane concentration is practically at the flame extinction limit, even for a preheated mixture [14]. Mole fraction mixtures of around 1% are commonly chosen for use in ignition delay experiments conducted using reflected shock tubes (for example, [36] and [37]) because they remain almost isothermal during the reaction. For combustion in a free-piston compressor, the state of the gas at the point of piston reversal and the combustion it promotes is very similar to these post-shock conditions, so experimental data was used to select a mechanism.

Finite-rate chemistry involves the application of a global reaction scheme or a chemical kinetic mechanism to model the production and destruction of chemical species in a gas. Global reactions model a single reaction, do not include intermediary species (going straight from reagents to products) and hold only over a limited range of conditions. In contrast, kinetic mechanisms comprise many elementary reactions, including those with intermediary species. The latter was deemed necessary simply because intermediary species such as CO, OH and N_2O were of interest. In addition, reactions in a free-piston engine may not always go to completion (due to the reactions being frozen by the rebounding piston).

Many kinetic mechanisms for methane exist in the literature. To determine which mechanisms were applicable to the conditions in a free-piston engine, two selection criteria were applied: the first (and the one given most attention here) was *ignition delay*[1], the second was heat release. Both are important, and the kinetic mechanism was selected based on how accurately it predicted both of these phenomena. As it happened, more than one mechanism was found to be suitable so computational expense was included in the selection criteria.

3.1 Chemistry of methane

Methane is the simplest hydrocarbon, yet kinetic mechanisms that describe its combustion can prove computationally prohibitive when used for numerical modelling of engines. As such, they are commonly reduced (simplified) using a number of techniques [38]. Six methane mechanisms were selected from the literature for investigation (see Table 3.1).

[1] Ignition delay was defined as a particular point in the CH_4 concentration history during combustion

Name	Species	Reactions	Ref
Li and Williams	18	24	[39]
Smooke and Giovangigli	16	25	[40]
Jazbec *et al.*	16	28	[41]
DRM19	21	84	[42]
DRM22	24	104	[43]
GRI-Mech3.0	53	325	[44]

Table 3.1: *Investigated Methane mechanisms*[2]*. The DRM19 mechanism is included as Listing 3.4 on page 48.*

Of these six, only GRI-Mech3.0 is a "full" mechanism. The remaining five are so-called "reduced" mechanisms which sacrifice accuracy in either ignition delay (such as the flame mechanism of Smooke and Giovangigli) or heat release (such as the ignition mechanisms of Li and Williams) in order to reduce the computational work. Reduced mechanisms may also exclude reactions that do not produce appreciable amounts of products over their intended range of temperatures and densities (for example, nitrogen chemistry for low temperature mechanisms as in DRM19 and DRM22).

3.1.1 Methane combustion

The equivalence ratio for methane is

$$\phi = \frac{2\mathrm{X}_{\mathrm{CH_4}}}{\mathrm{X}_{\mathrm{O_2}}} \ . \tag{3.1}$$

For air, N_2 may be treated as the diluent (also known as the bath gas). For a stoichiometric CH_4-air mixture (that is, for $\phi = 1.0$) the lower and higher[3] heating values are 76.271kJ.mol^{-1} and 84.677kJ.mol^{-1} of mixture respectively. By comparison, the nominal mole fraction of methane in ventilation air is 0.005 [6] which corresponds to heating values of 4.011kJ/mol and 4.454kJ/mol of mixture respectively.

For methane in air, the reaction

$$\mathrm{CH_4} + 2\mathrm{O_2} + 7.52\mathrm{N_2} \rightleftharpoons \mathrm{CO_2} + 2\mathrm{H_2O} + 7.52\mathrm{N_2},$$

was modelled using a kinetic mechanism.

Kinetic mechanisms are comprised of many elementary reactions and include intermediary species. *Elementary reaction rates* use integer exponents (equal to the stoichiometric mole numbers). The elementary reactions are either *bimolecular*, *third body* or *pressure-dependent* depending on their behaviour and number of participating species. These terms refer to the rate coefficients used in the calculation of the elementary reaction rates. All rate coefficients

[2]During a chemical reaction, the evaluation of specific heats (and hence temperature) is coupled to the reaction-rate (which is very temperature sensitive). Thus, it is important to use the same thermodynamic data when comparing two mechanisms, see §2.1.2.

[3]The higher heating value includes the latent heat of fusion of water, which is included for experimental reasons.

are strongly temperature dependent, however pressure-dependent coefficients are additionally dependent on the third body concentration. The investigated methane mechanisms contained all three types with the exception of Jazbec *et al.*, which excluded the pressure-dependent coefficients.

The decomposition of the methane molecule is one possible first step in the chain branching, and the reaction

$$\mathrm{CH_4} \rightleftharpoons \mathrm{CH_3} + \mathrm{H} \tag{3.2}$$

will receive special attention as it is known to have pressure-dependent rate coefficients.

3.2 Reaction modelling

The gas model and finite-rate chemistry module `Gaspy` was developed in-house[4] largely by Rowan Gollan [45]. The capability of multiple energy modes was added due to the work of Daniel Potter. Due to the work in this document, `Gaspy` now also includes real gas models and pressure-dependent reactions. This section validates the implementation of pressure-dependent rate coefficients in `Gaspy`. The associated program code is included as Listing A.7 on page 159.

3.2.1 Elementary reactions

From Anderson [46] the syntax of an elementary reaction is

$$\sum_{i=i'} c_i i \underset{k_r}{\overset{k_f}{\rightleftharpoons}} \sum_{i=i''} c_i i \tag{3.3}$$

where c is the stoichiometric coefficient, i' and i'' denote the reagents and products respectively, and k_f and k_r are the forward and reverse reaction rate coefficients respectively. The reaction rate is a function of the rate coefficient and reagent concentrations, and the rate coefficient is a function of temperature.

3.2.2 Species production rates

The production rate of a species is a function of temperature and reagent molar concentrations. The total production rate of species i is the sum of its production rate over all elementary reactions in a mechanism,

$$\frac{d[\mathrm{X}_i]}{dt} = \sum_{j=1}^{n} \left(\frac{d[\mathrm{X}_i]}{dt} \right)_j \quad , \tag{3.4}$$

where n is the total number of reactions.

Reaction rates can refer to either the rate of production or destruction of a species participating in a reaction. The net rate of change in the concentration of species i is

$$\frac{d[\mathrm{X}_i]}{dt} = (c_{i''} - c_{i'}) \left\{ k_f \prod_{i=i'} [\mathrm{X}_i]^{c_i} - k_r \prod_{i=i''} [\mathrm{X}_i]^{c_i} \right\} \qquad \text{for } i', i'' \in i \tag{3.5}$$

where k_f and k_r are the forward and reverse rates and $[\mathrm{X}_i]$ is the concentration of species i.

[4] Dept. of Mechanical Engineering, University of Queensland

Elementary rate coefficients

Rate coefficients are classed as either Arrhenius or Pressure-dependent.

Arrhenius form The Arrhenius form of the rate coefficient is based on collision theory and applicable to the majority of reactions.

$$k\left(T\right) = AT^{\beta} \exp\left(-\frac{E}{RT}\right) \tag{3.6}$$

where the frequency factor, AT^{β}, and the activation energy, E, are determined experimentally. The general units for these variables are mole(vol.time)$^{-1}$ and energy.mole^{-1} respectively.

In a third body reaction, different species effect a reaction at different efficiencies, η_i. The *third body value* is defined as

$$[\mathrm{M}] = \sum_{i=1}^{n_{\mathrm{sp}}} \eta_i [\mathrm{X}_i] \tag{3.7}$$

where X_i is the concentration of species i and n_{sp} is the total number of species. This is distinct from the *third body concentration* of a gas which is equivalent to the molar density.

Pressure-dependent form Pressure-dependent coefficients transition between two limiting Arrhenius coefficients depending on the third body value. This transition is due to the fact their reactions exhibit both unimolecular and bimolecular (also known as first and second order) characteristics. The rate coefficient tends to a constant value, k_∞, as the molar density goes to infinity. At high temperatures, essentially all unimolecular reactions exhibit this behaviour [47].

Pressure-dependent reactions are denoted using parentheses around the third body, as in

$$\mathrm{A}\left(+\mathrm{M}\right) \underset{k_r}{\overset{k_f}{\rightleftharpoons}} \mathrm{B} + \mathrm{C}\left(+\mathrm{M}\right) \ .$$

This signifies that the low pressure reaction involves two participants, whereas the high pressure reaction involves only one. Care must be taken when evaluating the units of the rate coefficient for this reaction. Pressure-dependent rate coefficients commonly take one of two forms: Lindemann-Hinshelwood or Troe.

The Lindemann-Hinshelwood form of the forward rate coefficient is given by

$$k_f = \frac{k_\infty k_0 [\mathrm{M}]}{k_\infty + k_0 [\mathrm{M}]} = \begin{cases} k_0[\mathrm{M}] & \lim_{[\mathrm{M}]\to 0} \\ k_\infty & \lim_{[\mathrm{M}]\to \infty} \end{cases} , \tag{3.8}$$

where the rate coefficients k_0 and k_∞ are provided by the kinetic mechanism in Arrhenius form. The derivation of this rate can be found in §3.A.7 on page 51. The reverse rate coefficient for these reactions may be calculated using the equilibrium constant (c.f. §3.A.6 on page 51) or specified explicitly, as in the mechanism of Smooke and Giovangigli (see §3.A.9 on page 53).

The Troe form is based on the Lindemann-Hinshelwood form but includes strong and weak collision broadening effects in one broadening factor, F [47].

$$k_f = \frac{k_\infty k_0[\mathrm{M}]}{k_\infty + k_0[\mathrm{M}]} F \tag{3.9}$$

F may be calculated using

$$\log F = \left[1 + \left(\frac{\log P_r + c}{n - d\,(\log P_r + c)}\right)^2\right]^{-1} \log F_{cent} \quad , \tag{3.10}$$

where

$$P_r = \frac{k_0[\mathrm{M}]}{k_\infty}$$
$$c = -0.4 - 0.67 \log F_{cent}$$
$$n = 0.75 - 1.27 \log F_{cent}$$
$$d = 0.14 \quad \text{and}$$
$$F_{cent} = (1-a)\exp\left(-\frac{T}{T^{***}}\right) + a\exp\left(-\frac{T}{T^{*}}\right) + \exp\left(-\frac{T^{**}}{T}\right) \quad .$$

The four parameters a, T^{***}, T^{*} and T^{**} must be specified as input for the Troe form.

3.2.3 Experimental methods for elucidating reaction models

To a large extent, rate coefficients for these models have been derived from combustion experiments. Experimental devices used include RCMs, reflected shock-tubes and so-called static- and flow-systems. Initially, rapid-compression devices were used to elucidate the rate coefficients (see [48, 49]). However, a number of shortcomings, including long *work delay* (relative to the chemical delay) and spatial non-uniformity of gas temperature within the compressed cylinder (especially at higher pressures and temperatures) led to the use of reflected shock tubes.

Reflected shock tubes use a shock compression process (generated by a rapidly expanding reservoir of high-pressure gas) to create ignition conditions in a test gas. When compared to rapid-compression devices, reflected shock tubes more completely isolate combustion from other phenomena and allow more control over the conditions at which combustion occurs. While remaining the best device, they are not without limitation. During combustion, the energy liberated by the reactions has a compounding effect on the reaction which makes their rates difficult to quantify. Isothermal conditions are also required to keep shock waves as one-dimensional as possible [50]. For methane oxidation, these conditions are only obtained if the CH_4 content of the mixture remains below 1% [36], although a correction may be used to account for shock attenuation and non-ideal reflection (for example, 20 K [51] or -35 K [52]).

A set of test cases were selected to validate the finite-rate chemistry of `Gaspy`, and particularly the pressure-dependent rate coefficients. Pressure-dependent rate coefficient values were plotted alongside their low- and high-pressure limiting coefficient values, and equilibrium mole fractions were validated by comparison to those from CEA. A constant-volume fixed-mass reactor was used to compare the ignition delay of the mechanisms in Table 3.1 with shock-tube tests in order to prove their applicability.

3.2.4 Methane decomposition

Kinetic mechanisms are models in their entirety, and elementary reactions are not accurate in isolation. An exception to this is when an experimental study of a single species is conducted to determine the rate coefficient. Take for example the decomposition of CH_4, which is known to have pressure-dependent rate coefficients. To find the rate coefficients, Hartig *et al.* [53] performed reflected shock tube experiments using pure CH_4. Experiments were performed at conditions between 1850 to 2500 K and total densities between 5.5×10^{-5} and 1.5×10^{-3}mol.cm^{-3}. Over these conditions they found the rate constant for methane decomposition to vary between first and second-order. The high and low limiting rate coefficients were found to be

$$k_\infty = 1 \times 10^{15} \exp\left(-104000/RT\right) \quad \text{and} \tag{3.11a}$$

$$k_0 = 1 \times 10^{17.8} \exp\left(-88000/RT\right) \tag{3.11b}$$

respectively (in units of cm, cal, K and mol).

Kondratiev [49] determined the rate coefficient for methane decomposition in a rapid-compression device. He overcame the shortcomings of the device noted earlier by combining the reaction rate formula with the piston dynamics. The experimental conditions ranged between 1590 to 1750 K at a pressure of 20 atm over which the rate constant was found to be

$$k = 1 \times 10^{15} \exp\left(-103000/RT\right) \tag{3.12}$$

which is remarkably similar to the high pressure rate coefficient of [53]. Importantly, he found that the thermal dissociation of methane under these conditions was seen to obey a strictly first-order law. This means that under compression ignition conditions, the high pressure rate coefficient of methane is reached before the reaction begins.

While the focus here is on CH_4 decomposition, there are many pressure-dependent rate coefficients in the full mechanism of GRI-Mech3.0. Take, for example, the rate coefficient of reaction 184

$$N_2O(+M) \rightleftharpoons N_2 + O(+M) \tag{3.13}$$

which is also pressure-dependent (confirmed in [26]). Nitrogen chemistry will be investigated in the §3.4.

3.2.5 Validation of pressure-dependent rate coefficients

Take reaction 51 from the GRI-Mech3.0 mechanism (see Table 3.2). As a validation test, the `Gaspy` module was called upon to return the rate coefficients for this reaction over a range of pressures (and hence third body values) to compare them to the limiting Arrhenius rates (see Figure 3.1). The Troe form includes some offset from the low pressure Arrhenius coefficient. While the decomposition methane is a pressure-dependent reaction, the high pressure limiting rate, k_∞, is the active rate during compression ignition.

Verification of equilibrium mole fractions

Equilibrium mole fractions were verified in the low and high pressure limit for the methane decomposition reaction of four mechanisms. When the equilibrium constant is used to calculate

Reaction	A (mol,cm,s,K)	β	E (cal.mol^{-1}K^{-1})
$H + CH_3(+M) \rightleftharpoons CH_4(+M)$	1.390×10^{16}	-5.34×10^{-1}	5.360×10^{2}
low	2.620×10^{33}	-4.76×10^{0}	2.440×10^{3}
Troe coefficients (a, T^{***}, T^{*}, T^{**})	0.7830, 74.00, 2941.00, 6964.00		
efficiencies	CH_4=3.00, Ar=0.70		

Table 3.2: *Reaction 51, GRI-Mech3.0*

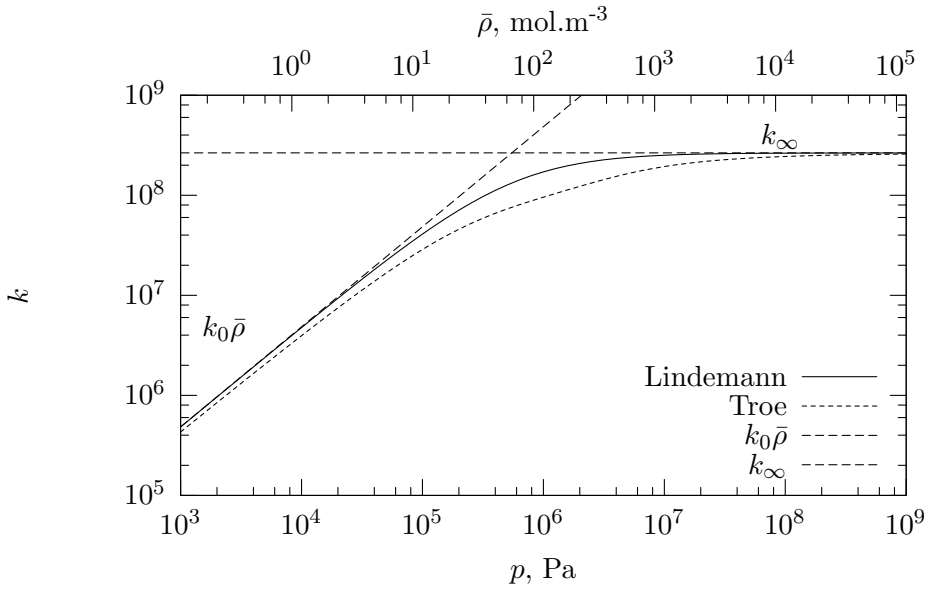
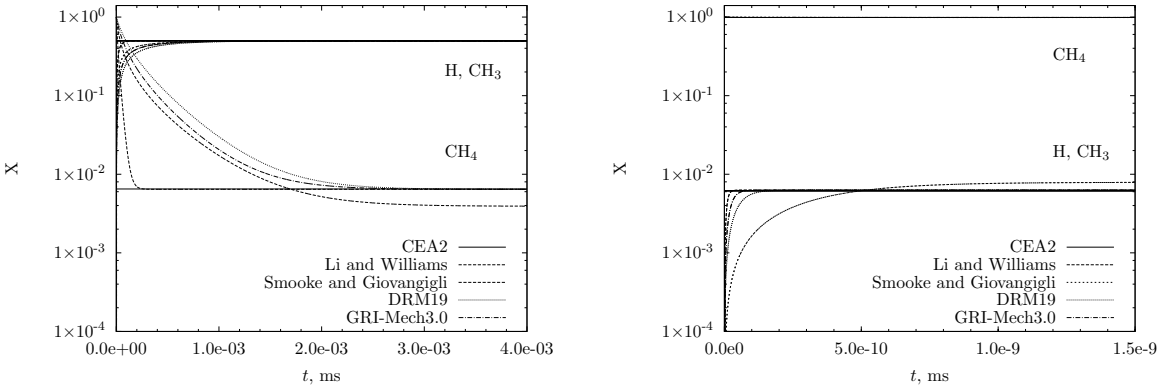


Figure 3.1: *Pressure-dependent rate coefficient and the Arrhenius asymptotes (Reaction 51, GRI-Mech3.0). Both the Lindemann-Hinshelwood and Troe forms of Reaction 51 were compared to show the effect of the Troe coefficients. Due to limitations in the code (as of revision 672), efficiencies could not be incorporated at this level. This feature will be added in future revisions.*

the backwards rate coefficient (as in (3.26)), the species' concentrations should always approach their equilibrium value, differing only by the rate at which this occurs.

Species production rates were integrated at constant temperature and constant pressure conditions using `Gaspy`. Equilibrium values were then compared to the output of CEA2 [54] (see Figure 3.2). These reactions were only used to verify the equilibrium condition since elementary reaction rates are not valid in isolation. The species concentrations predicted by the Smooke and Giovangigli reaction do not approach their equilibrium value because a backward rate was specified for this elementary reaction.

At this point, both the rate coefficient and the equilibrium mole fraction of products for pressure-dependent reactions have been verified. Thus it is appropriate to continue with the numerical modelling of reacting systems.



(a) $1\times10^3 Pa$ **(b)** $1\times10^9 Pa$

Figure 3.2: *Decomposition of CH_4 (const-T, p) showing equilibrium mole fractions. The species concentrations predicted by the Smooke and Giovangigli reaction do not approach their equilibrium value because a backward rate was specified for this elementary reaction.*

3.3 Programming methodology

The numerical models developed in this thesis used and extended existing code from the Dept. of Mechanical Engineering, University of Queensland. They were written in a number of computer languages: C++ was used to do the numerical work, Python was used to set up test case parameters and call the C++ functions (wrapped using SWIG) and finally Lua was used for the input files (including the kinetic mechanisms).

Code verification is the process of comparing the output of a program with a known result to show there are no mistakes. *Code validation* is the process of comparing the output of a program to experimental data to check the code is accurate. Both methods were used to reduce numerical error.

Sources of numerical error may be classified into five types [55]:

1. the degree to which a mathematical model approximates a physical process
2. the degree to which a numerical model approximates the mathematical model (for instance, the use of operator-split method to approximate coupled equations)
3. the error in computation of the discrete physical properties of the numerical model (such as floating point error and discretisation)
4. the ability of the grid (if applicable) to resolve the computational domain through the size and shape of individual elements (for discretisation) and
5. the inaccuracy in the interpolation of a discrete solution.

The extent of the first type of error is determined by comparison to numerical data, where possible. The extent of the second type is determined by comparison to the analytical solution.

The extent of the third type depends on the qualities of the time-marching integration scheme, any interpolation methods and machine precision. Machine precision error inherent to floating-point operations (including truncation and round-off error) depends largely on good coding practices. The extent of the fourth and fifth sources of error depends largely on the qualities of the timestep selection.

Timesteps

Chemical reactions occur very rapidly compared to changes in the thermodynamic state and the piston dynamics of a free-piston engine. In addition, it is desired that the state of the system only be recorded every so often. As such, four different timesteps were used during simulations:

dt_chem time between evaluating the species production rates

dt_therm time between evaluating the thermodynamic state

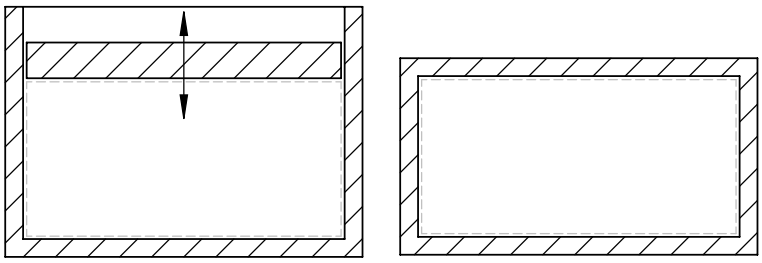
dt_sys time between evaluating piston dynamics

dt_print time between writing system state to file.

Timesteps were selected for each operation to minimise computational work whilst maintaining an acceptable level of error. Each timestep could include substeps of dt_therm and dt_chem. The state of the system was printed to file at intervals of dt_sys. When no chemistry is evaluated, dt_therm and dt_sys were equal.

3.4 Perfectly mixed reactors

For code verification of the pressure-dependent finite-rate chemistry, both constant-pressure and constant-volume fixed-mass reactors were used.



(a) *constant-pressure, fixed-mass reactor* **(b)** *constant-volume, fixed-mass reactor*

Figure 3.3: *Perfectly mixed reactors (homogeneous and adiabatic)*

A rapid-compression device is equivalent to the constant-pressure, fixed-mass perfectly mixed reactor (see Figure 3.3a). The process is not strictly at a constant pressure, but can be approx-

imated as such using an operator-split process. The differential equations for this reactor are

$$\begin{aligned} \frac{dT}{dt} &= \frac{-\sum_i^{n_{\text{sp}}} \left(\bar{h}_i \dot{\omega}_i\right)}{\sum_i^{n_{\text{sp}}} \left([\text{X}_i]\bar{c}_{p,i}\right)} \quad \text{and} \\ \frac{dY_i}{dt} &= \frac{\dot{\omega}_i M_i}{\rho}, \quad i = 1, \ldots, n_{\text{sp}} \end{aligned} \tag{3.14}$$

where n_{sp} is the number of species, $\bar{h}_i$ is the specific enthalpy, $\dot{\omega}_i$ is the rate of production of species i, $[\text{X}_i]$ is the concentration of species i, and $\bar{c}_{p,i}$ is the specific heat at constant pressure, Y_i is the mass fraction of species i and M_i is the molecular weight of species i.

A shock tube device can be modelled using a constant-volume, fixed-mass reactor (see Figure 3.3b). A shock tube is not strictly of fixed mass (since one boundary is a moving shock wave), yet this is a fair assumption given that the chemical timescale is much smaller than the bulk flow timescale of the nearly stagnant gas in the shock-reflection region of the shock tube. The differential equations for this reactor are

$$\begin{aligned} \frac{dT}{dt} &= \frac{-\sum_i^{n_{\text{sp}}} \left(\bar{e}_i \dot{\omega}_i\right)}{\sum_i^{n_{\text{sp}}} \left([\text{X}_i]\bar{c}_{\nu,i}\right)}, \\ \frac{dp}{dt} &= \mathscr{R} T \sum_{i=1}^{n_{\text{sp}}} \dot{\omega}_i + \mathscr{R} \frac{dT}{dt} \sum_{i=1}^{n_{\text{sp}}} [\text{X}_i] \quad \text{and} \\ \frac{dY_i}{dt} &= \frac{\dot{\omega}_i M_i}{\rho}, \quad i = 1, \ldots, n_{\text{sp}} \end{aligned} \tag{3.15}$$

where $\bar{e}_i$ is the internal energy and $\bar{c}_{\nu,i}$ is the specific heat at constant density. These equations are coupled by the term $\frac{dT}{dt}$.

The equations of (3.15) and (3.14) were implemented in the program `reactor.py` (see Listing 3.1 on page 44). The constant-pressure, fixed-mass equations (3.14) were then replaced by an operator-split process (see `simple_reactor.py`, Listing 3.2 on page 46). It consisted of three operations over each global timestep:

1. integrating the chemistry using a number of substeps (which changes internal and chemical energy)

2. expanding the gas isentropically such that it returned to its original pressure

3. evaluating the state (assuming correct properties for density and internal energy).

To verify this process, the difference between the output of `reactor.py`, `simple_reactor.py` and Chemkin-II for a constant-pressure fixed-mass reactor using a hydrogen mechanism was compared for a range of timesteps (see Figure 3.4). The gas state and chemical rates were evaluated only once each timestep, but the species concentrations were updated over multiple substeps. Upon inspection of Figure 3.4, it can be seen that the assumption of a small change in gas state (and hence chemical rates) is violated as the timestep becomes large.

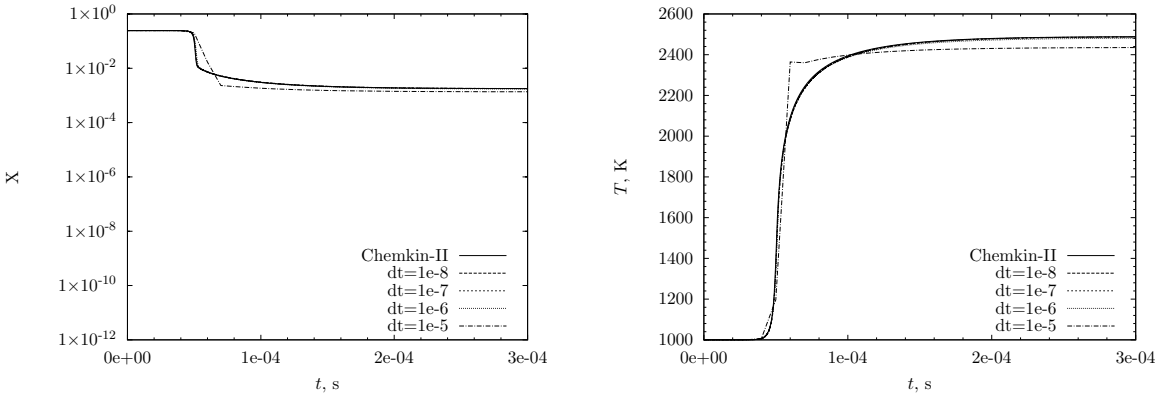


Figure 3.4: *Effect of timestep on the operator-split method. Effects are seen as the timestep becomes larger than the period between ignition delay and combustion delay.*

Constant-pressure, fixed-mass, perfectly mixed reactor

A constant-pressure fixed-mass perfectly mixed reactor using Chemkin-II is provided as an example in Appendix 2 of the Chemkin II manual [56]. This was used to verify both the chemistry of `Gaspy` using both `reactor.py` and `simple_reactor.py`. These programs integrate the constant-pressure fixed-mass equations (Case 2 of §3.4) in time, using a stiff ODE solver. In the latter case, the ODE solver of SciPy v0.6.0-r4 was used.

The comparison was made using two mechanisms so as to separate testing of the different reaction types. They were:

1. a hydrogen mechanism (Listing 3.3) to verify the two- and three-body rate implementations and

2. DRM19 (Listing 3.4 on page 48) to verify the pressure-dependent rate implementation.

Verification of two- and three-body rates

To verify the implementation of bimolecular and third-body rates, a comparison between `Gaspy` and Chemkin-II was performed using a hydrogen mechanism (Listing 3.3 on page 47) in a constant-pressure fixed-mass reactor. A summary of these tests is provided in the Table 3.3.

script	**mechanism**	**initial state**	n_{H_2},n_{O_2},n_{N_2}
`conp.f` [56]	`h2-chemkin.dat`	1000K,	1,3,0.1
`simple_reactor.py` (Listing 3.1)	`h2-chemkin.lua`	1×10^5Pa	

Table 3.3: *Hydrogen test condition*

The result in Figure 3.5 shows that an operator-split method using `Gaspy` and a dt_chem of 1×10^{-7}s achieves the same result as Chemkin-II. The mole fraction values were also checked the output in [56].

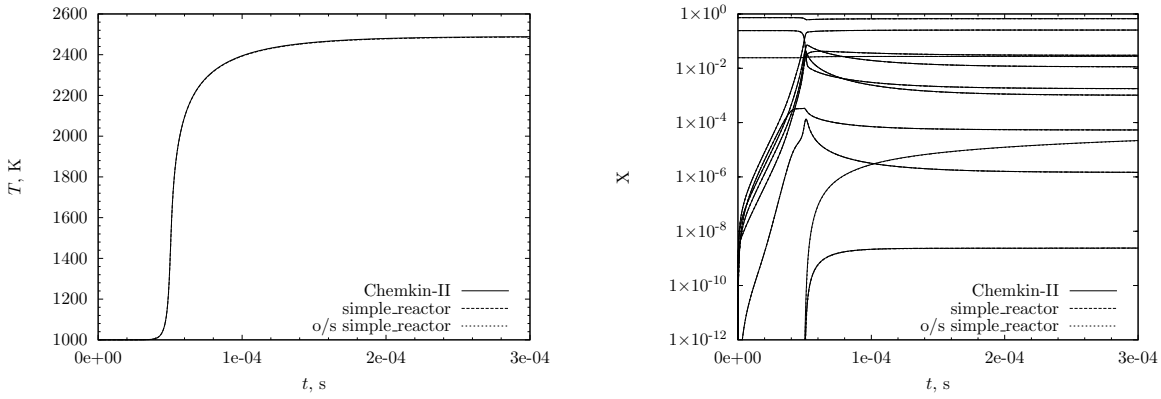


Figure 3.5: *Constant-pressure reaction using Hydrogen mechanism, dt_chem = 1×10⁻⁷. This is the example provided in Appendix 2 of the Chemkin II manual.*

To verify the implementation of pressure-dependent rates, a comparison between `Gaspy` and Chemkin-II was performed using the DRM19 mechanism (Listing 3.4) in a constant-pressure fixed-mass reactor. A summary of these tests is provided in the Table 3.4.

script	**mechanism**	**initial state**	$n_{\mathrm{CH_4}}$,$n_{\mathrm{O_2}}$,$n_{\mathrm{N_2}}$
`conp.f` [56]	`drm19.dat`	2000K,	1,2,7.52
`simple_reactor.py` (Listing 3.1)	`drm19.lua`	1×10^5Pa	

Table 3.4: *Methane test condition*

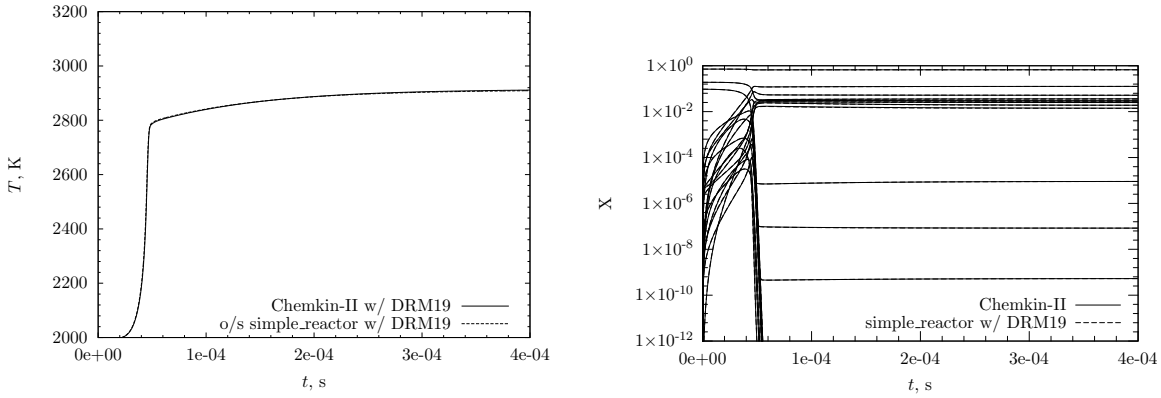


Figure 3.6: *CPFM PMR using DRM19 with mixture $n_{CH_4}, n_{O_2}, n_{N_2}$ = 1,2,7.52, dt_chem=2×10⁻⁷. On each of these, Chemkin-II matches well with* `Gaspy`*, even down to mole fractions of 1×10⁻¹⁰.*

3.5 Methane chemical delays

As is evident in (3.4), the rate of change in concentration of a species is proportional to its concentration. The *chemical delay* is defined as the time required for the concentration of a reagent, S, to fall to a specified fraction of its initial concentration [7]

$$\left.\begin{aligned}\tau_{\text{chem}} &= \frac{1}{k_f \prod_{i=i'} [\mathrm{X}_i]_0{}^{c_i}} \\ &= \frac{1}{A \exp\left(-\frac{E}{RT_0}\right) \prod_{i=i'} [\mathrm{X}_i]_0{}^{c_i}}\end{aligned}\right\} \quad i' \neq S \tag{3.16}$$

where $[\mathrm{X}_i]_0$ is the initial concentration of species i and T_0 is the initial temperature. Note this form is only correct when one species is in far greater abundance than the other (see [7, page 133]).

The *ignition delay* (also known as the induction time) of a reaction is a chemical delay defined by a characteristic point in the history of an indicative intermediary species. Being a special type of chemical delay, it has the same form as (3.16). For methane, experimenters commonly choose the peak concentration of OH, but the concentration histories of many other species have been measured along with pressure and temperature. For reflected shock tube experiments the concentrations used in (3.16) refer to the conditions behind the reflected shock. Experiments are conducted in order to obtain an ignition delay formula that characterizes a particular fuel.

Consider a homogeneous, combustible, CH_4-air mixture in an adiabatic chamber of a fixed volume at a given reference state. The physical and chemical state of the mixture will change with time until the dead state, or equilibrium, is reached. The *combustion delay* is defined here as the time between the reference state and the dead state.

Ignition and combustion delay formulae are closely related to global reactions and, as such, cannot be used to elucidate a kinetic mechanism [51]. Instead, ignition delay data will be used to validate and aid in selection of a suitable kinetic mechanism. The chosen mechanism will then be used in a free-piston engine model in §4.4.

3.5.1 Review of ignition delay experiments

The ignition delay of methane has been given different definitions depending on the measured property. The chosen definition has a large effect on the ignition delay constants and fair comparisons between papers can only be made when the same definition has been used. In most tests, ignition delay was defined as the delay between shock reflection and the onset of a sharp change in the emission (or absorption) history of an indicative species. Onset is defined as the intersection between the asymptote of the emission or absorption curve and a baseline value. The species selected differ between methane oxidation studies but, in general, the spike in emission of or absorption by the OH radical is preferred.

Seery and Bowman [52] and Lifshitz *et al.* [51] defined ignition delay as the delay between shock heating and the sudden rise in pressure due to combustion. Additionally, Seery and Bowman [52] measured species emission (OH, CH, C_2, CO) and absorption (OH) but decided

that chemiluminescence was a poor criterion for ignition delay due to the sensitivity of their apparatus.

In more recent experiments, with highly diluted CH_4-O_2 mixtures, chemiluminesence is used almost exclusively. Tsuboi and Wagner [36] used CH_4 emission (3.43μm) (although the onset of OH absorption (3100Å), CH_3 absorption (2160Å) and pressure increase were said to agree well with this indicator). Zallen and Wittig [57] used the first point of inflection of OH emission, while Grillo and Slack [58] used the delay between shock heating and the first rapid emission of OH (although this point also matched the onset of CO_2 and H_2O emission and pressure rise). Krishnan and Ravikumar [50] were less explicit, defining ignition delay as the time between shock heating the appearance of visible light. Petersen *et al.* [37] used OH absorption (306nm), CH_4 emission (3.4μm) and pressure measurements.

A complete list of definitions as quoted from selected papers is included in §3.A.5 on page 50. The ignition delay formulae selected for use were chosen based on their test conditions.

Test conditions

When considering the chemical delay equation (3.16), species concentrations, temperature and total molar density are the required variables. However, almost all ignition delay experiments provide initial conditions in the form of an equivalence ratio, diluent fraction of either Argon, Nitrogen or Helium, and total pressure.

When comparing experiments, total molar density is a more meaningful property than pressure [51] but not as easily measured. When not stated explicitly by the authors, the molar density was derived using the ideal gas law

$$\bar{\rho} = \frac{p_0}{\mathscr{R} T_0} \ . \tag{3.17}$$

Similarly, individual concentrations of CH_4 and O_2 are more meaningful properties than an equivalence ratio. Using (3.1), the species mole fractions could also be derived.

$$\mathrm{X}_{\mathrm{O}_2} = \frac{1 - \mathrm{X}_d}{1 + \phi/2} \tag{3.18a}$$

$$\mathrm{X}_{\mathrm{CH}_4} = \frac{\phi}{2} \mathrm{X}_{\mathrm{O}_2} \tag{3.18b}$$

Equations (3.18) and (3.17) were used to convert the reinterpreted data of Krishnan and Ravikumar. Unfortunately, not all papers provide sufficient data for even these conversions to be performed. A summary of some ignition studies (using reflected shock-tubes) is given in Tables 3.5 and 3.6, using units of species mole fraction and concentration respectively. The approach of Tsuboi and Wagner was particularly refined, as the mole fractions were kept constant and the temperature was varied at different total densities to achieve the test ranges.

In the literature, the general equation for ignition delay is given in the form

$$\tau_{\mathrm{ig}} = A \exp\left(\frac{E}{RT}\right) [\mathrm{CH}_4]^{c_{\mathrm{CH}_4}} [\mathrm{O}_2]^{c_{\mathrm{O}_2}} [\mathrm{Ar}]^{c_{\mathrm{Ar}}} \tag{3.19}$$

where c_i is the reaction order of species i and E is apparent activation energy. The coefficients for the mixtures in Table 3.6 are given in Table 3.7.

X_{CH_4}	X_{O_2}	X_{N_2}	X_{Ar}	**Ref**
2.000×10^{-2}–3.330×10^{-1}	1.960×10^{-1}–1.330×10^{-1}		7.840×10^{-1}–5.340×10^{-1}	[52]
1.000×10^{-2}–6.700×10^{-2}	2.000×10^{-2}–6.700×10^{-2}		9.700×10^{-1}–8.660×10^{-1}	[51]
2.000×10^{-3}	2.000×10^{-2}		9.780×10^{-1}–9.780×10^{-1}	[36]
1.690×10^{-2}–2.000×10^{-2}	1.690×10^{-2}–4.000×10^{-2}	1.520×10^{-1}–0.0	8.142×10^{-1}–9.400×10^{-1}	[58]
2.500×10^{-3}–5.000×10^{-2}	1.000×10^{-2}–5.000×10^{-2}	0.0–9.746×10^{-1}	9.875×10^{-1}–8.990×10^{-1}	[37]
4.545×10^{-3}–3.571×10^{-2}	4.545×10^{-2}–1.429×10^{-2}		9.500×10^{-1}–9.500×10^{-1}	[50]

Table 3.5: *Mole fractions used in ignition studies. Values have been derived when not stated explicitly. The range displayed here is based on maximum and minimum CH_4 mole fractions.*

$[CH_4]$, mol.cm^{-3}	$[O_2]$, mol.cm^{-3}	$[N_2]$, mol.cm^{-3}	$[Ar]$, mol.cm^{-3}	**Ref**
5.352×10^{-7}–9.886×10^{-6}	5.245×10^{-6}–3.949×10^{-6}		2.098×10^{-5}–1.585×10^{-5}	[52]
6.572×10^{-7}–5.641×10^{-6}	1.314×10^{-6}–5.641×10^{-6}		1.681×10^{-5}–7.292×10^{-5}	[51]
4.800×10^{-8}–3.600×10^{-6}	4.800×10^{-7}–3.600×10^{-5}		2.347×10^{-5}–1.760×10^{-3}	[36]
1.912×10^{-7}–7.127×10^{-7}	3.824×10^{-7}–7.127×10^{-7}	1.720×10^{-6}–0.0	9.020×10^{-6}–3.421×10^{-5}	[58]
3.140×10^{-7}–3.701×10^{-5}	6.392×10^{-7}–7.401×10^{-5}		5.512×10^{-5}–3.517×10^{-3}	[37]
8.440×10^{-8}–1.522×10^{-6}	8.440×10^{-7}–6.087×10^{-7}		1.764×10^{-5}–4.048×10^{-5}	[50]

Table 3.6: *Concentrations used in ignition studies. Values have been derived when not stated explicitly. The range displayed here is based on maximum and minimum $[CH_4]$.*

$\bar{\rho}$, mol.cm^{-3}	c_{CH_4}	c_{O_2}	c_{N_2}	c_{Ar}[5]	A, s(cm^{3}mol^{-1})$^{\sum_i c_i}$	$\frac{E}{R}$, K^{-1}	T, K	**Ref**
2.676×10^{-5} – 2.969×10^{-5}	0.40	-1.60		0	7.65×10^{-18}	25.87×10^{3}	1735 – 1646	[52]
1.878×10^{-5} – 8.420×10^{-5}	0.33	-1.03		0	3.62×10^{-14}	23.40×10^{3}	1655 – 1883	[51]
2.400×10^{-5} – 1.800×10^{-3}	0.32	-1.02		0, -0.5	4.00×10^{-15}	26.67×10^{3}	2068 – 1743	[36]
1.146×10^{-5} – 3.563×10^{-5}	0.33	-1.03	0	0	4.40×10^{-15}	26.32×10^{3}	2020 – 1710	[58]
5.607×10^{-5} – 3.628×10^{-3}	0.33	-1.05		0	4.05×10^{-15}	26.07×10^{3}	2043 – 1617	[37]
1.857×10^{-5} – 4.261×10^{-5}	0.33	-1.05		0	2.21×10^{-14}	22.65×10^{3}	1969 – 1716	[50]

Table 3.7: *A summary of ignition delay coefficients for the mixtures given in Table 3.6.*

From these tests it may be said in general that ignition delay for methane is strongly shortened by increasing the oxygen concentration and is slightly lengthened by increasing the methane concentration. This is encouraging when considering combustion of very low stoichiometries.

The apparent activation energy for the oxidation of methane increases with

- increasing CH_4 concentration
- increasing diluent concentration and
- decreasing O_2 concentration.

In the limit of decreasing O_2 concentration, the activation energy approaches that of the CH_3-H bond in low pressure experiments as methane oxidation simply becomes methane decomposition.

[5] For CH_4 concentrations below 5×10^{-8}mol.cm^{-3} and total densities below 5×10^{-5}mol.cm^{-3}. There is some debate as to whether the diluent affects the ignition delay. Spadaccini and Colket [11] provide a good discussion.

While all the coefficients in Table 3.7 are similar, we focus on experiments that include the conditions seen in a free-piston engine. These are low mole fraction (of the order of 0.005), low temperature ($\leq$1500 K) and high density experiments. Referring to Figure 3.7 and Table 3.5, these conditions are only matched by [36] and [37], although the former conducted tests over a wider range of total densities.

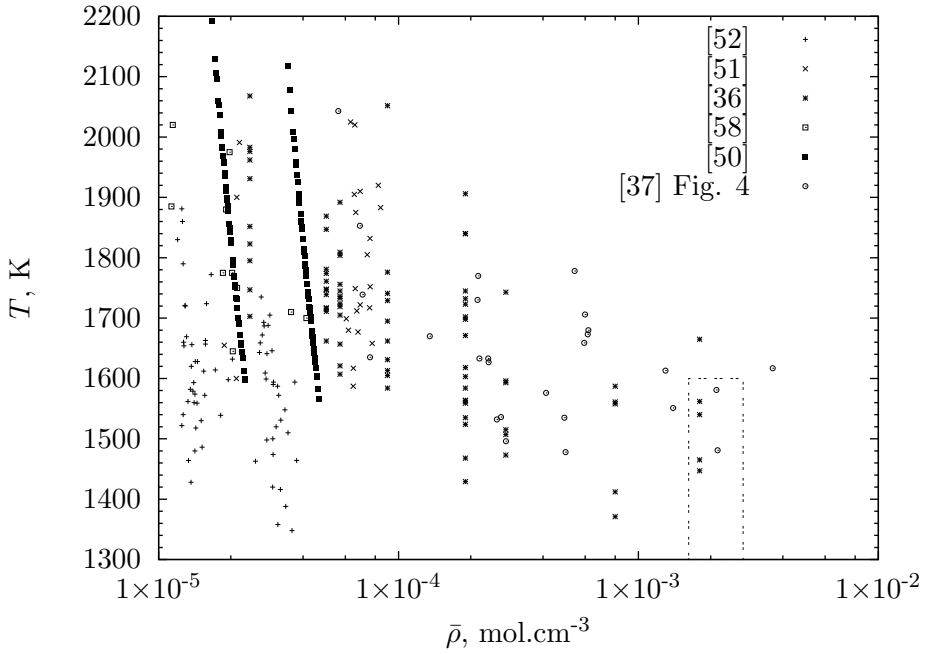


Figure 3.7: *Total density versus temperature for ignition delay studies. The boxed region shows conditions for isentropic compression of a thermally-perfect gas from the atmospheric state. Thus we will continue the investigation using the ignition delay data of [36] and [37].*

To verify data was correctly transcribed from these papers, the reduced induction times are shown in Figure 3.8 together with the ignition delay formula of [36]

$$\tau_{\text{ig}}[\text{CH}_4]^{-0.32}[\text{O}_2]^{1.02} = 4 \times 10^{-15} \exp\left(26.67 \times 10^3 T\right) \quad . \tag{3.20}$$

Petersen *et al.* found measured values of X_{OH} matched well with those predicted by the earlier GRI-Mech1.2. They used the peak in OH concentration to indicate ignition, even though a peak value was not always present (see results, Table 2 of [37]). When a peak in OH was present, it correlated well with the point at which X_{CH_4} goes to zero. Thus, in this thesis, the method used to get the ignition delay from numerical data was as follows:

1. A gas mixture was reacted under constant-volume, fixed-mass conditions until the point of inflection of pressure (midway through the reaction).

2. The last two $[\text{CH}_4]$ concentration values were then recorded and linearly extrapolated to zero.

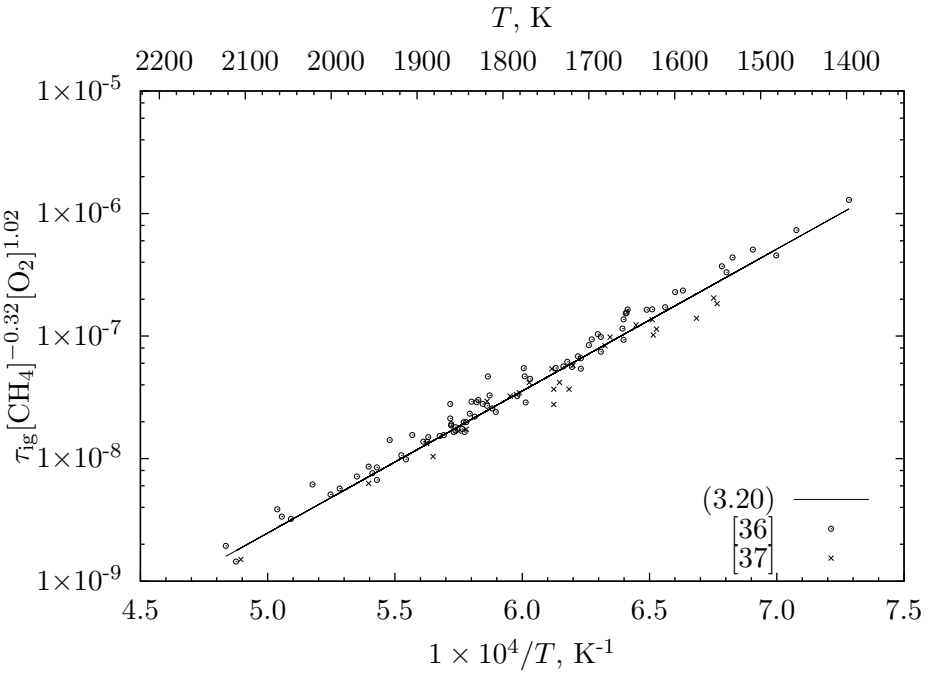


Figure 3.8: *Reduced induction times for transcribed data using the ignition delay formula of [36]. The data taken from [37] agrees well with this trend.*

3. The time at this point was recorded and taken to be the ignition delay.

The gradient of ignition delay (with respect to temperature) is a better indication of the accuracy of a mechanism than individual values. A comparison of the four mechanisms is shown in Figure 3.9. As can be seen, even the full mechanism underestimates measured ignition delay, although it did a better job than the ignition delay mechanism of Li and Williams (not shown). Somewhat fortuitously, DRM19 and DRM22 are closer to the correct values and trends. The Jazbec mechanism seems to predict the ignition delay at low density very well, but not as well at high density. This observation may be explained by the fact that this is a low pressure mechanism that does not include pressure-dependent rate coefficients for methane decomposition.

Methane oxidation pathway

Here a brief digression is made to look at the methane oxidation pathway. It is well known that methane oxidation takes a different pathway at high (2200 K) and low (<1500 K) temperatures (see page 166 of [7]). However, upon investigation of the literature, discrepancies were found with the ignition delay formula of different authors. In particular, it was proposed [57] (and later refuted [58]) that the third body had an effect on ignition delay. Unfortunately [57] does not include enough information for the results to be properly compared in Table 3.6. The former conducted experiments at total densities between 1.338×10^{-6} and 2.007×10^{-6}mol.cm^{-3} and CH_4 concentrations below 2.941×10^{-8}mol.cm^{-3}, derived assuming a constant temperature of 1870 K, whereas the latter conducted experiments at total densities between and 1.131×10^{-5} and 4.129×10^{-5}mol.cm^{-3} and CH_4 concentrations above 1×10^{-7}. Even though the densities given

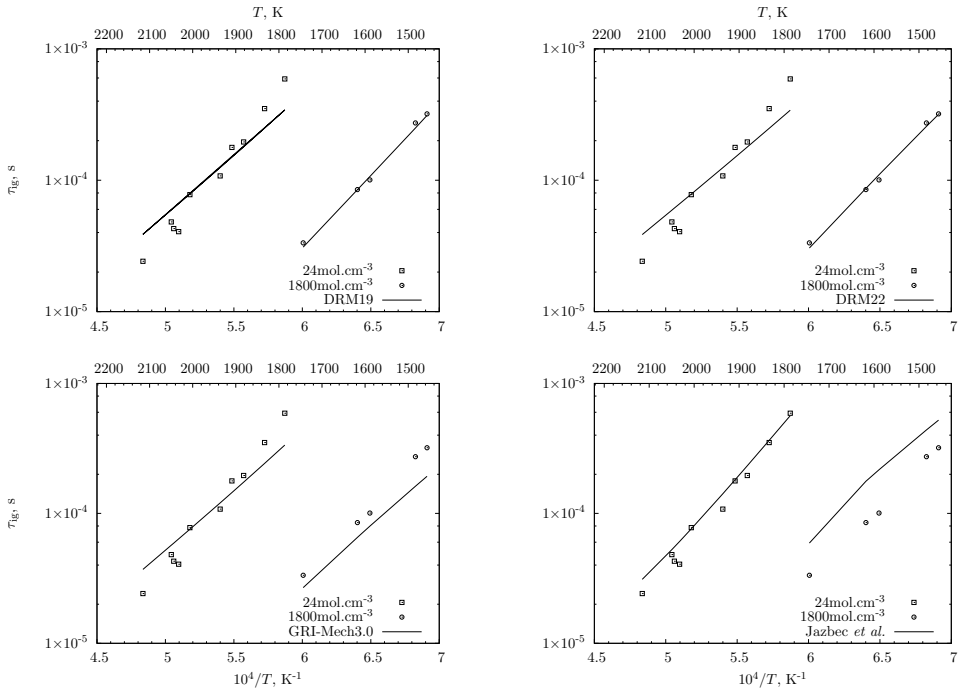


Figure 3.9: *Ignition delay of experimental results for the lowest and highest density tests by Tsuboi and Wagner compared with the numerical results over the same conditions using four kinetic mechanisms. The slope predicted by the kinetic mechanism is more important than the offset as the former is an indication of global activation energy whereas the latter is a consequence of the ignition delay definition. The mechanisms of Li and Williams and Smooke and Giovangigli were not included in the comparison. The former is solely an ignition mechanism and has non-smooth species production rates. The latter is a flame mechanism that does not attempt to capture ignition delay.*

here were derived (and so are approximate) the range of the former are an order of magnitude below those of the latter. Thus, it is unfair to say these results conflict.

Tsuboi and Wagner investigated ignition delay of methane for pressures from 3 to 300 atm at 1800 K. For total densities below 5×10^{-5}mol.cm^{-3} and CH_4 concentrations below 5×10^{-8}mol.cm^{-3}, the reaction order of the diluent tended towards $c_{\mathrm{Ar}} = -0.5 \pm 0.1$. For all total densities with CH_4 concentrations greater than 1%, the influence of the third body concentration disappeared, that is, $c_{\mathrm{Ar}} = 0.0 \pm 0.1$. If, at very low concentrations, the methane oxidation is limited by methane decomposition, the ignition delay would be somewhat dependent on the total molar density (see, for example, Figure 3.1). This would also explain the discrepancy found between Zallen and Wittig [57] and Grillo and Slack [58]. Thus, it is suggested in this thesis that the reaction pathway for the oxidation of methane changes not only according to temperature but also according to total density. Investigation of methane pathways at very low densities and concentrations is suggested as future work. Here ends the digression since, for a free-piston engine, only the low temperature, high density pathway need be considered.

3.6 Methane heat release

Along with correct ignition delay, the chosen kinetic mechanism must release the correct amount of heat. A fixed-mass perfectly mixed reactor test was performed at 2226 K and 1 atm using ventilation air and all six mechanisms. The final temperature is compared in Figure 3.10. All

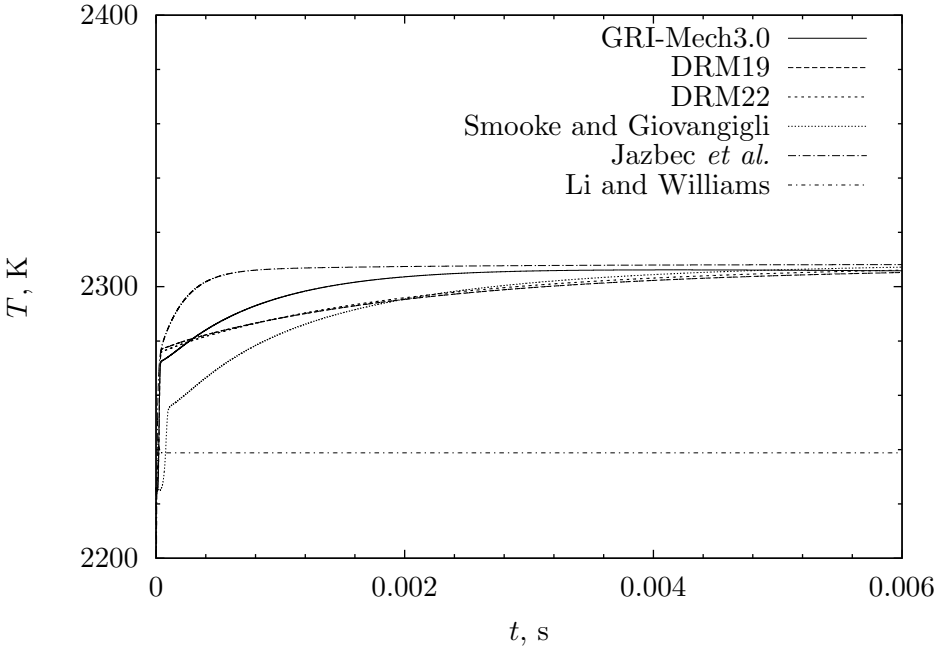


Figure 3.10: *A comparison of heat release for all six mechanisms. As can be seen, all but the ignition delay mechanism of Li and Williams result in equal heat release, albeit at different combustion delays. Tests were performed using a constant-pressure fixed-mass perfectly mixed reactor with ventilation air at initial conditions of T_0=2226 K and p=1 atm.*

mechanisms (except that of Li and Williams, which is an ignition mechanism and thus only predicts ignition delay) produce roughly the same amount of heat. The combustion delay at these conditions is about 6ms. Again, it can be seen from (3.6) that species production rates are polynomially dependent on concentration and at least exponentially dependent on temperature. Thus, the combustion delay in a rapid-compression device is shorter at the same temperature since species concentrations (or molar density) increase with compression. Alternatively it can be said that reactions occur at a lower temperature in rapid-compression devices.

3.6.1 The production of Nitrous Oxides

A perfectly mixed reactor of ventilation air at the stoichiometric adiabatic flame temperature (2226K) and atmospheric pressure produces the greenhouse species shown in Figure 3.11. The global warming potential during this reaction is shown in Figure 3.12. As can be seen, N_2O is produced in quantities of about 1ppm. This has a small but finite effect on the GWP.

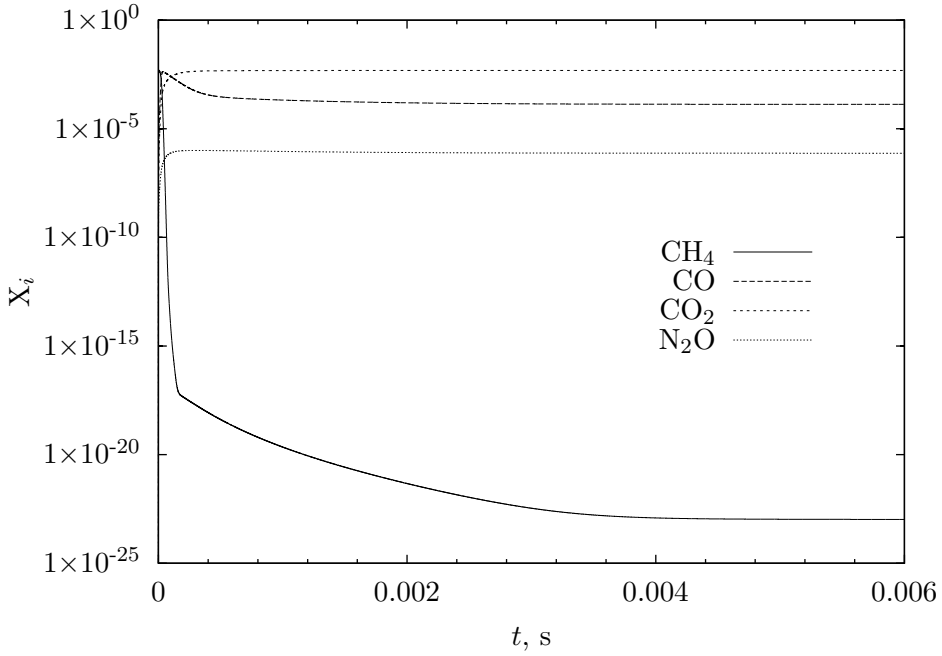


Figure 3.11: *Greenhouse species for a constant-pressure, fixed-mass, perfectly mixed reactor at 2226 K, 1 atm.*

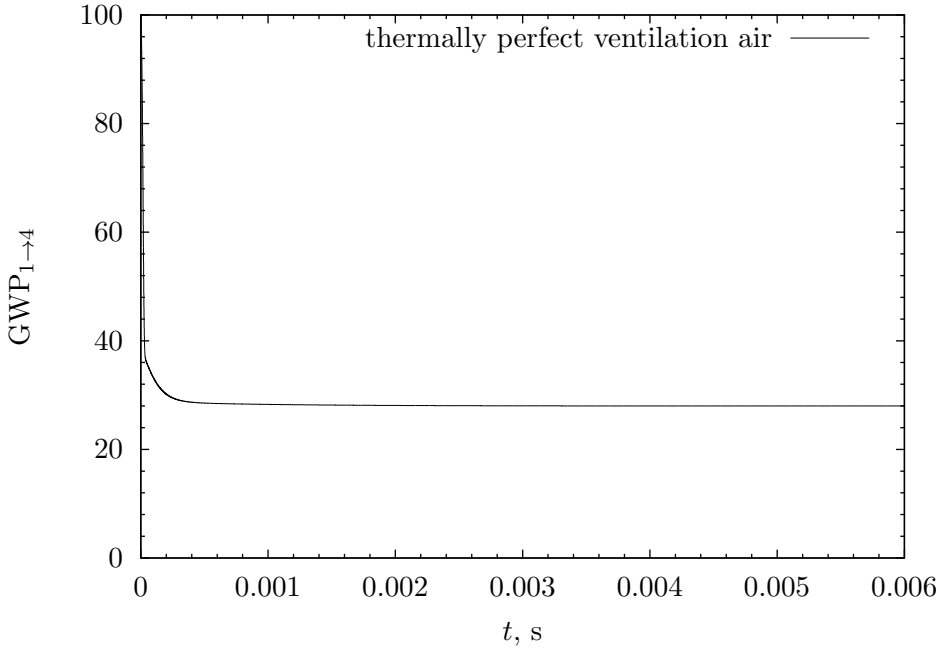


Figure 3.12: *GWP reduction for a constant-pressure, fixed-mass, perfectly mixed reactor at 2226 K, 1 atm.*

3.7 Mechanism selection summary

Of the five methane mechanisms, DRM19 was selected for use in the engine ignition delay study of Part II, based on the selection criteria presented in Table 3.8.

	correct ignition delay at high density	**correct heat release**	**relative computational expense**
Li and Williams	y	n	-
Smooke and Giovangigli	n	y	-
Jazbec *et al.*	n[6]	y	1.0
DRM19	y	y	3.07
DRM22	y	y	3.50
GRI-Mech3.0	y	y	11.45

Table 3.8: *Mechanism selection criteria.*

Considering these, the mechanisms DRM19, DRM22 and GRI-Mech3.0 all provide accurate modelling of ignition delay and heat release. Note that NO_2 is a greenhouse gas but is not included in the reduced kinetic mechanisms of DRM19 and DRM22. Since the peak conditions and work delay of a free-piston engine are unknown, it is also as yet unknown whether Nitrogen chemistry plays a role in reactions. Thus, GRI-Mech3.0 will be used to determine if this is the case, and DRM19 will be used subsequently (as it was the least computationally expensive) if not.

A special mention should be made of the mechanism of Jazbec *et al.*It was designed for methane-air mixtures methane mole fractions between 1 and 2% and temperatures between 1000 and 1200 K. Nevertheless, the mechanism performed remarkably well in ignition delay for temperatures up to 2000 K and provides correct heat release at still higher temperatures. Its computational expense is also the lowest of the mechanisms tested due to the small number of species and reactions it contains. The only reason it is not used in the engine modelling of Part II is that it incorrectly predicts ignition delay at high pressures.

At this point the program `Gaspy` has been validated in modelling both a real gas and finite-rate chemistry. Thus, a brief investigation into the combined effect of these models is made here.

3.8 The effect of real gas models on finite-rate chemistry

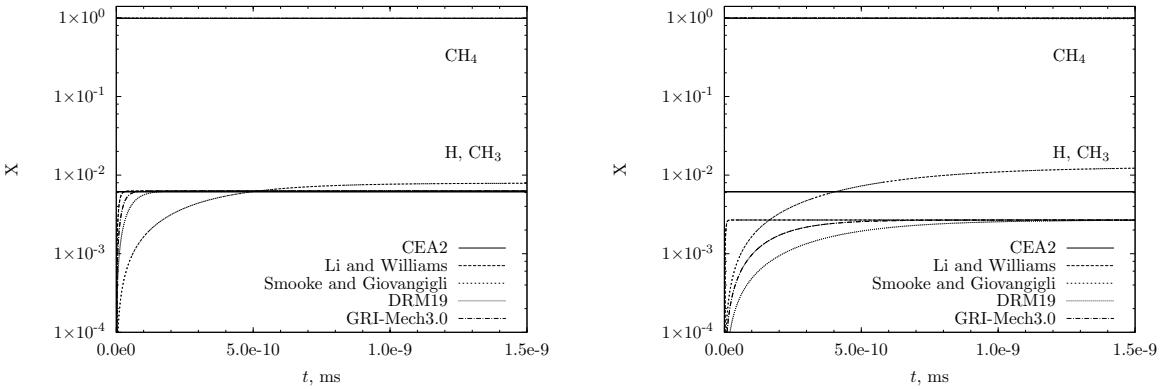
Once elucidated, a kinetic mechanism is independent of the gas model[7]. However, real gas models influence reactions via the thermal model (specifically, the calculation of the Gibbs free energy) which affects both the reverse reaction rate and the equilibrium mole fraction.

Take again the reaction for decomposition of methane at constant temperature and pressure.

[6]The ignition delay for this mechanism is not as accurate as the other (valid) mechanisms, but its low computational expense justifies its consideration.

[7]It is however, important to include real gas effects when deriving the rate coefficients from shock tube experiments operating at conditions where the perfect gas equation of state no longer applies [26]. It is particularly important for pressure-dependent rate coefficients in which the post-shock conditions lie within the bounds of the fall-off regime, since this compounds the error in the perceived reaction rate.

Figure 3.13 compares the equilibrium mole fractions reached by a thermally-perfect gas and a van der Waals gas under high pressure. Both the equilibrium mole fraction and the reaction



(a) *3000 K, 1×10^9 Pa, thermally-perfect gas*

(b) *3000 K, 1×10^9 Pa, van der Waals gas*

Figure 3.13: *High pressure decomposition of CH_4 (const-T, p) at T=3000K using different gas models. As can be seen, the van der Waals equation of state retards the rate of decomposition compared to the thermally perfect equation of state. In addition, the equilibrium value attained by Smooke and Giovangigli is incorrect. This is because a backwards rate is specified for this reaction. Thus, only kinetic mechanisms in which the reverse rates are calculated using the equilibrium constant should be used with real gas models. Specifying a reverse rate couples the kinetic mechanism and the thermal model, which is undesirable. The pressure was chosen to exaggerate the difference between thermally-perfect and thermally real models. Peak pressures seen in a free-piston engine are expected to be substantially lower.*

rates are different between the two gas models.

There is an additional complication for the reaction of Smooke and Giovangigli. Here, the backwards rate is specified based on low pressure experiments and as such, it is no longer independent of the equation of state. Thus, this mechanism is inapplicable at high pressures or with real gas models. More generally, it can be said that any kinetic mechanism with specified reverse rates should not be used away from the state at which they were derived, and are, in addition, incompatible with real gas models.

From Figure 3.13 it appears as though dissociation (and hence combustion) of methane is retarded by real gas effects. The compression ratio required for homogeneous combustion of ventilation air is much greater than that for a conventional diesel engine. As such, while the pressures in a free-piston engine are expected to be substantially lower, real gas effects may still be significant for the compression ignition of ventilation air. This possibility is investigated using a model of a a free-piston combustor in §4.4.1 on page 67.

At this point, the finite-rate chemistry module `Gaspy` has been validated and a mechanism that correctly models ignition delay and heat release for the conditions seen in a free-piston

engine has been selected.

3.A Chapter end notes

3.A.1 reactor.py

```
def const_v_fixed_mass(y, t, Q, g, r, dt_chem, testFlag):
    # Y - mass fractions (kg/kg)
    # w - concentration (mol/m**3)
    # dwdt - molar production rate (mol/s)
    # M - molecular weights (kg/mol)

    nsp = g.get_number_of_species()

    # unpack array
    Q.T[0] = y[0] # temperature
    Q.p = y[1] # pressure
    Y = y[2:nsp+2] # mass fraction

    dYdt = zeros(nsp)

    dwdt = vectord(nsp*[0.0])
    w = vectord(nsp*[0.0])

    # update gas
    for isp in range(nsp):
        Q.massf[isp] = Y[isp]

    g.eval_thermo_state_pT(Q) # concs _are_ updated here

    # copy concs to w
    w = convert_massf2conc(Q.rho, Q.massf, g.M())

    # get rates-of-change
    dwdt = r.rate_of_change_py(Q)

    sum_ew = 0.0
    sum_wcv = 0.0
    for i in range(nsp):
        e_i = g.internal_energy(Q, i)*g.M()[i]
        sum_ew += e_i*dwdt[i]
        sum_wcv += w[i]*g.Cv(Q)*g.M()[i] # where c_v is in J/mol.K

    dTdt = -sum_ew/sum_wcv
    dpdt = PC_R_u*Q.T[0]*sum(dwdt) + PC_R_u*dTdt*sum(w)
    dYdt = (array(dwdt)*g.M())/Q.rho

    return array([dTdt] + [dpdt] + list(dYdt))

def const_p_fixed_mass(y, t, Q, g, r, dt_chem, testFlag):
    # Y - mass fractions (kg/kg)
    # w - concentration (mol/m**3)
    # dwdt - molar production rate (mol/s)
    # M - molecular weights (kg/mol)

    nsp = g.get_number_of_species()

    # unpack array
    Q.T[0] = y[0] # temperature
```

```
    Q.p = y[1] # pressure
    Y = y[2:nsp+2] # mass fraction

    dYdt = zeros(nsp)

    dwdt = vectord(nsp*[0.0])
    w = vectord(nsp*[0.0])

    # update gas
    for isp in range(nsp):
        Q.massf[isp] = Y[isp]

    g.eval_thermo_state_pT(Q) # concs _are_ updated here

    # copy concs to w
    w = convert_massf2conc(Q.rho, Q.massf, g.M())

    # get rates-of-change
    dwdt = r.rate_of_change_py(Q)

    sum_hw = 0.0
    sum_wcp = 0.0

    for i in range(nsp):
        h_i = g.enthalpy(Q, i)*g.M()[i]
        sum_hw += h_i*dwdt[i]
        sum_wcp += w[i]*g.Cp(Q)*g.M()[i] # J/mol.K

    dTdt = -sum_hw/sum_wcp
    dpdt = 0.0
    dYdt = (array(dwdt)*g.M())/Q.rho

    return array([dTdt] + [dpdt] + list(dYdt))
```

Listing 3.1: *An implementation of the constant-pressure and constant-volume, fixed-mass equations. An ODE update method was used to march the solution in time, not shown for clarity.*

3.A.2 simple_reactor.py

```
def const_v_fixed_mass(dt, args):
    Q, g, r, dt_chem = args

    # update assuming const volume
    dt_chem = r.update_state_py(Q, dt, dt_chem)

    # update state
    g.eval_thermo_state_rhoe(Q)

    return [Q, g, r, dt_chem]

def const_p_fixed_mass(dt, args):
    Q, g, r, dt_chem = args

    p_0 = Q.p
    v_0 = (1.0/Q.rho)
    gamma = g.gamma(Q)

    # update assuming const volume
    dt_chem = r.update_state_py(Q, dt, dt_chem)
    g.eval_thermo_state_rhoe(Q)

    # expand isentropically
    dp = Q.p - p_0
    v_1 = v_0*(Q.p/p_0)**(1.0/gamma)
    dv = v_1 - v_0
    de = 0.5*(p_0 + Q.p)*dv

    Q.e[0] -= de
    Q.rho *= v_0/v_1

    # update state
    g.eval_thermo_state_rhoe(Q)

    return [Q, g, r, dt_chem]
```

Listing 3.2: *An operator-split implementation of the constant-pressure and constant-volume, fixed-mass equations. An ODE update method was used to march the solution in time, not shown for clarity.*

3.A.3 Hydrogen mechanism

```
ELEMENTS
H O N
END
SPECIES
H2 H O2 O OH HO2 H2O2 H2O N N2 NO
END
REACTIONS
H2+O2<=>2OH                          0.170E+14 0.00 44780
OH+H2<=>H2O+H                        0.117E+10 1.30  3626
O+OH<=>O2+H                          0.400E+15 -0.50    0
O+H2<=>OH+H                          0.506E+05 2.67  6290
H+O2+M<=>HO2+M                       0.361E+18 -0.72    0
      H2O/ 18.6/ H2/ 2.86/ N2/ 1.26/
OH+HO2<=>H2O+O2                      0.750E+13 0.00     0
H+HO2<=>2OH                          0.140E+15 0.00  1073
O+HO2<=>O2+OH                        0.140E+14 0.00  1073
2OH<=>O+H2O                          0.600E+09 1.30     0
H+H+M<=>H2+M                         0.100E+19 -1.00    0
      H2O/ 0.0/ H2/ 0.0/
H+H+H2<=>H2+H2                       0.920E+17 -0.60    0
H+H+H2O<=>H2+H2O                     0.600E+20 -1.25    0
H+OH+M<=>H2O+M                       0.160E+23 -2.00    0
      H2O/ 5/
H+O+M<=>OH+M                         0.620E+17 -0.60    0
      H2O/ 5/
O+O+M<=>O2+M                         0.189E+14 0.00 -1788
H+HO2<=>H2+O2                        0.125E+14 0.00     0
HO2+HO2<=>H2O2+O2                    0.200E+13 0.00     0
H2O2+M<=>OH+OH+M                     0.130E+18 0.00 45500
H2O2+H<=>HO2+H2                      0.160E+13 0.00  3800
H2O2+OH<=>H2O+HO2                    0.100E+14 0.00  1800
O+N2<=>NO+N                          0.140E+15 0.00 75800
N+O2<=>NO+O                          0.640E+10 1.00  6280
OH+N<=>NO+H                          0.400E+14 0.00     0
END
```

Listing 3.3: *Chemkin-II Hydrogen mechanism [56, Ch. VII].*

3.A.4 DRM19 mechanism

```
!<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>!
!  Reduced version of GRI-MECH 1.2. 19 species ( + N2, AR); 84 reactions. !
!                          PennState Dec, 1994                              !
!<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>!
ELEMENTS
O  H  C  N  AR HE
END
SPECIES
H2      H       O       O2      OH      H2O     HO2
CH2     CH2(S)  CH3     CH4     CO      CO2     HCO
CH2O    CH3O    C2H4    C2H5    C2H6
N2      AR      HE
END
REACTIONS
O+H+M<=>OH+M                          5.000E+17   -1.000      0.00
H2/2.00/ H2O/6.00/ CH4/2.00/ CO/1.50/ CO2/2.00/ C2H6/3.00/ AR/0.70/
O+H2<=>H+OH                           5.000E+04    2.670   6290.00
O+HO2<=>OH+O2                         2.000E+13    0.000      0.00
O+CH2<=>H+HCO                         8.000E+13    0.000      0.00
O+CH2(S)<=>H+HCO                      1.500E+13    0.000      0.00
O+CH3<=>H+CH2O                        8.430E+13    0.000      0.00
O+CH4<=>OH+CH3                        1.020E+09    1.500   8600.00
O+CO+M<=>CO2+M                        6.020E+14    0.000   3000.00
H2/2.00/ O2/6.00/ H2O/6.00/ CH4/2.00/ CO/1.50/ CO2/3.50/ C2H6/3.00/ AR/0.50/
O+HCO<=>OH+CO                         3.000E+13    0.000      0.00
O+HCO<=>H+CO2                         3.000E+13    0.000      0.00
O+CH2O<=>OH+HCO                       3.900E+13    0.000   3540.00
O+C2H4<=>CH3+HCO                      1.920E+07    1.830    220.00
O+C2H5<=>CH3+CH2O                     1.320E+14    0.000      0.00
O+C2H6<=>OH+C2H5                      8.980E+07    1.920   5690.00
O2+CO<=>O+CO2                         2.500E+12    0.000  47800.00
O2+CH2O<=>HO2+HCO                     1.000E+14    0.000  40000.00
H+O2+M<=>HO2+M                        2.800E+18   -0.860      0.00
O2/0.00/ H2O/0.00/ CO/0.75/ CO2/1.50/ C2H6/1.50/ N2/0.00/ AR/0.00/
H+2O2<=>HO2+O2                        3.000E+20   -1.720      0.00
H+O2+H2O<=>HO2+H2O                    9.380E+18   -0.760      0.00
H+O2+N2<=>HO2+N2                      3.750E+20   -1.720      0.00
H+O2+AR<=>HO2+AR                      7.000E+17   -0.800      0.00
H+O2<=>O+OH                           8.300E+13    0.000  14413.00
2H+M<=>H2+M                           1.000E+18   -1.000      0.00
H2/0.00/ H2O/0.00/ CH4/2.00/ CO2/0.00/ C2H6/3.00/ AR/0.63/
2H+H2<=>2H2                           9.000E+16   -0.600      0.00
2H+H2O<=>H2+H2O                       6.000E+19   -1.250      0.00
2H+CO2<=>H2+CO2                       5.500E+20   -2.000      0.00
H+OH+M<=>H2O+M                        2.200E+22   -2.000      0.00
H2/0.73/ H2O/3.65/ CH4/2.00/ C2H6/3.00/ AR/0.38/
H+HO2<=>O2+H2                         2.800E+13    0.000   1068.00
H+HO2<=>2OH                           1.340E+14    0.000    635.00
H+CH2(+M)<=>CH3(+M)                   2.500E+16   -0.800      0.00
     LOW  /  3.200E+27   -3.140    1230.00/
     TROE/  0.6800    78.00   1995.00   5590.00 /
H2/2.00/ H2O/6.00/ CH4/2.00/ CO/1.50/ CO2/2.00/ C2H6/3.00/ AR/0.70/
H+CH3(+M)<=>CH4(+M)                   1.270E+16   -0.630    383.00
     LOW  /  2.477E+33   -4.760    2440.00/
     TROE/  0.7830    74.00   2941.00   6964.00 /
```

```
H2/2.00/ H2O/6.00/ CH4/2.00/ CO/1.50/ CO2/2.00/ C2H6/3.00/ AR/0.70/
H+CH4<=>CH3+H2                          6.600E+08    1.620   10840.00
H+HCO(+M)<=>CH2O(+M)                    1.090E+12    0.480    -260.00
     LOW  /  1.350E+24   -2.570   1425.00/
     TROE/  0.7824  271.00  2755.00  6570.00 /
H2/2.00/ H2O/6.00/ CH4/2.00/ CO/1.50/ CO2/2.00/ C2H6/3.00/ AR/0.70/
H+HCO<=>H2+CO                           7.340E+13    0.000       0.00
H+CH2O(+M)<=>CH3O(+M)                   5.400E+11    0.454    2600.00
     LOW  /  2.200E+30   -4.800   5560.00/
     TROE/  0.7580   94.00  1555.00  4200.00 /
H2/2.00/ H2O/6.00/ CH4/2.00/ CO/1.50/ CO2/2.00/ C2H6/3.00/
H+CH2O<=>HCO+H2                         2.300E+10    1.050    3275.00
H+CH3O<=>OH+CH3                         3.200E+13    0.000       0.00
H+C2H4(+M)<=>C2H5(+M)                   1.080E+12    0.454    1820.00
     LOW  /  1.200E+42   -7.620   6970.00/
     TROE/  0.9753  210.00   984.00  4374.00 /
H2/2.00/ H2O/6.00/ CH4/2.00/ CO/1.50/ CO2/2.00/ C2H6/3.00/ AR/0.70/
H+C2H5(+M)<=>C2H6(+M)                   5.210E+17   -0.990    1580.00
     LOW  /  1.990E+41   -7.080   6685.00/
     TROE/  0.8422  125.00  2219.00  6882.00 /
H2/2.00/ H2O/6.00/ CH4/2.00/ CO/1.50/ CO2/2.00/ C2H6/3.00/ AR/0.70/
H+C2H6<=>C2H5+H2                        1.150E+08    1.900    7530.00
H2+CO(+M)<=>CH2O(+M)                    4.300E+07    1.500   79600.00
     LOW  /  5.070E+27   -3.420  84350.00/
     TROE/  0.9320  197.00  1540.00 10300.00 /
H2/2.00/ H2O/6.00/ CH4/2.00/ CO/1.50/ CO2/2.00/ C2H6/3.00/ AR/0.70/
OH+H2<=>H+H2O                           2.160E+08    1.510    3430.00
2OH<=>O+H2O                             3.570E+04    2.400   -2110.00
OH+HO2<=>O2+H2O                         2.900E+13    0.000    -500.00
OH+CH2<=>H+CH2O                         2.000E+13    0.000       0.00
OH+CH2(S)<=>H+CH2O                      3.000E+13    0.000       0.00
OH+CH3<=>CH2+H2O                        5.600E+07    1.600    5420.00
OH+CH3<=>CH2(S)+H2O                     2.501E+13    0.000       0.00
OH+CH4<=>CH3+H2O                        1.000E+08    1.600    3120.00
OH+CO<=>H+CO2                           4.760E+07    1.228      70.00
OH+HCO<=>H2O+CO                         5.000E+13    0.000       0.00
OH+CH2O<=>HCO+H2O                       3.430E+09    1.180    -447.00
OH+C2H6<=>C2H5+H2O                      3.540E+06    2.120     870.00
HO2+CH2<=>OH+CH2O                       2.000E+13    0.000       0.00
HO2+CH3<=>O2+CH4                        1.000E+12    0.000       0.00
HO2+CH3<=>OH+CH3O                       2.000E+13    0.000       0.00
HO2+CO<=>OH+CO2                         1.500E+14    0.000   23600.00
CH2+O2<=>OH+HCO                         1.320E+13    0.000    1500.00
CH2+H2<=>H+CH3                          5.000E+05    2.000    7230.00
CH2+CH3<=>H+C2H4                        4.000E+13    0.000       0.00
CH2+CH4<=>2CH3                          2.460E+06    2.000    8270.00
CH2(S)+N2<=>CH2+N2                      1.500E+13    0.000     600.00
CH2(S)+AR<=>CH2+AR                      9.000E+12    0.000     600.00
CH2(S)+O2<=>H+OH+CO                     2.800E+13    0.000       0.00
CH2(S)+O2<=>CO+H2O                      1.200E+13    0.000       0.00
CH2(S)+H2<=>CH3+H                       7.000E+13    0.000       0.00
CH2(S)+H2O<=>CH2+H2O                    3.000E+13    0.000       0.00
CH2(S)+CH3<=>H+C2H4                     1.200E+13    0.000    -570.00
CH2(S)+CH4<=>2CH3                       1.600E+13    0.000    -570.00
CH2(S)+CO<=>CH2+CO                      9.000E+12    0.000       0.00
CH2(S)+CO2<=>CH2+CO2                    7.000E+12    0.000       0.00
CH2(S)+CO2<=>CO+CH2O                    1.400E+13    0.000       0.00
```

```
CH3+O2<=>O+CH3O                                  2.675E+13     0.000   28800.00
CH3+O2<=>OH+CH2O                                 3.600E+10     0.000    8940.00
2CH3(+M)<=>C2H6(+M)                              2.120E+16    -0.970     620.00
      LOW   /   1.770E+50    -9.670    6220.00/
      TROE/   0.5325   151.00   1038.00   4970.00 /
H2/2.00/ H2O/6.00/ CH4/2.00/ CO/1.50/ CO2/2.00/ C2H6/3.00/ AR/0.70/
2CH3<=>H+C2H5                                    4.990E+12     0.100   10600.00
CH3+HCO<=>CH4+CO                                 2.648E+13     0.000       0.00
CH3+CH2O<=>HCO+CH4                               3.320E+03     2.810    5860.00
CH3+C2H6<=>C2H5+CH4                              6.140E+06     1.740   10450.00
HCO+H2O<=>H+CO+H2O                               2.244E+18    -1.000   17000.00
HCO+M<=>H+CO+M                                   1.870E+17    -1.000   17000.00
H2/2.00/ H2O/0.00/ CH4/2.00/ CO/1.50/ CO2/2.00/ C2H6/3.00/
HCO+O2<=>HO2+CO                                  7.600E+12     0.000     400.00
CH3O+O2<=>HO2+CH2O                               4.280E-13     7.600   -3530.00
C2H5+O2<=>HO2+C2H4                               8.400E+11     0.000    3875.00
END
```

Listing 3.4: *DRM19 methane mechanism [43].*

3.A.5 Ignition delay definitions

For the papers selected here, ignition delay was defined to be the time between:

1. Seery and Bowman [52] - "the heating of the gas by the reflected shock wave and the most rapid increase in pressure or characteristic emission (or absorption)" including "chemiluminescent emission of OH, CH, C_2 and CO and the 3067Å absorption of the OH* radical."

2. Lifshitz *et al.* [51] - the arrival of the shock and the sudden rise in "both the p and heat flux traces ... from their plateau values, or a change in their slope, in the shocked, slightly reacted gas caused by the onset of combustion."

3. Tsuboi and Wagner [36] - the reflected wave and the intersection between the $[CH_4]$ gradient and its original concentration. This was defined graphically. Measurements were made of the emission of CH_4 at wavelengths between 3.1–3.8μ.

4. Grillo and Slack [58] - "the maximum pressure rise and the maximum positive rate of change of the OH emission signal."

5. Krishnan and Ravikumar [50] - "the arrival of the shock at the end plate and the appearance of visible light."

6. Petersen *et al.* [37] - "the initial OH formation [or the arrival of the reflected shock wave] and the peak OH concentration."

7. Zallen and Wittig [57] - "the shock arrival to the maximum change in the signal gradient" of the "chemiluminescent radiation emitted from radicals such as OH*, CH* and others."

3.A.6 Calculation of the equilibrium constant

At equilibrium, the production and destruction rates of each species are equal and opposite. Hence, the time rate-of-change of all species is zero. Applying this to (3.5) and rearranging yields

$$\frac{k_f}{k_r} = K_c = \frac{\prod_{i=i''}[\mathrm{X}_i]^{n_i}}{\prod_{i=i'}[\mathrm{X}_i]^{n_i}} \quad . \tag{3.21}$$

The ratio of the forward to reverse rate coefficient is called the equilibrium constant, K_c. Pressure-dependent rate coefficients have no influence on this value.

$$K_c = \begin{cases} \dfrac{k_0[M]}{k_r} = \dfrac{[\mathrm{CH_4}][\mathrm{M}]}{[\mathrm{CH_3}][\mathrm{H}][\mathrm{M}]} = \dfrac{[\mathrm{CH_4}]}{[\mathrm{CH_3}][\mathrm{H}]} & \text{at low pressures} \\ \dfrac{k_\infty}{k_r} = \dfrac{[\mathrm{CH_4}]}{[\mathrm{CH_3}][\mathrm{H}]} & \text{at high pressures} \end{cases} \tag{3.22}$$

The condition for equilibrium can also be described using the Gibbs free energy

$$dg|_{T\,p\,\nu} = 0 \quad . \tag{3.23}$$

Using this definition, K_c becomes

$$K_c = K_p \left(\frac{p_{\mathrm{atm}}}{\mathscr{R}T}\right)^{\sum_{i=1}^{n_{\mathrm{sp}}} c_i} \tag{3.24}$$

where

$$K_p = \exp\left(\frac{\sum_{i=1}^{n_{\mathrm{sp}}} c_i g_i}{\mathscr{R}T}\right) \quad . \tag{3.25}$$

If the forward rate coefficient is known, the reverse rate coefficient (that yields the correct equilibrium species' concentrations) may be calculated as

$$k_r = \frac{k_f}{K_c} \quad . \tag{3.26}$$

Many mechanisms supply just the forward rate data with this intention.

3.A.7 Derivation of the Lindemann-Hinshelwood Form

Consider the decomposition for methane:

$$\mathrm{CH_3} + \mathrm{H}\,(+\mathrm{M}) \underset{k_r}{\overset{k_f}{\rightleftharpoons}} \mathrm{CH_4}\,(+\mathrm{M})$$

Applying (3.5), the net production rate of $[\mathrm{CH_4}]$ is

$$\frac{[\mathrm{CH_4}]}{dt} = [\mathrm{M}]\,(k_f[\mathrm{CH_3}][\mathrm{H}] - k_r[\mathrm{CH_4}]) \quad . \tag{3.27}$$

This was found experimentally to reduce to a unimolecular forward reaction at high pressures [59].

$$\frac{[\mathrm{CH_4}]}{dt} = k_f[\mathrm{CH_4}] \tag{3.28}$$

Lindemann proposed the following two-step reaction to model the forward rate

$$\begin{aligned} \mathrm{CH_4 + M} &\underset{k_1}{\overset{k_0}{\rightleftharpoons}} \mathrm{CH_4{}^* + M} \\ \mathrm{CH_4{}^*} &\xrightarrow{k_2} \mathrm{CH_3 + H} \end{aligned} \tag{3.29}$$

where $\mathrm{CH_4{}^*}$ is an intermittent radical which undergoes either a bimolecular or unimolecular reaction depending on the concentration of the third body. This model was further developed by Hinshelwood [60] to include multiple degrees of freedom.

The reactions k_0 and k_1 are bimolecular whereas k_2 is forward and unimolecular. The resulting reaction is said to be second order. Qualitatively, at the low pressure limit there are not enough third-bodies to facilitate the reverse reaction before the radicals go to completion. In numerical modelling of these reactions, $[\mathrm{CH_4{}^*}]$ is not treated as a species.

For a unimolecular reaction, it is necessary to rewrite equation (3.5)

$$\frac{d[\mathrm{X}_{i''}]}{dt} = (c_{i''} - c_{i'}) \left\{ k_2 [\mathrm{X}_i{}^*] \prod_{i=i'} [\mathrm{X}_{i'}]^{c_i} \right\} \quad (\mathrm{X}_i{}^* \notin \mathrm{X}_i) \quad . \tag{3.30}$$

From equation (3.30) the reaction rate for the reagents and products of (3.29) are

$$\frac{d[\mathrm{X}_{i'}]}{dt} = [\mathrm{M}] \left(k_0 [\mathrm{CH_4}] - k_1 [\mathrm{CH_4{}^*}] \right) \quad \text{and} \tag{3.31a}$$

$$\frac{d[\mathrm{X}_{i''}]}{dt} = k_2 [\mathrm{CH_4{}^*}] \tag{3.31b}$$

respectively. Equating (3.31) and assuming $\frac{d[\mathrm{CH_4{}^*}]}{dt} = 0$ (steady-state), $[\mathrm{CH_4{}^*}]$ may be expressed as

$$[\mathrm{CH_4{}^*}] = \frac{k_0 [\mathrm{M}]}{k_2 + k_1 [\mathrm{M}]} [\mathrm{CH_4}] \quad . \tag{3.32}$$

The forward rate can now be found by combining (3.32) and (3.31a)

$$\frac{d[\mathrm{X}_{i''}]}{dt} = \left(\frac{k_2 k_0 [\mathrm{M}]}{k_2 + k_1 [\mathrm{M}]} \right) [\mathrm{CH_4}] \tag{3.33}$$

This is the Lindemann-Hinshelwood form of the rate coefficient [56] which is one type of a greater class known as pressure-dependent rate coefficients. The reverse rate may be calculated using the equilibrium constant or specified explicitly.

In literature, k_f takes one of two forms, either

$$k_f = \frac{k_\infty}{1 + \frac{\alpha}{[M]}} \quad \text{where} \quad \alpha = \frac{k_2}{k_1} \tag{3.34}$$

as in [61], or

$$k_f = \frac{k_0 [\mathrm{M}]}{1 + \frac{k_0 [M]}{k_\infty}} \quad \text{where} \quad k_\infty = k_0 \frac{k_2}{k_1} \tag{3.35}$$

as in [56].

3.A.8 The rate coefficient of Li and Williams

The decomposition of CH_4 was elementary reaction 10 of Li and Williams included for comparison. It seems to overpredict the rate coefficient at low densities when compared to other mechanisms.

j	**Reaction**	A (mol,cm,s,K)	β	E (kJ.mol^{-1}K^{-1})
10	$H + CH_3(+M) \rightleftharpoons CH_4(+M)$	2.220×10^{16}	0.00×10^{0}	4.390×10^{2}
	low	6.590×10^{25}	-1.80×10^{0}	4.390×10^{2}

Table 3.9: *Reaction 10, Li and Williams. This seems to overpredict the rate coefficient at low densities when compared to other mechanisms.*

3.A.9 The rate coefficient of Smooke and Giovangigli

The decomposition of CH_4 is elementary reaction 10 of the reduced mechanism of Smooke and Giovangigli. Note that this rate coefficient is supplied in a slightly different form to that in [56] and specifies a backward rate coefficient.

j	**Reaction**	A (mol,cm,s,K)	β	E (cal.mol^{-1}K^{-1})
10f	$CH_4\,(+M) \longrightarrow CH_3 + H\,(+M)$	6.300×10^{14}	0.00	1.040×10^{5}
10r	$CH_3 + H\,(+M) \longrightarrow CH_4\,(+M)$[8]	5.200×10^{12}	0.00	-1.310×10^{3}

Table 3.10: *CH_4 decomposition model of [40].*

The rate coefficients are of the form

$$k = \frac{k_\infty}{1 + \frac{k_{\text{fall}}}{[\text{M}]}} \quad , \tag{3.36}$$

where $k_{\text{fall}} = 6.300 \times 10^{-3} \exp\left(-\frac{1.8\times10^4}{RT}\right)$. These coefficients must be converted before being used in a Chemkin-II-like finite-rate chemistry code. According to [40], the third body value does not include efficiencies and as such will be denoted here as the third body concentration $\bar{\rho} = \frac{p_{\text{atm}}}{RT}$.

The Chemkin-II equivalent format for this reaction is given in Table 3.11 for completion.

Flame kinetics are quite different from ignition kinetics and thus their rates may not be suitable for modelling ignition delay (private correspondence with Bilger, 2008).

3.A.10 Lua input for a pressure-dependent rate coefficient

The syntax for including a pressure-dependent rate coefficient in Lua is

[8] The keen observer will note that the reverse unimolecular reaction (given in this notation) is not consistent with the Lindemann-Hinshelwood model, as it is actually a unimolecular deactivation of the CH_4* radical. This is the result of the numerical implementation rather than the chemical process.

j	**Reaction**	A (mol,cm,s,K)	β	E (cal.mol^{-1}K^{-1})
10f	$CH_4\,(+M) \longrightarrow CH_3 + H\,(+M)$	6.300×10^{14}	0.00	1.040×10^{5}
	low	1.000×10^{17}	0.00	8.600×10^{4}
10r	$CH_3 + H\,(+M) \longrightarrow CH_4\,(+M)$	5.200×10^{12}	0.00	-1.310×10^{3}
	low	8.254×10^{14}	0.00	-1.931×10^{4}

Table 3.11: *CH_4 decomposition model of [40] in Chemkin-II form. Rate coefficients are of the form of* (3.6).

```
-- scaling factor, 1/R
S = 1.0/1.987

reaction{'H + CH3 ( + M ) <=> CH4 ( + M )',
    fr={'pressure dependent',
        k_inf={A=1.39000e+16, n=-5.34000e-01, T_a=5.36000e+02*S},
        k_0={A=2.62000e+33, n=-4.76000e+00, T_a=2.44000e+03*S},
        Troe={a=.7830, T3=74.00, T1=2941.00, T2=6964.00},
    },
    efficiencies={CH4=3.00, Ar=0.70},
    label='r0'
}
```

Listing 3.5: *Reaction 51 of GRI-Mech3.0*

where *fr*, *br*, *Troe*, *inf*, *low* and *efficiencies* are the forward and reverse rate coefficients (each consisting of an inf and low rate coefficient), the Troe variables and the third body efficiencies respectively.

Part II

Engine component models

Chapter 4

Engine dynamics

The engine design makes use of the "free-piston" concept, whereby an unconstrained piston is contained between two in-line, opposing combustors (see Figure 1.2 on page 7). This engine type has inherent mechanisms that improve efficiency over standard compression ignition engines and might allow operation at very low concentrations of methane. Oscillation of the piston along the axis of the cylinders drives the thermodynamic cycles in both combustors. A linear electric motor provides the energy to drive the engine with the moving piston representing the driven element. The same motor can also be used to extract energy from the piston if it is available.

To find the minimum mole fraction of CH_4 required to sustain operation this engine, a numerical model was developed by building up a verified set of component processes. These processes include piston and gas dynamics, finite-rate chemistry, one-dimensional fluid flow, heat transfer, friction and control by the electric motor. The energy imparted by the motor per stroke allowed the minimum mole fraction of CH_4 to be determined.

In addition to this, the effect of the Abel-Noble and van der Waals equations of state was assessed since it has been suggested that the compression ratios required in modern diesel engines approach the limits of accuracy of the perfect gas equation of state [27]. This assessment included thermal behaviour during isentropic compression and the rate of reaction near the point of piston reversal.

To begin with, an analytical model of a free-piston compressor was developed. This was used to both verify the numerical implementation and help in the development of the control system (see Chapter 7 on page 112). The exhausting process for this engine is described in Chapter 5. Models for friction and heat transfer are developed in Chapter 6.

4.1 Dynamics of a free-piston compressor

The peak temperature reached in a free-piston compressor is determined by the state of the system at the start of the stroke. Consider the idealised free-piston compressor of Figure 4.1. The gas is assumed to be at the atmospheric state. It is also assumed that, during compression, the rings form a perfect seal with the cylinder wall and no blow-by of gas occurs. For isentropic compression of an ideal gas from state 0, the acceleration of a frictionless piston may be written as

$$\frac{du}{dt} = -\frac{pA}{m_p} = -\frac{p_0 A}{m_p}\left(\frac{\nu_0}{\nu}\right)^{\gamma}. \tag{4.1}$$

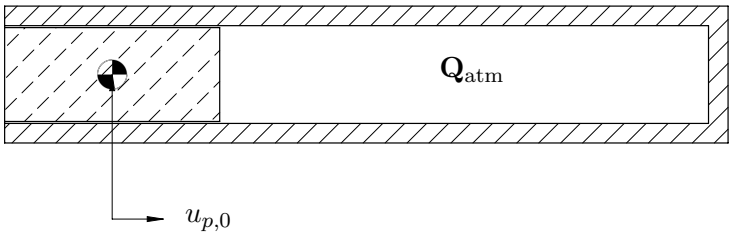


Figure 4.1: *Ideal free-piston compressor.*

Using separation of variables and performing the co-ordinate transform $L = (x_c - x_p)$, equation (4.1) may be written as

$$udu = \frac{p_0 A}{m_p} \left(\frac{L_0}{L}\right)^{\gamma} dL. \tag{4.2}$$

Completing the integration to the point of piston reversal $[x, u] = [x_s, 0]$ yields the stroke length

$$L_s = L_0 - \left[{L_0}^{(1-\gamma)} - \frac{m_p\,(1-\gamma)}{2p_0 A {L_0}^{\gamma}} {u_{p,0}}^2\right]^{\frac{1}{(1-\gamma)}}. \tag{4.3}$$

The *peak temperature* is defined here as the temperature a gas mixture reaches at the point of piston reversal. The initial piston velocity is defined as the piston velocity at poppet valve close. For a given initial piston velocity, the additional force required for the gas to reach a given peak temperature can also be determined. The procedure for this is shown in §7.A.1 on page 116.

Using the energy method, the acceleration of the piston and change in energy of the gas may be written as

$$\frac{du}{dt} = -\frac{pA}{m_p} \qquad \text{and} \tag{4.4a}$$

$$\frac{de}{dt} = pAu \tag{4.4b}$$

respectively. These are the state equations which (by incorporating a gas model) may be integrated numerically to completely describe an isentropic free-piston compressor. This numerical model has the advantage of allowing the inclusion of friction and heat transfer models as needed (see Chapter 6 on page 90).

This approach was implemented in the free-piston compressor code in Listing B.6 on page 194. The piston trajectory is shown in Figure 4.2. The final step here was linearly interpolated to $u_p = 0$.

This program was verified in two ways:

1. by an energy balance and
2. by comparison to the analytical solution (4.3).

For the energy balance, the state equations were integrated over a stroke using different gas models and the energy of the system was summated and shown to be constant (see Figure 4.3). The ideal gas model has constant specific heats and uses the perfect gas equation of state. The thermally-perfect gas model uses the thermodynamic curve fits and the perfect gas equation of

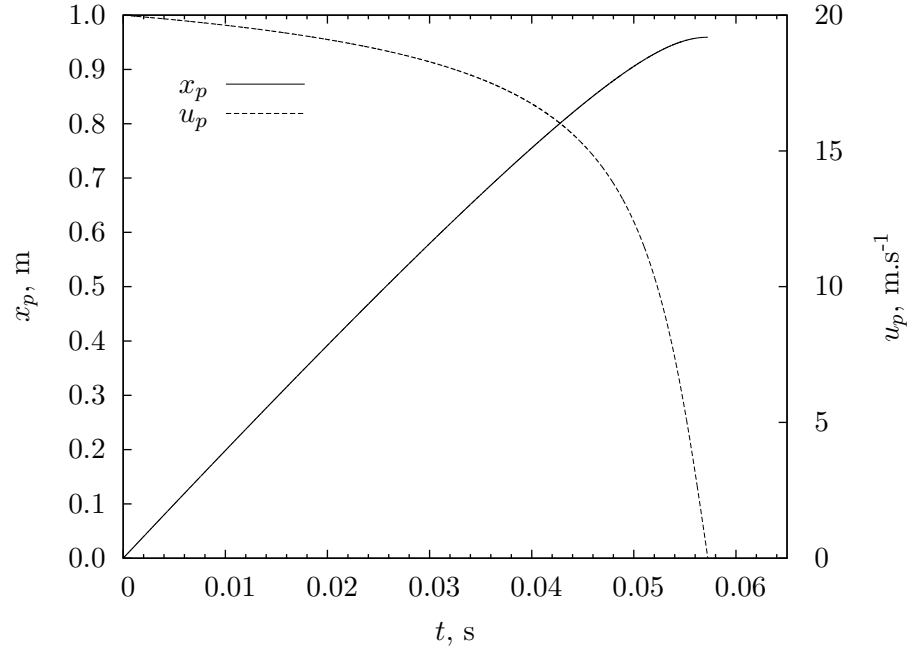


Figure 4.2: *Piston trajectory. Note the sharp deceleration. Initial conditions for this case were m_p=100kg, D=0.2m, L=1.0m filled with thermally-perfect ventilation air at atmospheric conditions.*

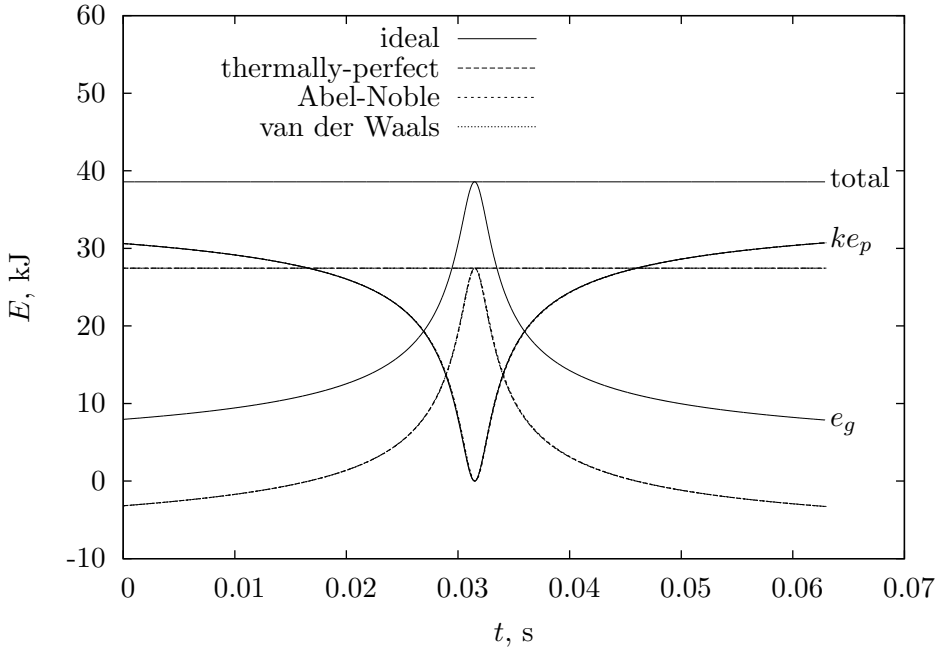


Figure 4.3: *System energy balance showing components of piston kinetic energy and gas sensible energy for an Otto cycle using air of various gas models starting at the atmospheric state.*

state. The Abel-Noble and van der Waals gas models combine thermodynamic curve fits with real thermal behaviour and their respective equations of state. These are more fully described in §2.1.

The second part of validation included the comparison between the analytical (4.3) and numerical solutions (using an ideal gas) shown in Figure 4.4. The curves in this figure are

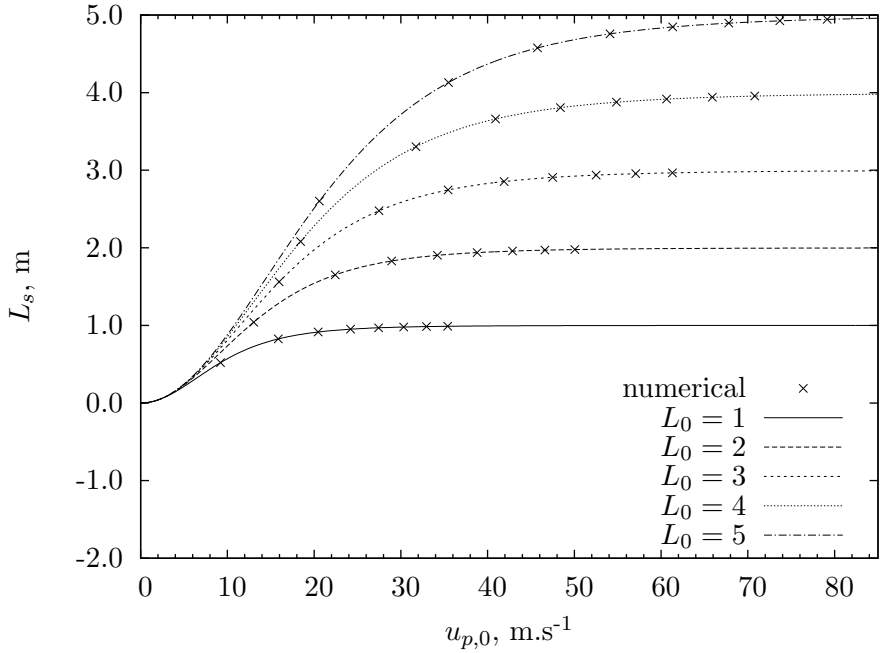


Figure 4.4: *Stroke length for free-piston compression of ideal standard air where m_p = 100kg and D = 0.25m. Numerical results correspond to peak temperatures of 400K to 1800K in 200K increments.*

similar, that is, they converge to a single curve when normalised. Not only does this figure verify the program, it shows important relationships between the system variables, namely that the compression ratio is determined for a given mass of air by particular initial kinetic energy. These relationships remained the most predominant as more secondary process models were incorporated.

The compression of a gas in a free-piston compressor was repeated using thermally-perfect, Abel-Noble and van der Waals gas models to show the difference between them. The resulting curves were then normalised for clarity (see Figure 4.5). The normalised initial piston velocity of Figure 4.5 can be interpreted as the velocity required by one kilogram of piston to compress one kilogram of gas to a given peak temperature. It can also be interpreted as the initial conditions required to achieve a particular volumetric compression ratio. The difference between the thermally-perfect and real gas models becomes more pronounced at higher volumetric compression ratios. Figure 4.5 can be used as is to design an ideal compressor, for example, for $u_{p,0}$=20m.s^{-1}, T_{ig}=1300K and $\rho_0 = \rho_{\text{atm}}$, $\frac{m_p}{AL_c}$ =4.765×10^3, which can be used to choose the remaining variables. Applying this result to an engine means that if large volumes of ventilation

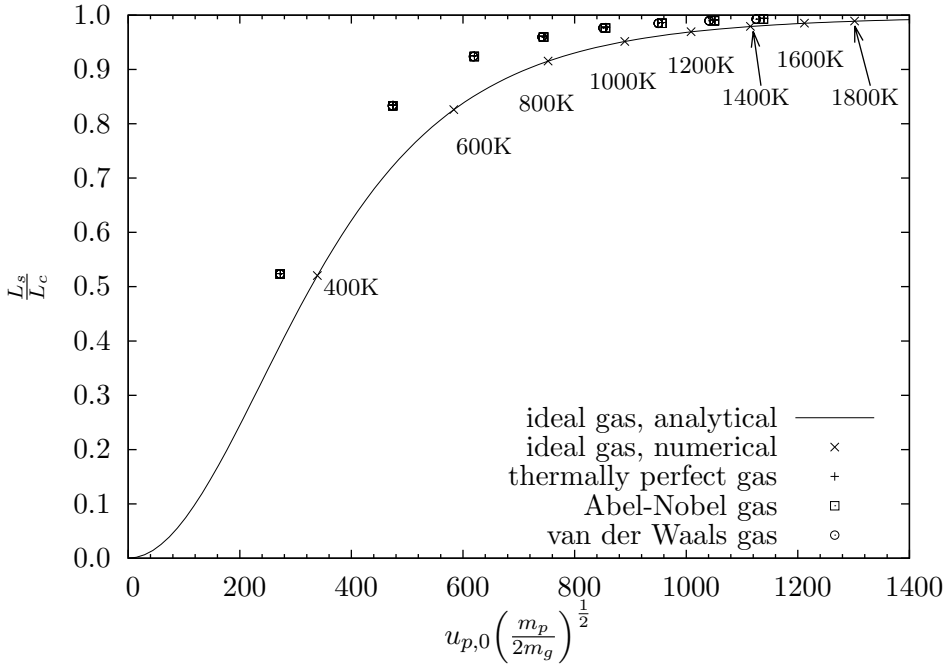


Figure 4.5: *The effect of real gas models on compression of ventilation air. The output here has been normalised as indicated. The analytical solution for an ideal gas is included for contrast. As can be seen, the real gas models require progressively lower piston velocities to reach the same peak temperature, which occurs at practically the same volumetric compression ratio. For example, for compression to 1400K, the piston speed needs to be about 0.8% slower using a van der Waals gas as opposed to a thermally-perfect gas. This effect increases with compression ratio.*

air required to be processed, heavy pistons and small bore areas or cylinder lengths will also be necessary.

4.2 Otto cycle

The ideal Otto cycle (see Figure 4.6) involves isentropic compression, constant-volume heat addition, isentropic expansion and constant-volume heat removal (usually through exhausting). Figure 4.6 shows the first three stages of the Otto cycle using the ideal gas model, where heat equivalent to the combustion enthalpy of ventilation air has been added to the gas at the point of piston reversal.

This cycle may be described using the state of the piston or the state of the gas (see Table 4.1). Work is performed between $x_{p,1}$ and $x_{p,2}$ as the piston moves through the electric motor. The Otto cycle is defined as the process $\mathbf{Q}_{1\to4}$. If this process is performed isentropically with an ideal gas, work is produced at the Otto cycle efficiency

$$\eta_{\text{th,Otto}} = \frac{w_{\text{out}}}{h_c} = 1 - \frac{1}{r^{(\gamma-1)}} \ , \tag{4.5}$$

where r is the compression ratio and γ is the isentropic exponent. Increasing the compression

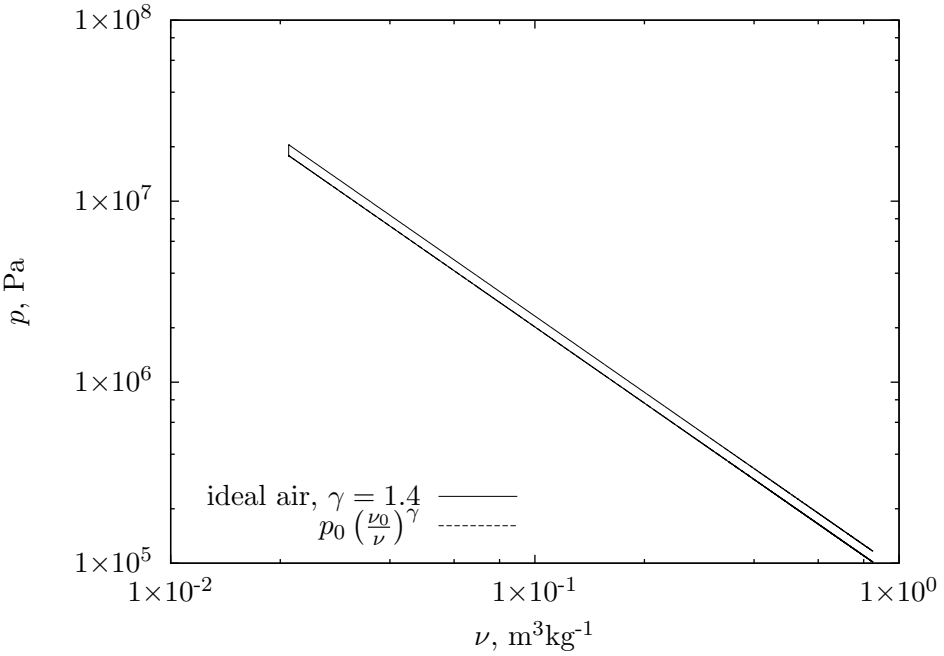


Figure 4.6: *The Otto cycle performed using ideal air starting at the atmospheric state and adding 140kJ.kg^{-1} heat at the point of piston reversal. This figure validates the ideal gas implementation and clearly shows the low energy content of ventilation air. This cycle has a volumetric compression ratio of 23.14, with parameters of m_p = 400kg, L_c = 4m, D = 0.1m and $u_{p,0}$ = 10m.s^{-1}.*

engine description	**piston location**	**gas description**	**gas state**
exhaust valves close	$x_{p,1}$	initial state	$\mathbf{Q}_{\text{atm}}$
piston leaves solenoid	$x_{p,2}$	compression	$\mathbf{Q}_1$
piston reversal	$x_{p,s}$	ignition and subsequent combustion	$\mathbf{Q}_{2,3}$
piston enters solenoid	$x_{p,2}$	expansion	$\mathbf{Q}_4$
exhaust valves open	$x_{p,1}$	atmospheric pressure	$\mathbf{Q}_{\text{atm}}$

Table 4.1: *Engine cycle stages.*

ratio increases the efficiency diminishingly. This trend is limited by the peak pressure that may be contained by the cylinder, which is the result of both the compression ratio and combustion enthalpy. Since the combustion enthalpy of ventilation air is low, the compression ratio may be greater for a given peak pressure. For this engine, the compression ratio is determined by the piston kinetic energy at poppet valve close.

For an ideal Otto cycle, heat release due to combustion is instantaneous and occurs at the point of piston reversal. Actual chemical reactions occur at a finite rate and are a function of temperature and species concentration as discussed in §3.2.1. Ideally these go to completion at

the point of piston reversal. The irreversibilites of a real Otto cycle are both internal (mixing, chemistry and viscous dissipation) and between the system and its surroundings (heat-transfer, friction and work). Under suitable conditions, the Otto cycle efficiency can be improved using regeneration of heat and/or pressure.

Heat regeneration is the process in which some of the enthalpy of the exhaust gas is transferred to the unprocessed gas prior to compression. Heat regeneration is attractive when excess heat is not used to produce work in the heat cycle (for example, when using fuel-rich mixtures). It is not possible for ventilation air because of the low calorific value.

Pressure regeneration may be performed either by expanding the gas to p_{atm} within the cylinder, by shaft work (between a turbine and compressor), or as an expansion wave (in a tuned exhaust pipe). The first option is preferable since it does not require an additional process (with an associated efficiency). The additional length required to expand the combustion products to p_{atm} is not a simple fraction of the cylinder length, but for high compression ratios, $0.5L_c$ was found to be sufficient.

4.3 Autoignition in a rapid compression machine

Prior to shock tubes, combustion was investigated using rapid compression machines [48]. A rapid compression machine is a piston-cylinder assembly that operates over a single stroke at subsonic speeds. Livengood and Leary [48] investigated premixed combustion in such a device. They noted, in particular, the spatial nonuniformity of the ignition process and that combustion was almost always preceded by bright spots near the circumference of the cylinder. Nonuniform of the temperature field was found to be the most convincing explanation. To visualise this, Schlieren photography of the compression of dry air was performed (see Figure 4.7). As can be seen, the compression process resulted in an undeveloped, turbulent boundary layer. The fact that ignition occurred first in the boundary layer suggested that the boundary layer had a higher temperature than the isentropic core temperature. It was surmised that this was due to either the heated walls generating this profile initially, or due to viscous work performed by the piston scraping the boundary layer off the cylinder wall. This unusual temperature field has a large effect on the heat transfer and has been the source of a number of difficulties in its modelling (see §6.2).

It is important to note that for homogeneous combustion there is no initiation and propagation of a flame as was seen here. As such, the nonuniform temperature field has no effect on the ignition process. Certainly, combustion occurs at a different rate in the boundary layer, but combustion of the majority of the gas in the isentropic core is determined solely by the compression process.

Livengood and Leary reviewed the literature on similar devices and found the ignition delay to vary between devices. Since the work delay of rapid compression machines is large compared to the ignition delay, the assumption that ignition begins after compression was being violated. This eventually led to the use of shock tubes for ignition delay experiments.

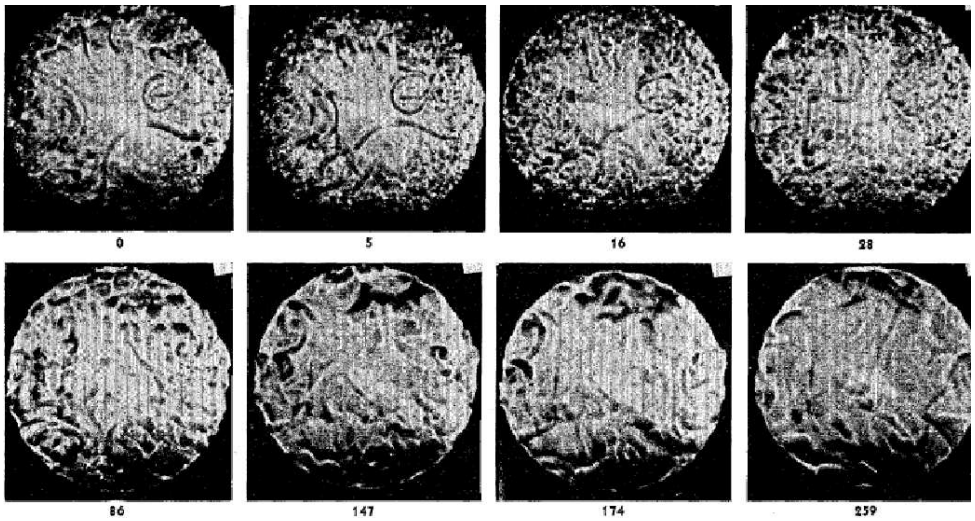


Figure 4.7: *Schlieren photographs of compressed dry air with time in ms from top dead center (reprinted from [48] with permission). The piston remains stationary throughout this sequence. As can be seen, the boundary layer is highly turbulent. The density gradients are the result of spatially nonuniform temperature, since pressure gradients would disappear in a few milliseconds. The dark areas correspond to regions of higher temperature. Test conditions were: initial pressure, 107kPa, initial temperature 338K and compression ratio 12.6. The thermal boundary layer at time $t = 0$ appears to be larger than the cylinder length.*

4.4 Combustion of ventilation air in a free-piston compressor

Consider a CH_4-air mixture undergoing compression ignition by a free-piston. Similar to the the operator-split, constant-pressure fixed-mass reactor (validated in §3.4) a free-piston compressor simulation (see Listing B.4 on page 168) performs the following operations each timestep:

1. integrating the chemistry using a number of substeps (which changes internal and chemical energy)[1]

2. expanding (or compressing) the gas isentropically subject to the dynamics of a free-piston

3. evaluating the state (assuming correct properties for density and internal energy).

The *combustion temperature* is defined as the temperature during compression at which combustion goes to completion. It is desired that the peak temperature is the same as the combustion temperature. That is, as the piston velocity approaches zero, the combustion delay approaches zero, such that combustion has gone to completion at $u_p = 0$.

The free-piston compressor described in §4.1 on page 56 was used along with the DRM19 and GRI-Mech3.0 kinetic models to find the combustion temperature (see Figure 4.8). This

[1]There are two separate substeps involved here: the reaction rate coefficients are updated every dt_chem and the state is updated every dt_therm.

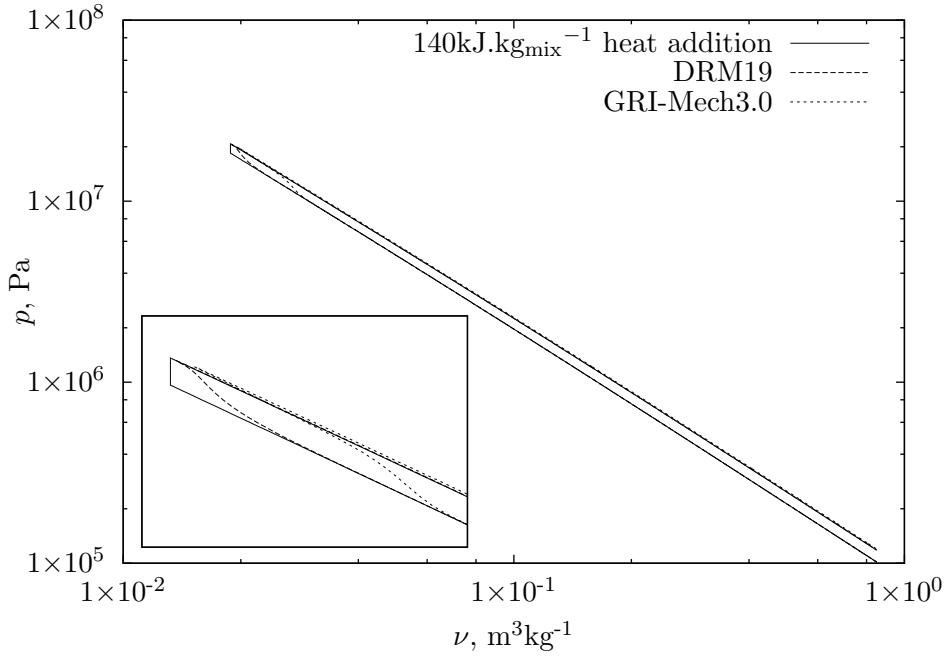


Figure 4.8: *p-ν diagram comparing finite-rate chemistry for DRM19 and GRI-Mech3.0 mechanisms with instantaneous heat addition of 140kJ.kg$_{mix}^{-1}$ in a free-piston compressor. The gas is thermally-perfect ventilation air at initial conditions of standard temperature and pressure. Engine parameters are m_p = 400kg, $u_{p,0}$ = 11.64m.s^{-1}, D = 0.1m and L_c = 4.0m. The compression ratio for these parameters is about 45:1 and the peak temperature (without combustion) is 1200K. Finite-rate chemistry is instrumental in answering the question of whether ventilation air can be combusted in a free-piston compressor since without finite-rate chemistry using a verified kinetic mechanism, the ignition point can only be approximated. It should also be noted that the two mechanisms given here predict different ignition delays, which is why the mechanism selection process was so thorough. The instantaneous heat addition (for the same initial conditions) shows a similar heat release, and thus the enthalpy of combustion can be said to be practically independent of the initial gas state.*

combustor has a peak temperature of 1200K which is sufficient for the residence time associated with the bore area, cylinder length and piston mass indicated.

When chemistry is included, the time to complete the simulation increases from 1m47s to 43m51s for DRM19 and 272m42s for GRI-Mech3.0 (although this varies depending on both the gas model used and the peak temperature reached) (see Listing 4.1 on page 73). Simulations were run on one core of an AMD Phenom X6 1090T CPU. Using `gprof` , this additional work was seen to be consumed by the evaluation of the reaction rates and, by extension, the evaluation of the rate coefficients, the gas state and associated functions. A comparison between two compression tests (one with and one without chemistry) is shown in Figure 4.9. Although it is not observable, reactions occur along the entire length of the stroke; they cannot be turned on

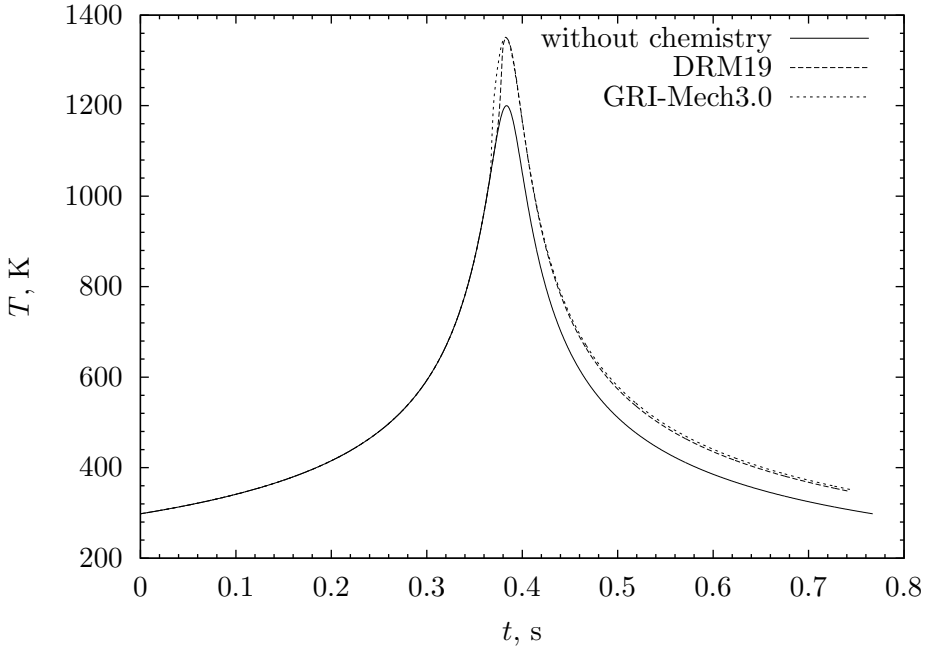


Figure 4.9: *Compression to 1200K with and without finite-rate chemistry. Reactions occur throughout the cycle, despite the fact they only release heat above 1200K. Turning the chemistry off at a lower temperature (to lower computational expense) yields incorrect results. Engine parameters are* m_p *= 400kg,* $u_{p,0}$ *= 11.64m.s*$^{-1}$*, D = 0.1m and* L_c *= 4.0m. The compression ratio for these parameters is 45.15:1 and the peak temperature (without combustion) is* $T_p = 1200K$.

part-way through compression (or expansion) to save computational work. The production and destruction of greenhouse species throughout this process is shown in Figure 4.10.

In a rapid compression machine, combustion does not always go to completion as the rebounding piston expands the reacting gas below combustion conditions. The burnt fraction of methane over the compression process $(1 \rightarrow 4)$ is defined here as

$$\mathrm{X}_{\mathrm{CH_4}}^{1\rightarrow 4} = \frac{\mathrm{X}_{\mathrm{CH_4}}^{1} - \mathrm{X}_{\mathrm{CH_4}}^{4}}{\mathrm{X}_{\mathrm{CH_4}}^{1}} \quad . \tag{4.6}$$

Similarly, the percentage reduction in GWP of an incomplete reaction is

$$\mathrm{GWP}_{\mathrm{mix}}^{1\rightarrow 4} = \left(1.0 - \frac{\mathrm{GWP}''}{\mathrm{GWP}'}\right) \times 100\%, \tag{4.7}$$

where GWP$''$ and GWP$'$ denote the global warming potential of the products and reactants respectively. This scalar allows for reactions that do not go to completion.

Using this scalar, the GWP reduction for ventilation air undergoing compression by a free piston is shown in Figure 4.11. The greatest reduction in GWP coincides with combustion completion. The full GRI-Mech3.0 mechanism was used here to show the production of nitrous oxides. While some nitrogen chemistry does occur, it is less than that for a CVFM PMR at the

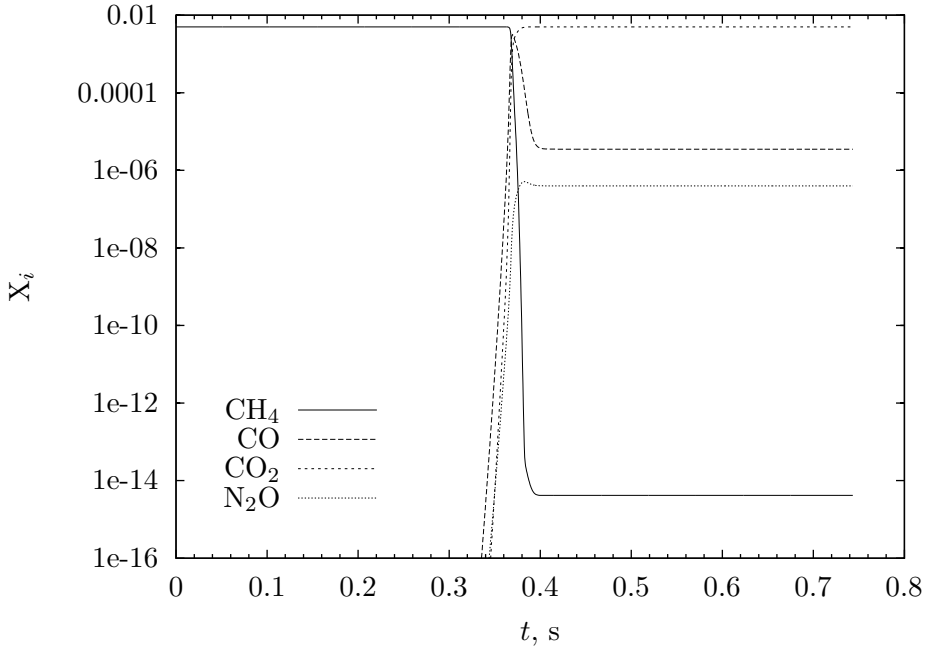


Figure 4.10: *Species production and destruction for ventilation air as it is raised to 1200K in a free-piston compressor. System initial conditions are* m_p *= 400kg,* $u_{p,0}$ *= 11.63m.s*$^{-1}$*,* D *= 0.1,* L_c *= 4.0m.*

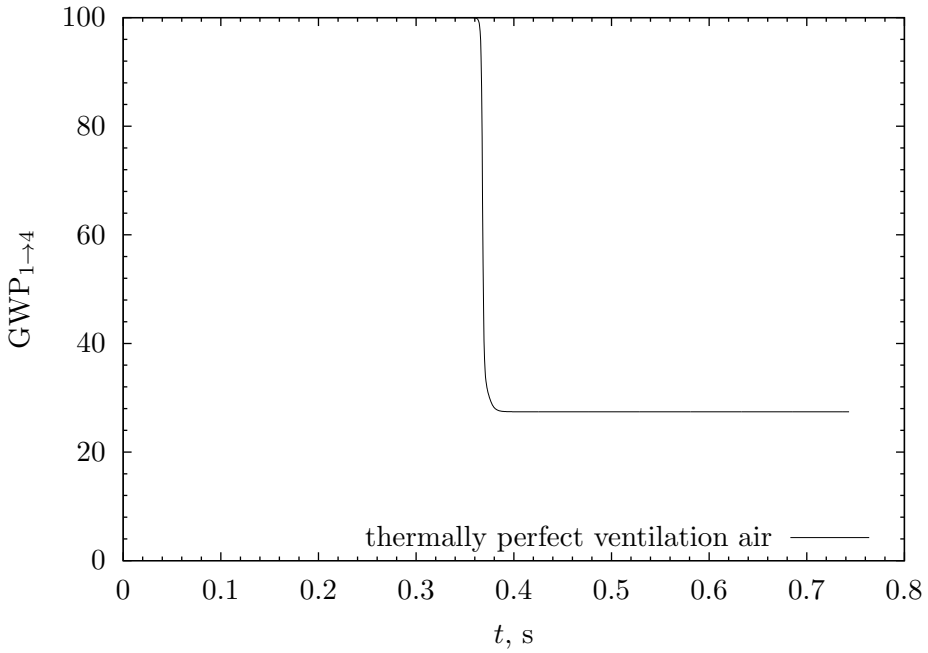


Figure 4.11: *Reduction in global warming potential of ventilation air as it is compressed to 1200K. System initial conditions are* m_p*=400kg,* $u_{p,0}$*=11.63m.s*$^{-1}$*,* D*=0.1,* L_c*=4.0m.*

2226 K and 1 atm see Figure 3.12) and does not greatly affect the reduction in global warming potential.

The combustion completion (also known as combustion yield) is dependent on the integral of gas temperature and residence time. To find the combustion temperature for a particular engine, compression tests over a range of peak temperatures were performed (see Figure 4.12). Figure 4.13 shows the GWP reduction for the same range of temperatures. For this engine, the

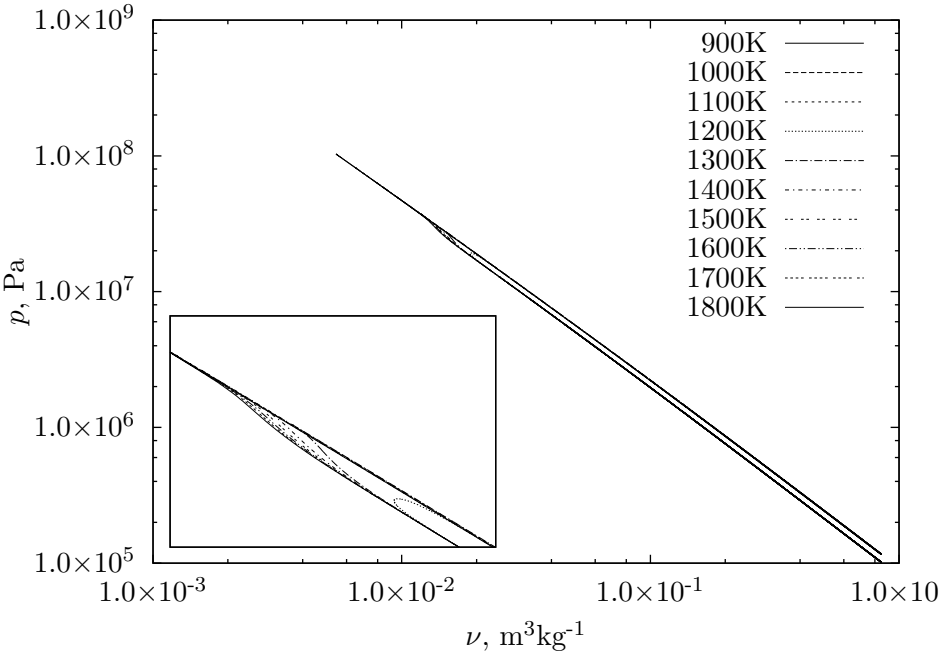


Figure 4.12: *p-ν diagram of ventilation air in a free-piston compressor reaching different peak temperatures using DRM19. As can be seen, for parameters m_p = 100kg, D = 0.2m and L_c = 5m, a peak temperature of between 1200 and 1300 K is required for complete combustion.*

combustion temperature is between 1200 and 1300 K. However, the same degree of combustion can be achieved at a slightly lower peak temperature and pressure for a longer residence time. This is demonstrated in Figure 4.14, where a range of piston masses used to complete the same engine cycle achieved different combustion yields. A lower pressure is desirable, since the combustion temperature must be reached without the pressure exceeding material limits. The residence time is largely proportional to $\left(\frac{m_p L_c}{A}\right)^{1/2}$ and so it may also be increased with heavier pistons, smaller cross-sectional areas and longer cylinder lengths. For a dual-piston type free-piston engine however, these variables also have an effect on cylinder exhausting and so cannot be changed independently. Finding the ideal operating condition is the topic of the parametric study in §8.3 on page 120.

4.4.1 Finite-rate chemistry of a real gas in a free-piston compressor

Real gas effects act to retard the rate of combustion. As discussed in §2.1, the compression ratio required to reach the combustion temperature of ventilation air may be high enough for

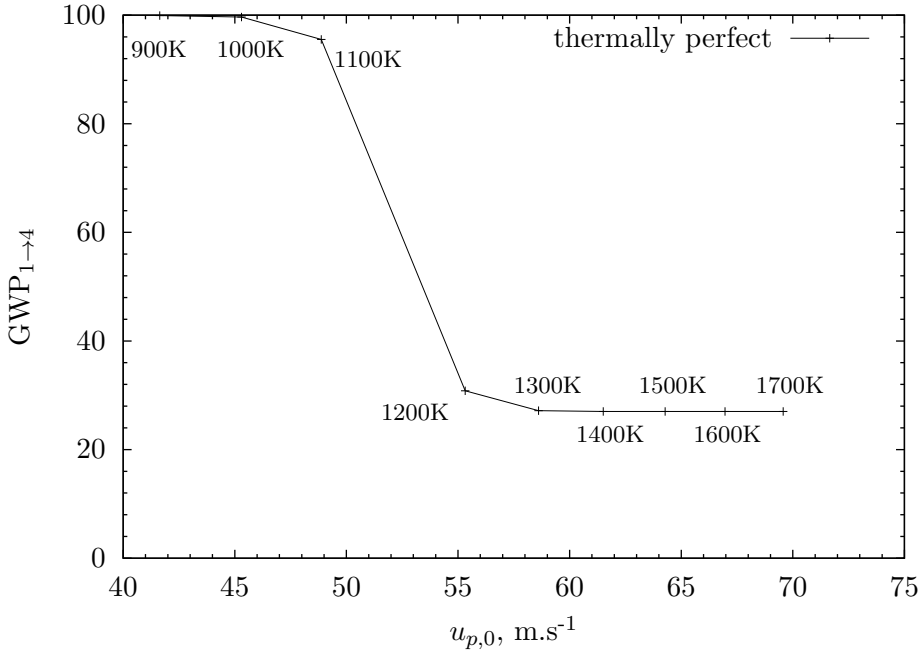


Figure 4.13: *GWP reduction of ventilation air in a free-piston compressor using a thermally perfect gas and DRM19. Temperatures correspond to peak temperature due to compression only.*

these effects to become significant. The effect of real gas models on the Otto cycle (with heat added instantaneously at the point of piston reversal) is shown in Figure 4.15. To determine the effect on finite-rate chemistry, a free-piston combustor was also run using real gas models. Combustion yield is shown in Figure 4.16 for a number of peak temperatures. Here it is made clear that real gas models retard the rate of reactions but not the products. An exception to this is when the reactions were frozen midway by the expanding piston for the peak temperature of 1200K. This case is undesirable: besides releasing less heat, exhaust gas will include more molecules of a higher GWP than CO_2 (such as CO) although the GWP will still be lower than that of ventilation air (see Figure 4.13). As such, real gas models will have no effect on the global warming potential only for a properly designed engine cycle where combustion has gone to completion.

It is conceivable that reactions may not go to completion in this way during the parametric study. To ensure this was not the case, a simulation with real-gas chemistry was performed for just the reference engine (see Chapter 8 on page 119). However, the computational expense of adding real gas effects to the chemistry throughout the study was not justified.

It has been shown at this point that it is possible to burn ventilation air in a free-piston compression device in order to reduce its GWP index. However, it is still unknown as to whether this can be done in a self-sustaining way, that is, by way of a reciprocating engine. The final stage in the engine cycle is that of cylinder exhausting. This stage is discussed in the next chapter.

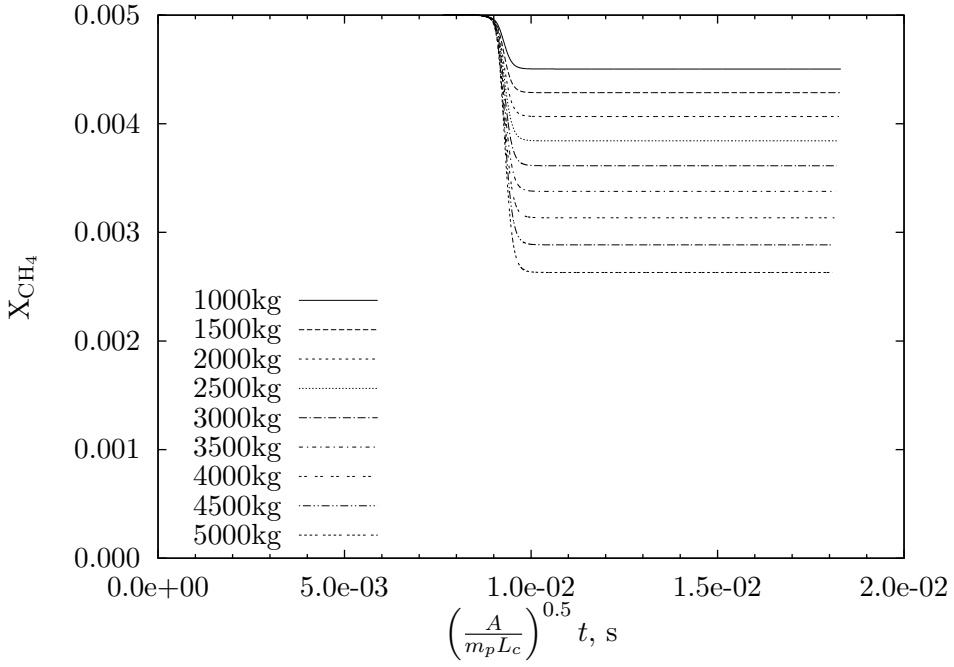


Figure 4.14: *The effect of piston mass on combustion yield for a peak temperature of 1100K. This scales linearly with residence time and exponentially with temperature (which is largely dictated by the initial velocity). Note that the piston period is largely dictated by the factor* $\left(\frac{A}{m_p L_c}\right)^{1/2}$ *which is related to the piston acceleration and the distance travelled.*

4.5 Dynamics of a dual piston type compressor

After the dynamics of a free-piston compressor were validated, the model was extended to two compression cylinders. Consider the dual piston type compressor of Figure 4.17. The energy method was used for this system, incorporating heat loss, friction and the application of an external force, to find the applicable state equations. To derive the system dynamics, Figure 4.17 was separated into components (see Figure 4.18). The acceleration of the piston is dependent on the sum of the forces acting upon it

$$\frac{du}{dt} = \frac{1}{m_p}\left[(p_0 - p_1)\,A + F_e - F_f\right] \tag{4.8}$$

where F_e is the force applied by the linear electric motor and F_f is the force due to friction. The required external force was determined by applying constraints on the cycle (see Chapter 7 on page 112). Investigation into and selection of appropriate heat transfer and friction models is addressed in Chapter 6.

Assuming the gas is at a homogeneous state $\mathbf{Q}_\infty$ far away from the cylinder wall, the conservation of energy equation (5.37) may be applied to the control volumes in Figure 4.18

$$de_0 = -dq_0 - p_0 A d\nu \tag{4.9a}$$

$$de_1 = -dq_1 + p_1 A d\nu, \tag{4.9b}$$

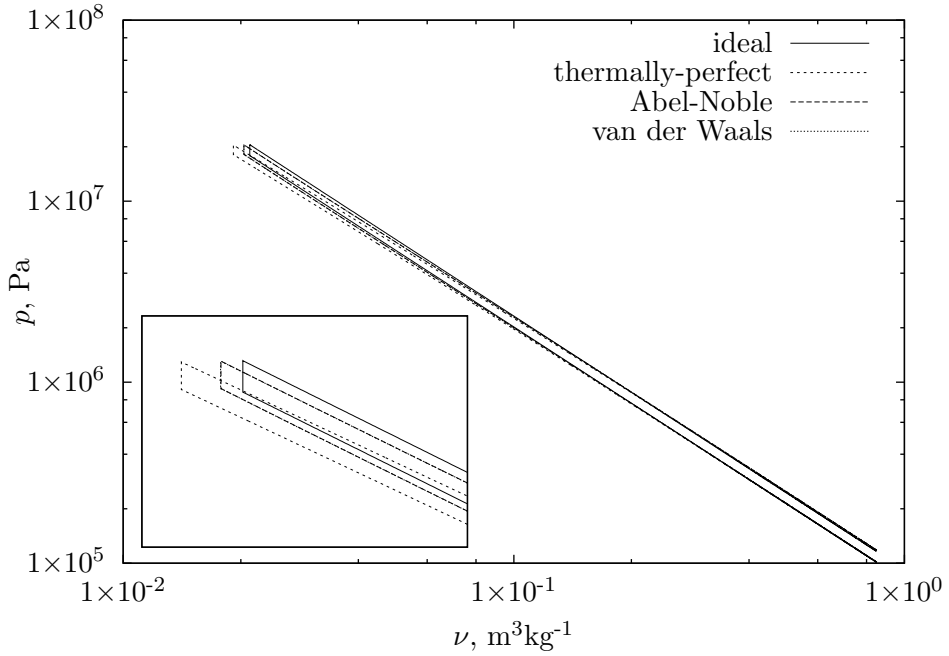


Figure 4.15: *The Otto cycle performed using air of different gas models, starting at the atmospheric state and adding 140kJ.kg^{-1} heat at the point of piston reversal. The thermally-perfect, Abel-Noble and van der Waals gas models have volumetric compression ratios of 24.5, 23.7 and 23.7 respectively with $m_p = 400kg$, $L_c = 4m$, $D = 0.1m$ and $u_{p,0} = 10m.s^{-1}$.*

where $q_w = Sdq_w{}''$, the wetted surface area is

$$S = \frac{\pi D^2}{2} + \pi DL \tag{4.10}$$

and the work done by the piston is

$$pd\nu = \frac{pAdx}{m_g}. \tag{4.11}$$

At this point, the state equations required to model a free-piston engine operating with a constant-mass of ventilation air have been described. They are prepared as

$$\left.\begin{aligned} \dot{x} &= u \\ \dot{u} &= \frac{1}{m_p}\left[(p_L - p_R)\,A - F_f - F_e\right] \\ \dot{w} &= F_e u \\ \dot{e}_L &= \frac{pAu}{m_{g,L}} - \dot{q}_R \\ \dot{e}_R &= -\frac{pAu}{m_{g,R}} - \dot{q}_R \end{aligned}\right\} \tag{4.12}$$

where the subscripts L and R denote the left- and right-hand cylinder respectively. The next stage in the engine cycle is the exhausting process which is investigated in the next chapter.

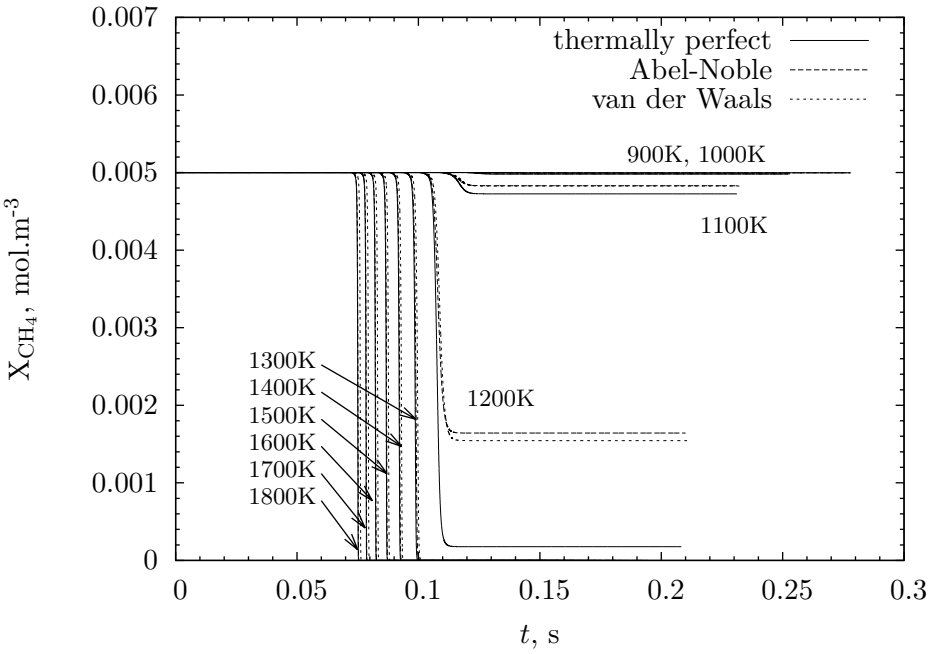


Figure 4.16: *Combustion yield in a free-piston compressor for peak temperatures of 900K-1800K (due to compression only) using DRM19, m_p =100kg, D=0.2 and L_c =5.0m. The largest discrepancy between real gas models occurs for a peak temperature of 1200K when reactions are frozen midway by the expanding piston. At temperatures higher than this, reactions go to completion for all gas models.*

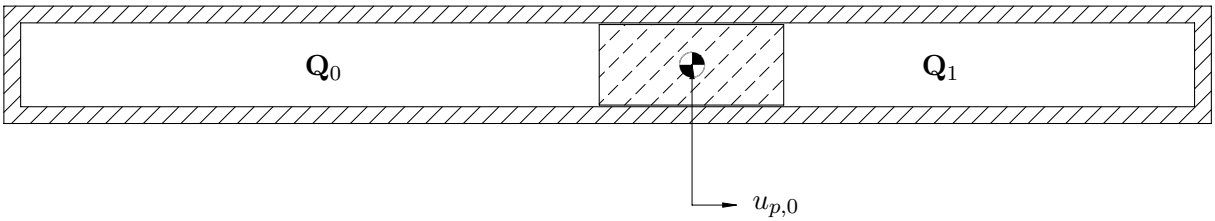


Figure 4.17: *A simplified dual piston type compressor.*

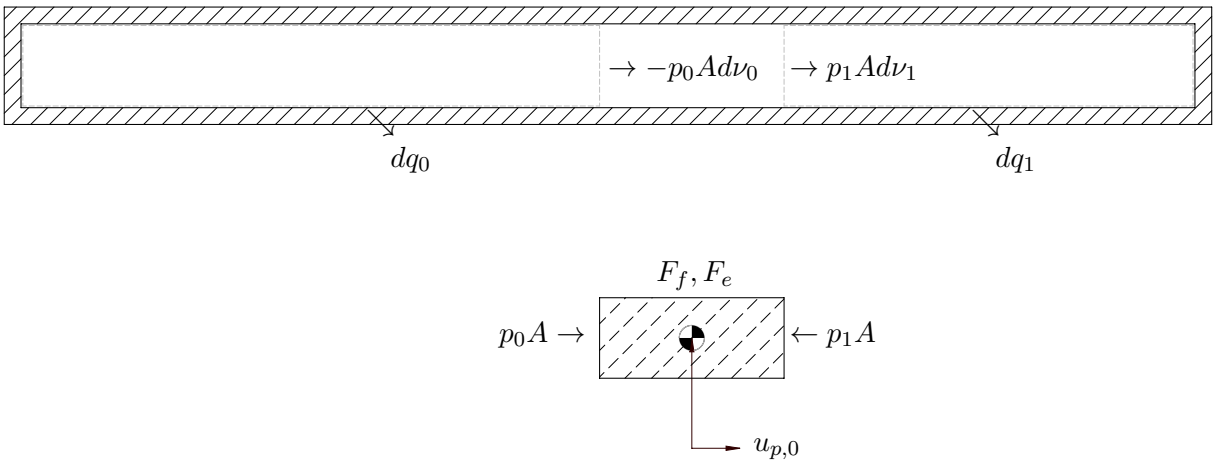


Figure 4.18: *The components of a simplified free-piston engine.*

4.A Chapter end notes

4.A.1 test_free_piston_compressor.py

```
from libfpe import *
from zero_finding import *
from numpy import array
import sys

def zero_function(u_p, args):
    fout, Q, T_ig, D, x_p, x_s, m_p, L_p, L_c, m_g, p_b = args

    d = get_global_data_ptr()
    y0 = vectord([x_p, u_p, Q.e[0], 0.0, 0.0])
    rval = test_free_piston_compressor(fout, y0, 1, Q, x_p, u_p, m_p, L_p, L_c,
                                       D, p_b, d.dh, d.t, d.tlast, d.dt_write,
                                       d.dt_sys, d.dt_therm, d.tol)

    x_s = rval[0]
    u_s = rval[1]
    e_s = rval[2]
    T = rval[3]
    args[5] = x_s
    print "%.7eK, %.7em/s, %.7em" % (T, u_p, x_s)
    return T - T_ig

def main(gasfile):
    datafile = "input.lua"
    #chemfile = "drm19"

    set_global_data(datafile)
    set_gas_model("../../input_files/"+gasfile+".lua")
    #set_reaction_update("../../input_files/"+chemfile+".lua")

    # gas
    g = get_gas_model_ptr()
    Q = gas_data()
    g.initialise_gas_data(Q)

    # air with 0.5% CH4
    X = 0.995*array([0.0, 0.21, 0.79])
    X[0] = 0.005
    molef = {'CH4':X[0], 'O2':X[1], 'N2':X[2]}

    #molef = {'O2':0.21, 'N2':0.79}
    set_molef(Q, g, molef)
    Q.T[0] = T_atm
    Q.p = p_atm
    g.eval_thermo_state_pT(Q)
    g.eval_transport_coefficients(Q)

    d = get_global_data_ptr()

    L_p = 0
    x_p = 0

    p_b = 0 # back pressure, atm
```

```
    fics = "ic-%s-%i.dat" % (gasfile, p_b)
    fout = open(fics, "w")
    fout.write("# m_p, m_g, T_ig, L_c, u_p, x_s, A\n")
    fout.close()
    x_s = 0.0

    u_p_v = []
    for m_p in d.m_p:
        for D in d.D:
            fname = ("ic-%s-%4.3fkg-%4.3fm.dat" % (gasfile, m_p, D))
            print "# writing to: %s" % fname
            fout = open(fname, "a")
            fout.write("# D, L_c, L_s, u_p, m_p, m_g, T_ig, y_CH4\n")
            fout.close()
            R = d.R[0]
            for T_ig in d.T_ig:
                L_c = D*R;
                x_R = L_c
                A = 0.25*PI*D*D
                V = A*L_c
                m_g = V*Q.rho

                print "# running %gK, %gm, %gkg, %gkg" % (T_ig, L_c, m_p, m_g)
                fout = "%s-%03.0fkg-%2.1fx%4.3fm-%04.0fK-%1.0fatm-bp.dat" % (gasfile,m_p,
                    L_c,D,T_ig,p_b)
                d.t = 0.0
                u_low = (1e2*T_ig*m_g/m_p)**0.5
                u_high = (3e3*T_ig*m_g/m_p)**0.5
                print("# u = [%g %g]" % (u_low, u_high))
                args = [fout, Q, T_ig, D, x_p, x_s, m_p, L_p, L_c, m_g, p_b*p_atm]

                u_p = muller_root(zero_function, u_low, u_high, tol=0.1, args=args)
                u_p_v.append(u_p)
                x_s = args[5]

                # print some data points
                fout = open(fics, "a")
                fout.write("%8.7e %8.7e %8.7e %8.7e %8.7e %8.7e %8.7e\n" % (m_p, m_g,
                    T_ig, L_c, u_p, x_s, A))
                fout.close()

            # you know at this point I do believe we can write our own input files.
            # and then maybe run them later with some chemistry.
            flua = "input-%s.lua" % (gasfile)
            write_lua_file(flua, [m_p], u_p_v, d.R, [D], d.T_ig)
            u_p_v = []

if __name__ == '__main__':
    # select from 'thermally-perfect-drm19', 'noble-abel-drm19', 'van-der-waals-drm19'
    main('thermally-perfect-drm19')
    print("#done.")
```

Listing 4.1: *Free-piston compressor test function.*

Chapter 5

Engine exhausting

The previous chapter examined piston dynamics using pre-charged cylinders. This chapter investigates the details of the exhausting process. Ventilation air must be forced into each cylinder under a pressure difference, either by precompression of the ventilation air or by expansion of the cylinder to a negative gauge pressure. The former is typically performed by a crankcase (as in a two-stroke engine) and the latter by a piston (as in a four-stroke engine). For a free-piston engine, it may be performed using one of three methods:

1. by a modified crankcase,
2. by a separate compressor or
3. by natural aspiration

with any combination of valves and/or ports. The method chosen was natural aspiration to avoid the complication of an additional process (with an associated efficiency). The level of modelling used for the exhausting stage was that of non-steady, isentropic, one-dimensional ideal gas flow in an infinite pipe of constant diameter.

5.1 Exhausting process

The exhausting process should aim to maximise the purity and density of the charge at valve closure for an exhausting efficiency of 100% [62].

Charge purity is measured by the *residual gas fraction*, which is defined as the mole fraction of products that remain after the exhausting stage. For this engine, the residual gas fraction acts to dilute the fuel content of the cylinder (which is undesirable given the low fraction of CH_4 in ventilation air) and the compression of exhaust products is wasted work.

After exhausting, although it is undesirable, some residual fraction of the exhaust gas will almost always remain due to mixing. The amount of fuel drawn into the engine is

$$m_{\mathrm{CH_4}} = \rho_0 L_c A \mathrm{X}_{\mathrm{CH_4}} \mathrm{X}_{va} \tag{5.1}$$

where X_{va} is the mole fraction of ventilation air in the cylinder and $\mathrm{X}_{\mathrm{CH_4}}$ is the mole fraction of CH_4 in ventilation air. The larger the entrained mass of CH_4 the better and thus, the larger the cylinder volume and charge density the better.

Charge density can only be increased through the use of a modified crankcase or separate compressor. A higher initial pressure has a correspondingly higher concentration of fuel in the cylinder. Mathematically,

$$[\mathrm{X}_{\mathrm{CH}_4}] = \mathrm{X}_{\mathrm{CH}_4} \frac{p}{\mathscr{R}T}. \tag{5.2}$$

For a fixed cylinder volume, precompression of the premixed fuel is desirable if the pressure work may be recuperated by regeneration. However the same effect (that is, increasing the mass of fuel) may be achieved by increasing cylinder length or cross-sectional area. As the stroke length is unconstrained and the engine was naturally aspirated, focus was instead placed on optimising the geometry of the cylinder.

Exhausting efficiency is effectively a measure of the pumping work. It is defined as the mass of ventilation air used to replace the combustion products, normalised by the mass of the combustion products.

$$\eta_{\mathrm{ex}} = \frac{m_{va}}{m_{g,0}} \tag{5.3}$$

Exhausting efficiency is largely dependent on the exhausting regime, limited here to either perfect mixing or perfect displacement.

In a free-piston compressor, the period of a cycle is roughly proportional to $\left(\frac{A}{m_p L_c}\right)^{\frac{1}{2}}$ for a given gas model. The longer the period, the more time available for cylinder exhausting. Too long, and excess pumping work is performed. Thus the aim is to find the shortest period that still allows for complete refuelling.

Uniflow port-port exhausting is the most effective exhausting method because it minimises the resistance to flow (or momentum) through the cylinder and approaches perfect displacement of the combustion products [63]. For a free-piston engine, a valve-valve configuration was required to allow for better control of the entrained mass of ventilation air. The inlet valve was a reed valve, and the exhaust valve a poppet valve. This was chosen to remove all but one of the controlling variables, namely, the timing of the poppet valve.

5.2 Gas dynamics

The Euler equations are a set of hyperbolic partial differential conservation equations [46]. Physically, they describe the rate of change of the conserved quantities of an inviscid compressible fluid. They are obtained by removing the viscous terms from the Navier-Stokes equations. The mass, momentum and energy of such a fluid are conserved due to three laws that are, in order, continuity, Newton's second law and the first law of thermodynamics. The characteristic features of inviscid compressible flow are finite waves that propagate pressure information through a fluid.

The one-dimensional Euler equations for conservation of mass, momentum, energy and species fraction may be expressed in integral form as

$$\frac{\partial}{\partial t} \iiint_{\vartheta} \mathbf{U} \, d\vartheta + \oiint_{S} \mathbf{F} \cdot \hat{\mathbf{n}} \, dS = \mathbf{H} \tag{5.4}$$

where $\mathbf{U}$ is the algebraic vector of conserved quantities, $\mathbf{F}$ is the inviscid flux vector and $\mathbf{H}$ is the source vector of mass, momentum, energy and species density

$$\mathbf{U} = \begin{bmatrix} \rho \\ \rho u \\ \rho(e + 0.5u^2) \\ \rho Y_i \end{bmatrix}, \quad \mathbf{F} = \begin{bmatrix} \rho\mathbf{u} \\ \rho u\mathbf{u} + p\hat{\mathbf{i}} \\ (\rho e + p)\,\mathbf{u} \\ \rho\mathbf{u}Y_i \end{bmatrix}, \quad \mathbf{H} = \begin{bmatrix} 0 \\ pA\hat{\mathbf{i}} - F_\mu\hat{\mathbf{i}} \\ Q \\ 0 \end{bmatrix}. \tag{5.5}$$

The column vector $\mathbf{H}$ contains source terms of mass, momentum and energy. For the homogeneous form of the Euler equations, these terms are zero. In its inhomogeneous form, they prescribe boundary sources such as mass addition, wall pressure and heat and work transfer.

For a cylinder undergoing exhausting, (5.4) becomes

$$\frac{\partial}{\partial t} \iiint_{\vartheta} \mathbf{U}\, d\vartheta = \begin{bmatrix} \rho A_v u_g \\ 0 \\ \rho A_v u_g e \\ \rho A_v u_g Y_i \end{bmatrix} \tag{5.6}$$

for a constant cylinder volume.

The state equations for a constant-volume cylinder undergoing exhausting are prepared as

$$\left.\begin{aligned} \frac{d}{dt}(\rho) &= [\rho_0 \mathbf{u}_0 A_{rv} + \rho_1 \mathbf{u}_1 A_{pv}]\,\frac{1}{\vartheta} \\ \frac{d}{dt}(\rho e) &= [\rho_0 \mathbf{u}_0 e_0 A_{rv} + \rho_1 \mathbf{u}_1 e_1 A_{pv}]\,\frac{1}{\vartheta} \\ \frac{d}{dt}(\rho Y_i) &= [\rho_0 \mathbf{u}_0 Y_{i'} A_{rv} + \rho_1 \mathbf{u}_1 Y_{i''} A_{pv}]\,\frac{1}{\vartheta} \end{aligned}\right\} \tag{5.7}$$

where the subscripts 0 and 1 denote value at the throat of the reed and poppet valves respectively, A is the instantaneous valve area and ϑ is the instantaneous cylinder volume.

5.2.1 Finite-waves in real gases

The bulk flow of a compressible fluid is driven by, and acts to reduce, a pressure difference. Disturbance information propagates through a compressible fluid by *finite waves*. From [46],

$$du = \begin{cases} -\dfrac{\nu}{a}dp & C_+ \text{ characteristic} \\ \dfrac{\nu}{a}dp & C_- \text{ characteristic.} \end{cases} \tag{5.8}$$

Substituting a form of the isentropic equation for an ideal gas,

$$dp = \frac{2\gamma}{(\gamma - 1)}\frac{p}{a}da \tag{5.9}$$

yields the general form for the Riemann invariants

$$du = \begin{cases} -\dfrac{2\gamma}{(\gamma - 1)}\dfrac{\nu p}{a^2}da & J_+ \text{ and} \\ \dfrac{2\gamma}{(\gamma - 1)}\dfrac{\nu p}{a^2}da & J_- \end{cases} \tag{5.10}$$

respectively. Consider now the unsteady isentropic expansion of an ideal gas down a constant area duct. The relation between the upstream and downstream Riemann invariants (along the C_+ characteristics) is

$$\cancelto{0}{u_0} + \frac{2a_0}{(\gamma-1)} = u + \frac{2a}{(\gamma-1)} \tag{5.11}$$

where the subscript 0 denotes the reservoir (stagnation) condition. The flow velocity is thus

$$u = \frac{2a_0}{(\gamma-1)}\left[1 - \left(\frac{a}{a_0}\right)\right] \tag{5.12}$$

which may be reduced to

$$u_{\max} = \left(\frac{2}{\gamma-1}\right) a_0, \tag{5.13}$$

which is the limiting velocity (if the gas were expanded to zero pressure). This effectively sets a limit to the proposed exhausting method. If the piston velocity were to exceed $u_{\max}$, a vacuum would form behind the piston. The general form for the speed of sound in a gas is

$$a^2 = -\gamma\nu^2 \left.\frac{\partial p}{\partial \nu}\right|_T \tag{5.14}$$

which reduces to

$$a^2 = \gamma RT \tag{5.15}$$

for an ideal gas. The derivatives for the covolume and van der Waals gas models are given in §2.A.3 on page 19.

The opening of a poppet valve in an engine results in a wave pattern somewhat similar to Sod's shock tube problem. Sod's shock-tube problem consists of a constant-area tube filled with a high- and low-pressure gas separated by a diaphragm [64]. The flow field at some time t after the diaphragm is removed (Figure 5.1) has an analytical solution comprised of an unsteady expansion, a contact discontinuity and a contact shock (also called a contact shock or slip-line). The solution to this problem will be used in the exhaust model. The details of this are included

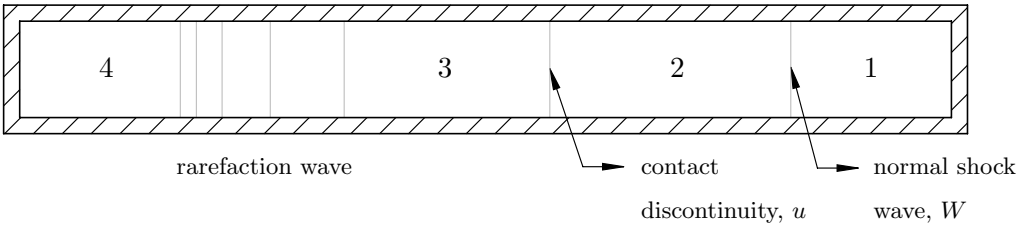


Figure 5.1: *Sod's shock-tube problem*

in §5.A.2 on page 86.

For validation (and to test the robustness of the model), the velocity of the contact discontinuity was plotted against a range of downstream to upstream pressure ratios (Figure 5.2). The velocity approaches the theoretical maximum as the downstream pressure approaches zero.

Consider a constant volume cylinder of gas being exhausted by an upstream reservoir of high pressure gas to the atmosphere. From (5.4) the rate of change of mass, momentum, energy and

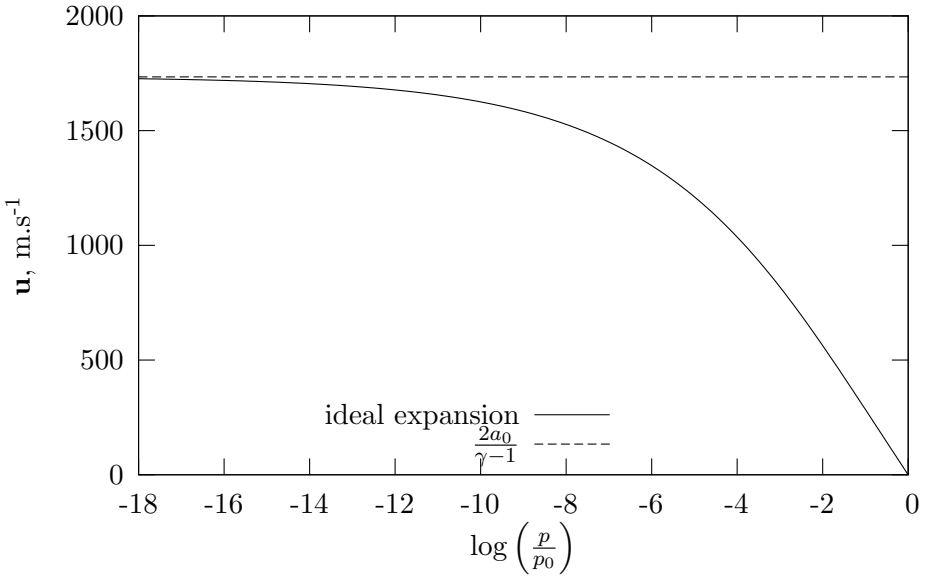


Figure 5.2: *Expansion of ideal air from the atmospheric state to a range of downstream pressures. This figure validates the implementation and tests for robustness over a large range of pressure ratios. Transport of species and energy have similar profiles.*

species fraction in the cylinder is

$$\frac{d}{dt}\mathbf{U}\vartheta = (\mathbf{F}_i + \mathbf{F}_e)\, S \tag{5.16}$$

where subscripts i and e denote the intake and exhaust valves respectively. The one-dimensional flux at an opening between two very long reservoirs is

$$\mathbf{F} = \begin{bmatrix} \rho\mathbf{u} \\ 0 \\ \rho\mathbf{u}e \\ \rho\mathbf{u}Y_i \end{bmatrix} \qquad \text{for } i \in [1 : n_{\text{sp}}]. \tag{5.17}$$

Using Sod's shock tube problem, the momentum may also be calculated (see Figure 5.3). Observe that this figure indicates that the fastest way to fill a cylinder is not to evacuate it, but instead to maintain a downstream-to-upstream pressure ratio of about 0.1.

5.2.2 Discharge coefficient

The mass flow rate was calculated using the finite wave equations for an ideal gas. However, these only capture the pressure effects of nozzle flow. To capture viscous effects, a *discharge coefficient* was applied to match the inviscid mass flow to the observed mass flow

$$c_d = \frac{\dot{m}_m}{\dot{m}} \tag{5.18}$$

where $\dot{m}_m$ is the measured mass flow rate.

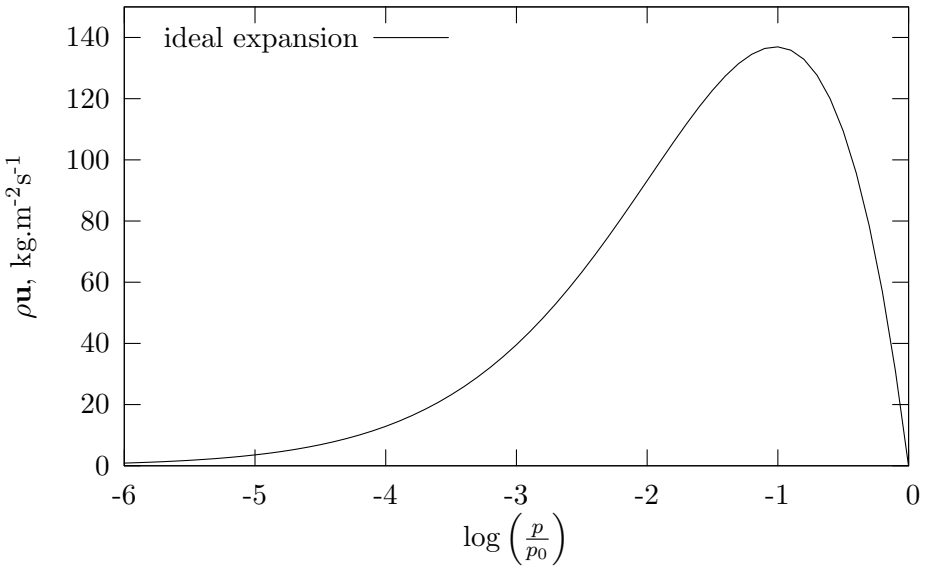


Figure 5.3: *Expansion of ideal air from the atmospheric state to a range of downstream pressures.*

When taking discharge coefficients from the literature, the same mass flow rate model should be adopted. Zhu and Reitz [65] used a one-dimensional model, which is similar to the model used here. They derived the discharge coefficients of Figure 5.4 using measurements from a four-valve Caterpillar engine. The Euler equations along with an exact Riemann solver (see §5.A.1 on page 85) were employed to solve the one-dimensional flow state caused by an opening valve.

5.3 Exhausting regimes

The incoming ventilation air and cylinder product gas do not remain separated by a contact discontinuity (as in Sod's shock tube problem) but instead undergo a degree of mixing. Unless short-circuiting occurs, the mixing regime is bounded by two simple models, namely *perfect-displacement* and *perfect-mixing*. Perfect-mixing is where the inlet ventilation gas instantaneously and completely mixes with the cylinder gas (implicit in (5.16)) and perfect-displacement is where the inlet ventilation gas does not mix with the cylinder gas (as in the ideal Sod solution). The perfect displacement exhaust model is

$$\frac{d}{dt}\left(\rho Y_i\right) = \left(\rho_0 \mathbf{u}_0 Y_{i'} + \rho_1 \mathbf{u}_1 Y_{i''}\right) \frac{A}{\vartheta} \tag{5.19}$$

where $Y_{i''}$ are the combustion product mass fractions. To implement this, the simulation code (see Listing B.6 on page 194) keeps track of the mass exhausted from the cylinder. When the initial mass of the cylinder has been removed, the code switches to the perfect mixing model. The mass contained within the control volume in this test is unsteady, so the complete first law of thermodynamics (5.34) was required to model the energy in a cylinder of changing volume. This is achieved by applying both equations (5.16) and (4.9) to the cylinder.

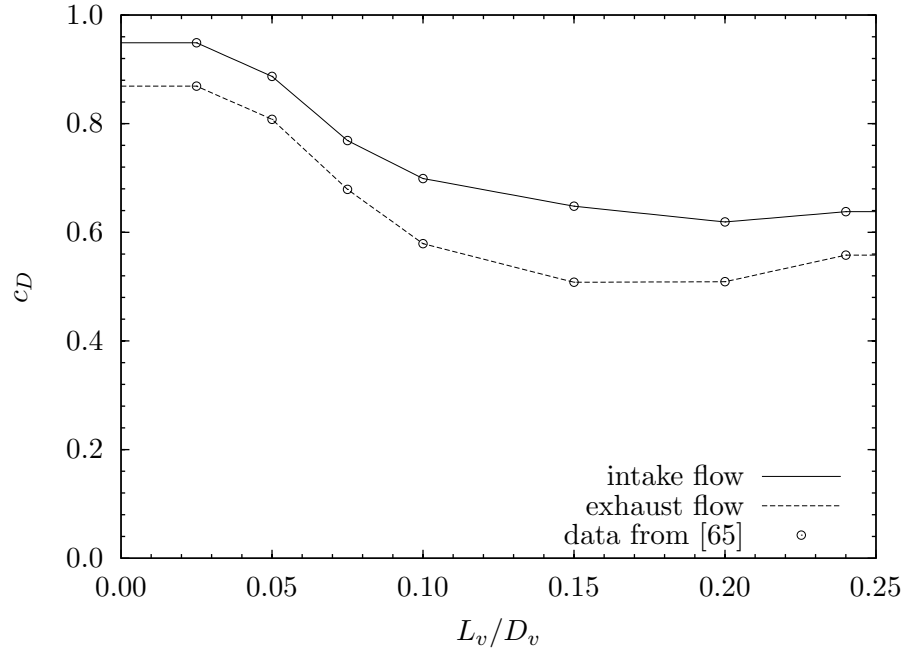


Figure 5.4: *Interpolated and measured discharge coefficients for a four-valve Caterpillar engine [65]. Coefficients are held constant outside the measured range.*

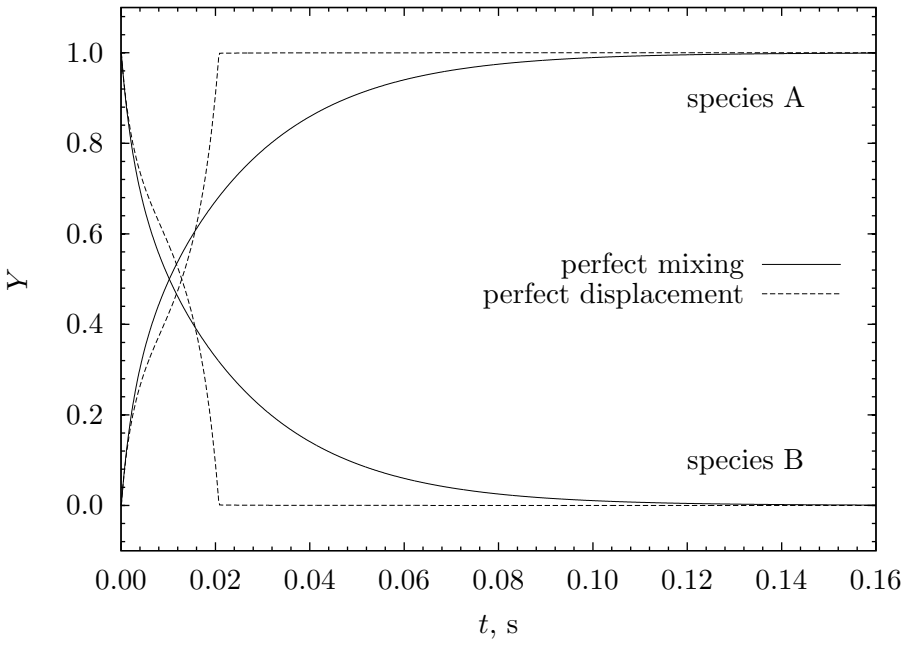


Figure 5.5: *Species 'B' is being exhausted with species 'A' using perfect-mixing and perfect-displacement regimes. The initial temperature of all volumes is 298.15K, with the upstream pressure being 2atm and the cylinder and downstream pressures being 1atm. The geometry is* $S = \vartheta = 1$.

5.4 Valve area

Valve area is a function of time. The faster the area can reach its maximum the better. In the chosen design, the intake valve is a reed valve that opens once the cylinder reduces below the atmospheric pressure and then closes again once it exceeds it. The over-expansion of the cylinder to approximately twice its original volume should allow for ample intake of ventilation air. Finally, during the compression stroke, the exhaust valve may be closed such that the desired mass of ventilation air is trapped in the cylinder (see Figure 5.6).

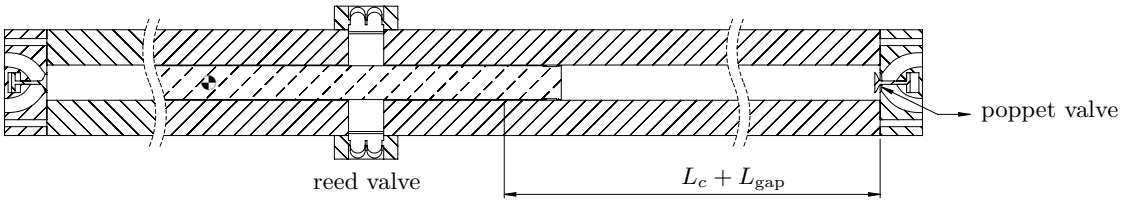


Figure 5.6: *Free-piston engine valves.*

For the reed valve, the maximum throat area is equal to the cylinder area, since the reed valve openings may be made as long as required. Because the piston ring travels over these openings, the fraction of the circumference taken up by the ports should be less than or equal to half the cylinder circumference. Taking the fraction to be half and the area to be at least that of the cylinder, the reed valve length is then

$$L_{rv} = 0.5D. \tag{5.20}$$

The maximum throat area of the poppet valve is determined by the cylinder head type (such as flat or peaked) and the ratio of the cylinder and poppet valve diameters. In this engine, the cylinder head will be flat to prevent collision by the unconstrained piston. For a single poppet valve there are two potential throats, one resulting from the annulus between the poppet valve head and cylinder, and one resulting from the poppet valve lift L_{pv} (5.21),

$$A_{xy} = \frac{\pi}{4}\left(D^2 - {D_{pv}}^2\right) \tag{5.21a}$$

$$A_{xz} = \pi D_{pv} L_{pv} \tag{5.21b}$$

where D is the cylinder diameter. Equating (5.21), the following conditions are found

$$\left.\begin{aligned} \frac{L_{pv}}{D_{pv}} &= 0.25 \\ \frac{D}{D_{pv}} &= \sqrt{2} \end{aligned}\right\} \tag{5.22}$$

indicating the largest throat area is reached when the poppet valve lift is greater than or equal to $0.25 L_{pv}/D_{pv}$.

5.5 Variable valve timing

Variable valve timing is an essential part of the control system as it allows for control over the purity and mass of the charge within the limits set by the system dynamics. For ventilation air, there are no benefits in heat regeneration from exhaust gas recirculation and compression of the combustion products is a waste of piston work [1].

The control variables used for valve actuation are the cylinder pressure and the entrained mass of ventilation air. Actuating valves according to cylinder pressure allows for a better control of the flow rate. Actuating valves according to the entrained mass allows for control of the compression ratio. The nominal valve actuation over the exhausting process is enumerated:

1. All valves remain closed whilst cylinder pressure is above atmospheric pressure during the expansion stroke.

2. As the cylinder pressure drops below the atmospheric pressure, the reed valve opens, allowing the piston to draw in ventilation air.

3. On the compression stroke the pressure rises above the atmospheric pressure and the reed valve closes whilst the poppet valve is opened.

4. The poppet valve is then closed as the entrained mass of ventilation air approaches the target mass.

The logic used for this operation is shown in Figure 5.7. A caveat to this method is that the

```
input  : dt, L, u_p, p, m_g
output: incremental valve movement

if u_p < 0 then
    if p < p_atm then
        open_intake_valve(dt)
else
    if p > p_atm then
        //we have taken in all the fuel possible at this point
        close_intake_valve(dt)
        if m_g > m_g,0 then
            open_exhaust_valve(dt)
        else
            close_exhaust_valve(dt)
```

Figure 5.7: *Valve timing logic for the right hand cylinder*

entrained mass which is a difficult value to quantify experimentally. While it cannot be directly measured, the relation $m_g = \frac{pV}{RT}$ may be used to evaluate it.

In addition to controlling the trapped mass of ventilation air, variable valve timing may be used to increase the purity of the charge. Assuming a perfect-displacement exhausting regime, the valve can be actuated so that just the combustion products are exhausted during compression, increasing the mole fraction of ventilation air in the cylinder. *Valve delay* is defined here to be the time delay between port close and valve close during the compression stroke which may be used to achieve this.

Actuation of the poppet valves is not simulated but is assumed to be pneumatically actuated with a lift velocity of $10 m.s^{-1}$. If valve actuation is slow compared to piston speed, this can have a significant effect on engine operation. In a real engine, the poppet valves should be actuated by a single-acting cylinder with a spring return (such that they fail in the closed position).

At this point the program includes all the operations required to complete a full engine cycle (namely, compression, combustion, expansion and exhausting), however, before answering the question of whether this process can be done in a self-sustaining way using ventilation air as the working fluid, the effect of heat transfer and friction must be investigated.

5.A Chapter end notes

5.A.1 Riemann flux solvers

An initial-value problem for a nonlinear conservation law with piecewise-constant data is referred to as a Riemann problem [66]. In computational fluid flow the Riemann problem originates from the discretisation of the domain space which produces a piecewise-constant flow state, $\mathbf{U}_{\mathbf{x}}^{\mathbf{t}}$. Consider two such adjacent flow states $\mathbf{U}_{\mathbf{i}}^{\mathbf{j}}$ and $\mathbf{U}_{\mathbf{i+1}}^{\mathbf{j}}$ depicted in Figure 5.8. The piecewise-

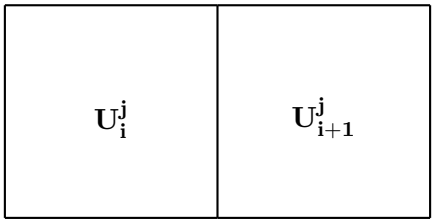


Figure 5.8: *A piecewise-constant flow state*

constant flow state may be expressed mathematically as

$$\mathbf{U}(\mathbf{x}, \mathbf{t_j}) = \begin{cases} \mathbf{U}(\mathbf{x_i}, \mathbf{t_j}) & \text{if } \; x < x_i + \Omega h, \\ \mathbf{U}(\mathbf{x_{i+1}}, \mathbf{t_j}) & \text{if } \; x \geq x_i + \Omega h, \end{cases} \tag{5.23}$$

where h is the node spacing and Ω is a number between 0 and 1 that determines the location of the discontinuity between the two states. Now consider one possible state after a small timestep Δt of Figure 5.9. The three possible wave states that may result from this problem are the

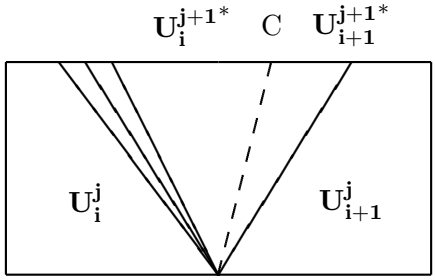


Figure 5.9: *An $x-t$ plot of one possible flow state after timestep Δt*

rarefaction-wave, the contact-discontinuity and the shock-wave, although other combinations are possible. These waves are created by the pressure difference between $\mathbf{U}_{\mathbf{i}}^{\mathbf{j}}$ and $\mathbf{U}_{\mathbf{i}}^{\mathbf{j+1}}$, hence the respective states at the next time interval find pressure equilibrium, that is

$$\begin{aligned} p_i^{j+1^*} &= p_{i+1}^{j+1^*} = p^* \;\; \text{and} \\ u_i^{j+1^*} &= u_{i+1}^{j+1^*} = u^* \end{aligned} \tag{5.24}$$

where u^* is the velocity of the contact discontinuity, given by (5.12).

The keen observer will recognise this as Sod's shock tube problem, and the procedure outlined previously as the exact solution. However, due to the iteration involved, this procedure is

expensive, and approximate Riemann solvers are commonly substituted in computational fluid dynamics. The number of approximate Riemann solvers that exist in the literature is vast. Due to the relatively few times the code will be required to solve this problem (compared to a 2- or 3-dimensional model), the exact solution will be used (along with programming techniques to reduce the computational cost).

5.A.2 Sod's shock tube [64]

The flow field of Sod's shock tube at some time after the diaphragm is removed is determined entirely by the initial diaphragm pressure ratio and the gas model. The energy change across a shock is provided by the Hugoniot equation

$$e_2 = e_1 + \frac{1}{2\rho_1}\left(p_1 + p_2\right)\left(1 - \frac{\rho_1}{\rho_2}\right) \ . \tag{5.25}$$

For a given density ratio $\frac{\rho_1}{\rho_2}$ there exists a unique solution for e_2 and p_2, where $p_2(e_2, \rho_2)$ is provided by the relevant equation of state.

Ideal gas

The solution for an ideal gas is outlined in [46], but the procedure used to find the pressure ratio is shown here for completion. Substituting the ideal equation of state, $p_2 = \rho_2 e_2 \left(\gamma - 1\right)$, into (5.25), the pressure ratio across the shock may be found using a zero-finding method with the zero function

$$f = \frac{p_4}{p_1} - \frac{p_2}{p_1}\left(1 - T_2\right)^{-\frac{2\gamma_4}{\gamma_4 - 1}} \tag{5.26}$$

where

$$T_2 = \frac{\left(\gamma_4 - 1\right)\left(\frac{a_1}{a_4}\right)\left(\frac{p_2}{p_1} - 1\right)}{\left\{2\gamma_1\left[2\gamma_1 + \left(\gamma_1 + 1\right)\left(\frac{p_2}{p_1} - 1\right)\right]\right\}^{0.5}} \tag{5.27}$$

within the range $p_2/p_1 \in [0 : p_4/p_1]$.

In the free-piston engine model, the bisection method was first used as the zero-finding method. Later, however, Muller's method was implemented due to its speed since the pressure ratio needs to be evaluated at *least* once per timestep during the exhausting portion of the stroke.

Real gas

Considering the physics of a real gas, the presence of intermolecular attractive forces acts to retard expansion. This has been observed to the reduced driver performance of hypersonic wind tunnels [25]. Here we investigate the numerical procedure for solving Sod's shock tube for a real gas. Since $e_2(T_2)$ for a thermally-perfect gas and $e(T_2, \rho_2)$ for a real gas, the whole flow state of Sod's shock tube must be used to evaluate the zero function.

To find the velocity behind the shock, take the continuity and momentum equations (in the shock frame of reference)

$$\rho_1 W = \rho_2\left(W - u_2\right) \tag{5.28a}$$

$$p_1 + \rho_1 W^2 = p_2 + \rho_2\left(W - u_2\right)^2 \tag{5.28b}$$

where W is the shock speed. Substituting (5.28a) into (5.28b) gives

$$W = \left[\frac{1}{\rho_1} \frac{(p_2 - p_1)}{\left(1 - \frac{\rho_1}{\rho_2}\right)} \right]^{\frac{1}{2}} . \tag{5.29}$$

Given continuity, $\frac{\rho_1}{\rho_2}$ and W

$$u_2 = W \left(1 - \frac{\rho_1}{\rho_2}\right) \tag{5.30}$$

and the conditions across the contact-discontinuity are

$$\left.\begin{aligned} p_2 &= p_3 \\ u_2 &= u_3 \end{aligned}\right\} . \tag{5.31}$$

Taking the general form for entropy (2.8c) and the C_+ characteristic (5.8), the following conditions must hold across an isentropic rarefaction wave:

$$dT = \frac{\nu}{c_p} dp \tag{5.32a}$$

$$du = -\frac{\nu}{a} dp \tag{5.32b}$$

assuming c_p is constant over a small step. With (5.32) we can integrate from $p_4 \to p_3$ to find u_3. The correct value for $\frac{\rho_1}{\rho_2}$ may then be found using a zero-finding method with the zero function $f = u_3 - u_2$. Once $\frac{\rho_1}{\rho_2}$ has been found, the entire solution may then be evaluated (remembering to step across the rarefaction wave using du this time).

One observation that can be made is that the speed of sound in a real gas is slightly faster than for an ideal gas. For example, take the sound speed for an Abel-Noble gas:

$$a^2 = \frac{\gamma p \nu^2}{\nu - \nu_0} \quad . \tag{5.33}$$

This is greater than for a perfect gas by the factor $\frac{\nu}{(\nu - \nu_0)}$. Thus, the tip of the expansion wave should be faster than that for an ideal gas by this factor.

The program was first validated by comparing the solution using the ideal gas procedure to the solution using the real gas procedure with an ideal gas model (see Figure 5.10). The real gas procedure was then applied using the gas models in Section 2.1.1 (see Figure 5.11).

For Sod's shock tube problem with a real gas, the following observations can be made:

- the shock speed is slower
- the gas velocity is faster and
- the gas density between the shock and the contact discontinuity is lower.

The result of these effects is to retard the gas momentum and further increase the entropy rise across the shock. Finally, when setting up Sod's shock tube problem with real gases, it is important to specify properties of density and pressure since two gas models at the same temperature and pressure will result in different masses.

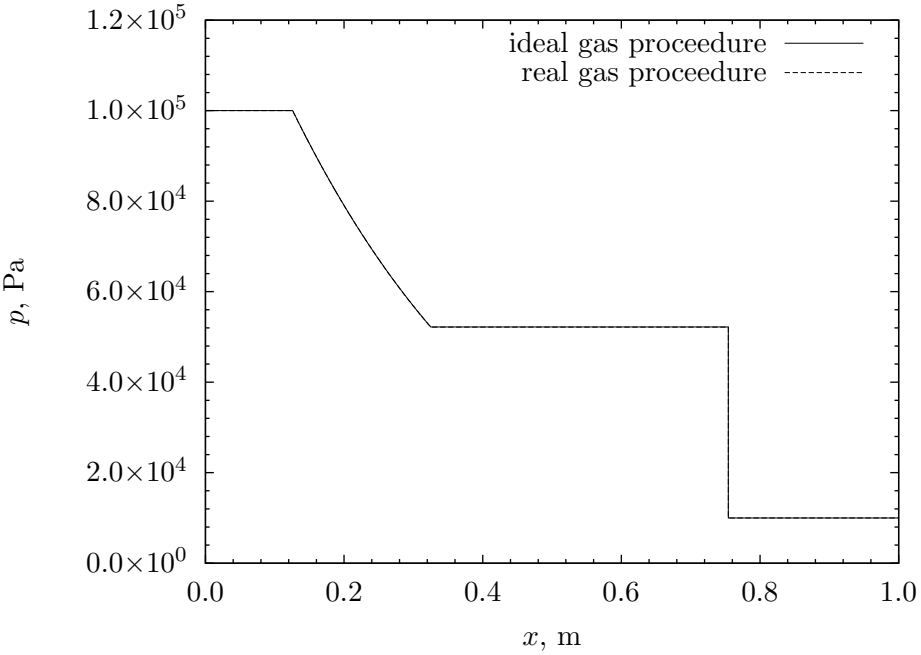


Figure 5.10: *Ideal solution to Sod's shock tube, generated using two numerical procedures: one specific to an ideal gas and one which may also be used for a real gas. There is error in both procedures (due to the iteration) which is controlled by tolerances.*

5.A.3 Full form of conservation of energy

The first law of thermodynamics is the conservation of energy, which states that the rate of change of energy in a system is equal to the sum of the rate of energy change within the volume and the energy transferred across its boundary via heat, work and mass [29].

$$dQ - dW + \sum \dot{m}_i \theta_i - \sum \dot{m}_e \theta_e = dE + m d\mathrm{ke} + m d\mathrm{pe} \tag{5.34}$$

where Q is the heat transfer to the gas, W is the work done by the gas,

$$\theta = h + \mathrm{ke} + \mathrm{pe}, \tag{5.35}$$

and the kinetic and potential energies are

$$\left.\begin{aligned} \mathrm{ke} &= \frac{1}{2}u^2 \\ \mathrm{pe} &= gz \\ d\mathrm{ke} &= \frac{1}{2}\left({u_1}^2 - {u_0}^2\right) \\ d\mathrm{pe} &= gdz \end{aligned}\right\} \tag{5.36}$$

respectively. For a fixed mass, (5.34) reduces to

$$dq - dw = de + d\mathrm{ke} + d\mathrm{pe}. \tag{5.37}$$

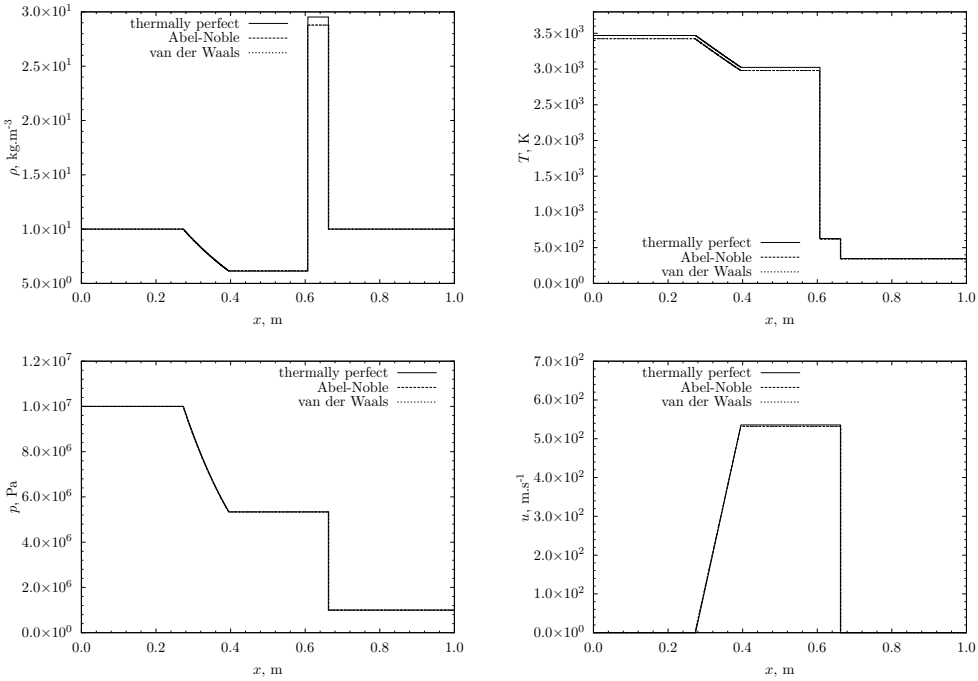


Figure 5.11: *Real gas solutions to Sod's shock tube problem. For this test* $p_4 = 1{\times}10^7\,Pa$, $\rho_4 = 10\,kg.m^{-3}$, $p_1 = 1{\times}10^5\,Pa$, $\rho_1 = 10\,kg.m^{-3}$. *Using the conditions for the classic shock tube problem (that is* $\rho_4 = \rho 1 = 1 kg.m^{-3}$*) resulted in temperatures too low in the region between the shock wave and the contact shock and caused numerical errors. The lower density in this region and the lower total velocity clearly show the effect of the real gas models. Note that the thermally-perfect model does not differ from the ideal solution.*

Chapter 6

Engine losses

6.1 Friction

Friction refers to forces between loaded surfaces in relative motion. It always acts to oppose the motion such that the vector force on the piston is

$$\mathbf{F}_f = -\hat{\mathbf{u}} F_f \tag{6.1}$$

where $\hat{\mathbf{u}}$ is the normalised velocity vector and F_f is the friction force magnitude.

Three regimes exist for lubricated sliding surfaces: boundary, mixed and hydrodynamic [67]. In the boundary regime, the surfaces are in contact and friction is the result of their interference. The mixed regime exists when the average film thickness is of the order of the apparent surface roughness and exhibits characteristics of both boundary and hydrodynamic regimes. In the hydrodynamic lubrication regime, friction is dependent on the oil viscosity and the velocity profile. As the relative speed increases, the friction first reduces as the surfaces separate due to hydrodynamic effects, then increases due to viscous effects [68]. The viscosity in turn is affected by the oil temperature and pressure which change throughout the cycle.

For this engine, three rings will be used per cylinder: a compression ring, a bearing ring and an oil ring. The compression ring is of chevron type with a thickness of 5mm and used to prevent blow-by of the high pressures. A bearing ring should be used to support the weight of the piston, but its thickness should not affect the normal force and so it was ignored for modelling purposes. The oil ring was assumed to carry no normal force and so was also ignored with regards to ring contact area. This design (not including the oil ring) is similar to that used for the piston in the T4 shock tunnel at the Mechanical Engineering Department, University of Queensland. For the current engine, a mixed model will be used for the friction factor.

Consider a piston of mass m_p supported by a ring set of this type. The friction force of each ring is proportional to the normal force by the friction coefficient f such that

$$F_f = f\left[\sum_{i=1}^{n_r}(A_{r,i}p_{r,i}) + m_p g\right] \tag{6.2}$$

where $A_{r,i}$ is the compression ring contact area and $p_{r,i}$ is the contact pressure of ring i, f is the friction factor, n_r is the number of rings and g is the acceleration due to gravity. Thus, the contact force is the combination of both the force of the gas pressure forming a seal with the

cylinder wall and the force due to the weight of the piston. In conventional engines, instead of the piston weight there is a force applied by the crankshaft pushing the piston against the cylinder wall. For chevron-type rings, the contact pressure $p_{r,i}$ is the sum of the ring tensile stress and the gas pressure acting behind the ring

$$p_{r,i} = p_i(t) + \sigma_t. \tag{6.3}$$

Some small tensile stress is required for the ring to make contact with the cylinder wall when there is no gas pressure, but for the purposes of modelling this was assumed to be zero.

Assuming the friction due to piston weight is small compared to the ring contact pressure, the stroke length with friction may be found using a similar method to (4.3)[1],

$$L_s = L_0 - \left[{L_0}^{(1-\gamma)} - \frac{m_p\,(1-\gamma)}{2p_0 {L_0}^{\gamma}\,(A + fA_r)} {u_{p,0}}^2 \right]^{\frac{1}{(1-\gamma)}}, \tag{6.4}$$

where A_r is the total piston ring area. From this it can be seen that the addition of friction acts to reduce the stroke length for a given initial piston velocity. Conversely, to achieve a particular stroke length, the initial piston velocity must be increased. The amount it must be increased by, for a given engine geometry, is $\left(1 + f\frac{A_r}{A}\right)^{\frac{1}{2}}$ (see Figure 6.1). Thus, for a given engine, the effect of friction can be reduced principally by reducing the ring surface area and friction coefficient.

6.1.1 Mixed friction coefficient

The friction coefficient in a mixed model is a linear combination of boundary and hydrodynamic friction models,

$$f = \alpha f_b + (1-\alpha)\, f_h \tag{6.5}$$

where α is a function of the bearing parameter, $\frac{\mu u_p}{p}$, which varies from unity to zero. The transition point depends on variables such as the quantity of lubricant, the ring and bore materials, the operating temperature and the age of the bearing.

Boundary model

A constant friction coefficient of 0.1 adequately models the boundary friction for cast iron compression rings like the ones used here.

Hydrodynamic model

Hydrodynamic pressure is a dynamic force resulting from the viscosity of the lubricant. For hydrodynamic lubrication to occur, two surfaces require sufficient relative motion and relative inclination (or equivalent geometry) such that a load carrying film is generated. Similar to modelling of the boundary layer for heat transfer, a detailled approach includes modelling of the boundary layer using an integral boundary layer method or a numerical method. Without experimental data, these models merely shift the unknown variables, for example, from friction coefficient to apparent roughness.

[1] While the effect of the piston weight could be included in this equation, to do so would make the analytical solution intractable.

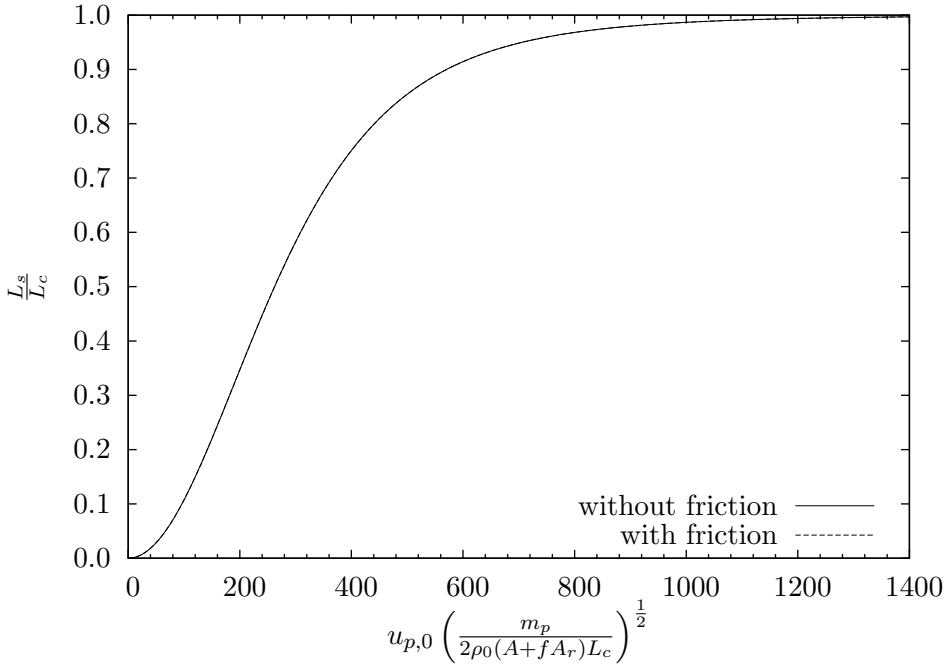


Figure 6.1: *Note the new normalised velocity, which includes the friction factor and ring contact area. For a frictionless compressor, these extra terms go to zero leaving the normalization factor seen previously (for example, in Figure 4.4). This new factor shows that friction may be overcome by increasing the initial piston velocity by a predictable amount. For a simple boundary friction model (where f is constant) this amount is $\left(1+f\frac{A_r}{A}\right)^{\frac{1}{2}}$. This provides a useful first-pass guess when estimating the additional required piston speed.*

McGeehan [69] conducted a literature review of compression ring friction. Experiments have shown that liner lubrication is predominantly hydrodynamic with localised contact near the point of piston reversal. A hydrodynamic friction coefficient model is also required given the potentially high ring velocities (see §4.2). In [69], the presence of a continuous oil film was detected by the measurement of electrical resistance and capacitance between the piston ring and the liner (low values corresponding to direct contact). The factors affecting the friction include the ring loading, the number of rings and their respective contact areas and operating profiles (after wear-in). The factors affecting the friction coefficient include dynamic viscosity, ring velocity, thickness, profile and effective pressure. The general form of the friction coefficient was found to be

$$f_h = c_1\left(\frac{\mu u_p}{p_r b}\right)^{c_2} \tag{6.6}$$

where μ is the dynamic viscosity (approximately 0.222 for SAE30 oil at 300K), and typically $c_1 \in [1:5]$ and $c_2 \in [0.33:0.66]$ depending on the ring profile [69]. The lowest viscosity oil that still generates a load carrying film for the majority of the stroke is ideal. A friction coefficient curve using a ring with a parabolic profile (that is, $c_1 = 4.8$ and $c_2 = 0.5$) was deemed

sufficient as a first approximation. The resultant function for the mixed friction coefficient is shown along with experimental data from [69] in Figure 6.2. The transition between boundary

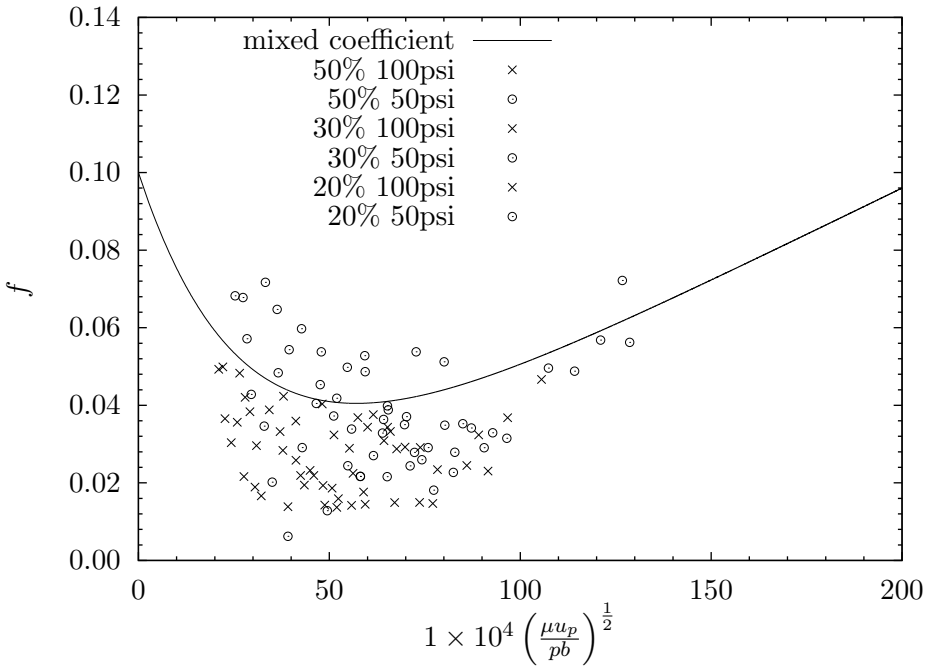


Figure 6.2: *A mixed friction coefficient* (6.5) *with experimental data from [69]. For this function,* $b = 0.03m$, $\mu = 0.222Pa.s$ *and the constants* $c_1 = 4.8$, $c_2 = 0.5$ *correspond to a parabolic profile.*

and hydrodynamic friction and the gradient of the hydrodynamic curve depends on the ring profile: a flat profile contacts the cylinder liner only at low speed due to squeeze action and generates a thin film at mid-stroke due to wedge action, whereas a curved profile contacts the cylinder liner at a higher speed due to squeeze action and generates a thick film at mid-stroke due to wedge action. The transition function used here was

$$\alpha = \exp\left(-300\nu\right) \qquad \text{for } \nu \in [0:1] \tag{6.7}$$

where $\nu = \left(\frac{\mu u_p}{p_{\text{atm}} b}\right)^{\frac{1}{2}}$.

Along with friction, heat transfer was also investigated for this engine.

6.2 Heat transfer in a reciprocating engine

In the 1980s there were attempts to create an "adiabatic" engine, with the promise of high fuel-efficiency. Insulated walls were used to increase the wall surface temperature, reducing the bulk gas-wall temperature difference in the hope of reducing heat transfer. Contrary to expectations, average heat transfer increased [70].

Heat transfer in a reciprocating engine is a complex phenomenon. Accurate modelling of the temperature gradient at the wall requires modelling of the compressible boundary layer equations (even though the piston velocity was small compared to the speed of sound) and can include effects such as rotational flow, turbulence and heat generation. As such, the assumptions made for the convection equation $q_w{}'' = h\,(T - T_w)$ do not apply to a rapid compression device. These assumptions include:

- incompressibility in the near-wall region,
- steady-state conditions in the freestream fluid and surface and
- the subsequent similarity between velocity and thermal boundary layers (Reynold's analogy).

In the adiabatic engine, increasing the wall surface temperature thinned the laminar boundary layer which correspondingly increased the surface temperature gradient and consequently the heat transfer rate. This explained the unexpected increase in average heat transfer. Due to the very low combustion enthalpy of ventilation air, the adiabatic engine is again pursued here.

Consider a compressible fluid element in motion close to the cylinder wall. The energy is determined by the internal energy, the conduction of heat, the convection of heat with the motion and the generation of heat through friction, as well as work due to due to expansion (or compression) as the volume is changed. The quantity of heat dq added to the volume externally and through friction serves to increase its internal energy and to perform expansion work:

$$dq = \underbrace{c_\nu dT}_{\text{internal energy}} + \underbrace{p d\nu}_{\text{work}} \,. \tag{6.8}$$

The quantity of heat added due to conduction obeys Fourier's law,

$$q_w{}'' = -k\nabla T \,. \tag{6.9}$$

Now consider an element of the cylinder wall adjacent to the fluid element. The heat lost from the fluid element through the shared surface is gained by the wall element. The temperature profile across the wall element develops in accordance with the diffusion equation

$$\frac{\partial T}{\partial t} = \alpha \nabla^2 T. \tag{6.10}$$

These two profiles form the basis for the majority of models in the literature.

Radiation may also be present but will be neglected here. For compression ignition engines, radiation sources comprise high temperature gas and particulates. Moderate combustion temperatures result in low gas emissivity. Homogeneous charge combustion and a low fuel concentration result in few or no particulates [24].

6.2.1 Heat transfer models

Models in the literature generally attempt to model either *quasi-steady* or *unsteady* heat transfer. The models may be further classed as either *boundary layer* or *diffusion* methods depending on whether measurements of the temperature gradient were made gas-side (as in [71]) or wall-side (as in [72, 73, 74]). In [75], both are modelled using a finite-difference method, matching the temperature at the wall. The different models in these areas are diverse and have not developed independently.

Despite the difficulties in modelling, some general observations can be made from heat transfer models in the literature. First, increasing engine speed decreases the heat transferred over a single stroke, even while increasing the heat-transfer rate. This is because the effect of a reduced stroke time is greater than the increase in effectiveness of the heat transfer mechanism. Second, the heat transfer per stroke can be reduced by minimising the surface area per unit volume of the cylinder[2].

Heat transfer in piston-compression devices has been investigated using gas heaters [76], reciprocating engines (both motored [71, 77, 78] and fired [19, 33] with either compression or spark ignition) and gun-tunnel drivers [79, 80]. A good review is provided in [81]. The literature on heat-transfer in reciprocating engines is the most extensive and models are commonly adapted from this material.

Since all models for reciprocating engines involved crankshaft driven pistons, the equations of motion for this type of engine had to be implemented for the purposes of comparison. They were prepared as

$$\left.\begin{aligned} \dot{x} &= u \\ \dot{u} &= \omega^2 \left[-l_s \cos\theta - \frac{{l_s}^2 \cos^2\theta - \sin^2\theta}{\left({l_r}^2 - {l_s}^2 \sin^2\theta\right)^{1/2}} - \frac{{l_s}^4 \sin^2\theta \cos^2\theta}{{l_r}^2 - {l_s}^2 \sin^2\theta^{3/2}} \right] \\ \dot{q} &= 0 \\ \dot{e} &= -\frac{pAu}{m_g} - \dot{q} \\ \dot{\theta} &= \omega \end{aligned}\right\} \tag{6.11}$$

where $l_s = \frac{L_c}{2}\left(1 - \frac{1}{r}\right)$ is the length of the crankshaft, $l_r = 2l_s$ is the length of the connecting rod and r is the compression ratio. The length of the connecting rod is commonly not provided in papers but variations on the value used here did not make a large difference to the piston trajectory. This system was used to validate the models with data from the literature.

After validation of these models a decision was made to develop a new unsteady integral boundary layer model. This model more accurately captured the instantaneous heat transfer data of Annand and Pinfold [73], particularly after top-dead-center. In addition, since this model was developed using the integral boundary layer method, more of the physics of the heat

[2]Finding the minimum volume to surface area of a cylinder using $\frac{dS}{dD} = 0$ yields $L = D$. To achieve this condition at the peak heat-flux (presumably the point of piston reversal) the volumetric compression ratio should be D/L_c, where L_c is the initial cylinder length.

transfer process was captured and so there was less reliance on the coefficients used to match the heat transfer rate to this data. As such, the model could be applied without modification to a free-piston engine with the hope that the change in piston trajectory did not largely affect the heat transfer rate.

6.2.2 Quasi-steady models

Quasi-steady (or averaged) models (a select number of which are described in §6.A.1 on page 108) assume fully developed flow. They provide correct average heat transfer rates for the engines upon which they are based. However, since the physics of a compression device are not correctly captured, they inevitably yield incorrect predictions outside their range of experimental data.

Taylor [19] collected experimental results on water-cooled spark ignition, water-cooled compression ignition and air-cooled compression ignition engines. For this data, engine sizes ranged from 3.25" by 4.5" to 28.3" by 47.3" bore by stroke, which is a comfortable range within which to conduct a parametric study. The original curve fit applied by Taylor was

$$\mathrm{Nu} = 10.4\mathrm{Re}^{0.75} \quad . \tag{6.12}$$

Annand [33] reinterpreted this data by making a distinction between the different engine types (see Figure 6.3). He then determined the heat transfer in a compression ignition engine was greater due to the radiation contribution and fit a model that included both radiation and convection to the data. The exception to this hypothesis was the heat transfer coefficient of the air-cooled compression ignition engine, which was assumed to be due to a higher bulk-gas wall temperature difference. The reinterpreted curves (shown in Figure 6.3) more closely approximated the trend of the data. For the experimental apparatus used by Annand (and after the inclusion of the radiation term), the coefficient reduced from between 14–20 to between 0.35–0.8, which is an entire order of magnitude reduction in the heat transfer coefficient. It wasn't clear which of these changes accounted for the reduction, however, using Annand's equation without the radiation term for fired engines underestimates the total heat-transfer. That said, radiation will not be considered for combustion of ventilation air due to the low combustion enthalpy and the lack of soot and radiative particles. Thus, it will be considered valid here to use the coefficients of Annand without the contribution due to radiation,

$$q_w{}'' = c_1 \frac{k}{D} \mathrm{Re}^{0.7} \left(T - T_w\right) \tag{6.13}$$

where c_1 is a coefficient corresponding to the experimental engine (between 0.35–0.8 in this paper), D is the bore diameter, k is the conduction coefficient, T is the average gas temperature, $T_w = T_{\mathrm{atm}}$ and the Reynolds number is calculated using

$$\mathrm{Re} = \frac{\rho \bar{u}_p D}{\mu}. \tag{6.14}$$

The average piston speed was taken to be the mean of the peak velocity, calculated using $\bar{u}_p = \frac{l_s}{2}\frac{\omega}{\sqrt{2}}$. The conduction coefficient is evaluated using Sutherland's law. Figure 6.4 shows the conduction coefficient for air along with an approximate equation used by Lawton [71] for

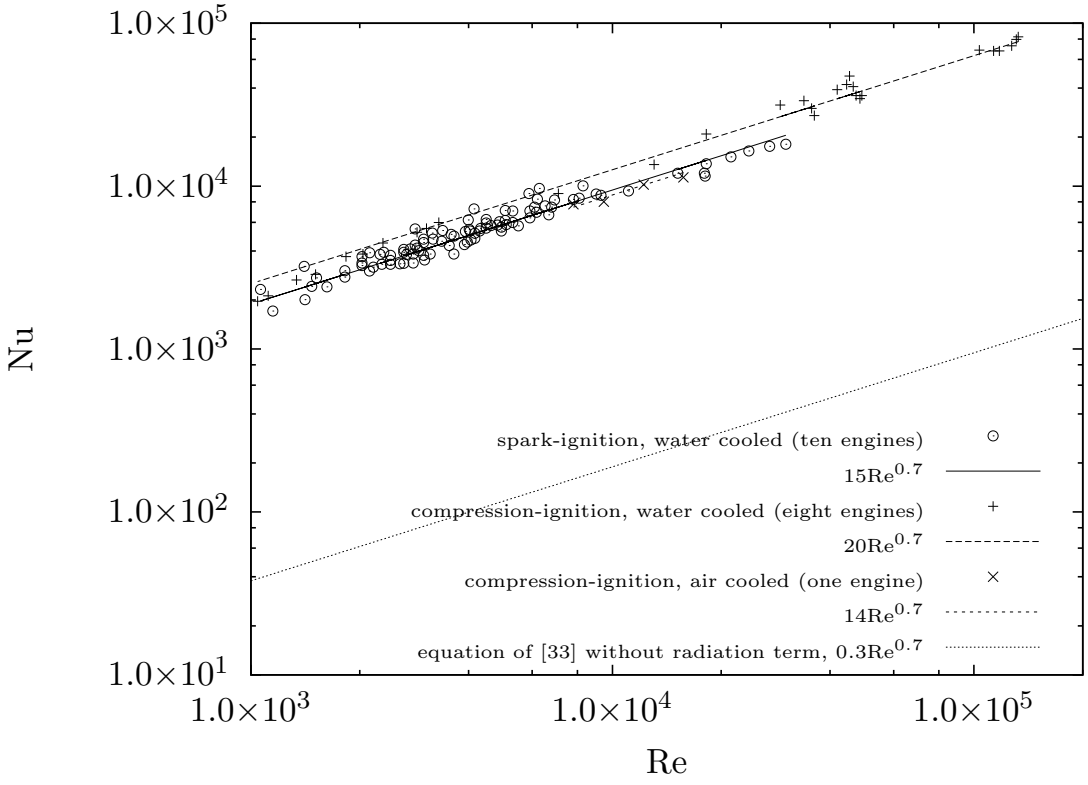


Figure 6.3: *Reinterpreted data of Annand [33] shown outside of its range of experimental data. Clearly, the coefficient of Annand's model does not fit this data, as it was based instead on experiments performed at Reynold's numbers between 1×10^5 and 1×10^7.*

comparison. Despite the variation in c_1 for quasi-steady models, the factor $\mathrm{Re}^{0.7}$ is remarkably consistent. Thus, the first test of any unsteady model should be that it is able to scale according to this factor.

6.2.3 Unsteady models

From experiments, the instantaneous heat transfer in an engine has been observed to be out-of-phase with the bulk gas-wall temperature difference [33, 72, 73, 78, 82, 83, 84]. This is due to the effect the compression work has on the spatial temperature gradient in the boundary layer, which acts to increase the heat transfer during the compression stroke and decrease (or somewhat surprisingly, even reverse) the heat transfer during the expansion stroke.

Unsteady (or instantaneous) models assume a partially developed boundary layer (as with external flow) and attempt to capture the instantaneous heat transfer rates. The early attempts were quasi-steady models that incorporated an unsteady term which increased the heat transfer during compression and reduced it during expansion to match what was observed. This term usually had a mean value of zero, such that the steady-state heat transfer was not affected. As the field progressed, unsteady models based on Fourier's law were solved numerically. Numerical attempts at turbulent boundary layer heat transfer in the literature apply incompressible flow

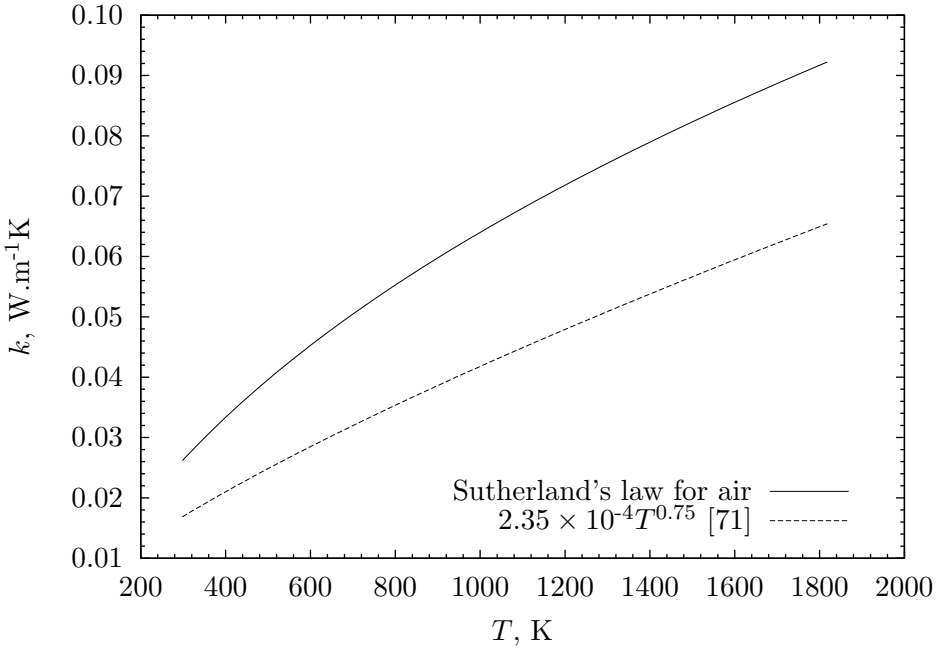


Figure 6.4: *Conduction coefficient of thermally-perfect air.*

theory as a first approximation [85, 86].

Gas-side finite-difference methods resolve the boundary layer to find its temperature profile. Annand and Pinfold [73] used a motored 122×140mm (bore×stroke) 4-stroke compression-ignition engine. Measurements found cycle to cycle variation in the gas velocity, particularly in the induction stroke, and to a lesser extent in the heat flux. Nevertheless, an empirical correlation was proposed containing a steady term to model the average cycle-to-cycle heat transfer and an unsteady term to model the phase lag (6.15).

$$q_w'' = 0.3\frac{k}{D}\mathrm{Re}^{0.7}\left[\underbrace{(T - T_w)}_{\text{steady}} + \underbrace{0.27\frac{D}{u}\frac{dT}{dt}}_{\text{unsteady}}\right] \tag{6.15}$$

The paper used the measured instantaneous gas velocity for u. It was observed that the measured gas velocity was roughly constant at $\bar{u}_p$, so this value was substituted in its implementation in this thesis. This substitution results in a lower negative heat flux during the expansion stroke than seen in [73].

Lawton's numerical model [71] assumes one-dimensional heat conduction along the cylinder axis (that is, that the cylinder wall is adiabatic). This along with the compressible boundary layer equations yielded

$$\frac{\partial T}{\partial t} + u\frac{\partial T}{\partial x} = \alpha\frac{\partial^2 T}{\partial x^2} + \frac{1}{\rho c_p}\frac{dp}{dt} \tag{6.16}$$

where $\alpha = \frac{k}{\rho c_p}$. Generally speaking, in (6.16) the first term on the right-hand side describes heat diffusion within the boundary layer and the second term describes isentropic compression in the

cylinder. This was solved using a Crank-Nicolson method which involved an explicit update for the time derivative and a second-order central difference update for the space derivative. During compression, work is done on the fluid directly in contact with the wall, so the real part of the heat transfer is in phase with the work ($\frac{dT}{dt}$) rather than the temperature difference ($T - T_w$). Based on this numerical model Lawton proposed an unsteady form of Annand's relation

$$q_w'' = \frac{k}{D}\left[0.28\mathrm{Re}^{0.7}\left(T - T_w\right) - 2.75CT_w\right] \tag{6.17}$$

which contained steady and unsteady terms. The first term is a function of the bulk gas-wall temperature difference and the second term is a function of the compressibility factor

$$C = (\gamma - 1)\frac{u_p}{L_c}\sqrt{\frac{D^3}{\alpha_0 \bar{u}_p}} \tag{6.18}$$

where $\alpha_0 = \frac{k_0}{\rho_0 c_{v,0}}$ is the diffusivity of air at port closure. This value for α_0 is faithfully reproduced here, even though it is commonly defined elsewhere as $\frac{k_0}{\rho_0 c_{p,0}}$.

The unsteady models of Annand and Pinfold and Lawton were compared, along with the steady model of Annand, to instantaneous data from a crankshaft driven compression cycle (see Figure 6.5). All models overestimate the heat transfer coefficient and even the unsteady models don't seem to adequately capture the heat transfer inversion about top dead center. Because of this, further investigation was performed into the use of boundary layer models.

6.2.4 Boundary layer models

A branch of heat transfer models that has been extensively developed is based on the boundary layer equations. Boundary layer models attempt to resolve the temperature profile across the boundary layer either analytically (by integration over an assumed profile) or numerically (using finite-difference methods) to determine the wall heat flux.

Consider the gas in a piston-compression device (Figure 6.6). We assume there exists a laminar thermal boundary layer between an isentropic core of gas and the cylinder wall across which heat is transferred. Boundary layer growth on the cylinder walls is unimpeded and so the local flow can be said to be external (see Figure 6.6). Neglecting buoyancy forces, the boundary layer equations for non-steady, compressible flow may be written [87]

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x}(\rho u) + \frac{\partial}{\partial y}(\rho v) = 0 \tag{6.19a}$$

$$\underbrace{\frac{\partial}{\partial t}(\rho u) + u\frac{\partial}{\partial x}(\rho u) + v\frac{\partial}{\partial y}(\rho u)}_{\text{convection}} = \underbrace{-\frac{\partial p}{\partial x}^{\nearrow 0}}_{\text{pressure work}} + \underbrace{\frac{\partial}{\partial y}\left(\mu\frac{\partial u}{\partial y}\right)}_{\text{viscous work}} \tag{6.19b}$$

$$\underbrace{\frac{\partial}{\partial t}(\rho h) + u\frac{\partial}{\partial x}(\rho h) + v\frac{\partial}{\partial y}(\rho h)}_{\text{change in enthalpy}} = \underbrace{\frac{\partial}{\partial y}\left(k\frac{\partial T}{\partial y}\right)}_{\text{heat transfer}} + \underbrace{\frac{\partial p}{\partial t} + u\frac{dp}{dx}^{\nearrow 0}}_{\text{work}} + \underbrace{\mu\left(\frac{\partial u}{\partial y}\right)^2 + \frac{dq_h}{dt}}_{\text{heat generation}} \tag{6.19c}$$

where μ and k may be temperature dependent. Equations (6.19) are usually reduced to their one-dimensional forms and linearised before being solved either numerically (as in Lawton and

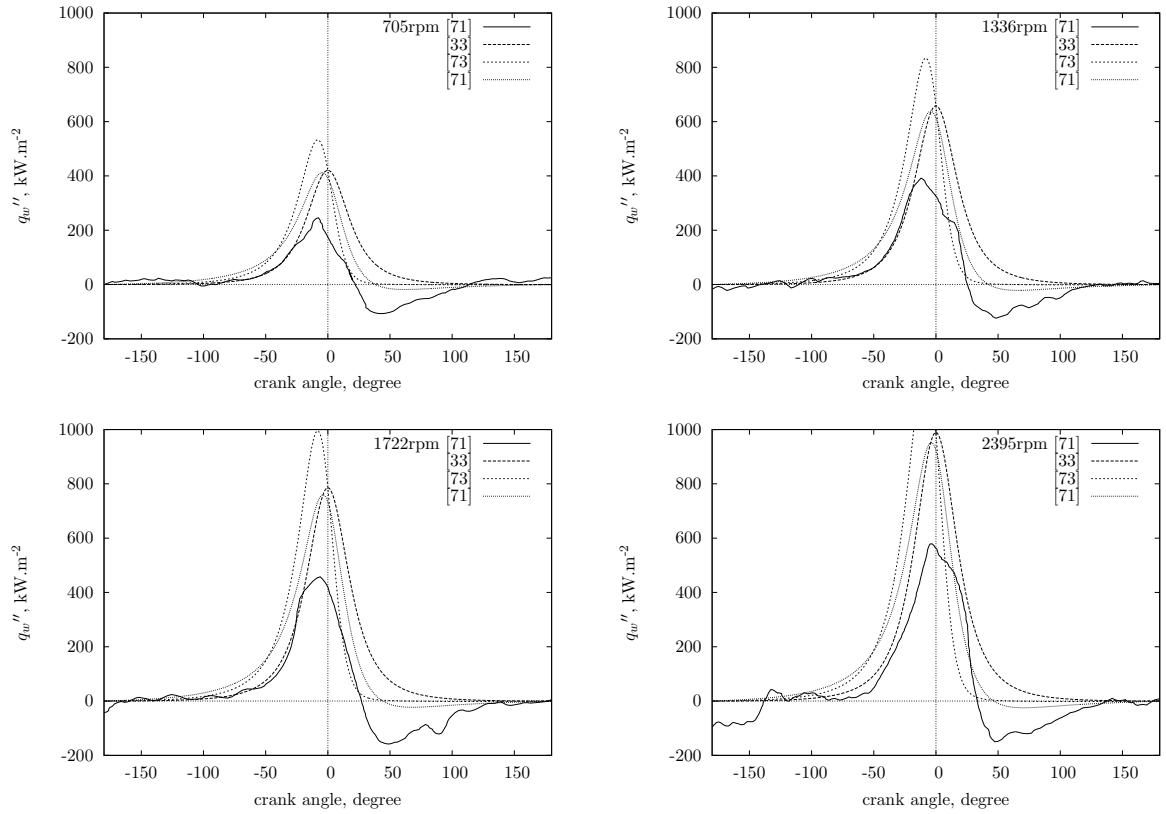


Figure 6.5: *A comparison between unsteady heat transfer models and the reinterpreted data of Lawton [71]. The discrepancies between heat-transfer models clearly show their specificity to a particular engine. The model of Lawton does not seem to match the data here as well as in his paper. The implementation here is clearly described in case of error.*

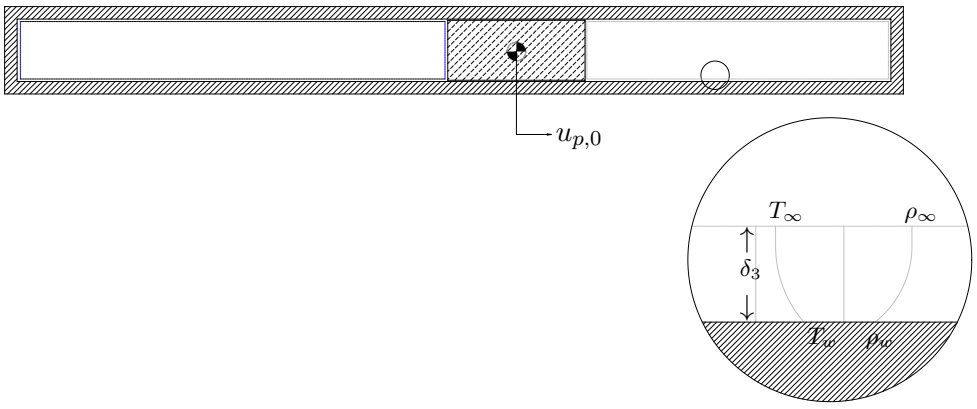


Figure 6.6: *Nomenclature for the unsteady thermal boundary layer and piston geometry. The idealised boundary layer shown was formed during the exhausting stage.*

Buttsworth who used finite-difference methods) or analytically. According to (6.19c), the energy of a volume of gas local to the wall is dependent on both the heat transfer and the work done

by the piston. To reduce the boundary layer equations to a one-dimensional form, heat transfer is considered "small" in one dimension or a stream function may be used. Most of the unsteady models in literature involve the use of finite-difference methods to find the heat transfer rate. Laminar boundary layers are modelled in [71, 76, 88]. Turbulent boundary layers are modelled in [85, 86, 89].

Tani *et al.* extended previous work on end wall heat transfer from a non reacting gas in a rapid compression device to include heat transfer to the cylinder wall. They used a pneumatically operated rapid compression device which had a typical stroke length of 12cm over 30ms and operated on a compression chamber of 3.8x3.8cm square. They noted that the heat transfer at the wall induced a convective transport of energy, $\rho c_p \frac{\partial T}{\partial t}$, towards the wall, and noted this occurs at all surfaces. In a similar method to Isshiki and Nishiwaki [90], Tani *et al.* used a stream function to linearise the two-dimensional temperature gradient which was then integrated numerically. Results compared very well to experiments and it was concluded that unsteady heat transfer to a cylinder wall during compression may be determined from the solution of the laminar boundary layer equations. The experiment also confirmed that the isothermal wall assumption was appropriate. However, since the experiment did not include the expansion stroke, no consideration was made to the inversion of heat transfer about the point of piston reversal.

Buttsworth [80] compared the numerical scheme of Lawton [71] to heat flux measurements from a gun tunnel barrel undergoing transient compression. The pressure was spatially heterogeneous in these experiments. Experimental results found that heat flux to the barrel and end walls were of the same magnitude, suggesting that the adiabatic cylinder wall assumption of [71] is not valid. In addition, Buttsworth found spatial variation of the heat transfer (which has also been seen in spherical bomb tests [33]) suggesting turbulence plays an important role, in contrast to the assumption of steady laminar flow used in the derivation of (6.50). He found reasonable agreement with the end wall peak heat flux values (when using the turbulent conduction coefficient (6.51)), but at other times found flux was underestimated by a factor of two or more. He also applied the model to the barrel wall and found the heat flux there was also underestimated. He concluded the model to be an improvement on quasi-steady models, but was limited by the lack of turbulent heat transport.

There are many other examples of numerical models (especially those based on the work of Isshiki and Nishiwaki [90]; see §6.A.2), however, to reduce the computational work required for the engine model, an integral boundary layer method is pursued instead. Turbulent models are usually developed from laminar models (indeed, laminar boundary layers form part of them) so a suitable laminar model is developed here and the turbulent model left to future work.

Integral boundary layer method

Integrating over the boundary layer is an approximate solution to the unsteady boundary layer equations. Suitable methods satisfy these equations only in the average across the boundary-layer thickness. Gas properties are continuous, but for approximate methods we define a boundary layer thickness, δ, and say the property of the fluid equals the freestream value at this point.

Momentum (δ_2) and thermal (δ_3) boundary layer thicknesses are defined in this way. There is a similarity between these two boundary layers, which is to say the shear $\tau = -\mu\nabla u$ and the heat flux $q'' = -k\nabla T$ behave similarly. To integrate across, say, the momentum thickness, an expression for the distribution of momentum in the boundary layer, $f(\eta)$ (where $\eta = y/\delta(x)$) is first chosen to satisfy the boundary conditions.

The assumption that most of the heat flux is lost to the cylinder wall is violated as the piston and cylinder head areas become the larger fraction of the total wetted surface area. However the piston and cylinder head were assumed to be adiabatic for the purpose of applying the integral boundary layer method. The resultant unsteady heat transfer model was then applied to the entire wetted area.

For this engine, flow is assumed to be inviscid, so there is a slip-wall condition and consequently no momentum boundary layer. However, there is still a thermal boundary layer due to the heat transfer at the wall. For subsonic compression, pressure is assumed to be isotropic, that is, $\frac{T}{T_\infty} = \frac{\rho_\infty}{\rho}$ holds throughout the cylinder. The boundary conditions then become

$$\left.\begin{aligned} f(\eta_2) = 0: \quad & \rho u = 0, v = 0 \\ f(\eta_2) = 1: \quad & \rho u = 0, v = \rho_\infty v_\infty \end{aligned}\right\} \tag{6.20}$$

for the momentum profile and

$$\left.\begin{aligned} f(\eta_3) = 0: \quad & T = T_w \\ f(\eta_3) = 1: \quad & T = T_\infty \end{aligned}\right\} \tag{6.21}$$

for the thermal profile. From these boundary conditions we can make the following observations:

1. the thermal profile varies from the wall temperature to the isentropic core temperature and

2. the thermal boundary layer thicknesses varies with cylinder length.

Knöös [76], modelled laminar heat transfer to an isothermal wall in a piston gas heater using an integral boundary layer approach. He assumed an ideal gas, subsonic flow and heat flux perpendicular to the cylinder wall. With the aid of the transformed continuity equation

$$\frac{\partial \rho}{\partial t} = -\frac{\partial}{\partial y}(\rho v) \tag{6.22}$$

and a form of the ideal equation of state

$$\rho h = \left[\frac{\gamma}{(\gamma - 1)}\right] p, \tag{6.23}$$

the energy equation (6.19c) was written as

$$q_w'' = -h_\infty \frac{d}{dt}\int_0^y (\rho(t, y) - \rho_\infty(t))\, dy. \tag{6.24}$$

Here, subscript ∞ refers to the condition in the core and $q_w{''}$ to the heat-transfer rate at the wall (see (6.9)). Since temperature and density do not go to zero at the wall, Knöös used the similarity profile

$$f(\eta_3) = \frac{\rho(t,\eta_3) - \rho_\infty(t)}{\rho_w(t) - \rho_\infty(t)} \tag{6.25}$$

where ρ and T exist at some point within the boundary layer, $\eta_3 = \frac{y}{\delta_3}$, and δ_3 is a characteristic thermal boundary layer thickness. Knöös then went on to find a time-dependent, non-dimensional heat-transfer equation which was correlated to experimental data. The paper is somewhat unique and has had almost no further application in the literature. Development of this integral boundary layer method diverges from Knöös at (6.25).

Applying the co-ordinate transform $L = (x_c - x_p)$ and the similarity profile, equation (6.24) becomes

$$h_\infty u_p \frac{d}{dL}\left(\rho_\infty - \frac{p}{RT_w}\right)\delta_3 \int_0^1 f(\eta_3)d\eta_3 = -q_w{''}. \tag{6.26}$$

With isentropic freestream conditions and constant wall temperature, the remaining unknowns may be found

$$\frac{dp}{dL} = -\frac{\gamma p_0 {L_0}^\gamma}{L^{(\gamma+1)}} \qquad \text{and} \tag{6.27a}$$

$$\frac{d\rho}{dL} = -\frac{\rho_0 L_0}{L^2}. \tag{6.27b}$$

where the subscript 0 denotes the property at valve close.

The similarity profile $f(\eta_3) = (1 + a\eta_3)\exp\left(\frac{\eta_3}{\eta_3 - 1}\right)$ was chosen as it has a number of useful properties. First, it satisfies the boundary conditions of (6.28)

$$\left.\begin{aligned} \eta_3 = 0: &\quad f(\eta_3) = 1 \\ \eta_3 = 1: &\quad f(\eta_3) = 0 \end{aligned}\right\} \tag{6.28}$$

but more importantly, it also allows for the temperature gradient at the wall to be independent of the bulk gas-wall temperature (see Figure 6.7).

Given $f'(\eta_3 = 0) = (1 - a)$, Fourier's law becomes

$$q_w{''} = -\frac{k}{\delta_3}(1 - a)\,(T_\infty - T_w) \tag{6.29}$$

Inserting this and (6.27) into (6.26) and given $\int_0^1 f(\eta_3)d\eta_3 = -(1 + a)$, the equation for the thermal boundary layer thickness may then be written

$$\delta_3 = \left[\frac{\alpha_0}{|u_p|}\frac{(1-a)}{(1+a)}\left(1 - \frac{T_w}{T_\infty}\right)\left(\frac{L^{(\gamma+1)}}{\gamma {L_0}^\gamma - L_0 L^{(\gamma-1)}}\right)\right]^{\frac{1}{2}} \tag{6.30}$$

where $\alpha_0 = \frac{k}{\rho_0 c_p}$.

This new model was then matched to data from [71]. The reverse of the temperature gradient at the wall (due to the addition of work to the boundary layer) is allowed for in the selection of the boundary layer profile. To model this, the value for a must be chosen such that the

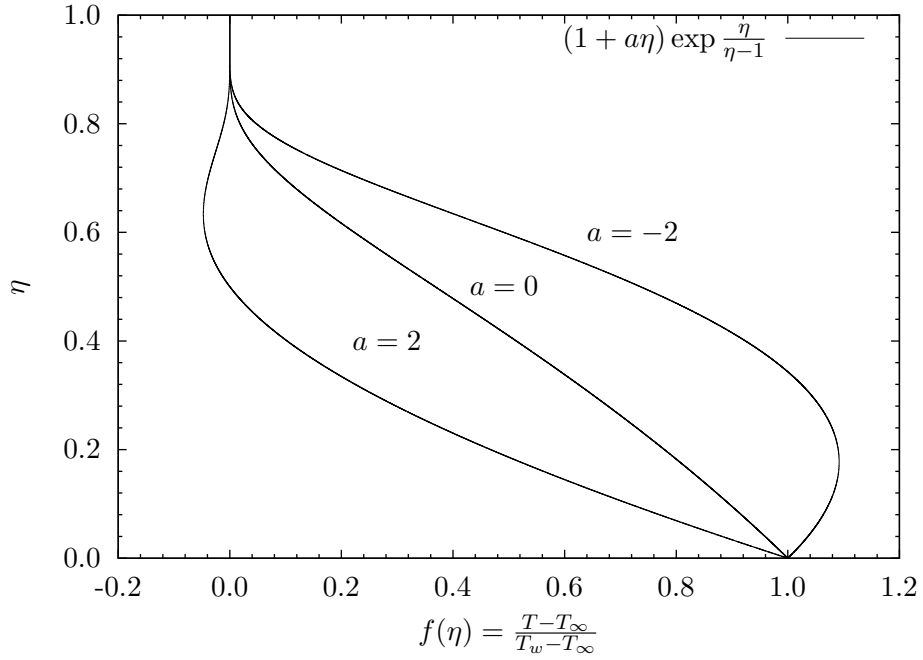


Figure 6.7: *Boundary layer profile.*

predictions match data but should still hold some relation to the work performed by the piston. In particular, a proportionality to the velocity was expected to yield a good result. The value for a was chosen to be

$$a = -0.8\frac{u_p}{u_{p,0}} \tag{6.31}$$

which is applicable to both the left and right-hand cylinder (remembering that $u_{p,0,L} = -u_{p,0,R}$). This value for a makes the boundary layer thickness asymmetrical about the point of piston reversal. The final equation became

$$q_w{}'' = -1.6\frac{k}{\delta_3}(0.5 - a)\,(T_\infty - T_w) \ , \tag{6.32}$$

This was used with a crankshaft compression cycle and compared to data from Lawton in Figure 6.8. Compared to other models (see Figure 6.5) it predicts a lower heat transfer but seems to match the heat flux of [71] well, particularly the negative heat transfer coefficient seen after top dead center.

Finally, the total heat transfer over a stroke was compared with that of the steady model of Annand with a coefficient of $c_1 = 0.15$ for different engine speeds (see Figure 6.9). The total heat transfer over a stroke scales relatively well with engine speed, which is seemingly the only consistent factor between steady-state experimental heat transfer data. At this point, then, it can be said that the unsteady heat transfer model, (6.32), is validated as much as is possible without an experimental apparatus.

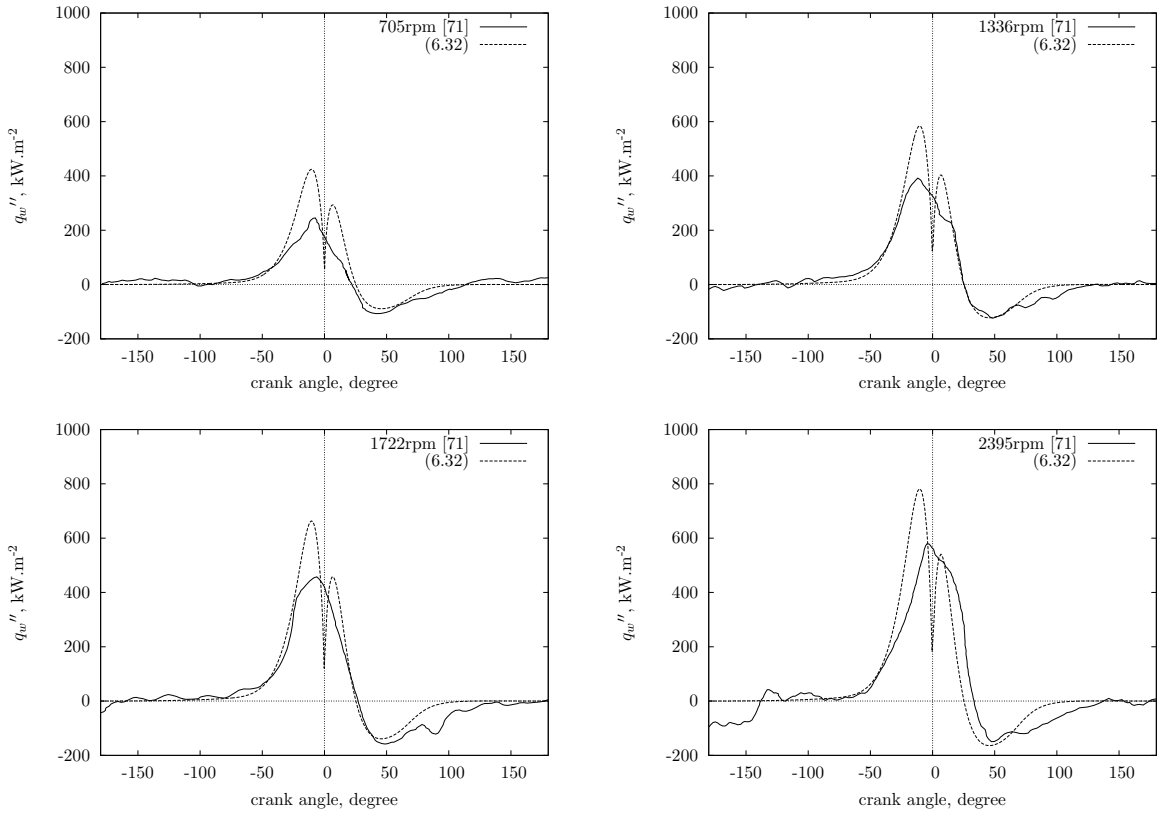


Figure 6.8: *Comparison of instantaneous heat transfer with data of [71]. The engine used is a Perkins 98.4mm bore, 127mm stroke, naturally aspirated, four-cylinder, water-cooled diesel of compression ratio 15.6:1. Coefficients were chosen particularly to match with this data. This model was applied using the state equations for a crankshaft engine.*

6.2.5 Heat transfer in a free-piston compressor with and without finite-rate chemistry

Once the integral boundary layer model was validated using the state equations for a crankshaft engine, it was compared to the steady model of Annand using the free-piston compressor. The result of this (not shown) was that the original steady model of Annand predicted a total heat-transfer over a stroke far greater than the integral boundary layer model. It was assumed that the result predicted by the steady-state model was incorrect and due (at least in part) to the difference in engine geometry and piston trajectory. It was also assumed that, since the integral boundary layer model developed was not specific to a particular engine type, that its result was at least more accurate. Thus, for the purpose of comparison, the coefficient c_1 of Annand's steady-state model was reduced until the total heat transfer over the stroke matched between the two models (see Figure 6.10).

The same simulation was then run with finite-rate chemistry to determine the effect of unsteady heat transfer on chemistry. It is interesting to note that while the total heat transfer

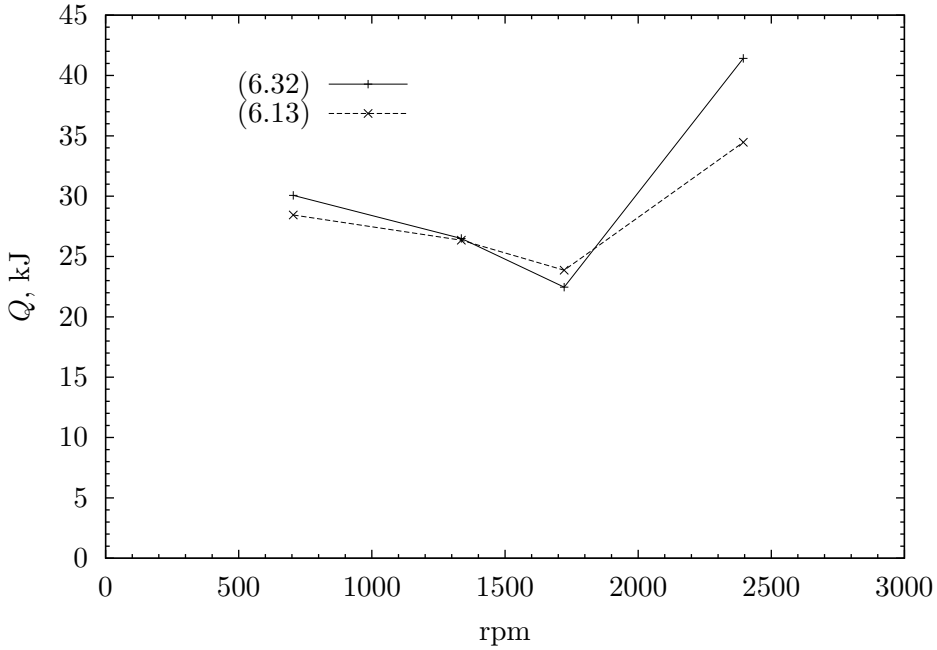


Figure 6.9: *Comparison of the total heat transfer over one stroke between Annand's model with* $c_1 = 0.12$ (6.13) *and the integral boundary layer model* (6.32)*. The engine parameters are a 98.4mm bore, 127mm stroke, with a compression ratio 15.6:1. This data was generated using a crankshaft engine model. It shows that the total heat transfer predicted by the integral boundary layer model scales relatively well with engine speed.*

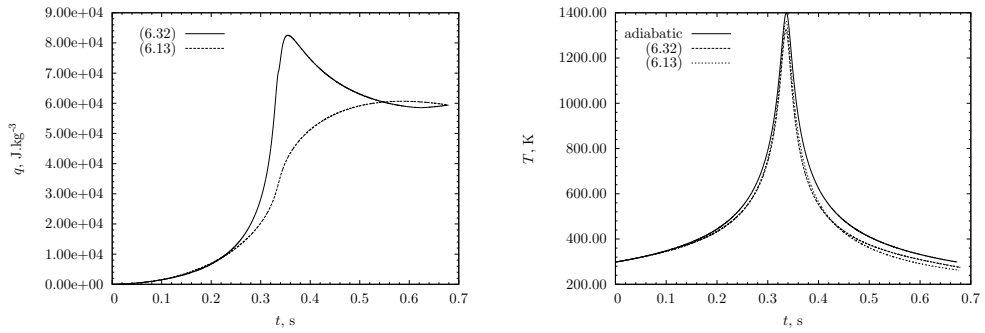


Figure 6.10: *Comparing the effect of steady and unsteady heat transfer models. Annand's model had a coefficient of* $c_1 = 0.0363$ (6.13) *and the integral boundary layer model* (6.32) *was unmodified. The total heat transfer over a stroke for both heat transfer models is about 60kJ. The peak temperature for adiabatic and steady and unsteady heat transfer was 1400K, 1363K and 1323K respectively. Engine parameters were* $u_{p,0} = 13.01 m.s^{-1}$, $m_p = 400kg$, $R = 40$, $D = 0.1m$ *and* $T_s = 1400K$ *(without chemistry). Each run takes about 1min on one core of an AMD 1090T Phenom X6 processor.*

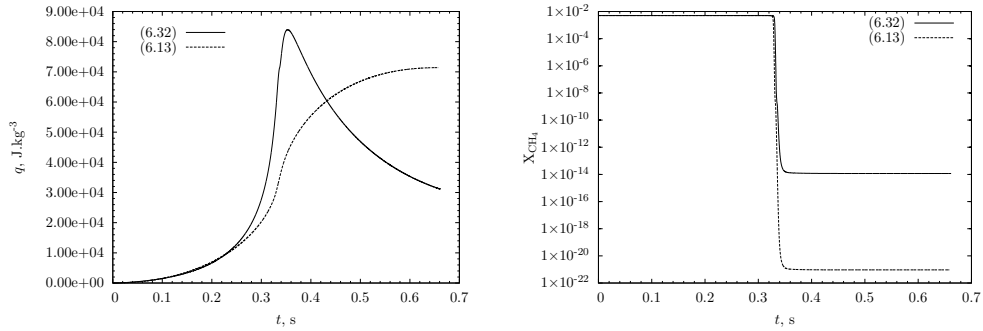


Figure 6.11: *Comparing the effect of steady and unsteady heat transfer models on finite-rate chemistry. Annand's model again had a coefficient of* $c_1 = 0.0363$ (6.13) *and the integral boundary layer model* (6.32) *was unmodified. The effect of the unsteady model was to reduce the total heat transfer over the stroke to 40kJ, even while it lowered the peak temperature at the point of piston reversal and thus retarded combustion. The unsteady model in contrast predicted a total heat transfer of about 72kJ over the stroke, which is greater than without combustion. Engine parameters were* $u_{p,0} = 13.01 m.s^{-1}$, $m_p = 400kg$, $R = 40$, $D = 0.1m$ *and* $T_s = 1400K$ *(without chemistry). Each run takes about 130mins on one core of an AMD 1090T Phenom X6 processor.*

matched between these two models without chemistry, it doesn't match with chemistry. In particular, the unsteady model increases heat release during combustion and subsequently reduces it during expansion, whereas the steady model increases heat transfer throughout combustion and expansion. Despite having a lower total heat loss over the stroke, the unsteady model reduced the peak temperature and so reduced the rate of combustion when compared to the steady model. This can be seen in the larger fraction of CH_4 remaining (see Figure 6.11). The unsteady heat transfer model predicted a lower peak temperature than Annand's model.

The purpose of developing this model was threefold. Firstly, it allowed for instantaneous heat transfer throughout the stroke which has an effect on the finite-rate chemistry. Secondly, since the model is not specific to a particular engine type, it should be directly transferable to a free-piston engine, and indeed, that is what was done. Finally, it is hoped that due to the boundary layer method employed, more of the actual physics of the heat transfer process was captured and so there was less reliance on the coefficients used to match the heat transfer rate to data. Thus, the heat transfer model developed here is expected to perform better in the parametric study than an existing heat transfer model from the literature.

6.A Chapter end notes

6.A.1 Quasi-steady models

The field of quasi-steady heat transfer models has been extensively developed [24, 72, 83, 91] *et al.* These models are correlated to the total heat transfer measured in motored or fired engines running at constant speed. They use the convective heat flux equation to model the instantaneous heat flux using the bulk gas-wall temperature difference

$$q_w'' = h\left(T_\infty - T_w\right). \tag{6.33}$$

Using Reynold's analogy, steady heat transfer between a forced turbulent flow and a wall obeys a Nusselt-Reynolds relation

$$\bar{\mathrm{Nu}} = \frac{hx}{k} = c_1 \mathrm{Re}^{c_2}, \tag{6.34}$$

although heat transfer in an engine is an unsteady process. The Nusselt number is defined as the dimensionless temperature gradient at a surface. The first such model was proposed by Annand [33] who, after a review of existing literature, used dimensional analysis to yield

$$q_w'' = c_1 \frac{k}{D} \mathrm{Re}^{c_2} \left(T - T_w\right) + c_3 \left(T^4 - T_w^4\right) \tag{6.35}$$

where Re is for fully developed pipe flow[3]

$$\mathrm{Re} = \frac{\rho \bar{u}_p D}{\mu}. \tag{6.36}$$

The mean piston speed is employed since flow in the cylinder due to other effects is largely unknown. This is a more accurate assumption for a motored engine, where there is no combustion. For an engine with a crankshaft, this may be calculated using $\frac{L_s \omega}{2\sqrt{2}}$. For a free piston engine, the mean piston speed was calculated using $\bar{u}_p = \frac{u_{p,0}}{\sqrt{2}}$. A value of c_1 within the range 0.35 to 0.8 was suggested, depending on the intensity of the charge motion. Analysis of previous experimental data provided correlations of

$$c_1 = \begin{cases} 0.38 & \text{for a } 460\times381\text{mm two-stroke engine} \\ 0.49 & \text{for a } 521\times390\text{mm four-stroke engine} \end{cases} \tag{6.37}$$

$c_2 = 0.7$ and

$$c_3 = \begin{cases} 1.6 \times 10^{-12} & \text{for diesel engines} \\ 2.1 \times 10^{-13} & \text{for spark-ignition engines} \end{cases} \tag{6.38}$$

although in other papers the radiative heat transfer term is commonly dropped. It is noted in the paper that while the instantaneous heat transfer coefficients provided by (6.35) are strictly incorrect (due to the phase lag) the average heat transfer prediction is still accurate.

[3] The mixing rule for viscosity is provided in §2.1.3.

Based on (6.34), Woschni [22] developed a more general formula for heat transfer based on bomb calorimeter and motored and fired engine experiments. The heat transfer formula for a motored engine was

$$h = 110 D^{-0.2} p^{0.8} T^{-0.53} c_1 \bar{u}_p{}^{0.8} \tag{6.39}$$

where $\bar{u}_p$ was the mean piston speed,

$$c_1 = \begin{cases} 6.18 & \text{during exhausting} \\ 2.28 & \text{during compression and expansion.} \end{cases} \tag{6.40}$$

and $c_2 = 0.8$. The coefficient c_1 seems to scale with engine size whereas c_2 seems to be related to fluid dynamics and is more constant between engines. Equation (6.39) is not dimensionally consistent and units of m, kp.cm^{-2}, K and m.s^{-1} must be observed. A heat transfer formula including terms for fuel injection and combustion was also presented. Subsequent papers included additional terms for effects such as swirl [72].

Kamel and Watson [91] studied an indirect injection diesel engine 100×127mm (stroke×bore) at speeds of 1320 to 2800rpm. An indirect injection engine has a greater heat loss than a similar direct injection engine due to the pre-chamber. To account for this, Kamel and Watson implemented a swirl model (based on conservation of kinetic energy) for the gas velocity instead of using the mean piston speed. Viscous decay in velocity was modelled by assuming a forced vortex flow field and applying rotating disc friction data. The velocity was increased by adding the kinetic energy of the injected fuel in the fired case. They arrived at coefficients of

$$c_1 = \begin{cases} 0.023 & \text{(pre-chamber)} \\ 0.012 & \text{(main chamber)} \end{cases} \tag{6.41}$$

and $c_2 = 0.8$. They included a radiation model with the emissivity based on gaseous CO_2, H_2O and soot, stating the influence was small but not insignificant at these conditions. The engine size is much smaller in this experiment and both Annand and Woschni's equations overestimated the heat loss by a factor of two.

Hohenberg [24] conducted an investigation on heat transfer in diesel engines ranging in size from 97mm×128mm to 128mm×142mm (bore×stroke) and speeds of 600 to 2300rpm. Surface temperature, local heat flux and pressure measurements were all taken and used independently (with appropriate relations) to recover total heat loss. The heat transfer predictions from (6.39) were found to be too low during compression and too high during combustion [24]. Hohenberg modified (6.39) by defining D as the diameter of a sphere with a volume equivalent to the cylinder volume and proposed a relation between the mean gas velocity and gas properties. The resulting convective heat transfer coefficient was

$$h = c_1 \vartheta^{-0.06} p^{0.8} T^{-0.4} \left(\bar{u}_p + c_4\right)^{0.8}, \tag{6.42}$$

where $c_1 = 130$, $c_4 = 1.4$. It was noted that for engines outside of this range or type (such as low speed, gasoline or pre-chamber engines) measurements must be made to determine the constants for the proposed heat transfer equation.

Aichlmayr [92] solved (6.16) by assuming spatially homogeneous temperature, producing a model that captured volume-to-surface-area effects for a millimeter-scale free-piston compressor. Spatially homogeneous temperature requires the condition $\mathrm{Bi}_L < 0.1$, which is very conservative for large engine geometries. Using a modified Bessel function of the second kind for a finite cylinder, the following relation was derived:

$$\dot{q} = \frac{2 \times 10^3 k \left(T_\infty - T_w\right)}{L} \left(\frac{0.440332\pi^2 + 5.09296 \frac{L}{D}^2}{\pi + 2\pi \frac{L}{D}} \right). \tag{6.43}$$

This was not compared with experimental data.

6.A.2 Unsteady models

Wendland [82] proposed a one-dimensional numerical model comparing numerical results to experimental data from a small (58×40mm bore×stroke) motored reciprocating engine. The so-called "adiabatic plane model" assumed heat transfer from an adiabatic plane that existed at a point midway along the cylinder to the isothermal end wall. The model used the enthalpy and conservation of energy equations to derive

$$q_w'' = \rho \frac{dh}{dt} - \frac{dp}{dt}. \tag{6.44}$$

This model captured the heat flux inversion after top dead center but underestimated the total heat transfer.

Kornhauser and Smith [74] measured wall-side heat flux and used a complex Nusselt number to account for the phase shift between the heat flux and the bulk gas-wall temperature difference. Writing (6.34) with a complex temperature and assuming sinusoidal wall temperature, they derived a form of the heat flux similar to that found empirically by [73].

$$q_w{}'' = \frac{k}{D_h} \left[\mathrm{Nu}_r \left(T - T_w\right) + \frac{\mathrm{Nu}_i}{\omega} \frac{dT}{dt} \right], \tag{6.45}$$

where $T = T_0 + T_a \cos\left(\omega t\right)$ and

$$\mathrm{Nu}_r = \mathrm{Nu}_i = 0.56 \mathrm{Pe}_\omega{}^{0.69}. \tag{6.46}$$

Wall-side finite-difference methods model the diffusion of heat through the cylinder wall (assuming a periodic boundary condition). Assuming constant properties, the temperature distribution in a solid wall is determined by the diffusion equation

$$\frac{dT_w}{dt} = \alpha \nabla^2 T_w \tag{6.47}$$

which is usually reduced to a one-dimensional form for use in analytical solutions.

Sihling and Woschni [72] performed measurements on an externally supercharged four-stroke 165×155mm (bore×stroke) diesel engine over a limited range in speed (1499 to 1506rpm). They solved the diffusion equation using a periodically varying temperature distribution to find the surface temperature using a Fourier transform [72]

$$T_w(t) = \bar{T}_w(0) + \sum_{n=1}^{\infty} \left(A_n \cos n\omega t + B_n \sin n\omega t\right) \tag{6.48}$$

The steady-state component normal to the surface could not be determined exactly from measurements. A correlation could not be made for two proposed reasons:

1. the heat flux was not one-dimensional or
2. the bulk gas-wall temperature difference is decisive to local heat transfer.

Comparisons with (6.39) agreed during the high pressure phase of the cycle but were found to be inaccurate during the exhausting stage.

6.A.3 Numerical boundary-layer methods

Isshiki and Nishiwaki [90] reduced the enthalpy equation to the form of the 1-D transient equation for heat conduction by assuming adiabatic walls, a laminar boundary layer and the Lagrangian coordinate transform

$$y' = \int_0^y \frac{\rho}{\rho_0} dy \tag{6.49}$$

where the subscript 0 denotes the value of a gas property at the beginning of the compression stroke. This has been solved numerically by a number of authors (e.g. [89]).

Lawton [71] developed a laminar one-dimensional heat transfer model which was solved both analytically (see §6.2.3) and numerically. The convective term of (6.19c) was removed using the continuity equation. The internal form of (6.19c) in the x-direction was then used (assuming $\frac{\partial T}{\partial x} = 0$) to derive

$$\frac{\partial T}{\partial t} = \frac{k}{\rho c_\nu} \frac{\partial^2 T}{\partial x^2} - \frac{(\gamma - 1)}{\nu} T \frac{d\nu}{dt}, \tag{6.50}$$

where k was taken to be the turbulent thermal conductivity of air modelled as

$$k_t = 25k = 2.35 \times 10^{-4} T^{0.75} \quad . \tag{6.51}$$

The coefficient of 25 may be due to the fact that the Nusselt-Reynolds relation was taken from Annand but excluded the radiation term.

Chapter 7

Engine control

The two cylinders in a free-piston engine of this type are coupled, that is, the exhausting process of one cylinder is dictated by the compression process of its opposing cylinder and the exhausting process in turn has a large effect on the next compression process. The initial piston velocity (that is, the piston velocity at poppet valve close) is paramount since it affects both of these processes.

For control of the initial piston velocity, energy was subtracted from (or added to) the piston during expansion stroke. This decreased (or increased) the piston velocity to the initial velocity required for the next exhausting process. Control of the entrained mass fraction was aided by the timing of the poppet valve.

7.1 Control

A dual piston type free-piston engine should ideally operate at its natural frequency [15]. The natural frequency is a function of the piston mass, the entrained mass of ventilation air, the piston surface area and the cylinder length. These parameters must be chosen to ensure sufficient time for both combustion and the exhausting process. For a given piston mass, the compression ratio is determined largely by the entrained mass of ventilation air, m_g, and the initial piston velocity $u_{p,0}$. These variables are coupled, but the former may be controlled largely by variable valve timing (as discussed in §5.2.1), while the latter may be controlled using a force imparted by a linear electric motor.

The design of the linear electric motor is left for future work, although a brief time-domain analysis of a similar linear system is provided in §7.A.2. It is envisaged that it will be designed to keep the engine operating at its natural frequency. As such, it will extract energy from the piston as its frequency increases above its natural frequency and provide energy below it, as in $F_e = \mathrm{fn}(f_p - f_{p,n})$, where f_p is the piston frequency and $f_{p,n}$ is its natural frequency.

A free-piston engine control system is developed here using the isentropic model developed in Chapter 4.

7.1.1 Finding the required electromagnetic force

Consider a frictionless free-piston cylinder arrangement where the gas is expanding. The change in momentum of the piston (where an external force is applied) is defined as

$$m_p u du = F_e dx + pA dx \quad . \tag{7.1}$$

Because the pressure at reed valve open is a known quantity (that is, p_{atm}) this was used to determine the electromagnetic force. Only one cylinder was considered because, by design, the opposing cylinder should be undergoing exhausting during this period and so not remove substantial energy from the piston. The required electromagnetic force was derived using the dynamics method. It may be calculated using

$$F_e = \frac{1}{(L_{\text{ex}} - L)} \left[\frac{p_{\text{atm}} A}{{L_{\text{ex}}}^{\gamma} (\gamma + 1)} \left({L_{\text{ex}}}^{\gamma+1} - L^{\gamma+1} \right) + \frac{1}{2} m_p \left({u_{p,0}}^2 - {u_p}^2 \right) \right] \quad , \tag{7.2}$$

$$L_{\text{ex}} = \left(\frac{p_{\text{atm}}}{p} \right)^{\frac{1}{\gamma}} \tag{7.3}$$

and instantaneous values for L, u_p and p. The first term of (7.2) accounts for the potential energy stored as pressure in the cylinder. The second term accounts for the error in the piston velocity. Limiting the force provided by the linear electric motor to 10 kN prevented overshoot of the target velocity and resulted in quite a robust control system.

7.1.2 Control of entrained mass of ventilation air

As shown in §5.5, the entrained mass of ventilation air is an important parameter for control of the volumetric compression ratio. This variable is controlled by the timing of the poppet valve, which may be closed as the entrained mass approaches the desired value. In a real engine, since it cannot be measured directly, this variable must be inferred using the gas pressure, temperature and the cylinder volume.

7.1.3 PID control of the piston velocity

There will be an error in the initial velocity $\epsilon = u_{p,0} - u_p$ due to

- incorrect approximation of the exhaust length L_{ex}
- the heat and frictional losses (which are not included in the control system analysis).

This error leads to a difference in the target velocity, and this difference is used by the electric motor to correct the force. The resulting control block diagram for this system is shown in Figure 7.1. The control logic that determined the electromagnetic force is provided as Figure 7.2.

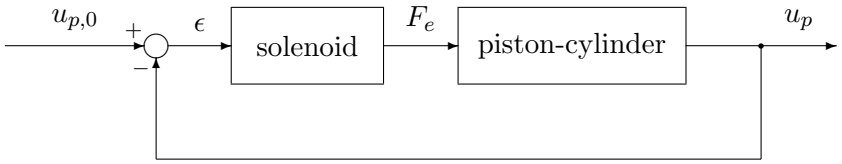


Figure 7.1: *Control block diagram. The solenoid here is used to adjust the piston velocity by application of an electromagnetic force.*

input : u_p, $u_{p,0}$, L_L, L_R, L_c, p_L, p_R, γ_L, γ_R
output: F_e

if $u_p \geq 0$ **then**
  if $L_R > L_c$ **then**
    if $p_L > p_{atm}$ **then**
      $a = \gamma_L + 1.0$
      $L_{\text{ex}} = L_L * \text{pow}((p_{\text{atm}}/p_L), (1.0/\gamma_L))$
      $b = p_{\text{atm}} * A/(a * \text{pow}(L_{\text{ex}}, \gamma_L))$
      $dx = L_L - L_{\text{ex}}$
      $F_e = (0.5 * m_p * (u_p0 * u_p0 - u_p * u_p) + b * (\text{pow}(L_{\text{ex}}, a) - \text{pow}(L_L, a)))/dx$
else
  if $L_L > L_c$ **then**
    if $p_R > p_{atm}$ **then**
      $a = \gamma_R + 1.0$
      $L_{\text{ex}} = L_R * \text{pow}((p_{\text{atm}}/p_R), (1.0/\gamma_R))$
      $b = p_{\text{atm}} * A/(a * \text{pow}(L_{\text{ex}}, \gamma_R))$
      $dx = -L_R + L_{\text{ex}}$
      $F_e = (0.5 * m_p * (u_p0 * u_p0 - u_p * u_p) + b * (\text{pow}(L_{\text{ex}}, a) - \text{pow}(L_R, a)))/dx$

if $F_e > 1 \times 10^4$ **then**
  $F_e = 1 \times 10^4$
if $F_e < -1 \times 10^4$ **then**
  $F_e = -1 \times 10^4$
return : F_e

Figure 7.2: *Logic for calculation of the electromagnetic force.*

7.2 Engine start-up

For the purpose of this thesis, a reasonable maximum value for the electromagnetic force was be chosen as 10kN, which was then applied to achieve a desired initial velocity (see Figure 7.3). As can be seen, the target initial velocity ($u_{p,0} = 14.77\,\text{m.s}^{-1}$) was reached with little error. Engine performance is then estimated from the steady-state cycles after $t = 4\,\text{s}$ in Figure 7.3.

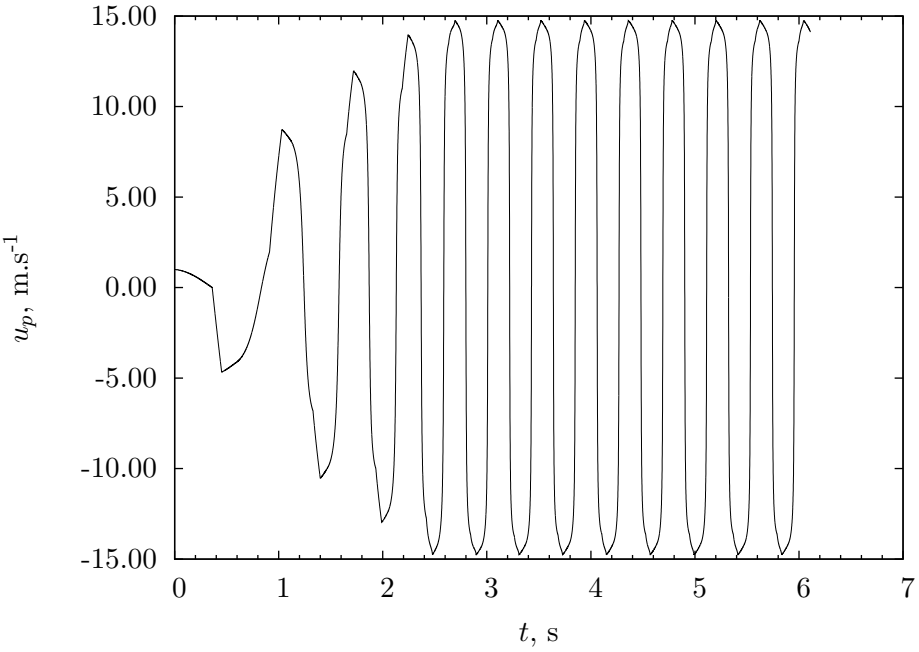


Figure 7.3: *Forced response of the piston during start-up for the reference engine with a target initial velocity of $u_{p,0}$=14.77 m.s^{-1}. The maximum electric motor force applied here is F_e = 10 kN. Exhausting and combustion is performed during this process.*

7.A Chapter end notes

7.A.1 Finding the required electromagnetic force to reach a peak temperature

Defining the instantaneous cylinder length and stroke length to be

$$L = (x_c - x_p) \qquad \text{and}$$
$$L_s = (x_s - x_0)$$

respectively, equation (4.3) may then be rewritten as

$$L_s = L - \left[L^{(1-\gamma)} - \frac{m_p (1-\gamma)}{2 p_0 A L^{\gamma}} {u_p}^2 \right]^{\frac{1}{(1-\gamma)}} . \tag{7.4}$$

Allowing an external force F_e to be applied to the piston, (4.1) becomes

$$F_e = \frac{1}{L_s} \left\{ \frac{1}{2} m_p {u_p}^2 - \frac{p_0 A L^{\gamma}}{(1-\gamma)} \left[(L - L_s)^{(1-\gamma)} - L^{(1-\gamma)} \right] \right\} \quad . \tag{7.5}$$

Applying the additional constraint $T_s = T_{\text{ig}}$,

$$L_s = L_c \left[1 - \left(\frac{T_{\text{ig}}}{T_0} \right)^{\frac{1}{(1-\gamma)}} \right] \tag{7.6}$$

which, when combined with (7.4), yields the force required to reach the ignition temperature for any initial conditions:

$$F_e = \frac{\left[\frac{1}{2} m_p {u_0}^2 - \frac{p_0 A {L_c}^{(1-2\gamma)}}{(1-\gamma)} \left(\frac{T_{\text{ig}}}{T_0} - 1 \right) \right]}{L_c \left[1 - \left(\frac{T_{\text{ig}}}{T_0} \right)^{\frac{1}{(1-\gamma)}} \right]} \quad . \tag{7.7}$$

Equation (7.2) can be used directly with the control system.

7.A.2 Piston-solenoid dynamics for a linear system

Consider a spring-mass-damper system whereby the mass carries a static magnetic field. Now consider an inductance-capacitance-resistance circuit, where the inductor is a solenoid. Finally, consider the two systems are coupled by allowing the mass to move through the solenoid, generating current.

The force on the mass is proportional to the current in the solenoid, and the voltage in the solenoid is in turn proportional to the velocity of the mass. Thus, the coupling equations for the force by the solenoid on the mass (F_e) and the back electromagnetic force on the solenoid (e_b) are

$$\begin{aligned} F_e &= -k_1 i \\ e_b &= k_2 u_m \end{aligned} \tag{7.8}$$

where k_1,k_2 are constants of proportionality. The resulting system dynamics are shown in Figure 7.4. As can be seen, the current is initially proportional to the velocity.

To a first approximation, a free-piston engine may be modelled by this system. Assuming the gas acts as a linear spring results in a lower bound to the actual dynamics. The piston trajectory

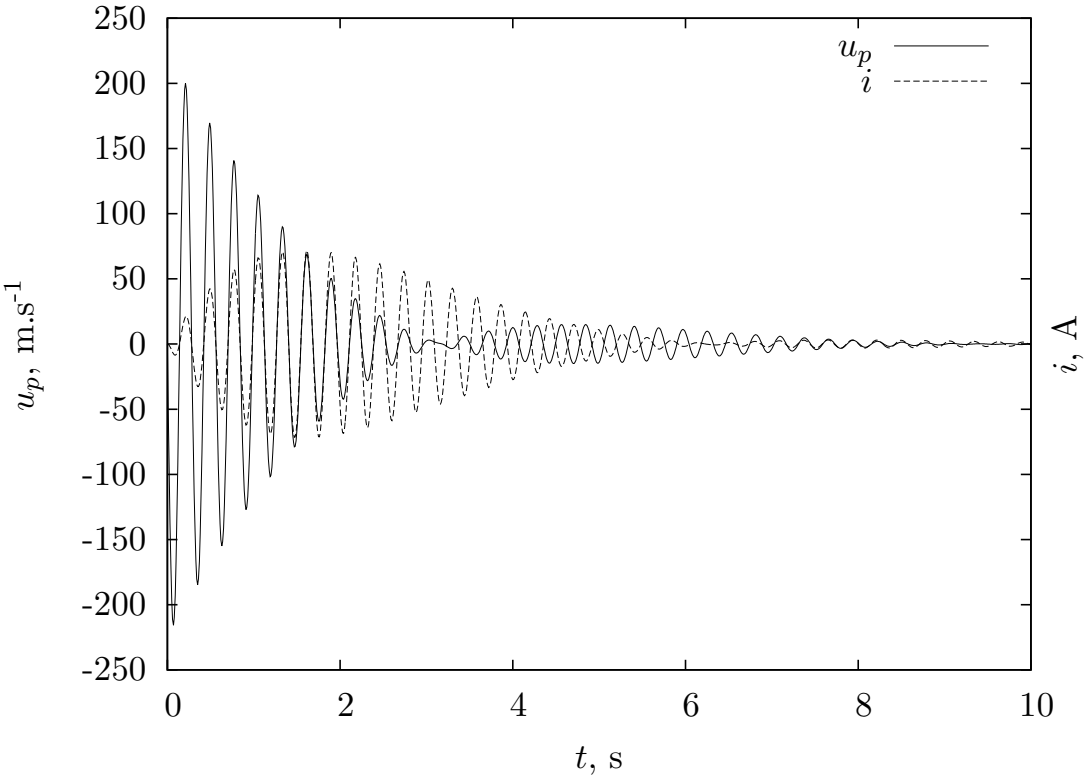


Figure 7.4: *System dynamics of a spring-mass-damper system coupled to an inductance-capacitance-resistance circuit. This shows the free response of the system given an initial displacement of the mass.*

of a free-piston engine is closer to a (nonlinear) triangle wave, which may be approximated by a superposition of sine waves[1]. It is envisaged that a number of the terms in this series could be used to mimic the actual piston trajectory such that the linear electric motor and associated power electronics may be designed in the frequency domain. Complete design of the linear electric motor and associated electronics was considered to be outside the scope of this thesis.

[1]The Fourier series for a triangle wave is

$$f(t) = \frac{8}{\pi^2} \sum_{n=1,3,5\ldots}^{\infty} \frac{(-1)^{\frac{n-1}{2}}}{n^2} \sin\left(\frac{n\pi}{L}t\right) .$$

where

$$\mathcal{L}(\sin(\omega t)) = \frac{\omega}{s^2 + \omega^2}$$

Part III

Complete engine model

Chapter 8

Engine cycle

Now that the component processes have been verified and validated using experimental data (where available), the final task was to combine them into a single program and verify the result. Once verified, this program was run using a range of engine parameters to find the set that minimised losses whilst maximising the entrained mass of CH_4.

8.1 Engine program overview

The complete free-piston engine model was created by combining

- the engine state equations (4.12) of Chapter 4
- the exhausting model state equations (5.7) of Chapter 5
- the friction model of Section 6.1
- the heat transfer model of Section 6.2.4 and
- the PID control system of Section 7.1

along with one of the gas models of Chapter 2 and either the DRM19 or GRI-Mech3.0 mechanism investigated in Chapter 3. The complete lumped-parameter engine model was then used to determine a range of suitable engine geometries.

To make the program versatile, these operations were performed using an operator-split method. *Operator-splitting* is defined here as evaluating independently the changes in the state variables due to the different coupled processes. This allowed the inclusion of one or all of these operations each timestep. It is based on the assumption that decoupling does not affect the system accuracy over a small timestep. Operator-splitting is used to simplify programs (and increase their versatility) and the associated errors are managed by selection of an appropriate timestep. This method was valuable as it allowed for both addition of operations when advancing the engine model, and removal of them when debugging.

The geometry of an engine is defined by choosing L_c and D and calculating the remaining geometry using

$$\left.\begin{aligned} L_p &= L_c + L_{rv} \\ x_{p,0} &= -0.5L_p \\ x_R &= L_c \\ x_L &= -L_c - L_p - L_{\text{gap}} \end{aligned}\right\} \tag{8.1}$$

where $L_{rv} = 0.5D$ as defined earlier. During simulations, it was found that the combustion enthalpy caused the reed valve to remain closed for too long during the expansion stroke. Extending the engine by $L_{\text{gap}} = 0.5L_c$ allowed for both pressure regeneration and effective cylinder exhausting. These variables were specified by a Lua input file. An example file is shown in Listing 8.1.

```
data = {
   u = {14.7679615940,},
   m_p = {200.0, },
   L_c = {1.10, },
   D = {0.16, },
   T_ig = {1317.2,},
   dh = 0,
   t = 0.0,
   tlast = 10,
   dt_write = 5e-4,
   dt_sys = 5e-6,
   dt_therm = 5e-6,
   tol = 1e-3,
}
```

Listing 8.1: *An example input file for an engine simulation.*

A flowchart of the complete free-piston engine program is shown in Figure 8.1 and the source code is attached as Appendix A and B.

8.2 Engine program verification

The engine model was run for a number of cycles such that it could approach a quasi-steady state. A mass and energy balance for this isentropic engine (for one cycle) is shown in Figure 8.2. During the first stroke the left-hand cylinder underwent exhausting while the right-hand cylinder underwent compression and expansion. During the next stroke these processes occurred in the opposing cylinder and so on. The piston velocity was set to $u_{p,0}$ at the beginning of each stroke to keep the engine running. Finite-rate chemistry was not included in this simulation.

The heat transfer and friction processes were then included in the engine update routine and a mass and energy balance over one stroke was repeated (see Figure 8.3).

8.3 Parametric study

The processes incorporated in the engine model, particularly the finite-rate chemistry, is computationally expensive and symmetric multiprocessing was used to conduct the parametric study.

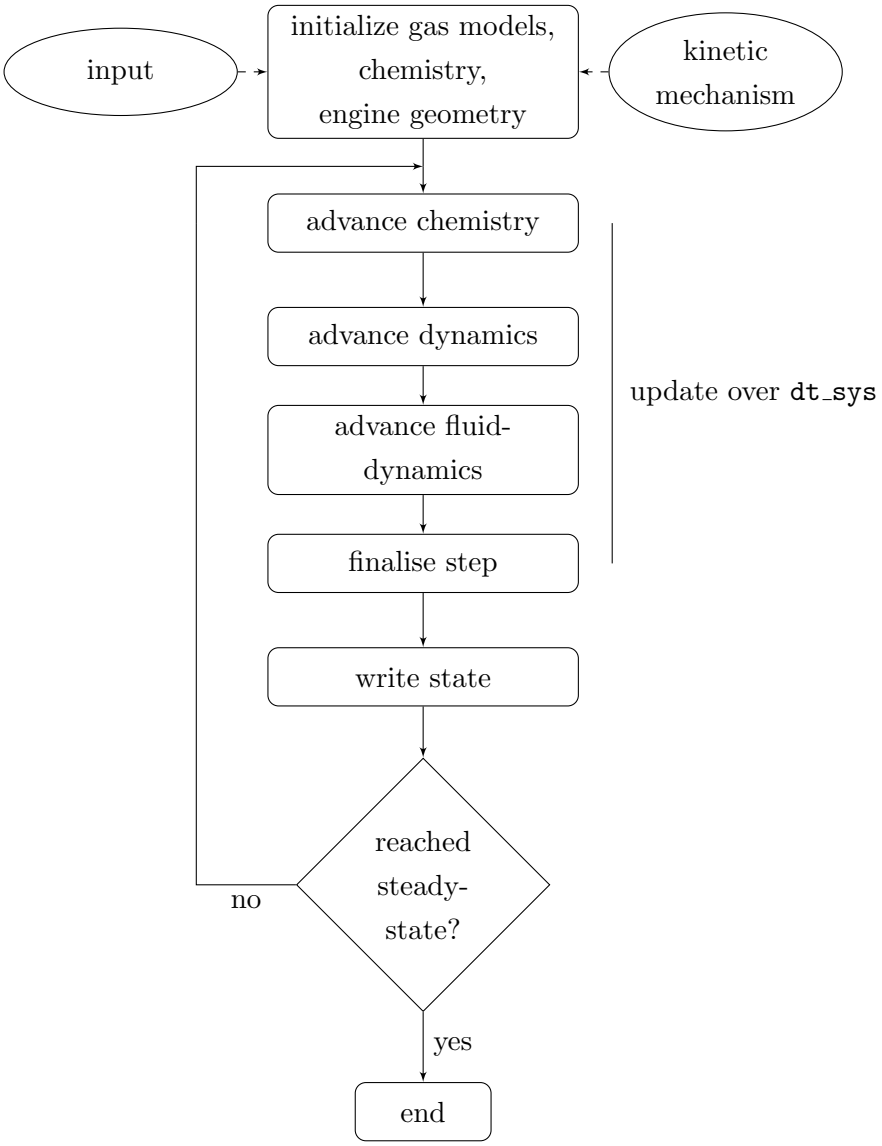


Figure 8.1: *A flowchart of the free-piston engine program showing the operator-split process.*

The aim of this study was to find the ideal operating conditions for an engine of this type, such that the entrained mass of ventilation air might be sufficient to sustain its operation.

A dual-piston type free-piston engine should ideally operate at its natural frequency. The piston velocity that corresponds to this frequency must also achieve both a high exhausting efficiency (preferably $\eta_{\text{ex}} = 100\%$) and a compression ratio sufficient for combustion. In addition to this, the heat transfer and friction loss must be less than or equal to the chemical energy of the entrained ventilation air.

Due to the low mole fraction of CH_4 in ventilation air, an exhausting efficiency of 100% was desirable. This condition was sought with the aid of a zero-finding algorithm for a range of piston masses, cylinder diameters and cylinder lengths. Figure 8.4 shows that for almost every set of engine parameters there is a small range of piston velocities between about 12 and 14 m.s^{-1}

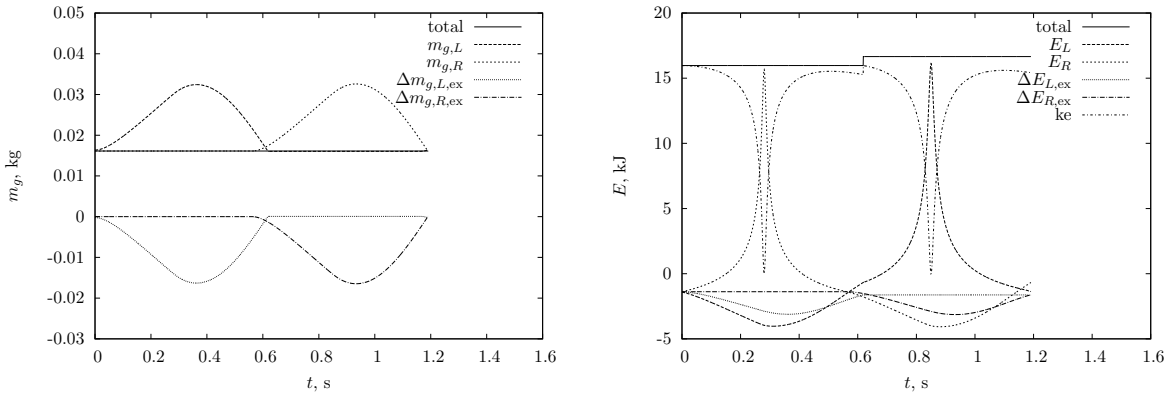


Figure 8.2: *Isentropic engine mass and energy balance. Note that the piston velocity was set to $u_{p,0}$ at the start of each stroke, which can be seen as a step-change in total energy.*

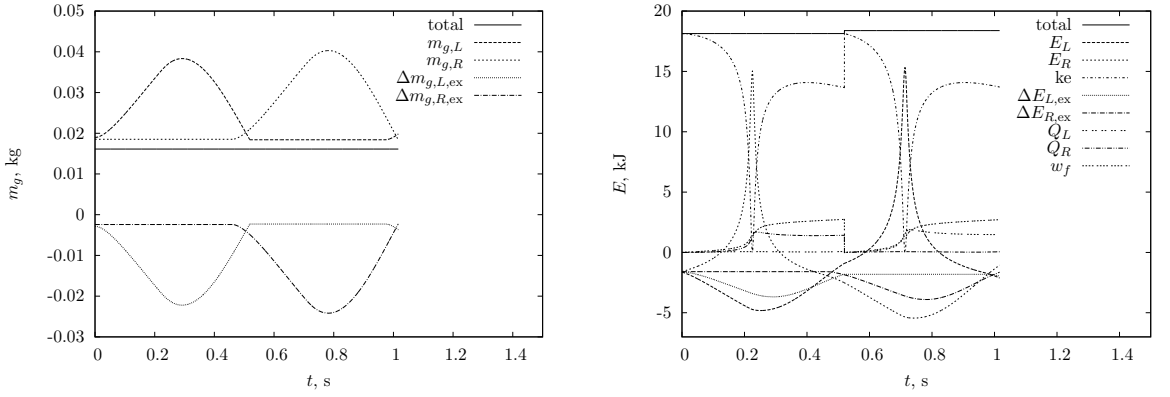


Figure 8.3: *Engine mass and energy balance with friction and heat loss. Here, the piston velocity was set to $u_{p,0}$ at the start of the stroke, which can be seen as a step-change in the kinetic energy. This is not obvious in the total system energy because w_f, Q_L and Q_R were set to zero at the same instant such that the losses over one stroke could be quantified.*

that will result in an exhausting efficiency of 100%. This range suggests that these velocities help maintain a favourable cylinder-to-atmospheric pressure ratio during the exhausting process (c.f. Figure 5.3).

The other constraint previously mentioned is that operation of such an engine is dependent on the chemical energy contained in ventilation air being greater than that the combustion enthalpy due to heat transfer and friction. This can be calculated using the specific entropy.

$$s = \frac{E_{\mathrm{loss}}}{m_g T_{\mathrm{atm}}} \tag{8.2}$$

This parameter is shown in Figure 8.5 for the same range of engines. Upon inspection of Figure 8.5, it may be said that larger diameters and shorter cylinder lengths are beneficial. This may simply be due to the larger cylinder volume and thus the larger mass of entrained methane

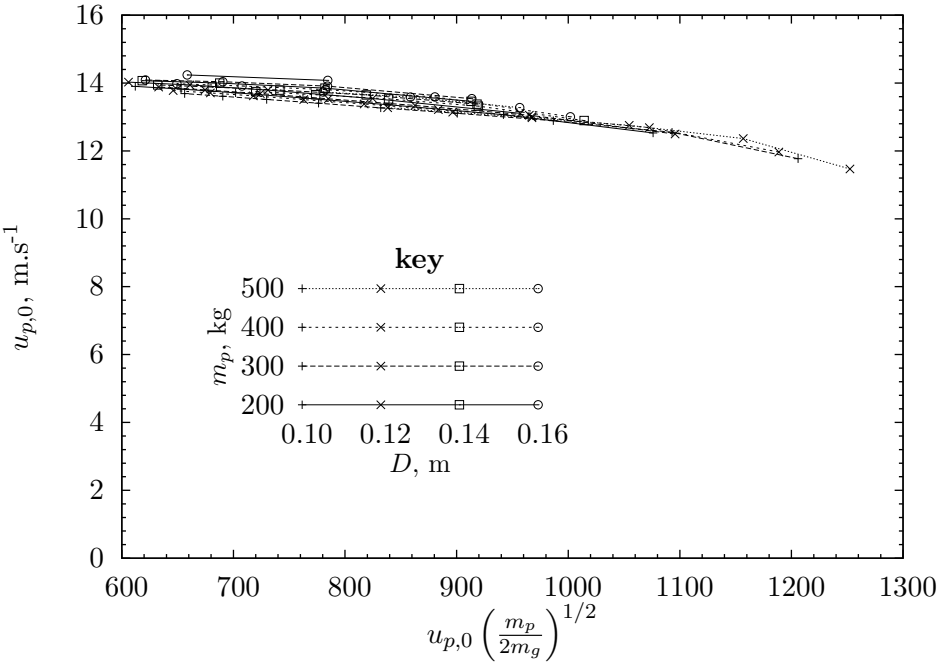


Figure 8.4: *Initial piston velocity for a range of engines operating at an exhausting efficiency of 100%. To read this plot, first pick a diameter, D, and piston mass, m_p, from the inset key. Then take this line and point style and find the corresponding line-point combination on the plot. Cylinder length, L_c, increases from 1.0m along each line from left-to-right at 0.5m increments. The initial velocity is within a remarkably small range across all engine types indicating that this velocity is favourable for exhausting.*

(given the fixed mole fraction). Further simulations showed that increasing the diameter much beyond 0.16m had no additional benefit.

Thus, the bore diameter and piston mass combination {0.16m, 200kg} was taken. For these parameters, and a piston velocity range between 12 and 14m.s^{-1}, only a limited range of stroke lengths results in a volumetric compression ratio suitable for combustion. The cylinder length parameter searched to obtain a peak temperature that would meet this criteria (see Figure 8.6). The cylinder length $L_c = 1.1$m resulted in a peak temperature of 1330 K. Thus, a near-optimum set of engine parameters was found to be:

$$\left.\begin{aligned} D &= 0.16\text{m} \\ L_c &= 1.1\text{kg} \\ m_p &= 200\text{kg} \\ u_{p,0} &= 14.77\text{m.s}^{-1}\ . \end{aligned}\right\} \tag{8.3}$$

This engine will be referred to as the reference engine. It reaches a suitable peak temperature of about 1330 K at a volumetric compression ratio of about 37 and a peak pressure of about 16.3 MPa.

The reference engine was then run with finite-rate chemistry included in the update routine.

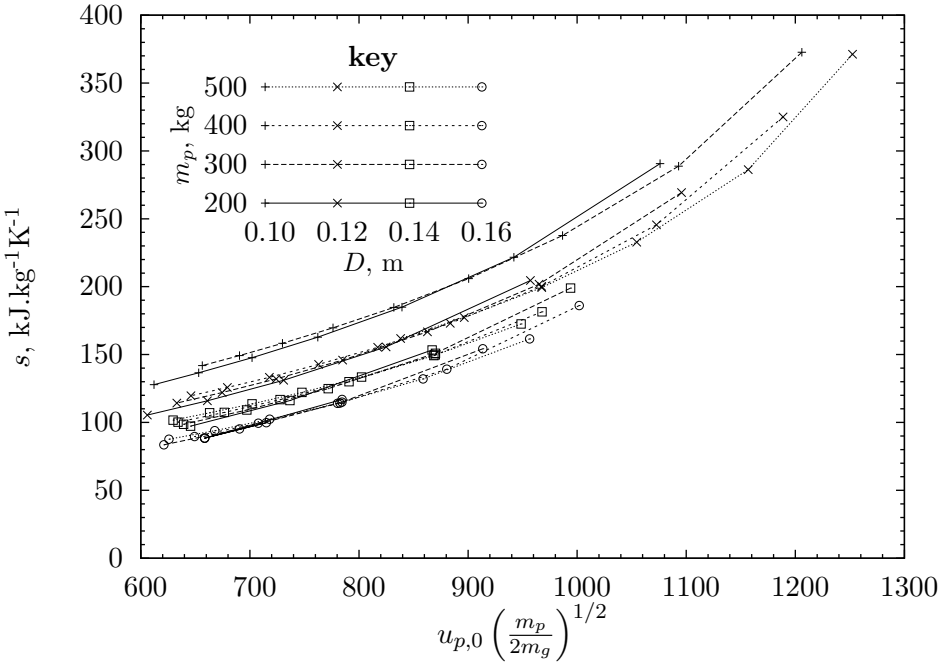


Figure 8.5: *Specific entropy for the same range of engines. To read this plot, first pick a diameter, D, and piston mass, m_p, from the inset key. Then take this line and point style and find the corresponding line-point combination on the plot. Cylinder length, L_c, increases from 1.0m along each line from left-to-right at 0.5m increments. It seems that the specific entropy reduces with increasing cylinder diameter and reducing cylinder length. Piston mass has little effect on the specific entropy, but does affect the compression ratio. The effect of diameter seems to diminish as the diameter increases. Indeed this was the case: the improvement beyond $D = 0.16$m was minimal.*

During this testing, two things were noticed: first, that a slightly higher initial velocity was required when the control system was used (as opposed to the step change in velocity applied earlier) and second, that the engine became unstable when burning CH_4 mole fractions above 0.5%, with initial conditions of $u_p = u_{p,0}$ and $\mathbf{Q}_L = \mathbf{Q}_R = \mathbf{Q}_{\text{atm}}$. Thus, it was deemed that the engine needed to be started-up like a real engine to enable it to reach a quasi-steady-state. This is shown in Figure 8.7. For this simulation, the DRM19 mechanism was used. The piston trajectory for the final stroke for this engine is shown in Figure 8.8. The work used by the electric motor during the quasi-steady state portion of this simulation was about 2 kJ per stroke. This means work is still required to keep this engine operating, even with complete combustion. The entrainment and combustion of methane for this engine is shown in Figure 8.9. Whilst this engine configuration resulted in an exhausting efficiency of $\eta_{\text{ex}} = 100\%$ without combustion, this efficiency reduced when combustion was included. The effect of combustion was to slightly reduce the exhausting efficiency. It was deemed too expensive to optimise this engine when including finite-rate chemistry in the update routine.

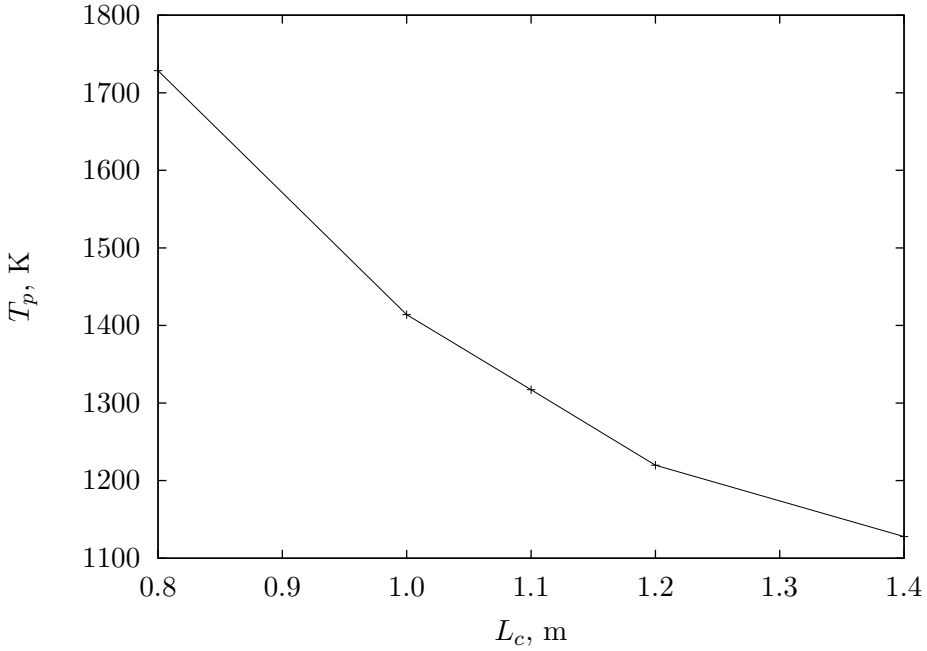


Figure 8.6: *Peak temperature reached for engines where D = 0.16m, m_p = 200kg and η_{ex} = 100%.*

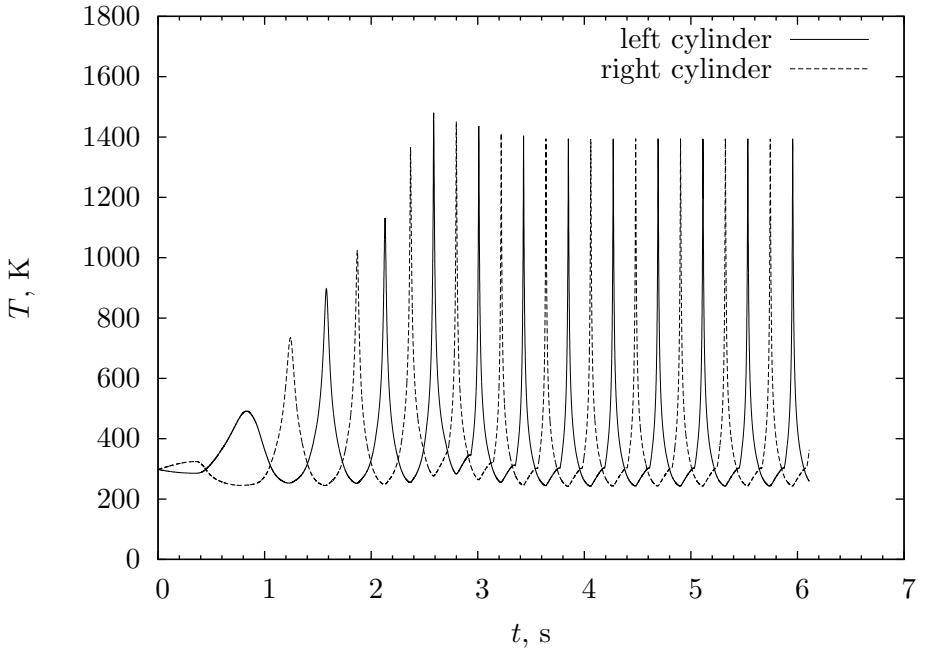


Figure 8.7: *Temperature of the left and right-hand cylinders during start-up of the reference engine. Exhausting and combustion is performed during this process.*

To ensure that nitrous oxides were not generated during this engine cycle, the GRI-Mech3.0 mechanism was also used, the species fraction and GWP of which is shown in Figure 8.10. The

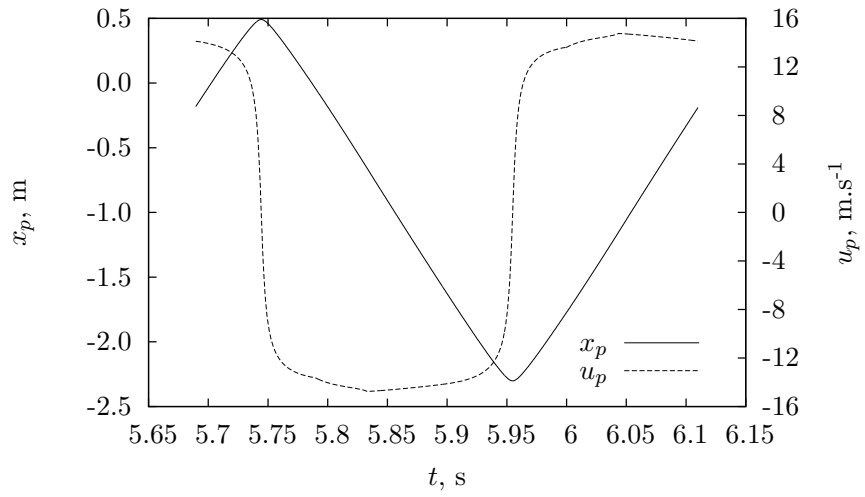


Figure 8.8: *Piston trajectory*

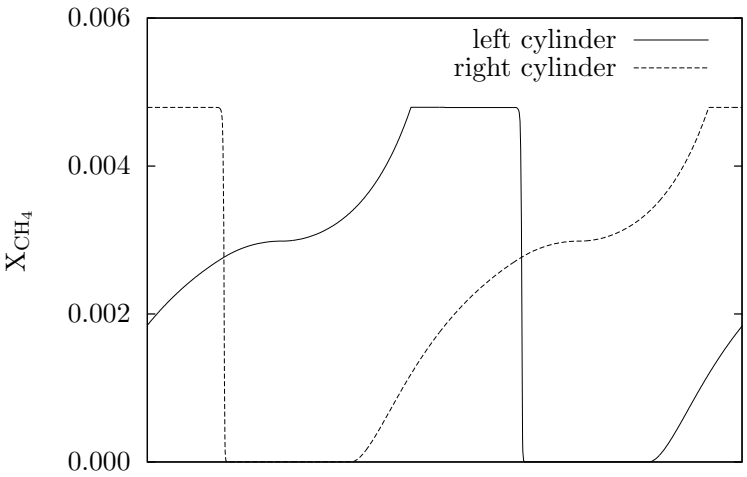


Figure 8.9: *Methane entrainment and combustion*

shortcomings of the engine model should be taken into account when considering the accuracy of this prediction. As can be seen, nitrogen chemistry is not significant. Thus it can be said that the reference engine is capable of reducing the GWP of ventilation air by 72%, but cannot operate in a self-sustaining way.

The work required to sustain operation of the reference engine with different mole fractions of CH_4 in air is shown in Figure 8.11. The average work required per stroke for these mole fractions was extrapolated to zero to find the minimum concentration of CH_4 required to sustain operation (see Figure 8.12). It was found to be about 0.92%, which is higher than that for ventilation air. As can be seen in Figure 8.11, the higher the mole fraction, the more unstable the cycles become. This is because the system became undamped at the break even point. Thus, some additional control may be required for operation of this engine above a CH_4 mole fraction of 0.92%.

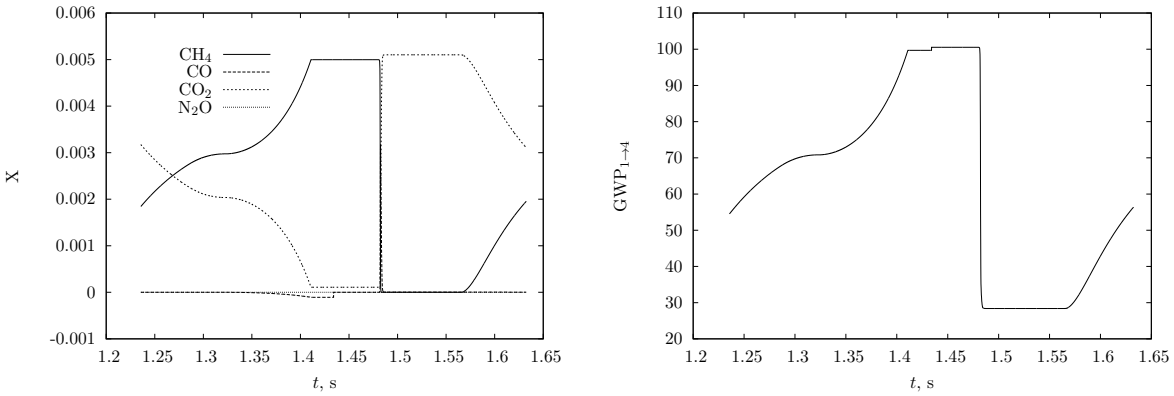


Figure 8.10: *Species and GWP for a free-piston engine operating on ventilation air. This simulation uses the GRI-Mech3.0 mechanism to show that the generation of N_2O is negligible, even for the peak temperature (without combustion) of about 1400K. As before, the piston velocity was reset at the beginning of each stroke to sustain the engine. Only data for the left-hand cylinder is shown here for clarity.*

The trends observed during the parametric study are summarised here:

1. As with the free-piston compressor, the initial normalised piston velocity and compression ratio are coupled in a free-piston engine.

2. Piston frequency is reduced by heavier pistons, smaller bore areas and longer cylinder lengths.

3. Including exhausting results in an additional dependency on piston velocity (about 12 to $14\,\mathrm{m.s^{-1}}$ being ideal for the engines tested here).

4. Including friction results in an additional dependency on the piston mass and stroke length: lower values of both piston mass and stroke length reduce friction.

5. Including heat transfer results in an additional dependency on piston frequency: higher frequency results in lower heat loss. The dependency on geometry (that is, the volume to surface area ratio) was not seen to be a dominant factor.

6. Including finite-rate chemistry results in an additional dependency on piston frequency, as lower piston frequency corresponds to higher work delay (and hence residence time).

Ultimately, while this engine has very low losses, *the mole fraction of methane in ventilation air is simply too low to sustain its operation.* If desired, the reference engine could be used to processes ventilation air by performing the additional work on the piston (via the electric motor) each stroke. In effect the engine would be partially motored.

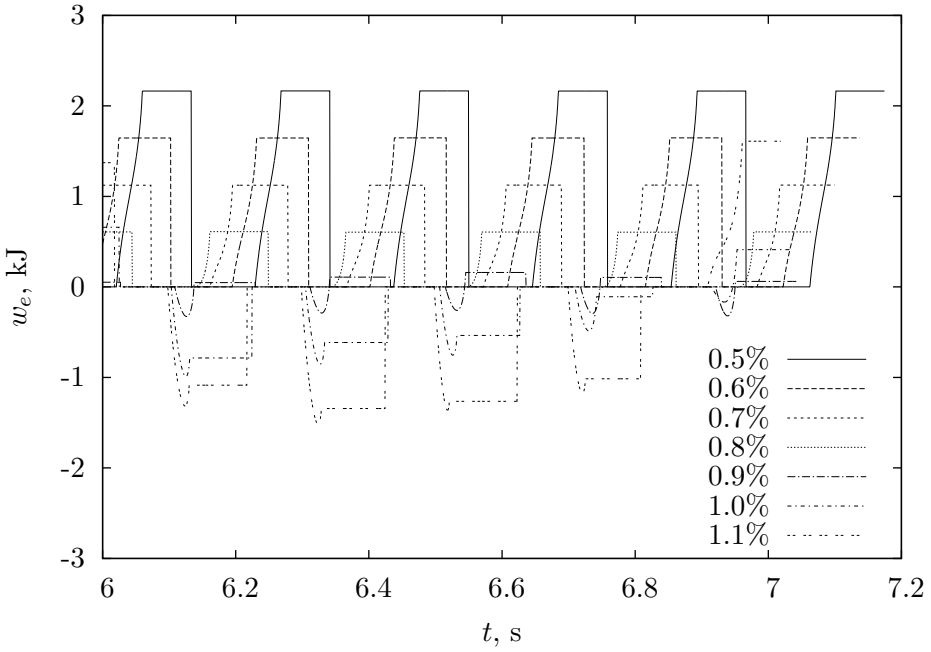


Figure 8.11: *Work required to operate engine for different mole fractions of CH_4 in air.*

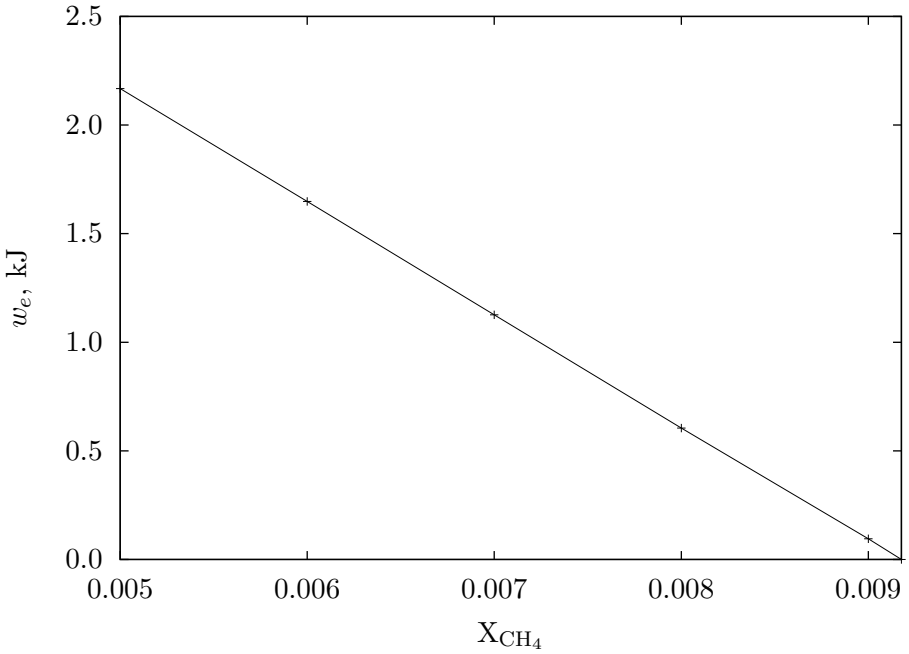


Figure 8.12: *Extrapolation was used to find the minimum mole fraction of CH_4 required to sustain operation of the reference engine. From this graph it can seen to be about 0.92%.*

Chapter 9

Conclusions

This thesis details the design of an engine whose purpose is not to generate power, but rather to process large volumes of coal mine ventilation air in order to to reduce its global warming potential. The sustained operation of this engine hinged on the condition that the chemical energy contained in ventilation air is greater than the energy that would be lost due to exhausting, heat-transfer and friction.

9.1 Summary of the reference engine design

The reference engine design was a naturally aspirated dual piston type free-piston engine. Validated models for mixed friction and unsteady heat transfer were used. The piston ring set was chosen such that the high pressures could be sealed whilst minimising the friction loss. A novel heat transfer formula was developed which captured the inversion of heat flux about top dead center. The unsteady heat flux was found to have an effect on the peak temperature and hence combustion. The total heat loss was also seen to scale well with engine speed. The resulting engine simulation program incorporated all of these processes and was run to quasi-steady-state to find an optimum thermal efficiency by varying the engine geometry and piston mass. This optimization process elucidated a few trends.

With a fixed mole fraction of methane, larger cylinder volumes of ventilation air result in a larger entrained mass of methane. This trend is limited by the ability of the engine to refuel the cylinder after each stroke and the increasing piston mass required to achieve a sufficient compression ratio. An optimum engine has an exhausting efficiency of 100%, that is, a full cylinder of ventilation air is replaced each stroke without pumping ventilation air out the poppet valve. For control of the piston velocity via the electric motor, priority was given to achieving an exhausting efficiency of $\eta_{\text{ex}} = 100\%$. The cylinder length was then chosen such that a compression ratio suitable for combustion was reached for this piston initial velocity. If the entrained mass of fuel was not a concern, then priority could conceivably have be given to controlling the peak temperature instead of exhausting, however, for this application, the reverse was required.

The optimised reference engine had a bore of 0.16 m, a cylinder length of 1.1 m, a piston mass of 200 kg and maximum piston speed of 14.77 m.s^{-1} (or a frequency of about 2.3 Hz). Ideally this engine would entrain about 0.035 kg of ventilation air each stroke, requiring at least 0.14 m^3s^{-1} for continuous operation. Each stroke had a compression ratio of 34 which resulted in a peak

temperature of about 1330 K and peak pressure of about 37 MPa. It was found that this engine required a mole fraction of CH_4 of 0.92% to sustain operation. Thus, it was *not capable* of burning ventilation air in a self-sustaining way. That said, if 15 $MJ.kg_{CH_4}{}^{-1}$ of work were added per stroke for this mole fraction, this engine may be used to burn ventilation air. In effect, the engine would have to be partially motored.

Even with motoring, only some of the coal mine ventilation air will be able to be processed. For 150 m^3s^{-1} flow rate, a bank of hundreds of engines of this size would be required. It was thought initially that larger volumes could be processed per cylinder, but the high piston masses required by these volumes made these engines impractical to pursue.

9.2 Findings

This investigation produced a number of findings for this type of engine:

1. The initial normalised piston velocity, defined as $u_{p,0}\left(\frac{m_p}{2m_g}\right)^{\frac{1}{2}}$, may be used along with Figure 4.5 to determine the compression ratio of a free-piston compressor.

2. Real gas models, as opposed to the thermally-perfect gas model, behave slightly differently during the compression process. Specifically, a van der Waals gas requires a 0.8% lower initial normalised piston velocity than a thermally-perfect gas to reach the same peak temperature.

3. Kinetic mechanisms that specify reverse reaction rates are coupled to the thermal model used during their derivation. Thus, only kinetic mechanisms where the reverse rates are calculated using the equilibrium constant are compatible with real gas models.

4. Combustion using a real gas (as opposed to a thermally-perfect gas) increases the ignition temperature of methane in a free-piston engine. In the example presented in this thesis, it increases it from approximately 1200K to 1250K, but this range will vary with piston work delay. Real gas models retard the rate of reaction, but affect neither the combustion enthalpy nor the products. Thus, the net effect is zero provided combustion goes to completion. For the design condition, the computational expense of real gas chemistry is not justified for a free-piston engine. For the parametric study performed in this thesis, this observation allowed thermally perfect gas models to be used to find the final engine geometry.

5. The effect of combustion on unsteady heat transfer acts to increase the heat transfer rate during compression and reduce it during expansion. This effect is not captured by quasi-steady models. Thus it is important to use an unsteady model when simulating compression ignition of CH_4, and a new model was developed for this.

6. Careful control of both the entrained mass of ventilation air and piston speed is required to both reach the combustion temperature and avoid exceeding material limits. For the reference engine, the peak temperature of about 1330K corresponded to a peak pressure of 20MPa.

9.3 Future work

Before a free-piston engine of this type can be operated, the following work is still recommended as a minimum:

1. the design of the linear electric motor and associated power electronics,
2. the investigation into the effect of initial conditions for ventilation air (such as humidity and ambient temperature) on engine operation,
3. a control system that can handle varying mole fractions of methane, specifically those that generate net work and
4. the development of low friction seals and lubricants.

To operate the engine solely on ventilation air, the development of low friction piston seals is a critical enabling technology. Lower friction seals may be able to be developed since the sealing demands for this engine are not as rigorous as for normal engines, where the sump oil has to be protected from gas leakage.

In addition to these areas, a number of other findings were made during this thesis that have an opportunity for future work:

1. The experimental validation of the laminar heat transfer model presented here.
2. The development of this model by including a stream function or turbulent boundary layer model.
3. The investigation of the methane oxidation pathway at very low densities and concentrations. It is suggested that under these conditions, the reaction pathway and (so ignition delay) is determined by both temperature and total density.
4. The application of the real gas functionality of `Gaspy` (developed during this thesis) to the study of other areas such as the operation of reflected shock tunnels

References

[1] O'Flaherty, B.: 2004. Mitigation of Methane from Ventilation Air using a Free-Piston Combustor. Technical Report 2005/04, Department Mechanical Engineering, University of Queensland, Australia

[2] Jacobs, P., Gollan, R., Denman, A., O'Flaherty, B., Potter, D., Petrie-Repar, P., Johnston, I.: 2010. Eilmer3 theory book. Technical Report 2010/09, Department Mechanical Engineering, University of Queensland, Australia

[3] Gollan, R., O'Flaherty, B., Jacobs, P., Johnston, I.: 2009. Casbar User's Guide. Technical Report DSTO-GD-0594, Defence Science and Technology Organisation, Edinburgh, South Australia

[4] Jacobs, P., Gollan, R., Blyton, P., Bosco, A., Boutamine, D., Brown, L., Buttsworth, D., Chan, W., Chiu, S., Craddock, C., Cook, B., Czapla, J., de Miranda-Ventura, C., Denman, A., Gildfind, D., Goozeé, R., Hess, S., Jacobs, C., Johnston, I., Joshi, O., Kirchhartz, R., McGilvray, M., Mee, D., Montgomery, L., Nap, J.-P., O'Flaherty, B., Petrie-Repar, P., Potter, D., Ramanath, D., Scott, M., Sheikh, U., Stewart, B., Tang, J., Tanimizu, K., van der Laan, P., Vesudevan, J., Wendt, M., Wheatley, V., Window, A., Wojciak, H., Zander, F.: 2008. The Eilmer3 Code: User Guide and Example Book. Technical Report 2008/07, Department Mechanical Engineering, University of Queensland, Australia

[5] Solomon, S., Qin, D., Manning, M., Chen, Z., Marquis, M., Averyt, K., Tignor, M., H.L., M.: 2007. Contribution of Working Group I to the Fourth Assessment Report of the Intergovernmental Panel on Climate Change. Technical report, Cambridge, United Kingdom and New York, NY, USA

[6] Wendt, M., Mallett, C., Lapszewicz, J., Xue, S., Foulds, G., Mark, R., Sharma, S., Dannell, R., Worrall, R., Balusu, R.: 2000. Methane Capture and Utilisation. Technical Report 723R, CSIRO

[7] Turns, S.: 2000. *An Introduction to Combustion: Concepts and Applications.* McGraw-Hill, 2nd edition

[8] Lashof, D., Ahuja, D.: 1990. Relative contributions of greenhouse gas emissions to global warming. *Nature* **344**(6266):529–531

[9] Khalil, M., Shearer, M.: 2000. Sources of Methane: An Overview. In: *Atmospheric Methane Its Role in the Global Environment*, ed. Khalil, M., pp. 98–111. Springer-Verlag

[10] Egerton, A., Powling, J.: 1948. The Limits of Flame Propagation at Atmospheric Pressure. II. The Influence of Changes in the Physical Properties. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences (1934-1990)* **193**(1033):190–209

[11] Spadaccini, L., Colket, M.: 1994. Ignition delay characteristics of methane fuels. *Progress in energy and combustion science* **20**(5):431–460

[12] Higgin, R., Williams, A.: 1969. A Shock-Tube Investigation of the Ignition of Lean Methane and n-Butane Mixtures with Oxygen. *Proceedings of the Combustion Institute* **12**:579–590

[13] Wendt, M.: 2004. personal communication

[14] Hardesty, D., Weinberg, F.: 1974. Burners Producing Large Excess Enthalpies. *Combustion Science and Technology* **8**(5):201–214

[15] Achten, P.: 1994. A Review of Free Piston Engine Concepts. *SAE Transactions* **103**(3):1836–1868

[16] Krzysztof, G., Yurii, S., Krzysztof, W., Jaschik, M., Tanczyk, M.: 2008. Homogeneous vs. catalytic combustion of lean methane — air mixtures in reverse flow reactors. *Chemical Engineering Science* **63**:5010–5019

[17] Salomons, S., Hayes, R., Poirier, M., Sapoundjiev, H.: 2003. Flow reversal reactor for the catalytic combustion of lean methane mixtures. *Catalysis Today* **83**:59–69

[18] Jones, A. R., Lloyd, S. A., Weinberg, F. J.: 1978. Combustion in heat exchangers. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences* **360**(1700):97–115

[19] Taylor, C.: 1960. *The internal-combustion engine in theory and practice.* MIT and John Wiley

[20] Stone, C.: 1992. *Introduction to Internal Combustion Engines.* SAE International

[21] Goldsborough, S., van Blarigan, P.: 1999. A numerical study of a free piston IC engine operating on homogeneous charge compression ignition combustion. *SAE Transactions* **108**(3):959–972

[22] Woschni, G.: 1967. A Universally Applicable Equation for the Instantaneous Heat Transfer Coefficient in the Internal Combustion Engine. *SAE Transactions* **76**(670931):3065–3084

[23] Mikalsen, R., Roskilly, A.: 2008. The design and simulation of a two-stroke free-piston compression ignition engine for electrical power generation. *Applied Thermal Engineering* **28**:589–600

[24] Hohenberg, G.: 1979. Advanced Approaches for Heat Transfer Calculations. In: *Diesel Engine Thermal Loading*, ed. Chu, A., pp. 17–22. SP-449

[25] Enkenhus, K., Parazzoli, C.: 1970. Dense gas phenomena in a free-piston hypersonic wind tunnel. *AIAA Journal* **8**:60–65

[26] Davidson, D., Hanson, R.: 1996. Real gas corrections in shock tube studies at high pressures. *Israel Journal of Chemistry* **36**:321–326

[27] Evlampiev, A., Somers, L. M. T., G., B. R. S., de Goey, L. P. H.: 2008. On The Impact of the Ideal Gas Assumption to High-Pressure Combustion Phenomena in Engines. *Combustion Science and Technology* **180**(2):371–390

[28] Akin, S.: 1950. The Thermodynamic Properties of Helium. *Transactions of the ASME* **72**(6):751–757

[29] Çengel, Y., Boles, M.: 2002. *Thermodynamics, An Engineering Approach.* McGraw Hill, New York, 4th edition

[30] Gordon, S., McBride, B.: 1971. Computer Probram for Calculation of Complex Chemical Equilibrium Compositions, Rocket Performance, Incident and Reflected Shocks and Chapman-Jouguet Detonations. Technical Report NASA SP-273, NASA

[31] Kee, R., Rupley, F., Miller, J.: 1993. The Chemkin Thermodynamic Data Base. Technical Report SAND87-8215B, Sandia National Laboratories, Livermore, California

[32] Chase, M. R. J., Davies, C. A., Downey, J. J.: 1985. JANAF Thermochemical Tables. *Journal of Physical and Chemical Reference Data* **14**(1)

[33] Annand, W.: 1963. Heat transfer in the cylinders of reciprocating internal combustion engines. *Proceedings of the Institution of Mechanical Engineers* **177**(36):973–990

[34] Poling, B., Prausnitz, J., O'Connell, J.: 2001. *The Properties of Gases and Liquids.* McGraw Hill, 5th edition

[35] Sandler, S.: 1999. *Chemical and Engineering Thermodynamics.* John Wiley & Sons, 3rd edition

[36] Tsuboi, T., Wagner, H.: 1975. Homogeneous Thermal Oxidation of Methane in Reflected Shock Waves. In: *Fifteenth Symposium (International) on Combustion*, pp. 883–890

[37] Petersen, E., Röhrig, M., Davidson, D., Hanson, R., Bowman, C.: 1996. High-Pressure Methane Oxidation Behind Reflected Shock Waves. In: *Twenty-Sixth Symposium (International) on Combustion*, pp. 799–806

[38] Frenklach, M. Reduction of chemical reaction models. In: *Numerical approaches to combustion modeling*, eds. Oran, E., Boris, J.

[39] Li, S., Williams, F.: 2002. Reaction Mechanisms for Methane Ignition. *Journal of Engineering for Gas Turbines and Power* **124**:471

[40] Smooke, D. M., Giovangigli, V.: 1991. Formulation of the Premixed and Nonpremixed Test Problems. In: *Lecture Notes in Physics*, volume 384, ed. Smooke, M. D., pp. 1–28. Springer-Verlag

[41] Jazbec, M., Fletcher, D., Haynes, B.: 2000. Simulation of the ignition of lean methane mixtures using CFD modelling and a reduced chemistry mechanism. *Applied Mathematical Modelling* **24**(8-9):689–696

[42] Kazakov, A., Frenklach, M. DRM19. `http://www.me.berkeley.edu/drm/drm19.dat`. [Online; accessed July 2008]

[43] Kazakov, A., Frenklach, M. DRM22. `http://www.me.berkeley.edu/drm/drm22.dat`. [Online; accessed July 2008]

[44] Smith, G. P., Golden, D. M., Frenklach, M., Moriarty, N. W., Eiteneer, B., Goldenberg, M., Bowman, T. C., Hanson, R. K., Song, S., Gardiner, W. C. J., Lissianski, V. V., Qin, Z. GRI-Mech3.0. `http://www.me.berkeley.edu/gri\_mech/`. [Online; accessed June 2008]

[45] Gollan, R.: 2003. Yet Another Finite-Rate Chemistry Module for Compressible Flow Codes. Technical Report 2003/09, Department Mechanical Engineering, University of Queensland, Australia

[46] Anderson, J.: 2003. *Modern Compressible Flow with Historical Perspective*. McGraw Hill, 3rd edition

[47] Gilbert, R. G., Luther, K., Troe, J.: 1983. Theory of thermal unimolecular reactions in the fall-off range. II. Weak collision rate constants. *Berichte der Bunsengesellschaft fr physikalische Chemie* **87**(2):169–177

[48] Livengood, J., Leary, W.: 1951. Autoignition by Rapid Compression. *Industrial & Engineering Chemistry* **43**(12):2797–2805

[49] Kondratiev, V.: 1965. Determination of the rate constant for thermal cracking of methane by means of adiabatic compression and expansion. In: *Tenth Symposium (International) on Combustion*, pp. 319–322

[50] Krishnan, S., Ravikumar, R.: 1981. Ignition Delay of Methane in Reflected Shock Waves. *Combustion Science and Technology* **24**(5):239–245

[51] Lifshitz, A., Scheller, K., Burcat, A., Skinner, G.: 1971. Shock-Tube Investigation of Ignition in Methane-Oxygen-Argon Mixtures. *Combustion and Flame* **16**(3):311–321

[52] Seery, D., Bowman, C.: 1970. An Experimental and Analytical Study of Methane Oxidation Behind Shock Waves. *Combustion and Flame* **14**(1):37–48

[53] Hartig, R., Troe, J., Wagner, H.: 1971. Thermal Decomposition of Methane Behind Reflected Shock Waves. In: *Thirteenth Symposium (International) on Combustion*

[54] Gordon, S., McBride, B. J.: 1996. Computer program for calculation of complex chemical equilibrium compositions and applications. Technical report, NASA Reference Publication 1311

[55] Liseikin, V. D.: 1999. *Grid Generation Methods*. Springer-Verlag, Berlin

[56] Kee, R., Rupley, F., Miller, J.: 1993. Chemkin-II: A Fortran Chemical Kinetics package for the analysis of gas phase chemical kinetics. Technical Report SAND-89-8009, Sandia National Laboratories, Livermore, California

[57] Zallen, D., Wittig, S.: 1975. Effects of Nitrogen on the Shock Induced Ignition of Methane. In: *Tenth International Shock Tube Symposium*, ed. Kamimoto, G., pp. 640–647

[58] Grillo, A., Slack, M.: 1976. Shock Tube Study of Ignition Delay Times in Methane-Oxygen-Nitrogen-Argon Mixtures. *Combustion and Flame* **27**(3):377–381

[59] Pilling, M., Seakins, P.: 1995. *Reaction Kinetics*. Oxford University Press

[60] Hinshelwood, C.: 1926. On the theory of unimolecular reactions. *Proceedings of the Royal Society of London. Series A* **113**(763):230–233

[61] Bilger, R., Esler, M., Starner, S.: 1991. On Reduced Mechanisms for Methane-Air Combustion. In: *Lecture Notes in Physics*, volume 384, ed. Smooke, M. D., pp. 86–110. Springer-Verlag

[62] Annand, W., Roe, G.: 1974. *Gas flow in the internal combustion engine*. G.T. Foulis

[63] Goldsborough, S.: 2004. *Optimizing the Scavenging System for High Efficiency and Low Emissions : a Computational Approach*. Ph.D. thesis, Colorado State University

[64] Sod, G. A survey of several finite difference methods for systems of nonlinear hyperbolic conservation laws. *Journal of Computational Physics*

[65] Zhu, Y., Reitz, R.: 1999. A 1-D gas dynamcis code for subsonic and supersonic flows applied to predict EGR levels in a heavy-duty diesel engine. *International Journal of Vehicle Design* **22**(3/4):227–252

[66] Gustafsson, B., Kreiss, H.-O., Oliger, J.: 1995. *Time Dependent Problems and Difference Methods*. Wiley-Interscience

[67] Patir, N., Cheng, H. S.: 1978. Average Flow Model for Determining Effects of 3-Dimensional Roughness on Partial Hydrodynamic Lubrication. *Journal of Lubrication Technology — Transactions of the ASME* **100**(1):12–17

[68] Groth, C. P. T., Gottlieb, J. J.: 1988. Numerical Study of Two-Stage Light-Gas Hypervelocity Projectile Launchers. Technical Report 327, University of Toronto

[69] McGeehan, J.: 1987. A Literature Review of the Effects of Piston and Ring Friction and Lubricating Oil Viscosity on Fuel Economy. *SAE Transactions* **87**(780673):2619–2638

[70] Woschni, G., Spindler, W.: 1988. Heat Transfer With Insulated Combustion Chamber Walls and Its Influence on the Performance of Diesel Engines. *Transactions of the ASME* **110**:482–502

[71] Lawton, B.: 1987. Effect of compression and expansion on instantaneous heat transfer in reciprocating internal combustion engines. *Proceedings of the Institution of Mechanical Engineers. Part A. Power and process engineering* **201**(3):175–186

[72] Sihling, K., Woschni, G.: 1979. Experimental investigation of the instantaneous heat transfer in the cylinder of a high speed diesel engine. In: *Diesel Engine Thermal Loading*, ed. Chu, A., pp. 95–102. SP-449

[73] Annand, W., Pinfold, D.: 1980. Heat Transfer in the Cylinder of a Motored Reciprocating Engine. *SAE Technical Paper Series* (800457):1–6

[74] Kornhauser, A., Smith, J.: 1994. Application of a complex Nusselt number to heat transfer during compression and expansion. *Journal of heat transfer* **116**(3):536–542

[75] Greif, R., Namba, T., Nikanham, M.: 1979. Heat transfer during piston compression including side wall and convection effects. *International Journal of Heat and Mass Transfer* **22**(6):901–907

[76] Knöös, S.: 1971. Theoretical and experimental study of piston gas-heating with laminar energy losses. *AIAA Journal* **9**:2119–2127

[77] Yang, J., Pierce, P., Martin, J., Foster, D.: 1989. Heat Transfer Predictions and Experiments in a Motored Engine. *SAE Transactions* **97**:1608–1622

[78] Dao, K., Uyehara, O., Myers, P.: 1973. Heat Transfer Rates at Gas-Wall Interfaces in Motored Piston Engine. *SAE Transactions* **82**(730632):2237–2258

[79] Edney, B.: 1967. Temperature measurements in a hypersonic gun tunnel using heat-transfer methods. *Journal of fluid mechanics* **27**(3):503–512

[80] Buttsworth, D.: 2002. Heat Transfer During Transient Compression: Measurements and Simulations. *Shock Waves* **12**(2002):87–91

[81] Borman, G., Nishiwaki, K.: 1987. Internal-combustion engine heat transfer. *Progress in energy and combustion science* **13**(1):1–46

[82] Wendland, D.: 1968. The effect of periodic pressure and temperature fluctuations on unsteady heat transfer in a closed system. Technical Report 19680012217, NASA

[83] LeFeuvre, T., Myers, P., Uyehara, O.: 1969. Experimental Instantaneous Heat Fluxes in a Diesel Engine and Their Correlation. *SAE Transactions* **78**(690464):1717–1738

[84] Annand, W., Ma, T.: 1970. Instantaneous heat transfer rates to the cylinder head surface of a small compression-ignition engine. *Proceedings of the Institution of Mechanical Engineers* **71**(185):976–987

[85] Han, Z., Reitz, R.: 1997. A temperature wall function formulation for variable-density turbulent flows with application to engine convective heat transfer modeling. *International journal of heat and mass transfer* **40**(3):613–625

[86] Yang, J., Martin, J.: 1989. Approximate Solution— One-Dimensional Energy Equation for Transient, Compressible, Low Mach Number Turbulent Boundary Layer Flows. *Journal of Heat Transfer* **111**(3):619–624

[87] Schlichting, H.: 1968. *Boundary-layer Theory.* McGraw-Hill

[88] Tani, K., Itoh, K., Takahashi, M., Tanno, H., Komuro, T., Miyajima, H.: 1994. Numerical study of free-piston shock tunnel performance. *Shock Waves* **3**(4):313–319

[89] Yang, J.: 1995. IC Engine Gas-Wall Convective Heat Transfer — History, Problems, and Solutions. *Transport Phenomena in Combustion* **2**:1611–1621

[90] Isshiki, N., Nishiwaki, N.: 1970. Study on laminar heat transfer of inside gas with cyclic pressure change on an inner wall of a cylinder head. *Heat Transfer* pp. 1–10

[91] Kamel, M., Watson, N.: 1979. Heat Transfer in the Indirect Injection Diesel Engine. In: *Diesel Engine Thermal Loading*, ed. Chu, A., pp. 81–94. SP-449

[92] Aichlmayr, H., Kittelson, D., Zachariah, M.: 2002. Miniature Free-Piston Homogeneous Charge Compression Ignition Engine-Compressor Concept – Part II: Modeling HCCI Combustion in Small Scales with Detailed Homogeneous Gas Phase Chemical Kinetics. *Chemical Engineering Science* **57**(19):4173–4186

Appendix A

Gas model source code

A.1 Real equations of state

```
// \author: Brendan T. O'Flaherty
// \brief: Abel-Noble equation of state

#ifndef NOBLE_ABEL_GAS_EOS_HH
#define NOBLE_ABEL_GAS_EOS_HH

extern "C" {
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>
}
#include "gas_data.hh"
#include "equation-of-state.hh"

class Noble_Abel_gas : public Equation_of_state {
public:
    Noble_Abel_gas(lua_State *L);
    ~Noble_Abel_gas();

    // python function
    double covolume(gas_data Q) { int status; return s_covolume(Q, status); }

private:
    std::vector<double> R_;
    std::vector<double> nu_0_;

    int s_eval_pressure(gas_data &Q);
    int s_eval_temperature(gas_data &Q);
    int s_eval_density(gas_data &Q);
    double s_gas_constant(const gas_data &Q, int &status);
    double s_prho_ratio(const gas_data &Q, int isp);
    double s_dTdp_const_rho(const gas_data &Q, int &status);
    double s_dTdrho_const_p(const gas_data &Q, int &status);
    double s_dpdrho_const_T(const gas_data &Q, int &status);
    double s_dpdrho_i_const_T(const gas_data &Q, int isp, int &status);
    double s_dpdT_i_const_rho(const gas_data &Q, int itm, int &status);
    double s_covolume(const gas_data &Q, int &status);
};

double nag_pressure(double rho, double T, double R, double nu_0);
double nag_temperature(double rho, double p, double R, double nu_0);
```

```
double nag_density(double T, double p, double R, double nu_0);

#endif
```

Listing A.1: *Abel-Noble equation of state, header file.*

```
#include <iostream>
#include <sstream>
#include <cmath>
#include <cstdlib>
#include "../../util/source/useful.h"
#include "../../util/source/lua_service.hh"
#include "physical_constants.hh"
#include "noble-abel-gas-EOS.hh"

using namespace std;

Noble_Abel_gas::
Noble_Abel_gas(lua_State *L)
    : Equation_of_state()
{
    lua_getglobal(L, "species");

    if ( !lua_istable(L, -1) ) {
        ostringstream ost;
        ost << "Noble_Abel_gas::Noble_Abel_gas():\n";
        ost << "Error in the declaration of species: a table is expected.\n";
        input_error(ost);
    }

    int nsp = lua_objlen(L, -1);

    for ( int isp = 0; isp < nsp; ++isp ) {
        lua_rawgeti(L, -1, isp+1); // A Lua list is offset one from the C++ vector index
        const char* sp = luaL_checkstring(L, -1);
        lua_pop(L, 1);

        // Now bring the specific species table to TOS
        lua_getglobal(L, sp);
        if ( !lua_istable(L, -1) ) {
            ostringstream ost;
            ost << "Noble_Abel_gas::Noble_Abel_gas()\n";
            ost << "Error locating information table for species: " << sp << endl;
            input_error(ost);
        }

        double M = get_positive_value(L, -1, "M");
        M_.push_back(M);
        R_.push_back(PC_R_u/M);

        double T_c = get_value(L, -1, "T_c");
        double p_c = get_positive_value(L, -1, "p_c");

        double nu_0 = 0.125*PC_R_u*T_c/p_c;
        nu_0_.push_back(nu_0);
        lua_pop(L, 1); // pop "sp" off stack
    }
```

```
    lua_pop(L, 1); // pop "species" off stack
}

Noble_Abel_gas::
~Noble_Abel_gas() {}

int
Noble_Abel_gas::
s_eval_pressure(gas_data &Q)
{
    int status;
    double R = s_gas_constant(Q, status);
    if ( status != SUCCESS )
        return status;
    double nu_0 = s_covolume(Q, status);
    if ( status != SUCCESS )
        return status;
    // else proceed
    Q.p = nag_pressure(Q.rho, Q.T[0], R, nu_0);
    return SUCCESS;
}

int
Noble_Abel_gas::
s_eval_temperature(gas_data &Q)
{
    int status;
    double R = s_gas_constant(Q, status);
    if ( status != SUCCESS )
        return status;
    double nu_0 = s_covolume(Q, status);
    if ( status != SUCCESS )
        return status;
    // else proceed
    Q.T[0] = nag_temperature(Q.rho, Q.p, R, nu_0);
    return SUCCESS;
}

int
Noble_Abel_gas::
s_eval_density(gas_data &Q)
{
    int status;
    double R = s_gas_constant(Q, status);
    if ( status != SUCCESS )
        return status;
    double nu_0 = s_covolume(Q, status);
    if ( status != SUCCESS )
        return status;
    // else proceed
    Q.rho = nag_density(Q.T[0], Q.p, R, nu_0);
    return SUCCESS;
}

double
Noble_Abel_gas::
s_gas_constant(const gas_data &Q, int &status)
```

```
{
    status = SUCCESS;
    return calculate_gas_constant(Q.massf, M_);
}

double
Noble_Abel_gas::
s_prho_ratio(const gas_data &Q, int isp)
{
    double M = calculate_molecular_weight(Q.massf, M_);
    return Q.p/Q.rho*M/M_[isp];
}

double
Noble_Abel_gas::
s_dTdp_const_rho(const gas_data &Q, int &status)
{
    double R = s_gas_constant(Q, status);
    double nu_0 = s_covolume(Q, status);
    double dTdp_const_nu = (1.0/Q.rho - nu_0)/R;
    return dTdp_const_nu;
}

double
Noble_Abel_gas::
s_dTdrho_const_p(const gas_data &Q, int &status)
{
    double R = s_gas_constant(Q, status);
    double dTdnu_const_p = Q.p/R;
    return -1.0/(Q.rho*Q.rho)*dTdnu_const_p;
}

double
Noble_Abel_gas::
s_dpdrho_const_T(const gas_data &Q, int &status)
{
    double R = s_gas_constant(Q, status);
    double nu_0 = s_covolume(Q, status);
    double dpdnu_const_T = -R*Q.T[0]*pow((1/Q.rho - nu_0), -2);
    return -1.0/(Q.rho*Q.rho)*dpdnu_const_T;
}

double
Noble_Abel_gas::
s_dpdrho_i_const_T(const gas_data &Q, int isp, int &status)
{
    double R_i = PC_R_u / M_[isp];
    double nu_0_i = nu_0_[isp];
    double rho_i = Q.rho * Q.massf[isp];
    double dpdnu_i_const_T = -R_i*Q.T[0]*pow((1/rho_i - nu_0_i), -2);
    return -1.0/(rho_i*rho_i)*dpdnu_i_const_T;
}

double
Noble_Abel_gas::
s_dpdT_i_const_rho(const gas_data &Q, int itm, int &status)
{
```

```
    double R = s_gas_constant(Q, status);
    double nu_0 = s_covolume(Q, status);
    double dTdp_const_nu = (1.0/Q.rho - nu_0)/R;
    return 1.0/dTdp_const_nu;
}

double
Noble_Abel_gas::
s_covolume(const gas_data &Q, int &status)
{
    vector<double> molef;
    molef.resize(M_.size());
    convert_massf2molef(Q.massf, M_, molef);

    double nu_0 = 0.0;
    for ( size_t isp = 0; isp < Q.massf.size(); ++isp ) {
        nu_0 += molef[isp]*nu_0_[isp];
    }

    double M = calculate_molecular_weight(Q.massf, M_);

    status = SUCCESS;
    return nu_0/M;
}

double nag_pressure(double rho, double T, double R, double nu_0)
{
    double nu_1 = 1/rho - nu_0;
    if (nu_1 <= 0) {
        cout << "A Noble-Abel gas cannot have a specific-volume smaller than the co-
            volume.\n";
        cout << "Bailing out!\n";
        exit(BAD_INPUT_ERROR);
    }
    return R*T/nu_1;
}

double nag_temperature(double rho, double p, double R, double nu_0)
{
    double nu_1 = 1/rho - nu_0;
    if (nu_1 <= 0) {
        cout << "A Noble-Abel gas cannot have a specific-volume smaller than the co-
            volume.\n";
        cout << "Bailing out!\n";
        exit(BAD_INPUT_ERROR);
    }
    return p*nu_1/R;
}

double nag_density(double T, double p, double R, double nu_0)
{
    double nu = R*T/p + nu_0;
    return 1/nu;
}
```

Listing A.2: *Abel-Noble equation of state, source file.*

```
// \author: Brendan T. O'Flaherty
// \brief: van der Waals equation of state

#ifndef VAN_DER_WAALS_GAS_EOS_HH
#define VAN_DER_WAALS_GAS_EOS_HH

extern "C" {
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>
}
#include "gas_data.hh"
#include "equation-of-state.hh"

class van_der_Waals_gas : public Equation_of_state {
public:
    van_der_Waals_gas(lua_State *L);
    ~van_der_Waals_gas();

    // python function
    double covolume(gas_data Q) { int status; return s_covolume(Q, status); }
    double a(gas_data Q) { int status; return s_a(Q, status); }

private:
    std::vector<double> R_;
    std::vector<double> nu_0_;
    std::vector<double> a_;

    int s_eval_pressure(gas_data &Q);
    int s_eval_temperature(gas_data &Q);
    int s_eval_density(gas_data &Q);
    double s_gas_constant(const gas_data &Q, int &status);
    double s_prho_ratio(const gas_data &Q, int isp);
    double s_dTdp_const_rho(const gas_data &Q, int &status);
    double s_dTdrho_const_p(const gas_data &Q, int &status);
    double s_dpdrho_const_T(const gas_data &Q, int &status);
    double s_dpdrho_i_const_T(const gas_data &Q, int isp, int &status);
    double s_dpdT_i_const_rho(const gas_data &Q, int itm, int &status);
    double s_covolume(const gas_data &Q, int &status);
    double s_a(const gas_data &Q, int &status);
};

double vdwg_pressure(double rho, double T, double R, double nu_0, double a);
double vdwg_temperature(double rho, double p, double R, double nu_0, double a);
double vdwg_density(double T, double p, double R, double nu_0, double a);

#endif
```

Listing A.3: *van der Waals equation of state, header file.*

```
#include <iostream>
#include <sstream>
#include <cmath>
#include <cstdlib>
#include "../../util/source/useful.h"
#include "../../util/source/lua_service.hh"
#include "physical_constants.hh"
```

```
#include "van-der-waals-gas-EOS.hh"

using namespace std;

van_der_Waals_gas::
van_der_Waals_gas(lua_State *L)
    : Equation_of_state()
{
    lua_getglobal(L, "species");

    if ( !lua_istable(L, -1) ) {
        ostringstream ost;
        ost << "van_der_Waals_gas::van_der_Waals_gas():\n";
        ost << "Error in the declaration of species: a table is expected.\n";
        input_error(ost);
    }

    int nsp = lua_objlen(L, -1);

    for ( int isp = 0; isp < nsp; ++isp ) {
        lua_rawgeti(L, -1, isp+1); // A Lua list is offset one from the C++ vector index
        const char* sp = luaL_checkstring(L, -1);
        lua_pop(L, 1);

        // Now bring the specific species table to TOS
        lua_getglobal(L, sp);
        if ( !lua_istable(L, -1) ) {
            ostringstream ost;
            ost << "van_der_Waals_gas::van_der_Waals_gas()\n";
            ost << "Error locating information table for species: " << sp << endl;
            input_error(ost);
        }

        double M = get_positive_value(L, -1, "M");
        M_.push_back(M);
        R_.push_back(PC_R_u/M);

        double T_c = get_value(L, -1, "T_c");
        double p_c = get_positive_value(L, -1, "p_c");

        double nu_0 = 0.125*PC_R_u*T_c/p_c;
        nu_0_.push_back(nu_0);

        double a = (27.0/64.0)*((PC_R_u*T_c)*(PC_R_u*T_c))/p_c;
        a_.push_back(a);

        lua_pop(L, 1); // pop "sp" off stack
    }
    lua_pop(L, 1); // pop "species" off stack
}

van_der_Waals_gas::
~van_der_Waals_gas() {}

int
van_der_Waals_gas::
s_eval_pressure(gas_data &Q)
```

```
{
    int status;
    double R = s_gas_constant(Q, status);
    if ( status != SUCCESS )
        return status;
    double nu_0 = s_covolume(Q, status);
    if ( status != SUCCESS )
        return status;
    double a = s_a(Q, status);
    if ( status != SUCCESS )
        return status;
    // else proceed
    Q.p = vdwg_pressure(Q.rho, Q.T[0], R, nu_0, a);
    return SUCCESS;
}

int
van_der_Waals_gas::
s_eval_temperature(gas_data &Q)
{
    int status;
    double R = s_gas_constant(Q, status);
    if ( status != SUCCESS )
        return status;
    double nu_0 = s_covolume(Q, status);
    if ( status != SUCCESS )
        return status;
    double a = s_a(Q, status);
    if ( status != SUCCESS )
        return status;
    // else proceed
    Q.T[0] = vdwg_temperature(Q.rho, Q.p, R, nu_0, a);
    return SUCCESS;
}

int
van_der_Waals_gas::
s_eval_density(gas_data &Q)
{
    int status;
    double R = s_gas_constant(Q, status);
    if ( status != SUCCESS )
        return status;
    double nu_0 = s_covolume(Q, status);
    if ( status != SUCCESS )
        return status;
    double a = s_a(Q, status);
    if ( status != SUCCESS )
        return status;
    // else proceed
    Q.rho = vdwg_density(Q.T[0], Q.p, R, nu_0, a);
    return SUCCESS;
}

double
van_der_Waals_gas::
s_gas_constant(const gas_data &Q, int &status)
```

```
{
    status = SUCCESS;
    return calculate_gas_constant(Q.massf, M_);
}

double
van_der_Waals_gas::
s_prho_ratio(const gas_data &Q, int isp)
{
    double M = calculate_molecular_weight(Q.massf, M_);
    return Q.p/Q.rho*M/M_[isp];
}

double
van_der_Waals_gas::
s_dTdp_const_rho(const gas_data &Q, int &status)
{
    double R = s_gas_constant(Q, status);
    double nu = 1/Q.rho;
    double nu_0 = s_covolume(Q, status);
    return (nu - nu_0)/R;
}

double
van_der_Waals_gas::
s_dTdrho_const_p(const gas_data &Q, int &status)
{
    double R = s_gas_constant(Q, status);
    double a = s_a(Q, status);
    double nu = 1/Q.rho;
    double nu_0 = s_covolume(Q, status);
    double dTdnu_const_p = (1/R)*(Q.p - a/(nu*nu*nu)*(nu + 2*nu_0));
    return -nu*nu*dTdnu_const_p;
}

double
van_der_Waals_gas::
s_dpdrho_const_T(const gas_data &Q, int &status)
{
    double R = s_gas_constant(Q, status);
    double a = s_a(Q, status);
    double nu = 1/Q.rho;
    double nu_0 = s_covolume(Q, status);
    double dpdnu_const_T = -R*Q.T[0]/((nu - nu_0)*(nu - nu_0)) - 2*a/nu;
    return -nu*nu*dpdnu_const_T;
}

double
van_der_Waals_gas::
s_dpdrho_i_const_T(const gas_data &Q, int isp, int &status)
{
    double R_i = PC_R_u / M_[isp];
    double a_i = a_[isp];
    double nu_i = 1/(Q.rho*Q.massf[isp]);
    double nu_0_i = nu_0_[isp];
    double dpdnu_i_const_T = -R_i*Q.T[0]/((nu_i - nu_0_i)*(nu_i - nu_0_i)) - 2*a_i/nu_i;
    return -nu_i*nu_i*dpdnu_i_const_T;
```

```
}


double
van_der_Waals_gas::
s_dpdT_i_const_rho(const gas_data &Q, int itm, int &status)
{
    double R = s_gas_constant(Q, status);
    double nu = 1/Q.rho;
    double nu_0 = s_covolume(Q, status);
    return R/(nu - nu_0);
}

double
van_der_Waals_gas::
s_covolume(const gas_data &Q, int &status)
{
    vector<double> molef;
    molef.resize(M_.size());
    convert_massf2molef(Q.massf, M_, molef);

    double nu_0 = 0.0;
    for ( size_t isp = 0; isp < Q.massf.size(); ++isp ) {
        nu_0 += molef[isp]*nu_0_[isp];
    }

    double M = calculate_molecular_weight(Q.massf, M_);

    status = SUCCESS;
    return nu_0/M;
}

double
van_der_Waals_gas::
s_a(const gas_data &Q, int &status)
{
    vector<double> molef;
    molef.resize(M_.size());
    convert_massf2molef(Q.massf, M_, molef);

    double a = 0.0;
    for ( size_t isp = 0; isp < Q.massf.size(); ++isp ) {
        a += molef[isp]*sqrt(a_[isp]);
    }
    double M = calculate_molecular_weight(Q.massf, M_);
    status = SUCCESS;
    return (a*a)/M;
}

double vdwg_pressure(double rho, double T, double R, double nu_0, double a)
{
    double nu_1 = 1/rho - nu_0;
    if (nu_1 <= 0) {
        cout << "Error in vdwg_pressure\n";
        cout << "nu = " << 1/rho << ", nu_0 = " << nu_0 << endl;;
        cout << "A van der Waals gas cannot have a specific-volume smaller than the co-
            volume.\n";
```

```
        cout << "Bailing out!\n";
        exit(BAD_INPUT_ERROR);
    }
    return R*T/nu_1 - a*rho*rho;
}

double vdwg_temperature(double rho, double p, double R, double nu_0, double a)
{
    double nu_1 = 1/rho - nu_0;
    if (nu_1 <= 0) {
        cout << "Error in vdwg_temperature\n";
        cout << "A van der Waals gas cannot have a specific-volume smaller than the co-
            volume.\n";
        cout << "Bailing out!\n";
        exit(BAD_INPUT_ERROR);
    }
    return (p + a*rho*rho)*nu_1/R;
}

double vdwg_density(double T, double p, double R, double nu_0, double a)
{
    double nu_old, nu, tol;
    nu = R*T/p + nu_0;
    tol = 1e-8;
    do {
        nu_old = nu;
        nu = R*T/(p + a/(nu_old*nu_old)) + nu_0;
    } while (fabs(nu_old - nu) > tol);

    return 1.0/nu;
}
```

Listing A.4: *van der Waals equation of state, source file.*

A.2 Real thermal behaviour

```
// \author: Brendan T. O'Flaherty
// \brief: real thermal behaviour

#ifndef REAL_THERMAL_BEHAVIOUR_HH
#define REAL_THERMAL_BEHAVIOUR_HH

#include <vector>
extern "C" {
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>
}
#include "../../nm/source/segmented-functor.hh"
#include "gas_data.hh"
#include "thermal-behaviour-model.hh"

class Real_thermal_behaviour : public Thermal_behaviour_model {
public:
    Real_thermal_behaviour(lua_State *L);
    ~Real_thermal_behaviour();

private:
    double T_COLD_;
    std::vector<double> M_;
    std::vector<double> R_;
    std::vector<Segmented_functor *> Cp_;
    std::vector<Segmented_functor *> h_;
    std::vector<Segmented_functor *> s_;

    int s_decode_conserved_energy(gas_data &Q, const std::vector<double> &rhoe);
    int s_encode_conserved_energy(const gas_data &Q, std::vector<double> &rhoe);
    int s_eval_energy(gas_data &Q, Equation_of_state *EOS_);
    int s_eval_temperature(gas_data &Q, Equation_of_state *EOS_);
    double s_dhdT_const_p(const gas_data &Q, int &status);
    double s_dedT_const_v(const gas_data &Q, Equation_of_state *EOS_, int &status);
    double s_eval_energy_isp(const gas_data &Q, Equation_of_state *EOS_, int isp);
    double s_eval_enthalpy_isp(const gas_data &Q, Equation_of_state *EOS_, int isp);
    double s_eval_entropy_isp(const gas_data &Q, Equation_of_state *EOS_, int isp);
    double zero_function(gas_data &Q, Equation_of_state *EOS_, double e_given, double T);
    double deriv_function(gas_data &Q, Equation_of_state *EOS_, double T);
    bool test_T_for_polynomial_breaks(double T);
};

#endif
```

Listing A.5: *Real thermal behaviour, header file.*

```
#include <iostream>
#include <sstream>
#include <cmath>
#include <stdlib.h>
#include "../../util/source/useful.h"
#include "../../util/source/lua_service.hh"
#include "../../nm/source/functor.hh"
#include "CEA-Cp-functor.hh"
```

```
#include "CEA-h-functor.hh"
#include "CEA-s-functor.hh"
#include "real-thermal-behaviour.hh"

using namespace std;

Real_thermal_behaviour::
Real_thermal_behaviour(lua_State *L)
    : Thermal_behaviour_model()
{
    T_COLD_ = get_positive_number(L, LUA_GLOBALSINDEX, "T_COLD");

    lua_getglobal(L, "species");

    if ( !lua_istable(L, -1) ) {
        ostringstream ost;
        ost << "Real_thermal_behaviour::Real_thermal_behaviour():\n";
        ost << "Error in the declaration of species: a table is expected.\n";
        input_error(ost);
    }

    int nsp = lua_objlen(L, -1);

    for ( int isp = 0; isp < nsp; ++isp ) {
        vector<Univariate_functor*> Cp;
        vector<Univariate_functor*> h;
        vector<Univariate_functor*> s;
        vector<double> breaks;
        double T_low, T_high;

        lua_rawgeti(L, -1, isp+1);
        const char* sp = luaL_checkstring(L, -1);
        lua_pop(L, 1);

        // Now bring specific species table to TOS
        lua_getglobal(L, sp);
        if ( !lua_istable(L, -1) ) {
            ostringstream ost;
            ost << "Real_thermal_behaviour::Real_thermal_behaviour():\n";
            ost << "Error locating information table for species: " << sp << endl;
            input_error(ost);
        }

        double M = get_positive_value(L, -1, "M");
        M_.push_back(M);
        R_.push_back(PC_R_u/M);

        lua_getfield(L, -1, "CEA_coeffs");
        if ( !lua_istable(L, -1) ) {
            ostringstream ost;
            ost << "Real_thermal_behaviour::Real_thermal_behaviour():\n";
            ost << "Error locating 'CEA_coeffs' table for species: " << sp << endl;
            input_error(ost);
        }
        for ( size_t i = 1; i <= lua_objlen(L, -1); ++i ) {
            lua_rawgeti(L, -1, i);
            T_low = get_positive_number(L, -1, "T_low");
```

```
            T_high = get_positive_number(L, -1, "T_high");

            Cp.push_back(new CEA_Cp_functor(L, R_.back()));
            h.push_back(new CEA_h_functor(L, (*Cp.back())(T_low), (*Cp.back())(T_high),
                R_.back()));
            s.push_back(new CEA_s_functor(L, (*Cp.back())(T_low), (*Cp.back())(T_high),
                R_.back()));
            breaks.push_back(T_low);
            lua_pop(L, 1);
        }
        breaks.push_back(T_high);
        lua_pop(L, 2); // pop coeffs, species

        Cp_.push_back(new Segmented_functor(Cp, breaks));
        h_.push_back(new Segmented_functor(h, breaks));
        s_.push_back(new Segmented_functor(s, breaks));

        for ( size_t i = 0; i < Cp.size(); ++i ) {
            delete Cp[i];
            delete h[i];
            delete s[i];
        }
    }
}

Real_thermal_behaviour::
~Real_thermal_behaviour()
{
    for ( size_t isp = 0; isp < Cp_.size(); ++isp ) {
        delete Cp_[isp];
        delete h_[isp];
        delete s_[isp];
    }
}

int
Real_thermal_behaviour::
s_decode_conserved_energy(gas_data &Q, const vector<double> &rhoe)
{
    return tbm_decode_conserved_energy(Q.e, rhoe, Q.rho);
}

int
Real_thermal_behaviour::
s_encode_conserved_energy(const gas_data &Q, vector<double> &rhoe)
{
    return tbm_encode_conserved_energy(rhoe, Q.e, Q.rho);
}

double
Real_thermal_behaviour::
s_dhdT_const_p(const gas_data &Q, int &status)
{
    status = SUCCESS;
    return tbm_dhdT_const_p(Cp_, Q.massf, Q.T);
}
```

```
int
Real_thermal_behaviour::
s_eval_temperature(gas_data &Q, Equation_of_state *EOS_)
{
    // given a correct value for e and rho
    // evaluate a new value for T.
    const int MAX_ATTEMPTS = 20;
    const int MAX_OSCIL = 3;
    const double TOL = 1.0e-6; // Experience shows this is a good value
    const bool use_T_guess = true;

    double e_given = Q.e[0];
    double T_0 = 0.0;
    double T_1 = 0.0;
    double T_prev = 0.0;
    double alpha = 0.0;

    // Because the CEA curve fits cut off at 200.0, the iteration
    // doesn't always work well when starting below this value.
    // So when that's the case, we use our crude starting guess.
    if ( use_T_guess && Q.T[0] > 200.0 )
        T_0 = Q.T[0];
    else
        T_0 = Q.e[0] / 717.0; // C_v for "normal" air as a crude guess

    // We use a Newton solver to iterate for temperature
    // 1. Evalute f_0 based on guess
    EOS_->eval_pressure(Q);
    double f_0 = zero_function(Q, EOS_, e_given, T_0);
    double dfdT = deriv_function(Q, EOS_, T_0);

    if( fabs( f_0 ) < TOL )
        return SUCCESS;

    T_prev = T_0;

    int oscil_count = 0;
    int attempt;
    for ( attempt = 0; attempt < MAX_ATTEMPTS; ++attempt ) {
        if ( fabs(dfdT) < 1.0e-12 ) {
            cout << "WARNING - Real_thermal_behaviour::solve_for_T()\n";
            cout << "Nearly zero derivative, dfdT= " << dfdT << endl;
            return FAILURE;
        }
        T_1 = T_0 - f_0/dfdT;
        Q.T[0] = T_1;
        EOS_->eval_pressure(Q);
        if ( T_1 < T_COLD_ ) {
            Q.T[0] = T_COLD_;
            s_eval_energy(Q, EOS_);
            return SUCCESS;
        }
        if ( fabs(T_0 - T_1) < TOL ) {
            Q.T[0] = T_1;
            Q.e[0] = e_given;
            return SUCCESS;
        }
```

```
        if ( fabs(T_1 - T_prev) < 0.01*TOL ) {
            // This is when we hit a symmetric oscillation
            if ( ++oscil_count == MAX_OSCIL ) break;
            alpha = fabs((double) rand() / (double) RAND_MAX);
            T_prev = T_0;
            T_1 = T_0 - alpha*f_0/dfdT;
            T_0 = T_1;
            f_0 = zero_function(Q, EOS_, e_given, T_0);
            dfdT = deriv_function(Q, EOS_, T_0);
        }
        else {
            T_prev = T_0;
            T_0 = T_1;
            f_0 = zero_function(Q, EOS_, e_given, T_0);
            dfdT = deriv_function(Q, EOS_, T_0);
        }
    }

    if( test_T_for_polynomial_breaks(T_1) ) {
        Q.T[0] = T_1;
        Q.e[0] = e_given;
    }

    // If we get this far, then the iterations did not converge.
    // We should print a warning and return -1.0 as temperature.
    // DFP: Customised the error message for the case where
    //      successive symmetric oscillations has been determined.
    cout << "WARNING - ThermallyRealGasMix::solve_for_T() \n";
    if ( oscil_count == MAX_OSCIL ) {
        cout << oscil_count << " symmetric oscillations were encountered by the Newton
            solver.\n";
        cout << "In addition, the temperature does not correspond to a CEA polynomial
            break.\n";
    }
    else {
        cout << "The Newton solver did not converge after " << attempt << " iterations.\n
            ";
    }
    cout << "T_1= " << T_1 << " T_0= " << T_0 << " f_0= " << f_0 << " dfdT= " << dfdT <<
        endl;
    if ( use_T_guess ) {
        cout << "T_guess - from gas_data = " << Q.T[0] << endl;
    }
    else {
        cout << "T_guess - from e[0], Cv = " <<  Q.e[0] / 717.0 << endl;;
    }
    cout << "Gas state...\n";
    print_gas_data(Q);
    cout << "Bailing out!\n";
    exit(ITERATION_ERROR);
}

double
Real_thermal_behaviour::
s_dedT_const_v(const gas_data &Q, Equation_of_state *EOS_, int &status)
{
    // Reference:
```

```
    // Cengel and Boles (2002)
    // Thermodynamics: an Engineering Approach, 3rd edition
    // 4th Ed.
    // McGraw Hill
    // Equation 11-46 on p. 617

    double nu = 1.0/Q.rho;
    double dnudT = -(nu*nu)/EOS_->dTdrho_const_p(Q, status);
    double dpdnu = -1.0/(nu*nu)*EOS_->dpdrho_const_T(Q, status);
    double Cv = 0.0;
    for ( size_t isp = 0; isp < Cp_.size(); ++isp ) {
        Cv += Q.massf[isp]*((*Cp_[isp])(Q.T[0]));
    }
    Cv += Q.T[0]*dnudT*dnudT*dpdnu;
    return Cv;
}

int
Real_thermal_behaviour::
s_eval_energy(gas_data &Q, Equation_of_state *EOS_)
{
    double e = 0.0;
    for ( size_t isp = 0; isp < Cp_.size(); ++isp ) {
        e += Q.massf[isp]*s_eval_energy_isp(Q, EOS_, isp);
    }
    Q.e[0] = e;
    return SUCCESS;
}

double
Real_thermal_behaviour::
s_eval_energy_isp(const gas_data &Q, Equation_of_state *EOS_, int isp)
{
    // Reference:
    // Cengel and Boles (2002)
    // Thermodynamics: an Engineering Approach, 3rd edition
    // 4th Ed.
    // McGraw Hill
    // Equation 11-29 on p. 614

    double h = s_eval_enthalpy_isp(Q, EOS_, isp);
    double pv = EOS_->prho_ratio(Q, isp);
    return h - pv;
}

double
Real_thermal_behaviour::
s_eval_enthalpy_isp(const gas_data &Q, Equation_of_state *EOS_, int isp)
{
    // Reference:
    // Cengel and Boles (2002)
    // Thermodynamics: an Engineering Approach, 3rd edition
    // 4th Ed.
    // McGraw Hill
    // Equation 11-35 on p. 615

    int status;
```

```
    double nu = 1.0/Q.rho;
    double dnudT = -(nu*nu)/EOS_->dTdrho_const_p(Q, status);
    double M = calculate_molecular_weight(Q.massf, M_);
    return (*h_[isp])(Q.T[0]) + (nu - Q.T[0]*dnudT)*Q.p*(M/M_[isp]);
}

double
Real_thermal_behaviour::
s_eval_entropy_isp(const gas_data &Q, Equation_of_state *EOS_, int isp)
{
    // Reference:
    // Cengel and Boles (2002)
    // Thermodynamics: an Engineering Approach, 3rd edition
    // 4th Ed.
    // McGraw Hill
    // Equation 11-40 on p. 615

    int status;
    double nu = 1.0/Q.rho;
    double dnudT = -(nu*nu)/EOS_->dTdrho_const_p(Q, status);
    double M = calculate_molecular_weight(Q.massf, M_);

    return (*s_[isp])(Q.T[0]) - dnudT*Q.p*(M/M_[isp]);
}

double
Real_thermal_behaviour::
zero_function(gas_data &Q, Equation_of_state *EOS_, double e_given, double T)
{
    // given correct energy
    // evaluate the zero function using temperature
    // f(T) = e_given - e(T) = 0
    Q.T[0] = T;
    s_eval_energy(Q, EOS_);
    return Q.e[0] - e_given;
}

double
Real_thermal_behaviour::
deriv_function(gas_data &Q, Equation_of_state *EOS_, double T)
{
    // return dedT_const_v
    Q.T[0] = T;
    int status;
    return dedT_const_v(Q, EOS_, status);
}

bool
Real_thermal_behaviour::
test_T_for_polynomial_breaks(double T)
{
    // Sometimes the iterations run into trouble because of badly formed
    // polynomials near the breaks.
    // It turns out the the CEA polynomials are not that continuous on the
    // the finer detail.

    if( T >= 999.99 && T <= 1000.01 )
```

```
        return true;
    if( T >= 5999.99 && T <= 6000.01 )
        return true;

    return false;
}
```

Listing A.6: *Real thermal behaviour, source file.*

A.3 Pressure-dependent rate coefficient

```
// Author: Rowan J. Gollan
// Date: 16-Apr-2009
// Place: NIA, Hampton, Virginia, USA
//
// This a port from Brendan O'Flaherty's implementation
// found in gas_models2.
//

#ifndef PRESSURE_DEPENDENT_RATE_HH
#define PRESSURE_DEPENDENT_RATE_HH

#include <vector>
extern "C" {
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>
}
#include "reaction-rate-coeff.hh"
#include "generalised-Arrhenius.hh"
#include "../models/gas_data.hh"

double compute_third_body_value(const gas_data &Q, std::map<int, double> efficiencies,
    std::vector<double> M);

class Pressure_dependent : public Reaction_rate_coefficient {
public:
    Pressure_dependent(lua_State *L, Gas_model &g);
    ~Pressure_dependent();

    double get_third_body_value(const gas_data &Q) { return
        compute_third_body_concentration(Q); }

private:
    int s_eval(const gas_data &Q);
    double compute_third_body_concentration(const gas_data &Q) { return
        compute_third_body_value(Q, efficiencies_, M_); }

    std::vector<double> M_;

    std::map<int, double> efficiencies_;

    Generalised_Arrhenius *k_inf_;
    Generalised_Arrhenius *k_0_;

    // Some (possible) Troe model parameters
    bool Troe_model_;
    bool T2_supplied_;
    double a_;
    double T1_;
    double T2_;
    double T3_;

};
```

```
Reaction_rate_coefficient* create_pressure_dependent_coefficient(lua_State *L, Gas_model
    &g);

#endif
```

Listing A.7: *Pressure-dependent rate coefficient, header file.*

```
// Author: Rowan J. Gollan
// Date: 16-Apr-2009
// Place: NIA, Hampton, Virginia, USA
//
// This is a port of code written by Brendan O'Flaherty
// which is found in gas_models2.
//

#include <cmath>
#include <sstream>

#include "../../util/source/useful.h"
#include "../../util/source/lua_service.hh"
#include "pressure-dependent-rate.hh"

using namespace std;

Pressure_dependent::
Pressure_dependent(lua_State *L, Gas_model &g)
    : Reaction_rate_coefficient()
{
    // Initialise k_inf
    lua_getfield(L, -1, "k_inf");
    k_inf_ = new Generalised_Arrhenius(L, g);
    lua_pop(L, 1);

    // Initialise k_0
    lua_getfield(L, -1, "k_0");
    k_0_ = new Generalised_Arrhenius(L, g);
    lua_pop(L, 1);

    // Pull out efficiencies
    read_table_as_map(L, -1, "efficiencies", efficiencies_);

    // Fill in Troe values, if available...
    Troe_model_ = false;
    lua_getfield(L, -1, "Troe");
    if ( lua_istable(L, -1) ) {
        Troe_model_ = true;
        lua_getfield(L, -1, "a"); a_ = luaL_checknumber(L, -1); lua_pop(L, 1);
        lua_getfield(L, -1, "T1"); T1_ = luaL_checknumber(L, -1); lua_pop(L, 1);
        lua_getfield(L, -1, "T3"); T3_ = luaL_checknumber(L, -1); lua_pop(L, 1);
        lua_getfield(L, -1, "T2");
        if ( !lua_isnumber(L, -1) ) {
            T2_supplied_ = false;
            T2_ = 0.0;
        }
        else {
            T2_supplied_ = true;
            T2_ = luaL_checknumber(L, -1);
```

```
        }

        lua_pop(L, 1);
    }
    lua_pop(L, 1);

    // Setup array for storage of molecular weights
    M_.resize(g.get_number_of_species());

    // Fill in molecular weights...
    for ( int isp = 0; isp < g.get_number_of_species(); ++isp ) {
        M_[isp] = g.molecular_weight(isp);
    }

}

Pressure_dependent::
~Pressure_dependent()
{
    delete k_inf_;
    delete k_0_;
}

int
Pressure_dependent::
s_eval(const gas_data &Q)
{
    // Find value of third body concentration
    double M = compute_third_body_concentration(Q);

    // Evaluate the limiting reaction rates (at high and low
    // pressure limits)
    k_inf_->eval(Q);
    k_0_->eval(Q);

    double k_inf = k_inf_->k();
    double k_0 = k_0_->k();

    double p_r = k_0*M/k_inf;
    double small = 1.0e-30;
    double log_p_r = log10(max(p_r, small));

    // Lindemann-Hinshelwood model
    double F = 1.0;

    // Troe model
    if ( Troe_model_ ) {

        double F_cent = (1.0 - a_)*exp(-Q.T[0]/T3_) + a_*exp(-Q.T[0]/T1_);

        if ( T2_supplied_ ) {
            F_cent += exp(-T2_/Q.T[0]);
        }

        double log_F_cent = log10(max(F_cent, small));
        double c = -0.4 - 0.67*log_F_cent;
        double n = 0.75 - 1.27*log_F_cent;
```

```
        double d = 0.14;

        double numer = log_p_r + c;
        double denom = n - d*numer;
        double frac = numer/denom;
        double log_F = log_F_cent / (1.0 + frac*frac);
        F = pow(10,log_F);
    }

    k_ = F*k_0*k_inf*M/(k_0*M + k_inf);
    return SUCCESS;
}

// double
// Pressure_dependent::
// compute_third_body_concentration(const gas_data &Q)
// {
//     // First compute concentration of requisite species...
//     double M = 0.0;
//     map<int, double>::const_iterator it;
//     for ( it = efficiencies_.begin(); it != efficiencies_.end(); ++it ) {
//      int isp = it->first;
//      double eff = it->second;
//      M += eff * (Q.massf[isp]*Q.rho / M_[isp]);
//     }

//     return M;
// }

double
compute_third_body_value(const gas_data &Q, map<int, double> efficiencies, vector<double>
    M)
{
    // First compute concentration of requisite species...
    double tbv = 0.0;
    map<int, double>::const_iterator it;
    for (it = efficiencies.begin(); it != efficiencies.end(); ++it) {
        int isp = it->first;
        double eff = it->second;
        tbv += eff * (Q.massf[isp]*Q.rho/M[isp]);
    }

    return tbv;
}

Reaction_rate_coefficient* create_pressure_dependent_coefficient(lua_State *L, Gas_model
    &g)
{
    return new Pressure_dependent(L, g);
}
```

Listing A.8: *Pressure-dependent rate coefficient, source file.*

Appendix B

Engine source code

B.1 Free-piston engine kernel

```
#ifndef FPE_KERNEL_HH
#define FPE_KERNEL_HH

// \author Brendan T. O'Flaherty
// \brief  Free-piston engine kernel

#include <string>
#include <vector>

#include "../../lib/util/source/useful.h"
#include "../../lib/gas/models/gas-model.hh"
#include "../../lib/gas/kinetics/reaction-update.hh"

// free-piston compressor
#define __X 0
#define __U 1
#define __E 2
#define __Q 3
#define __A 4

// free-piston engine
#define __E_L 2
#define __E_R 3
#define __Q_L 4
#define __Q_R 5
#define __F 6
#define __W 7

// exhausting
#define __M 0
#define __ME 1

struct global_data
{
    std::vector<double> u; // initial velocity
    std::vector<double> m_p; // piston mass
    std::vector<double> L_c; // cylinder length
    std::vector<double> D; // cylinder diameter
    std::vector<double> T_ig;
    double dh; // heat of combustion
    double t; // current time
```

```
    double tlast; // final time
    double dt_write; // write interval
    double dt_sys; // system step size
    double dt_therm; // thermodynamic step size
    double tol; // tolerance of dynamics
};

Gas_model* get_gas_model_ptr();
int set_gas_model(std::string file_name);

Reaction_update* get_reaction_update_ptr();
int set_reaction_update(std::string file_name);

global_data* get_global_data_ptr();
int set_global_data(std::string file_name);

struct global_data* create_global_data(std::string cfile);

#endif
```

Listing B.1: *Free-piston engine kernel, header file.*

```
#include "fpe_kernel.hh"
#include "../../lib/util/source/lua_service.hh"

using namespace std;

Gas_model* gmodel = 0;

Gas_model* get_gas_model_ptr()
{
    return gmodel;
}

int set_gas_model(std::string file_name)
{
    gmodel = create_gas_model(file_name);
    return SUCCESS;
}

Reaction_update* rupdate = 0;

Reaction_update* get_reaction_update_ptr()
{
    return rupdate;
}

int set_reaction_update(std::string file_name)
{
    rupdate = create_Reaction_update(file_name, *(get_gas_model_ptr()));
    if ( rupdate != 0 )
        return SUCCESS;
    else
        return FAILURE;
}

static global_data gd;
```

```
global_data* get_global_data_ptr()
{
    return &gd;
}

int set_global_data(std::string file_name)
{
    global_data* gdata = create_global_data(file_name);
    if (gdata != 0)
        return SUCCESS;
    else
        return FAILURE;
}

global_data*
create_global_data(string cfile)
{
    lua_State *L = initialise_lua_State();

    if (luaL_dofile(L, cfile.c_str()) != 0) {
        printf("Error in global data input file %s\n", cfile.c_str());
    }

    lua_getglobal(L, "data");
    if (!lua_istable(L, -1)) {
        printf("Error in global data declaration: a table is expected.\n");
    }

    global_data* gdata = get_global_data_ptr();

    gdata->u = get_vector(L, -1, "u");
    gdata->m_p = get_vector(L, -1, "m_p");
    gdata->L_c = get_vector(L, -1, "L_c");
    gdata->D = get_vector(L, -1, "D");
    gdata->T_ig = get_vector(L, -1, "T_ig");
    gdata->dh = get_number(L, -1, "dh");
    gdata->t = get_number(L, -1, "t");
    gdata->tlast = get_number(L, -1, "tlast");
    gdata->dt_write = get_number(L, -1, "dt_write");
    gdata->dt_sys = get_number(L, -1, "dt_sys");
    gdata->dt_therm = get_number(L, -1, "dt_therm");
    gdata->tol = get_number(L, -1, "tol");
    lua_close(L);

    return gdata;
}
```

Listing B.2: *Free-piston engine kernel, source file.*

B.2 Free-piston engine tests

```
#ifndef FPE_TEST_HH
#define FPE_TEST_HH

// \author Brendan T. O'Flaherty
// \brief  Test functions

#include <vector>
#include <valarray>
#include <string>

#include "../../lib/gas/models/gas_data.hh"

int test_discharge_coefficient();
int test_get_geometry();
int test_friction_coefficient(double b, std::vector<double> u_p);
int test_valve_opening(std::vector<double> t, double t0);

int test_ideal_sod_shock_tube(std::string fname,
                              double tlast,
                              gas_data Q1,
                              gas_data Q4,
                              int NX = 1000);
int test_sod_shock_tube(std::string fname,
                        double tlast,
                        gas_data Q1,
                        gas_data Q4,
                        int NX = 1000);

int test_exhaust_system(std::string fname,
                        std::vector<double> y0v,
                        gas_data Q_up,
                        gas_data Q,
                        gas_data Q_dn,
                        double D,
                        double V,
                        bool displace,
                        double t0,
                        double tlast,
                        double dt,
                        double tol);

int print_energy(std::string fname,
                 double t,
                 std::valarray<double> yout,
                 gas_data Q,
                 double m_p,
                 double m_g);
int print_state(std::string fname,
                double t,
                std::valarray<double> yout,
                gas_data &Q);
int print_state(std::string fname,
                double t,
                std::valarray<double> yout,
                std::valarray<double> yout_eL,
```

```
                std::valarray<double> yout_eR,
                gas_data &Q1,
                gas_data &Q2,
                double mgL,
                double mgR);

int test_otto_cycle(std::string fname,
                    std::vector<double> y0v,
                    int event, gas_data Q,
                    double x0, double u0,
                    double m_p,
                    double L_p,
                    double x_R,
                    double D,
                    double dh,
                    double t0,
                    double tlast,
                    double dt_write,
                    double dt_sys,
                    double dt_therm,
                    double tol);

std::vector<double>
test_free_piston_compressor(std::string fname,
                            std::vector<double> y0v,
                            int event,
                            gas_data Q,
                            double x0,
                            double u0,
                            double m_p,
                            double L_p,
                            double L_c,
                            double D,
                            double p_back,
                            double dh,
                            double t0,
                            double tlast,
                            double dt_write,
                            double dt_sys,
                            double dt_therm,
                            double tol);

std::vector<double>
test_free_piston_engine(std::string fname,
                        std::vector<double> y0v_fpe,
                        std::vector<double> y0v_es,
                        int event,
                        int no_cycles,
                        bool displace,
                        gas_data Q_in,
                        gas_data Q_ex,
                        gas_data Q_L,
                        gas_data Q_R,
                        double x0,
                        double u0,
                        double m_p,
                        double L_p,
```

```
                        double L_c,
                        double D,
                        double dh,
                        double t0,
                        double tlast,
                        double dt_write,
                        double dt_sys,
                        double dt_therm,
                        double tol);

#endif
```

Listing B.3: *Test functions, header file.*

```
#include <cstdio>

#include "../../lib/util/source/useful.h"
#include "../../lib/gas/models/gas-model.hh"
#include "../../lib/nm/source/ode_solver.hh"
#include "../../lib/nm/source/linear_interpolation.hh"

#include "fpe_kernel.hh"
#include "sod.hh"
#include "fpe_system.hh"
#include "fpe_models.hh"
#include "fpe_control.hh"
#include "fpe_tests.hh"

using namespace std;

FILE *fout = 0; // output file pointer

int
test_discharge_coefficient()
{
    double cdi, cde;

    vector<double> Lv;
    for (size_t i = 0; i <= 60; ++i) {
        Lv.push_back((double)i*0.005);
    }
    double Dv = 1.0;

    string intake("intake");
    string exhaust("exhaust");

    for (size_t i = 0; i < Lv.size(); ++i) {
        discharge_coefficient(cdi, Lv[i], Dv, intake);
        discharge_coefficient(cde, Lv[i], Dv, exhaust);
        printf("%5.4f %5.4f %5.4f\n", Lv[i]/Dv, cdi, cde);
    }
    printf("# done.\n");

    return SUCCESS;
}

int
```

```
test_get_geometry()
{
    double x = 0.0;
    double D = 0.2;
    double L_c = 0.8;
    double x_L = -1.0;
    double x_R = 1.0;

    vector<double> geom(11);
    get_geometry(geom, x, D, L_c, x_L, x_R);

    printf("A    = %g\n", geom[0]);
    printf("xp_L = %g\n", geom[1]);
    printf("xp_R = %g\n", geom[2]);
    printf("L_i  = %g\n", geom[3]);
    printf("L_p  = %g\n", geom[4]);
    printf("L_L  = %g\n", geom[5]);
    printf("L_R  = %g\n", geom[6]);
    printf("V_L  = %g\n", geom[7]);
    printf("V_R  = %g\n", geom[8]);
    printf("S_L  = %g\n", geom[9]);
    printf("S_R  = %g\n", geom[10]);
    printf("# done.\n");

    return SUCCESS;
}

int
test_friction_coefficient(double b,
                          vector<double> u_p)
{
    double f;
    double eta = 0.222; // kinematic viscosity of SAE30 at 300K, cSt (approx)
    double nu;

    printf("#10^4*(eta*u/(p*b))^0.5, f_h, f\n");
    for (size_t i = 0; i < u_p.size(); ++i) {
        mixed_friction_coefficient(f, u_p[i], eta, P_ATM, b);
        nu = pow((eta*fabs(u_p[i])/(P_ATM*b)),0.5);
        printf("%5.4e %5.4e\n", 1e4*nu, f);
    }
    printf("# done.\n");

    return SUCCESS;
}

int
test_valve_opening(vector<double> t, double t0)
{
    // tests both intake and exhaust valves
    double D = sqrt(4.0/PI);
    double t1 = t0 + 0.2;
    double A_max = 0.25*PI*D*D;

    Valve v_in("intake", 10.0, D);
    Valve v_out("exhaust", 10.0, D);
```

```
    printf("# t, A\n");
    printf("%4.3f %4.3f %4.3f %4.3f %4.3f %4.3f\n",
           t[0], v_in.get_area(), v_in.eval_discharge_coefficient(),
           v_out.get_area(), v_out.eval_discharge_coefficient(), A_max);
    double dt;
    for (size_t i = 1; i < t.size(); ++i) {
        // a simple criterion for the start of valve open and close
        dt = t[i] - t[i-1];
        if (t[i] > t0 && t[i] < t1) {
            v_in.open(dt);
            v_out.open(dt);
        }
        if (t[i] > t1) {
            v_in.close(dt);
            v_out.close(dt);
        }
        printf("%4.3f %4.3f %4.3f %4.3f %4.3f %4.3f\n",
               t[i], v_in.get_area(), v_in.eval_discharge_coefficient(),
               v_out.get_area(), v_out.eval_discharge_coefficient(), A_max);
    }
    printf("# done.\n");

    return SUCCESS;
}

int
test_ideal_sod_shock_tube(string fname,
                          double tlast,
                          gas_data Q1,
                          gas_data Q4,
                          int NX)
{
    // Enter in at what time you want the profile.
    double t = tlast;
    // Enter in diaphragm location
    double x0 = 0.5;
    // How long do you want the shock tube to be?
    double L = 1.0;

    double u4 = 0.0;

    Gas_model* g = get_gas_model_ptr();

    gas_data Q2, Q3;
    g->initialise_gas_data(Q2);
    g->initialise_gas_data(Q3);
    copy_gas_data(Q1, Q2);
    copy_gas_data(Q4, Q3);

    if (x0 - Q4.a*t < 0.0) {
        printf("Expansion wave reaches end of tube.\n");
        printf("Minimum diaphragm position must be: %e\n", (Q4.a*t));
        return FAILURE;
    }
    if (L - (x0 + Q1.a*t) < 0.0) {
        printf("Shock wave reaches end of tube.\n");
        printf("Minimum length must be: %e\n", (x0 + Q1.a*t));
```

```
    return FAILURE;
}

double tol = 1e-6;
p2p1_fun *zfun = new p2p1_fun(Q1, Q2, Q3, Q4);
Muller p2p1_solver(zfun, tol);

// these bounds should always be sufficient to guarantee a root
double p2p1 = p2p1_solver(0.0, Q4.p/Q1.p);
double g1 = g->gamma(Q1);
double r2r1 = (1.0 + ((g1 + 1.0)/(g1 - 1.0))*p2p1)/((g1 + 1.0)/(g1 - 1.0) + p2p1);
double W = Q1.a*sqrt(((g1 + 1.0)/(2*g1))*(p2p1 - 1.0) + 1.0);
double u2 = get_velocity_behind_shock(1.0/r2r1, W);

printf("p2/p1 = %12.11e\n", p2p1);
printf("r2/r1 = %12.11e, W = %12.11e, up = %12.11e\n", r2r1, W, u2);

Q2.rho = Q1.rho*r2r1;
Q2.p = Q1.p*p2p1;
g->eval_thermo_state_rhop(Q2);

// start at Q3(=Q4) and step across wave
double u3 = 0.0;
vector< vector<double> > sol;
double soln[] = {Q3.rho, Q3.p, Q3.T[0], u3};
vector<double> temp(soln, soln+4);
sol.push_back(temp);

double du = (u2 - u4)/NX;
for (int i = 0; i < NX; ++i) {
    step_across_expansion_du(Q3, u3, du);
    //printf("step %i, p = %g: ", i, Q3.p);
    double soln[] = {Q3.rho, Q3.p, Q3.T[0], u3};
    vector<double> temp(soln, soln+4);
    sol.push_back(temp);
}

// because we stepped across the shock using u, there is an error in
// the pressure boundary condition, re-evaluate...
Q2.p = Q3.p;
g->eval_thermo_state_rhop(Q2);

double xs = x0 + W*t; // Location of shock
double xc = x0 + u2*t; // Location of contact discontinuity
double dx = ((t*(u2 - Q3.a)) - (-Q4.a*t))/NX;

// Now we can print the solution.
fout = fopen(fname.c_str(), "w");
fprintf(fout, "# x, rho, p, T, u\n");

// We go left to right
// Region 4 to expansion fan
fprintf(fout ,"# region 4 start to expansion fan\n");
fprintf(fout, "%e %e %e %e %e\n", 0.0, Q4.rho, Q4.p, Q4.T[0], u4);
fprintf(fout, "%e %e %e %e %e\n", x0 - Q4.a*t, Q4.rho, Q4.p, Q4.T[0], u4);

vector<double> space(2, 0.0);
```

```
    vector< vector<double> > rx, Tx, px, ux;
    for (int i = 0; i < NX; ++i) {
        rx.push_back(space);
        Tx.push_back(space);
        px.push_back(space);
        ux.push_back(space);
    }
    rx[0][0] = x0 - Q4.a*t;
    px[0][0] = rx[0][0];
    Tx[0][0] = rx[0][0];
    ux[0][0] = rx[0][0];

    for (int i = 1; i < NX; ++i) {
        rx[i][0] = rx[i-1][0] + dx;
        px[i][0] = rx[i][0];
        Tx[i][0] = rx[i][0];
        ux[i][0] = rx[i][0];
    }
    for (int i = 0; i < NX; ++i) {
        rx[i][1] = sol[i][0];
        px[i][1] = sol[i][1];
        Tx[i][1] = sol[i][2];
        ux[i][1] = sol[i][3];
    }
    // Expansion fan
    fprintf(fout, "# expansion fan\n");
    for (int i = 0; i < NX; ++i) {
        fprintf(fout, "%e %e %e %e %e\n", rx[i][0], rx[i][1], px[i][1], Tx[i][1], ux[i
            ][1]);
    }
    // Fan to contact surface
    fprintf(fout, "# fan to contact surface\n");
    fprintf(fout, "%e %e %e %e %e\n", t*(u2 - Q3.a)+x0, Q3.rho, Q3.p, Q3.T[0], u2);
    fprintf(fout, "%e %e %e %e %e\n", xc, Q3.rho, Q3.p, Q3.T[0], u2);
     // Contact surface to shock
    fprintf(fout, "# contact surface to shock\n");
    fprintf(fout, "%e %e %e %e %e\n", xc, Q2.rho, Q2.p, Q2.T[0], u2);
    fprintf(fout, "%e %e %e %e %e\n", xs, Q2.rho, Q2.p, Q2.T[0], u2);
     // Shock to region 1's end
    fprintf(fout, "# shock to region 1 end\n");
    fprintf(fout, "%e %e %e %e %e\n", xs, Q1.rho, Q1.p, Q1.T[0], 0.0);
    fprintf(fout, "%e %e %e %e %e\n", L, Q1.rho, Q1.p, Q1.T[0], 0.0);
    fprintf(fout, "# done.\n");
    fclose(fout);

    delete zfun;
    return SUCCESS;
}

int
test_sod_shock_tube(string fname,
                    double tlast,
                    gas_data Q1,
                    gas_data Q4,
                    int NX)
{
    // Enter the tolerance
```

```
double tol = 1e-6;
// Enter at what time you want the profile.
double t = tlast;
// Enter diaphragm location
double x0 = 0.5;
// How long do you want the shock tube to be?
double L = 1.0;

double u4 = 0.0;
Gas_model* g = get_gas_model_ptr();

gas_data Q2, Q3;
g->initialise_gas_data(Q2);
g->initialise_gas_data(Q3);
copy_gas_data(Q1, Q2);
copy_gas_data(Q4, Q3);

if (x0 - Q4.a*t < 0.0) {
    printf("Expansion wave reaches end of tube.\n");
    printf("Minimum diaphragm position must be: %e\n", (Q4.a*t));
    return FAILURE;
}
if (L - (x0 + Q1.a*t) < 0.0) {
    printf("Shock wave reaches end of tube.\n");
    printf("Minimum length must be: %e\n", (x0 + Q1.a*t));
    return FAILURE;
}

// first solve the ideal density ratio, then account for real gas effects
p2p1_fun *zfun = new p2p1_fun(Q1, Q2, Q3, Q4);
Muller p2p1_solver(zfun, tol);
double p2p1 = p2p1_solver(0.0, Q4.p/Q1.p);
double g1 = g->gamma(Q1);
double r2r1 = (1.0 + ((g1 + 1.0)/(g1 - 1.0))*p2p1)/((g1 + 1.0)/(g1 - 1.0) + p2p1);
Q2.p = p2p1*Q1.p;

r1r2_fun *rfun = new r1r2_fun(Q1, Q2, Q3, Q4, NX);
Muller r1r2_solver(rfun, tol);

// experience shows these bounds should work
double r1r2 = r1r2_solver(0.3/r2r1, 1.1/r2r1);

// At this point, we have solved the density ratio
// across the shock and thereby almost evaluated
// the entire solution. However, because this solution
// has been wrapped up in the zero finding method,
// we must do it again here to get the data.
//
// Choose portability over efficiency (Rule 4, Unix Philosophy).

double W;
get_state_behind_shock(Q2, Q1, r1r2);
get_shock_speed(W, r1r2, Q1, Q2);
double u2 = get_velocity_behind_shock(r1r2, W);

printf("p2/p1 = %12.11e\n", Q2.p/Q1.p);
printf("r2/r1 = %12.11e, W = %12.11e, up = %12.11e\n", 1/r1r2, W, u2);
```

```
// start at Q3(=Q4) and step across wave
double u3 = 0.0;
vector< vector<double> > sol;
double soln[] = {Q3.rho, Q3.p, Q3.T[0], u3};
vector<double> temp(soln, soln+4);
sol.push_back(temp);

double du = (u2 - u4)/NX;
for (int i = 0; i < NX; ++i) {
    step_across_expansion_du(Q3, u3, du);
    double soln[] = {Q3.rho, Q3.p, Q3.T[0], u3};
    vector<double> temp(soln, soln+4);
    sol.push_back(temp);
}

// because we stepped across the shock using u, there is an error in
// the pressure boundary condition, re-evaluate...
Q2.p = Q3.p;
g->eval_thermo_state_rhop(Q2);

double xs = x0 + W*t; // Location of shock
double xc = x0 + u2*t; // Location of contact discontinuity
double dx = ((t*(u2 - Q3.a)) - (-Q4.a*t))/NX;

// Now we can print the solution.

fout = fopen(fname.c_str(), "w");
fprintf(fout, "# x, rho, p, T, u\n");

// We go left to right
// Region 4 to expansion fan
fprintf(fout ,"# region 4 start to expansion fan\n");
fprintf(fout, "%e %e %e %e %e\n", 0.0, Q4.rho, Q4.p, Q4.T[0], u4);
fprintf(fout, "%e %e %e %e %e\n", x0 - Q4.a*t, Q4.rho, Q4.p, Q4.T[0], u4);

vector<double> space(2, 0.0);
vector< vector<double> > rx, Tx, px, ux;
for (int i = 0; i < NX; ++i) {
    rx.push_back(space);
    Tx.push_back(space);
    px.push_back(space);
    ux.push_back(space);
}
rx[0][0] = x0 - Q4.a*t;
px[0][0] = rx[0][0];
Tx[0][0] = rx[0][0];
ux[0][0] = rx[0][0];

// printf("%i %i %i %i\n", rx.size(), px.size(), Tx.size(), ux.size());

for (int i = 1; i < NX; ++i) {
    rx[i][0] = rx[i-1][0] + dx;
    px[i][0] = rx[i][0];
    Tx[i][0] = rx[i][0];
    ux[i][0] = rx[i][0];
}
```

```
    for (int i = 0; i < NX; ++i) {
        rx[i][1] = sol[i][0];
        px[i][1] = sol[i][1];
        Tx[i][1] = sol[i][2];
        ux[i][1] = sol[i][3];
    }
    // Expansion fan
    fprintf(fout, "# expansion fan\n");
    for (int i = 0; i < NX; ++i) {
        fprintf(fout, "%e %e %e %e %e\n", rx[i][0], rx[i][1], px[i][1], Tx[i][1], ux[i
            ][1]);
    }
    // Fan to contact surface
    fprintf(fout, "# fan to contact surface\n");
    fprintf(fout, "%e %e %e %e %e\n", t*(u2 - Q3.a)+x0, Q3.rho, Q3.p, Q3.T[0], u2);
    fprintf(fout, "%e %e %e %e %e\n", xc, Q3.rho, Q3.p, Q3.T[0], u2);
     // Contact surface to shock
    fprintf(fout, "# contact surface to shock\n");
    fprintf(fout, "%e %e %e %e %e\n", xc, Q2.rho, Q2.p, Q2.T[0], u2);
    fprintf(fout, "%e %e %e %e %e\n", xs, Q2.rho, Q2.p, Q2.T[0], u2);
     // Shock to region 1's end
    fprintf(fout, "# shock to region 1 end\n");
    fprintf(fout, "%e %e %e %e %e\n", xs, Q1.rho, Q1.p, Q1.T[0], 0.0);
    fprintf(fout, "%e %e %e %e %e\n", L, Q1.rho, Q1.p, Q1.T[0], 0.0);
    fprintf(fout, "# done.\n");
    fclose(fout);

    delete zfun;
    delete rfun;
    return SUCCESS;
}

int
test_exhaust_system(string fname,
                    vector<double> y0v,
                    gas_data Q_i,
                    gas_data Q,
                    gas_data Q_e,
                    double D,
                    double V,
                    bool displace,
                    double t0,
                    double tlast,
                    double dt,
                    double tol)
{
    int ndim = (int)y0v.size();
    valarray<double> y0(ndim);
    for (int i = 0; i < ndim; ++i) y0[i] = y0v[i];

    Gas_model* g = get_gas_model_ptr();
    int nsp = g->get_number_of_species();

    // make a local copy of the gas data
    gas_data Q_0, Q_1, Q_2;
    g->initialise_gas_data(Q_0);
    g->initialise_gas_data(Q_1);
```

```
    g->initialise_gas_data(Q_2);

    copy_gas_data(Q_i, Q_0);
    copy_gas_data(Q, Q_1);
    copy_gas_data(Q_e, Q_2);
    bool test_flag = false;

    Exhaust_system_ode *es_ode = new Exhaust_system_ode(&Q_1, &Q_0, &Q_2, D, tol, ndim,
        test_flag, displace);
    Exhaust_system es(es_ode, y0, dt, V, Q_1);
    fout = fopen(fname.c_str(), "w");
    fprintf(fout, "# t, m, m*e, ");
    for (int i = 0; i < nsp; ++i) {
        fprintf(fout, "m*Y_%s, ", g->species_name(i).c_str());
    }
    fprintf(fout, "mout, T, p, rho, ");
    for (int i = 0; i < nsp; ++i) {
        fprintf(fout, "X_%s, ", g->species_name(i).c_str());
    }
    fprintf(fout, "\n");
    fclose(fout);

    valarray<double> yout, yin;
    yout.resize(ndim);
    yin.resize(ndim);
    //printf("m0 = %g\n", V*Q_1.rho);
    es.get_yout(yout);
    print_state(fname, t0, yout, Q_1);
    //print_energy(fname, t0, es.get_yout(), Q, m_p, m_g);

    while (t0 < tlast) {
        es.advance_fluid_dynamics(dt);

        es.get_yin(yin);
        es.get_yout(yout);

        // increment time
        t0 += dt;

        // finalise step
        es.finalise_step(Q_1);

        print_state(fname, t0, yout, Q_1);
        //print_energy(fname, t0, fpc.get_yout(), Q, m_p, m_g);
    }
    fout = fopen(fname.c_str(), "a");
    fprintf(fout, "# done.\n");
    fclose(fout);

    return SUCCESS;
}

int
print_energy(string fname, double t, valarray<double> yout, gas_data Q, double m_p,
    double m_g)
{
    //global_data* gd = get_global_data_ptr();
```

```
    double ke = 0.5*m_p*yout[1]*yout[1];
    //double mg = Q.rho*0.25*PI*gd->D*gd->D*(gd->x_R - yout[0]);
    double e = m_g*yout[2];
    fout = fopen(fname.c_str(), "a");
    fprintf(fout, "%12.11e, %12.11e, %12.11e\n", t, ke, e);
    fclose(fout);

    return SUCCESS;
}

int
print_energy(string fname, double t, valarray<double> yout, gas_data QL, gas_data QR,
    double m_p, double m_gL, double m_gR)
{
    double ke = 0.5*m_p*yout[1]*yout[1];
    double eL = m_gL*yout[2];
    double eR = m_gR*yout[3];
    fout = fopen(fname.c_str(), "a");
    fprintf(fout, "%12.11e, %12.11e, %12.11e, %12.11e\n", t, ke, eL, eR);
    fclose(fout);

    return SUCCESS;
}

int
print_state(string fname, double t, valarray<double> yout, gas_data &Q)
{
    Gas_model* g = get_gas_model_ptr();
    int nsp = g->get_number_of_species();

    fout = fopen(fname.c_str(), "a");
    fprintf(fout, "%12.11e, ", t);
    for(size_t i = 0; i < yout.size(); ++i) {
        fprintf(fout, "%12.11e, ", yout[i]);
    }
    fprintf(fout, "%12.11e, %12.11e, %12.11e, ", Q.T[0], Q.p, Q.rho);
    vector<double> molef(nsp, 0.0);
    convert_massf2molef(Q.massf, g->M(), molef);
    for (int isp = 0; isp < nsp; ++isp) {
        fprintf(fout, "%12.11e, ", molef[isp]);
    }
    fprintf(fout, "\n");
    fclose(fout);

    return SUCCESS;
}

int
print_state(string fname, double t, valarray<double> yout, valarray<double> yout_L,
    valarray<double> yout_R, gas_data &Q1, gas_data &Q2, double mgL, double mgR)
{
    Gas_model* g = get_gas_model_ptr();
    int nsp = g->get_number_of_species();

    fout = fopen(fname.c_str(), "a");
    fprintf(fout, "%12.11e, ", t);
```

```
    for(size_t i = 0; i < yout.size(); ++i) {
        fprintf(fout, "%12.11e, ", yout[i]);
    }
    for(size_t i = 0; i < 2; ++i) { // only mass and energy
        fprintf(fout, "%12.11e, ", yout_L[i]);
    }
    for(size_t i = 0; i < 2; ++i) { // only mass and energy
        fprintf(fout, "%12.11e, ", yout_R[i]);
    }
    // fprintf(fout, "%12.11e, %12.11e, %12.11e, %12.11e, %12.11e, ", t, yout[__X], yout[
        __U], yout[__E_L], yout[__E_R]);
    fprintf(fout, "%12.11e, %12.11e, %12.11e, %12.11e, %12.11e, %12.11e, %12.11e, %12.11e
        , ", Q1.T[0], Q1.p, Q1.rho, mgL, Q2.T[0], Q2.p, Q2.rho, mgR);

    vector<double> molef(nsp, 0.0);
    convert_massf2molef(Q1.massf, g->M(), molef);
    for (int isp = 0; isp < nsp; ++isp) fprintf(fout, "%12.11e, ", molef[isp]);
    convert_massf2molef(Q2.massf, g->M(), molef);
    for (int isp = 0; isp < nsp; ++isp) fprintf(fout, "%12.11e, ", molef[isp]); fprintf(
        fout, "\n");
    fclose(fout);

    return SUCCESS;
}

int
test_otto_cycle(string fname,
                vector<double> y0v,
                int event,
                gas_data Q0,
                double x0,
                double u0,
                double m_p,
                double L_p,
                double L_c,
                double D,
                double dh,
                double t0,
                double tlast,
                double dt_write,
                double dt_sys,
                double dt_therm,
                double tol)
{
    int ndim = (int)y0v.size();
    valarray<double> y0(ndim);
    for (int i = 0; i < ndim; ++i) y0[i] = y0v[i];

    Gas_model* g = get_gas_model_ptr();
    int nsp = g->get_number_of_species();

    // make a local copy of the gas data
    gas_data Q;
    g->initialise_gas_data(Q);
    copy_gas_data(Q0, Q);

    // determine the number of counts between writing the solution
```

```
int iwrite = 0;
int nwrite = (int)(dt_write/dt_sys);

// uncomment free-piston or crankshaft driven engine as required
Free_piston_compressor_ode *fpc_ode = new Free_piston_compressor_ode(&Q, m_p, x0, u0,
    L_p, L_c, D, tol, ndim, true);
Free_piston_compressor fpc(fpc_ode, y0, dt_sys, dt_therm, Q.rho);
//Crankshaft_compressor_ode *cc_ode = new Crankshaft_compressor_ode(&Q, m_p, x0, u0,
    L_p, L_c, D, tol, ndim, true);
//Crankshaft_compressor fpc(cc_ode, y0, dt_sys, dt_therm, Q.rho);

fout = fopen(fname.c_str(), "w");
fprintf(fout, "# t, x, u, e, q, T, p, rho, ");
for (int i = 0; i < nsp; ++i) {
    fprintf(fout, "%s, ", g->species_name(i).c_str());
}
fprintf(fout, "\n");
//fprintf(fout, "# t, KE, E\n");
fclose(fout);

valarray<double> yout, yin;
yout.resize(ndim);
yin.resize(ndim);

print_state(fname, t0, fpc.get_yout(), Q);
//double m_g = fpc_ode->get_m0();
//print_energy(fname, t0, fpc.get_yout(), Q, m_p, m_g);

double L0 = L_c;
double L1;

bool heat_added = false;
double tevent = 0.0;
//printf("%g %g %g\n", m_p, m_g, fpc_ode->get_m0());
while (t0 < tlast) {
    fpc.advance_dynamics(Q, dt_sys);

    yin = fpc.get_yin();
    yout = fpc.get_yout();

    if (yout[event] < 0 && !heat_added) {
        // if the piston reverses direction
        // and we are running an otto cycle
        // take a controlled step, print, then continue.
        double tf[] = {t0+dt_sys, t0};
        double yf[] = {yout[event], yin[event]};
        vector<double> tfa(tf, tf+2);
        vector<double> yfa(yf, yf+2);
        linear_eval(0.0, tevent, yfa, tfa);

        double dtf = tevent - t0;
        fout = fopen(fname.c_str(), "a");
        fprintf(fout, "# taking a final step from %g to %g\n", t0, tevent);
        fclose(fout);

        fpc.advance_dynamics(Q, dtf);
```

```
            // increment time
            t0 = tevent;

            // prepare for continuation
            fpc.finalise_step(Q);

            print_state(fname, t0, fpc.get_yout(), Q);
            //print_energy(fname, t0, fpc.get_yout(), Q, m_p, m_g);

            // add heat
            fpc.instantaneous_heat_addition(dh);
            fpc.finalise_step(Q);

            print_state(fname, t0, fpc.get_yout(), Q);
            //print_energy(fname, t0, fpc.get_yout(), Q, m_p, m_g);

            heat_added = true;

            L1 = L_c - fpc.get_yout()[0];
            printf("compression ratio = %g\n", L0/L1);
            continue;
        }
        // if (Q.p < P_ATM) break;
        if (yout[0] < 0.0) break;
        //if (yout[4] > 3*PI) break;

        // increment time
        t0 += dt_sys;

        // prepare for next step
        fpc.finalise_step(Q);

        // increment iwrite
        ++iwrite;
        if (iwrite >= nwrite) {
            print_state(fname, t0, fpc.get_yout(), Q);
            //print_energy(fname, t0, fpc.get_yout(), Q, m_p, m_g);
            iwrite = 0;
        }
    }
    fout = fopen(fname.c_str(), "a");
    fprintf(fout, "# done.\n");
    fclose(fout);

    return SUCCESS;
}

vector<double>
test_free_piston_compressor(string fname,
                            vector<double> y0v,
                            int event,
                            gas_data Q0,
                            double x0,
                            double u0,
                            double m_p,
                            double L_p,
                            double L_c,
```

```
                              double D,
                              double p_back,
                              double dh,
                              double t0,
                              double tlast,
                              double dt_write,
                              double dt_sys,
                              double dt_therm,
                              double tol)
{
    int ndim = (int)y0v.size();
    valarray<double> y0(ndim);
    for (int i = 0; i < ndim; ++i) y0[i] = y0v[i];

    Gas_model* g = get_gas_model_ptr();
    int nsp = g->get_number_of_species();

    // make a local copy of the gas data
    gas_data Q;
    g->initialise_gas_data(Q);
    copy_gas_data(Q0, Q);

    // determine the number of counts between writing the solution
    int iwrite = 0;
    int nwrite = (int)(dt_write/dt_sys);

    Free_piston_compressor_ode *fpc_ode = new Free_piston_compressor_ode(&Q, m_p, x0, u0,
        L_p, L_c, D, tol, ndim, true, p_back);
    Free_piston_compressor fpc(fpc_ode, y0, dt_sys, dt_therm, Q.rho);

    fout = fopen(fname.c_str(), "w");
    fprintf(fout, "# t, x, u, e, q, T, p, rho, ");
    for (int i = 0; i < nsp; ++i) {
        fprintf(fout, "%s, ", g->species_name(i).c_str());
    }
    fprintf(fout, "\n");
    //fprintf(fout, "# t, KE, E\n");
    fclose(fout);

    valarray<double> yout, yin;
    yout.resize(ndim);
    yin.resize(ndim);
    print_state(fname, t0, fpc.get_yout(), Q);
    //double m_g = fpc_ode->get_m0();
    //print_energy(fname, t0, fpc.get_yout(), Q, m_p, m_g);

    double tevent = 0.0;
    while (t0 < tlast) {
        fpc.advance_chemistry(Q, dt_sys);
        fpc.advance_dynamics(Q, dt_sys);

        yin = fpc.get_yin();
        yout = fpc.get_yout();

        if (yout[event] < 0) {
            // if the piston position goes below zero
            // take a controlled step, print, then exit.
```

```
            double tf[] = {t0+dt_sys, t0};
            double yf[] = {yout[event], yin[event]};
            vector<double> tfa(tf, tf+2);
            vector<double> yfa(yf, yf+2);
            linear_eval(0.0, tevent, yfa, tfa);

            double dtf = tevent - t0;
            fout = fopen(fname.c_str(), "a");
            fprintf(fout, "# taking a final step from %g to %g\n", t0, tevent);
            fclose(fout);

            fpc.advance_chemistry(Q, dtf);
            fpc.advance_dynamics(Q, dtf);

            // increment time
            t0 = tevent;
            tlast = tevent;

            fpc.finalise_step(Q);

            print_state(fname, t0, fpc.get_yout(), Q);
            //print_energy(fname, t0, fpc.get_yout(), Q, m_p, m_g);

            // break
            break;
        }

        // increment time
        t0 += dt_sys;

        // prepare for next step
        fpc.finalise_step(Q);

        // increment iwrite
        ++iwrite;
        if (iwrite >= nwrite) {
            print_state(fname, t0, fpc.get_yout(), Q);
            //print_energy(fname, t0, fpc.get_yout(), Q, m_p, m_g);
            iwrite = 0;
        }
    }
    fout = fopen(fname.c_str(), "a");
    fprintf(fout, "# done.\n");
    fclose(fout);

    double temp[] = {fpc.get_yout()[0], fpc.get_yout()[1], fpc.get_yout()[2], Q.T[0]};
    vector<double> rvalue(temp, temp+4);

    return rvalue;
}

vector<double>
test_free_piston_engine(string fname,
                        vector<double> y0v_fpe,
                        vector<double> y0v_es,
                        int event,
                        int no_cycles,
```

```
                        bool displace,
                        gas_data Qin,
                        gas_data Qex,
                        gas_data QL,
                        gas_data QR,
                        double x0,
                        double u0,
                        double m_p,
                        double L_p,
                        double L_c,
                        double D,
                        double dh,
                        double t0,
                        double tlast,
                        double dt_write,
                        double dt_sys,
                        double dt_therm,
                        double tol)
{
    int ndim_fpe = (int)y0v_fpe.size();
    valarray<double> y0_fpe(0.0, ndim_fpe);
    for (int i = 0; i < ndim_fpe; ++i) y0_fpe[i] = y0v_fpe[i];

    double gap = 0.5*L_c; // should be as big as ever needed
    double port = D/2;
    double x_L = -L_p - L_c - port - gap;
    double x_R = L_c;
    int ndim_es = (int)y0v_es.size();
    valarray<double> y0_es(0.0, ndim_es);
    for (int i = 0; i < ndim_es; ++i) {
        y0_es[i] = y0v_es[i];
    }

    valarray<double> yout_L(0.0, ndim_es);
    valarray<double> yout_R(0.0, ndim_es);

    Gas_model* g = get_gas_model_ptr();
    int nsp = g->get_number_of_species();

    // make a local copy of the gas data
    gas_data Q_in, Q_L, Q_R, Q_exL, Q_exR;
    g->initialise_gas_data(Q_in);
    g->initialise_gas_data(Q_L);
    g->initialise_gas_data(Q_R);
    g->initialise_gas_data(Q_exL);
    g->initialise_gas_data(Q_exR);

    copy_gas_data(Qin, Q_in); // intake air
    copy_gas_data(QL, Q_L);  // left cylinder
    copy_gas_data(QR, Q_R);  // right cylinder
    copy_gas_data(Qex, Q_exL); // exhaust air
    copy_gas_data(Qex, Q_exR); // exhaust air

    double A = 0.25*PI*D*D;
    double L_L = (x0 - L_p/2.0) - x_L;
    double L_R = x_R - (x0 + L_p/2.0);
    double V_L = L_L*A;
```

```
double V_R = L_R*A;
double m_gL = V_L*Q_L.rho;
double m_gR = V_R*Q_R.rho;

// determine the number of counts between writing the solution
int iwrite = 0;
int nwrite = (int)(dt_write/dt_sys);

Free_piston_engine_ode *fpe_ode = new Free_piston_engine_ode(&Q_L, &Q_R, m_p, x0, u0,
     L_p, L_c, x_L, x_R, D, m_gL, m_gR, tol, ndim_fpe, true);
Exhaust_system_ode *es_ode_L = new Exhaust_system_ode(&Q_L, &Q_in, &Q_exL, D, tol,
    ndim_es, displace);
Exhaust_system_ode *es_ode_R = new Exhaust_system_ode(&Q_R, &Q_in, &Q_exR, D, tol,
    ndim_es, displace);
Free_piston_engine fpe(fpe_ode, es_ode_L, es_ode_R, y0_fpe, y0_es, dt_sys, dt_therm,
    m_gL, m_gR);

fout = fopen(fname.c_str(), "w");
fprintf(fout, "# t, x, u, e_L, e_R, Q_L, Q_R, w_f, dm_L, dme_L, dm_R, dme_R, T_L, p_L
    , rho_L, m_gL, T_R, p_R, rho_R, m_gR, ");
for (int i = 0; i < nsp; ++i) {
    fprintf(fout, "%s_L, ", g->species_name(i).c_str());
}
for (int i = 0; i < nsp; ++i) {
    fprintf(fout, "%s_R, ", g->species_name(i).c_str());
}
fprintf(fout, "\n");
fclose(fout);

valarray<double> yout_fpe, yin_fpe;
yout_fpe.resize(ndim_fpe);
yin_fpe.resize(ndim_fpe);

vector<double> Tig, rvalue;

fpe.get_yout(yout_fpe);
fpe.get_yout_L(yout_L);
fpe.get_yout_R(yout_R);

print_state(fname, t0, yout_fpe, yout_L, yout_R, Q_L, Q_R, m_gL, m_gR);
//print_energy(fname, t0, yout_fpe, Q_L, Q_R, m_p, m_gL, m_gR);
global_data* gd = get_global_data_ptr();
double tevent = 0.0;
double dt_sys0 = dt_sys;

// events
//int event0 = 0; // x = 0
int event1 = 1; // u = 0

if (fabs(x0) < 1e-6) {
    printf("expecting an initial position of zero.\n");
    exit(1);
}
if (u0 < 0.0) {
    printf("expecting a positive intial velocity.\n");
    exit(1);
}
```

```
int x_flag = -1;
int nc = 0; // number of cycles
double x_p, x_s;
double u_p;

while (gd->t < tlast) {
    // advance chemistry
    if (es_ode_L->get_A_e() == 0) { fpe.advance_chemistry_left_cylinder(Q_L, dt_sys);
        }
    if (es_ode_R->get_A_e() == 0) { fpe.advance_chemistry_right_cylinder(Q_R, dt_sys)
        ; }
    fpe.advance_dynamics(Q_L, Q_R, dt_sys);

    fpe.get_yin(yin_fpe);
    fpe.get_yout(yout_fpe);
    fpe.get_yout_L(yout_L);
    fpe.get_yout_R(yout_R);

    x_p = yout_fpe[__X];
    u_p = yout_fpe[__U];

    if (yin_fpe[event1]*yout_fpe[event1] < 0.0) {
        // piston just changed direction, do a controlled step and continue
        double tf[] = {t0+dt_sys, t0};
        double yf[] = {yout_fpe[event], yin_fpe[event]};
        vector<double> tfa(tf, tf+2);
        vector<double> yfa(yf, yf+2);
        linear_eval(x0, tevent, yfa, tfa);

        dt_sys = tevent - t0;

        // advance chemistry
        if (es_ode_L->get_A_e() == 0) {
            fpe.advance_chemistry_left_cylinder(Q_L, dt_sys);
        }
        if (es_ode_R->get_A_e() == 0) {
            fpe.advance_chemistry_right_cylinder(Q_R, dt_sys);
        }
        fpe.advance_dynamics(Q_L, Q_R, dt_sys);

        // exhaust cylinders
        fpe.advance_fluid_dynamics_left_cylinder(u_p, &Q_L, dt_sys);
        fpe.advance_fluid_dynamics_right_cylinder(u_p, &Q_R, dt_sys);

        // increment time
        gd->t += dt_sys;

        // finalise step
        fpe.finalise_step(Q_L, Q_R);

        // set the velocity to zero to prevent doing this step again
        fpe.get_yout(yout_fpe);
        fpe.get_yout_L(yout_L);
        fpe.get_yout_R(yout_R);

        yout_fpe[__U] = 0;
```

```
        print_state(fname, gd->t, yout_fpe, yout_L, yout_R, Q_L, Q_R, fpe_ode->
            get_m_gL(), fpe_ode->get_m_gR());

        dt_sys = dt_sys0; // the timestep has changed...
        Tig.push_back(Q_L.T[0]);
        Tig.push_back(Q_R.T[0]);
        x_s = yout_fpe[__X];

        continue;
    }

    // exhaust cylinders
    fpe.advance_fluid_dynamics_left_cylinder(u_p, &Q_L, dt_sys);
    fpe.advance_fluid_dynamics_right_cylinder(u_p, &Q_R, dt_sys);
    if ((es_ode_L->get_A_e() == 0) && (x_p < (x0 - port - gap)) && (u_p < 0.0)) {
        if (x_flag < 0) {
            // assign mass of left cylinder
            fpe.reset_intake_mass_left_cylinder(Q_L);

            // switch flag
            x_flag *= -1;
            //fpe.set_velocity(-u0);
        }
        // it doesn't matter how many times we do this.
        Q_exL.massf = Q_L.massf;
    }
    if ((es_ode_R->get_A_e() == 0) && (x_p > x0) && (u_p > 0.0)) {
        if (x_flag > 0) {
            // assign mass of right cylinder
            fpe.reset_intake_mass_right_cylinder(Q_R);

            // switch flag
            x_flag *= -1;
            //fpe.set_velocity(u0);

            ++nc; // increment cycle counter
            if (nc == no_cycles - 1) {
              string str;
              str.append(fname.begin(),fname.end()-4);
              str.append("-final-stroke.dat");
              fname = str;

              fout = fopen(fname.c_str(), "w");
              fprintf(fout, "# t, x, u, e_L, e_R, Q_L, Q_R, w_f, dm_L, dme_L, dm_R,
                  dme_R, T_L, p_L, rho_L, m_gL, T_R, p_R, rho_R, m_gR, ");
              for (int i = 0; i < nsp; ++i) {
                  fprintf(fout, "%s_L, ", g->species_name(i).c_str());
              }
              for (int i = 0; i < nsp; ++i) {
                  fprintf(fout, "%s_R, ", g->species_name(i).c_str());
              }
              fprintf(fout, "\n");
              fclose(fout);
            }
            if (nc == no_cycles) {
                rvalue.push_back(x_s);
                rvalue.push_back(fpe_ode->get_m_gL());
```

```
                    rvalue.push_back(fpe.get_mgLout());
                    rvalue.push_back(Tig[2]);
                    rvalue.push_back(Tig[3]);
                    rvalue.push_back(yout_fpe[__Q_L]);
                    rvalue.push_back(yout_fpe[__Q_R]);
                    rvalue.push_back(yout_fpe[__F]);

                    return rvalue;
                }
            }
            // it doesn't matter how many times we do this.
            Q_exR.massf = Q_R.massf;
        }

        // increment time
        gd->t += dt_sys;

        // finalise step
        fpe.finalise_step(Q_L, Q_R);

        // increment iwrite
        ++iwrite;
        if (iwrite >= nwrite) {
            fpe.get_yout(yout_fpe);
            fpe.get_yout_L(yout_L);
            fpe.get_yout_R(yout_R);

            print_state(fname, gd->t, yout_fpe, yout_L, yout_R, Q_L, Q_R, fpe_ode->
                get_m_gL(), fpe_ode->get_m_gR());
            //print_energy(fname, gd->t, yout_fpe, Q_L, Q_R, m_p, m_gL, m_gR);
            iwrite = 0;
        }
    }

    fout = fopen(fname.c_str(), "a");
    fprintf(fout, "# done.\n");
    fclose(fout);

    return rvalue;
}
```

Listing B.4: *Test functions, source file.*

B.3 Free-piston engine system

```
#ifndef FPE_SYSTEM_HH
#define FPE_SYSTEM_HH

// \author Brendan T. O'Flaherty
// \brief  Free-piston engine systems

#include <vector>
#include <valarray>
#include "../../lib/util/source/useful.h"
#include "../../lib/nm/source/ode_solver.hh"
#include "../../lib/gas/models/physical_constants.hh"
#include "../../lib/gas/models/gas_data.hh"
#include "../../lib/gas/models/gas-model.hh"
#include "fpe_kernel.hh"
#include "fpe_control.hh"

int copy_array(std::valarray<double> &src, std::valarray<double> &dst);

class Free_piston_compressor_ode : public OdeSystem {
public:
    Free_piston_compressor_ode(int ndim, bool test_flag);
    Free_piston_compressor_ode(gas_data *Q,
                               double m_p,
                               double x,
                               double u_p,
                               double L_p,
                               double L_c,
                               double D,
                               double error_tol,
                               int ndim,
                               bool test_flag,
                               double p_b=0);
    Free_piston_compressor_ode(const Free_piston_compressor_ode &fpc);
     ~Free_piston_compressor_ode();

    void set_m0(double m_g) { m_g_ = m_g; }
    void set_gamma(double gamma) { gamma_ = gamma; }
    int calculate_geometry(double x_p);
    int eval(const std::valarray<double> &y, std::valarray<double> &ydot);
    double get_qdd() { return qdd_; }
    double get_m0() { return m_g_; }
    double get_V() { return V_R_; }
    double stepsize_select(const std::valarray<double> &y);
    double calculate_system_energy(double u_p, double V_R);
    bool passes_system_test(double initial_energy, double V_R, std::valarray<double> &y);

private:
    gas_data *Q_; // the gas state is not changed in the eval routine

    double F_c_, F_f_, q_, qdd_, d_, a_, gamma_; // control, friction and heat loss
    double m_p_, u_p0_, m_g_; // piston mass, initial velocity, gas mass
    double L_p_, L_c_, x_R_, D_, A_; // fixed geometry
    double L_R_, V_R_, S_R_; // variable geometry
    double err_tol_; // tolerance
    double p_b_; // back pressure
```

```
    std::valarray<double> ydot_;
};

class Crankshaft_compressor_ode : public OdeSystem {
public:
    Crankshaft_compressor_ode(int ndim, bool test_flag);
    Crankshaft_compressor_ode(gas_data *Q,
                              double m_p,
                              double x_p,
                              double u_p,
                              double L_p,
                              double L_c,
                              double D,
                              double error_tol,
                              int ndim,
                              bool test_flag,
                              double p_b=0);
    Crankshaft_compressor_ode(const Crankshaft_compressor_ode &fpc);
    ~Crankshaft_compressor_ode();

    void set_m0(double m_g) { m_g_ = m_g; }
    void set_gamma(double gamma) { gamma_ = gamma; }
    int eval(const std::valarray<double> &y, std::valarray<double> &ydot);
    int calculate_geometry(double x_p);
    double get_qdd() { return qdd_; }
    double get_m0() { return m_g_; }
    double get_V() { return V_R_; }
    double stepsize_select(const std::valarray<double> &y);
    double calculate_system_energy(double u_p, double V_R);
    bool passes_system_test(double initial_energy, double V_R, std::valarray<double> &y);

private:
    gas_data *Q_; // the gas state is not changed in the eval routine

    double F_c_, F_f_, q_, qdd_, d_, a_, gamma_; // control, friction and heat loss
    double m_p_, u_p0_, omega_, m_g_; // piston mass, initial velocity, gas mass
    double L_p_, L_c_, x_R_, D_, A_; // fixed geometry
    double L_R_, V_R_, S_R_; // variable geometry
    double err_tol_; // tolerance
    double p_b_; // back pressure
    std::valarray<double> ydot_;
};

class Free_piston_compressor {
public:
    Free_piston_compressor(int ndim, bool test_flag);
    Free_piston_compressor(Free_piston_compressor_ode *fpc_ode,
                           std::valarray<double> yin,
                           double dt_sys,
                           double dt_therm,
                           double rho0);
    Free_piston_compressor(const Free_piston_compressor &fpc);
    ~Free_piston_compressor();

    std::valarray<double> get_yin() { return yin_; }
    std::valarray<double> get_yout() { return yout_; }
    int finalise_step(gas_data &Q);
```

```
    int instantaneous_heat_addition(double dh);
    int advance_chemistry(gas_data &Q, double &dt);
    int advance_dynamics(gas_data &Q, double &dt);

private:
    OdeSolver *ode_solver_;
    Free_piston_compressor_ode *fpc_ode_;

    double dt_sys_; // system step
    double dt_therm_; // thermal step
    double dt_chem_; // chemical step
    double m_g0_; // gas mass
    double initial_energy_; // system energy
    std::valarray<double> yin_;
    std::valarray<double> yout_;
};

class Crankshaft_compressor {
public:
    Crankshaft_compressor(int ndim, bool test_flag);
    Crankshaft_compressor(Crankshaft_compressor_ode *fpc_ode,
                          std::valarray<double> yin,
                          double dt_sys,
                          double dt_therm,
                          double rho0);
    Crankshaft_compressor(const Crankshaft_compressor &fpc);
    ~Crankshaft_compressor();

    std::valarray<double> get_yin() { return yin_; }
    std::valarray<double> get_yout() { return yout_; }
    int finalise_step(gas_data &Q);
    int instantaneous_heat_addition(double dh);
    int advance_chemistry(gas_data &Q, double &dt);
    int advance_dynamics(gas_data &Q, double &dt);

private:
    OdeSolver *ode_solver_;
    Crankshaft_compressor_ode *fpc_ode_;
    double dt_sys_; // system step
    double dt_therm_; // thermal step
    double dt_chem_; // chemical step
    double m_g0_; // gas mass
    double initial_energy_; // system energy
    std::valarray<double> yin_;
    std::valarray<double> yout_;
};

class Exhaust_system_ode : public OdeSystem {
public:
    Exhaust_system_ode(int ndim, bool test_flag);
    Exhaust_system_ode(gas_data* Q,
                       gas_data* Q_i,
                       gas_data* Q_e,
                       double D,
                       double error_tol,
                       int ndim,
                       bool displace,
```

```
                        bool test_flag=false);
    Exhaust_system_ode(const Exhaust_system_ode &fpe);
    ~Exhaust_system_ode();

    void set_displace(bool displace) { displace_ = displace; }
    void set_Q_e(gas_data *Q_e) { Q_e_ = Q_e; }
    int eval(const std::valarray<double> &y, std::valarray<double> &ydot);
    int fully_open_valves();
    int move_valves_left_cylinder(double m_g,
                                  double u,
                                  double m_g0,
                                  double p,
                                  double &dt);
    int move_valves_right_cylinder(double m_g,
                                   double u,
                                   double m_g0,
                                   double p,
                                   double &dt);
    bool passes_system_test(double initial_energy, std::valarray<double> &y);
    bool displace() { return displace_; }
    double get_A_i() { return A_i_; }
    double get_A_e() { return A_e_; }
    double stepsize_select(const std::valarray<double> &y);
    std::vector<double> get_F_i() { return F_i_; }
    std::vector<double> get_F_e() { return F_e_; }

private:
    gas_data *Q_i_, *Q_, *Q_e_; // gas at intake valve, exhaust valve and in cylinder
    Valve ve_, vi_; // exhaust and intake valves

    double A_i_, A_e_; // area of intake and exit valves
    double err_tol_; // tolerance
    bool displace_;
    std::valarray<double> ydot_;
    std::vector<double> F_i_, F_e_;
};

class Exhaust_system {
public:
    Exhaust_system(int ndim, bool test_flag);
    Exhaust_system(Exhaust_system_ode *es_ode,
                   std::valarray<double> yin,
                   double dt_sys,
                   double V,
                   gas_data &Q);
    Exhaust_system(const Exhaust_system &es);
    ~Exhaust_system();

    void set_volume(double V) { V_ = V; }
    void set_m0(double m0) { m0_ = m0; }
    int get_yin(std::valarray<double> &dst) { return copy_array(yin_, dst); }
    int get_yout(std::valarray<double> &dst) { return copy_array(yout_, dst); }
    int finalise_step(gas_data &Q);
    int advance_fluid_dynamics(double &dt);
    double get_m0() { return m0_; }

private:
```

```
    OdeSolver *ode_solver_;
    Exhaust_system_ode *es_ode_;

    double dt_sys_; // system step
    double V_; // volume
    double m0_; // initial mass
    std::valarray<double> yin_;
    std::valarray<double> yout_;
};

class Free_piston_engine_ode : public OdeSystem {
public:
    Free_piston_engine_ode(int ndim, bool test_flag);
    Free_piston_engine_ode(gas_data* Q2,
                           gas_data* Q3,
                           double m_p,
                           double x_p,
                           double u_p,
                           double L_p,
                           double L_c,
                           double x_L,
                           double x_R,
                           double D,
                           double m_gL,
                           double m_gR,
                           double error_tol,
                           int ndim,
                           bool test_flag);
    Free_piston_engine_ode(const Free_piston_engine_ode &fpe);
     ~Free_piston_engine_ode();

    void set_m_gL(double m_gL) { m_gL_ = m_gL; }
    void set_m_gR(double m_gR) { m_gR_ = m_gR; }
    void set_gamma_L(double gamma_L) { gamma_L_ = gamma_L; }
    void set_gamma_R(double gamma_R) { gamma_R_ = gamma_R; }
    int eval(const std::valarray<double> &y, std::valarray<double> &ydot);
    int calculate_geometry(double x_p);
    bool passes_system_test(double initial_energy,
                            double V_L,
                            double V_R,
                            std::valarray<double> &y);
    double get_m_gL() { return m_gL_; }
    double get_m_gR() { return m_gR_; }
    double get_F() { return F_c_; }
    double get_V_L() { return V_L_; }
    double get_V_R() { return V_R_; }
    double get_u_pL0() { return u_pL0_; }
    double get_u_pR0() { return u_pR0_; }
    double stepsize_select(const std::valarray<double> &y);
    double calculate_system_energy(double u_p, double V_L, double V_R);

private:
    // the gas state is not changed in the eval routine
    gas_data* Q_L_;
    gas_data* Q_R_;

    double F_c_, F_f_; // control, friction
```

```
    double q_ddL_, q_ddR_, q_L_, q_R_; // heat loss
    double d_L_, a_L_, d_R_, a_R_, gamma_L_, gamma_R_; // boundary layer variables
    double L_p_, L_c_, x_L_, x_R_, D_, A_; // fixed geometry
    double L_R_, V_R_, S_R_; // variable geometry
    double L_L_, V_L_, S_L_; // variable geometry
    double b_; // compression ring thickness
    double m_p_, m_gL_, m_gR_; // piston mass, gas masses
    double u_pL0_, u_pR0_; // initial velocity
    double err_tol_; // tolerance
    std::valarray<double> ydot_;
};

class Free_piston_engine {
public:
    Free_piston_engine(int ndim_fpe,
                       int ndim_es,
                       bool test_flag,
                       double dt_sys,
                       double dt_therm);
    Free_piston_engine(Free_piston_engine_ode *fpe_ode,
                       Exhaust_system_ode* es_ode_L,
                       Exhaust_system_ode* es_ode_R,
                       std::valarray<double> y_fpe,
                       std::valarray<double> y_es,
                       double dt_sys,
                       double dt_therm,
                       double m_gL0,
                       double m_gR0);
    Free_piston_engine(const Free_piston_engine &fpe);
     ~Free_piston_engine();

    void set_velocity(double u) { yout_[__U] = u; }
    int get_yin(std::valarray<double> &dst) { return copy_array(yin_, dst); }
    int get_yout(std::valarray<double> &dst) { return copy_array(yout_, dst); }
    int get_yin_L(std::valarray<double> &dst) { return copy_array(yin_L_, dst); }
    int get_yout_L(std::valarray<double> &dst) { return copy_array(yout_L_, dst); }
    int get_yin_R(std::valarray<double> &dst) { return copy_array(yin_R_, dst); }
    int get_yout_R(std::valarray<double> &dst) { return copy_array(yout_R_, dst); }
    int finalise_step(gas_data &Q_L, gas_data &Q_R);
    int instantaneous_heat_addition_left_cylinder(double dh);
    int instantaneous_heat_addition_right_cylinder(double dh);
    int reset_intake_mass_left_cylinder(gas_data &Q);
    int reset_intake_mass_right_cylinder(gas_data &Q);
    int move_left_valves(double u_p, double pL, double &dt);
    int move_right_valves(double u_p, double pR, double &dt);
    int advance_chemistry_left_cylinder(gas_data &Q, double &dt);
    int advance_chemistry_right_cylinder(gas_data &Q, double &dt);
    int advance_dynamics(gas_data &Q_L, gas_data &Q_R, double &dt);
    int advance_fluid_dynamics_left_cylinder(double u_p, gas_data *Q, double &dt);
    int advance_fluid_dynamics_right_cylinder(double u_p, gas_data *Q, double &dt);
    double get_mgLout() { return m_gLout_; }
    double get_mgRout() { return m_gRout_; }
    double get_system_energy() { return system_energy_; }

private:
    OdeSolver *ode_solver_fpe_;
    Free_piston_engine_ode *fpe_ode_;
```

```
    OdeSolver *ode_solver_es_;
    Exhaust_system_ode *es_ode_L_; // exhaust ode
    Exhaust_system_ode *es_ode_R_; // exhaust ode
    double m_g0_;  // ideal cylinder mass
    double m_gL0_; // left cylinder mass
    double m_gR0_; // right cylinder mass
    double m_gRout_; // left exhaust mass
    double m_gLout_; // right exhaust mass
    double dt_sys_; // system step
    double dt_therm_; // thermal step
    double dt_chem_; // chemical step
    double initial_energy_; // system energy
    double system_energy_; // system energy
    std::valarray<double> yin_;
    std::valarray<double> yout_;
    std::valarray<double> yin_L_;
    std::valarray<double> yout_L_;
    std::valarray<double> yin_R_;
    std::valarray<double> yout_R_;
};

int perform_chemical_increment(gas_data &Q, double &dt, double &dt_therm, double &dt_chem
    );

#endif
```

Listing B.5: *Free-piston engine system, header file.*

```
#include <cstdio>
#include "fpe_kernel.hh"
#include "fpe_models.hh"
#include "fpe_system.hh"
#include "sod.hh"

using namespace std;

const double valve_velocity = 10.0;
const double eps1 = 0.001;
const double step_upper_limit = 1.0e-5;
const double step_lower_limit = 1.0e-10;
const double zero_tol = 1.0e-30;
const double DT_THERM_REDUCE = 0.2;

Free_piston_compressor_ode::
Free_piston_compressor_ode(int ndim, bool test_flag)
    : OdeSystem(ndim, test_flag) {}

Free_piston_compressor_ode::
Free_piston_compressor_ode(gas_data *Q,
                           double m_p,
                           double x,
                           double u_p,
                           double L_p,
                           double L_c,
                           double D,
                           double error_tol,
                           int ndim,
```

```
                            bool test_flag,
                            double p_b)
    : OdeSystem(ndim, test_flag),
      m_p_(m_p),
      u_p0_(u_p),
      D_(D),
      err_tol_(error_tol),
      p_b_(p_b)
{
    Q_ = Q; // assign pointer
    F_c_ = F_f_ = q_ = qdd_ = d_ = a_ = 0.0; // no losses by default
    gamma_ = 0.0;
    m_g_ = 0.0;
    L_p_ = L_p;
    L_c_ = L_c;
    x_R_ = L_c;
    D_ = D;
    A_ = 0.25*PI*D*D;
    Gas_model* g = get_gas_model_ptr();
    gamma_ = g->gamma(*Q);
    ydot_.resize(ndim);

    calculate_geometry(x);
}

Free_piston_compressor_ode::
Free_piston_compressor_ode(const Free_piston_compressor_ode &fpc)
    : OdeSystem(fpc.ndim_, fpc.apply_system_test_)
{
    // incomplete
}

Free_piston_compressor_ode::
~Free_piston_compressor_ode() {}

int
Free_piston_compressor_ode::
eval(const valarray<double> &y, valarray<double> &ydot)
{
    // unpack y
    // double x = y[__X];
    double u = y[__U];

    // friction
    // double b = 0.005; // 5mm ring thickness
    // chevron_seal_friction(F_f_, Q_->p, PC_P_atm, b, D_, m_p_, u);

    double a = (-Q_->p*A_ + p_b_*A_ + F_c_ - F_f_)/m_p_;
    double dedt = (Q_->p*A_*u)/m_g_;

    // heat transfer models, uncomment as needed
    // annand_heat_flux(qdd_, Q_->k[0], D_, u_p0_, Q_->mu, Q_->rho, Q_->T[0]);
    // boundary_layer_thickness(d_, a_, x, u, u_p0_, Q_->T[0], PC_T_ref, L_R_, x_R_, D_,
        gamma_);
    // laminar_heat_flux(qdd_, Q_->k[0], d_, a_, Q_->T[0], PC_T_ref);
    // lawton_heat_flux(qdd_, Q_->k[0], D_, u, u_p0_, Q_->mu, Q_->T[0], Q_->rho,
        PC_T_ref, L_R_, gamma_);
```

```
    // Gas_model* g = get_gas_model_ptr();
    // double dTdt = dedt/g->Cv(*Q_);
    // annand_and_pinfold_heat_flux(qdd_, Q_->k[0], D_, u, u_p0_, Q_->mu, Q_->T[0], Q_->
        rho, PC_T_ref, dTdt);

    q_ = qdd_*S_R_/m_g_;
    if(isnan(q_)) { q_ = 0.0; }
    dedt = dedt - q_;

    double temp[] = {u, a, dedt, q_};//, F_f_*u};
    for (size_t i = 0; i < ydot.size(); ++i) {
        ydot[i] = temp[i];
    }

    return SUCCESS;
}

double
Free_piston_compressor_ode::
stepsize_select(const valarray<double> &y)
{
    eval(y, ydot_);
    double min_dt = step_upper_limit; // to get us started
    double old_dt = 0.0;

    for (int i = 0; i < ndim_; ++i) {
        if (fabs(ydot_[i]) > zero_tol) {
            old_dt = fabs(y[i]/ydot_[i]);
            if( old_dt < min_dt ) {
                min_dt = old_dt;
            }
        }
    }

    double dt = eps1 * min_dt;

    // Impose upper and lower step limits
    if( dt > step_upper_limit )
        dt = step_upper_limit;

    if ( dt < step_lower_limit )
        dt = step_lower_limit;

    return dt;
}

double
Free_piston_compressor_ode::
calculate_system_energy(double u_p, double V_R)
{
    double ke = 0.5*m_p_*u_p*u_p;
    Gas_model* g = get_gas_model_ptr();
    double e = g->total_internal_energy(*Q_);

    return ke + V_R*Q_->rho*e;
}
```

```
bool
Free_piston_compressor_ode::
passes_system_test(double initial_energy, double V_R, valarray<double> &y)
{
    double u = y[__U];
    double current_energy = calculate_system_energy(u, V_R);
    double err = (fabs(current_energy) - fabs(initial_energy))/fabs(initial_energy);

    if (err < err_tol_)
        return true;
    else
        return false;
}

int
Free_piston_compressor_ode::
calculate_geometry(double x_p)
{
    // cylinder length, volume, surface area
    L_R_ = x_R_ - (x_p + L_p_/2.0);
    V_R_ = L_R_*A_;
    S_R_ = 2*A_ + PI*D_*L_R_;

    return SUCCESS;
}

Crankshaft_compressor_ode::
Crankshaft_compressor_ode(int ndim, bool test_flag)
    : OdeSystem(ndim, test_flag) {}

Crankshaft_compressor_ode::
Crankshaft_compressor_ode(gas_data *Q,
                          double m_p,
                          double x_p,
                          double u_p,
                          double L_p,
                          double L_c,
                          double D,
                          double error_tol,
                          int ndim,
                          bool test_flag,
                          double p_b)
    : OdeSystem(ndim, test_flag),
      m_p_(m_p),
      u_p0_(u_p),
      D_(D),
      err_tol_(error_tol),
      p_b_(p_b)
{
    Q_ = Q; // assign pointer
    F_c_ = F_f_ = q_ = qdd_ = d_ = a_ = 0.0; // no losses by default
    gamma_ = 0.0;
    m_g_ = 0.0;
    omega_ = u_p;
    L_p_ = L_p;
    L_c_ = L_c;
```

```
    x_R_ = L_c;
    D_ = D;
    A_ = 0.25*PI*D*D;
    Gas_model* g = get_gas_model_ptr();
    gamma_ = g->gamma(*Q);
    ydot_.resize(ndim);

    calculate_geometry(x_p);
}

Crankshaft_compressor_ode::
Crankshaft_compressor_ode(const Crankshaft_compressor_ode &cc)
    : OdeSystem(cc.ndim_, cc.apply_system_test_)
{
    // incomplete
}

Crankshaft_compressor_ode::
~Crankshaft_compressor_ode() {}

int
Crankshaft_compressor_ode::
eval(const valarray<double> &y, valarray<double> &ydot)
{
    // unpack y
    //double x = y[__X];
    double u = y[__U];
    double A = y[__A];

    double c = 15.6;
    double r = L_c_*(1.0 - 1.0/c)/2.0;
    double l = 2*r;
    double den = l*l - r*r*sin(A)*sin(A);
    double a = (-r*cos(A) - r*r*(cos(A)*cos(A) - sin(A)*sin(A))/pow(den,0.5) -
                pow(r,4)*sin(A)*sin(A)*cos(A)*cos(A)/pow(den,1.5))*omega_*omega_;
    double dedt = (Q_->p*A_*u)/m_g_;
    u_p0_ = omega_ * r;

    // heat transfer models, uncomment as needed
    //annand_heat_flux(qdd_, Q_->k[0], D_, u_p0_, Q_->mu, Q_->rho, Q_->T[0]);
    //boundary_layer_thickness(d_, a_, x, u, u_p0_, Q_->T[0], PC_T_ref, L_R_, x_R_, D_,
        gamma_);
    //laminar_heat_flux(qdd_, Q_->k[0], d_, a_, Q_->T[0], PC_T_ref);
    //lawton_heat_flux(qdd_,  Q_->k[0], D_, u, u_p0_, Q_->mu, Q_->T[0], Q_->rho, PC_T_ref
        , L_, gamma_);
    //Gas_model* g = get_gas_model_ptr();
    //double dTdt = dedt/g->Cv(*Q_);
    //annand_and_pinfold_heat_flux(qdd_, Q_->k[0], D_, u, u_p0_, Q_->mu, Q_->T[0], Q_->
        rho, PC_T_ref, dTdt);

    q_ = qdd_*S_R_/m_g_;
    if(isnan(q_)) { q_ = 0.0; }
    dedt = dedt - q_;

    double temp[] = {u, a, dedt, q_, omega_};
    for (size_t i = 0; i < ydot.size(); ++i) {
        ydot[i] = temp[i];
```

```
    }

    return SUCCESS;
}

double
Crankshaft_compressor_ode::
stepsize_select(const valarray<double> &y)
{
    eval(y, ydot_);
    double min_dt = step_upper_limit; // to get us started
    double old_dt = 0.0;

    for (int i = 0; i < ndim_; ++i) {
        if (fabs(ydot_[i]) > zero_tol) {
            old_dt = fabs(y[i]/ydot_[i]);
            if( old_dt < min_dt ) {
                min_dt = old_dt;
            }
        }
    }

    double dt = eps1 * min_dt;

    // Impose upper and lower step limits
    if( dt > step_upper_limit )
        dt = step_upper_limit;

    if ( dt < step_lower_limit )
        dt = step_lower_limit;

    return dt;
}

double
Crankshaft_compressor_ode::
calculate_system_energy(double u_p, double V_R)
{
    double ke = 0.5*m_p_*u_p*u_p;
    Gas_model* g = get_gas_model_ptr();
    double e = g->total_internal_energy(*Q_);

    return ke + V_R*Q_->rho*e;
}


bool
Crankshaft_compressor_ode::
passes_system_test(double initial_energy, double V_R, valarray<double> &y)
{
    double u = y[__U];
    double current_energy = calculate_system_energy(u, V_R);
    double err = (fabs(current_energy) - fabs(initial_energy))/fabs(initial_energy);

    if (err < err_tol_)
        return true;
    else
```

```
        return false;
}

int
Crankshaft_compressor_ode::
calculate_geometry(double x_p)
{
    // cylinder length, volume, surface area
    L_R_ = x_R_ - (x_p + L_p_/2.0);
    V_R_ = L_R_*A_;
    S_R_ = 2*A_ + PI*D_*L_R_;

    return SUCCESS;
}

Free_piston_compressor::
Free_piston_compressor(int ndim, bool test_flag)
{
    yin_.resize(ndim);
    yout_.resize(ndim);
}

Free_piston_compressor::
Free_piston_compressor(Free_piston_compressor_ode* fpc_ode,
                       valarray<double> yin,
                       double dt_sys,
                       double dt_therm,
                       double rho0)
{
    std::valarray<double> ydot_;
    int ndim = yin.size();
    ode_solver_ = new OdeSolver(string("rkf_solver"), ndim, string("rkf"));
    fpc_ode_ = fpc_ode;

    yin_.resize(ndim);
    yout_.resize(ndim);

    for (int i = 0; i < ndim; ++i) {
        yin_[i] = yout_[i] = yin[i];
    }

    m_g0_ = rho0*fpc_ode_->get_V();
    fpc_ode_->set_m0(m_g0_);

    dt_sys_ = dt_sys;
    dt_therm_ = dt_therm;
    dt_chem_ = -1.0;

    double u0 = yin[__U];
    initial_energy_ = fpc_ode_->calculate_system_energy(u0, fpc_ode_->get_V());
}

Free_piston_compressor::
Free_piston_compressor(const Free_piston_compressor &fpc)
{
    ode_solver_ = new OdeSolver(*(fpc.ode_solver_));
    fpc_ode_ = new Free_piston_compressor_ode(*(fpc.fpc_ode_));
```

```
    yin_ = fpc.yin_;
    yout_ = fpc.yout_;

    dt_sys_ = fpc.dt_sys_;
    dt_therm_ = fpc.dt_therm_;
    dt_chem_ = fpc.dt_chem_;
}

Free_piston_compressor::
~Free_piston_compressor()
{
    delete ode_solver_;
    delete fpc_ode_;
}

int
Free_piston_compressor::
instantaneous_heat_addition(double dh)
{
    yout_[__E] += dh;

    return SUCCESS;
}

int
Free_piston_compressor::
advance_chemistry(gas_data &Q, double &dt)
{
    // as long as this operation is performed on a finalised state (i.e. at the
    // end or beginning of a timestep) there should be no problem. If it is
    // performed before yin_ and yout_ have been synchronised, then the energy
    // released from combustion will not be added to the state vector.

    int status = perform_chemical_increment(Q, dt, dt_therm_, dt_chem_);
    yout_[__E] = Q.e[0];

    return status;
}

int
Free_piston_compressor::
advance_dynamics(gas_data &Q, double &dt)
{
    global_data* gd = get_global_data_ptr();
    ode_solver_->solve_over_interval(*fpc_ode_, gd->t, gd->t+dt, &dt_sys_, yin_, yout_);

    return SUCCESS;
}

int
Free_piston_compressor::
finalise_step(gas_data &Q)
{
    // unpack state
    double x = yout_[__X];
    double u = yout_[__U];
```

```
    double e = yout_[__E];

    // update state
    fpc_ode_->calculate_geometry(x);
    Q.rho = m_g0_/fpc_ode_->get_V();
    Q.e[0] = e;

    Gas_model* g = get_gas_model_ptr();
    if (g->eval_thermo_state_rhoe(Q) != SUCCESS) {
        printf("error calculating eval_thermo_state_rhoe()\n");
        printf("y = [%g, %g, %g]\n", x, u, e);
        print_gas_data(Q);
        exit(1);
    }
    if (g->eval_transport_coefficients(Q) != SUCCESS) {
        printf("error calculating eval_transport_coefficients()\n");
        print_gas_data(Q);
        exit(1);
    }
    fpc_ode_->set_gamma(g->gamma(Q));

    // copy vectors for next timestep
    yin_ = yout_;

    return SUCCESS;
}

Crankshaft_compressor::
Crankshaft_compressor(int ndim, bool test_flag)
{
    yin_.resize(ndim);
    yout_.resize(ndim);
}

Crankshaft_compressor::
Crankshaft_compressor(Crankshaft_compressor_ode* fpc_ode,
                      valarray<double> yin,
                      double dt_sys,
                      double dt_therm,
                      double rho0)
{
    int ndim = yin.size();
    ode_solver_ = new OdeSolver(string("rkf_solver"), ndim, string("rkf"));
    fpc_ode_ = fpc_ode;

    yin_.resize(ndim);
    yout_.resize(ndim);

    for (int i = 0; i < ndim; ++i) {
        yin_[i] = yout_[i] = yin[i];
    }

    m_g0_ = rho0*fpc_ode_->get_V();
    fpc_ode_->set_m0(m_g0_);

    dt_sys_ = dt_sys;
    dt_therm_ = dt_therm;
```

```
    dt_chem_ = -1.0;

    double u0 = yin[__U];
    initial_energy_ = fpc_ode_->calculate_system_energy(u0, fpc_ode_->get_V());
}

Crankshaft_compressor::
Crankshaft_compressor(const Crankshaft_compressor &fpc)
{
    ode_solver_ = new OdeSolver(*(fpc.ode_solver_));
    fpc_ode_ = new Crankshaft_compressor_ode(*(fpc.fpc_ode_));

    yin_ = fpc.yin_;
    yout_ = fpc.yout_;

    dt_sys_ = fpc.dt_sys_;
    dt_therm_ = fpc.dt_therm_;
    dt_chem_ = fpc.dt_chem_;
}

Crankshaft_compressor::
~Crankshaft_compressor()
{
    delete ode_solver_;
    delete fpc_ode_;
}

int
Crankshaft_compressor::
instantaneous_heat_addition(double dh)
{
    yout_[__E] += dh;

    return SUCCESS;
}

int
Crankshaft_compressor::
advance_chemistry(gas_data &Q, double &dt)
{
    // as long as this operation is performed on a finalised state (i.e. at the
    // end or beginning of a timestep) there should be no problem. If it is
    // performed before yin_ and yout_ have been synchronised, then the energy
    // released from combustion will not be added to the state vector.

    int status = perform_chemical_increment(Q, dt, dt_therm_, dt_chem_);
    yout_[__E] = Q.e[0];

    return status;
}

int
Crankshaft_compressor::
advance_dynamics(gas_data &Q, double &dt)
{
    global_data* gd = get_global_data_ptr();
    ode_solver_->solve_over_interval(*fpc_ode_, gd->t, gd->t+dt, &dt_sys_, yin_, yout_);
```

```
    return SUCCESS;
}

int
Crankshaft_compressor::
finalise_step(gas_data &Q)
{
    // unpack state
    double x = yout_[__X];
    double u = yout_[__U];
    double e = yout_[__E];

    // update state
    fpc_ode_->calculate_geometry(x);
    Q.rho = m_g0_/fpc_ode_->get_V();
    Q.e[0] = e;

    Gas_model* g = get_gas_model_ptr();
    if (g->eval_thermo_state_rhoe(Q) != SUCCESS) {
        printf("error calculating eval_thermo_state_rhoe()\n");
        printf("y = [%g, %g, %g]\n", x, u, e);
        print_gas_data(Q);
        exit(1);
    }
    if (g->eval_transport_coefficients(Q) != SUCCESS) {
        printf("error calculating eval_transport_coefficients()\n");
        print_gas_data(Q);
        exit(1);
    }
    fpc_ode_->set_gamma(g->gamma(Q));
    // copy vectors for next timestep
    yin_ = yout_;

    return SUCCESS;
}

Exhaust_system_ode::
Exhaust_system_ode(int ndim, bool test_flag)
    : OdeSystem(ndim, test_flag) {}

Exhaust_system_ode::
Exhaust_system_ode(gas_data* Q,
                   gas_data* Q_i,
                   gas_data* Q_e,
                   double D,
                   double error_tol,
                   int ndim,
                   bool displace,
                   bool test_flag)
    : OdeSystem(ndim, test_flag),
      err_tol_(error_tol),
      displace_(displace)
{
    Q_i_ = Q_i;
    Q_ = Q;
    Q_e_ = Q_e;
```

```
    vi_ = Valve("intake", valve_velocity, D/sqrt(2));
    ve_ = Valve("exhaust", valve_velocity, D/sqrt(2));

    ydot_.resize(ndim,0);
    F_i_.resize(ndim,0);
    F_e_.resize(ndim,0);
}

Exhaust_system_ode::
Exhaust_system_ode(const Exhaust_system_ode &es)
    : OdeSystem(es.ndim_, es.apply_system_test_)
{
    // incomplete
}

Exhaust_system_ode::
~Exhaust_system_ode() {}

int
Exhaust_system_ode::
eval(const std::valarray<double> &y, std::valarray<double> &ydot)
{
    // calculate valve intake and exit areas
    A_i_ = vi_.eval_discharge_coefficient()*vi_.get_area();
    A_e_ = ve_.eval_discharge_coefficient()*ve_.get_area();

    if ((A_i_ == 0) && (A_e_ == 0)) {
        // both valves closed, set flux to zero and return
        for (int i = 0; i < ndim_; ++i) {
            ydot[i] = 0.0;
        }
        return SUCCESS;
    }

    if (Q_i_->p >= Q_->p) {
        eval_flux(F_i_, Q_i_, Q_, err_tol_);
    } else {
        eval_flux(F_i_, Q_, Q_i_, err_tol_);
        for (int i = 0; i < ndim_-1; ++i) F_i_[i] = -F_i_[i];
    }
    if (Q_->p >= Q_e_->p) {
        eval_flux(F_e_, Q_, Q_e_, err_tol_);
        for (int i = 0; i < ndim_-1; ++i) F_e_[i] = -F_e_[i];
    } else {
        eval_flux(F_e_, Q_e_, Q_, err_tol_);
    }

    Gas_model *g = get_gas_model_ptr();
    int nsp = g->get_number_of_species();

    int i;
    for (i = 0; i < 2; ++i) {
        ydot[i] = A_i_*F_i_[i] + A_e_*F_e_[i];
    }
    if (displace_) {
        // perfect displacement
```

```
        for (i = 0; i < nsp; ++i) {
            ydot[i+2] = (A_i_*F_i_[i+2] + A_e_*F_e_[0]*Q_e_->massf[i]);
        }
    }
    else {
        // perfect mixing
        for (i = 0; i < nsp; ++i) {
            ydot[i+2] = A_i_*F_i_[i+2] + A_e_*F_e_[i+2];
        }
    }
    ydot[2+nsp] = -A_e_*F_e_[0]; // positive mass exiting cylinder

    return SUCCESS;
}

double
Exhaust_system_ode::
stepsize_select(const std::valarray<double> &y)
{
    eval(y, ydot_);
    double min_dt = step_upper_limit; // to get us started
    double old_dt = 0.0;

    for (int i = 0; i < ndim_; ++i) {
        if (fabs(ydot_[i]) > zero_tol) {
            old_dt = fabs(y[i]/ydot_[i]);
            if( old_dt < min_dt ) {
                min_dt = old_dt;
            }
        }
    }

    double dt = eps1 * min_dt;

    // Impose upper and lower step limits
    if( dt > step_upper_limit )
        dt = step_upper_limit;

    if ( dt < step_lower_limit )
        dt = step_lower_limit;

    return dt;
}

bool
Exhaust_system_ode::
passes_system_test(double initial_energy , std::valarray<double> &y)
{
    // incomplete
    return true;
}

int
Exhaust_system_ode::
fully_open_valves()
{
    vi_.fully_open();
```

```
    ve_.fully_open();

    return SUCCESS;
}

int
Exhaust_system_ode::
move_valves_left_cylinder(double m_g, double u, double m_g0, double p, double &dt)
{
    return move_left_valves(dt, m_g, u, p, m_g0, vi_, ve_);
}

int
Exhaust_system_ode::
move_valves_right_cylinder(double m_g, double u, double m_g0, double p, double &dt)
{
    return move_right_valves(dt, m_g, u, p, m_g0, vi_, ve_);
}

Exhaust_system::
Exhaust_system(int ndim, bool test_flag)
{
    yin_.resize(ndim);
    yout_.resize(ndim);
}

Exhaust_system::
Exhaust_system(Exhaust_system_ode *es_ode,
               std::valarray<double> yin,
               double dt_sys,
               double V,
               gas_data &Q)
{
    int ndim = yin.size();
    ode_solver_ = new OdeSolver(string("rkf_solver"), ndim, string("rkf"));
    es_ode_ = es_ode;
    es_ode_->fully_open_valves();

    yin_.resize(ndim);
    yout_.resize(ndim);
    for (size_t i = 0; i < yin.size(); ++i) {
        yin_[i] = yout_[i] = yin[i];
    }

    dt_sys_ = dt_sys;

    V_ = V;
    m0_ = Q.rho*V;
}

Exhaust_system::
Exhaust_system(const Exhaust_system &es)
{
    // incomplete
    ode_solver_ = new OdeSolver(*(es.ode_solver_));
    es_ode_ = new Exhaust_system_ode(*(es.es_ode_));
```

```
    yin_ = es.yin_;
    yout_ = es.yout_;

    dt_sys_ = es.dt_sys_;
}

Exhaust_system::
~Exhaust_system()
{
    delete ode_solver_;
    delete es_ode_;
}

int
Exhaust_system::
advance_fluid_dynamics(double &dt)
{
    global_data* gd = get_global_data_ptr();
    int status = ode_solver_->solve_over_interval(*es_ode_, gd->t, gd->t+dt, &dt_sys_,
        yin_, yout_);

    return status;
}

int
Exhaust_system::
finalise_step(gas_data &Q)
{
    Gas_model* g = get_gas_model_ptr();
    int nsp = g->get_number_of_species();

    // unpack y
    double m = yout_[0];
    double me = yout_[1];
    vector<double> my(nsp, 0.0);
    for (int i = 0; i < nsp; ++i) {
        my[i] = yout_[2+i];
    }

    double m_out = yout_[2+nsp];
    // as we approach filling a cylinder volume, switch to perfect mixing
    if (m_out/m0_ >= 0.999 && es_ode_->displace()) {
        es_ode_->set_displace(false);
    }

    Q.rho = m/V_;
    Q.e[0] = me/m;
    for (int i = 0; i < nsp; ++i) {
        Q.massf[i] = my[i]/m;
    }
    if (g->eval_thermo_state_rhoe(Q) != SUCCESS) {
        printf("error calculating eval_thermo_state_rhoe()\n");
        printf("y = [%g, %g]\n", Q.rho, Q.e[0]);
        print_gas_data(Q);
        exit(1);
    }
    // copy vector for next timestep
```

```
    yin_ = yout_;

    return SUCCESS;
}

Free_piston_engine_ode::
Free_piston_engine_ode(int ndim, bool test_flag)
    : OdeSystem(ndim, test_flag) {}

Free_piston_engine_ode::
Free_piston_engine_ode(gas_data* Q_L,
                       gas_data* Q_R,
                       double m_p,
                       double x_p,
                       double u_p,
                       double L_p,
                       double L_c,
                       double x_L,
                       double x_R,
                       double D,
                       double m_gL,
                       double m_gR,
                       double error_tol,
                       int ndim,
                       bool test_flag)
    : OdeSystem(ndim, test_flag),
      D_(D),
      m_p_(m_p),
      u_pL0_(-u_p),
      u_pR0_(u_p),
      err_tol_(error_tol)
{
    F_c_ = F_f_ = q_L_ = q_R_ = 0.0; // no losses by default
    Q_L_ = Q_L;
    Q_R_ = Q_R;
    m_gL_ = m_gL;
    m_gR_ = m_gR;
    A_ = 0.25*PI*D*D;
    L_p_ = L_p;
    L_c_ = L_c;
    x_L_ = x_L;
    x_R_ = x_R;
    Gas_model* g = get_gas_model_ptr();
    gamma_L_ = g->gamma(*Q_L);
    gamma_R_ = g->gamma(*Q_R);
    b_ = 0.005; // ring thickness per cylinder
    ydot_.resize(ndim);

    calculate_geometry(x_p);
}

Free_piston_engine_ode::
Free_piston_engine_ode(const Free_piston_engine_ode &fpe)
    : OdeSystem(fpe.ndim_, fpe.apply_system_test_)
{
    // incomplete
}
```

```
Free_piston_engine_ode::
~Free_piston_engine_ode() {}

int
Free_piston_engine_ode::
eval(const valarray<double> &y, valarray<double> &ydot)
{
    // unpack y
    double x = y[__X];
    double u = y[__U];

    // friction
    chevron_seal_friction(F_f_, Q_L_->p, Q_R_->p, b_, D_, m_p_, u);

    double T_ig = 1400.0;
    F_c_ = control_system(u, u_pL0_, m_p_, A_, L_L_, L_R_, L_c_, gamma_L_, gamma_R_, T_ig
        , Q_L_, Q_R_);

    double a = ((Q_L_->p - Q_R_->p)*A_ + F_c_ - F_f_)/m_p_;
    double de_Ldt = -Q_L_->p*A_*u/m_gL_;
    double de_Rdt =  Q_R_->p*A_*u/m_gR_;

    // heat transfer
    boundary_layer_thickness(d_L_, a_L_, x, u, u_pL0_, Q_L_->T[0], PC_T_ref, L_L_, L_c_,
        D_, gamma_L_);
    boundary_layer_thickness(d_R_, a_R_, x, u, u_pR0_, Q_R_->T[0], PC_T_ref, L_R_, L_c_,
        D_, gamma_R_);
    laminar_heat_flux(q_ddL_, Q_L_->k[0], d_L_, a_L_, Q_L_->T[0], PC_T_ref);
    laminar_heat_flux(q_ddR_, Q_R_->k[0], d_R_, a_R_, Q_R_->T[0], PC_T_ref);
    q_L_ = q_ddL_*S_L_/m_gL_;
    q_R_ = q_ddR_*S_R_/m_gR_;
    if(isnan(q_L_)) { q_L_ = 0.0; }
    if(isnan(q_R_)) { q_R_ = 0.0; }

    de_Ldt = de_Ldt - q_L_;
    de_Rdt = de_Rdt - q_R_;

    double temp[] = {u, a, de_Ldt, de_Rdt, q_L_*m_gL_, q_R_*m_gR_, F_f_*u, F_c_*u};
    for (size_t i = 0; i < ydot.size(); ++i) {
        ydot[i] = temp[i];
    }

    return SUCCESS;
}

double
Free_piston_engine_ode::
stepsize_select(const valarray<double> &y)
{
    eval(y, ydot_);
    double min_dt = step_upper_limit; // to get us started
    double old_dt = 0.0;

    for (int i = 0; i < ndim_; ++i) {
        if (fabs(ydot_[i]) > zero_tol) {
            old_dt = fabs(y[i]/ydot_[i]);
```

```
            if( old_dt < min_dt ) {
                min_dt = old_dt;
            }
        }
    }

    double dt = eps1*min_dt;

    // Impose upper and lower step limits
    if (dt > step_upper_limit)
        dt = step_upper_limit;

    if (dt < step_lower_limit)
        dt = step_lower_limit;

    return dt;
}

double
Free_piston_engine_ode::
calculate_system_energy(double u_p, double V_L, double V_R)
{
    double ke = 0.5*m_p_*u_p*u_p;

    Gas_model* g = get_gas_model_ptr();
    double e_L = g->total_internal_energy(*Q_L_);
    double e_R = g->total_internal_energy(*Q_R_);

    return ke + V_L*Q_L_->rho*e_L + V_R*Q_R_->rho*e_R;
}

bool
Free_piston_engine_ode::
passes_system_test(double initial_energy, double V_L, double V_R, valarray<double> &y)
{
    double u = y[__U];
    double current_energy = calculate_system_energy(u, V_L, V_R);
    double err = (fabs(current_energy) - fabs(initial_energy))/fabs(initial_energy);

    if (err < err_tol_)
        return true;
    else
        return false;
}

int
Free_piston_engine_ode::
calculate_geometry(double x_p)
{
    // cylinder length, volume, surface area
    L_L_ = (x_p - L_p_/2.0) - x_L_;
    L_R_ = x_R_ - (x_p + L_p_/2.0);
    V_L_ = L_L_*A_;
    V_R_ = L_R_*A_;
    S_L_ = 2*A_ + PI*D_*L_L_;
    S_R_ = 2*A_ + PI*D_*L_R_;
```

```
    return SUCCESS;
}

Free_piston_engine::
Free_piston_engine(int ndim_fpe, int ndim_es, bool test_flag, double dt_sys, double
    dt_therm)
{
    // incomplete
}

Free_piston_engine::
Free_piston_engine(Free_piston_engine_ode* fpe_ode,
                   Exhaust_system_ode* es_ode_L,
                   Exhaust_system_ode* es_ode_R,
                   valarray<double> y_fpe,
                   valarray<double> y_es,
                   double dt_sys,
                   double dt_therm,
                   double m_gL0,
                   double m_gR0)
{
    int ndim_fpe = y_fpe.size();
    ode_solver_fpe_ = new OdeSolver(string("rkf_solver"), ndim_fpe, string("rkf"));

    int ndim_es = y_es.size();
    ode_solver_es_ = new OdeSolver(string("rkf_solver"), ndim_es, string("rkf"));

    fpe_ode_ = fpe_ode;
    es_ode_L_ = es_ode_L;
    es_ode_R_ = es_ode_R;
    yin_.resize(ndim_fpe);
    yout_.resize(ndim_fpe);
    for (int i = 0; i < ndim_fpe; ++i) {
        yin_[i] = yout_[i] = y_fpe[i];
    }
    m_g0_  = m_gR0; // assume the right hand cylinder is the correct mass
    m_gL0_ = m_gL0;
    m_gR0_ = m_gR0;
    m_gLout_ = 0.0;
    m_gRout_ = 0.0;
    yin_L_.resize(ndim_es);
    yout_L_.resize(ndim_es);
    for (int i = 0; i < ndim_es; ++i) {
        yin_L_[i] = yout_L_[i] = y_es[i]*fpe_ode_->get_V_L(); //extensive properties
    }
    yin_R_.resize(ndim_es);
    yout_R_.resize(ndim_es);
    for (int i = 0; i < ndim_es; ++i) {
        yin_R_[i] = yout_R_[i] = y_es[i]*fpe_ode_->get_V_R(); //extensive properties
    }
    dt_sys_ = dt_sys;
    dt_therm_ = dt_therm;
    dt_chem_ = -1.0;
    initial_energy_ = fpe_ode_->calculate_system_energy(y_fpe[__U], fpe_ode_->get_V_L(),
        fpe_ode_->get_V_R());
    system_energy_ = initial_energy_;
}
```

```
Free_piston_engine::
Free_piston_engine(const Free_piston_engine &fpe)
{
    ode_solver_fpe_ = new OdeSolver(*(fpe.ode_solver_fpe_));
    ode_solver_es_ = new OdeSolver(*(fpe.ode_solver_es_));
    fpe_ode_ = new Free_piston_engine_ode(*(fpe.fpe_ode_));
    es_ode_L_ = new Exhaust_system_ode(*(fpe.es_ode_L_));
    es_ode_R_ = new Exhaust_system_ode(*(fpe.es_ode_R_));

    yin_ = fpe.yin_;
    yout_ = fpe.yout_;
    yin_L_ = fpe.yin_L_;
    yout_L_ = fpe.yout_L_;
    yin_R_ = fpe.yin_R_;
    yout_R_ = fpe.yout_R_;
    m_gL0_ = fpe.m_gL0_;
    m_gR0_ = fpe.m_gR0_;
    dt_sys_ = fpe.dt_sys_;
    dt_therm_ = fpe.dt_therm_;
    dt_chem_ = fpe.dt_chem_;
}

Free_piston_engine::
~Free_piston_engine()
{
    delete ode_solver_fpe_;
    delete ode_solver_es_;
    delete es_ode_L_;
    delete es_ode_R_;
    delete fpe_ode_;
}

int
Free_piston_engine::
instantaneous_heat_addition_left_cylinder(double dh)
{
    yout_[__E_L] += dh;

    return SUCCESS;
}

int
Free_piston_engine::
instantaneous_heat_addition_right_cylinder(double dh)
{
    yout_[__E_R] += dh;

    return SUCCESS;
}

int
Free_piston_engine::
reset_intake_mass_left_cylinder(gas_data &Q)
{
    m_gL0_ = fpe_ode_->get_V_L()*Q.rho;
    fpe_ode_->set_m_gL(m_gL0_);
```

```
    es_ode_L_->set_displace(true);

    // reset friction and heat loss
    yout_[__Q_L] = 0.0;
    yout_[__Q_R] = 0.0;
    yout_[__F] = 0.0;
    yout_[__W] = 0.0;

    // reset exhausted mass
    Gas_model* g = get_gas_model_ptr();
    int nsp = g->get_number_of_species();
    m_gLout_ = yout_L_[2+nsp];
    yout_L_[2+nsp] = 0;

    return SUCCESS;
}

int
Free_piston_engine::
reset_intake_mass_right_cylinder(gas_data &Q)
{
    m_gR0_ = fpe_ode_->get_V_R()*Q.rho;
    fpe_ode_->set_m_gR(m_gR0_);
    es_ode_R_->set_displace(true);

    // reset friction and heat loss
    yout_[__Q_L] = 0.0;
    yout_[__Q_R] = 0.0;
    yout_[__F] = 0.0;
    yout_[__W] = 0.0;

    // reset exhausted mass
    Gas_model* g = get_gas_model_ptr();
    int nsp = g->get_number_of_species();
    m_gRout_ = yout_R_[2+nsp];
    yout_R_[2+nsp] = 0;

    return SUCCESS;
}

int
Free_piston_engine::
move_left_valves(double u_p, double pL, double &dt)
{
    return es_ode_L_->move_valves_left_cylinder(fpe_ode_->get_m_gL(), u_p, 0.5*(m_gL0_ +
        m_g0_), pL, dt);
}

int
Free_piston_engine::
move_right_valves(double u_p, double pR, double &dt)
{
    return es_ode_R_->move_valves_right_cylinder(fpe_ode_->get_m_gR(), u_p, 0.5*(m_gR0_ +
         m_g0_), pR, dt);
}

int
```

```
Free_piston_engine::
advance_chemistry_left_cylinder(gas_data &Q, double &dt)
{
    int status = perform_chemical_increment(Q, dt, dt_therm_, dt_chem_);
    yout_[__E_L] = Q.e[0];

    // copy mass fractions back to state vectors
    Gas_model* g = get_gas_model_ptr();
    int nsp = g->get_number_of_species();
    double m_gL = fpe_ode_->get_m_gL();
    for (int i = 0; i < nsp; ++i) {
        yin_L_[2+i] = m_gL*Q.massf[i];
    }

    return status;
}

int
Free_piston_engine::
advance_chemistry_right_cylinder(gas_data &Q, double &dt)
{
    int status = perform_chemical_increment(Q, dt, dt_therm_, dt_chem_);
    yout_[__E_R] = Q.e[0];

    // copy mass fractions back to state vectors
    Gas_model* g = get_gas_model_ptr();
    int nsp = g->get_number_of_species();
    double m_gR = fpe_ode_->get_m_gR();
    for (int i = 0; i < nsp; ++i) {
        yin_R_[2+i] = m_gR*Q.massf[i];
    }

    return status;
}

int
Free_piston_engine::
advance_dynamics(gas_data &Q_L, gas_data &Q_R, double &dt)
{
    global_data* gd = get_global_data_ptr();
    ode_solver_fpe_->solve_over_interval(*fpe_ode_, gd->t, gd->t+dt, &dt_sys_, yin_,
        yout_);

    return SUCCESS;
}

int
Free_piston_engine::
finalise_step(gas_data &Q_L, gas_data &Q_R)
{
    Gas_model* g = get_gas_model_ptr();
    int nsp = g->get_number_of_species();

    // unpack state
    double x = yout_[__X];
    double u = yout_[__U];
    double e_L = yout_[__E_L];
```

```
double e_R = yout_[__E_R];
double m_gL = fpe_ode_->get_m_gL();
double m_gR = fpe_ode_->get_m_gR();
vector<double> my_L(nsp, 0.0);
for (int i = 0; i < nsp; ++i) {
    my_L[i] = yout_L_[2+i];
}
double m_outL = yout_L_[2+nsp];

// as we approach filling a cylinder volume, switch to perfect mixing
if (m_outL/m_gL0_ >= 0.999 && es_ode_L_->displace()) {
    es_ode_L_->set_displace(false);
}

vector<double> my_R(nsp, 0.0);
for (int i = 0; i < nsp; ++i) {
    my_R[i] = yout_R_[2+i];
}
double m_outR = yout_R_[2+nsp];

// as we approach filling a cylinder volume, switch to perfect mixing
if (m_outR/m_gR0_ >= 0.999 && es_ode_R_->displace()) {
    es_ode_R_->set_displace(false);
}

double dm_L = yout_L_[__M] - yin_L_[__M];
double dme_L = yout_L_[__ME] - yin_L_[__ME];
double dm_R = yout_R_[__M] - yin_R_[__M];
double dme_R = yout_R_[__ME] - yin_R_[__ME];

fpe_ode_->calculate_geometry(x);

// this is where we calculate the actual gas states from the complete second law
Q_L.rho = (m_gL + dm_L)/fpe_ode_->get_V_L();
Q_L.e[0] = (dme_L + e_L*m_gL)/(m_gL + dm_L);

for (int i = 0; i < nsp; ++i) {
    Q_L.massf[i] = my_L[i]/(m_gL + dm_L);
}
fpe_ode_->set_m_gL(m_gL + dm_L);

if (g->eval_thermo_state_rhoe(Q_L) != SUCCESS) {
    printf("error calculating eval_thermo_state_rhoe()\n");
    printf("y = [%g, %g, %g, %g]\n", x, u, e_L, e_R);
    print_gas_data(Q_L);
    exit(1);
}
if (g->eval_transport_coefficients(Q_L) != SUCCESS) {
    printf("error calculating eval_transport_coefficients()\n");
    print_gas_data(Q_L);
    exit(1);
}

Q_R.rho = (m_gR + dm_R)/fpe_ode_->get_V_R();
Q_R.e[0] = (dme_R + e_R*m_gR)/(m_gR + dm_R);

for (int i = 0; i < nsp; ++i) {
```

```
        Q_R.massf[i] = my_R[i]/(m_gR + dm_R);
    }
    fpe_ode_->set_m_gR(m_gR + dm_R);

    if (g->eval_thermo_state_rhoe(Q_R) != SUCCESS) {
        printf("error calculating eval_thermo_state_rhoe()\n");
        printf("y = [%g, %g, %g, %g]\n", x, u, e_L, e_R);
        print_gas_data(Q_R);
        exit(1);
    }
    if (g->eval_transport_coefficients(Q_R) != SUCCESS) {
        printf("error calculating eval_transport_coefficients()\n");
        print_gas_data(Q_R);
        exit(1);
    }

    // copy energy and work back to state vectors
    yout_[__E_L] = Q_L.e[0];
    yout_[__E_R] = Q_R.e[0];
    system_energy_ = fpe_ode_->calculate_system_energy(u, fpe_ode_->get_V_L(), fpe_ode_->
        get_V_R());
    fpe_ode_->set_gamma_L(g->gamma(Q_L));
    fpe_ode_->set_gamma_R(g->gamma(Q_R));

    // copy vectors for next timestep
    yin_ = yout_;
    yin_L_ = yout_L_;
    yin_R_ = yout_R_;

    return SUCCESS;
}

int
Free_piston_engine::
advance_fluid_dynamics_left_cylinder(double u_p, gas_data *Q, double &dt)
{
    global_data* gd = get_global_data_ptr();
    move_left_valves(u_p, Q->p, dt);
    ode_solver_es_->solve_over_interval(*es_ode_L_, gd->t, gd->t+dt, &dt_sys_, yin_L_,
        yout_L_);

    return SUCCESS;
}

int
Free_piston_engine::
advance_fluid_dynamics_right_cylinder(double u_p, gas_data *Q, double &dt)
{
    global_data* gd = get_global_data_ptr();
    move_right_valves(u_p, Q->p, dt);
    ode_solver_es_->solve_over_interval(*es_ode_R_, gd->t, gd->t+dt, &dt_sys_, yin_R_,
        yout_R_);

    return SUCCESS;
}

int
```

```
perform_chemical_increment(gas_data &Q, double &dt, double &dt_therm, double &dt_chem)
{
    Gas_model* g = get_gas_model_ptr();
    Reaction_update* r = get_reaction_update_ptr();

    // no chemisty file has been set!
    if (r == 0) return SUCCESS;

    if (dt_therm >= dt) {
        if (r->update_state(Q, dt, dt_chem) != SUCCESS) {
            printf("error updating chemical step\n");
            exit(1);
        }
            if (g->eval_thermo_state_rhoe(Q) != SUCCESS) {
                printf("error calculating eval_thermo_state_rhoe()\n");
                print_gas_data(Q);
                exit(1);
        }
    } else {
        // step through interval
        double substep = 0.0;
        for (; substep < (dt - dt_therm); substep += dt_therm) {
            if (r->update_state(Q, dt_therm, dt_chem) != SUCCESS) {
                // the chemistry update has failed
                // reduce dt_therm and try again
                dt_therm = dt_therm*DT_THERM_REDUCE;
                if (r->update_state(Q, dt_therm, dt_chem) != SUCCESS) {
                    printf("error updating chemical step\n");
                    printf("dt_sys = %g\n", dt);
                    printf("dt_therm = %g\n", dt_therm);
                    printf("dt_chem = %g\n", dt_chem);
                    exit(1);
                }
            }
            if (g->eval_thermo_state_rhoe(Q) != SUCCESS) {
                printf("error calculating eval_thermo_state_rhoe()\n");
                print_gas_data(Q);
                exit(1);
            }
        }

        // take final step
        double dtf = dt - substep;
        if (r->update_state(Q, dtf, dtf) != SUCCESS) {
            printf("error updating chemical step during last substep\n");
            printf("dt_sys = %g\n", dt);
            printf("dt_therm = %g\n", dt_therm);
            printf("dt_chem = %g\n", dt_chem);
            exit(1);
        }
        if (g->eval_thermo_state_rhoe(Q) != SUCCESS) {
            printf("error calculating eval_thermo_state_rhoe()\n");
            print_gas_data(Q);
            exit(1);
        }
    }
    return SUCCESS;
```

```
}

int copy_array(valarray<double> &src, valarray<double> &dst) {
    // for the sake of speed, we are making NO error checks here
    // and assuming the user knows what they're doing.
    for (size_t i = 0; i < dst.size(); ++i) {
        dst[i] = src[i];
    }
    return SUCCESS;
}
```

Listing B.6: *Free-piston engine system, source file.*

B.4 Free-piston engine models

```
#ifndef FPE_MODELS_HH
#define FPE_MODELS_HH

// \author Brendan T. O'Flaherty
// \brief  C-style implementation of some useful models

#include <vector>
#include <string>
#include "../../lib/util/source/useful.h"
#include "../../lib/gas/models/physical_constants.hh"
#include "../../lib/gas/models/gas-model.hh"

#define GRAVITY 0
#define PI 3.14159265359 // restate my assumptions: mathematics is the language of nature
#define T_ATM 298.15
#define P_ATM 101325.0

// heat flux models
int aichlmayr_heat_flux(double &q,
                        double k,
                        double D,
                        double L,
                        double T,
                        double Tw=T_ATM);
int annand_heat_flux(double &q,
                     double k,
                     double D,
                     double u_p,
                     double mu,
                     double rho,
                     double T,
                     double Tw=T_ATM);
int annand_and_pinfold_heat_flux(double &q,
                                 double k,
                                 double D,
                                 double u_p,
                                 double u_p0,
                                 double mu,
                                 double T,
                                 double rho,
                                 double Tw,
                                 double dTdt);
int lawton_heat_flux(double &q,
                     double k,
                     double D,
                     double u_p,
                     double u_p0,
                     double mu,
                     double T,
                     double rho,
                     double Tw,
                     double L,
                     double gamma,
                     double alpha0=2.2160e-5);
int woschni_heat_flux(double &q,
```

```
                        double k,
                        double D, double u_p,
                        double p,
                        double T,
                        double Tw=T_ATM,
                        std::string name="stroke");
int boundary_layer_thickness(double &d,
                             double &a,
                             double x_p,
                             double u_p,
                             double u_p0,
                             double T,
                             double Tw,
                             double L,
                             double L_c,
                             double D,
                             double gamma,
                             double alpha0=2.2160e-5);
int laminar_heat_flux(double &q,
                      double k,
                      double d,
                      double u_p,
                      double T,
                      double Tw);

// friction coefficient models
int constant_friction_coefficient(double &f);
int mcgeehan_friction_coefficient(double &f,
                                  double u_p,
                                  double eta,
                                  double p_r,
                                  double b);
int mixed_friction_coefficient(double &f,
                               double u_p,
                               double eta,
                               double p_r,
                               double b);
int chevron_seal_friction(double &F,
                          double p0,
                          double p1,
                          double b,
                          double D,
                          double m_p,
                          double u_p);

// fluid dynamics
int discharge_coefficient(double &cd,
                          double Lv,
                          double Dv,
                          std::string name);

// control system
double control_system(double u_p,
                      double u_p0,
                      double m_p,
                      double A,
                      double L_L,
```

```
                        double L_R,
                        double L_c,
                        double gamma_L,
                        double gamma_R,
                        double T_ig,
                        gas_data *Q_L,
                        gas_data *Q_R);

// engine geometry
int get_geometry(std::vector<double> &geom,
                 double x,
                 double D,
                 double L_c,
                 double x_L,
                 double x_R);

#endif
```

Listing B.7: *Free-piston engine models, header file.*

```
#include <cstdio>
#include <cstdlib>
#include <cmath>
#include <vector>
#include "../../lib/nm/source/linear_interpolation.hh"
#include "../../lib/gas/models/gas_data.hh"
#include "fpe_models.hh"
#include "fpe_kernel.hh"

using namespace std;

// heat transfer
int
aichlmayr_heat_flux(double &q,
                    double k,
                    double D,
                    double L,
                    double T,
                    double Tw)
{
    // quasi-steady heat transfer of Aichlmayr et al (2002)
    // Bessel function of the second kind for a finite cylinder

    double LoD = L/D;
    double num = 0.440332*PI*PI + 5.09296*LoD*LoD;
    double den = PI + 2*PI*LoD;
    q = (2e3*k*(T - Tw))/L*(num/den);

    return SUCCESS;
}

int
annand_heat_flux(double &q,
                 double k,
                 double D,
                 double u_p0,
                 double mu,
```

```
                  double rho,
                  double T,
                  double Tw)
{
    // quasi-steady heat transfer of Annand (1963)
    // Nu-Re relation for a 460 381mm two-stroke engine at steady state
    // Nu = a*Re^0.7

    double u_m = fabs(u_p0/sqrt(2)); // mean speed
    double a = 0.0363; //0.12 Nu-Re coefficient
    double Re = rho*u_m*D/mu;
    double h = a*(k/D)*pow(Re,0.7);
    q = h*(T - Tw);

    return SUCCESS;
}

int
annand_and_pinfold_heat_flux(double &q,
                             double k,
                             double D,
                             double u_p,
                             double u_p0,
                             double mu,
                             double T,
                             double rho,
                             double Tw,
                             double dTdt)
{
    double u_m = fabs(u_p0/sqrt(2)); // mean speed
    double a = 0.3; // Nu-Re coefficient
    double Re = rho*u_m*D/mu;
    double h = a*(k/D)*pow(Re,0.7);

    q = h*(T - Tw)*(1.0 + 0.27*D*dTdt/((T-Tw)*u_m));

    return SUCCESS;
}

int
lawton_heat_flux(double &q,
                 double k,
                 double D,
                 double u_p,
                 double u_p0,
                 double mu,
                 double T,
                 double rho,
                 double Tw,
                 double L,
                 double gamma,
                 double alpha0)
{
    // unsteady heat transfer of Lawton (1987)
    // based on Annand's quasi-steady form

    double u_m = fabs(u_p0/sqrt(2)); // mean speed
```

```
    double Re = rho*u_m*D/mu; // Reynolds
    double t0 = sqrt(pow(D,3.0)/(alpha0*u_m)); // time constant
    double c = (gamma - 1.0)*t0*(u_p/L); // compressibility number
    double a = 0.28;
    q = (k/D)*(a*pow(Re,0.7)*(T - Tw) + 2.75*c*Tw);

    return SUCCESS;
}

int
woschni_heat_flux(double &q,
                  double k,
                  double D,
                  double u_p,
                  double p,
                  double T,
                  double Tw,
                  string name)
{
    // quasi-steady heat transfer of Woschni (1967)
    // Nu-Re relation for a motored engine at steady state

    double c1;
    if (name == "exhaust") {
        c1 = 6.18;
    } else if (name == "stroke") {
        c1 = 2.28;
    } else {
        printf("unknown name: %s", name.c_str());
        printf("options are 'exhaust' or 'stroke'\n");
        exit(BAD_INPUT_ERROR);
    }
    p = p/(P_ATM*1.033); // convert Pa to kp/cm^2
    double uref = c1*u_p;
    double h = 110.0*pow(p,0.8)*pow(uref,0.8)/(pow(T,0.53)*pow(D,0.2));
    q = h*(T - Tw);

    return SUCCESS;
}

int
boundary_layer_thickness(double &d,
                         double &a,
                         double x_p,
                         double u_p,
                         double u_p0,
                         double T,
                         double Tw,
                         double L,
                         double L_c,
                         double D,
                         double gamma,
                         double alpha0)
{
    a = -0.8*(u_p/u_p0);
    double dd = (alpha0/fabs(u_p))*(1.0 - Tw/T)*(1 - a)/(1 + a)*pow(L, (gamma + 1))/(
        gamma*pow(L_c,gamma) - L_c * pow(L, (gamma - 1)));
```

```
    if (T < Tw) d = sqrt(-dd); // gradient flips
    else d = sqrt(dd);
    if (d > D/2.0) { d = D/2.0; } // fully developed

    return SUCCESS;
}

int
laminar_heat_flux(double &q,
                  double k,
                  double d,
                  double a,
                  double T,
                  double Tw)
{
    q = 0.0;
    if (d == 0.0) return SUCCESS;
    q = (1.6*k/d)*(0.5 - a)*(T - Tw);

    return SUCCESS;
}

// friction
int
constant_friction_coefficient(double &f)
{
    f = 0.1;
    return SUCCESS;
}

int
mcgeehan_friction_coefficient(double &f,
                              double u_p,
                              double eta,
                              double p_r,
                              double b)
{
    // friction coefficient of McGeehan (1979)
    // parabolic ring profile
    double c_1 = 4.8;
    double c_2 = 0.5;
    double f0 = eta*fabs(u_p)/(p_r*b);
    f = c_1*pow(f0, c_2);

    return SUCCESS;
}

int
mixed_friction_coefficient(double &f,
                           double u_p,
                           double eta,
                           double p_r,
                           double b)
{
    double f_s, f_h;
    double nu;
```

```
    double alpha;

    constant_friction_coefficient(f_s);
    mcgeehan_friction_coefficient(f_h, fabs(u_p), eta, P_ATM, b);

    nu = pow((eta*fabs(u_p)/(P_ATM*b)),0.5);
    alpha = exp(-300*nu);
    if (alpha > 1.0) alpha = 1.0;
    if (alpha < 0.0) alpha = 0.0;

    f = f_s*alpha + (1.0 - alpha)*f_h;

    return SUCCESS;
}

int
chevron_seal_friction(double &F,
                      double p0,
                      double p1,
                      double b,
                      double D,
                      double m_p,
                      double u_p)
{
    // friction calculation
    double f;
    double eta = 0.222; // kinematic viscosity of SAE30 at 300K = 200cSt (approx),
        density = 900kg/m^3
    double sigma = 0.0; // ring tensile stress
    double p_r0 = p0 + sigma;
    double p_r1 = p1 + sigma;
    double A_r = PI*D*b;

    F = 0.0;

    if (fabs(u_p) < 1.0e-6) {
        return SUCCESS;
    } else {
        mixed_friction_coefficient(f, u_p, eta, p_r0, b);
        F += f*(m_p*GRAVITY/2 + p_r0*A_r);
        mixed_friction_coefficient(f, u_p, eta, p_r1, b);
        F += f*(m_p*GRAVITY/2 + p_r1*A_r);
    }
    F = F*u_p/fabs(u_p);

    return SUCCESS;
}

int
discharge_coefficient(double &cd,
                      double Lv,
                      double Dv,
                      string name)
{
    int status;
    double LoD = Lv/Dv;
    double xarray[] = {0.025, 0.050, 0.075, 0.100, 0.150, 0.200, 0.240};
```

```
    vector<double> x(xarray, xarray+7);

    string intake("intake");
    string exhaust("exhaust");

    if (name == intake) {
        // use intake coefficients
        double yarray[] = {0.949, 0.887, 0.769, 0.699, 0.648, 0.619, 0.638};
        vector<double> y(yarray, yarray+7);
        status = linear_eval(LoD, cd, x, y);

    } else if (name == exhaust) {
        // use exhaust coefficients
        double yarray[] = {0.869, 0.808, 0.679, 0.579, 0.508, 0.509, 0.558};
        vector<double> y(yarray, yarray+7);
        status = linear_eval(LoD, cd, x, y);
    } else {
        printf("unknown name: %s", name.c_str());
        exit(BAD_INPUT_ERROR);
    }

    if (status != SUCCESS) {
        printf("discharge_coefficient failed\n");
        printf("trying to get Cd for %s\n", name.c_str());
        printf("exiting to system...\n");
        exit(VALUE_ERROR);
    }
    return SUCCESS;
}

double
control_system(double u_p,
               double u_p0, // target velocity
               double m_p,
               double A,
               double L_L,
               double L_R,
               double L_c,   // cylinder length
               double gamma_L, // LHS gamma
               double gamma_R, // RHS gamma
               double T_ig,  // target temperature
               gas_data *Q_L,
               gas_data *Q_R)
{
    double L_ex, dx, a, b;
    double F = 0.0;

    if (u_p >= 0) {
        if (L_R > L_c && L_L > 0.5*L_c){
            if (Q_L->p > P_ATM) {
                a = gamma_L + 1.0;
                L_ex = L_L*pow((P_ATM/Q_L->p), (1.0/gamma_L));
                b = P_ATM*A/(a*pow(L_ex, gamma_L));
                dx = L_L - L_ex;
                F = (0.5*m_p*(u_p0*u_p0 - u_p*u_p) + b*(pow(L_ex, a) - pow(L_L, a)))/dx;
            }
        }
```

```
    } else {
        if (L_L > L_c && L_R > 0.5*L_c) {
            if (Q_R->p > P_ATM) {
                a = gamma_R + 1.0;
                L_ex = L_R*pow((P_ATM/Q_R->p), (1.0/gamma_R));
                b = P_ATM*A/(a*pow(L_ex, gamma_R));
                dx = - L_R + L_ex;
                F = (0.5*m_p*(u_p0*u_p0 - u_p*u_p) + b*(pow(L_ex, a) - pow(L_R, a)))/dx;
            }
        }
    }

    // what is the force limit?
    if (F > 1e4) { F = 1e4; }
    if (F < -1e4) { F = -1e4; }

    return F;
}

// free-piston engine geometry
int
get_geometry(vector<double> &geom,
             double x,
             double D,
             double L_c,
             double x_L,
             double x_R)
{
    if (geom.size() < 11) {
        printf("geom vector size %i < 11\n", (int)geom.size());
        printf("exiting to system...\n");
        exit(1);
    }
    double A, xp_L, xp_R, L_i, L_p, L_L, L_R, V_L, V_R, S_L, S_R;

    // length of intake ports
    L_i = 0.5*D;

    // required piston length
    L_p = L_c + 2*L_i;

    // bore area
    A = PI*D*D/4.0;

    // port opening point
    xp_L = x_L + L_c + L_p/2.0;
    xp_R = x_R - L_c - L_p/2.0;

    // instantaneous cylinder length
    L_L = (x - L_p/2.0) - x_L;
    L_R = x_R - (x + L_p/2.0);

    // cylinder volume
    V_L = L_L*A;
    V_R = L_R*A;

    // cylinder surface area
```

```
    S_L = 2*A + PI*D*L_L;
    S_R = 2*A + PI*D*L_R;

    double geom_array[] = {A, xp_L, xp_R, L_i, L_p, L_L, L_R, V_L, V_R, S_L, S_R};
    for (size_t i = 0; i < geom.size(); ++i) {
        geom[i] = geom_array[i];
    }
    return SUCCESS;
}
```

Listing B.8: *Free-piston engine models, source file.*

B.5 Free-piston engine control

```
#ifndef FPE_CONTROL_HH
#define FPE_CONTROL_HH

// \author Brendan T. O'Flaherty
// \brief  Free-piston engine control

#include <string>
#include <vector>
#include "../../lib/util/source/useful.h"
#include "../../lib/nm/source/linear_interpolation.hh"
#include "fpe_models.hh"

class Valve
{
public:
    Valve();
    Valve(std::string type, double v, double D);
    ~Valve();

    double get_area() { return A_; }

    void fully_open() { lift_ = 0.25*D_; A_ = 0.25*PI*D_*D_; }

    int open(double dt);
    int close(double dt);
    double eval_discharge_coefficient();

    double get_t() { return tlift_; }
    double get_vel() { return v_; }
    double get_lift() { return lift_; }

private:
    std::string type_; // either "intake" or "exhaust"

    double lift_; // lift
    double l_max_; // max lift
    double D_; // diamter
    double cd_; // discharge coefficient

    double tlift_; // opening time
    double v_; // speed
    double A_; // current area
};

int move_right_valves(double dt,
                      double m_g,
                      double u_p,
                      double p,
                      double m_g0,
                      Valve &vi,
                      Valve &ve);
int move_left_valves(double dt,
                     double m_g,
                     double u_p,
                     double p,
```

```
                    double m_g0,
                    Valve &vi,
                    Valve &ve);

#endif
```

Listing B.9: *Free-piston engine controls, header file.*

```
// \author Brendan T. O'Flaherty
// \brief  Free-piston engine control

#include <cstdio>
#include "../../lib/gas/models/physical_constants.hh"
#include "fpe_control.hh"
#include "fpe_kernel.hh"

using namespace std;

Valve::
Valve() {}

Valve::
Valve(string type, double v, double D)
    : type_(type), lift_(0.0), l_max_(0.25*D), D_(D), cd_(0.0), tlift_(l_max_/v), v_(v),
      A_(0.0)
{}

Valve::
~Valve() {}

int
Valve::
open(double dt)
{
    lift_ += v_*dt;
    if (lift_ > l_max_) lift_ = l_max_;
    A_ = PI*D_*lift_;

    return SUCCESS;
}

int
Valve::
close(double dt)
{
    lift_ -= v_*dt;
    if (lift_ < 0.0) lift_ = 0.0;
    A_ = PI*D_*lift_;

    return SUCCESS;
}

double
Valve::
eval_discharge_coefficient()
{
    discharge_coefficient(cd_, lift_, D_, type_);
```

```
    return cd_;
}

int
move_right_valves(double dt, double m_g, double u_p, double p, double m_g0, Valve &vi,
    Valve &ve)
{
    //  L_R is the current right cylinder length
    //  L_cyl is the geometric cylinder length
    if (u_p < 0) {
        if (p < PC_P_atm) {
            vi.open(dt);
        }
    } else {
        if (p > PC_P_atm) {
            vi.close(dt);
            if (m_g > m_g0) {
                ve.open(dt);
            } else {
                ve.close(dt);
            }
        }
    }

    return SUCCESS;
}

int
move_left_valves(double dt, double m_g, double u_p, double p, double m_g0, Valve &vi,
    Valve &ve)
{
    //  L_L is the current right cylinder length
    //  L_cyl is the geometric cylinder length
    if (u_p > 0) {
        if (p < PC_P_atm) {
            vi.open(dt);
        }
    } else {
        if (p > PC_P_atm) {
            vi.close(dt);
            if (m_g > m_g0) {
                ve.open(dt);
            } else {
                ve.close(dt);
            }
        }
    }

    return SUCCESS;
}
```

Listing B.10: *Free-piston engine controls, source file.*

B.6 Sod's shock tube simulator

```
#ifndef SOD_HH
#define SOD_HH

// \author Brendan T. O'Flaherty
// \brief  Functions used to solve Sod's shock tube problem

#include <vector>
#include "../../lib/util/source/useful.h"
#include "../../lib/util/source/dbc_assert.hh"
#include "../../lib/gas/models/gas_data.hh"
#include "../../lib/gas/models/gas-model.hh"
#include "../../lib/nm/source/zero_system.hh"
#include "../../lib/nm/source/zero_finders.hh"

class p2p1_fun : public ZeroFunction {
public:
    p2p1_fun();
    p2p1_fun(gas_data Q1, gas_data Q2, gas_data Q3, gas_data Q4);
    p2p1_fun(const p2p1_fun& zfun);
    ~p2p1_fun();

    int eval(double p2p1, double &y);

private:
    double T2_;
    gas_data Q_1_, Q_2_, Q_3_, Q_4_;
};

class r1r2_fun : public ZeroFunction {
public:
    r1r2_fun();
    r1r2_fun(gas_data Q1, gas_data Q2, gas_data Q3, gas_data Q4, int nx);
    r1r2_fun(const r1r2_fun& zfun);
    ~r1r2_fun();

    int eval(double r1r2, double &y);

private:
    int nx_;
    gas_data Q_1_, Q_2_, Q_3_, Q_4_;
};

int step_across_expansion_dp(gas_data &Q3, double &u3, double dp);
int step_across_expansion_du(gas_data &Q3, double &u3, double du);
int get_state_behind_shock(gas_data &Q2, gas_data Q1, double r1r2);
int get_shock_speed(double &W, double r1r2, gas_data Q1, gas_data Q2);
double get_velocity_behind_shock(double r1r2, double W);
int eval_flux(std::vector<double> &flux, gas_data *Q4, gas_data *Q1, double tol=1e-6);
std::vector<double>
python_eval_flux(gas_data *Q4, gas_data *Q1, double tol=1e-6);

#endif
```

Listing B.11: *Sod's shock tube simulator, header file.*

```
#include <cstdio>
#include <cmath>
#include <vector>
#include "fpe_kernel.hh"
#include "sod.hh"

using namespace std;

// p2p1 solver
p2p1_fun::
p2p1_fun()
    : ZeroFunction() {}

p2p1_fun::
p2p1_fun(gas_data Q1, gas_data Q2, gas_data Q3, gas_data Q4)
    : ZeroFunction()
{
    Gas_model *g = get_gas_model_ptr();

    g->initialise_gas_data(Q_1_);
    g->initialise_gas_data(Q_2_);
    g->initialise_gas_data(Q_3_);
    g->initialise_gas_data(Q_4_);

    copy_gas_data(Q1, Q_1_);
    copy_gas_data(Q2, Q_2_);
    copy_gas_data(Q3, Q_3_);
    copy_gas_data(Q4, Q_4_);
}

p2p1_fun::
p2p1_fun(const p2p1_fun& zfun)
    : ZeroFunction()
{
    Gas_model *g = get_gas_model_ptr();

    g->initialise_gas_data(Q_1_);
    g->initialise_gas_data(Q_2_);
    g->initialise_gas_data(Q_3_);
    g->initialise_gas_data(Q_4_);

    copy_gas_data(zfun.Q_1_, Q_1_);
    copy_gas_data(zfun.Q_2_, Q_2_);
    copy_gas_data(zfun.Q_3_, Q_3_);
    copy_gas_data(zfun.Q_4_, Q_4_);

}

p2p1_fun::~p2p1_fun() {}

int
p2p1_fun::
eval(double p2p1,
     double &y)
{
    Gas_model *g = get_gas_model_ptr();
```

```
    // assumes ideal equation of state
    g->eval_sound_speed(Q_1_);
    g->eval_sound_speed(Q_4_);

    ASSERT(!isnan(Q_1_.a) && !isinf(Q_1_.a));
    ASSERT(!isnan(Q_4_.a) && !isinf(Q_4_.a));

    T2_ = (g->gamma(Q_4_) - 1.0)*(Q_1_.a/Q_4_.a)*(p2p1 - 1.0)/
        sqrt(2.0*g->gamma(Q_1_)*(2.0*g->gamma(Q_1_) + (g->gamma(Q_1_) + 1.0)*(p2p1 - 1.0)
            ));
    if (T2_ > 1.0) { T2_ = 1.0; } // prevent error for very small downstream pressures
    y = ((Q_4_.p/Q_1_.p) - p2p1 * pow((1.0 - T2_), -2.0*g->gamma(Q_4_)/(g->gamma(Q_4_) -
        1.0)));

    return SUCCESS;
}

// r1r2 solver
r1r2_fun::
r1r2_fun()
    : ZeroFunction() {}

r1r2_fun::
r1r2_fun(gas_data Q1, gas_data Q2, gas_data Q3, gas_data Q4, int nx)
    : ZeroFunction()
{
    Gas_model *g = get_gas_model_ptr();

    g->initialise_gas_data(Q_1_);
    g->initialise_gas_data(Q_2_);
    g->initialise_gas_data(Q_3_);
    g->initialise_gas_data(Q_4_);

    copy_gas_data(Q1, Q_1_);
    copy_gas_data(Q2, Q_2_);
    copy_gas_data(Q3, Q_3_);
    copy_gas_data(Q4, Q_4_);

    nx_ = nx;
}

r1r2_fun::
r1r2_fun(const r1r2_fun& zfun)
    : ZeroFunction()
{
    Gas_model *g = get_gas_model_ptr();

    g->initialise_gas_data(Q_1_);
    g->initialise_gas_data(Q_2_);
    g->initialise_gas_data(Q_3_);
    g->initialise_gas_data(Q_4_);

    copy_gas_data(zfun.Q_1_, Q_1_);
    copy_gas_data(zfun.Q_2_, Q_2_);
    copy_gas_data(zfun.Q_3_, Q_3_);
    copy_gas_data(zfun.Q_4_, Q_4_);
```

```
    nx_ = zfun.nx_;
}

r1r2_fun::~r1r2_fun() {}

int
r1r2_fun::
eval(double r1r2,
     double &y)
{
    // accepts a r1r2, returns velocity difference across the contact shock
    double u2, u3, p2p1, W, dp;
    // state 2
    if (get_state_behind_shock(Q_2_, Q_1_, r1r2) != SUCCESS) { return FAILURE; }
    if (get_shock_speed(W, r1r2, Q_1_, Q_2_) != SUCCESS) { return FAILURE; }
    u2 = get_velocity_behind_shock(r1r2, W);

    p2p1 = Q_2_.p/Q_1_.p;

    // begin at reservoir
    copy_gas_data(Q_4_, Q_3_);
    u3 = 0.0;

    dp = (Q_2_.p - Q_4_.p)/nx_; // assumption here that Q_2_.p is correct pressure
    for (size_t i = 0; i < (size_t)nx_; ++i) {
        step_across_expansion_dp(Q_3_, u3, dp);
    }
    y = (u3 - u2);

    return SUCCESS;
}

int
step_across_expansion_dp(gas_data &Q3,
                         double &u3,
                         double dp)
{
    Gas_model *g = get_gas_model_ptr();

    double du = -dp/(Q3.rho*Q3.a);
    double dT = dp/(Q3.rho*g->Cp(Q3));

    Q3.p = Q3.p + dp;
    Q3.T[0] = Q3.T[0] + dT;
    g->eval_thermo_state_pT(Q3);
    u3 = u3 + du;

    return SUCCESS;
}

int
step_across_expansion_du(gas_data &Q3,
                         double &u3,
                         double du)
{
    Gas_model *g = get_gas_model_ptr();
```

```
    double dp = -Q3.rho*Q3.a*du;
    double dT = dp/(Q3.rho*g->Cp(Q3));

    Q3.p = Q3.p + dp;
    Q3.T[0] = Q3.T[0] + dT;
    g->eval_thermo_state_pT(Q3);
    u3 = u3 + du;

    return SUCCESS;
}

int
get_state_behind_shock(gas_data &Q2,
                       gas_data Q1,
                       double r1r2)
{
    Gas_model* g = get_gas_model_ptr();
    double tol = 1e-6; // hard coded tolerance
    double p_old;
    do {
        Q2.e[0] = Q1.e[0] + 0.5*(1/Q1.rho)*(Q1.p + Q2.p)*(1 - r1r2);
        Q2.rho = Q1.rho/r1r2;
        p_old = Q2.p;
        if (g->eval_thermo_state_rhoe(Q2) != SUCCESS) {
            printf("get_state_behind_shock failed.\n");
            return FAILURE;
        }
    } while (fabs(p_old - Q2.p) > tol);

    return SUCCESS;
}

int
get_shock_speed(double &W,
                double r1r2,
                gas_data Q1,
                gas_data Q2)
{
    double WW = (1/Q1.rho)*(Q2.p - Q1.p)/(1.0 - r1r2);
    if (WW < 0.0) {
        printf("get_shock_speed failed.\n");
        return FAILURE;
    }
    W = sqrt(WW);
    return SUCCESS;
}

double
get_velocity_behind_shock(double r1r2,
                          double W)
{
    return W*(1.0 - r1r2);
}

int
eval_flux(vector<double> &flux, gas_data *Q4, gas_data *Q1, double tol)
```

```
{
    Gas_model* g = get_gas_model_ptr();

    gas_data Q2, Q3;
    g->initialise_gas_data(Q2);
    g->initialise_gas_data(Q3);
    copy_gas_data(*Q1, Q2);
    copy_gas_data(*Q4, Q3);

    int nsp = g->get_number_of_species();
    if (abs(Q4->p - Q1->p) < tol) {
        for (size_t i = 0; i < flux.size(); ++i) { flux[i] = 0.0; }
        return SUCCESS;
    }
    p2p1_fun *pfun = new p2p1_fun(*Q1, Q2, Q3, *Q4);
    Muller p2p1_solver(pfun, tol);

    // these bounds should always be sufficient to guarantee a root
    double p2p1 = p2p1_solver(0.0, Q4->p/Q1->p);
    double g1 = g->gamma(*Q1);
    double r2r1 = (1.0 + ((g1 + 1.0)/(g1 - 1.0))*p2p1)/((g1 + 1.0)/(g1 - 1.0) + p2p1);
    double W = Q1->a*sqrt(((g1 + 1.0)/(2*g1))*(p2p1 - 1.0) + 1.0);
    double u2 = get_velocity_behind_shock(1.0/r2r1, W);

    Q2.rho = Q1->rho*r2r1;
    Q2.p = Q1->p*p2p1;
    g->eval_thermo_state_rhop(Q2);

    double g4 = g->gamma(*Q4);
    double p3p4 = p2p1/(Q4->p/Q1->p);
    double r3r4 = pow(p3p4, 1.0/g4);
    Q3.rho = r3r4*Q4->rho;
    Q3.p = p3p4*Q4->p;
    g->eval_thermo_state_rhop(Q3);

    if ((int)flux.size() != 3+nsp) {
        printf("flux size incorrect, %i != %i\n", (int)flux.size(), 3+nsp);
        delete pfun;
        exit(1);
    }

    // Return the flux (without momentum)
    flux[0] = (Q3.rho*u2);
    flux[1] = (Q3.rho*u2*Q3.e[0]);
    for (int i = 0; i < nsp; ++i) {
        flux[2+i] = (u2*Q3.rho*Q3.massf[i]);
    }
    flux[2+nsp] = Q3.rho*u2;

    delete pfun;
    return SUCCESS;
}

vector<double>
python_eval_flux(gas_data *Q4, gas_data *Q1, double tol)
{
    Gas_model *g = get_gas_model_ptr();
```

```
    int nsp = g->get_number_of_species();
    vector<double> F(nsp+2, 0.0);
    eval_flux(F, Q4, Q1, tol);

    return F;
}
```

Listing B.12: *Sod's shock tube simulator, source file.*