The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.
Declaration of Original Work

This is to certify that the calculations, computer code, results and other work presented in this dissertation are essentially my own work, except where otherwise indicated, and has not been submitted for a degree at any other university.

Prakash Parbhoo

April 2000
Title

A Numerical Investigation into the Tactical Advantages of Supercruise for Combat Aircraft

By

P. Parbhoo

A dissertation submitted in partial fulfilment of the requirements for the degree of Master of Science in Engineering

Department of Mechanical Engineering
University of Cape Town
Rondebosch 7700
South Africa
Acknowledgements

I wish to thank the following for their contribution to this dissertation:

My supervisor, Dr C. Redelinghuys for his support, encouragement, and the opportunity to work with him and gain some insight into the attributes that a good engineer should possess.

My mother and late father who provided substantial support and encouragement.

Kirsten Winkler, without whom this document would not have been completed.

Shelley Adams for providing motivation and encouragement over many years.

Alfred Hoernle and Clem van der Riet for demanding that this dissertation be completed.
Terms of Reference

Funding for this work was provided by the Denel Group. The objective of the work would be to determine the tactical advantages, if any, of providing a Mirage III class combat aircraft with supercruise capability. This would be determined by developing numerical models of the aircraft components and producing performance predictions from these.
Synopsis

Developments in the field of combat aircraft design have resulted in a new generation of fighter aircraft. These new aircraft sport better aerodynamic design, advanced avionics and modern power plants. This leaves legacy combat aircraft vulnerable in a combat scenario. Although updated avionics provide some improvement for legacy combat aircraft, the aerodynamic design cannot be substantially modified. The power plant, or turbojet engine, can however be upgraded to a new model.

This dissertation describes a numerical investigation into the tactical advantages offered by upgrading a hypothetical aircraft with a newer engine model. The airframe is largely based on the Dassault Mirage III, and the engines are based on the SNECMA Atar 9k50 and M53.

A brief overview of recent developments in combat aircraft design and engine performance is provided which illustrates the trend towards high thrust to weight ratio aircraft engines with lower fuel consumption.

A thermodynamic model of a turbojet / low bypass ratio turbofan engine is developed and a parametric exercise is provided to illustrate the key factors influencing engine performance. A numerical model is developed for the supersonic intake which allows the prediction of both pressure recovery and mass flow rate through the component. The combined result of these models is shown to produce predictions which compare favourably with published data.

An aerodynamic model is built around the US Airforce Digital DATCOM. Some example output from this program is provided and compared to expected analytical trends.

The propulsion, intake and aerodynamic models are combined to provide predictions of key performance indicators such as Thrust Specific Fuel Consumption, Sustainable and Attainable Turn Rate, Specific Excess Power and Range.

These predictions indicate that upgrading a hypothetical combat airframe with a new propulsion unit offers some tactical advantages such as increased Specific Excess Power, Sustained Turn Rate and Range.

A recommendation is made that further simulations be performed to obtain a more detailed indication of how the suggested improvements in performance will be of benefit in a combat scenario.

Shortcomings of the models developed in this investigation are identified and suggestions are made as to where they should be improved.

All the source code for the various simulations is provided in the appendices.
Table of Contents

DECLARATION OF ORIGINAL WORK I

TITLE II

ACKNOWLEDGEMENTS III

TERMS OF REFERENCE IV

SYNOPSIS V

TABLE OF CONTENTS VI

LIST OF ILLUSTRATIONS XI

LIST OF TABLES XIII

LIST OF SYMBOLS XIV

CHAPTER 1: INTRODUCTION 1

CHAPTER 2: BACKGROUND 4

2.1 MILITARY AIRCRAFT PROPULSION 4

2.2 AIR COMBAT 5

CHAPTER 3: SIMULATION OVERVIEW 7

3.1 PERFORMANCE INDICATORS 7

3.1.1 RANGE 7

3.1.2 THRUST SPECIFIC FUEL CONSUMPTION 8

3.1.3 SPECIFIC EXCESS POWER 8

3.1.4 SUSTAINED TURN RATE AND ATTAINED TURN RATE 9

3.1.5 THRUST - DRAG - LOAD FACTOR CHART 10

3.2 MODELLING THE ENTIRE AIRCRAFT 11

CHAPTER 4: PROPULSION MODEL 12

4.1 TURBOJET MODEL 12

4.2 TYPICAL THERMODYNAMIC CYCLE 12

4.3 ENGINE COMPONENTS AND PROPERTIES 13
4.4 ESTIMATING THE THRUST

4.4.1 MODELLING ASSUMPTIONS AND SIMPLIFICATIONS

4.4.2 CALCULATION OF THE THRUST PER UNIT AIRFLOW INTO THE GAS GENERATOR

4.4.3 CALCULATION OF NET THRUST OUTPUT

4.5 ENGINE PERFORMANCE INDICATORS

4.5.1 THRUST SPECIFIC FUEL CONSUMPTION (TSFC) AND SPECIFIC IMPULSE

4.5.2 T-S DIAGRAM CALCULATIONS

4.6 FACTORS AFFECTING ENGINE PERFORMANCE

4.6.1 REFERENCE CONFIGURATION FOR THE PARAMETRIC INVESTIGATION

4.6.2 TURBINE INLET TEMPERATURE

4.6.3 COMPRESSOR PRESSURE RATIO

4.6.4 INTAKE PRESSURE RECOVERY

4.7 SOFTWARE IMPLEMENTATION OF THE PROPULSION MODEL

CHAPTER 5: INTAKE MODELLING

5.1 OVERVIEW OF INTAKE OPERATION

5.1.1 SUBSONIC FLIGHT

5.1.2 DETACHED SHOCK MODE

5.1.3 CRITICAL MODE

5.1.4 SUB CRITICAL MODE

5.1.5 SUPER CRITICAL MODE

5.1.6 MASS FLOW AND PRESSURE RECOVERY FOR DIFFERENT OPERATING MODES

5.2 CONICAL SHOCK PROPERTIES

5.3 STANDARD INTAKE PERFORMANCE CURVES

5.4 NUMERICAL ESTIMATION OF INTAKE PERFORMANCE

5.4.1 DETACHED SHOCK SOLUTION

5.4.2 CRITICAL SOLUTION

5.4.3 SUB CRITICAL SOLUTION

5.4.4 SUPER CRITICAL SOLUTION

5.5 SOFTWARE IMPLEMENTATION OF THE INTAKE MODEL

CHAPTER 6: AERODYNAMIC MODEL

6.1 OVERVIEW OF BASIC AERODYNAMICS

6.1.1 SUBSONIC FLIGHT AND WING SECTIONS
6.1.2 VARIATION OF DRAG WITH MACH NUMBER 42
6.2 DIGITAL DATCOM MODEL 42
6.2.1 DESCRIBING THE AIRFRAME 42
6.2.2 FLIGHT CONDITIONS 46
6.2.3 LIMITATIONS OF THE MODEL 46
6.3 RESULTS OF THE DATCOM MODEL 47
6.3.1 $C_L$ VS. AOA 47
6.3.2 $C_D$ VS. AOA 47
6.3.3 $C_{L \text{ max}}$ VS. $M$ 48
6.3.4 $C_D$ VS. MACH NUMBER 49
6.3.5 $C_l/C_D$ VS. $C_L$ 50
6.4 SOFTWARE IMPLEMENTATION OF THE AERODYNAMIC MODEL 51

CHAPTER 7: SIMULATION RESULTS 52
7.1 COMPARISON OF ENGINE PERFORMANCE 52
7.1.1 PROPULSION MODEL INPUT PARAMETERS 52
7.1.2 THERMODYNAMIC CYCLE: T-S DIAGRAMS 54
7.1.3 THRUST SPECIFIC FUEL CONSUMPTION (TSFC) 56
7.1.4 NET THRUST 58
7.2 COMPARISON OF THE INTAKE MODEL WITH STANDARD CURVES 60
7.3 THRUST - DRAG - LOAD FACTOR CHART 61
7.4 SPECIFIC EXCESS POWER 64
7.5 SUSTAINED TURN RATE 65
7.6 RANGE 66

CHAPTER 8: CONCLUSIONS AND RECOMMENDATIONS 68
8.1 CONCLUSIONS 68
8.2 RECOMMENDATIONS 69

REFERENCES 70

APPENDIX A: INTAKE MODEL 72
APPENDIX A-1 FLOW CHART 72
<table>
<thead>
<tr>
<th>APPENDIX</th>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-2</td>
<td>DETACHED SHOCK EQUATIONS</td>
<td>73</td>
</tr>
<tr>
<td>A-3</td>
<td>SAMPLE CONICAL SHOCK DATA</td>
<td>75</td>
</tr>
<tr>
<td>A-4</td>
<td>INTAKE MODEL: CLASS CConeIntake</td>
<td>76</td>
</tr>
<tr>
<td>A.4.1</td>
<td>Header File: ConeIntake.h</td>
<td>76</td>
</tr>
<tr>
<td>A.4.2</td>
<td>Implementation File: ConeIntake.cpp</td>
<td>77</td>
</tr>
<tr>
<td>B</td>
<td>DATCOM MODEL</td>
<td>92</td>
</tr>
<tr>
<td>B-1</td>
<td>SCHEMATIC DRAWINGS</td>
<td>92</td>
</tr>
<tr>
<td>B-2</td>
<td>SAMPLE INPUT FILE</td>
<td>94</td>
</tr>
<tr>
<td>B-3</td>
<td>SAMPLE OUTPUT FILE</td>
<td>95</td>
</tr>
<tr>
<td>B-4</td>
<td>DATCOM Post-Processing Source Code</td>
<td>101</td>
</tr>
<tr>
<td>B.4.1</td>
<td>Program: DATCOM Post</td>
<td>101</td>
</tr>
<tr>
<td>B.4.2</td>
<td>Program: CD_CL</td>
<td>107</td>
</tr>
<tr>
<td>B.4.3</td>
<td>Program: CL_CD_FULL</td>
<td>111</td>
</tr>
<tr>
<td>C</td>
<td>PROPULSION MODEL</td>
<td>116</td>
</tr>
<tr>
<td>C-1</td>
<td>FLOW CHART FOR ESTIMATING THRUST OUTPUT</td>
<td>116</td>
</tr>
<tr>
<td>C-2</td>
<td>PROPULSION MODEL: CLASS TurboJet1D</td>
<td>117</td>
</tr>
<tr>
<td>C.2.1</td>
<td>Header File: 1DPropModel.hpp</td>
<td>117</td>
</tr>
<tr>
<td>C.2.2</td>
<td>Implementation File: 1DPropModel.cpp</td>
<td>120</td>
</tr>
<tr>
<td>C-3</td>
<td>PROPULSION SIMULATION CODE</td>
<td>138</td>
</tr>
<tr>
<td>C.3.1</td>
<td>Program: TSFC</td>
<td>138</td>
</tr>
<tr>
<td>C.3.2</td>
<td>Program: THRUST</td>
<td>142</td>
</tr>
<tr>
<td>C.3.3</td>
<td>Program: CALCTSDiagram</td>
<td>148</td>
</tr>
<tr>
<td>D</td>
<td>PERFORMANCE ESTIMATION SOURCE CODE</td>
<td>152</td>
</tr>
<tr>
<td>D-1</td>
<td>SUSTAINED TURN RATE</td>
<td>152</td>
</tr>
<tr>
<td>D-2</td>
<td>SPECIFIC EXCESS POWER</td>
<td>158</td>
</tr>
<tr>
<td>D-3</td>
<td>DRAG - THRUST - LOAD FACTOR CHART</td>
<td>168</td>
</tr>
<tr>
<td>D.3.1</td>
<td>Program: CDMNALT</td>
<td>168</td>
</tr>
<tr>
<td>D.3.2</td>
<td>Program: CLvSMvsn</td>
<td>172</td>
</tr>
</tbody>
</table>
APPENDIX E: GENERAL SOURCE CODE

APPENDIX E-1 TAYLOR-MACOLL FLOW
E.1.1 MAIN FILE: TMFLOWPROPSMAIN.cpp
E.1.2 HEADER FILE: TMFLOWPROPS.h
E.1.3 IMPLEMENTATION FILE: TMFLOWPROPS.cpp

APPENDIX E-2 LOOKUP TABLES
E.2.1 HEADER FILE: LOOKUPTB.hpp
E.2.2 IMPLEMENTATION FILE: LOOKUPTB.cpp
E.2.3 HEADER FILE: LOOKUP4D.h
E.2.4 IMPLEMENTATION FILE: LOOKUP4D.cpp

APPENDIX E-3 IDEAL GAS BEHAVIOUR
E.3.1 HEADER FILE: IDEALGAS.h
E.3.2 IMPLEMENTATION FILE: IDEALGAS.cpp
E.3.3 HEADER FILE: IDEALGASSTREAM.h
E.3.4 IMPLEMENTATION FILE: IDEALGASSTREAM.cpp

APPENDIX E-4 DATA INPUT FILE SYSTEM
E.4.1 HEADER FILE: DATAFILEINPUT.hpp
E.4.2 IMPLEMENTATION FILE: DATAFILEINPUT.cpp
List of Illustrations

Figure 2.1: Trends in Thrust to Weight Ratio ................................................................. 4
Figure 2.2: Typical Aircraft Operating Envelopes and Stagnation Temperature limits .......... 5
Figure 2.3: Trends in TSFC ......................................................................................... 5
Figure 3.1: STR and ATR in simulated short range combat ............................................. 10
Figure 4.1: Schematic of a turbojet engine with bypass .................................................. 12
Figure 4.2: Typical T-s diagram for the Brayton cycle, with bypass and losses ................. 12
Figure 4.3: Variation of TSFC and $F/\dot{m}_s$ with turbine inlet temperature .................. 24
Figure 4.4: Variation of TSFC and $F/\dot{m}_s$ with compressor pressure ratio .................. 24
Figure 4.5: Variation of TSFC and $F/\dot{m}_s$ with diffuser pressure recovery .................... 25
Figure 5.1: Schematic of a supersonic conical intake with two-shock system .................. 27
Figure 5.2: Detached shock arrangement ........................................................................ 28
Figure 5.3: Critical mode shock arrangement ................................................................. 28
Figure 5.4: Two flow arrangements in sub critical mode .................................................. 29
Figure 5.5: Super critical mode shock arrangement ........................................................ 30
Figure 5.6: Mass flow and Pressure Recovery for different operating modes .................... 30
Figure 5.7: Streamlines for supersonic flow over a cone ................................................ 31
Figure 5.8: Comparison between published conical shock wave data and calculated results . 32
Figure 5.9: Standard pressure recovery curves ............................................................... 33
Figure 5.10: Schematic representation of a detached shock and its discretisation ............... 34
Figure 5.11: Pressure recovery in critical mode operation ................................................ 35
Figure 5.12: Parameters for determining the mass flow through the intake ....................... 36
Figure 6.1: Basic aerodynamic forces acting on a wing ................................................... 39
Figure 6.2: Variation of lift coefficient with angle of attack ............................................. 40
Figure 6.3: Idealised variation of drag coefficient with angle of attack ............................. 41
Figure 6.4: Variation of Lift-to-Drag ratio with $C_L$ ....................................................... 41
Figure 6.5: Variation of $C_{D_0}$ with Mach number ......................................................... 42
Figure 6.6: Schematic aircraft model for use in the Digital DATCOM ................................ 43
Figure 6.7: Airfoil section used for wing specification ........................................ 44
Figure 6.8: Lift coefficient ($C_L$) for various angles of attack, at various Mach numbers ....... 47
Figure 6.9: Drag coefficient (untrimmed) for various angles of attack and Mach numbers .... 48
Figure 6.10: Maximum lift coefficient vs. Mach number ........................................... 48
Figure 6.11: $C_D$ for maximum $C_L$ at 4000m ..................................................... 49
Figure 6.12: Variation of $C_{D0}$ with Mach Number at 4000m ................................. 50
Figure 6.13: Variation of Lift to Drag ratio with Lift in the supersonic regime at sea level ..... 50
Figure 7.1: T-s Diagram for the 9k50, $M=0.66$, $A/B$ = Off, Sea Level .................... 54
Figure 7.2: T-s Diagram for the M53, $M=0.66$, $A/B$ = Off, Sea Level ................... 55
Figure 7.3: T-s Diagram for the 9k50, $M=0.66$, $A/B$ = On, Sea Level .................... 55
Figure 7.4: T-s Diagram for the M53, $M=0.66$, $A/B$ = On, Sea Level .................... 56
Figure 7.5: Comparison of TSFC between the 9k50 and M53 at Sea Level ................. 56
Figure 7.6: Comparison of TSFC between the 9k50 and M53 at 14 km ...................... 57
Figure 7.7: Comparison of Thrust output between the 9k50 and M53 at Sea Level .......... 58
Figure 7.8: Comparison of Thrust output between the 9k50 and M53 at 4 km ............... 59
Figure 7.9: Comparison of Thrust output between the 9k50 and M53 at 14km .............. 60
Figure 7.10: Diffuser performance comparison .................................................... 60
Figure 7.11: $C_L$ required for a given load factor and Mach number at 4km ............... 61
Figure 7.12: $C_D$ for $n=1$ to $n=9$ with $C_T$ superimposed at 4km ....................... 62
Figure 7.13: Load Factor Map for Mach number and Altitude ............................... 63
Figure 7.14: Specific Excess Power (SEP) [m/s] for the 9k50 and M53, $n=1$ .............. 64
Figure 7.15: Sustained Turn rate comparison for the 9k50 and M53 at 4km ............... 65
Figure 7.16: Sustained Turn rate, with Attainable Turn Rate (ATR) superimposed .......... 66
Figure 7.17: Relative Range for different engine configurations at 4000m ................... 67
Figure B.1: Plan view of the hypothetical fighter .................................................. 92
Figure B.2: Elevation view of the hypothetical fighter ........................................... 92
Figure B.3: Schematic outline superimposed on drawings from Jane's [3] .................... 93
List of Tables

Table 4.1: Turbojet engine components and properties ........................................ 13
Table 4.2: Parameter values for the reference configuration .................................. 23
Table 4.3: Parameters for Function Cycle Thermodynamics .................................. 26
Table 6.1: DATCOM parameters for the wing planform ..................................... 44
Table 6.2: DATCOM parameters for modelling the Elevators ............................ 45
Table 6.3: DATCOM parameters for modelling the Tail ...................................... 46
Table 7.1: Propulsion model input parameters .................................................. 52
Table 7.2: Rated and Predicted Static Thrust for the 9k50 and M53 at Sea Level ..... 58
Table 7.3: Flight conditions for Range estimation ............................................. 67
List of Symbols

<table>
<thead>
<tr>
<th>Roman</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Area</td>
</tr>
<tr>
<td>$C_p$</td>
<td>Specific heat</td>
</tr>
<tr>
<td>D</td>
<td>Drag</td>
</tr>
<tr>
<td>F</td>
<td>Thrust</td>
</tr>
<tr>
<td>f</td>
<td>Fuel/air ratio</td>
</tr>
<tr>
<td>g</td>
<td>Gravitational acceleration</td>
</tr>
<tr>
<td>h</td>
<td>Enthalpy or Height</td>
</tr>
<tr>
<td>L</td>
<td>Lift</td>
</tr>
<tr>
<td>M</td>
<td>Mach number</td>
</tr>
<tr>
<td>m</td>
<td>Mass</td>
</tr>
<tr>
<td>n</td>
<td>Load factor</td>
</tr>
<tr>
<td>p</td>
<td>Pressure</td>
</tr>
<tr>
<td>$Q_R$</td>
<td>Lower heating value of fuel</td>
</tr>
<tr>
<td>R</td>
<td>Gas constant or Range</td>
</tr>
<tr>
<td>r</td>
<td>Radius (capture streamtube)</td>
</tr>
<tr>
<td>S</td>
<td>Aerodynamic reference area</td>
</tr>
<tr>
<td>s</td>
<td>Entropy or Distance</td>
</tr>
<tr>
<td>T</td>
<td>Temperature or Thrust</td>
</tr>
<tr>
<td>t</td>
<td>Time</td>
</tr>
<tr>
<td>u</td>
<td>Fluid velocity</td>
</tr>
<tr>
<td>V</td>
<td>Speed</td>
</tr>
<tr>
<td>W</td>
<td>Weight</td>
</tr>
<tr>
<td>x</td>
<td>Linear position along conical centrebody axis</td>
</tr>
</tbody>
</table>

Aerodynamic Coefficients

<table>
<thead>
<tr>
<th>Subscript</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_L$</td>
<td>Lift coefficient</td>
</tr>
<tr>
<td>$C_D$</td>
<td>Drag coefficient</td>
</tr>
<tr>
<td>$C_T$</td>
<td>Thrust coefficient</td>
</tr>
<tr>
<td>$C_W$</td>
<td>Weight coefficient</td>
</tr>
</tbody>
</table>

Subscripts

<table>
<thead>
<tr>
<th>Subscript</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Actual intake capture streamtube</td>
</tr>
<tr>
<td>0-9</td>
<td>Station identifiers</td>
</tr>
<tr>
<td>a</td>
<td>Afterburner or Primary air stream entering gas generator</td>
</tr>
<tr>
<td>avg</td>
<td>Average</td>
</tr>
<tr>
<td>b</td>
<td>Burner or Bypass stream</td>
</tr>
<tr>
<td>c</td>
<td>Compressor or Cone surface or Capture</td>
</tr>
<tr>
<td>d</td>
<td>Diffuser or Design</td>
</tr>
<tr>
<td>f</td>
<td>Fan or Fuel</td>
</tr>
<tr>
<td>lip</td>
<td>Cowl/Intake lip position</td>
</tr>
<tr>
<td>n</td>
<td>Nozzle</td>
</tr>
<tr>
<td>t</td>
<td>Stagnation value or Turbine</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>$\infty$</td>
<td>Free stream</td>
</tr>
<tr>
<td>$\beta$</td>
<td>Bypass ratio or Conical shock angle</td>
</tr>
<tr>
<td>$\delta$</td>
<td>Stagnation / static pressure ratio</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>Drag to Lift ratio or Shock angle (Taylor-Macoll flow)</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>Specific heat ratio</td>
</tr>
<tr>
<td>$\eta$</td>
<td>Efficiency</td>
</tr>
<tr>
<td>$\pi$</td>
<td>Stagnation pressure ratio</td>
</tr>
<tr>
<td>$\theta$</td>
<td>Stagnation / static temperature ratio</td>
</tr>
<tr>
<td>$\rho$</td>
<td>Density</td>
</tr>
<tr>
<td>$\tau$</td>
<td>Stagnation temperature ratio</td>
</tr>
</tbody>
</table>
Chapter 1: Introduction

Recent advances in the areas of computational fluid dynamics, stability and control, structural and material design have had a great impact on the aircraft industry. In particular, the design of combat aircraft, which have always been at the forefront of technology, has benefited greatly from innovations stemming from these advances.

New fighter aircraft are designed to be operated into the post-stall region to afford high agility, but remain controllable through advanced aerodynamic design and control systems. New combat aircraft engines provide greater thrust to weight ratios and exhibit lower fuel consumption [1]. More advanced technologies such as thrust vectoring [2] provide for an even more manoeuvrable aircraft.

New developments in armaments have also taken place with a wide selection of short-range air-to-air and ground-to-air missiles available. These weapons typically employ advanced guidance/seeking systems which render them lethal and allow them to be all aspect weapons [1]. Developments have also been made in the field of all aspect guns. Because these weapons need not be fired with the aircraft pointing towards the target, the traditional combat scenario of gaining an advantage over the opponent and firing from behind no longer applies to the same extent as it did in the 1950's through 1980's.

New combat aircraft are typically capable of supersonic flight without the use of an afterburner. This capability is termed supercruise and has arisen from the development of propulsion unit technology. Older aircraft do not have this capability and rely on the utilisation of an afterburner to gain the necessary thrust to perform supersonic flight.

Thus, older aircraft are becoming increasingly more vulnerable in short-range combat. New aircraft employing the latest technology are expensive, and generally subject to export restrictions from the country of manufacture. One component of these older aircraft that can be readily upgraded, however, is the propulsion unit. Newer propulsion units offer the high thrust to weight ratios and low fuel consumption that the new aircraft have as standard.

However, combat performance is the integrated response of aerodynamics, avionics, structural strength and propulsion. The investigation described in this report therefore aims to determine what benefits and, possibly, drawbacks upgrading only the engine would have for an aircraft with an old airframe design. I.e. would an old airframe reap any tactical advantages by having supercruise capability?
In order to perform this task, a simulation model had to be developed. An overview of the simulation model will be given in Chapter 3. The airframe, aerodynamics and propulsion unit are identified as key components of this model. Chapter 3 also describes some of the performance metrics identified as being indicative of the performance of an engine, and engine-airframe combination.

The Dassault Mirage III was chosen as the basis of a hypothetical aircraft due to its age (the first Mach 2 capable prototype was rolled out in 1956 [3]) and the large number that were deployed by numerous airforces around the world, making it one of the most successful fighter aircraft ever produced. Two engines from French manufacturer SNECMA were chosen for evaluation, the Atar 9k50 and the M53. The 9k50, a pure afterburning turbojet, was deployed in the Mirage III fleet obtained by South Africa and subsequently utilised in the Cheetah. The M53 was first used in the Mirage 2000 and is a low bypass ratio afterburning turbofan engine.

The development of the propulsion model is detailed in Chapter 4. The model developed is one-dimensional and ignores three-dimensional flow effects. Instead the development concentrates on the thermodynamic cycle and a theoretical analysis of it to arrive at a set of equations which fully describe its behaviour. Some of the key factors affecting a turbojet's performance are detailed through some simple simulation tests. The software implementation is described at the end of the chapter.

The analysis of the propulsion unit identifies the air intake as a key factor in the performance of the propulsion unit. Chapter 5 deals with this complex component. The various modes of operation are described and a numerical method for determining both the pressure recovery and mass flow is developed. A brief description of the software implementation of this model is then provided.

Chapter 6 provides some background on basic aerodynamic theory in order to put into context the results obtained later in the chapter. The use of a numerical tool developed by the US Airforce for estimating stability and control parameters, the Digital DATCOM, is described. The model parameters and software for output processing from the DATCOM are described, along with some of the primary aerodynamic coefficient results obtained.

The combined model results are detailed in Chapter 7. Results from the combined intake and propulsion unit models are used to compare two engine models, and an explanation of the differences in their performance is given. The combined aerodynamic, intake and propulsion model results are then presented, highlighting the changes in overall performance due a change in engine.
Chapter 8 describes some of the conclusions that can be drawn from the results obtained and offers some suggestions with regard to areas that require more detailed study and how the model can be improved.
Chapter 2: Background

This chapter provides a brief background to developments in military aircraft propulsion and some of the developments in air combat practices.

2.1 Military Aircraft Propulsion

Early fighter aircraft were developed with pure turbojet engines. A reheat, or afterburner, section was generally provided to boost the maximum thrust that the propulsion unit could deliver. In general, these engines provided insufficient dry (without afterburning) thrust to achieve supersonic cruising [4].

Furthermore, these aircraft tended to have specialised roles such as, for example, bombing, combat, interception and reconnaissance. Combat typically took place at high subsonic speeds at fairly low altitudes in a head-to-tail chase arrangement [1].

Developments in military aircraft propulsion have focussed around providing engines which have higher thrust to weight ratios (see Figure 2.1) and higher thrust per unit airflow through the engine [4]. Maximum speed has not changed greatly due to limitations imposed by supersonic aerodynamics and aerodynamic heating due to high stagnation temperatures associated with high supersonic flow (see Figure 2.2). Thus, higher thrust to weight ratios and higher thrust per unit airflow allow a smaller, lighter engine to be utilised. This reduces the wing loading and allows an aircraft to have increased agility.

Modern combat aircraft also tend to have multi-role requirements [4]. This multi-role capability imposes supersonic flight and cruise requirements on the aircraft. The supersonic flight capability can be achieved at the expense of engine mass. By utilising the gains in thrust to weight ratio and thrust per unit airflow, a modern engine of the same dimensions and mass as an older engine can deliver significantly more thrust. This ability to cruise at supersonic speeds without the use of an afterburner is termed supercruise.
These gains in engine performance have been achieved by moving from pure turbojet engines to low bypass ratio turbofans (see Figure 2.3), with or without an afterburner. In addition, new materials and design techniques have resulted in more efficient and higher pressure-ratio compressors, and turbines with higher working temperatures. Old engines such as the SNECMA 9k50 operate with compressor pressure ratios of around 6 and turbine inlet temperatures around 1200K. Modern engines can have compressor pressure ratios in excess of 10 and turbine inlet temperatures approaching the stoichiometric burning temperature of the fuel (approximately 2400K) [4].

### 2.2 Air Combat

Traditional combat scenarios have typically involved a head-to-tail chase arrangement with firing on the target taking place from the rear. Armaments utilised were fired with the aircraft nose pointing towards the rear of the target, in a similar flight path. These weapons are termed rear-aspect weapons. Combat of this nature required high sustainable performance in, for
example, a turn, in order to maintain a position of superiority over the opponent. High thrust to weight ratios were therefore critical [1,6].

Recent developments in air-to-air weapons and fire control systems have resulted in all-aspect capability short-range missiles and guns. These weapons no longer need to be fired in the classical head-to-tail configuration. In fact, Herbst [1] has shown that these weapons are far better utilised in head-to-head combat where they score high hit rates.

Furthermore, all-aspect weapons leave no part of an aircraft safe from attack - attacks can be made from all directions. The best way to avoid being shot down is to respond aggressively and achieve a position from which to fire before the opponent does. The ability to respond rapidly requires the utilisation of unsustainable transient performance in order to gain an advantage over the opponent. The energy lost through these manoeuvres can be recovered at a later stage [1,6]. This has, in recent years, placed more emphasis on high agility aircraft, and has placed greater requirements on the aerodynamic and avionic performance of the aircraft. The performance of the aircraft in combat is now somewhat less sensitive to thrust to weight ratio, but it is still important in order to be able to recover lost energy as rapidly as possible.

In general, combat manoeuvres require a constant interchange of potential and kinetic energy. High engine performance is essential for [1]:

- **Turning**: excess thrust is needed for high drag.
- **Climbing**: excess power is required for the accumulation of potential energy.
- **Acceleration**: excess power is required for the accumulation of kinetic energy.
Chapter 3: Simulation Overview

This chapter describes the approach utilised for the numerical modelling of the aircraft's performance. The indicators used to evaluate the performance are described along with a brief description of how they are calculated. The manner in which the aircraft is discretised for modelling purposes is also covered.

3.1 Performance Indicators

3.1.1 Range

The maximum flight range of a fighter aircraft is an important tactical parameter since it determines the furthest distance from which an airforce can launch a strike against an enemy. The range for a military aircraft is usually determined by a mission profile [5]. However, a simple estimate of the cruising range at constant altitude and airspeed is given by the Breguet range equation for propeller driven aircraft, and the modified Breguet equation for turbojet aircraft.

While the aircraft is cruising in level flight, the engine provides sufficient thrust only to counteract the drag. The drag in turn is dependent on the required lift, which is in turn dependent on the weight of the aircraft. As the aircraft burns fuel, the total mass, and therefore the total lift required is reduced. The range of the aircraft can therefore be calculated as follows:

The fuel weight consumption rate is given by Eq. 3.1

\[ \dot{W}_f = (TSFC)D \]  

Eq. 3.1

where

- (TSFC) = Thrust specific fuel consumption [N/Ns]
- \( D \) = Drag [N]

The weight of the aircraft then varies according to:

\[ \frac{dW}{W} = \frac{(TSFC) \cdot \varepsilon \cdot ds}{V} \]  

Eq. 3.2

where

- \( \varepsilon \) = Drag to Lift ratio
- \( V \) = Cruising speed
- \( s \) = Distance
Assuming that the aircraft operates at constant $e/V$ and TSFC, Eq. 3.2 can be integrated [5] to yield the modified Breguet range equation for turbojet driven aircraft:

$$R = \frac{V}{(TSFC)E} \ln \left( 1 + \frac{W_F}{W_E} \right)$$  \hspace{1cm} \text{Eq. 3.3}$$

where:  
$W_F$ = Weight full  
$W_E$ = Weight empty

3.1.2 Thrust Specific Fuel Consumption

Thrust Specific Fuel Consumption (TSFC) is a measure of the rate at which fuel is consumed for each unit of thrust generated. Given two turbojet engines, the one with the lower TSFC value will consume less fuel for a given thrust output. This has implications for range, as discussed in Section 3.1.1.

$$TSFC = \frac{\dot{m}_f}{F}$$  \hspace{1cm} \text{Eq. 3.4}$$

where:  
$\dot{m}_f$ = Fuel flow rate  
$F$ = Engine thrust

The calculation of TSFC from the thermodynamic cycle for a turbojet engine with a bypass stream will be discussed in more detail in Section 4.4.3.

3.1.3 Specific Excess Power

The rate of climb for an aircraft can be derived from the equations of motion [5] as:

$$\frac{dh}{dt} = V \left[ \frac{T-D}{W} - \frac{1}{g} \frac{dV}{dt} \right]$$

or

$$(T-D)V = W \frac{dh}{dt} + \frac{d}{dt} \left( \frac{W}{2g} V^2 \right)$$

writing

$$h_e = h + \frac{V^2}{2g}$$

$h_e$ represents the total energy per unit weight of the aircraft, i.e. both the kinetic and potential energy.
The Specific Excess Power (SEP) is then given by

$$SEP = \frac{dh}{dt} = \frac{V(T-D)}{W} \quad \text{Eq. 3.5}$$

It can be seen that $TV$ is the available power and that $DV$ is the required power [5], the difference being the excess power. This excess power can be used either to climb or to accelerate and is therefore an important parameter for combat aircraft.

Eq. 3.5 also shows that for two aircraft with the same airframe, SEP is affected primarily by the thrust term $T$, but also through $W$ which is affected by engine mass. In addition, the lift required is affected by $W$ and therefore the drag term $D$ is also affected by engine mass. A lighter engine therefore has benefits for the aircraft's performance in terms of SEP.

3.1.4 Sustained Turn Rate and Attained Turn Rate

The Sustained Turn Rate (STR) of an aircraft is the maximum constant rate at which it can turn, at constant altitude, forward speed and bank angle. This parameter is critical for combat aircraft since it influences engagement with enemy aircraft, as will be shown later. Sustained turn rate is calculated from Eq. 3.6.

$$STR = g \sqrt{\frac{\rho C_L (n^2 - 1)}{2n(W/S_w)}} \quad \text{Eq. 3.6}$$

where $n$ is the load factor and is given by Eq. 3.7:

$$n = \frac{L}{W} \quad \text{Eq. 3.7}$$

for a condition where the thrust matches the drag, i.e. $T = D$, Eq. 3.7 can be written as:

$$n = \frac{L}{D} \cdot \frac{T}{W} = \frac{C_L}{C_D} \cdot \frac{C_T}{C_w} \quad \text{Eq. 3.8}$$

In order to compare the performance of different aircraft engines, a plot of STR vs. Mach number can be generated [6]. The following algorithm is used to calculate this plot:

For a given altitude and free stream Mach number, $M_{aoa}$, the thrust coefficient, $C_T$ can be determined from the propulsion model. From the aerodynamic performance data, the angle of attack at which a matching drag coefficient equal to the thrust coefficient occurs can be determined. The value of the lift coefficient at this angle of attack can then be determined. The STR can then be calculated from Eq. 3.6.
The Attainable Turn Rate (ATR) is primarily governed by aerodynamic and structural strength considerations. ATR is the highest achievable turn rate that the aircraft is capable of, although it might not be sustainable. ATR is calculated from the following algorithm:

For a particular Mach number, the maximum value for \( C_L \) and its corresponding \( C_R \) are determined. A value for weight coefficient, \( C_w \), is calculated. \( C_L/C_D \) and \( n \) (from Eq. 3.7) are calculated. If \( n \) exceeds the structural limit of the airframe, it is set to \( n_{\text{max}} \) and a new value, \( C_{\text{ATR}} \), is calculated as the required lift coefficient. The ATR is then calculated from Eq. 3.9, which is seen to be similar to Eq. 3.6:

\[
\text{ATR} = \frac{\sqrt{(\rho \times C_{\text{ATR}}(n^2 - 1))}}{2n(W/S_p)}
\]

Figure 3.1 illustrates a simulated short-range combat scenario. It can be clearly seen how the pilot utilises the attainable turn rate limits, and how he then flies below the STR line in order to regain energy lost by flying above the STR line.

The figure also clearly shows the two limits which bound the attainable turn rate: The maximum lift condition at low Mach numbers and the maximum load factor, or structural limit, at higher Mach numbers.

Furthermore, the maximum attainable turn rate occurs at relatively low Mach numbers and is significantly larger than the sustained rate.

3.1.5 Thrust - Drag - Load Factor chart

At a given altitude and Mach number, a value for \( C_L \) can be calculated which yields flight at a given load factor, \( n \), as given by Eq. 3.7.

Utilising the aerodynamic data for the airframe, values for \( C_D \) can be determined from the calculated values for \( C_L \). These can be plotted against Mach number, as lines of constant \( n \), as shown in Figure 7.12. Values of the thrust coefficient, \( C_T \), can then be superimposed on this graph.

From this plot, the maximum attainable Mach number in level flight can be determined as the intersection of the \( C_T \) and \( C_D \times n - 1 \) curves. The maximum attainable load factor can also be determined. Thus by plotting the \( C_T \) curves for various engines, a quick comparison of their performance can be obtained.

![Figure 3.1: STR and ATR in simulated short range combat](source [1])
3.2 Modelling the entire Aircraft

An aircraft in flight can be viewed as a free body with only four primary forces acting upon it. These are Lift opposing Weight in the vertical direction, and Drag opposing Thrust in the horizontal direction. Of these, the Lift and Drag components are dependent on the aircraft's aerodynamic properties, while weight is governed by the airframe and fuel consumption. Thrust is dependent upon the output of the engine. The aircraft model can therefore be broken down into the following components:

**Airframe**

Of the three aircraft sub-models, the airframe is the simplest to model for the applications listed in Section 3.1 above. The required parameters are:

- Airframe mass (including or excluding armaments).
- Current fuel load.
- Maximum load factor that the airframe can withstand.

**Aerodynamic Model**

The aerodynamic model must provide values for the lift and drag coefficients \( C_l \) and \( C_d \), respectively at various Mach numbers and altitudes.

The trimmed coefficients are desired, as these would give the values of the parameters in steady flight. However, for the purposes of this investigation, the assumption can be made that the aircraft is neutrally stable and the coefficients can be utilised as calculated, without taking into account trim effects. This is acceptable because the same aerodynamic data is used for all the simulations, and no dynamic aerodynamic effects are being evaluated.

**Propulsion Model**

The propulsion model must provide values for the thrust and fuel consumption at various Mach numbers and altitudes.

The assumption is made that each of these aspects of the aircraft can be modelled independently and that the results can then be combined to yield an integrated simulated response for the entire aircraft.
Chapter 4: Propulsion Model

4.1 Turbojet Model

The turbojet model used for the development of a one-dimensional propulsion model is based on a turbojet model developed by Kerrebrock [7], but has been modified to account for a bypass stream. Figure 4.1 shows a schematic diagram on which the model and thermodynamic analysis are based.

![Schematic of a turbojet engine with bypass.](image)

Figure 4.1: Schematic of a turbojet engine with bypass.

4.2 Typical Thermodynamic Cycle

The ideal cycle for the turbojet engine is the Brayton cycle [8]. A typical T-s diagram for this cycle, modified for a bypass stream, and including component losses, is shown in Figure 4.2. Station numbers correspond to those shown in Figure 4.1, with isentropic end states indicated by numbers subscripted with 's'.

![Typical T-s diagram for the Brayton cycle, with bypass and losses.](image)

Figure 4.2: Typical T-s diagram for the Brayton cycle, with bypass and losses
### 4.3 Engine components and properties

#### Table 4.1: Turbojet engine components and properties

<table>
<thead>
<tr>
<th>COMPONENT</th>
<th>STAGE</th>
<th>PRESSURE RATIO</th>
<th>TEMP RATIO</th>
<th>EFFICIENCY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diffuser</td>
<td>0-2</td>
<td>( \frac{p_{i2}}{p_{i1}} ) (known)</td>
<td>( \frac{T_{i2}}{T_{i1}} ) (adiabatic)</td>
<td>( \eta_d = \frac{h_{i2} - h_{j2}}{h_{i2} - h_{j1}} )</td>
</tr>
<tr>
<td>Fan</td>
<td>2-8</td>
<td>( \frac{p_{i6}}{p_{i7}} ) (known)</td>
<td>( \frac{T_{i6}}{T_{i7}} )</td>
<td>( \eta_f = \frac{\pi_f}{\tau_f} )</td>
</tr>
<tr>
<td>Compressor</td>
<td>2-3</td>
<td>( \frac{p_{i3}}{p_{i4}} ) (known)</td>
<td>( \frac{T_{i3}}{T_{i4}} )</td>
<td>( \eta_c = \frac{\pi_c}{\tau_c} )</td>
</tr>
<tr>
<td>Burner</td>
<td>3-4</td>
<td>( \frac{p_{i6}}{p_{i7}} ) (assumed)</td>
<td>( \frac{T_{i6}}{T_{i7}} )</td>
<td>( \eta_b = C_p \left[ \left( \dot{m}<em>{i6} + \dot{m}</em>{j6} \right) T_{i4} - \dot{m}<em>{i5} T</em>{i5} \right] )</td>
</tr>
<tr>
<td>Turbine</td>
<td>4-5</td>
<td>( \frac{p_{i4}}{p_{i5}} )</td>
<td>( \frac{T_{i4}}{T_{i5}} )</td>
<td>( \eta_t = \frac{h_{i4} - h_{i5}}{h_{i4} - h_{i6}} )</td>
</tr>
<tr>
<td>Afterburner</td>
<td>5-6</td>
<td>( \frac{p_{i6}}{p_{i7}} )</td>
<td>( \frac{T_{i6}}{T_{i7}} )</td>
<td>( \eta_a = C_p \left[ \left( \dot{m}<em>{i6} + \dot{m}</em>{j6} \right) T_{i6} - \dot{m}<em>{i5} T</em>{i6} \right] )</td>
</tr>
<tr>
<td>Nozzle</td>
<td>6-7</td>
<td>( \frac{T_{i7}}{T_{i6}} ) (adiabatic)</td>
<td>( \eta_n = \frac{h_{i6} - h_{i7}}{h_{i5} - h_{i6}} )</td>
<td></td>
</tr>
</tbody>
</table>
4.4 Estimating the Thrust

4.4.1 Modelling assumptions and simplifications

Stagnation conditions

The assumption is made that the flow velocity is sufficiently small at the station numbers suffixed with a '1' in Figure 4.1, that the thermodynamic state of the gas at those points is represented by the stagnation properties [9]. However, this is not actually the case, and internal velocities might be quite large [9].

Average specific heats

The assumption is made that the specific heats and specific heat ratios can be accurately represented by an average value for each component or groups of components.

Furthermore, the gas composition is assumed to remain constant, whereas the molar mass of the working fluid in fact changes with the addition of fuel in the burner and afterburner.

Adiabatic engine components

The engine components, especially the compressor and turbine are assumed to be adiabatic [9]. In modern engines, however, turbine blade cooling may result in heat transfer in the turbine [7]. Air cooling of the afterburner stage by the bypass stream might also be present [4]. Other components might also utilise air-cooling.

4.4.2 Calculation of the thrust per unit airflow into the gas generator

From manipulation of the momentum and energy laws [9], Eq. 4.1 can be derived:

\[ F' = \dot{m}_a u_a + \dot{m}_0 a_1 (p_2 - p_0) + \dot{m}_a u_a + \dot{m}_e (p_2 - p_0) \]  \hspace{1cm} Eq. 4.1

Let:

\[ \dot{m}_0 - \text{mass flow rate into gas generator, i.e. mass flow rate at station 3.} \]  \hspace{1cm} Eq. 4.2

\[ f = \frac{\dot{m}_o}{\dot{m}_a} = \text{fuel/air ratio} \]  \hspace{1cm} Eq. 4.3

\[ \beta = \frac{\dot{m}_o}{\dot{m}_e} = \text{bypass ratio} \]  \hspace{1cm} Eq. 4.3
This gives:

\[
\dot{m}_i = \dot{m}_f + \dot{m}_e = f \dot{m}_a + m_e = \dot{m}_i (1 + f) \tag{Eq. 4.4}
\]

\[
\dot{m}_a = \beta \dot{m}_i \tag{Eq. 4.5}
\]

\[
\dot{m} = \dot{m}_e + \dot{m}_a = \dot{m}_i (1 + \beta) \tag{Eq. 4.6}
\]

Also,

\[
\dot{m}_i = \rho_i A_i \dot{u}_i \tag{Eq. 4.7}
\]

Using the substitution \( \rho = \frac{p}{RT} \), Eq. 4.7 can be rewritten as:

\[
A_i = (1 + f) \frac{1}{\rho_i \dot{u}_i} \tag{Eq. 4.8}
\]

\[
\dot{m}_a = (1 + f) \frac{\rho_0 \dot{u}_a}{\rho_i \dot{u}_i} \frac{1}{\rho_i \dot{u}_i} = (1 + f) \frac{1}{\rho_0 \dot{u}_a} \frac{p_0 T_0}{p_i T_i} R \dot{u}_e
\]

where

- \( R_i \) = gas constant for combustion products
- \( R_a \) = gas constant for incoming air

Similarly for the bypass stream,

\[
\dot{m}_a = \rho_a A_a \dot{u}_a
\]

\[
A_a = \beta \frac{1}{\rho_a \dot{u}_a} \tag{Eq. 4.9}
\]

\[
\dot{m}_e = \beta \frac{1}{\rho_a \dot{u}_a} \frac{p_0 T_0}{p_i T_i} \dot{u}_e
\]

Rewriting Eq. 4.1, making substitutions with Eq. 4.2 to Eq. 4.6, an expression for the thrust per unit airflow into the gas generator is obtained.

\[
\frac{\Gamma}{\dot{m}_a} = \frac{\dot{m}_i \dot{u}_i - \dot{m}_i \dot{u}_0 + A_e (p_i - p_e) + \dot{m}_e \dot{u}_0 + A_e (p_i - p_e)}{(1 + f) \dot{u}_i - (1 + \beta) \dot{u}_0 + \beta \dot{u}_e + A_e (p_i - p_e)} \tag{Eq. 4.10}
\]
The ratios $A_e/\dot{m}_e$ and $A_e/\dot{m}_a$ are readily determined from Eq. 4.8 and Eq. 4.9 once the exit state of the fluid has been determined. In order to determine these unknown quantities, the energy balance in the engine will be considered, with reference to the $T$-$s$ diagram given in Figure 4.2.

From compressible flow relations,

$$\theta_0 = \frac{T_{e0}}{T_{e1}} = 1 + \frac{1}{2} (1 - 1) M_0^2$$  \hspace{1cm} \text{Eq. 4.11}

$$\delta_0 = \frac{P_{e0}}{P_{e1}} = 1 + \frac{1}{2} (1 - 1) M_0^2 \frac{T_{e1}}{T_{e0}}$$  \hspace{1cm} \text{Eq. 4.12}

The compressor pressure ratio and efficiency are known, so from the definition of the compressor efficiency:

$$\eta_c = \frac{h_{e2} - h_{e1}}{h_{e2} - h_{e1}}$$  \hspace{1cm} \text{Eq. 4.13}

$$C_x T_{e1} \left[ \frac{p_{e1}}{p_{e2}} \right]^{\gamma - 1} - 1 \right]$$

$$= \frac{\tau_{e1}^{(\gamma - 1) \eta_c} - 1}{(\gamma - 1)}$$

Rearranging this, an expression for the temperature ratio across the compressor is obtained:

$$\tau_{e2} = \frac{T_{e1}}{T_{e2}} = \left[ \tau_{e1}^{(\gamma - 1) \eta_c} - 1 \right] / \eta_c + 1$$  \hspace{1cm} \text{Eq. 4.14}

The total temperature ratio across the diffuser is unity (adiabatic diffuser assumption), and thus $T_{d0}$ may be found from:

$$T_{d0} = \frac{T_{e1}}{T_{e2}} \frac{T_{d1}}{T_{d0}} \frac{T_{d1}}{T_{d0}}$$

$$= \tau_{e2} T_{e2}$$

The maximum turbine inlet temperature, $T_{in}$, is known (determined by material limitations). The average temperature in the burner, as given by

$$\overline{T_b} = \frac{1}{2} (T_{in} + T_{in})$$

is used to determine an average specific heat for the burner, i.e. $C_{pb} = C_{pb}(T_b)$. 
From the definition of burner efficiency [7],

\[
\eta_b = \frac{C_p \left[ (m_a + m_{i,2}) T_{i,2} - \dot{m}_a T_{i,1} \right]}{\dot{m}_a Q_R} = \frac{C_p \left[ (1 + f_i) T_{i,2} - T_{i,1} \right]}{f_i Q_R}
\]

Rearranging this equation, an expression for the air/fuel ratio can be determined:

\[
f_i = \frac{C_p \left( T_{i,2} - T_{i,1} \right)}{Q_R \eta_b - C_p T_{i,2}}
\]

Eq. 4.15

Now,

\[
T_s = \frac{T_{i,2}}{T_{i,1}}
\]

Eq. 4.16

Using a similar reasoning as that for \( \tau_c \), Eq. 4.13, an expression for the temperature ratio across the bypass compression stage can be determined:

\[
\tau_f = \frac{T_{c,1}}{T_{c,2}} = \left[ \left( \frac{T_{c,1}}{T_{c,2}} \right)^{\varepsilon_f - 1} \right]^{1/\eta_f} - 1
\]

Eq. 4.17

To determine the temperature ratio across the turbine, the energy balance in the compressor/turbine system is considered:

\[
\dot{m}_a C_v (T_{i,2} - T_{c,1}) + \dot{m}_a C_v (T_{c,3} - T_{c,2}) = \dot{m}_a (1 + f_i) C_v (T_{i,1} - T_{i,2})
\]

\[
\frac{C_p T_{c,1}}{C_v} = \left[ \left( \frac{T_{c,1}}{T_{c,2}} \right)^{\varepsilon_f - 1} \right]^{1/\eta_f} - 1 + f_i \left[ \left( \frac{T_{c,1}}{T_{c,2}} \right)^{\varepsilon_f - 1} \right]^{1/\eta_f} + f_i \left( \frac{T_{c,1}}{T_{c,2}} \right)^{\varepsilon_f - 1} + f_i \left( \frac{T_{c,1}}{T_{c,2}} \right)^{\varepsilon_f - 1} - 1
\]

\[
\frac{C_p}{C_v} = \left( \frac{1 + f_i}{1 - f_i} \right) \left[ \beta (\varepsilon_f - 1)(\varepsilon_f - 1) \right]^{1/\eta_f} + \varepsilon_f (1 - \varepsilon_f)
\]

Eq. 4.18

Solving for \( \varepsilon_f \),

\[
\varepsilon_f = 1 - \frac{C_v}{C_p} \left( \frac{1 + f_i}{1 - f_i} \right) \varepsilon_f (1 - \varepsilon_f)
\]

If the afterburner is operational, then \( T_{i,1} \) will be known from the afterburner performance data (this is again limited by the materials and design of the afterburner). If the afterburner is not operational, then \( T_{i,1} = T_{i,2} \). Since the nozzle is assumed to be adiabatic,

\[
T_{c,1} = T_{i,1}
\]

Eq. 4.19
Analogous to Eq. 4.15, an expression for the fuel/air ratio for the afterburner can be developed:

$$f_a = \frac{C_a (1 + f_a (T_{a} - T_{b}))}{Q_a h_a - C_{T} T_{b}}$$

Eq. 4.20

The total fuel/air ratio is then simply

$$f = f_a + f_e$$

Eq. 4.21

From the definition of turbine efficiency,

$$\eta = \frac{h_{t4} - h_{t1}}{h_{t4} - h_{t3}} = \frac{C_{p}}{C_{T}} \frac{T_{t4} - T_{t1}}{T_{t4} - T_{t3}} = \frac{1 - T_{c}}{T_{t4}} \left[ 1 - \left( \frac{P_{t4}}{P_{t3}} \right)^{\frac{1}{\gamma}} \right]$$

Rearranging,

$$\eta = \frac{P_{t4}}{P_{t3}} = \left[ 1 - \frac{1 - T_{c}}{\eta} \right]^{\frac{1}{\gamma}}$$

Eq. 4.22

The stagnation pressure at station 6 can now be determined from the following identity:

$$P_{6} = \frac{\rho_{6} P_{6} \rho_{6} P_{6} \rho_{6} P_{6} \rho_{6} P_{6} \rho_{6} P_{6}}{\rho_{6} P_{6} \rho_{6} P_{6} \rho_{6} P_{6} \rho_{6} P_{6} \rho_{6} P_{6} \rho_{6} P_{6}}$$

Eq. 4.23

where $\pi_{a}$ and $\pi_{b}$ are the stagnation pressure losses due to heat addition and are assumed or estimated (close to unity). $\pi_{c}$ is calculated from the diffuser pressure recovery.

There are two limiting cases for the exhaust state [7]:

1. The nozzle is fully expanded, and $P_{e} = P_{e}$.
2. The nozzle is choked and $M_{e} = 1$. 
Case 1: Fully expanded nozzle

The enthalpy drop of the exhaust gases at the exit determines the exhaust velocity.

Thus:

\[
\frac{u_2^2}{2} = h_{e2} - h_{e1} = \eta_u C_p \left( T_e - T_i \right) \\
= \eta_u C_p T_e \left[ 1 - \left( \frac{p_i}{p_m} \right)^{\gamma - 1} \right]
\]

or

\[
u_2^2 = \sqrt{\frac{2 \eta_u C_p T_e \left[ 1 - \left( \frac{p_i}{p_m} \right)^{\gamma - 1} \right]}{\gamma - 1}}
\]

Eq. 4.24

\[T_i = T_e - \frac{u_2^2}{2C_{pe}}
\]

Eq. 4.25

\[M_2 = \frac{u_2}{\sqrt{\gamma \rho_2 T_2}}
\]

Eq. 4.26

Case 2: Nozzle choked and \( M_2 = 1 \)

From Eq. 4.26:

\[
u_2^2 = M_2^2 \gamma \rho_2 T_2
\]

From Eq. 4.25:

\[
M_2^2 \gamma \rho_2 T_i = 2C_{pe} \left[ T_i - T_2 \right]
\]

Eq. 4.27

\[T_2 = \frac{2C_{pe} T_i}{\left[ M_2^2 \gamma \rho_2 + 2C_{pe} \right]}
\]

\[u_2 \] can now be found from Eq. 4.25 or Eq. 4.26.

From Eq. 4.24,

\[
p_2 = p_m - \frac{u_2^2}{2 \eta_u C_p T_m}
\]

Eq. 4.28

A similar approach can be adopted to determine the exit condition of the bypass stream.
Case 1: Fully expanded nozzle

Analogous to Eq. 4.24,

\[ u_a = \left[ \frac{\sqrt{2\eta_0 C_P T_{\infty}}}{2\eta_0 C_P T_{\infty}} \left( \frac{p_s}{p_{\infty}} \right)^{(\gamma - 1)/\gamma} \right] \]

Analogous to Eq. 4.25,

\[ T_p = T_{\infty} - \frac{u_a^2}{2C_P} \]

Case 2: Choked bypass stream nozzle, \( M_e = 1 \)

Analogous to Eq. 4.27,

\[ T_p = \frac{2C_P T_{\infty}}{[M_e^2 + R_e + 2C_P]} \]

Analogous to Eq. 4.28,

\[ p_p = p_{\infty} \left[ 1 - \frac{u_a^2}{2\eta_0 C_P T_{\infty}} \right]^{\gamma - 1} \]

4.4.3 Calculation of Net Thrust Output

In section 4.4.2 an expression for the thrust per unit airflow into the gas generator was derived as given by Eq. 4.10. In order to determine the net thrust output, \( F \), the mass flow parameter \( \dot{m}_0 \) must be determined.

The mass flow through the engine is controlled by the following mechanisms:

- **The intake choke the flow**

  In this case, the mass flow rate into the engine is limited by the capture streamtube diameter, free stream density and Mach number. Determination of the mass flow in this condition will be deferred to Section 5.4.

- **The exhaust nozzle choke the flow**

  In this case the mass flow through each nozzle (primary and bypass) is determined by the exit conditions as given in Eq. 4.8 and Eq. 4.9 which provide a relationship between the exit nozzle areas and \( \dot{m}_0 \). A maximum nozzle area is specified, and it is assumed that this area is variable between this maximum and some smaller value. The maximum mass flow rate for a given thermodynamic exit condition can therefore be calculated.
• The engine chokes the flow

The engine components such as the compressor, burner and turbine are capable of passing certain maximum mass flow rate. This flow rate is given as a specification by the manufacturer.

• The compressor 'suction' at low free stream velocities

When the engine is stationary, the mass flow as given by the product of the capture streamtube diameter and forward velocity is zero. The compressor, however, acts as a 'pump' in this condition and a certain mass flow is achieved.

The mass flow through the engine is therefore determined in the following manner:

• Subsonic Flight Regime:

\[
\begin{align*}
\text{if } \dot{m}_{\text{diff}} < \dot{m}_{\text{prop}} \text{ then } \dot{m} &= \dot{m}_{\text{prop}} \\
\text{else } \dot{m} &= \dot{m}_{\text{diff}} \\
\text{if } \dot{m} > \dot{m}_{\text{inj}} \text{ then } \dot{m} &= \dot{m}_{\text{inj}} \\
\text{if } \dot{m} > \dot{m}_{\text{max}} \text{ then } \dot{m} &= \dot{m}_{\text{max}}
\end{align*}
\]

where,

- \( \dot{m}_{\text{diff}} \) = maximum flow through the diffuser
- \( \dot{m}_{\text{prop}} \) = mass flow that the engine is capable of 'pumping'
- \( \dot{m}_{\text{inj}} \) = maximum mass flow through the exhaust
- \( \dot{m}_{\text{max}} \) = maximum mass flow through the compressor & turbine

• Supersonic Flight Regime:

In this case, the lesser of \( \dot{m}_{\text{diff}} \), \( \dot{m}_{\text{inj}} \), \( \dot{m}_{\text{max}} \) determines the total net flow through the engine, and hence the resultant thrust.

4.5 Engine Performance Indicators

4.5.1 Thrust Specific Fuel Consumption (TSFC) and Specific Impulse

A frequently used measure of engine performance is the thrust specific fuel consumption (TSFC) value. This value indicates the rate at which fuel is used for every unit of thrust produced. Thus,
\[
\text{TSFC} = \frac{\dot{m}_i}{f} \left( \frac{F}{m_i} \right)^f
\]

Eq. 4.33

This parameter is easily calculated once Eq. 4.19 and Eq. 4.21 have been evaluated.

Specific impulse is the inverse of TSFC.

### 4.5.2 T-s Diagram Calculations

The thermodynamic cycle of a turbojet engine can be plotted on a T-s diagram such as Figure 4.2. After the temperatures and pressures at the various engine stations have been calculated, the entropy generation of each stage can be evaluated. In order to plot the T-s diagram, the entropy at each station was calculated as the sum of the entropy at the previous station and the entropy change across the relevant component:

\[
s_i = s_{i-1} + C_p \ln \left( \frac{T_i}{T_{i-1}} \right) - R \ln \left( \frac{p_i}{p_{i-1}} \right)
\]

Eq. 4.34

The isobaric lines on the T-s diagram can be plotted from Eq. 4.34, by setting \( p_i = p_{i-1} \). This can then be rearranged as follows:

\[
T = T_{i-1}e^{\frac{s_i - s_{i-1}}{C_p}}
\]

Eq. 4.35

The isobaric line passing through a particular station point can be plotted using Eq. 4.35 by choosing \( s_{i-1} \) and \( T_{i-1} \) to be the conditions at the station point, and then plotting \( T \) as a function of \( s \).

### 4.6 Factors Affecting Engine Performance

This section provides the results of a brief parametric investigation into the factors affecting the overall performance of the engine.

#### 4.6.1 Reference configuration for the parametric investigation

Table 4.2 provides a list of the engine parameters and their values utilized for the parametric investigation. These values are largely based upon the SNFCMA 9k50 engine which will be discussed in more depth in Chapter 7.
### Table 4.2: Parameter values for the reference configuration

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Symbol</th>
<th>Value</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Altitude</td>
<td></td>
<td>0</td>
<td>m</td>
</tr>
<tr>
<td>Pressure ratios</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fan</td>
<td>$\pi_f$</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Compressor</td>
<td>$\pi_c$</td>
<td>6.18</td>
<td></td>
</tr>
<tr>
<td>Burner</td>
<td>$\pi_b$</td>
<td>0.97</td>
<td></td>
</tr>
<tr>
<td>Afterburner</td>
<td>$\pi_a$</td>
<td>0.98</td>
<td></td>
</tr>
<tr>
<td>Diffuser</td>
<td>$\pi_d$</td>
<td>AIA Curve</td>
<td></td>
</tr>
<tr>
<td>Bypass Ratio</td>
<td>$\beta$</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Efficiencies</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fan</td>
<td>$\eta_f$</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>Compressor</td>
<td>$\eta_c$</td>
<td>0.85</td>
<td></td>
</tr>
<tr>
<td>Burner</td>
<td>$\eta_b$</td>
<td>0.96</td>
<td></td>
</tr>
<tr>
<td>Turbine</td>
<td>$\eta_t$</td>
<td>0.90</td>
<td></td>
</tr>
<tr>
<td>Afterburner</td>
<td>$\eta_a$</td>
<td>0.90</td>
<td></td>
</tr>
<tr>
<td>Primary Nozzle</td>
<td>$\eta_{no}$</td>
<td>0.97</td>
<td></td>
</tr>
<tr>
<td>Bypass Nozzle</td>
<td>$\eta_{bo}$</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>Temperatures</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Turbine Inlet Temperature</td>
<td>$T_{T_n}$</td>
<td>1203</td>
<td>K</td>
</tr>
<tr>
<td>Maximum Afterburner Temperature</td>
<td>$T_{Ba}$</td>
<td>2500</td>
<td>K</td>
</tr>
<tr>
<td>Heating value of fuel</td>
<td>$Q_h$</td>
<td>42,961.4</td>
<td>kJ/kg</td>
</tr>
<tr>
<td>Primary Nozzle Type</td>
<td></td>
<td>Convergent</td>
<td></td>
</tr>
<tr>
<td>Bypass Nozzle Type</td>
<td></td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>
4.6.2 Turbine Inlet Temperature

Figure 4.3: Variation of TSFC and $F/m_a$ with turbine inlet temperature.

Figure 4.3 shows that both $F/m_a$ and TSFC increase with $T_{in}$. $F/m_a$ however, increases at a greater rate than TSFC and thus higher turbine temperatures provide better engine performance.

4.6.3 Compressor Pressure Ratio

Figure 4.4: Variation of TSFC and $F/m_a$ with compressor pressure ratio.
Figure 4.4 shows that an optimal compressor pressure ratio exists for a given engine. \( F/\dot{m}_a \) at first increases due to the gains from the higher compressor exit pressure, but then decreases as more energy is removed from the working fluid in order to power the compressor.

TSCF shows an overall decrease with an increase in the compressor pressure ratio, and this would be beneficial for extending the range of the aircraft.

The trends indicated in Figure 4.4 are similar to those found in Fig 6-14 of Hill and Petersen [9]. In order to check the model for errors, the parameters for Hill and Petersen’s Fig 6-14 was entered into the propulsion model and closely matching results were obtained.

4.6.4 Intake Pressure Recovery

![Graph showing variation of TSFC and F/ma with diffuser pressure recovery.](image)

Figure 4.5: Variation of TSFC and \( F/\dot{m}_a \) with diffuser pressure recovery.

An increase in the diffuser pressure recovery, \( \pi_d \), is seen to increase \( F/\dot{m}_a \) and decrease TSCF. An improvement in intake performance therefore benefits the overall performance of the engine and it is for this reason that intake performance is such a critical part of the propulsion system and must be designed correctly. The intake will be discussed in more detail in Chapter 5.
4.7 Software Implementation of the Propulsion Model

The one-dimensional propulsion model described in this chapter was coded into a C++ class representing the entire turbojet. The source code for this class, "Turbojet1D", can be found in Appendix C-1.

A flow chart indicating the algorithm for determining the net thrust output can be found in Appendix C-1.

The class stores all the data for component efficiencies, maximum temperatures, pressure ratios, fuel energy, etc. i.e. all the data required to represent the turbojet. The Turbojet1D class also contains a CConeIntake object that describes the intake geometry and behaviour. The CConeIntake class will be discussed in Section 5.5.

The primary function in this class is CycleThermodynamics, and its parameters are given in Table 4.3. This function is called by the TSFC and Thrust functions to calculate the overall thermodynamics of the cycle. It follows the procedures, and utilises the equations, described in this chapter to perform its calculations.

Table 4.3: Parameters for Function Cycle Thermodynamics

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Units</th>
<th>In/Out</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Value:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSFC - Thrust Specific Fuel Consumption</td>
<td>kg/(N.s)</td>
<td>Out</td>
<td>Double precision</td>
</tr>
<tr>
<td>Arguments:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M0 - Free stream Mach number</td>
<td>In</td>
<td>Double precision</td>
<td></td>
</tr>
<tr>
<td>T0 - Free stream temperature</td>
<td>K</td>
<td>In</td>
<td>Double precision</td>
</tr>
<tr>
<td>p0 - Free stream pressure</td>
<td>Pa</td>
<td>In</td>
<td></td>
</tr>
<tr>
<td>ABONOff - Afterburner: On = 1, Off = 0</td>
<td>In</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>pi_d - diffuser pressure recovery</td>
<td>In</td>
<td>Double precision</td>
<td></td>
</tr>
<tr>
<td>ex7 - exit stream properties at station 7</td>
<td>Out</td>
<td>IdealGasStream</td>
<td></td>
</tr>
<tr>
<td>ex9 - exit stream properties at station 9</td>
<td>Out</td>
<td>IdealGasStream</td>
<td></td>
</tr>
<tr>
<td>A7_ma - exit area to mass flow rate</td>
<td>m²/s/kg</td>
<td>Out</td>
<td>Double precision</td>
</tr>
<tr>
<td>A9_ma - exit area to mass flow rate</td>
<td>m²/s/kg</td>
<td>Out</td>
<td>Double precision</td>
</tr>
<tr>
<td>F_t - Thrust to mass flow ratio</td>
<td>N s/kg</td>
<td>Out</td>
<td>Double precision</td>
</tr>
<tr>
<td>f - fuel/air ratio</td>
<td>Out</td>
<td>Double precision</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 5: Intake Modelling

In Chapter 4 it was shown that the intake performance is critical to the overall performance of the propulsion unit in terms of its pressure recovery and the mass flow that it permits. This section describes the manner in which the conical-centrebody, supersonic intake is modelled. It starts with an overview of intake operation modes and proceeds to describe how these are mathematically modelled. A method for determining the mass flow through the intake is also presented.

5.1 Overview of intake operation

Figure 5.1 shows a schematic of a typical supersonic, conical centrebody intake. The supersonic free stream airflow over the cone generates a conical shock wave. A second, normal, shock then occurs at the cowl lip, or in the diverging passage. This two-shock system, discussed in more detail later, results in a favourable stagnation pressure ratio from the free stream to the compressor face [10].

The centrebody can translate along the X-axis, in order to vary the design angle, $\beta_d$.

![Schematic of a supersonic conical intake with two-shock system](image)

Figure 5.1: Schematic of a supersonic conical intake with two-shock system

A conical centrebody, supersonic intake, operates in one of the following modes, dependent on flight condition and turbomachine performance [10, 11]:

5.1.1 Subsonic flight

This mode of operation is effective in the free stream Mach number range from $0 \leq M_0 \leq 1$. The stagnation pressure recovery, $\chi_d$ is nominally constant [10] and close to unity.
5.1.2 Detached Shock Mode

A detached bow shock is formed in the Mach number range from \( 1 \leq M_e < M_{crit} \) where \( M_{crit} \) is the Mach number at which a shock wave attaches to the conical centred body. This shock wave has a hyperbolic geometry and asymptotes towards the some (Mach) angle \( \mu \).

![Detached shock arrangement](image)

**Figure 5.2: Detached shock arrangement**

5.1.3 Critical Mode

Critical mode pressure recovery occurs when the intake is operating at its design condition. Therefore, the Mach number is given by the design Mach number \( M_d \) which is variable and dependent upon the centred body position.

The back pressure on the intake at the compressor face is such that the normal shock occurs at the cowl lip. This results in the capture stream tube having the same diameter as the cowl lip and the maximum possible mass flow rate through the intake is achieved.

![Critical mode shock arrangement](image)

**Figure 5.3: Critical mode shock arrangement**
5.1.4 Sub Critical Mode

Sub critical mode operation occurs when the back pressure on the intake is too high. This results in the normal shock being pushed forward, outside the cowl. Part of the capture streamtube therefore passes through the two-shock system and the rest only passes through a stronger, single shock. The stagnation pressure recovery is therefore lower than at critical operation for the same Mach number. In addition, streamlines passing through the single shock are curved, and the capture streamtube diameter is therefore smaller than in critical mode operation, resulting in a lower mass flow rate through the intake.

![Two flow arrangements in sub critical mode](image)

Figure 5.4: Two flow arrangements in sub critical mode

5.1.5 Super Critical Mode

In super critical mode, the back pressure acting on the intake is lower than in critical mode operation. The normal shock therefore moves to a location further down the diverging passage inside the cowl, as illustrated in Figure 5.5. The Mach number at this location is higher than at the cowl lip and the normal shock is therefore stronger. The overall pressure recovery is therefore lower than at the critical condition. The mass flow rate through the intake remains constant and equal to the mass flow rate obtained at the critical condition.
Chapter 5: Intake Modeling

5.1.6 Mass flow and pressure recovery for different operating modes

Figure 5.5: Super critical mode shock arrangement

Figure 5.6: Mass flow and Pressure Recovery for different operating modes. Source: [10]

Figure 5.6 illustrates the various operating modes that have been described in this section. The effect of subcritical and supercritical operation on the total pressure recovery is clearly illustrated. The reduced mass flow due to operation at an off-design condition is also indicated.

For this reason, the design of modern intakes is extremely complicated. Generally, facilities are provided for bleeding excess intake mass flow when the engine is choked, so as to avoid a subcritical operational mode. Additional intake gates are provided for engine starting and low Mach number operation such as to allow sufficient mass flow through the intake [10, 5].

Determination of the exact mass flow through the engine is therefore difficult to calculate, and so simplifications and assumptions regarding the intake behaviour have been incorporated into the model developed in this section to estimate the actual mass flow rate.
5.2 Conical Shock Properties

In order to develop a numerical model for a conical intake, some understanding of the behaviour of the conical shock is required. This section will provide a very brief overview of the key properties of this shock type.

When a supersonic stream encounters a wedge, an oblique shock wave forms and the stream is diverted such that the streamlines are parallel to the wedge disturbance. In the case of a conical body in the supersonic stream, the flow around the cone is axisymmetric. From the conservation of mass principle as applied to the fluid between streamlines, the streamlines converge towards the cone surface as the their radius increases. This situation is shown in Figure 5.7.

Due to the curvature of the streamlines, compression of the gas takes place between the shock wave and the cone surface. Thermodynamic properties are therefore not constant along a streamline, but are instead constant along rays emanating from the cone apex.

The shock angle is dependent on the free stream Mach number and the cone semi-angle. Unlike the simple oblique shock, the shock angle cannot be determined from a simple gas relationship. Instead, it requires an iterative procedure such as that described in Section 16.5 of reference 13, based on the work of Taylor & Macoll. This flow pattern is termed Taylor-Macoll flow.

Once the shock angle for a given cone and free stream Mach number have been determined, the properties behind the shock are readily calculated from the standard ideal gas shock relationships. Furthermore, the properties along any ray can be determined by performing the required numerical integration between the shock and the ray in question.

In order to provide data for the behaviour of a conical shock system, a computer program was written based on the algorithm presented in reference 13. It utilises the Taylor-Macoll approach to solve for the shock angles corresponding to a range of Mach numbers at a given free stream temperature. A comparison of the output of this program to the data published in reference 12 is given in Figure 5.8. This shows that the calculated output matches the published data very well. The gaps in the curves for the 10° and 20° cone semi-angle are due to the algorithm not converging within the specified convergence period for that particular data point. This missing data is readily calculated by increasing the maximum number of iterations the software is allowed to perform.
Figure 5.8: Comparison between published conical shock wave data and calculated results.

The code for program which calculates these properties is presented in Appendix E-1.

A sample output file from the program for a 30° cone semi-angle is presented in Appendix A-3.

The calculation of properties along any ray was included in the ideal gas utility code which is presented in Appendix E-3.
5.3 Standard intake performance curves

Due to the complex nature of intake performance, several standard intake pressure recovery curves have been developed. These allow a quick estimate of how an intake should behave at different Mach numbers. However, they do not predict the mass flow that the intake will be capable of passing. As discussed earlier, the mass flow rate through the intake is affected by operational mode and is critical to determining the net thrust output of the engine. Two standard curves [11] are presented below for comparison with the intake model developed in this chapter.

AIA Pressure Recovery Curve

The Aircraft Industries Association (AIA) standard pressure recovery curve approximates the subsonic recovery as unity, and utilises Eq. 5.1 for the supersonic region.

\[
\pi_d = \begin{cases} 
1 & 0 < M_a \leq 1 \\
1 - 0.1(M_a - 1)^{1.5} & M_a > 1 
\end{cases}
\]

Eq. 5.1

MIL-E-5008B curve

This curve represents a revision of the AIA curve, intended to be somewhat more indicative of actual intake performance. The curve is intended to provide a guide to good, optimum intake performance behaviour. In the subsonic region, the recovery is unity, and Eq. 5.2 applies in the supersonic region.

\[
\pi_d = 1 - 0.0075(M_a - 1)^{1.5} \quad 1 < M_a < 5
\]

Eq. 5.2

Figure 5.9: Standard pressure recovery curves
5.4 Numerical estimation of intake performance

The equations and algorithms utilised in the numerical intake performance model are described below. Detailed derivations of some of the equations have been placed in appendices for further reference. A comparison between the performance predicted by the algorithms presented in this section and the standard curves is given in Section 7.2.

5.4.1 Detached Shock Solution

Figure 5.10 below shows a schematic diagram of a detached shock system upstream from a conical body. The shock has a hyperbolic geometry [10] which asymptotes towards the Mach angle, \( \mu \).

![Schematic representation of a detached shock and its discretisation](image)

Figure 5.10: Schematic representation of a detached shock and its discretisation

At the apex of the hyperbola, the shock properties are similar to those of a normal shock [12]. Towards the outer reaches, the shock behaves as a Mach shock. Between these two positions, the properties behind the shock can be estimated from the oblique shock relations.

The solution algorithm utilises the assumption that the shock angle at the capture radius, \( \theta_c \), is a linear function of Mach number between a normal shock (90°) and the first attached shock (\( \beta \) at \( M_{\text{max}} \)). In reality, the shock asymptotes to the Mach angle at very large distances from the apex of the cone.

The calculation of the parameters describing the hyperbola is given in Appendix A-2.

Pressure recovery is calculated by dividing the capture streamtube into annuli of equal area as given by Eq. 5.3. The flow properties before and after the oblique shock are calculated for each section, and the results are averaged to yield the pressure recovery for the capture streamtube.

\[
A_c = \pi (r_a^2 - r_c^2)
\]

Eq. 5.3
where \( r_n \) is given by:

\[
r_n = r_i \sqrt{\frac{n}{m}}
\]

and where:
- \( A_n \) = area of annulus \( n \)
- \( m \) = number of annuli
- \( r_n - r_i \) = streamtube capture radius
- \( r_n, r_{in} \) are the inner and outer radius of annulus \( n \)

Thus the pressure recovery is given by Eq. 5.4 below:

\[
\frac{p_A}{p_{in}} = \frac{1}{\pi} \sum_{n=1}^{m} \left( \frac{p_n}{p_{in}} \right)
\]

Eq. 5.4

Mass flow rate in this condition is assumed to be governed by the engine and exhaust.

5.4.2 Critical Solution

![Figure 5.11: Pressure recovery in critical mode operation](image)

Figure 5.11 shows the pressures at various locations in the intake shock system during critical mode operation. The solution for the stagnation pressure \( p_{st} \) is as follows:

- The conical shock wave angle \( \beta \) is determined from previously calculated Taylor-Macoll conical flow properties, as described in Section 5.2.
- The algorithm determines whether or not the conical centrebody can be positioned such that \( \beta_0 \) matches \( \beta \). If this is not possible, \( \beta_0 \) is adjusted to a value that is as close to \( \beta \) as possible.
- The average Mach number of the airstream entering the cowl is determined as the average of the Mach number at the cowl lip, \( M_{lp} \) and the cone surface, \( M_c \). If \( \beta_0 \) matches \( \beta \), then \( M_{lp} \) is given by the Mach number immediately after the conical shock. However, if \( \beta_0 \) is less than \( \beta \), \( M_{lp} \) must be determined by integrating the Taylor-Macoll flow equations between \( \beta \) and \( \beta_0 \).
The pressure recovery across the conical shock is calculated from Eq. 5.5:

\[ P_n = \frac{P_{s2}}{P_n} \frac{P_s}{P_n} \]

\[ \frac{M_{\text{avg}}}{2} = \frac{M_{\text{avg}}}{2} \]

- The average Mach number of the stream entering the cowl is then calculated Eq. 5.6. If \( M_{\text{avg}} < 1 \), the stream entering the intake is subsonic, and there will be no normal shock. It is possible that \( M_{\text{avg}} > 1 \) while \( M_{\text{avg}} < 1 \), however, in this case, \( M_{\text{avg}} \) will be close to unity and the pressure ratio across the shock would be approximately unity as well.

- If \( M_{\text{avg}} > 1 \), a normal shock occurs and the pressure ratio across the shock is calculated from the normal shock equations.

The total pressure recovery for the intake is therefore given by Eq. 5.7:

\[ \pi = \frac{P_n}{P_0} \frac{P_0}{P_0} \frac{P_0}{P_0} \]

The maximum mass flow rate that the intake is capable of passing is calculated by considering the position of the shock system, which determines the maximum capture streamtube diameter.

**Figure 5.12: Parameters for determining the mass flow through the intake**

Figure 5.12 illustrates the shock arrangement and flow streamline of the actual capture streamtube. \( r_o(x) \). The conical shock (a) is described by Eq. 5.8 while the average streamline (b) is described by Eq. 5.9.

\[ r_o(x) = x \tan \beta \]

\[ r_o(x) = r_o + (x - L_o) \tan \theta \]

Solving for the intersection of these lines yields Eq. 5.10 below:

\[ r_o = r_o - L_o \tan \theta \]

\[ \frac{1}{\tan \beta} \]
The only unknown on the right hand side in Eq. 5.10 is \( \theta \), the flow angle. \( \theta \) represents the average flow direction of a streamline after the conical shock. This streamline curves, starting at the flow direction given by the oblique shock equations, and approaching the slope of the cone as it nears the cone surface. The average flow angle is given by the average of the flow angle immediately after the shock and the flow angle at the intake lip. The flow angle at the lip can be calculated by integrating the Taylor-Macoll flow equations.

The maximum flow possible through the intake is then given by Eq. 5.11.

\[
m_{\text{in}} = \rho \cdot A_0 \cdot v_0
= \rho \cdot m_0^2 \cdot M_0 \cdot \sqrt{\gamma \cdot R \cdot T_0}
\]

Eq. 5.11

5.4.3 Sub Critical Solution

When the maximum flow possible through the engine is less than the maximum flow through the intake operating in critical mode, as given by Eq. 5.11, the engine appears as a high back pressure to the intake, and it will operate in sub critical mode as discussed in Section 5.1.4. The engine effectively acts as a choke, and the mass flow rate through the intake is limited to the maximum flow through the engine. The maximum flow in turn determines the capture streamtube diameter.

The shock arrangement is illustrated in Figure 5.4. The pressure recovery is therefore dependent on the location of the normal shock as this determines what proportion of the capture streamtube passes through two shocks, and what passes through only the normal shock.

The simulation procedure utilized assumes that for zero mass flow through the engine, the normal shock is located at the apex of the conical centbody. The position of the normal shock for a given flow ratio is then interpolated between the normal shock placed at the cowl lip (flow ratio = 1) and the cone apex (flow ratio = 0).

5.4.4 Super Critical Solution

In super critical mode, the mass flow is limited to the maximum mass flow obtained in critical mode, as discussed in Section 5.1.5 above. The pressure recovery in this case depends on the Mach number at which the normal shock occurs, which in turn is dependent upon the throat area and the pressure at the compressor face.
5.5 Software implementation of the Intake Model

The methods, procedures and equations of this chapter were coded into a C++ class called "ConeIntake". This class has data members which represent the geometry and fluid properties which affect the performance of the intake. It also has member functions which provide pressure recovery calculations in the various operating modes described in the preceding sections.

A flow chart representing the procedure utilised for determining the pressure recovery of the intake is presented in Appendix A-1.

Conical shock properties are provided in a lookup table which can be changed at run-time to allow the intake to be used at different attitudes within the same simulation. The conical shock properties are generated by a standalone program "TKEFlowProps" which implements the Taylor-Macoll solution [13] for conical shock properties. This was done due to the iterative nature of the conical shock solution algorithm. Conical shock properties along rays at angles other than the shock angle are then easily calculated by Runge-Kutta integration from the previously calculated shock angle and properties. The functions for determining these properties are included in the IdealGas class which encapsulates the behaviour of an ideal gas.

The code for the ConeIntake class is provided in Appendix A-1.

The code for the IdealGas class is provided in Appendix E-3.
Chapter 6: Aerodynamic Model

This chapter describes the manner in which the aerodynamic model is implemented in the simulation. It covers some basic aerodynamic concepts and then proceeds to discuss the development of the Digital DATCOM model used, and how the output from this model was processed for use in the overall simulation.

6.1 Overview of Basic Aerodynamics

Figure 6.1 illustrates the basic forces acting on a wing section [14]. The nomenclature is as follows:

- $V_c =$ free stream velocity
- $a =$ angle of attack
- $c =$ chord length
- $R =$ resultant aerodynamic force
- $N =$ component of $R$ perpendicular to $c$
- $A =$ component of $R$ parallel to $c$
- $L =$ Lift - the component of $R$ perpendicular to $V_c$
- $D =$ Drag - component of $R$ parallel to $V_c$

By defining a dynamic pressure, $q_c$, as given by Eq. 6.1, and dividing the relevant force (e.g. $L$) by $q_c$ and $S$, the reference area, a dimensionless coefficient is obtained. Dimensionless coefficients of this form are a more fundamental indicator of the aerodynamic properties of a wing than the actual forces themselves and allow comparison between different sizes of wings and their behaviour at different altitudes.

$$q_c = \frac{1}{2} \rho_c V_c^2$$

Eq. 6.1
6.1.1 Subsonic flight and wing sections

In the subsonic flight regime, a typical subsonic airfoil behaves as shown in Figure 6.2. The variation of $C_L$ with $\alpha$ is largely linear up to the point where flow separation begins. This is the point at which the maximum lift $C_{L\text{max}}$ is achieved. Beyond this point, the lift coefficient decreases as the wing stalls.

By defining a lift slope as given by Eq. 6.2, the lift coefficient can be easily calculated up to $C_{L\text{max}}$. This approach, however, does not allow the value of $C_{L\text{max}}$ to be determined.

$$C_{L_{\alpha}} = \frac{dC_L}{d\alpha}$$

Eq. 6.2

The variation of drag with angle of attack is schematically shown in Figure 6.3. For a general wing, this can be estimated by Eq. 6.3 [14]:

$$C_{D_{\alpha}} = C_{D_{\text{in}}} + \frac{C_{L_{\text{in}}}}{\kappa \cdot AR \cdot e}$$

Eq. 6.3

where,

$AR = \text{aspect ratio of the wing (b/5)}$

$e = \text{span efficiency factor}$
Figure 6.3: Idealised variation of drag coefficient with angle of attack.

The lift-to-drag ratio is an indicator of how efficiently a wing generates lift. Generally, the higher the lift to drag ratio of an aircraft, the better its performance. Using Eq. 6.3, an idealised curve of lift-to-drag vs. lift can be plotted, as shown in Figure 6.4.

Figure 6.4: Variation of Lift-to-Drag ratio with \( C_L \)

Although Eq. 6.3 predicts a gradually decreasing \( C_L/C_D \) with an infinitely increasing value of \( C_L \), this does not occur in reality. As the angle of attack increases, a condition where the flow separates from the wing occurs and the wing stalls, as indicated in Figure 6.2. The value of \( C_L \) therefore decreases. This results in the curve shown in Figure 6.4 curving back towards zero. For delta wing aircraft, this stall condition occurs at high angles of attack.
6.1.2 Variation of Drag with Mach number

Figure 6.5 shows a schematic of the variation of drag coefficient $C_{D0}$ with Mach number. It shows that $C_{D0}$ is nominally constant until some critical Mach number $M_{cr}$ after which it starts to increase, reaching a maximum value at sonic speed. Thereafter, the value decreases. One of the factors influencing the ratio of $C_{D\text{max}}$ to $C_{D0}$ is the so-called 'area rule' [14]. The application of the area rule results in the waisted appearance of many combat aircraft.

6.2 Digital DATCOM model

The Digital DATCOM [15, 16] was used to provide estimates of the aerodynamic coefficients of a hypothetical delta-winged aircraft loosely based on the Dassault Mirage III. The following sections describe how the aircraft was idealized for modelling with the DATCOM. Some of the limitations of the model are discussed along with some results.

6.2.1 Describing the airframe

The airframe model was generally based on the Dassault Mirage III, which was flown in the 1960's to 1980's by many of the world's airforces. The approximate model is shown in Figure 6.6, with the dimensions used. A more detailed set of drawings with dimensions is available in Appendix B-1. The modelling of various aspects of the airframe such as the fuselage, wings, and tail section are described in more detail in the sections to follow.
Figure 6.6: Schematic aircraft model for use in the Digital DATCOM

**Body**

The body was modelled by providing the Digital DATCOM with four parameter arrays. These are x-coordinate, cross-section half-width, cross-sectional area and perimeter as indicated in Figure 6.6. Data for these was obtained from scaled drawings from Jane's [3] and a 1/48th-scale model of a Mirage IIIIC.

Centre of mass was placed such that it was within the aircraft's wheelbase, and on the aircraft centreline, even though it probably lies somewhat lower when taking into account the position of the wings, and any armaments.

**Wing**

*Planform Geometry*

Planform geometry is that of a swept-back tapered wing, resulting in a delta shape. Dimensions were specified such that the planform would be similar to the Mirage III, dimensions of which were obtained from drawings from Jane's [3].

Dimensions are given in Table 6.1:
Table 6.1: DATCOM parameters for the wing planform

<table>
<thead>
<tr>
<th>Description</th>
<th>Notes</th>
<th>Eng. Symbol</th>
<th>DATCOM Name/Id</th>
<th>DATCOM Variable</th>
<th>Value</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Longitudinal location of wing apex</td>
<td>1</td>
<td>SYNTHS</td>
<td>XW</td>
<td></td>
<td>4.36</td>
<td>m</td>
</tr>
<tr>
<td>Vertical location of wing apex</td>
<td></td>
<td>SYNTHS</td>
<td>ZW</td>
<td></td>
<td>0</td>
<td>m</td>
</tr>
<tr>
<td>Wing incidence</td>
<td>2</td>
<td>WGPLNF</td>
<td>AIW</td>
<td></td>
<td>0</td>
<td>Deg</td>
</tr>
<tr>
<td>Tip chord</td>
<td></td>
<td>WGPLNF</td>
<td>CHRDTP</td>
<td></td>
<td>0.47</td>
<td>m</td>
</tr>
<tr>
<td>Root chord</td>
<td></td>
<td>WGPLNF</td>
<td>CHRDTR</td>
<td></td>
<td>8.22</td>
<td>m</td>
</tr>
<tr>
<td>Semi-span</td>
<td>2</td>
<td>WGPLNF</td>
<td>SSPN</td>
<td></td>
<td>4.11</td>
<td>m</td>
</tr>
<tr>
<td>Exposed semi-span</td>
<td></td>
<td>WGPLNF</td>
<td>SSPNE</td>
<td></td>
<td>3.6</td>
<td>m</td>
</tr>
<tr>
<td>Sweepback angle</td>
<td></td>
<td>WGPLNF</td>
<td>SAVSI</td>
<td></td>
<td>60</td>
<td>Deg</td>
</tr>
<tr>
<td>Reference location for sweepback angle</td>
<td></td>
<td>WGPLNF</td>
<td>CSTAT</td>
<td></td>
<td>0</td>
<td>% chord</td>
</tr>
<tr>
<td>Dihedral</td>
<td>3</td>
<td>WGPLNF</td>
<td>DIHADI</td>
<td></td>
<td>0</td>
<td>Deg</td>
</tr>
<tr>
<td>Twist</td>
<td></td>
<td>WGPLNF</td>
<td>TWISTA</td>
<td></td>
<td>0</td>
<td>Deg</td>
</tr>
<tr>
<td>Planform type</td>
<td></td>
<td>WGPLNF</td>
<td>TYPE</td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**
1. Measured from aircraft nose to theoretical vertex of delta.
2. From Jane's [3].
3. Mirage III has 17° anhedral, but this value is only used for lateral coefficient estimation and could therefore be ignored.

**Airfoil Section**

Jane's [3] states that the airfoil section is of conical camber, and varies in thickness from 4.5% to 3.5% chord, root to tip. The airfoil used in the Digital DATCOM model has a 4% maximum thickness at 50% chord, with a maximum camber of 1%.

![Figure 6.7: Airfoil section used for wing specification](image)
Table 6.2: DATCON parameters for modelling the Elanors

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Height</td>
<td>10</td>
<td>Flip nose types — round nose</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Average thickness of the bucanne</td>
</tr>
<tr>
<td></td>
<td>90</td>
<td>Average depth of the bucanne</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>Flip height at maximum depth</td>
</tr>
<tr>
<td>Spangle</td>
<td>5</td>
<td>Flip stern at maximum depth</td>
</tr>
<tr>
<td>Spans</td>
<td>4.05</td>
<td>Flip stern at minimum depth</td>
</tr>
<tr>
<td>Spangle</td>
<td>0.3</td>
<td>Flip stern at minimum depth</td>
</tr>
<tr>
<td>Spans</td>
<td>0.05</td>
<td>Flip stern at minimum depth</td>
</tr>
<tr>
<td>Spangle</td>
<td>6.02</td>
<td>Flip stern at minimum depth</td>
</tr>
<tr>
<td>Spans</td>
<td>6.02</td>
<td>Flip stern at minimum depth</td>
</tr>
<tr>
<td>E-type</td>
<td>2</td>
<td>Flip stern at minimum depth</td>
</tr>
</tbody>
</table>

The table above lists the parameters used in the DATCON model for the Elanor class. Parameters that were determined empirically or from previous experience are marked with an asterisk (*) in the second column.
Table 6.3: DATCOM parameters for modelling the Tail

<table>
<thead>
<tr>
<th>Description</th>
<th>Eng. Symbol</th>
<th>Datcom Namelist</th>
<th>Datcom Variable</th>
<th>Value</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Longitudinal location of tail apex on centreline</td>
<td>SYNTHS</td>
<td></td>
<td>XV</td>
<td>8.74</td>
<td>m</td>
</tr>
<tr>
<td>Vertical tail above reference line?</td>
<td>SYNTHS</td>
<td></td>
<td>VERT1UP</td>
<td>True</td>
<td></td>
</tr>
<tr>
<td>Tip chord</td>
<td>c_t</td>
<td>VTPLNF</td>
<td>CHRD</td>
<td>0.941</td>
<td>m</td>
</tr>
<tr>
<td>Root chord</td>
<td>c_r</td>
<td>VTPLNF</td>
<td>CHRD</td>
<td>5.974</td>
<td>m</td>
</tr>
<tr>
<td>Semi-spar</td>
<td>b</td>
<td>VTPLNF</td>
<td>SSPN</td>
<td>2.649</td>
<td>m</td>
</tr>
<tr>
<td>Exposed semi-spar</td>
<td>VTPLNF</td>
<td>SSSPNE</td>
<td></td>
<td>1.969</td>
<td>m</td>
</tr>
<tr>
<td>Sweepback angle</td>
<td>A</td>
<td>VTPLNF</td>
<td>SAVSI</td>
<td>64</td>
<td>Deg</td>
</tr>
<tr>
<td>Reference location for sweepback angle</td>
<td>VTPLNF</td>
<td>CHSTAT</td>
<td></td>
<td>0</td>
<td>% chord</td>
</tr>
<tr>
<td>Planform type</td>
<td>VTPLNF</td>
<td>TYPE</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Wave drag factor</td>
<td>VTISCHIR</td>
<td>KSHPAR</td>
<td></td>
<td>16/3</td>
<td></td>
</tr>
</tbody>
</table>

6.2.2 Flight conditions

The Digital DATCOM was instructed to produce $C_L$ and $C_D$ data for Mach numbers in the range from 0 to 2.2 and in the altitude range from sea level to 20km.

6.2.3 Limitations of the model

The Digital DATCOM only provides trimmed data for the subsonic flight regime. The assumption was therefore made that the aircraft in question was neutrally stable at all flight conditions. Since the model is required to produce only limited hypothetical aerodynamic data that does not vary for the various simulations, this is a reasonable assumption.

In the transonic region (0.7 < $M_A$ < 1.4), the DATCOM does not provide an estimate of stall behaviour. Instead it utilises a constant value for the lift slope and does not limit the maximum value that $C_L$ achieves. In addition, $C_D$ is not calculated beyond a certain angle of attack, $C_{D_{max}}$, and the missing values of $C_D$ were therefore estimated, as described later.

In general, the output data from the DATCOM was not always smooth, and this is evident in some of the results presented in Chapter 7.
6.3 Results of the DATCOM model

The following section presents selected results from the Digital DATCOM model. Due to the large volume of data generated, it is not possible to present all the results in this report. However, the selection presented below particularly covers the data most relevant for the simulations described in this document.

6.3.1 $C_l$ vs AoA

Figure 6.8 shows the untrimmed lift coefficient data for various angles of attack and various Mach numbers. Of particular interest are the straight lines obtained for flight in the transonic range. Due to the complexities of transonic aerodynamics, the Digital DATCOM only provides lift data based on a constant value for $C_{l_{\infty}}$ in the transonic range as discussed earlier.

![Figure 6.8: Lift coefficient ($C_l$) for various angles of attack, at various Mach numbers.](image)

6.3.2 $C_D$ vs AoA

Figure 6.9 shows the Digital DATCOM results for untrimmed drag coefficient for various Mach numbers at a varying angle of attack. The lack of data at high angles of attack mentioned earlier can be observed. The noisy curve for $M = 1.1$ is also illustrated.
Figure 6.9: Drag coefficient (untrimmed) for various angles of attack and Mach numbers

6.3.3 \( C_{L_{\text{max}}} \) vs. \( M \)

Figure 6.10: Maximum lift coefficient vs. Mach number.

Figure 6.10 shows the variation of \( C_{L_{\text{max}}} \) with Mach number. The estimated values shown in the transonic range are based on interpolating the angle of attack between \( M=0.6 \) and \( M=1.4 \). \( C_{L_{\text{max}}} \) was then calculated based on this angle of attack.
6.3.4 $C_D$ vs. Mach Number

Figure 6.11 and Figure 6.12 show the variation of drag with Mach number at an altitude of 4000m. The increase in drag as the Mach number approaches unity is quite evident. Figure 6.11, the value of $C_D$ at $C_{\text{max}}$, also shows a good correlation to Figure 6.5. In the transonic range this is in part due to the fact that $C_D$ was estimated based on the value of $C_a$, as given by Eq. 6.3.

![Graph of $C_D$ vs. Mach Number](image)

**Figure 6.11: $C_D$ for maximum $C_a$ at 4000m**

Figure 6.12 does not show as good a correlation with Figure 6.5 as Figure 6.11. However, it must be borne in mind that Figure 6.5 is only an indication of an idealised behaviour pattern. In the transonic region the aerodynamic behaviour is complex and reliable estimates are difficult to obtain from empirically or theoretically based formulations. The values indicated in Figure 6.12 are therefore unlikely to be accurate, but the trends indicated are not necessarily incorrect.

Furthermore, for the purposes of the numerical investigation under discussion, these aerodynamic effects are common to all the simulations and are therefore of secondary importance to the results and can therefore be tolerated.
Figure 6.12: Variation of $C_{L0}$ with Mach Number at 4000 m

6.3.5 $C_L/C_D$ vs. $C_L$

Figure 6.13 shows the Digital DATCOM prediction for variation of Lift to Drag ratio with increasing Lift. It can be seen that the trends indicated resemble those suggested in Figure 6.4. All the DATCOM data utilised in the simulations was limited to values of $C_L$ up to and including $C_{L_{max}}$. The start of the curving of the Lift to Drag ratio curve can be seen at high values of $C_L$, but is not completed due to the data having been removed.

Figure 6.13: Variation of Lift to Drag ratio with Lift in the supersonic regime at sea level
6.4 Software Implementation of the Aerodynamic Model

As discussed in this chapter, most of the aerodynamic coefficient calculations were performed with the aid of the Digital DATCOM. The DATCOM, however, produces output which is human-readable. Software was therefore written to extract the lift, drag and angle of attack values for each flight condition and automatically store these in a format suitable for accessing from a lookup-table.

Each lookup table contained one flight condition, i.e. one Mach number and altitude. Separate lookup tables were generated to allow data to be accessed via $C_L$, $C_D$ and angle of attack.

Furthermore, the Digital DATCOM does not provide values for $C_L$ at high angles of incidence. Those were therefore estimated using Eq. 6.3. Data beyond the stall point was not required, so the data was cropped during processing to the maximum value of $C_L$. In the transonic regime, the DATCOM does not provide maximum values for $C_L$ as it utilises a constant value for the lift slope as discussed in Section 6.3.1. Those were then estimated by determining a stall angle of attack by linearly interpolating between the angle of attack for $C_L_{max}$ at $M=0.6$ and at $M=1.4$. The maximum value for $C_L$ was then determined from the stall angle.

A multidimensional lookup table system was developed to allow interpolation across tables, thus allowing aerodynamic coefficients to be determined at any Mach number, altitude and angle of attack.

The code for the single- and multi-file lookup tables is presented in Appendix I-2.

The code for extracting and processing the Digital DATCOM data is presented in Appendix A-1.
Chapter 7: Simulation Results

Two engine models were constructed based on production engines manufactured by French company SNECMA [17, 18, 19]. These were compared from an engine-only performance perspective, as detailed in Section 7.1, and combined with the aerodynamic data obtained in Section 6.3 to yield the integrated aerodynamic - intake - propulsion-unit performance comparisons presented in the remainder of this chapter.

7.1 Comparison of Engine Performance

This section presents the results obtained from engine-only simulations such as the thermodynamic cycle, thrust specific fuel consumption and net thrust output. Results are based on the SNECMA Atar 9k50 and the SNECMA M53. The Atar series was widely used in the Mirage III, while the M53 has been used in the Mirage 2000. The source code for this section can be found in Appendix C-3.

7.1.1 Propulsion Model Input Parameters

Table 7.1 lists the input parameters utilised for the propulsion model. Notes are listed at the end of the table, indicating sources for the values and assumptions utilised.

Table 7.1: Propulsion model input parameters.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Variable Name</th>
<th>Unit</th>
<th>9k50</th>
<th>M53</th>
</tr>
</thead>
<tbody>
<tr>
<td>Afterburner</td>
<td>AB-ON-OFF</td>
<td></td>
<td>0-OFF; 1-ON</td>
<td></td>
</tr>
<tr>
<td>Fan pressure ratio</td>
<td>p1-f</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Compressor pressure ratio</td>
<td>p1-c</td>
<td>6.15</td>
<td>0.8</td>
<td></td>
</tr>
<tr>
<td>Burner pressure ratio</td>
<td>p1-b</td>
<td>0.97</td>
<td>0.97</td>
<td></td>
</tr>
<tr>
<td>A/B pressure ratio</td>
<td>p1-a</td>
<td>0.98</td>
<td>0.98</td>
<td></td>
</tr>
<tr>
<td>Bypass ratio</td>
<td>Beta</td>
<td></td>
<td>0</td>
<td>0.36</td>
</tr>
<tr>
<td>Fan efficiency</td>
<td>eta-f</td>
<td>1</td>
<td>1</td>
<td>0.95</td>
</tr>
<tr>
<td>Compressor efficiency</td>
<td>eta-c</td>
<td>0.85</td>
<td>0.88</td>
<td></td>
</tr>
<tr>
<td>Burner efficiency</td>
<td>eta-b</td>
<td>0.96</td>
<td>0.97</td>
<td></td>
</tr>
<tr>
<td>Turbine efficiency</td>
<td>eta-t</td>
<td>0.9</td>
<td>0.92</td>
<td></td>
</tr>
<tr>
<td>Afterburner efficiency</td>
<td>eta-a</td>
<td>0.96</td>
<td>0.96</td>
<td></td>
</tr>
<tr>
<td>Primary nozzle efficiency</td>
<td>eta-n</td>
<td>0.97</td>
<td>0.97</td>
<td></td>
</tr>
<tr>
<td>Secondary nozzle efficiency</td>
<td>eta-nmid</td>
<td>0.97</td>
<td>0.97</td>
<td></td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Variable Name</td>
<td>Unit</td>
<td>9.85</td>
<td>MS3</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>--------------------</td>
<td>------</td>
<td>------</td>
<td>-----</td>
</tr>
<tr>
<td>Max turbine inlet temperature</td>
<td>Max_T14</td>
<td>K</td>
<td>1203</td>
<td>1200</td>
</tr>
<tr>
<td>Max afterburner temperature</td>
<td>Max_T16</td>
<td>K</td>
<td>2500</td>
<td>2500</td>
</tr>
<tr>
<td>Generator mass flow max</td>
<td>MaxMassFlow</td>
<td></td>
<td>94</td>
<td></td>
</tr>
<tr>
<td>Compressor suction flow</td>
<td>CompressMassFlow</td>
<td></td>
<td>85</td>
<td></td>
</tr>
<tr>
<td>Fuel energy</td>
<td>JetFuelQR</td>
<td>J/kg</td>
<td>406.40</td>
<td></td>
</tr>
<tr>
<td>Primary nozzle type (converging = 1)</td>
<td>Primary_NozzleTyp</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Bypass nozzle type</td>
<td>Bypass_NozzleType</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Maximum primary nozzle area</td>
<td>A7-max</td>
<td>m²</td>
<td>0.32</td>
<td>0.24</td>
</tr>
<tr>
<td>Minimum primary nozzle area</td>
<td>A7-min</td>
<td>m²</td>
<td>0.15</td>
<td>0.1</td>
</tr>
<tr>
<td>Maximum bypass nozzle area</td>
<td>A9-max</td>
<td>m²</td>
<td>0</td>
<td>0.04</td>
</tr>
<tr>
<td>Minimum bypass nozzle area</td>
<td>A9-max</td>
<td>m²</td>
<td>0</td>
<td>0.04</td>
</tr>
<tr>
<td>Aerodynamic reference area</td>
<td>SREF</td>
<td>m²</td>
<td>35.716</td>
<td>35.716</td>
</tr>
<tr>
<td>Takeoff mass</td>
<td>MASS</td>
<td>kg</td>
<td>13700</td>
<td>13700</td>
</tr>
<tr>
<td>Maximum intake Mach number</td>
<td>Intake_Mmax</td>
<td></td>
<td>2.3</td>
<td>2.3</td>
</tr>
<tr>
<td>Mach number for specified flow rate</td>
<td>Intake_Mmax</td>
<td></td>
<td>0.5</td>
<td>0.6</td>
</tr>
<tr>
<td>Mass flow rate specification</td>
<td>Intake_mdot</td>
<td>kg/s</td>
<td>72</td>
<td>94</td>
</tr>
<tr>
<td>Density spec for flow rate</td>
<td>Intake_rho</td>
<td>kg/m³</td>
<td>1.91</td>
<td>1.91</td>
</tr>
<tr>
<td>Subsonic pressure recovery</td>
<td>Intake_S_bPDR</td>
<td></td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>Intake cone angle (half angle)</td>
<td>Intake_ConeAngle</td>
<td>°</td>
<td>25</td>
<td>25</td>
</tr>
</tbody>
</table>

Notes:

1. Source: [17, 18].
2. Source: [19].
3. From the reference area calculated by the Digital DATCOM.
4. This is the maximum rated takeoff mass for the Mirage III [31].
5. Source: Jane’s [2].
6. Estimated to produce correct net thrust output.
7. Other parameters are estimated based on typical values in the literature.
7.1.2 Thermodynamic Cycle: T-s Diagrams

Figure 7.1 through to Figure 7.4 show the thermodynamic cycles for the 9k50 and M53 on Temperature - Entropy (T-s) diagrams at a free stream Mach number of 0.66 and at an altitude of 0m.

Figure 7.1 shows the T-s diagram for the 9k50 with the afterburner off. This diagram shows that the compressor exit pressure, $p_{c2}$, is 791 kPa and that the turbine inlet temperature is 1200 K. It also shows that $p_{e1}$ the exit pressure is not the same as the free stream pressure as the nozzle is a converging flap type, and the exit stream is therefore not fully expanded, i.e., the nozzle is choked.

![T-s Diagram](image)

**Figure 7.1: T-s Diagram for the 9k50, M=0.66, A/B = Off, Sea Level**

Figure 7.2 shows the T-s diagram for the M53. Key differences between the 9k50 and the M53 are also evident - the M53 has:

- a compressor exit pressure of 1260 kPa
- a turbine inlet temperature of 1600 K
- a bypass stream
Figure 7.8: Comparison of Thrust output between the 9k50 and M53 at 4 km

At an altitude of 4km, the M53 again outperforms the 9k50. For the non-afterburning case, the performance difference is significant at high Mach numbers, but not so pronounced in the afterburning case. Below M = 0.5, the net thrust produced by the M53 with the afterburner operational is marginally lower than with the afterburner off. This is due to exhaust choking.

Figure 7.9 shows the thrust output at 14km. In the non-afterburning case, the mass flow is choked by the exhaust in the subsonic regime and by the intake (low density at high altitude, thus requiring a large capture streamtube) in the supersonic regime. The discontinuity in this case is due to a discontinuity in the algorithm that calculates the mass flow rate as the intake shock pattern changes from detached to attached. However, since this discontinuity is small, the error introduced is deemed acceptable for this investigation. It is therefore also clear that the mass flow in this condition is being choked by the intake.

In the afterburning case, the mass flow is choked by the exhaust in the supersonic regime due to the converging nozzle, and the highly energised exhaust gasses.

Note:
The results presented in this section represent a theoretical solution to the equations presented in Chapter 4 and Chapter 5. Although the trends indicated in Figure 7.7, Figure 7.8 and Figure 7.9 are similar to those for certain turbojet and turbofan engines, the reader is cautioned from using these results directly. Empirical data for the performance of these engines obtained after the compilation of this document indicate that thrust output at high Mach numbers is larger than that predicted by the model. The empirical data for these engines should therefore be consulted should accurate performance data for these engines be required.
7.2 Comparison of the Intake Model with Standard curves

Standard intake performance curves were presented in Section 5.3. Figure 7.10 presents a comparison of these curves with two intake performance curves calculated for the M53 engine configuration at an altitude of 14 km. The subsonic recovery was specified to be 0.99 in this configuration.
In the afterburner off condition, the curve displays small discontinuities as the intake operating mode switches from subsonic to detached, to critical and then supercritical. It can be seen that the deviation of this curve from the standard curves is small (between 3% and 8%).

In the afterburner on condition, the engine is choked by the exhaust and the intake operates in subcritical mode. Thus the lower pressure recovery indicated by this curve is expected.

### 7.3 Thrust - Drag - Load Factor chart

From the relationship given in Eq. 3.7, Figure 7.11 can be plotted for a given aircraft mass and altitude. It shows that as the load factor, \( n \), is increased at a given Mach number, a higher lift coefficient \( C_L \) is required. The achievable load factor is therefore limited by the maximum lift coefficient that the aircraft is capable of at a given flight condition.

![Diagram](image)

**Figure 7.11:** \( C_L \) required for a given load factor and Mach number at 4km

From Figure 7.11, and the aerodynamic data, where the calculated value for \( C_L \) lies within the aircraft's capabilities, the corresponding drag coefficient, \( C_D \), can be determined. This is plotted in Figure 7.12 where the lines of constant \( n \) shown for \( C_L \) in Figure 7.11 are now plotted for \( C_D \).
Figure 7.12: \( C_D \) for \( n = 1 \) to \( n = 9 \) with \( C_T \) superimposed at 4km

In addition, values for the thrust coefficient, \( C_T \), are also plotted for the two engines under consideration, for both the afterburning and non-afterburning cases. From this plot, the maximum Mach number that the aircraft can attain in level flight (\( n = 1 \)) at an altitude of 4000m for the various engine configurations can be determined from the intersection of the \( C_D \) and \( C_T \) at \( n = 1 \) curves. This shows that the M53 can attain a higher Mach number than the 9k50 in both the afterburning and non-afterburning cases, with the effect being more pronounced for the non-afterburning case. This is as expected from the results shown in Figure 7.8.

Furthermore, Figure 7.12 shows the maximum attainable load factor for a given flight condition. Here it can be seen that at an altitude of 4km, the maximum load factor that the aircraft can attain with the 9k50 is just in excess of 2 in the subsonic range with for both the afterburning and non-afterburning cases, and just in excess of 3 in the low supersonic (transonic) range with the afterburner operational.

The M53 is seen to perform considerably better, achieving a load factor well in excess of 2 in the subsonic case, and in excess of three in the supersonic case with the afterburner operational.

The maximum load factors indicated by these results would appear to be quite low compared to the maximum structural load factor of 8 specified in the simulations. However, for all these simulations, the maximum takeoff mass of a Mirage III, 13700kg, was utilised. The 'clean' mass is closer to 9000kg, and the empty mass is around 7000kg. This heavy aircraft mass would therefore account for the apparently poor performance of the hypothetical aircraft.

Figure 7.13 extends the concepts illustrated in Figure 7.12 to various altitudes for the four engine configurations under consideration. The outer boundaries indicate the maximum
attainable Mach number for a unity load factor (where the \( C_I \) and \( C_D \), curves intersect for unity \( n \)). The internal boundaries represent the maximum Mach number achievable for a given load factor, or conversely, the maximum load factor that can be achieved for a given Mach number and altitude.

Many of the detailed features of these plots are common to the SEP plot given by Figure 7.14, and will be discussed in more depth in Section 7.4.

![Load Factor Map for Mach number and Altitude](image)
7.4 Specific Excess Power

The calculation of specific excess power (SEP) was discussed in Section 3.1.3, and is plotted in Figure 7.14 for the two power plants under discussion for both the afterburning and non-afterburning cases. A load factor of 1 was utilised for these plots.

Figure 7.14: Specific Excess Power (SEP) [m/s] for the 9k50 and M53, n=1

Figure 7.14 reflects the data already presented in Section 7.2. The outer boundary representing an SEP of zero reflects the condition where $C_T$ equals $C_D$ on the high Mach number side of the plot. This was seen to be the Mach number at which the $C_T$ and $C_D$ curves intersected in Figure 7.12, and the maximum attainable Mach number for a load factor of unity.

Furthermore, it can be seen that the aircraft has a significantly higher service ceiling with the M53 than with the 9k50. Maximum SEP is also significantly higher with the M53 than with the
9k50. The non-afterburning M53 is able to offer SEP values close to those for the afterburning 9k50.

The decrease in SEP evident in all four plots around $M=1$ is primarily due to the significant rise in $C_D$ in the transonic region.

### 7.5 Sustained Turn Rate

The concept and calculation of Sustained Turn Rate (STR) was discussed in Section 3.1.4. STR is plotted in Figure 7.15 for the two powerplants under discussion for both the afterburning and non-afterburning conditions.

![Figure 7.15: Sustained Turn rate comparison for the 9k50 and M53 at 4km](image)

The effect of drag rise in the transonic regime is again evident. Furthermore, the highest STR occurs at high subsonic Mach numbers, one of the primary reasons for combat taking place in this regime [6]. Again, the M53 appears to offer a higher STR in both the afterburning and non-afterburning configurations. At the peak STR condition, the non-afterburning M53 offers even better performance than the afterburning 9k50.

It is also interesting to note that the values for STR shown in Figure 7.15 are significantly lower than those typically found in literature [1, 6]. Eq. 3.8 indicates that if the thrust matches the drag, the maximum attainable load factor is directly proportional to the thrust to weight ratio. For the aircraft and engine configurations under consideration, the thrust to weight ratio (weight between 94kN and 134kN, Thrust < 65kN for the 9k50 and < 94kN for the M53 at sea level) is significantly less than unity, and hence the low sustained turn rate capability.
Figure 7.16 contains the same results as Figure 7.15, however, the Attainable Turn Rate (ATR) has been superimposed. Where STR occurs at a thrust to drag ratio of one, the ATR is limited by the maximum lift attainable and the structural strength of the aircraft, i.e. the maximum load factor that it is capable of withstanding. The left hand boundary (low Mach numbers) is limited by maximum lift constraint, while the maximum load factor constraint is manifested at higher Mach numbers (right hand boundary).

![Graph showing STR and ATR](image.png)

Figure 7.16: Sustained Turn rate, with Attainable Turn Rate (ATR) superimposed

ATR is significantly faster than STR, but the aircraft operates in a condition where the drag is higher than the thrust and therefore loses velocity or energy height in SEP terms. ATR is primarily an aerodynamic effect, and is only indirectly influenced through power plant by virtue of mass differences. However, since the M53 and 9k50 are similar in mass, this difference was ignored for this investigation.

### 7.6 Range

As discussed in Section 3.1.1, range is an important characteristic of a combat aircraft since it determines the maximum distance from which an airforce can strike an opponent and also how deep into an opponent's territory strikes can be made.

Utilising the assumptions and simplifications given in Section 3.1.1, Eq. 3.3 can provide an estimate of the cruising range of an aircraft. In order to compare the engines under discussion, a baseline range was calculated for the 9k50 without the afterburner at a Mach number of 0.6 and altitude of 4000m. The other engine configurations were then rated against this, as shown in Figure 7.17. The velocities and drag to lift ratios for the two Mach numbers considered is shown in Table 7.3. The 'full' and 'empty' masses were estimated at 13700kg (maximum rated
take-off mass) and a fuel load of 3300 litres, yielding an empty mass of 11320 kg. These parameters indicate a range of 3211 km for the baseline range (green bar in Figure 7.17). Jane’s [3] suggests a combat radius of 1200 km or 2400 km round trip for the M53. This would compare favourably with the range obtained for a Mach number just in excess of 1.

Table 7.3: Flight conditions for Range estimation

<table>
<thead>
<tr>
<th>Mach Number</th>
<th>Velocity [m/s]</th>
<th>Drag/Lift Ratio (c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.6</td>
<td>194.8</td>
<td>0.151</td>
</tr>
<tr>
<td>1.2</td>
<td>389.8</td>
<td>0.373</td>
</tr>
</tbody>
</table>

Figure 7.17: Relative Range for different engine configurations at 4000 m

Since only the 1S1C parameter was varied in Eq. 3.3 for the two different engines, Figure 7.17 reflects the trends shown for 1S1C in Section 7.1.1. At 4000 m, the M53 is seen to provide only a marginal improvement over the 9k50 for dry operation at M = 0.6. The 9k50 is unable to provide sufficient thrust for the aircraft to cruise at a Mach number of 1.2, and the M53 therefore shows a significant improvement over the 9k50. Furthermore, the non-afterburning M53 at a Mach number of 1.2 provides marginally more range than the non-afterburning 9k50 at its maximum Mach number (just in excess of 1). However, in the afterburning condition, the M53 shows an all-round increased range over the 9k50.

The drop in range from a Mach number of 0.6 to 1.2 is due to the increase in the drag/lift ratio.
Chapter 8: Conclusions and Recommendations

8.1 Conclusions

From the results presented in Section 7.6, the M53 results in a range capability which is only marginally better than with the 9k50 in the dry case. However, the M53 does allow a higher Mach number to be achieved in the dry case. In the afterburning case, the M53 shows a distinct improvement over the 9k50 in terms of range. The M53 therefore presents a tactical advantage in an interception role where range and speed are required.

When the two engines are compared in terms of Specific Excess Power, as given in Section 7.4, the M53 presents a clear advantage over the 9k50. It allows a larger flight envelope, and presents an opportunity for attainable performance to be exploited by allowing a more rapid reacquisition of lost energy.

The results obtained for Sustained Turn Rate in Section 7.5 again indicate that an airframe equipped with an M53 engine has an advantage over one equipped with the 9k50. The M53 offers a dry peak STR which outperforms the 9k50 with an afterburner. In traditional short range combat, this would offer significant benefits.

Section 7.3 illustrated that the M53 allows significantly higher load factors to be achieved which is reflected in the STR and SEP data. Furthermore, the M53 allows higher maximum speeds to be achieved than the 9k50.

In general, the M53 appears to offer better all-round performance for the hypothetical aircraft. Its ability to produce significantly more dry thrust than the 9k50 allows supersonic cruising without the use of an afterburner (supercruise). This has additional benefits beyond the scope of detailed discussion in this dissertation such as minimising the aircraft's infrared signature.

A conclusion can therefore be reached from this preliminary investigation that supercruise capability for an old airframe does offer tactical advantages over other similarly classed combat aircraft. However, these performance gains should be viewed in light of a new generation of aircraft that have superior handling and agility that arises not only from their superior power plants, but also from their aerodynamics and control systems.

A conclusion can also be drawn from the results obtained that the simulation algorithms and routines developed performed satisfactorily and the results obtained compare favourably to published data, but that scope remains for improvement.
8.2 Recommendations

This simulation exercise has provided a preliminary indication that supercruise capability for an old airframe offers some tactical advantages. A more detailed simulation of the aircraft in combat should be performed to obtain a more comprehensive evaluation of the performance gains that are potentially offered by this technology.

In terms of the simulations developed in this investigation, the following shortcomings should be addressed in future work:

Aerodynamic Model:

Although the Digital DATCOM provided useful information, it has serious shortcomings. In particular, an improved aerodynamic model should provide:

- Values for the lift and drag coefficient for trimmed flight at all flight conditions.
- A better estimation of the lift curve in the transonic region.
- An improved drag behaviour prediction.

Propulsion Model:

The propulsion model performed well and provided reasonable estimates when compared to published data. However, performance corrections for operation at less than full throttle should be developed. The interaction with the aerodynamic model should then be tightened, such that flight conditions such as lift, drag and thrust are better matched.

The model should also be enhanced to provide a better estimate of the mass flow rate through the engine.

Intake Model:

The intake model should be further developed to better account for the following:

- Mass flow rate through the intake under the various operating modes.
- Corrections for the pressure recovery at angles of attack other than zero. This should then be linked to the aerodynamic data and propulsion model to provide more realistic values for the thrust at an angle of attack.
- The effect of bleeds and auxiliary intake gates should also possibly be added.
References


Appendix A: Intake Model

Appendix A-1 Flow Chart

Engine

Exhaust

m_{max}

Engine

Exhaust

m_{max}

Engine

Exhaust

m_{max}

Engine

Exhaust

m_{max}

Engine

Exhaust

m_{max}

Engine

Exhaust

m_{max}
Appendix A-2 Detached Shock Equations

The general equation of an hyperbola is given by

\[ \frac{x^2}{a^2} - \frac{r^2}{b^2} = 1 \]

rewriting:

\[ r = \pm \frac{b}{a} \sqrt{x^2 - a^2} \]

considering only the positive portion of the hyperbola, the slope is found by taking the derivative with respect to \( x \):

\[ \frac{dr}{dx} = \frac{b}{a} \frac{x}{\sqrt{x^2 - a^2}} \]

The slope of the asymptote is then given by

\[ \left. \frac{dr}{dx} \right|_{x=\infty} = \lim_{x \to \infty} \frac{b}{a} \frac{x}{\sqrt{x^2 - a^2}} = \frac{b}{a} \]

From the Mach line condition,

\[ \left. \frac{dr}{dx} \right|_{x=\infty} = \tan \mu \]

where

\[ \mu = \sin^{-1} \left( \frac{1}{M_\infty} \right) \]

Thus,

\[ \frac{b}{a} = \tan \left( \sin^{-1} \left( \frac{1}{M_\infty} \right) \right) \]

The value for \( a \) must now be chosen such that the slope at \( r = r_c \) is greater than the slope of the conical shock wave generated at \( M_{\text{attach}} \). Solving for \( x \) in terms of \( r \),

\[ \frac{dr}{dx} = \frac{b}{a} \frac{1}{r^2 + b^2} \]
Now,

\[ b^2 = \left( \frac{b}{a} \right)^2 a^2 \]

Substituting, and solving for \( a \),

\[ r = \frac{\sqrt{\left( \frac{dr}{dx} \right)^2 - \left( \frac{b}{a} \right)^2}}{\left( \frac{b}{a} \right)^2} \]

It will be assumed that the slope of the hyperbola at \( r = r_c \) is a linear function of Mach number between \( 90^\circ \) (normal) and \( \varepsilon_m \), the angle of the conical shock at \( M_{attach} \).

Thus the angle of the detached shock at the capture radius is given by:

\[ \theta_c = \frac{\pi}{2} - \varepsilon_m \left( M_{\infty} - 1 \right) + \frac{\pi}{2} \]

and the slope is therefore:

\[ \left. \frac{dr}{dx} \right|_{r=r_c} = \tan(\theta_c) \]

This completely defines the shape of the hyperbola, but not its position relative to the apex of the cone. However, since the mass flow rate the detached shock condition is assumed to be controlled by the mass flow rate through the engine, the capture streamtube diameter is not required, and therefore the position of the shock is not required.

In order to calculate the pressure recovery, a capture streamtube with radius \( r_c \) is assumed, and this area is then divided into annuli of equal area as described in Section 5.4.1.
Appendix A-3 Sample Conical Shock Data

The following data shows the typical output from the Taylor Macoll flow calculation program. The data represents solutions for a $30^\circ$ cone semi-angle at sea level (temperature of 288.15K). The Mach number of attachment may be determined by requesting output between Mach numbers of 1.4 and 1.6 with a smaller increment size.

<table>
<thead>
<tr>
<th>$M_1$</th>
<th>$\text{eps}$</th>
<th>$M_2$</th>
<th>$\rho_2/\rho_1$</th>
<th>$\alpha_2/\alpha_1$</th>
<th>$M_\alpha$</th>
<th>$p_\alpha/\rho_1$</th>
<th>$r_\alpha/\rho_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2</td>
<td>No Solution at this Mach Number</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.4</td>
<td>No Solution at this Mach Number</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.6</td>
<td>50.288</td>
<td>1.07131</td>
<td>1.99466</td>
<td>1.62199</td>
<td>0.932775</td>
<td>2.34512</td>
<td>1.82061</td>
</tr>
<tr>
<td>1.8</td>
<td>51.9339</td>
<td>1.22905</td>
<td>2.17556</td>
<td>1.71871</td>
<td>1.10661</td>
<td>2.54597</td>
<td>1.92294</td>
</tr>
<tr>
<td>2</td>
<td>48.0819</td>
<td>1.36293</td>
<td>2.41734</td>
<td>1.84163</td>
<td>1.2534</td>
<td>2.80687</td>
<td>2.04898</td>
</tr>
<tr>
<td>2.2</td>
<td>45.4174</td>
<td>1.48558</td>
<td>2.69797</td>
<td>1.97573</td>
<td>1.38651</td>
<td>3.10648</td>
<td>2.18501</td>
</tr>
<tr>
<td>2.4</td>
<td>43.4488</td>
<td>1.60008</td>
<td>3.01169</td>
<td>2.11574</td>
<td>1.50954</td>
<td>3.43949</td>
<td>2.32625</td>
</tr>
<tr>
<td>2.6</td>
<td>41.9378</td>
<td>1.70754</td>
<td>3.3562</td>
<td>2.25867</td>
<td>1.62403</td>
<td>3.80392</td>
<td>2.46996</td>
</tr>
<tr>
<td>2.8</td>
<td>40.7461</td>
<td>1.8083</td>
<td>3.73034</td>
<td>2.40243</td>
<td>1.73085</td>
<td>4.19884</td>
<td>2.61423</td>
</tr>
<tr>
<td>3</td>
<td>39.7866</td>
<td>1.90341</td>
<td>4.13351</td>
<td>2.54546</td>
<td>1.83062</td>
<td>4.62375</td>
<td>2.75757</td>
</tr>
<tr>
<td>3.2</td>
<td>39.0012</td>
<td>1.99247</td>
<td>4.56533</td>
<td>2.68654</td>
<td>1.92381</td>
<td>5.07841</td>
<td>2.89884</td>
</tr>
<tr>
<td>3.4</td>
<td>38.3493</td>
<td>2.07598</td>
<td>5.0256</td>
<td>2.82474</td>
<td>2.01084</td>
<td>5.56267</td>
<td>3.03715</td>
</tr>
<tr>
<td>3.6</td>
<td>37.8018</td>
<td>2.15423</td>
<td>5.51415</td>
<td>3.05934</td>
<td>2.0921</td>
<td>6.07642</td>
<td>3.17282</td>
</tr>
<tr>
<td>3.8</td>
<td>37.3374</td>
<td>2.22749</td>
<td>6.03089</td>
<td>3.2899</td>
<td>2.16796</td>
<td>6.61661</td>
<td>3.30231</td>
</tr>
<tr>
<td>4</td>
<td>36.9398</td>
<td>2.29604</td>
<td>6.57576</td>
<td>3.31976</td>
<td>2.23877</td>
<td>7.1922</td>
<td>3.42925</td>
</tr>
<tr>
<td>4.2</td>
<td>36.5968</td>
<td>2.36015</td>
<td>7.14872</td>
<td>3.33694</td>
<td>2.30486</td>
<td>7.79417</td>
<td>3.54942</td>
</tr>
<tr>
<td>4.4</td>
<td>36.2988</td>
<td>2.4201</td>
<td>7.74972</td>
<td>3.45319</td>
<td>2.36655</td>
<td>8.4255</td>
<td>3.66564</td>
</tr>
<tr>
<td>4.6</td>
<td>36.0382</td>
<td>2.47614</td>
<td>8.37875</td>
<td>3.56446</td>
<td>2.42413</td>
<td>9.0862</td>
<td>3.77687</td>
</tr>
<tr>
<td>4.8</td>
<td>35.8089</td>
<td>2.52855</td>
<td>9.03579</td>
<td>3.67073</td>
<td>2.47799</td>
<td>9.77624</td>
<td>3.88311</td>
</tr>
<tr>
<td>5</td>
<td>35.6062</td>
<td>2.57756</td>
<td>9.7208</td>
<td>3.77207</td>
<td>2.52812</td>
<td>10.4956</td>
<td>3.98441</td>
</tr>
<tr>
<td>5.2</td>
<td>35.4261</td>
<td>2.62339</td>
<td>10.4338</td>
<td>3.86858</td>
<td>2.57504</td>
<td>11.2443</td>
<td>4.08088</td>
</tr>
<tr>
<td>5.4</td>
<td>35.2633</td>
<td>2.66628</td>
<td>11.1749</td>
<td>3.96039</td>
<td>2.61891</td>
<td>12.0224</td>
<td>4.17264</td>
</tr>
<tr>
<td>5.6</td>
<td>35.1222</td>
<td>2.70643</td>
<td>11.9439</td>
<td>4.04764</td>
<td>2.65994</td>
<td>12.8299</td>
<td>4.25905</td>
</tr>
<tr>
<td>5.8</td>
<td>34.9916</td>
<td>2.74403</td>
<td>12.7408</td>
<td>4.13051</td>
<td>2.69834</td>
<td>13.6666</td>
<td>4.34268</td>
</tr>
<tr>
<td>6</td>
<td>34.8745</td>
<td>2.77926</td>
<td>13.5658</td>
<td>4.20917</td>
<td>2.73429</td>
<td>14.5327</td>
<td>4.42131</td>
</tr>
</tbody>
</table>

### Start Time: 956488692
### End Time: 956488721
Total Time: 29 seconds
Appendix A-4  Intake Model: Class CConelntake

A.4.1 Header File: Conelntake.h

// Conelntake.h: interface for the CConelntake class.

#ifndef AFX_CONEINTAKE_H__4B68A533_2017_11D1_BB68_000000000000_INCLUDED__
#define AFX_CONEINTAKE_H__4B68A533_2017_11D1_BB68_000000000000_INCLUDED__

#include "LOOKUPTABLE\LOOKUPTB.HPP" // Added by ClassView
#include "IdeaIGas\IdeaIGas.h" // Added by ClassView
#include "IdeaIGas\IdeaIGasStream.h"

#ifndef _MSC_VER
#pragma once
#endif

class CConelntake
{
public:

    void GetIntakeGeometry(double &rc, double &L, double &dx);

    LookupTable * ReplaceLUT(const char *lutfname);

    double PiD(CIdealGasStream* fluid, double aoa, const double mdotmin, const double mdotmax, double &mdot);
    double PiD(double MO, CIdealGasStream* fluid);
    void SubsonicPiD(double newPiD);
    double PiD(double MO);

    double GetRc(void) const { return m_rc; }

    void SetMassFlows(double mdmax, double mdotsuck) { m_mdotmax = mdmax; m_mdotsuction = mdotsuck; }

    void ManDesign(double rcapture, double dL, double travel);
    int AutoDesign(double massflow, double mdotM, double rho, double Mmax);
    int AutoDesign(double massflow, double mdotM, double rho, double Mmax, double BetaMin);
    CConelntake(double ConeAngle, CIdealGasStream *Fluid, const char* lutfname);
    CConelntake();
    virtual -CConelntake();
};
typedef enum enum_OPMODE_tag { eOM_SUBSONIC, eOM_DETACHED,
  eOM_SUBCRITICAL, eOM_CRITICAL,
  eOM_SUPERCRITICAL } eOPMODE;

eOPMODE LastOpMode(void) { return m_opmode; }

protected:
  int CalcSuperCritPiD(double massflow, double flowratio, double & pid);
  int SubCritPiD(double massflow, double flowratio, double & pid);
  int CalcCritCapture(double & rO);
  double DetachedShock(double M0, double *pmassflowrate= NULL);
  double m_betadmax;
  double m_betadmin;
  double m_subpid;
  CIdealGasStream* m_pfluid;
  double m_L;
  double m_dx;
  double m_rc;
  LookupTable *m_pLUTCSP;
  double m_deltac;

  double m_mdotmax;
  double m_mdot suction;

  double m_crittol;  // critical mass flow tolerance (Fraction of critical mass flow)

  eOPMODE e_opmode;

enum enum_LUTCOLS {MACHNO, SANGLE, MACHNOS, P2_P1S, R2_R1S, MACHNOC, PC_P1, RC_R1};
};

#endif // !defined(APX_CONEINTAKE_H_4B68A533_2017_11DI_BB68_000000000000_INCLUDED)

A.4.2 Implementation File: ConeIntake.cpp

// ConeIntake.cpp: implementation of the CConelntake class.
/**
  *  
  */
#include "LookupTable.hpp"
#include "IdealGas.h"
/*
#include <stdlib.h>
#include <math.h>
#include <mathutil.h>
#include <fstream.h> // for cout
#include "ConeIntake.h"

// Construction/Destruction

CConIntake::CConIntake()
{
    m_deltac = 0.0;
    m_pfluid = NULL;
    m_pLUTCSP = NULL;
    m_crittol = 0.05;
}

CConIntake::~CConIntake()
{
    // if we've created a lookup table, free the memory it uses
    if(m_pLUTCSP) delete m_pLUTCSP;
}

CConIntake::CConIntake(double ConeAngle, CIdealGasStream *Fluid, const char* lutfname)
{
    m_deltac = ConeAngle;
    m_pfluid = Fluid;
    m_pLUTCSP = new LookupTable(lutfname, LookupTable::LINEAR, LookupTable::DID);
    m_crittol = 0.05;
}

int CConIntake::AutoDesign(double massflow, double mdotM, double rho, double Mmax)
{
    if(!m_pLUTCSP->IsOK()) return 0;
// get the shock angle for the max Mach number (degrees)
if(m_pLUTCSP->OutOfRange(Mmax)) return 0;
double MmaxBeta = DTR(m_pLUTCSP->Lookup(Mmax, (long)SANGLE));
double Mmin = m_pLUTCSP->RangeMin();
double MminBeta = DTR(m_pLUTCSP->Lookup(Mmin, (long)SANGLE));

// calculate the capture radius based on mass flow and a Mach number for that flow
m_rc = sqrt(massflow/(rho*M_PI*mdotM*m_pfluid->SonicV()));

// calculate the distance from the cone apex to the intake lip,
// assuming that the conical shock touches the intake lip at Mmax
m_L = m_rc/tan(MmaxBeta);

// calc the travel dist of the cone as half the dist
// required to make the lowest attached Mach number touch the intake lip
m_dx = 0.5*(m_L - m_rc/tan(MminBeta));

// store the minimum and maximum values of betad
m_betamin = MmaxBeta;  // note: smallest beta at largest Mach
m_betamax = atan(m_rc/(m_L - m_dx));  // vice versa

return 1;
}

void CConelntake::ManDesign(double rcapture, double dL, double travel)
{
    m_rc = rcapture;
    m_L = dL;
    m_dx = travel;

    // store the minimum and maximum values of betad
    m_betamin = atan(m_rc/m_L);  // note: smallest beta at largest Mach
    m_betamax = atan(m_rc/(m_L - m_dx));  // vice versa
}

// determine the critical mode pressure recovery
double CConelntake::PiD(double MO)
{
    double Mattaeh;
    double beta;
    double M0, Ms;  // Mach numbers just after shock and on cone surface
    double Mlip;  // Mach number at intake lip

    // code to compute the critical mode pressure recovery.
    Mattaeh = ...;
    beta = ...;
    M0 = ...;
    Ms = ...;
    Mlip = ...

    return Mattaeh;
}

Appendix A-4 Intake Model: Class CConelntake Implementation File: Coneintake.cpp
double Mav;    // average Mach number
double Thetalip; // flow angle at the lip

double pt1_p0; // first stage stagnation/static pressure recovery
double pt2_ptl; // second stage stagnation pressure recovery

// if flow is subsonic, return constant recovery
if(M0<1.0) {
    m_opmode = eOM_SUBSONIC;
    return m_subpid;
}

// check if shock is attached.
Mattach = m_pLUTCSP->RangeMin();
if(M0 < Mattach) {
    m_opmode = eOM_DETACHED;
    return DetachedShock(M0);
}

// we now have an attached shock. determine its geometry
beta = DTR(m_pLUTCSP->Lookup(M0, (long)SANGLE));

// determine the Mach number on the cone surface
Me = m_pLUTCSP->Lookup(M0, (long)MACHNOC);

// check if betadimin <= beta <= betamax
if((m_betadimin <= beta) && (beta <= m_betadimax)) {
    Mlip = m_pLUTCSP->Lookup(M0, (long)MACHNOS);
} else if(beta > m_betadimax) {
    m_pfluid->TMRayProps(M0, beta, m_betadimax, Mlip, Thetalip);
}

// calc the average Mach number at the cowl inlet.
Mav = 0.5*(Mlip + Me); // note that this might not be quite accurate
// since the Mach number distribution probably
// isn't linear across the cowl opening.

// calculate the pressure recovery across the shock system
pt1_p0 = m_pLUTCSP->Lookup(M0, (long)PC_P1)*m_pfluid->pt_p(Mc);

// check if a normal shock will occur:
if(Mav > 1.0) pt2_ptl = m_pfluid->NS_pt2_ptl(Mav);
else pt2_ptl = 1.0;
void CConelntake::SubsonicPiD(double newPiD)
{
    // specify a constant pressure recovery for subsonic operation
    if(newPiD <= 1.0) m_subpid = newPiD;
    else m_subpid = 1.0;
}

double CConelntake::DetachedShock(double MO, double *pmassflowrate)
{
    // this calculates the pressure recovery when a detached bow shock is present
    double a; // constant in equation of hyperbola
    double b_a; // ratio of b/a - constant in hyperbola
    double bsq; // b^2
    double theta_rc; // slope of shock at capture radius
    double betaatt; // conical shock angle for attached shock at smallest Mach number
    double Mmin; // lowest Mach number which results in attached shock
    int n;
    double r; // radius being evaluated
    double dr_dx; // slope at given radius
    double pt2_pt1;
    double r2_r1;

    // from the Mach line condition at x = infinity,
    b_a = tan(asin(1.0/MO));
    Mmin = m_pLUTCSF->RangeMin();
    betaatt = DTR(m_pLUTCSF->Lookup(Mmin, (long)SANGLE));

    // estimate the slope at the capture radius
    theta_rc = (M_PI_2 - betaatt)*M_PI_2*(MO-1.0)/(1.0-Mmin) + M_PI_2;
    a = m_rc*sqrt(SQ(tan(theta_rc)) - SQ(b_a))/SQ(b_a);
    bsq = SQ(b_a*a);

    // divide the capture streamtube into 3 rings of equal area, evaluate props
    // at the weighted mean radius of each ring

Appendix A-4 Intake Model: Class CConelntake
Implementation File: Conelntake.cpp
pt2_pt1 = 0.0;

for(n=0;n<6;n+=2) {
    rn = m_rc*sqrt((n+1.0)/6.0);    // calc the radius
    dr_dx = b_a*sqrt(SQ(rn)+bseg)/m_rc;    // calc the slope
    pt2_pt1 += m_pfluid->OS_pt2_pt1(M0, atan(dr_dx));    // get the pressure recovery
}

// cout << M0 << "\t" << b_a << "\t" << a << "\t" << RTD(theta_rc) << "\t" << pt2_pt1/3.0 << endl;

// calculate the mass flow rate if we've been asked for it
if(pmassflowrate)
    */ Note that if we assume that the flow angle is negligible up to
    the intake lip, then the mass flow into the intake is the same
    as the mass flow for the capture streamtube BEFORE the shock.
    this follows from the fact that v2/v1 = r1/r1 (JEA John Eq 4.12)
    */
    *pmassflowrate = m_pfluid->r()*m_pfluid->v()^M_PI^SQ(m_rc);

    // return the average pressure recovery
    return pt2_pt1/3.0;
}

double CConelntake::PiD(double M0, CIDealGasStream * fluid)
{
    m_pfluid = fluid;
    return PiD(M0);
}

double CConelntake::PiD(CIdealGasStream * intakefluid, double aca, const double mdotmin, const double mdotmax, double & mdot)
{
    double pid_crit;
    double M0;
    double Mattach;

    m_pfluid = intakefluid;
    // just do some basic stuff to make life simpler
    M0 = intakefluid->M();

Appendix A-4 Intake Model: Class CConelntake Implementation File: ConeIntake.cpp
// calculate the critical mode pressure recovery
pid_crit = PID(M0);

/* This function must return the pressure recover under all operating modes.  
   Thus it takes as arguments the mass flow rate through the rest of the  
   engine. This is then used to check whether or not the back pressure  
   presented by the compressor face is too high or too low resulting in  
   subcritical or supercritical performance respectively. */

/* The approach will be to calculate the critical mode pressure recover and then  
   correct this for operating condition */

/* This function must also calculate the maximum mass flow rate possible. This  
   is determined by the capture streamtube diameter. This must therefore be  
   calculated. */

// follow the same initial steps as used in the critical mode recovery to determine

/******************
/* SUBSONIC */
/******************
if(M0<1) {
  m_opmode = eOM_SUBSONIC;

  double mdotIntake_Max; // maximum mass flow possible into the intake
  mdotIntake_Max = SQ(mrc)*M_PI*m_pfluid->r() * m_pfluid->v();

  // return the flow through the diffuser as the max flow through the engine
  // mdot = mdotmax/mdotIntake_Max?mdotIntake_Max:mdotmax;
  // mdot = M0*mdotIntake_Max + (1.0 - M0)*mdotmax;
  // mdot = mdotmax;

  if(mdotIntake_Max < mdotmax) {
    double v = m_pfluid->v();
    double r = m_pfluid->r();
    double vint = m_mdotmax/( SQ(mrc)*M_PI*r);
    mdot = m_mdotmax + v/vint*(m_mdotmax - m_mdot suction);
if(mdot > mdotmax) mdot = mdotmax;
else mdot = mdotmax;
return m_subpid;

// now check if the shock is attached
Mattach = m_plUTCSP->RangeMin();
if(MO < Mattach) {
    
    ******************
    /* DETACHED SHOCK */
    ******************

    m_opmode = eOM_DETACHED;

    // The mass flow in this case is determined by the mass flow through the engine or the mass flow possible through the capture streamtube.
    double masstemp, p2_pl;
    p2_pl = DetachedShock(MO, &masstemp);
    mdot = (masstemp < mdotmax) ? masstemp : mdotmax;
    return p2_pl;
}

    ******************
    /* ATTACHED SHOCK */
    ******************

    /* Here the critical part is to determine the operational mode, i.e. whether or not we are sub, super or just critical. Once we determine this, we need to apply some correction for pressure recovery from the critical value */

    double mdotIntake_Max; // maximum mass flow possible into the intake
    double flowratio;
    double r0;
// determine the capture radius for critical flow
if(CalcCritCapture(r0)==0) r0 = m_rc;

// the max flow is given by the capture streamtube area.
mdotIntake_Max = SQ(r0)*M_PI*m_pfluid->r()"m_pfluid->v();

flowratio = (mdotmax - mdotIntake_Max)/mdotIntake_Max;

if(flowratio > m_crittol)
    // The engine can handle more mass flow than the intake. Therefore, back pressure
    // that the diffuser sees will be low and we will probably operate in supercritical
    // mode
    "
    m_opmode = eOM_SUPERCRITICAL;
else if(flowratio < -m_crittol)
    // The maximum engine mass flow is less than the critical intake flow rate. The
    // engine therefore appears as a high back pressure to the intake and the intake
    // will operate in subcritical mode
    "
    m_opmode = eOM_SUBCRITICAL;
else
    m_opmode = eOM_CRITICAL;

switch(m_opmode) {
    case eOM_CRITICAL:
        mdot = (mdotmax > mdotIntake_Max)?mdotIntake_Max:mdotmax;
        return pid_crit;
    case eOM_SUPERCRITICAL:
        mdot = mdotIntake_Max;
        // return pid_crit * sqrt(1.0 - flowratio);
        double pisup;
        CalcSupercritPid(mdot, flowratio, pisup);
        return pisup;
    case eOM_SUBCRITICAL:
        mdot = mdotmax;
        // return pid_crit * pow((1.0 + flowratio), 1.0/3.0);
        double pisub;
        SubCritPid(mdot, flowratio, pisub);
        return pisub;


    return pid_crit;
}
```cpp
int CConelntake::CalcCritCapture(double r0)
{
    double beta;
    double Mlip, MO;
    // flow angle after the shock and at the lip
    double thetas, Thetalip;
    double theta;
    double Ld;
    MO = m_pfluid->M();

    // we now have an attached shock. determine its geometry
    beta = DTR(m_pLUTCSP->Lookup(MO, (long)SANGLE));

    // check if betadmin <= beta <= betadmax
    if((m_betadmin <= beta) && (beta <= m_betadmax)) {
        // we can move the cone, so the capture radius will be the
        // same as the cowl inlet
        r0 = m_rc;
        return 1;
    }
    else if(beta > m_betadmax) {
        // get the mach number at the lip as well as the flow angle
        m_pfluid->XMBRayProps(MO, beta, m_betadmax, Mlip, Thetalip);

        // get the flow angle after the shock
        thetas = m_pfluid->ShockFlowAngle(MO, beta);
        theta = 0.5*(thetas + Thetalip);

        // get the cone position - it will be at an extreme
        Ld = m_L - m_dx;
        r0 = (m_rc - Ld*tan(theta)) / (1.0 - tan(theta)/tan(beta));
        return 1;
    }
}
```
int CConelntake::SubCritPiD(double massflow, double flowratio, double & pid) {
    double M0, v0, rho;
    double Mc, Ms, Mav;
    double beta;
    double r0;

    // determine the capture streamtube for the massflow given
    v0 = m_pfluid->v();
    rho = m_pfluid->r();
    r0 = sqrt( massflow / (M_PI*rho*v0) );

    // we now have an attached shock. determine its geometry
    M0 = m_pfluid->M();
    beta = DTR(m_pLUTCSP->Lookup(M0, (long)SANGLE));

    // intersection between the normal and conical shocks
    double rint;
    // rint = -m_rc * flowratio;
    rint = M_rc * (1.0+flowratio);
    // note that rint is based on an assumption that the normal shock moves in a
    // linear fashion to flow ratio.

    // everything at a capture radius lower than rint passes through both
    // shocks. everything above, through only the normal;

    // areas
    double Aboth;
    double Anorm;
    double Atotal;
    Atotal = M_PI*SQ(r0);
    if( r0 < rint ) { // the entire capture streamtube passes through both
        Aboth = Atotal;
        Anorm = 0.0;
    }
}

Appendix A-4 Intake Model: Class CConelntake
Implementation File: ConeIntake.cpp
Aboth M_PI*SQ(rint);
Anorm = Atotal - Aboth;

// recoveries
double piconical;
double pinormal;
double pt1_p0_c;
double pt2_p1_c;

// determine the Mach number on the cone surface
Mc = m_pLUTCSHP->Lookup(MO, (long)MACHNOC);

// determine the mach number just after the conical shock
Ms = m_pLUTCSHP->Lookup(MO, (long)MACHNOS);

// calculate the average mach number
Mav = 0.5*(Mc + Ms);

// pressure ratio across the conical shock portion
pt1_p0_c = m_pLUTCSHP->Lookup(M0, (long)PC_P1)*m_pfluid->pt_p(Mc);

// check if a normal shock will occur:
if(Mav> 1.0) pt2_p1_c = m_pfluid->NS_p2_p1(Mav);
else pt2_p1_c = 1.0;
piconical = pt2_p1_c*pt1_p0_c/m_pfluid->pt_p(M0);
pinormal = m_pfluid->NS_p2_p1(M0);

pid = (piconical*Aboth + pinormal*Anorm)/Atotal;
return 1;

int CConeIntake::CalcSuperCritPiD(double massflow, double flowratio, double & pid)
{
    double Mattach;
    double MO;
    double beta;

double Mc, Ms; // Mach numbers just after shock and on cone surface
double Mlip; // Mach number at intake lip
double Mav; // average Mach number
double Thetalip; // flow angle at the lip
double pt1_p0; // first stage stagnation/static pressure recovery
double pt2_pt1; // second stage stagnation pressure recovery

M0 = m_pfluid->M();
// we now have an attached shock. determine its geometry
beta = DTR(m_pLUTCSP->Lookup(M0, (long)SANGLE));

// determine the Mach number on the cone surface
Mc = m_pLUTCSP->Lookup(M0, (long)MACHNOC);

// check if betadmin <= beta <= betadmax
if( (m_betadmin <= beta) && (beta <= m_betadmax) ) {
    Mlip = m_pLUTCSP->Lookup(M0, (long)MACHNOS); // Mlip = Ms
} else if(beta > m_betadmax) {
    m_pfluid->TMRayProps(M0, beta, m_betadmax, Mlip, Thetalip);
}

// calc the average Mach number at the cowl inlet.
Mav = 0.5*(Mlip + Mc); // note that this might not be quite accurate
// since the Mach number distribution probably
// isn't linear across the cowl opening.

double Mns;
Mns = (M0-Mav)*(flowratio) + Mav;

// calculate the pressure recovery across the shock system
pt1_p0 = m_pLUTCSP->Lookup(M0, (long)PC_P1)*m_pfluid->pt_p(Mc);

// check if a normal shock will occur:
// note that if Mav is subsonic, then there will be no normal shock further down
// so we check on Mav, but use Mns in the calculation
if(Mav > 1.0) {
    pt2_pt1 = m_pfluid->NS_pt2_pt1(Mns);
    m_opmode = eOM_SUPERCRITICAL;
}
else {
    pt2_pt1 = 1.0;
    m_opmode = eOM_CRITICAL;
}

pid = pt2_pt1*pt1_p0/m_pfluid->pt_p(M0);
return 1;
}

LookupTable * CConeIntake::ReplaceLUT(const char * lutfname)
{
    if(m_pLUT CSP) delete m_pLUT CSP;
    m_pLUT CSP = new LookupTable(lutfname, LookupTable::LINEAR, LookupTable::D1D);
    return m_pLUT CSP;
}

int CConeIntake::AutoDesign(double massflow, double mdotM, double rho, double Mmax, double BetaMin)
{
    if(!m_pLUT CSP->IsOK()) return 0;

    // get the shock angle for the max Mach number (degrees)
    if(!m_pLUT CSP->OutOfRange(Mmax)) return 0;

    // calculate the capture radius based on mass flow and a Mach number for that flow
    m_rc = sqrt(massflow/(rho*M_PI*mdotM*m_pfluid->SonicV()));

    // calculate the distance from the cone apex to the intake lip,
    // assuming that the conical shock touches the intake lip at Mmax
    // m_L = m_rc/tan(MmaxBeta);
    m_L = m_rc/tan(Mmin); // corrected

    // calc the travel dist of the cone as half the dist
    // required to make the lowest attached Mach number touch the intake lip
    m_dx = 0.5*(m_L - m_rc/tan(Mmin));
    // m_dx = (m_L - m_rc/tan(MminBeta));

Appendix A-4 Intake Model: Class CConeIntake Implementation File: ConeIntake.cpp
/ store the minimum and maximum values of betad
// m_betadmin = MmaxBeta; // note: smallest beta at largest Mach
m_betadmin = BetaMin; // note: smallest beta at largest Mach
m_betadmax = atan(m_rc/(m_L - m_dx)); // vice versa

return 1;

void CConelntake::GetIntakeGeometry(double &rc, double &L, double &dx)
{
    rc = m_rc;
    L = m_L;
    dx = m_dx;
}
Appendix B: DATCOM Model

Appendix B-1  Schematic Drawings

Figure B.1 and Figure B.2 present plan and elevation views of the hypothetical combat aircraft, utilised in the Digital DATCOM model. These dimensions are based on schematic drawings presented in reference 3, as shown in Figure B.3.

These views also indicate the cross-sectional area estimations which the DATCOM uses to estimate drag related to the ‘area’ rule. Hatched areas indicate portions of the wing and tail section which are considered to be hidden by the fuselage.

All dimensions are indicated in metres.

![Figure B.1: Plan view of the hypothetical fighter](image)

![Figure B.2: Elevation view of the hypothetical fighter](image)
Figure B.3: Schematic outline superimposed on drawings from Jane's [3]
Appendix B-2  Sample Input File

```
APPENDIX

284.

END
```
Appendix B-3  Sample Output File

1  CONTENT - INPUT ERROR CHECKING
2  ALPHON CODES:  N* DENOTES THE NUMBER OF OCCURRENCES OF EACH ERROR
3  A  UNDEFINED VARIABLE NAME
4  B - MULTIPLE OCCURANCE FOLLOWING VARIABLE NAME
5  C  NON-ARRAY VARIABLE HAS AN ARRAY ELEMENT DESIGNATION (x)
6  D - NON-ARRAY VARIABLE HAS MULTIPLE VALUE ASSIGNMENT
7  E - ASSIGNMENT VALUES EXCEED ARRAY DIMENSION
8  F - SYNTAX ERROR
9

*********** INPUT DATA CASES ***********

SYMMETRIC BODY SOLUTION, EXAMPLE PROBLEM 1, CASE 1

CASE 1  AXIAL

CASM  AXISYM

CASE 2  ANALYTIC

CASM  B2(1):= 0.255, 0.475, 0.75, 1.065, 1.375, 1.685

CASE 3  AXI-SYMMETRIC BODY SOLUTION, EXAMPLE PROBLEM 1, CASE 2

CASE 4  AXI-SYMMETRIC (COMBINED) BODY SOLUTION, EXAMPLE PROBLEM 1, CASE 3

CASE 5  AXI-SYMMETRIC (COMBINED) BODY SOLUTION, EXAMPLE PROBLEM 1, CASE 4

CASE 6  AXI-SYMMETRIC (COMBINED) BODY SOLUTION, EXAMPLE PROBLEM 1, CASE 5

CASE 7  AXI-SYMMETRIC (COMBINED) BODY SOLUTION, EXAMPLE PROBLEM 1, CASE 6

CASE 8  AXI-SYMMETRIC (COMBINED) BODY SOLUTION, EXAMPLE PROBLEM 1, CASE 7

CASE 9  AXI-SYMMETRIC (COMBINED) BODY SOLUTION, EXAMPLE PROBLEM 1, CASE 8
// DATA OMISSION MADE FOR BREVITY

NEXT CASE

// RESULTS CHAIN: 1. MACA(1)-1 : MACH(1)= 1.0, MACH(2)-0.0 : MACA(2)= 1.0, MACA(3)= 0.0, 0.1, 0.2, 0.4, 0.6, 0.8, 1.0, 1.0, 1.2, 1.4, 1.6, 1.8, 2.0

CASED APPROXIMATE AZISYMMETRIC RODY SOLUTION. EXAMPLE PROBLEM 1. CASE 1

// NEXT CASE

// DATA OMITTED DUE TO LENGTH LIMITATIONS.
### DATCOM Model

**1. Automatic Stability and Control Analysis**

**DATCOM Version 1976**

**Characteristics at Angle of Attack and in Lateral DATCOM Body Moment Configuration**

---

**Sample Output File**

---

**APPENDIX B: DATCOM Model**
### Flight Condition

<table>
<thead>
<tr>
<th>Model</th>
<th>Altitude (ft)</th>
<th>Velocity (ft/sec)</th>
<th>MACH</th>
<th>Temperature (°F)</th>
<th>Temperature (°C)</th>
<th>Reference Number</th>
<th>Lateral Acceleration</th>
<th>Derivative (per degree)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FI</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PC</td>
<td>0.003</td>
<td>0.015</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
</tr>
<tr>
<td>LE</td>
<td>0.003</td>
<td>0.015</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
</tr>
<tr>
<td>LN</td>
<td>0.003</td>
<td>0.015</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
</tr>
</tbody>
</table>

### Reference Dimensions

<table>
<thead>
<tr>
<th>Model</th>
<th>Altitude (ft)</th>
<th>Velocity (ft/sec)</th>
<th>MACH</th>
<th>Temperature (°F)</th>
<th>Temperature (°C)</th>
<th>Reference Number</th>
<th>Lateral Acceleration</th>
<th>Derivative (per degree)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FI</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PC</td>
<td>0.003</td>
<td>0.015</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
</tr>
<tr>
<td>LE</td>
<td>0.003</td>
<td>0.015</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
</tr>
<tr>
<td>LN</td>
<td>0.003</td>
<td>0.015</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
</tr>
</tbody>
</table>

### Lateral Acceleration and Control Methods

- **Model**: DATCOM Model
- **Method**: Sample Output File

### Data for DATCOM Model

- **Altitude (ft)**: 0.003
- **Velocity (ft/sec)**: 0.015
- **MACH**: 0.002
- **Temperature (°F)**: 0.002
- **Temperature (°C)**: 0.002
- **Reference Number**: 0.002
- **Lateral Acceleration**: 0.002
- **Derivative (per degree)**: 0.002

---

**Appendix B**: DATCOM Model

**Sample Output File**: 98
Appendix B-4  DATCOM Post-Processing Source Code

B.4.1 Program: DATCOM Post

```c
#include "ctype.h"
#include "stdio.h"
#include "string.h"
#include "iostream.h"
#include "fstream.h"
#include "strstream.h"
#include "math.h"

int stripspaces(char *strbuf, char *inbuf);

int main(void)
{
  double mach; 
  double alt;
  double vel;
  double pcrs;
  double temp;
  double fns;
  double sect();
  double Tref();
  double span;
  double beamon;
  double vref;

  int numfl;
  int curxcol;
  int linecol=0;
  int pos;
  int csi;

  int csiid;
  int coldelims[] = {2, 11, 18, 28, 27, 46, 55, 64, 76, 99, 103, 115, 127};
  // int coldelims[] = {2, 18, 22, 43, 46, 50, 64, 76, 99, 94, 106, 118};

  char buf[512];
  char buf2[256];
```
char buvs[256];
char inName[12];
char surname[112];
char courid[112];
char code[112];
char alphabet[10];
char clf[100];
char codebuff[50];
double cl, clold;
int colres;

int rowstart;

ofstream OutFile;
int freadyton;

cout << "Enter the filename: ";
cin >> inName;
ifstream InFile(inName, ios::in, ifstream::skip_eof);
InFile.eof() = false; // set first char to space
while(!InFile.eof()) {
    freadyton = 0;
    while(!freadyton) {
        // extract all data until we find a line starting with "=
        while(InFile.getline(buvs, sizeof(buvs))) {
            if(buvs[0] == '=') {  // line starts with '='
                InFile.getline(clf, sizeof(clf));
                line++;  // increment line counter
                break;
            }
        }
        if(InFile.eof()) break;
    }
    // extract the next line
    InFile.getline(buvs, sizeof(buvs));
    line++;  // increment line counter
}
// Check if it says "CHARACTERISTICS AT ANGLE OF ATTACK AND IN SIDESLIP"
if (strcmp(buf, "CHARACTERISTICS AT ANGLE OF ATTACK AND IN SIDESLIP"))
    // if not ref_out, "CHARACTERISTICS OF HIGH-LIFT AND CONTROL SYSTEM"
    freadxyz = 1;

buf[0] = ' '; // set first char to space

if (inFile.eof()) continue;

// get job description data
InFile.eofwhite();
InFile.getline(buf, sizeof(buf));
lineNo++;

// strip space/config, buf:
strcpy(config, buf);

InFile.eofwhite();
InFile.getline(buf, sizeof(buf));
lineNo++;

// strip space/nn, buf:
strcpy(totend, buf);

if (totend[0] == ' ')
    cout << lineNo << " " << totend << end;
else
    cout << lineNo << " " << config << " " << end;

// extract ref out: is we encounter data, marked with a zero.
if (buf[0] == '0')
    while (buf[0] == '0')
        InFile.getline(buf, sizeof(buf));

lineNo++;

// extract the zero
InFile.ignore();

// now extract some useful data
InFile >> mach >> alt >> vel >> pico >> temp >> Re >> ref;
InFile >> ref >> same >> referenced >> unref;

mach = cosl(3456.0); // just to let the compiler know we need fp support
```c
// // extract another line
FILE *getline(file, sizeof(buf));
lineno--;

// set number of returns
mode = sizeof(colcolJim)/sizeof(colcolJim[0]);

// create the output file name
sprint(output, "%s-%03d-%03d.%03d.%03d.dat", inname, mach, alt, nstep);

// open the output file
OutFile.open(outname, ios::app|ios::out|ios::filebuf::openopt);

// output the state parameters read in earlier
OutFile << "Read file(s) << inname << endl;
OutFile << "Time Elapsed" << output << " " << endl;

 OutFile << "Mach Number" << output << endl;
 OutFile << "Alititude" << endl;
 OutFile << "Velocity" << endl;
 OutFile << "Pressure" << endl;
 OutFile << "Temperature" << endl;
 OutFile << "Kinematic Viscosity" << endl;
 OutFile << "Gas Constant" << endl;
 OutFile << "Gas Viscosity" << endl;
 OutFile << "Vertical Momentum Fluxes" << endl;

 // set the line with the headers
 InFile.lineToBuf(infile, sizeof(buf));
 lineno--;

 // OutFile << inname << " \n";

trad(buf1, " ");
_sprintf(buf2, " ");
```

Appendix B: DATCOM model

Program: DATCOM Post

104
/*

// extract headers
for(jourcol=0;jourcol<cols;jourcol++) {
    colwid = coldelims[jourcol];
    if(coldelims[jourcol] == 
    stringpy(buf, bufcoldelims[jourcol], colwid);

    bufl[colwid] = '0'; // append terminating null
    stripaces(buf, bufl);

    // count += buf2 = "ef;
    OutFile << buf2 << "et"
    // count += end;
    OutFile << endl;
+

    // strip another line
    InFile getline(buf, sizeof(buf));
    lineno++;

    // now we want to start processing the data
    buf[0] = "1;"
    InFile getline(buf, sizeof(buf));
    lineno++;

    restart = lineno; // record the starting line

    // create a temporary output stream
    ofstreamstream TempOut;

    // set colcol to very negative number
    colcol = -999999.00;

    while(buf[0] != '1;') {
        curcol=0;
        // alpha
        colwid = coldelims[curcol];
        stringpy(buf1, buf+coldelims[curcol], colwid);
        bufl[colwid] = '0'; // append terminating null
        stripaces(alphabet, bufl);
        // ed
        curcol++;
        colwid = coldelims[curcol+1]-coldelims[curcol];
        stringpy(buf1, buf+coldelims[curcol], colwid);
        buf1[colwid] = '0'; // append terminating null
        llen = stripaces(coldbuf, buf1);
        if(llen == 0) printf("Badbuf, \"-999.0\") // there was no number
*/
// check if we've reached the peak in the left curve
    if (colwid < 'W') { ...
      continue;
    }
    colwid = 'C';
    Tempout << char << '\' << char << '\' << char << 'c' << endl;
    // read the next line
    Tempout.getline(buf, sizeof(buf));
    line++;...

char *Tempstream;
int pcount;
pcount = Tempout.pcount();
Tempstream = tempout.stream();
// append null value to the list we're stored
Tempstream[pcount] = '\0';
OutFile << line++ << rowstart << endl;
OutFile << 'I' << line;
OutFile << Tempstream;
OutFile.close();

return line;
}

int stripspaces(char *bufb, char *buf) {
    char *pos = ...

Appendix B. DATCOM Model
Program: DATCOM Post
char *buf;
int len;
int strc, ends;

// duplicate the string
buf = strdup(inbuf);

// get total length
len = strlen(inbuf);

// reverse the string so that we can strip trailing spaces
revstr = strrev(buf);

// find first non-space
ends = len - strlen(revstr, ' ');

str = strsep(inbuf, " ");

// clear the output buffer
stdout[output, 0];

// this will happen if string is all blank
if(ends + str) ends = strc - 0;
else strcpy(output, inbuf + str + ends strc);

// append null character to string
output[ends + str] = '\0';

// free memory allocated by strdup();
tfree(buf);

return ends strc;

B.4.2 Program: CD_CL

#include <iostream.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "./lookupTable/lookupTB.cpp"

bool GetMachNameProcName(const char *pszName, double &Mach);
```c
int main(void)
{
    char infilename[512];
    char infile[512];
    char outfile[512];
    int i, j, n;
    int row, cols;
    int row, col;

    double cd[100], cd[100], alpha[100], cd[100];
    double cmax, cmax, alpha_cmax, mwclaw, nclaw;
    int n, m;
    
    cout << "Enter an input file: ";
    cin >> infilename;
    ifstream infilenm(infilename);
    List >> cout;

    // allocate memory for the clmax stuff
    Cmax = new double[100];
    cd_cmax = new double[100];
    alpha_cmax = new double[100];
    mwclaw = new double[100];
    
    bool bMatch;
    double Xtemp;

    for (ent = n; ent > count; ent--)
    {
        List >> infile;
        bMatch = getMatchFromName(infilename, Xtemp);
        if (bMatch) { Xlist[ent] = Xtemp; }
        else { C = 0; }
        n = 0;
    }
}
```
// this block will ensure that the lookup table is
// created and then destroyed when we no longer need it.
// create a lookup table
LookupTable CLCD(infile, LookupTable::INPLACE, LookupTable::D1D);

CLD = CLCD.Lookup(0, {10g});

ifstream Data(infile);
strncpy(coutfile, infile);
strncpy(ofoutfile + strlen(infile), "CLCD.dat", 100);

ofstream Out(ofoutfile);
Data >> rows;
Data >> cols;
RMxetl = 0.0;
for (row = 0; row < rows; row++) {
   Data >> C1[row];
   Data >> Cd[row];
   Data >> alp[row];
   Data;
   if(Cd[row] < 2.0) {
      C0 = Cd[row];
      C0 = C0 / (1.0 + C0);
      C0 = C0 / (1.0 + C0);
      C1 = C1 + C1[row] * C0[row] * factor;
   } else {
      C0 = Cd[row];
      C1 = C1[row];
   }
}
// extract maximum CI
if (Cl[row] > CImax[cut]) {
    CImax[cut] = Cl[row];
    Cd[CImax[cut]] = Cd[row];
    Alpha_CImax[cut] = alpha[row];
}

if (Cl[row] > 0.0) Cl[row] = 0; // just write out for > CI

// dump the CD CL Alpha lut
Out << clp << endl;
Out << colu << endl;
for (row=0; row<rows; row++)
    if (Cl[row] > 0)
        Out << Cd[row] << \"t\" << Cl[row] << \"t\" << alpha[row] << endl;
Out.close();

Data.close();
// Out.close();

// dump the climax and cd at cx max data
strcpy(cnxfile, infiles);
strcpy(outfile, strlen(infiles) + \".clmaxcd.lut\");
ofstream CMAXCd(outfile);
CMAXCd << ext << \"t\" << 0 << endl;
for (cnt=0; cnt<cut; cnt++)
    CMAXCd << Mlist[cnt] << \"t\" << CMAX[cnt] << \"t\" << Cd_CMAX[cnt] << \"t\" << Alpha_CMAX[cnt] << endl;
CMAXCd.close();

return 0;
}
bool GetMachineName(const char *pszName, double &mach)
{
    char *start;
    char *end;
    start = strstr(pszName, "-MHI");
    if(!start) return false;
    // move string by 6 to pass over "-MHI"
    start += 6;
    Mach = strtol(start, &end);
    return true;
}

B.4.3 Program: CL_CD_Full

#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <stdio.h>
#include <math.h>

#include "../LookingTable/lookingTable.hg"

int main(void)
{
    char infolist[512];
    char infile[512];
    char outf ile[512];
    int test, test2;
    int rows, cols;
    int row, col;
    double CL[100], CD[100], alpha[100], CDf;
    double LCD, LCI;
    double factor;
    double CLt, CDt;

Appendix B: DATCOM Model
```c
int clp, on;

char nameEnt[] = "C:/D1CDAT/M111/CLCD/M111 MDLIE 3.1fm100.dat";

// Open file "Enter an input file:");
// clp => initialize;
// list => initialize;
// list => empty;

double M_List[] = { 0.2, 0.4, 0.6, 0.7, 0.8, 0.9, 1.1, 1.2, 1.3, 1.4, 1.6, 1.8, 2.0, 2.2 };

double A_List[] = { 0.0, 2000.0, 4000.0, 6000.0,
                    8000.0, 10000.0, 12000.0, 14000.0,
                    16000.0, 18000.0, 20000.0 };

int M[14];
int A[11];

int DENT, ACnt;
double Alt, R;
double alpha_06, alpha_14, alpha_clp;

// go by article
for(ACnt=0, ACnt=A, ACnt++) {
    ACnt = A_List[ACnt];

    // get the stuff at 0.6 and 1.4 Mach
    char buff[1024];
    int r, o;
    int a;
    float at, bt, ct;
    sprintf(buff, nameEnt, 0.6, Alt);
    fscanf(stuff, buff);
    stuff >> r >> o;
    stuff.getline(buff, 1024); // just get the rest of the line on which note in.
    for(n=0; n<r; n++) stuff.getline(buff, 1024);
    mconf[buf, "mf " et, nat, abt, cct);
    alpha_06 = ct;
```
```c
stuff.close();

char buf[1024];
int r, c;
int n;
float at, br, et;
sprintf(buf, namefmt, 1, i.e, Alt);
if (readstuff(buf)) { stuff >> r >> c;
stuff.getline(buf, 1024); // just get the rest of the line to which data file
for (n=0, n<r++) stuff.getline(buf, 1024);
assert(out, "% d % 1", &at, &br, &et);
alpha_11 = at;
stuff.close();
}

// now go by each number
for (Menu=0, MenuSize, Menu++) {

M = Menu; Menu++; Menu++)

// generate the filename
sprintf(filename, namefmt, M, Alt);

// this block will ensure that the lookup table is
// created and then destroyed when we no longer need it.
{
// create a lookup table
lookupTable(C2D(filename, LookupTable::LINEAR, LookupTable::DIR);
C0 = CLCD.Lookups[0.0, (long)];
}
```
```c
ifstream DataInfile;
strncpy(outfile, infile); 
strncpy(outfile + strlen(infile) + 1, "CLCD_Full.dat"); 
ofstream Outfile(outfile);

// get the number of rows and columns
Data >> rows;
Data >> cols;

// if we're 0.6 < N < 1.4 then we want to clip alpha
if (0.6 < M && M < 1.4)
    alphaclip = (M * 0.6) + (alpha_05 - alpha_14) / (1.4 - 0.6) + alpha_06;
clip = 0;
nn = 0;
for (row=0, row<rows, row++) {
    Data >> Cl[row];
    Data >> Cd[row];
    Data >> alpha[row];
    if (0.6 < M && M < 1.4) {
        if (alpha[row] < alphaclip) nn++;
        else n++;}
    if (Cd[row] > 0.3) {
        // classical equation:
        // (Cf = 0.5 + C2/2 + AR*V)
        factor = 1.0 + (Cd[row] - M0) / (C2*ICL);
        else {
            ICL = Cd[row];
            ICL = Cd[row];
        }
        if (Cf[row] >= 0.0) clip++; // only write out for > 0.0
    }
```
if (0.6 < M && M <= 1.4) {
    MBI = 0.6;
    C1[nn-1] = C1c;
    Cdc[nn-1] = Cdc;
    alpha[nn-1] = alphaclip;
}

Out << " M << endl;
Out << "C1c << endl;
for (row = 0; row <= mm; row++)
    if (C1[row] >= 6.0) {
        Out << C1[row] << " \t " << Cdc[row] << " \t " << alpha[row] << " \t " << endl;
    }
Data.close();
Out.close();
}

return 0;
}
Appendix C: Propulsion Model

Appendix C-1 Flow Chart for estimating Thrust Output

1. Initial
2. \( M_{\text{in}} \)
3. \( M_{\text{old}} \)
4. \( M_{\text{old}} \approx M_{\infty} \)
5. Estimate \( M_{\infty} \) at critical operation
6. Calculate exit conditions from thermo model
7. Calculate \( f \) and max flow, then get max intake flow
8. Re-evaluate \( f \) using new intake flow, and correct for operating mode, get actual intake flow
9. Actual intake flow
10. Calculate exit conditions from thermo model
11. Calculate \( f \)
12. Calculate exit flow
13. Thrust
Appendix C-2  Propulsion Model: Class TurboJet1D

C.2.1 Header File: 1DPropModel.hpp

```cpp
// ---------------------------------------------------------------------------
// FILENAME: 1DPropModel.hpp
// NAME:      CLASS TurboJet1D
// Description: Header file defining class TurboJet1D - a one dimensional
// gas-turbine propulsion model. Model accounts for component
// efficiencies and allows for a bypass stream.
// Authors:   [Authors Name]
// Date:      [Date]
// Revision:  [Revision]
// ---------------------------------------------------------------------------

#include <math.h>
#include <iostream>
#include "1DPropModel.h" // Added by ClassView
#include "\include\1DPropModel\1DPropModel.h"

// Define - square roots
#ifdef SQRT
   #define SQRT(x) (x*2.0)
#else
   #define SQRT(x) (x+1.0)
#endif

const double Rair = 287;

class TurboJet1D
{
   protected:
      double njetmax;

   private:

      // *** Compressor Parameters ***
      // State Pressure ratios
      double pi_0;  // bypass fan pressure ratio
      double pi_c;  // compressor pressure ratio
      double pi_b;  // bypass pressure ratio
      double pi_e;  // afterburner pressure ratio
```
// stall notices
double e[5];  // fan
double e[3];  // compressor
double e[1];  // turbine
double e[1];  // afterburner
double e[1];  // nozzle
double e[1];  // bypass nozzle

// Maximum Properties
double T[5];  // max turbine inlet temp
double T[5];  // max afterburner temp

// Maximum Nozzle Areas
double n_A[5];  // max primary nozzle area
double n_A[5];  // max bypass nozzle area

// Minimum Nozzle Areas
double n_A[5];  // min primary nozzle area
double n_A[5];  // min bypass nozzle area

// Fuel
double OR;  // lower heating value of jet fuel

// Fuel Types
enum Nolitude {nozzleprimary, nozzlebypass};
enum Nolitude {nozzleprimary, nozzlebypass};

// absolute pressure loss function
double ThrottleDef(double, double, double, double);

/**** Operational Parameters ****/
double beta;  // bypass ratio for bleed air

/**** Gas Property Utility Functions ****/

// constant double heat = 287.0;
// Cengel & Boles, Throttle 5.2, note: only really valid for T = 273K to 1500K

// double Cp(double temp) { return (18800.0/287.0)*(28.11 + 0.00125*temp + 0.000025)*(temp*(temp) + 1.0666*exp(-1.0666*temp)); }
double Cp(double temp); double s;  // double Cp(temp); return s/cp(temp) - s;

double gasn(double temp, double s);  // double Cp(temp); return s/cp(temp) - s;
double PI(double r) { return (1.0 + 0.5*exp(-0.5*SQRT(r))); } // C88 16-14 pg 729.
double PI_1(double r, double gamma) { return pow(r/2.0, 1.0 - 0.5*gamma) * SQRT(gamma) * gamma^(gamma-1.0); } // C88 16-14 pg 729.
double PI_2(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

double PI_1r_0(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

double PI_1r_0(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

double PI_2(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

double PI_2(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

double PI_2(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

double PI_2(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

double PI_2(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

double PI_2(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

double PI_2(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

double PI_2(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

double PI_2(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

double PI_2(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

double PI_2(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

double PI_2(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

double PI_2(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

double PI_2(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

double PI_2(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

double PI_2(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

double PI_2(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

double PI_2(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

double PI_2(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

double PI_2(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

// Standard ATA without prescribed velocity function.

double PI_1r_0(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

double PI_2r_0(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

double PI_3r_0(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

double PI_4r_0(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

// Virtual function to set component parameters.

void PI_1r_0(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

// Functions to set component parameters.

void PI_1r_0(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

void PI_2r_0(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

void PI_3r_0(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

void PI_4r_0(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

void PI_1r_0(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

void PI_2r_0(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

void PI_3r_0(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

void PI_4r_0(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

void PI_1r_0(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

void PI_2r_0(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

void PI_3r_0(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

void PI_4r_0(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

void PI_1r_0(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

void PI_2r_0(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

void PI_3r_0(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

void PI_4r_0(double r_0, double r_p, double r) { return (1.0 - pow(2.0*r_0/r, 2.0)); } // C88 16-14 pg 729.

void MainOutletType( enum NusseltNum nusType )
{
    nusselt = nusType;
}
void dBhruType( enum NusseltNum nusType )
{
    nusselt = nusType;
}

enum NusseltNum { CONSTANT, VARYING, ...};
enum ABEtype { OFF, ON, ...};

int draw(double M, double Mx, double p, Atmnum AT, Mf, int printscreen, ofstream OutputFile);
double Thru(double M, double T, double p, ABEtype ABE, double Pmus, double L);
int Yeto(double M, Mf, double R, double p, double atm, double bcho, double Atenoff, double Aten, double snf, double yt);

/* state variables - use for diagnosis:
  double c_M, double M,
  double c_M_max,
  double c_M_stagn,
  double c_M_stop,
  double d_diameter,
  double c_diameter,
  double c_hole, double c_M, double c_cp,
  double c_d,
  double c_diam, double c_hole_dist,
  double c_M_diam, double c_M_hole, double c_M_hole_dist;
*/

protected:
  double pi_d_diam;
  double M0, Lw,
  Constant M0diff;

double CyclEnergetics(double M, double T, double p, Atennum Atenoff, double pi_d, Cdeadenerstrea ex, Cdeadenerstrea ex, double
  a_M, double b_M, doubles c_M, doubles f);
}

C.2.2 Implementation File: 1DPropModel.cpp

******************************************************************************

File: 1DPropModel.cpp
NAME: 1DPropModel.cpp
DESCRIPT: 1D propeller model - a one dimensional gas turbine propeller model. Model accounts for component
  interaction and allows for a bypass stream.
AUTHOR: Prakash Mathew
DATE: 26 April 1989
DESCRIPTION HISTORY:

Appendix C-2 Propeller Model (1D)
# include "1DPropModel.hpp"

// C->calculus/ob struct. for all 4 given temperature.

double turboset1(double t)
{
    if (t <= T1max - T1) return (1.0058 + (1.338056e-3)*t - (1.914136e-6)*t*t + (0.689156e-9)*t*t*t + (0.0056475e-12)*t*t*t*t))/(1.0057*1.32e-12); 
    return (1.0043 + (1.338056e-3)*t - (0.689156e-9)*t*t + (0.0056475e-12)*t*t*t)/(1.0057*1.32e-12);
}

// This function is used to see that everything lines up if a constant specific heat is used.

double turboset2(double t)
{
    return 1.0057;
}

/*

// we may later want to add to this function, hence out of line inline definitions

void turboset1D:TL4(double val)
{
    T1max = val;
}

void turboset1D:TL6(double val)
{
    T1max = val;
}

int turboset1D:DrawCO(double QC, double T0, double p0, ADDRESS ARXOff, int plotchoice, position OutFile)
{
    // specific heat values
    double gamma0; // gasideal
    double gamma0v;
    double gamma0v;
    // specific heat
    double Qc;
    // purposes

Appendix C-2: Propulsion Model Class Turbojet1D
Implementation File: 1DPropModel.cpp
double Cпп; // burner
double Cпп; // burner
double Cпп; // afterburner
double Cпп; // nozzle exit

double theta_u;

double del_d:

// stagnation temperature ratios

double tau_u; // compressor exit
double tau_u; // burner
double tau_u; // exit

double T_u; // turbine exit

double T_u; // afterburner exit

double T_u; // afterburner exit

double T_u; // bypass fan exit

double T_u; // bypass nozzle exit

// other:

double f_y; // fuel/air ratio for primary burner

double f_y; // fuel/air ratio for afterburner

double p_i;
double p_i;
double p_i;
double p_i;
double p_i;
double p_i;

double u_i;

double u_i;

double u_i;

double u_i;

// enter stagnation conditions of intake exit

double u_i; // exit velocity of primary burner

double u_i; // exit velocity of bypass burner

double M_i; // exit Mach number

double M_i;

// determine stagnation conditions of intake exit

// determine flow, air properties for compressor:
gamma_c = gamma(0.5*TD*(1.0 + theta));

Cpc = Cp(0.5*TD*(1.0 + theta));

// Calculate pressure ratio across the compressor

tau_c = (powp1_c * gamma(1.0))/gamma(1.0); // Eq 38/3/69 Eq. 13

Tt5 = tau_c*theta*TD; // Eq 38/3/69 Eq. 14

// Determine average burner specific heat:

Cpb = Cp(0.5*(Tt2+Tt4));

// Determine burner fuel ratio

th = Cpb/Tt4; // Eq 28/4/77 Eq. 15

tau_b = Tt4/Tt1;

// Calculate temperature ratio across the fan

tau_f = (powp1_f * gamma(1.0))/gamma(1.0); // Eq 28/4/77 Eq. 27

Tc6 = tau_f*theta*TD;

// Determine average turbine specific heat, assume drop in temp roughly equal to compressor temp rise:

Cpt = Cp(Tt6 - 0.5*(Tt6*theta*TD)); // Eq 28/4/77 Eq. 20

gamma_t = gamma(Tt6 - 0.5*(Tt6*theta*TD)); // Eq 28/4/77 Eq. 20

// Determine temperature ratio across the turbine

tau_t = 1.0 - Cpt*(theta(1.0 - tau_c)/(tau_c + 1.0))/tau_c; // Eq 28/4/77 Eq. 28

Tt5 = tau_t*Tt4;

// Check if afterburner is on or off and apply appropriate combustion

if (afterburner on) {
    Tt7 = Tt6 + Tt6max;
    Cpa = Cp(0.5*(Tt7+Tt6));
    ta = Cpa*(1.0 - (Tt5/Tt6))/gamma(1.0); // Eq 28/4/77 Eq. 20
} else { // Afterburner is off
    Tt6 = Tt4 = Tt5 = TA;
    Cpa = Cp(Tt6);
    ta = 0.0; // No fuel is used
}

// Calculate pressure ratio across the compressor

p1_c = pow((1.0 - tau_c)/eta_c, gamma_c/gamma(1.0)); // Eq 28/4/67 Eq. 28

// Determine the stagnation pressure ratio across the diffuser

p1_c = P1_diffuser(T0, T3, p0);
p6 = po*delta*pi; delc_pl1 = p1_l; // FF 28/4/97 Eq. 21

// If either of the two different exit conditions must now be considered.

// If the nozzle is convergent, we need to check if it is choked. We then consider
// if the throat is convergent or divergent and check the exit Mach number.
// If the Mach number exceeds 1, we then apply the choked solution, otherwise we use the
// fully expanded solution.

// **** evaluate the fully expanded case: ****

// Estimate Cpo by first estimating T7, using turbine constants

gamma = qgamman;
Cpo = lpt;
p7 = po;

u7eq = 12*eta_n*Cpo*T7^1.0-pow(p7/p6, (qgamman-1.0)/qgamman); // FF 28/4/97 Eq. 24
T7 = Tt6 - u7eq/(2*Cpo); // FF 28/4/97 Eq. 25

// now reestimate Cpo

gamma = qgamman(0.5*T7^4-T7). / Ra/r;
Cpo = (p6*exp(T7^4-T7));

u7eq = (2*eta_n*Cpo*T7^1.0-pow(p7/p6, (qgamman-1.0)/qgamman)); // FF 28/4/97 Eq. 26
T7 = Tt6 - u7eq/(2*Cpo);

N7 = sqrt(u7eq/[gamma*n*Ra/r]); // FF 28/4/97 Eq. 27

// **** now evaluate convergent, choked solution only if N7 > 1

if (!isneg1primary || CONVERGENT == true) { // FF 28/4/97 Eq. 27

u7eq = gamma*Ra/r; // FF 28/4/97 Eq. 28

p7 = p6*pow(1.0-u7eq/(2*eta_n*exp(T7^4-T7)), (qgamman-1.0)/qgamman); // FF 28/4/97 Eq. 29

N9 = sqrt(u9eq/[gamma*n*Ra/r]);

Appendix C-2 Propulsion Model: Class TurboJet1D
Implementation Files: 1DPropModel.cpp
```c
/**
 * @brief Evaluate convergence, check printed only if NT = 1
 */
if (mode != bypass && CONVERGED && (freq > 1.0))

T9 = 3.0*Cp*sT/(gamma+Rair*)2; // Eq 28/4/97 Eq. 31

p9 = gamma/2*Cp*Ra/Rair;

p9 = pdi*pdi/(1.0 + 4.0*ep/an_ph*(pdi*sTh)), gamma/(gamma + 1.0); // Eq 28/4/97 Eq. 32

// we have now calculated everything that we need.
// so we can output our results to the file specified.

double n[3]; // we will ignore point 1

t[0] = 0;
t[1] = deltan(theta[0], pi*diu0, Cpi/3^Tc(1+theta[0]), Rair);
t[2] = t[1] - deltan(tau_a, pi, Cpi, Rair);
t[3] = t[2] - deltan(tau_b, pi_b, Cpb, Rair);
t[4] = t[3] + deltan(tau_t, pi_t, Cpt, Rair);
t[5] = t[4] + deltan(Th_Tci, pi_a, Cpa, Rair);

// the following eq for sfil looks better as the gran, but
// is actually less accurate since it does not take into account
// the various specific heats which were present along the process
// path:
// sfil = deltan(Th/Thc, p5/p0, Cot(6.5^TcTci)), Rair

t[6] = t[5] + deltan(tau_f, pi_f, Cpf, Rair);

// double na;

if(outputFile.good())
{
    outputFile << "end";
    // if we want p0, the pressure labels for each isobar.
    if(p0 != 0.0)
    {
        outputFile << "$p0": " << p0/1000.0 << "$kPa": " << (p0*p0)*pdi/1000.0;
        outputFile << "$kWi": " << deltax*50*diu0*pi*pi/1000.0;
        outputFile << "$kW": " << deltax*50*diu0*pi*pi/1000.0;
        outputFile << "$kW": " << p5/1000.0 << "$kPa": " << p5/1000.0;
        outputFile << "$kPa": " << p5/1000.0;
    }
}

outputFile << endl;
```
double gamma;

// specific heats
double cp_c; // compressor
double cp_b; // burner
double cp_t; // turbine
double cp_a; // afterburner
double cp_M; // nozzle exit

double theta0;
double deltheta;

// stagnation temperature ratio
double tau_u; // upstream
double tau_b; // burner
double tau_t; // turbine

double tau_l; // farfield

// temperatures
double T3; // compressor exit
double T4 = T1max; // turbine entry
double T5; // turbine exit
double T6; // afterburner exit
double T7; // afterburner exit

double T8; // bypass fan exit

double T9; // bypass nozzle exit

// other

double at; // exit area ratio for primary burner

double fa; // flow area ratio for afterburner

double pt2;
double p1;
double x1;
double ps_a_d;
double ps_a_d;
double pl1;
double pl2;

double u1g; // exit velocity of primary stream

double u2g; // exit velocity of bypass stream

double M1; // exit Mach number

double M2;

// determine stagnation conditions of intake air
theta_AO = Tt_1(Ro, gamma{Tg, Ra0});  // PM 26/4/97 Eq. 12
theta_Au0 = pc_p(R0, gamma{Tg, Ra0});  // PM 28/4/97 Eq. 13

// determine gas-air properties for combustor
gamma_c = gamma(0.5 + 1.0 + theta_A0);  // PM 26/4/97 Eq. 11
cp_c = cp(0.5 + 1.0 + theta_A);  // PM 26/4/97 Eq. 14

// calculate temperature ratio among the combustor
Tt3 = Tau_c(theta_A0 + Tg);  // PM 26/4/97 Eq. 11
Tt4 = TAU_c(theta_A0 + Tg);  // PM 26/4/97 Eq. 14

// determine average burner specific heat
Cpb = cp(0.5 + 1.0 + theta_A());

// determine burner fuel/air ratio
E_b = cp_b(Tt4 - Tt4) / (Q_R + Etu_b - Cpb(Tt4));  // PM 28/4/97 Eq. 15
tau_b = Tt4/Tt3;

// determine temperature ratio across the fan
Tt1 = Tau_c(1.0 + theta_A0(gamma_c) - 1.0/theta_A + 1.0);  // PM 26/4/97 Eq. 14
Tt3 = TAU_c(theta_A0 + Tg);

// determine average burner specific heat, assume drop in temp roughly equal to compressor temp rise
Cpb = cp(Tt4 - 0.5 + Tt3(theta_A0 + Tg));  // Tt3 - (theta_A0 + Tg) gamma_c = gamma(Tt4 - 0.5 + Tt3(theta_A0 + Tg), Ra0);

// determine temperature ratio across the turbine
Tt4 = 1.0 - Cpb(theta_A0, Tt4, tau_c(1.0) + (tau_c(1.0)) / (Cpb * tau_d_c(Pa_c, Tt4, 1.0));  // PM 26/4/97 Eq. 15
Tt6 = tau_u_Tt6;

// check if afterburner is on or off and apply appropriate condition
if(A3000ff = 0) {
    Tt6 = Tt6 = Tt6_max;
    Cpa = cp(0.5 + (Tt5 + Tt6));
    fa = Cpa(1 + Tt6) / (Q_R * etu_a * Cpa(Tt6));  // PM 26/4/97 Eq. 20
}
else { // afterburner is on
    Tt6 = Tt6 = Tt1 * Tt4;
    Cpa = cp(Tt4);
    fa = 0.5;  // no heat is used
}

// we could now calculate f
f = fa + E_b;

// determine the pressure ratio across the turbine

Appendix C.2 Propulsion Model: Class TurboJet1D Implementation File: 1DPropModel.cpp
p.t = pow((1.0-(1.6-0.6*c.t0)/c.t0), gammat/(gammat-1.0)); // PP 28/4/97 Eq. 22

// determine the stagnation pressure ratio across the diffuser
p.0 - P.0 diffuser(M0, T0, p0);
p0 = pow(del-np*p0*np_c*p0*np_c*p0, T0); // PP 28/4/97 Eq. 23

// the two different exit conditions must now be considered.
//
// if the nozzle is convergent, we need to check if it is choked. We thus consider
// if W0 >> W0c, we apply the convergent-convergent solution, and vice versa.
// the Mach number exceeds 1, we thus apply the choked solution, otherwise we use the
// fully expanded solution.

**** evaluate the fully expanded case: ****/

// estimate cp0 by linear extrapolating T0, using turbine constants

gammat = gamma;
cp0 = Cp0;
p7 = p0;
W0sq = (2*eta*r*c*cp0^2*1.0-pow(p7/p0, (gammat-1.0)/gammat)); // PP 28/4/97 Eq. 24
T7 = T0 - W0sq/(2*cp0);

// now calculate Cp0

gammaw = gammat*(T0/T0^2); // PP 28/4/97 Eq. 25

// now calculate Cp0

cp0 = Cpl[0]*T0^2; // PP 28/4/97 Eq. 26

W0sq = (2*eta*r*c*cp0^2*1.0-pow(p7/p0, (gammaw-1.0)/gammaw)); // PP 28/4/97 Eq. 27
T7 = T0 - W0sq/(2*cp0);
M7 = sqrt(W0sq/(gammaw*cp0)) // PP 28/4/97 Eq. 28

**** now evaluate convergent, choked condition: M7 > 1 **
if ( (nozzleprimary == CONVERGENT) && (M7 >= 1.0) )

T7 = 2.0*Cpl*T0/(gammaw*cp0+2.0*cp0); // PP 28/4/97 Eq. 29
W0sq = gammaw*cp0+4.77;
p7 = p0*(pow((1.3+0.029*2.0*eta*r*c*cp0*T0), gammaw)/(gammaw-1.0)); // PP 28/4/97 Eq. 30

**** Converge the upstream stream exit now ****/
// fully expanded case
p0 = Sp0
p0 = pi * pi_a
```c
using gamma = (gamma-1.0)/gamma; // 29/4/87 Eq. 29
T9 = T8 * gamma/2.0; // 28/4/87 Eq. 31

// We have now calculated everything that we need.

double phi = p/(R*T9); double u6 = M*sqrt(gamma*U6) * T9; double u7 = q/(h7*phi); double u8 = u7 * u8 * u9;

// Calculate the engine thrust in Newtons (N).

int Turbojet1D::Thrust(double N0, double lmbda, double p6, double phi, double u6, double u7, double u8, double u9, double v0, double a0, double alfa, double gamma, double M, double phi, double u8, double u9) { 
```

```c
*/
N0 = flux * Area * Mach / (gamma-1.0)
phi = Mach / speed_of_sound
p6 = free_stream_ambient_pressure
v0 = Angle_of_attack
f = flow_speed
M0 = Mach_number
v = thrust
rho = fuel_consumption
```
Return value: Error evaluation:
E = Error
I = Success
*/

/* Refer to ThrustFlowChannel for a flow chart of
the procedure used in this routine. */

double pl_d_crit;
double pl_d_act;
double mdot;
double mdota;
double mdotcm;
double mdotcmx;
double mdotmin;
double mdotcmin;
double AT_ma;
double AS_ma;
// double p_ma;
double l;

int retval;

// create input fluid objects, e.g., intake stream
CFileGasStream GasStream[6][6];
GasStream[5][0];
GasStream[5][0];

// calculate critical mode pressure recovery
pl_d_crit = PCCdiff->P1X(0), &GasStream[5][0];
pl_d_crit = Pd_4DA[00, 00, 00, 00];

// create the exit fluid objects from the ambient gas
CFileGasStream GasStream[6][6];
CFileGasStream GasStream[6][6];

// determine exit conditions
CycleThermodynamics(M, A7, p0, AM0off, pl_d_crit, GasStream[5][0], GasStream[5][0], A7_ma, A9_ma, F, F);

// the ratio of exit area to mass flow through the gas generator has now been calculated
// Since we know the maximum primary mass flow, we can calculate the mass flow rate.
// Mass flow possible through the engine - Total exhaust flow rate
m_ductmax = m_Amax/A_nu;
if(m_ductmax > m_ductmax) m_ductmax = m_ductmax;

m_ductmax = m_ductmax*(1-beta);
if(m_ductmax > m_ductmax) m_ductmax = m_ductmax;

// Determine the minimum flow possible through the engine.

m_ductmin = m_Amin/A_nu;
if(m_ductmin > m_ductmin) m_ductmin = m_ductmin;

// Create an in-take gas stream

double m_ductactual;

// Now see how the model is really performing

p_d_4 = p_d_4 + r_inletStream.c_d + m_ductactual.c_d + m_ductactual.c_d;

// Determine exit conditions

double f_p_nu;
f_p_nu = CylThermodynamics(C_d, atm.C_d, p0, M_d, pi_d_4, GdStream, GdStream, A_nu, A_nu, A_nu, E_nu, F);

// Determine the exit mass flow rates

double m_duct?, m_duct9;

// Calculate m_duct

m_duct9 = m_duct + m_duct?

// m_duct9 = m_ductactual + m_duct?

// Determine the exit areas

double A7, A9;

A7 = A7 + m_duct9;
A9 = A9 + m_duct9;
// dump some stuff to an output list
o_s7 = A7;
o_s9 = A9;
o_s9ma = A7ma;
o_mdotarenmax = mdotarenmax;
o_mdotmax = mdotmax;
o_mdotarenmin = mdotarenmin;
o_mdotmin = mdotmin;
o_mdotactual = mdotactual;
o_p1_d = p1_d_act;
o_T87C = f_T87C;

// now calculate the thrust
F = F_ma * miota;

return 1;
}

double Turbojet1D::CycleThermodynamics(double M0, double T0, double p0, RPMnum RPMOff, double p1_d, CIdealGasStreams act, CIdealGasStreams ext, doubles A7 ma, doubles A9 ma, doubles 7 ma, doubles f)
{
    // specific heat ratio
    double gammaC; // compressor
    double gammaAb; // afterburner
    double gammaCp = 1.4; // compressor
    double gammaTp = 1.3; // turbine
    double Cps; // specific heat at constant pressure
    double Cpa; // specific heat at constant pressure
    double Cpp = 1.0; // nozzle exit
    double theta0;
    double del0;

    // stagnation temperature ratio
    double tau_c; // compressor
    double tau_t; // turbine
    double tau_f = 1.0; // fan
    double tau_c; // turbine

    //...
double T3; // compressor exit
double T4; // turbine entry
double T5; // turbine exit
double T6; // afterburner exit
double T7; // afterburner exit
double T8; // bypass fan exit
double T9; // bypass nozzle exit

// other
double lb; // fuel/air ratio for primary burner
double lb; // fuel/air ratio for afterburner

double pt6;
double pt4;
double pt9;
double pt8;

double u eq; // exit velocity of primary stream
double u9eq; // exit velocity of bypass stream

double NS;

double Td;

evaluate stagnation conditions of intake air
theta_t = Td_Td(Nd, gamma(10, Ra)); // Eq 28/4/97 Eq 11

dl t a = pc_t_p(Nd, gamma(10, Ra)); // Eq 28/4/97 Eq 12

evaluate avg. air properties for compressor

gamma = gamma(0.3*Td(10, C-cham3)); Ra;
Cpt = Cp(0.85*0.1*0.04*theta_t);

evaluate temperature ratio across the compressor

theta_c = (p*V*V_c*gamma-c+1.0)/gamma_c - 1.0; // Eq 28/4/97 Eq 11

theta_c = (Tc*V*V_c*gamma-c+1.0)/gamma_c - 1.0; // Eq 28/4/97 Eq 14

evaluate average burner specific heat
Cp_b = Cp(C_b - Theta_c);

evaluate burner fuel/air ratio

n_b = C_p_b*(T_d - Tf)/10*eta_b - C_p_b*theta_c; // Eq 28/4/97 Eq 15

T_s_b = T_t_d(Tf); // calculate temperature ratio across the fan
tauT = (p0*(pl, (gamma - 1.0))/gamma) - 1.0)/etaT - 1.0; // PP 284/97 Rq. 77

ItT = tauT*thetaT; // determine average turb air specific heat, assume drop in temp roughly equal to convective temp rise

COp = CpT*(0.5*(ItT-thetaT)); // T3 = tauT*70

gammaT = gamma(ItT = 0.5*(ItT-thetaT), thetaT); // determine temperature ratio through the turbine

tauT = 1.0 + COp*(delta*(tauT-1.0)+(tauT-1.0))/COp*(tauT-1.0*(1.0+tauT)); // PP 284/97 Rq. 15

T35 - tauT*T4; // check if afterburner is on or off and apply appropriate condition

if(AAONoff == ON) {
    T35 = T35 + T4; // Cpa = Cp(0.5*T35+2); T35 = (Cpa-T5)*(T35-T5)/((CR(itab_a Cpa*TL6)); // PP 284/97 Rq. 20
}
else {
    // determine the pressure ratio across the turbine
    pl_b = p0*(1.0 + tauT)*etaT; // PP 284/97 Rq. 32

    // determine the stagnation pressure ratio across the diffuser
    pl_d = pl_b - itab; // PP 284/97 Rq. 53

    pl_b = p0*1.8*0.8*(Cpa*(1.0+B)*pl_c*pl_b*pl_c*pl_b); // PP 284/97 Rq. 53

    // the two different exit conditions must now be considered:

    if the model is convergent, we need to check if it is choked. We thus consider
    it so if it is either a convergent divergent nozzle and check the exit Mach number.
    If the Mach number exceeds 1, we then apply the choked solution, otherwise we use the
    fully expanded solution.

    **** evaluate the fully expanded case: ****

    estimate Cpa by firm motivation T3, using turbine constants

    gammaC - gammaT;

Appendix C-2 Propulsion Model (Open TurboArt) Implementation File: 1DPropModel.cpp
Cpa = Cpt;
p7 = pc;
T79 = \frac{\gamma_{u} \cdot (Cpa \cdot Tt6 \cdot 1.0 \cdot \text{pow}(p7/p_t6, (\text{gamma} - 1.0)/(\text{gamma})))}{\gamma_{u} \cdot \text{pow}(p7/p_t6, (\text{gamma} - 1.0)/(\text{gamma}))}; // PP 28/4/97 Eq. 24
T7 = Tt6 - u79g/(2*Cpt);

// new recalculate CEG
gamma = gamma(1.5 \cdot Tt6 \cdot Tt11, \text{Kail});
Cpa = \frac{\gamma_{u} \cdot (Cpa \cdot Tt6 \cdot 1.0 \cdot \text{pow}(p7/p_t6, (\text{gamma} - 1.0)/(\text{gamma})))}{\gamma_{u} \cdot \text{pow}(p7/p_t6, (\text{gamma} - 1.0)/(\text{gamma}))}; // PP 28/4/97 Eq. 24
T7 = Tt6 - u79g/(2*Cpt);

M7 = sqrt(25U9/\text{(gamma \cdot Kail \cdot T})) ; // PP 28/4/97 eq. 26

//** now evaluate converged, choked solution only if M7 = 1.0 */
if (\text{nozzle bypass = CONVERGED} \&\& M7 = 1.0) : 
T7 = 2.0 \cdot Cpa \cdot Tt6/(\text{gamma} \cdot \text{Kail} + 2.0 \cdot \text{Cpa}); // PP 28/4/97 Eq. 27
u79g = \text{gamma} \cdot \text{Kail} \cdot T7;
p7 = p_t6 \cdot \text{pow}(1.0 - \text{u79g}/(2.0 \cdot \text{eta} \cdot \text{Cpa} \cdot Tt6), \text{gamma} \cdot (\text{gamma} - 1.0)); // PP 28/4/97 Eq. 28

//*** Consider the nozzle stream exit now ***
// fully expanded case
p9 = pc;
p9a = p9 \cdot \text{d} \cdot \text{deg} \cdot \text{L}^2 \cdot \rho_9;
u9g = \frac{(2 \cdot \text{eta} \cdot \text{ub} \cdot \text{Cpa} \cdot Tt6 \cdot 3.0 \cdot \text{pow}(p9/p_t9, (\text{gamma} - 1.0)/(\text{gamma})))}{\gamma_{u} \cdot \text{pow}(p9/p_t9, (\text{gamma} - 1.0)/(\text{gamma}))}; // PP 28/4/97 Eq. 29
T9 = Tt8 - u9g/(2*Cpa);
M9 = sqrt(25U9/\text{(gamma \cdot Kail \cdot T4)});

//** now evaluate converged, choked solution only if M9 = 1.0 */
if (\text{nozzle bypass = CONVERGED} \&\& M9 = 1.0) : 
T9 = 2.0 \cdot Cpa \cdot Tt6/(\text{gamma} \cdot \text{Kail} + 2.0 \cdot \text{Cpa}); // PP 28/4/97 Eq. 31
u9g = \text{gamma} \cdot \text{Kail} \cdot T9;
p9 = p_t9 \cdot \text{pow}(1.0 - \text{u9g}/(2.0 \cdot \text{eta} \cdot \text{ub} \cdot \text{Cpa} \cdot Tt6), \text{gamma} \cdot (\text{gamma} - 1.0)); // PP 28/4/97 Eq. 32

// we have now calculated everything that we need.
double rhoC = pc/(\text{Kail} \cdot \text{T} \cdot \text{T0} \cdot \text{Kail});
double &; : M9 = sqrt((\text{gamma} \cdot \text{T9} \cdot \text{Kail} \cdot \text{T0} \cdot \text{Kail});
double uT = sqrt(u7^2);
double u0 = (u0sq>0)?sqrt(u0sq):0.0;

A7 va = (1.0f*p0+T7)/(-rho0*p0^2+T0+u7^2); // PP 28/4/97 Eq. 8
A7 ma = u0>0?(beta*(p0*T7)/(rho0+rho0*T0)*u7^2):0.0; // PP 28/4/97 Eq. 9

F,ma = (1.0f*:F7 + F7beta:u0 + A7 ma^2+u7-p0); beta=u0 + A7 ma^2*p0-p0); // PP 28/4/97 Eq. 4;

ex7C(T7);
ex7.p[p7];
ex7.u(u7);
ex8.C(T8);
ex9.p[p9];
ex9.u(u9);

return f/F_m;
}

Turbojet1D::Turbojet1D(
{
    pCCIdiff = NULL;
    m_pid_const = 0.95;
}
CConIntake * Turbojet1D::CreateIntake(double ConeAngle, CIdexGasStream *Fluid, const char *IntName)
{
    // create the intake
    pCCIdiff = new CConIntake(ConeAngle, Fluid, IntName);
    return pCCIdiff;
}
Appendix C-3  Propulsion Simulation Code

C.3.1 Program: TSFC

```c
#include "\1\TripModel1.hpp"
#include "\1\\2\Data\1\1\TripInput.hpp"

int main( int argc, char *argv[] )
{

    if ( argc < 2 ) {
        cout << "No input file specified on command line.\n";
        cout << "Usage: TSFC tripfile\n" << endl;
        return 1;
    }

    ifstream TurbojetData(argv[1], ios::in | ios::nocreate, ifstream_h_read);

    if (!TurbojetData) {
        cout << "Input file " << argv[1] << " not found\n";
        cout << "INPUTFILE NOT GOOD\n" << endl;
        return 2;
    }

    double M0_2;
    double M0_2;
    double T0;
    double p0;
    int alterflue;
    // static Pressure Ratio
    double p1_r;  // bypass fan pressure ratio
    double p2_r;
```
// double pi_e_start; // compressor pressure ratio
// double pi_e_end; // compressor pressure ratio
double pi_d_start; // compressor pressure ratio
double pi_d_end; // compressor pressure ratio
double pi_h; // burner pressure ratio
double pi_a; // afterburner pressure ratio
// T's
double eta_f; // fan
double eta_c; // compressor
double eta_d; // burner
double eta_r; // turbine
double eta_a; // afterburner
double eta_nb; // bypass nozzle

double hae,
// dynamic temperature
double T_a_max; // max turbine inlet temp
// double T_d_end;

double T_d_max; // max afterburner temp
// fuel
double CP; // lower heating value of jet fuel
// nozzle type
int nozzlePrimary;
int nozzleBypass;

cchar *outname[10];

TurboJetData >> new FileInputVar("PID", &kpi дл);
TurbojetData >> new FileInputVar("PID_end", &kpi_end);
TurboJetData >> new FileInputVar("TO", &t0);
TurboJetData >> new FileInputVar("p0", &p0);
TurboJetData >> new FileInputVar("A2 ON CRP", &afterBurner);

TurboJetData >> new FileInputVar("pi_a", &pi_a);
// ValveJetData >> new FileInputVar("pi-d-start", &pi_d_start);
// ValveJetData >> new FileInputVar("pi-d-end", &pi_d_end);
TurboJetData >> new FileInputVar("pi-c", &p1_c);
TurboJetData >> new FileInputVar("p1", &p1);
TurboJetData >> new FileInputVar("p1_a", &p1_a);

TurboJetData >> new FileInputVar("pi-d-start", &pi_d_start);
TurboJetData >> new FileInputVar("pi-d-end", &pi_d_end);
I
double TSFC[2];
double phi[8];

double pi_d;

// dataout = "pi-climNum-l+t/m.a-1+m=+z/tP/m.a-2+m="

for(pi_e=pi_e_start;pi_e<pi_e_end;pi_e+=pi_e_step)/100.) {
    Tsi(Jcl.pi,vul"C")
    TSFC[0] = Testjet.TURBOM 1, T0, p0, (Tsaccl1.DS.Ensun.discharge, F_m(m0[0], e[0]));
    TSFC[1] = Testjet.TURBOM 2, T0, p0, (Tsaccl1.DS.Ensun.discharge, F_m(m0[1], e[1]));
    Gcalcout << pi_e << "(\"" << TSFC[0] << "\"" << pi_m_data[0] << "\"" << TSFC[1] << "\"" << pi_m_data[1] << "\"" << endl;
}

// dataout = "Tsi\""TSFC [N\""NO_\""=""]/t\""m.a << "(\""m0\""2 << "(\""m1\""2 << "(\""m0\""2 << "(\""m1\""2 << "(\""m0\""2 tool

for(Tsi=Ti_e_start;Ti<ti_e_end+ti_e_step)/100.) {
    Tsi(Gcl.Ti,4(Ti1)
    TSFC[0] = Testjet.TURBOM 1, T0, p0, (Tsaccl1.DS.Ensun.discharge, F_m(m0[0], e[0]));
    TSFC[1] = Testjet.TURBOM 2, T0, p0, (Tsaccl1.DS.Ensun.discharge, F_m(m0[1], e[1]));
    Gcalcout << Ti << "(\"" << TSFC[0] << "\"" << pi_m_data[0] << "\"" << TSFC[1] << "\"" << pi_m_data[1] << "\"" << endl;
}

// dataout = "(\""m\""2 \""m\""2 << (\""m\""2 \""m\""2 << (\""m\""2 tool

for(pi_depi_d_start;pi_depi_d<pi_depi_end;pi_depi_d+=pi_depi_d_step)/100.) {
    Testcl.m_2LA_com = pi_d;
    TSFC[0] = Testjet.TURBOM 1, T0, p0, (Tsaccl1.DS.Ensun.discharge, F_m(m0[0], e[0]));
    TSFC[1] = Testjet.TURBOM 2, T0, p0, (Tsaccl1.DS.Ensun.discharge, F_m(m0[1], e[1]));
    Gcalcout << pi_d << "\"" << TSFC[0] << "\"" << pi_m_data[0] << "\"" << TSFC[1] << "\"" << pi_m_data[1] << "\"" << endl;
}
C.3.2 Program: Thrust

```c
#include "../PropModel.h"
#include "../DatafileInput\DatafileInput.h"
#include "../\CodeIntake\ToneIntake.h"
#include <iostream>

int main(int argc, char *argv[]) {

    if (argc < 1) {
        cout << "No input file specified on command line.
        exit(1); return 0;
    }

    InputFile TurbojetData(argv[1], ios::in); TurboJetFile ifstream::sh_read);

    if (!TurbojetData.good()) {
        cerr << "Error with input file in argv[1]" << endl;
        return 2;
    }

    char *expansion[] = { "COM_BLOCKED", "COM_DETACHED", "COM_TERMINATED", "COM_CRITICAL";
```
`#OM_SUPERCRITICAL`

```c
enum {ALTITUDE, TEMPERATURE, PRESSURE, DENSITY, GRAVACC, VELOCITY, TWC, COMPCV
};

double M0_start;
double M0_end;
double M0_step;
// double TWC;
// double pW;

double ALt;
int afterburner;

// Static Pressure Ratio
double pl_f; // bypass fan pressure ratio
double ps;   // compressor minimum ratio
double pb;   // bypass pressure ratio
double ps_a; // afterburner pressure ratio

// Static Temperature
double eta_f; // fan
double eta_s; // compressor
double eta_b; // bypass

double eta_t; // turbine

double eta_a; // afterburner

double eta_r; // nozzle

double eta_n; // bypass nozzle

double beta; // bypass ratio

// Maximum Temperature
double Tmx_f; // max bypass fan inlet temp

double Tmx_s; // max compressor temp

double Tmx_b; // max bypass temp

// maximum fuel flow and afterburner capability
double maxfuel; // maximum fuel flow the engine is capable of

double maxeta; // max fuel at the jet is feeding current

// Fuel
double Qf; // lower heating value of jet fuel

// Nozzle types

int nozzleprimary;
int nozzlebypass;

// Exit words
```

---

Appendix G-3: Propulsion Simulation Code
double A_t_max;
double A_t_min;
double A_t_medium;
double A_t_lat;

// reference area
double S_w;

// Input Data
double con_angle; // angle of take take some degrees!
char latitude[812]; // looks like for the latitude

double latitude;
double longitude;
double altitude;
double lat_min;
double lat_max;

char longitude[812]; // earth's data lookup
char latitude[812];

TurbojetData >> new FileInputVar("Hi Start", A_t_start);
TurbojetData >> new FileInputVar("LO End", A_t_end);
TurbojetData >> new FileInputVar("LO-Start", A_t_start);

TurbojetData >> new FileInputVar("Hi", A_t)

TurbojetData >> new FileInputVar("LO-CW-CHR", EarthRadius);
TurbojetData >> new FileInputVar("Latitude", A_t);

TurbojetData >> new FileInputVar("pi", A_t);

TurbojetData >> new FileInputVar("pi- start", A_t_start);

TurbojetData >> new FileInputVar("pi-end", A_t_end);

TurbojetData >> new FileInputVar("pi-c", A_t_c);

TurbojetData >> new FileInputVar("pi-c", A_t_c);

TurbojetData >> new FileInputVar("pi-d", A_t_d);

TurbojetData >> new FileInputVar("BetA", A_t_Beta);

TurbojetData >> new FileInputVar("BetA", A_t_Beta);

TurbojetData >> new FileInputVar("BetA", A_t_Beta);

TurbojetData >> new FileInputVar("BetA", A_t_Beta);

TurbojetData >> new FileInputVar("BetA", A_t_Beta);

TurbojetData >> new FileInputVar("BetA", A_t_Beta);
```cpp
int main()
{
    // Some code here
    return 0;
}
```
cout << "Atmosphere data loaded" << endl;

void CreateThrottle(Turbine &T) {
    // Create the turbine
    ThrottleID Throttle;
    Air T, AirIn;
    Air T, Atmosphere, LookUpAlt, (long TEMPERATURE);
    Credos Cstream FreeStream(Air);
    FreeStream, (Atmosphere, LookUpAlt, (long TEMPERATURE));
    FreeStream, p(Atmosphere, LookUpAlt, (long TEMPERATURE));
    FreeStream, M(0.0);

    // Create the interface
    CConStartakeCool, - T, CoolStartake, (DIR:lookout[2], ), m2FreeStream, X, CoolStartake;

    // pStartake, AutoDesign(0.0, 2.1, 1.01, 2.3);
    // pStartake, AutoDesign(1.0, 1.6, 0.29, 2.4);
    // pStartake, AutoDesign(0.2, 0.0, 1.01, 2.1);
    pStartake, AutoDesign(mldol, pMldol, pRho1, pMmax);
    pStartake, AutoDesign(mldol, pMldol, pRho1, pMmax);
    pStartake, AutoDesign(mldol, pMldol, pRho1, pMmax);
    pStartake, AutoDesign(mldol, pMldol, pRho1, pMmax);

    TestJet, m_E, (pi, 2);
    TestJet, m_E, (pi, 2);
    TestJet, m_E, (pi, 2);
    TestJet, m_E, (pi, 2);

    TestJet, BypassRatio(beta);

    TestJet, m_E, (beta, 2);
    TestJet, m_E, (beta, 2);
    TestJet, m_E, (beta, 2);
    TestJet, m_E, (beta, 2);

    TestJet, A7, (A7, max);
    TestJet, A7, (A7, max);
    TestJet, A7, (A7, max);
    TestJet, A7, (A7, max);

    TestJet, A, (A, min);
    TestJet, A, (A, min);
    TestJet, A, (A, min);
    TestJet, A, (A, min);

    TestJet, m_E, (Cmax);
    TestJet, m_E, (Cmax);
    TestJet, m_E, (Cmax);
    TestJet, m_E, (Cmax);
Inside: /T_g::textaux;

int dec::nextStateflow(void);

TestDec::halt();
TestDec::mainType(4);TestDec::mainType(0);TestDec::mainType(1);
TestDec::mainType(0);TestDec::mainType(1);
TestDec::mainType(0);TestDec::mainType(1);
TestDec::mainType(0);TestDec::mainType(1);

Of strangely odd things, if you could figure out:

// ok for now

double TSC[2];
double Tnuwta[2];
double T[2];
double _ma;

double p0;
p0 = Atmosphere:LookupAlt(1, 1500); PSSSURF;

// here more info

dataout << "Attitude\n" << Alt;
dataout << "Reference\n" << Ett << endl;
dataout << En << u << dx << endl;
dataout << En << u << dx << endl;
dataout << Ud << Ud << Ud << Ud << ud << endl;
dataout << H0 << endl;
dataout << "Well\" << "\" << endl;
dataout << H0 << endl;
dataout << H0 << endl;
dataout << H0 << endl;
dataout << H0 << endl;
dataout << H0 << endl;
dataout << H0 << endl;

double H, T, nu, CT;

for(M=0; M<100; M++)
{
    TestJet::Print(T, 200, p0, 0.3, 1.5, (true)/*Jet\"&\"Alstrom\"*/); TestJet::mainType(T, nu, CT);
    CC = T_{/}158.5;FreeStream(x, t, x, y, z, phi, \theta, \r, \psi, \Omega, \chi, \varphi, \kappa, \xi, \zeta);

dataout << H << "\n" << T << "\n" << nu << "\n" << CT << "\n" << endl;
dataout << TestJet::mainType(H, T) << endl;
    TestJet::mainType(T, nu, CT);
}

Program: Thrst

Appendix D-3 Propulsion Simulation Code
C.3.3 Program: CalcTSDiagram

```c++
#include "\:IDropModel\:hpp"
#include "\:\:DatafileInput\:hpp"

int main( int argc, char *argv[]) {

    if(argc<2) {
        cout << "No input file specified on command line.\n";
        cout << "Usage: IDropModel\:s inputfile\" << endl;
        return 2;
    }

    // ... (rest of the code)
}
```
```c
// Code snippet
```
TurbobjData = new FileInputStream("spi_b");
TurbobjData = new FileInputStream("spi_a");
TurbobjData = new FileInputStream("meta", false);
TurbobjData = new FileInputStream("eta-f", meta_f);
TurbobjData = new FileInputStream("eta-o", meta_o);
TurbobjData = new FileInputStream("eta-n", meta_n);
TurbobjData = new FileInputStream("eta-0", meta_0);
if(TurbobjData.readData() < 0) {
    cout << "Error retrieving data" << endl;
    return 3;
}
TurbobjData.heleraVardir();
TurbobjData.close();

// create the matrix
TurbobjB instact;
TestSet pi (pi_f);
TestSet psi (psi_a);
TestSet psi (psi_b);
TestSet ppsi (ppsi_a);
TestSet hyp (hyp);
TestJet.mks_nb(eta_nb);
TestJetIFT_4(iffmax);
Joucute.ift_6(iffmax);
TestJet.PNFQIQR;
TestJet.MainNozzleType((Nozzle_num)nozzleprimary);
TestJet.AFNozzleType((Nozzle_num)nozle bypass);
ofstream dataout(filename, ios::out); filebuf::openprot);
int result = TestJet.DrawTs(MD, TO, p0, (TurbojetID::ARBnum)afterburner, 1, dataout);
dataout.close();
return (result==1); 4;
Appendix D: Performance Estimation Source Code

Appendix D-1  Sustained Turn Rate

```c
#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>

int main(void) {
    double M;
    double alt;
    double Clng;
    double Clg;
    double M1;
    double Cl;
    double C2;
    double C1;

    // char *name[2];
    // double Mlist[] = {0.2, 0.8, 1.6, 2.4, 3.2, 4.0, 4.8, 5.6, 6.4, 7.2};
    double AltList[] = {0.0, 2000.0, 4000.0, 6000.0, 8000.0, 10000.0, 12000.0, 14000.0, 16000.0, 18000.0, 20000.0, 22000.0};

    double *Mlist;
    double *AltList;

    int m, n;
    int *Mlist, *AltList;
    char restmach[034]; // buffers to hold next form of Mach and Alt lists
```
```c
char posAlt[1024];

int retultat=0;

char infname[512];    // input file
char nullname[512];   // output file
char stffname[512];   // atmosphere file
char mmsfname[512];   // model input data new format
char prppname[512];   // simulation data

double altstep;
double Mstep;

double octary;
double tend;
double notep;

// double tentep;

double p;
double g;
double gamma;
double rho;
double maes;
double Gref;

double STR, STRnew;

int errval;
int UKData=0;

enum {ALTITUDE, TEMPERATURE, PRESSURE, DENSITY, CRWCY, VISCOSITY, XNK, SONCY };
enum [MACHNUMBER, THRST, CC, PDH, PHOT];

// Set the Type: File

cout << "Enter the name of the input file: ";
cin >> infname;

Inputfile ::Data(infname);

// check that input file is clear
if(stffname.bad () | infname.good () )
    cout << "Bad input file: " << infname << endl;
```
return 0;

// Extract the first bit of data
InData >> new FileInputVar("SOLSTICE", countWeeks);
InData >> new FileInputVar("ATMOSPHEREDATA", atmname);
InData >> new FileInputVar("PROFSTANDARD", profnamed);
InData >> new FileInputVar("MUR-MUR", mur);
InData >> new FileInputVar("MACHNAME", machname);
InData >> new FileInputVar("NMLT", nmlt);
InData >> new FileInputVar("C/RPOS", cRpos);
InData >> new FileInputVar("RANKFORM", rankform);
InData >> new FileInputVar("RANKNAME", rankname);
InData >> new FileInputVar("MAXTRANS", maxtrans);
InData >> new FileInputVar("MAXSTAR", maxstar);
// get necessary variables
// InData >> new FileInputVar("AERO", aero);
InData >> new FileInputVar("AVG", avgcnt);
// InData >> new FileInputVar("MST", mstcnt);
// InData >> new FileInputVar("MYP", myp);
// InData >> new FileInputVar("MARP", marp);
// InData >> new FileInputVar("MSTAR", mstarp);
InData >> new FileInputVar("NO", ncnt);
InData >> new FileInputVar("NO1", n1cnt);
InData >> new FileInputVar("NO2", n2cnt);
InData >> new FileInputVar("NO3", n3cnt);
InData >> new FileInputVar("NO4", n4cnt);
InData >> new FileInputVar("NO5", n5cnt);
// get other data needed
InData >> new FileInputVar("MSTB", mstb);
InData >> new FileInputVar("MAG2", mag2);
InData >> new FileInputVar("MAG3", mag3);
InData >> new FileInputVar("MAG4", mag4);
// generate the 'end'
InData.ReadData();
// close the input file
InData.Close();

// generate the value of Altitude and Mach Number
char *tempstr1, *tempstr2;
Altitude = new double_val;
tempptr = ptAlt;
for(Altint=0;Altint<Altint++;)
    Altint [Altint] = utilod (tempptr, &tempptr2);
    tempptr = tempptr2;
}

MList = new double[OK];
tempptr = posMach;
for(Mech=0;Mech< Mech++;)
    MList [Mech] = utilod (tempptr, &tempptr2);
    tempptr = tempptr2;

// Create a 4D lookup table to hold the Aero Data at Mach number and Altitude
LookupAI AeroData (MList, AltList, Mech, nAlt, aeroset);

if (Interval = AeroData (LastError)) { // error not --- but assignment
    error << "Error creating lookup table. " << error;
}

cost << "All Data Loaded" << endl;

Exiting...


```c
const float m = 1.0;
const float s = 0.5;
const float k = 1.0;
const float r = 1.0;
const float q = 1.0;
const float t = 1.0;
const float g = 1.0;
const float a = 1.0;
const float b = 1.0;
const float c = 1.0;
const float d = 1.0;
const float e = 1.0;
const float f = 1.0;
const float u = 1.0;
const float v = 1.0;
const float w = 1.0;
const float x = 1.0;
const float y = 1.0;
const float z = 1.0;

// Compute the value of k for a given value of m.
const float compute_k(float m) {
    return m * k;
}

// Compute the value of s for a given value of m.
const float compute_s(float m) {
    return m * s;
}

// Compute the value of k for a given value of m.
const float compute_r(float m) {
    return m * r;
}

// Compute the value of q for a given value of m.
const float compute_q(float m) {
    return m * q;
}

// Compute the value of t for a given value of m.
const float compute_t(float m) {
    return m * t;
}

// Compute the value of g for a given value of m.
const float compute_g(float m) {
    return m * g;
}

// Compute the value of a for a given value of m.
const float compute_a(float m) {
    return m * a;
}

// Compute the value of b for a given value of m.
const float compute_b(float m) {
    return m * b;
}

// Compute the value of c for a given value of m.
const float compute_c(float m) {
    return m * c;
}

// Compute the value of d for a given value of m.
const float compute_d(float m) {
    return m * d;
}

// Compute the value of e for a given value of m.
const float compute_e(float m) {
    return m * e;
}

// Compute the value of f for a given value of m.
const float compute_f(float m) {
    return m * f;
}

// Compute the value of u for a given value of m.
const float compute_u(float m) {
    return m * u;
}

// Compute the value of v for a given value of m.
const float compute_v(float m) {
    return m * v;
}

// Compute the value of w for a given value of m.
const float compute_w(float m) {
    return m * w;
}

// Compute the value of x for a given value of m.
const float compute_x(float m) {
    return m * x;
}

// Compute the value of y for a given value of m.
const float compute_y(float m) {
    return m * y;
}

// Compute the value of z for a given value of m.
const float compute_z(float m) {
    return m * z;
}

// Compute the value of k for a given value of m.
const float compute_k(float m) {
    return m * k;
}

// Compute the value of s for a given value of m.
const float compute_s(float m) {
    return m * s;
}

// Compute the value of k for a given value of m.
const float compute_r(float m) {
    return m * r;
}

// Compute the value of q for a given value of m.
const float compute_q(float m) {
    return m * q;
}

// Compute the value of t for a given value of m.
const float compute_t(float m) {
    return m * t;
}

// Compute the value of g for a given value of m.
const float compute_g(float m) {
    return m * g;
}

// Compute the value of a for a given value of m.
const float compute_a(float m) {
    return m * a;
}

// Compute the value of b for a given value of m.
const float compute_b(float m) {
    return m * b;
}

// Compute the value of c for a given value of m.
const float compute_c(float m) {
    return m * c;
}

// Compute the value of d for a given value of m.
const float compute_d(float m) {
    return m * d;
}

// Compute the value of e for a given value of m.
const float compute_e(float m) {
    return m * e;
}

// Compute the value of f for a given value of m.
const float compute_f(float m) {
    return m * f;
}

// Compute the value of u for a given value of m.
const float compute_u(float m) {
    return m * u;
}

// Compute the value of v for a given value of m.
const float compute_v(float m) {
    return m * v;
}

// Compute the value of w for a given value of m.
const float compute_w(float m) {
    return m * w;
}

// Compute the value of x for a given value of m.
const float compute_x(float m) {
    return m * x;
}

// Compute the value of y for a given value of m.
const float compute_y(float m) {
    return m * y;
}

// Compute the value of z for a given value of m.
const float compute_z(float m) {
    return m * z;
}

// Compute the value of k for a given value of m.
const float compute_k(float m) {
    return m * k;
}

// Compute the value of s for a given value of m.
const float compute_s(float m) {
    return m * s;
}

// Compute the value of k for a given value of m.
const float compute_r(float m) {
    return m * r;
}

// Compute the value of q for a given value of m.
const float compute_q(float m) {
    return m * q;
}

// Compute the value of t for a given value of m.
const float compute_t(float m) {
    return m * t;
}

// Compute the value of g for a given value of m.
const float compute_g(float m) {
    return m * g;
}

// Compute the value of a for a given value of m.
const float compute_a(float m) {
    return m * a;
}

// Compute the value of b for a given value of m.
const float compute_b(float m) {
    return m * b;
}

// Compute the value of c for a given value of m.
const float compute_c(float m) {
    return m * c;
}

// Compute the value of d for a given value of m.
const float compute_d(float m) {
    return m * d;
}

// Compute the value of e for a given value of m.
const float compute_e(float m) {
    return m * e;
}

// Compute the value of f for a given value of m.
const float compute_f(float m) {
    return m * f;
}

// Compute the value of u for a given value of m.
const float compute_u(float m) {
    return m * u;
}

// Compute the value of v for a given value of m.
const float compute_v(float m) {
    return m * v;
}

// Compute the value of w for a given value of m.
const float compute_w(float m) {
    return m * w;
}

// Compute the value of x for a given value of m.
const float compute_x(float m) {
    return m * x;
}

// Compute the value of y for a given value of m.
const float compute_y(float m) {
    return m * y;
}

// Compute the value of z for a given value of m.
const float compute_z(float m) {
    return m * z;
}

// Compute the value of k for a given value of m.
const float compute_k(float m) {
    return m * k;
}

// Compute the value of s for a given value of m.
const float compute_s(float m) {
    return m * s;
}

// Compute the value of k for a given value of m.
const float compute_r(float m) {
    return m * r;
}

// Compute the value of q for a given value of m.
const float compute_q(float m) {
    return m * q;
}

// Compute the value of t for a given value of m.
const float compute_t(float m) {
    return m * t;
}
```
```cpp
OutFile << ST << "\t";
OutFile << STnew << "\t";
OutFile << n << "\t";
OutFile << C1 << "\t";
OutFile << C2 << "\t";
OutFile << Cw << "\t";
OutFile << Ctvw << "\t";
OutFile << C1/Cw << endl;
break;
case Cutoff4D::erICUTOFFRNGE:
    OutFile << n << "\t";
    OutFile << "\t" << "\t";
    OutFile << Ctvw << "\t";
    OutFile << Ctvw << "\t";
    OutFile << Ctvw << "\t";
    OutFile << Ctvw << "\t";
    break;
default:  OutFile << n << endl;
    break;
}
return 1;
```
Appendix D-2  Specific Excess Power

```c
#include "./DPropModel/DDPropModel.h"
#include "./DatafillimpAth.tufile.spar.h"
#include "./DataInp Srk/Combintake.h"
#include "./Lockmp10.lockmp10.h"
#include <cmath.h>
#include <math.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    if(argc <= 1)
    {
        exit(0); // No input file specified on command line.
    }
    const char *inputfile = optarg; // Get inputfile
    return 1;
}

int TurbojetData(char *argv[], int argc, char *inFile, char *shutfile, char *loadfile)
{
    if(TurbojetData.good())
    {
        perror("Inputfile " + argv[1] + " ends.
        return 0 ;
    }
    char *upgrades[] = { "COM_SUBSONIC", "COM_SUPERSONIC"
                        "COM_CPLUSPLUS", "COM_CPPPLUS"
                        "COM_SUPERCRITICAL" };
    enum [ALTIITUDE, TEMPERATURE, PRESSURE, DENSITY, GRAVITY, VISCOSITY, RPK, SENSITY ];

Appendix D-2 Specific Excess Power
Program: CalTSDiagram
```c
// INPUT VARIABLES

double M0_start;
double M0_end;
double M0_step;
double Alt_start;
double Alt_end;
double Alt_step;

// double u0:
// double gb:
    int afterburner;

// Static Pressure Ratio
    double pi_1: // bypass fan pressure ratio
    double pi_2: // compressor pressure ratio
    double pi_b: // burner pressure ratio
    double pi_a: // afterburner pressure ratio

// Efficiencies
    double eta_c: // compressor
    double eta_b: // bypass
    double eta_a: // afterburner
    double eta_nb: // bypass nozzle

    double beta; // bypass ratio

// Maximum temperature
    double T4maxx: // max cooling intake temp
    double T4maxy: // max afterburner temp

// Maximum mass flow and suction capability
    double m_dot_max: // maximum mass flow the engine is capable of
    double m_dot_mbrm: // mass flow at M=0 due to pumping effect

// Fuel
    double CR: // lower heating value of jet fuel

// Nozzle types
    int nozzleprimary;
    int nozzlebypass;
```

Appendix D-2 Specific Excess Power

Program: CalcTSDiagram
// atmospheric data
TurbojetData >> new FileInputVar("ATMEnvironment", atmname);

// aerodynamic data stuff
TurbojetData >> new FileInputVar("N-MAX", 500);
TurbojetData >> new FileInputVar("MACROS", "psaMax1");
TurbojetData >> new FileInputVar("N-ALT", Alt.");
TurbojetData >> new FileInputVar("N-FLX", psaFLX);
TurbojetData >> new FileInputVar("ASR0NOVARMT", coreInnum);

// end run
TurbojetData >> new FileInputVar("MO Start", MO_start);
TurbojetData >> new FileInputVar("MO End", MO_end);
TurbojetData >> new FileInputVar("MO Step", MO_step);

// Altitude
TurbojetData >> new FileInputVar("Alt Start", ALT_start);
TurbojetData >> new FileInputVar("Alt End", ALT_end);
TurbojetData >> new FileInputVar("Alt Step", ALT_step);

// TurbojetData >> new FileInputVar("ATD", ATD);
TurbojetData >> new FileInputVar("pi", pi);
TurbojetData >> new FileInputVar("SO-NOVT", Softerburner);
TurbojetData >> new FileInputVar("pi-1", pi_1);

// TestobjetData >> new FileInputVar("pi-2-act", pi_2_act);
TurbojetData >> new FileInputVar("pi-2", pi_2);
TurbojetData >> new FileInputVar("pi-3", pi_3);
TurbojetData >> new FileInputVar("pi-4", pi_4);

TurbojetData >> new FileInputVar("Macla", Zeta);
TurbojetData >> new FileInputVar("Zeta", Beta);
TurbojetData >> new FileInputVar("eta", eta);
TurbojetData >> new FileInputVar("CTA-c", CTA_c);
TurbojetData >> new FileInputVar("CTA-b", CTA_b);
TurbojetData >> new FileInputVar("CTO-n", CTO_n);
TurbojetData >> new FileInputVar("CTO-a", CTO_a);
TurbojetData >> new FileInputVar("Maflow", sadcmax);
TurbojetData >> new FileInputVar("TurboMaxFlow", Sadcmax);

TurbojetData >> new FileInputVar("MaxT1", MT1max);
TurbojetData >> new FileInputVar("MaxT2", CTRmax);
A\nList(Alctelt) = std::tempotr1, Alctg0st1);
    tempotr1 = tempotr2+1;
    
    MList = new double[\n1];
tempotr1 = tempotr1;
for(Mont = 0; Mont < M; Mont++)
    MList[Mont] = std::tempotr1, shaper2); 
tempotr1 = tempotr1+1;

    // Create a 4D lookup table to hold the Aero Data at Mach Number and Altitude
    Lookup4D AeroData(MList, Alctelt, m, nAlt, aeroEndament);
    if(!error = AeroData.LastError())
        // note: not =, but assignment
        err << c.str(\"Error creating Lookup Table: \"");
    return 0;
}

cout << \"Aerodynamic data loaded\" << endl;

    // now create the atmosphere lookup
    LookupTable Atmosphere(aeroShape, LookupTable::LINEAR, LookupTable::CUB);
    cout << \"Atmosphere data loaded\" << endl;
    cout << \"All Data loaded\" << endl;

    // create the UAV:
    Turbojet1\nCenJet;

    // set up the intake
    IdealGas Air(287, 0.0);

    // Air: 1.0@1.0:
    Air.TAtmosphere.Lookups(0.0, (long)\TEMPERATURE\),
    IdealGasStream FreeStream(Air);
    FreeStream.T.Set();
    // FreeStream.push(); // p /\ 15 km
    FreeStream.TAtmosphere.Lookups(0.0, (long)\TEMPERATURE\),
    FreeStream.m.AirFlow.Lookups(0.0, (long)\PRESSURE\),
    FreeStream.m.WH.10.0;

    // create the state
```c

// p1, p2, T, lambda, mu, c, b

void calculatePower(float p1, float p2, float T, float lambda, float mu, float c, float b) {
    float v1 = p1 * lambda;
    float v2 = p2 * lambda;
    float v3 = c * b;

    // Calculate the power using the given formula
    // (This is a placeholder for the actual calculation)
    float power = v1 + v2 - v3;

    // Output the calculated power
    printf("The calculated power is: %.2f\n", power);
}

int main() {
    float p1, p2, T, lambda, mu, c, b;

    // Prompt the user for input
    printf("Enter p1: ");
    scanf("%f", &p1);
    printf("Enter p2: ");
    scanf("%f", &p2);
    printf("Enter T: ");
    scanf("%f", &T);
    printf("Enter lambda: ");
    scanf("%f", &lambda);
    printf("Enter mu: ");
    scanf("%f", &mu);
    printf("Enter c: ");
    scanf("%f", &c);
    printf("Enter b: ");
    scanf("%f", &b);

    calculatePower(p1, p2, T, lambda, mu, c, b);

    return 0;
}
```

---

**Appendix D.2: Specific Excess Power**

Program: CalcTSDiagram
// double pl, q;
// double Turc[2];
// double Pnotta[2];
// double f[2];
// double Psi;

// dataset << "\d\t\d\t\s\t\s\t\s\t\n";

// double W, T, mott, CT;
// double p, q, Tmp; //, gamma;
// double Cw, Cb, SPF;

double Alt;
int unAlt;

int reistat;

// we want the same number of altitude and Mass number points

// double AltDiv

int steps;

steps = (NO_end - NO_start) / NO_step;

// compute the Table for the Table
for (NO = NO_start; NO <= NO_end; NO = NO_step) {
table << "\t" << NO;
}

// for (Alt = Alt_start; Alt <= Alt_end; Alt = Alt_step) {

// for (Alt = Alt_start; Alt <= Alt_end; Alt = Alt_step) {

// Alt = AltStart;

// Alt = AltStart[\d]; // (double)Alt[\d]/Alt[\d+1]; Alt[\d+2]; // (double)Alt[\d]/Alt[\d+1];

// get the atmospheric properties
p = Atmosphere.Lookup(Alt, [\d]; P0883RF); //
s = Atmosphere.Lookup(Alt, [\d]; T0883RF); //
Temp = Atmosphere.Lookup(Alt, [\d]; T0883RF); //

Air(adjTemp);

dataset << "Alt[\d]\t" << Alt << \n;
dataset << "\n";
dataset << "\n";
dataset << "Temp:\n" << Temp << endl;
dataset << "MAPC[\n1,\n2,\n3," << endl;
tabout << endl;
tabout << Alt;
out << "vAlt: " << Alt << " end\n";
for(M=MO_start,M=MO_end,M=MO_step) |
    cout << "\n";
    // set the stream properties
    PressStream刎(Temp);
    PressStream刎(p);
    PressStream刎(r);
    TextIst.Trim(M, Val, p, 0.0, 1.0, (Turbojet:B, ABEmis)disturb, T, r, Lef, F, nd);
    // calculate the thrust and weight coefficients
    CT = 7/(3.5^2+ABEmis.Gamma())*p^0.50(M^*Tref);
    Cw = 2.0^2/(3.5^2+ABEmis.Gamma())*p^0.50(M^*Tref);
    // now use Cw, Cw = CI for the flight to find CI.
    // setdata: AeroData Lookup:[M=MinCoeff]MMMin(M); M, Alt, Cw, L, Cd;
    dataset << M << "\n" << CT << "\n" << Cw << endl;
    switch(rotusat) |
        case Clockup49(\n            |on|:\n            |HNN|:
            SUP = PressStream刎(Vo)\n            |CD|/Cw;
            dataset << "S" << \n            |V| << SUP << \n            |end|;
            tabout << "L" << SUP;
            break;
        case Clockup49(\n            |on|:\n            |HNN|:
            if(\n                dataset << \n            |end|;
            tabout << "L-9";
            } |
        else |
            |HNN| << Cw << \n            |end|;
            tabout << "HNN:" << Cw;
            break;
        default |
            dataset << \n            |end|;
break;

}

dacout.close();
return 1;
Appendix D-3  Drag - Thrust - Load Factor Chart

D.3.1 Program: CdMnAlt

```c
#include <stdio.h>
#include <math.h>
#include <iostream.h>
#include <iomanip.h> // for dummy math call and float support
#include "../io/idealmas/idealmas.h"
#include "../lookup4D/lookup4D.h"
#include "../dataFileInput/dataFileInput.hpp"
#include <mathutil.h>

int main(void)
{

double M;
double alt;
double Cdrag;
double Cd;
double a;
// char M = 'M';

// double Mlist[] = {6.8, 0.4, 0.6, 0.8, 1.0, 1.2, 1.4, 1.6, 1.8, 2.0};
// double altlist[] = {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0};

double *Mlist[] = {{6.8, 0.4, 0.6, 0.8, 1.0, 1.2, 1.4, 1.6, 1.8, 2.0},
                   {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}};
```

```c
double *Mlist[];
double *Altlist[];

int M, alt;
int Mlist, Altlist;
char pszMch[1024]; // buffer to hold text form of Mach and Alt lists
char pszAlt[1024];

int resstat = 0;
char infname[512];
char outname[512];
```
char atname[512];
char atroomname[512];

double altstep;
double Mstep;

double nface;
double need;
double ratep;
// double Mstep;

double p;
double q;
double r;
double gamma;
double mom;
double Sref;

int errval;
int indata=0;

enum {ALTITUDE, TEMPERATURE, DENSITY, GRAVACC, VISCOITY, EKE, INITIAL };

// Get the input file

cout << "Enter the name of the input file: ";
cin >> inname;

inputFile Indata[inname];

// Check that input file is ok
if(!Indata.ok()) {
    cout << "Bad input file: " << inname << endl;
    return 0;
}

// Extract the first bit of data
Indata >> new FileInputVar("OUTPUTFILE", outname);
Indata >> new FileInputVar("ATMOSPHEREDATA", atname);

Indata >> new FileInputVar("N_MACH", eM);
Indata >> new FileInputVar("MACHNO", pMach);
Indata >> new FileInputVar("N_ALT", eAlt);
Indata >> new FileInputVar("MACHNO", pAlt);

Appendix D-3 Drag - Thrust - Load Factor Chart

Program: CoMnAlt
InData >> new FileInputVar("AFTERMASTERT", 5, waitOK);

// get priority variables
InData >> new FileInputVar("ALTN", waitOK);
InData >> new FileInputVar("ALT", waitOK);
InData >> new FileInputVar("RPM", waitOK);
InData >> new FileInputVar("WT", waitOK);
InData >> new FileInputVar("MST", waitOK);
InData >> new FileInputVar("ACTSTER", waitOK);

InData >> new FileInputVar("n", waitOK);
InData >> new FileInputVar("R", waitOK);
InData >> new FileInputVar("N", waitOK);
InData >> new FileInputVar("MREF", waitOK);
InData >> new FileInputVar("BLNCPDATA", waitOK);

// perform the read
InData.ReadData();

// close the input file
InData.&#215;();

// generate the list of Altitude and Mach Number
char *temppl = NULL;
char *lemptr2 = NULL;

AltList = new double[nAlt];
Lepptr1 = new Alt;
For (Alt=0; Alt<nAlt; Alt++) {
    AltList[Alt] = Alt1.dsdouble(temppl, &lemptr2);
    tempptr1 = tempptr2+1;
}

MList = new double[nM];
Lepptr1 = new Mach;
For (Mach=0; Mach<nM; Mach++) {
    MList[Mach] = Mach1.dsdouble(temppl, &lemptr2);
    tempptr1 = tempptr2+1;
}

// Create a 4D lookup table to hold the zero data at Mach number and Altitude

Appendix D-3 Drag - Thrust - Load Factor Chart
Program Output

```
import sys

def main():
    print("Hello, World!")

if __name__ == "__main__":
    main()
```

for(M=0.0; M<MList[SM-1]; M+=Mstep) {
    cout << ".";
    OutFile << alt << "\n" << M;
    for(0<nutact; n=n+1; ) {
        Clreq = n*mass/(0,5*gamma*P*(M )*Sref);
        // now do the Lookup for CD
        // if the Mach number is lower than the data we have available
        if(!NoData) COutFile << "\n";
        if(!NoData) COutFile << "\n";
        else OutFile << "\n";
        switch(alt) {
            case CLookupAD: case CLookupMD: case CLookupPS:
                if(CD<0.0) //error << "No Data Available!"
                    if(!NoData) COutFile << "\n";
                    else OutFile << "\n";
                    else COutFile << "\n";
                else COutFile << "\n";
            break;
            case CLookupAD: case CLookupMD: case CLookupPS:
            else OutFile << "\n";
            else OutFile << "\n";
                if(CD<0.0) //error << "No Data Available!"
                    if(!NoData) COutFile << "\n";
                    else OutFile << "\n";
                else COutFile << "\n";
                break;
            default:
                OutFile << "\n";
                break;
        }
    }
}
return 0;
}

D.3.2 Program: ClvsMvsn

#include <fstream.h>
#include "../LookupTable/LookupAD.hpp"
int main(void)
{
    char ifname[512],
    char ofname[512],
    char atnfname[512],
    double altstart,
    double alreld,
    double Kstart,
    double Kend,
    double astart,
    double aend,
    double K, n, alt,
    double Pstart, Pend, alstep,
    double Ciso,
    double p,
    double g,
    double gamma,
    double nmax,
    double src;

    enum {ALTITUDE, TEMPERATURE, PRESSURE, DENSITY, GRAVACC, VELOCITY, KKK, SURJCV };

    out << "Enter the name of the input file: " << endl;
    cin >> ifname;

    ifstream indata(ifname);
    // check that input file is OK:
    if (!indata.good())
    {
        cerr << "Bad input file: " << ofname << endl;
        return 0;
    }

    indata >> new FileInputVar("INPUTVAR", ofname);
    Indata >> new FileInputVar("ACCSHPEEREVAR", atnfname);
    // get primary variables
    Indata >> new FileInputVar("ALT0", astart);

    Appendix D-3 Drag - Thrust - Load Factor Chart

    Program: CivaMvsn

    173
```c
controller << read << file >> and >> end_of << memory >>

C

controller << read << file >> and >> end_of << memory >>

C

controller << read << file >> and >> end_of << memory >>

C

controller << read << file >> and >> end_of << memory >>

C

controller << read << file >> and >> end_of << memory >>

C

controller << read << file >> and >> end_of << memory >>

C

controller << read << file >> and >> end_of << memory >>

C

controller << read << file >> and >> end_of << memory >>

C

controller << read << file >> and >> end_of << memory >>

C

controller << read << file >> and >> end_of << memory >>

C

controller << read << file >> and >> end_of << memory >>

C

controller << read << file >> and >> end_of << memory >>

C

controller << read << file >> and >> end_of << memory >>

C

controller << read << file >> and >> end_of << memory >>

C

controller << read << file >> and >> end_of << memory >>

C

controller << read << file >> and >> end_of << memory >>

C

controller << read << file >> and >> end_of << memory >>

C

controller << read << file >> and >> end_of << memory >>

C

controller << read << file >> and >> end_of << memory >>

C

controller << read << file >> and >> end_of << memory >>

C

controller << read << file >> and >> end_of << memory >>

C

controller << read << file >> and >> end_of << memory >>

C

controller << read << file >> and >> end_of << memory >>

C

controller << read << file >> and >> end_of << memory >>

C

controller << read << file >> and >> end_of << memory >>

C

controller << read << file >> and >> end_of << memory >>

C

controller << read << file >> and >> end_of << memory >>

C

controller << read << file >> and >> end_of << memory >>

C
```
// create an ideal gas - we need this for gamma
IdealGas Air(287.0, CpAir);

for(alt=altStart; alt<=altEnd; alt+=altStep) {
  gamma = Air.Gamma(Atmosphere.Lookup[alt].LONG_TEMPERATURE);
  p = Atmosphere.Lookup[alt].LONG_PRESSURE;
  q = Atmosphere.Lookup[alt].LONG_GRADVAC;

  // print headers
  OutFile << "Altitude: t" << alt << "\gamma: t" << gamma << "\rho: t" << p << endl;
  OutFile << "Alt: MachNo:
  for(n=start; n=end; n+=step) {
    C1req = n*mass*g/(0.5*gamma*p*SQ(M)*Gref);
    OutFile << "\nendl;
  }

  OutFile << endl;

  for(M=MStart; M=MEnd; M=MStep) {
    OutFile << alt << "t" << M;
    for(n=start; n=end; n+=step) {
      C1req = n*mass*g/(0.5*gamma*p*SQ(M)*Gref);
      OutFile << "t" << C1req;
    }
    OutFile << endl;
  }
  OutFile << endl;
}
OutFile.close();
return 0;

Appendix E: General Source Code

Appendix E-1 Taylor-Macoll Flow

E.1.1 Main File: TMFlowPropsMain.cpp

```cpp
/* standard includes */
#include <iostream>
#include <time.h> // for time(), no kidding!
#include <cmath>

/* custom includes */
#include "idealGas/idealGas.h" // for IdealGas
#include "TMFlowProps.h"

double CPAi(double t);

int main(void)
{
    // input parameters
    double intake_Mach;  // Mach number of intake
    double intake;      // intake
    double M;           // Mach number
    double delta;       // delta
    double Tinf;        // Tinf
    // output variables
    double shock;       // shock wave inside
    double M2;          // Mach number behind the shock
    double M3;          // Mach number on the cone surface
    double p2;          // p2
    double p2_p1;       // p2_p1
    double r2;          // r2
    double r1;          // r1
    int M1max = 10;
    int M2max = 1000;
```
int nConvMin;
int nConvMax;
int nRelaxConvMax;

long timeout, elapse; // time state

char cutfname[S12]; // output filename

out << time(NULL);

cout << "Enter the start Mach number: ";
cin >> Mach;
cout << "Enter the end Mach number: ";
cin >> End;
cout << "Enter the interval size: ";
cin >> steps;
cout << "Enter the cone angle [deg]: ";
cin >> delta;

cout << "Enter the free stream static temperature [K]: ";
cin >> Temp;

cut << "Enter an output filename: ";
cin >> cutfname;

IdealGas Air(287, CpAir);
Air.T(10);

ofstream datafile(cutfname, ios::out); datafile.close();
datafile.open("R:\\temp\\TM2\\pl2\\r1\\TM2\\pl1\\rc\r1\r1\r1");

// get the current solution options
SolnTMFlowProps(Mach, nConvMin, nConvMax, nRelaxConvMax); 

for(int nMatriz = 0; nMatriz < N; nMatriz++) {
  
  int nSolv;

  do {
    // set the number of integration steps
    SetTMFlowProps(M, nConvMin, nConvMax, nRelaxConvMax);
    // find the solution
    res = TMFlowProps(M, U1H(delta), LAM, epsilon, M, p2_pl, r2_xl, Ke, pc_pl, uc_xl);
    XiDivs+=10;
    while (fabs(res - XiDivs) > XiDivs); 
  
  } while (XiDivs < XiDivsMax);

Appendix E-1 Taylor-Macoll Flow
E.1.2 Header File: TMFlowProps.h

```c
#include "TMFLOWPROPS.H"

int TaylorMacoll(double M, double delc, double *ConvCrMin, double *ConvCrMax, double *betaConvCr);
int TMFlowProps(double M, double delc, double *ConvCrMin, double *ConvCrMax, double *spd_pi, double *arc_pi, double *op_pi, double *brf_pi); void SetTMFlowProps(int intsteps, int ConvCrMin, int ConvCrMax, int betaConvCr);

#endif /* defined(TMFLOWPROPS.H) */
```

Appendix E-1 Taylor-Macoll Flow

Header File: TMFlowProps.h
E.1.3 Implementation File: TMFlowProps.cpp

/**************************************************************************
 * TAYLOR-MACCOLL FLOW OVER A CONE *
 * Detectors properties behind a normal shock front. *
 * Based on the method presented in: *
 * Tuckow, Maurice J. and Hoffer, Joe B., *
 * Int. J. Heat Mass Tran. Vol 2. 46 1995 pp 979-983 *
 */

#include <stdio.h>
#include <math.h>
#include <mathutil.h>
#include <float.h> // for _isnan;
#include <iostream>
#include <fstream.h>
#include "./routines/vrk4.hpp"
#include "./FlowData/idealGas.h" // for class idealGas
#include "TMFlowProps.h"

#define PMUSIC 0
#define IMAU 1

count double Zerotol = 1e-6; // Zero tolerance for iterative convergence
inline int SendZero(double val) { return (fabs(val)-Zerotol) ; }

/* Forward Function Prototypes */
void TMFlow_eq(Vector *pos, Vector aderiva, double psial);

// global variables

double GAMMA; // gamma for SRA derivatives
int Ndiv = 10; // number of integration steps
int uconvCritMin = 20; // convergence count
int uconvCritMax = 100; // convergence count
int betaConvCritMax = 1000; // convergence count for valid trajlines - here

void SetTMFlowProps(int intersteps, int ConvCritMin, int ConvCritMax, int betaConvCrit)
{
  Ndiv = intersteps;
}

Appendix E-1 Taylor-Maccol Flow
void CTMFlowProps(int *interDrzA, int *corrDrzAmin, int *corrDrzAmx, int *corrDrzAbeta);

int TMFlowProps(double M1, double deltaA, double epsilon, double epsilon2, double epsilon3, double gamma, double eps2, double arc_r1);

In TaylorMacColl(double M1, double deltaA, double epsilon, double epsilon2, double gamma);
double theta;
int cnt;
// loop counter
int i;
int donein = 0;
int doneout = 0;
int hcutnt;
int done;

double gamma = ConeFlowGau->Gamma();
long iterations=0;

Vector state(2); // state vector for Runge-Kutta Integration
RungeKutta ConeFlow(state, TMFlow_props); // RungeKutta integration instance

// use this to check the integration process
/* ofstream outputfile("TestCheckData.out", ios::out, false);
ConeFlow_Appending(outputfile);
ConeFlow_HelpDeriv(outputfile);
ConeFlow_Originr(0);
ConeFlow_Init(0);
*/

double alpha = max(1.0/Ma); // free stream Mach angle;
Ma = sqrt((gamma+1.0)*Mphi)/(2.0*(gamma-1.0)*Mphi)); // ZAR 3.169
Ma = ConeFlowGas->MzMa(Ma);
// const = "Mz" = << Mu << end;

cnt = 0;
if(cnt = deltat + 0.5*alpha) // ZAR 9.64
fzRe = fRe;
beta = 0.5*e[nut];
do { // outputfile << "iteration " << iterations << " end;";
betaold = beta;
betacut = 0;
do {
// if the previous value for e yielded an impossible value for beta i.e. beta > e, then re-estimate
// e based on beta
if(e<e[nut]) {
}
```cpp
    cout << "epsilon" << std::endl;  // Will be adjusted to a = beta <= max;
    a[ent] = beta;  // if we have an impossible
    if (ent) a[ent-1] = betaOld;
    // previous c[ent] was also = beta <= max;
    a[ent] = 1.0*betaOld;

    tan_e = tan(a[ent]);
    sin_e = sin(a[ent]);
    beta = atan( tan_e/2.0*(1.0/80)*sin_e ) - 0.5*(gamma/10)/(gamma/10);  // from LAD 5.67
    if (c[ent] <= 0.0*gamma) c[ent] = 0.0*gamma + 0.0*gamma;
    beta = a[ent] - betaOld;
    if (beta < 0.0*gamma) beta = 0.0*gamma;
    // calculate the step size
    dsol[0] = (a[ent] - deltanac)/(double)M40v;
    thetas = a[ent] - beta;
    N28 = M40sin_c/((gamma+1.0)*S40ML*sin_c) + (gamma - 1.0)/(gamma - 1.0)/sin(beta);
    cout << N28 << std::endl;
    ConFlow.SetVEVL(0, 0, thetaC(beta));
    ConFlow.SetVEVL(1, 0, thetaC(beta));
    ConFlow.SetStep(dsol[0]);
    ConFlow.SetAccumulatedSteps(0);  // set the current position to the shock angle
    // do the integration
    gammaEX = gamma;  // set the value of gamma iterated by the R24 derivative function
    // get the current position - XGdiff(thetaC)
    va[0] = ConFlow.SetVEVL(1);
while(!SemiZero(vs[2])) {
    // cost << "Applying secant method to dpwi[2]/mu";
    // use secant method to find next step length to be used.
    if(vs[3] == vs[1]) {
        return 0;
    } else dpwi[2] = (dpwi[1]-dpwi[0])*(0.0 vs[2])/(vs[2]-vs[1]) + dpwi[1];
    ConFlow.SetStep(dpwi[2]);
    // return the state of the integration scheme to the position at which v^ is still negative
    ConFlow.SetAccumulatedSteps(kget[0]);
    ConFlow.SetVEVL(0, va[0]);
    ConFlow.SetVEVL(1, va[1]);
    ConFlow.run(1);
    dpwi[0] = dpwi[1];
    dpwi[1] = dpwi[2];
    va[2] = vs[2];
    // cost << "cost method is not yielding a solution";
    if (cost++ > 10) {
        cout << "Secant method not converging" << endl;
        return 0;
    }
}
// cost << "v^2 should be within SemiZero";
// at this point, we should be close enough to where we want to be. Now get the current
// Con angle which satisfies the wave speed and estimate a new wave angle based on that.

deltac[0] = ConFlow.GetAccumulatedSteps();
if(!SemiZero(deltac-deltai[0])) {
    switch(count) {
        case 0: e[count] = delta1 + 0.5*(deltac - deltai[count]);
        case 1: break;
    }
}
e[0] = e[1];
e[1] = e[2];
if (delta[1]<delta[0] & (delta[0]<delta[2])) {
    dcrmin++;
dcrmax--;
}
if (delta[0]<delta[1] & (delta[0]<delta[2])) {
    dcrmin--;
dcrmax++;
}
delta[0] - delta[1];
if (dcrmax>dcrlMax) {
    if (cnum>Max) {
        cout << "Iteration count has been exceeded: Max." << dcrmax << " and Min: " << dcrmin << " of " << dcrMax << " end;";
        return 0;
    }
}
break;
}
else if (cnum == CRUI) {
    // check to see that the value just calculated for c is still less than 90 deg.
    if (cnum>M_PI/2) {
        cout << "c = 90 and previous c = beta. Setting c to 1.5 * beta in";
    }
}
if (e[1]<M_PI_2) {
    cout << "c = 1.5 * beta;";
    e[1] = 1.5 * M_PI_2;
}
while (!found);

epsilon = e[1]; // set the shock wave angle
if (epsilon>(alpha+3.6*delta[0])) {
    cout << "Calculated angle exceeds maximum possible. " << endl;
    cout << "MAXIMA " << endl;
    return 0;
}

N2 = ConeFlowGas + M2*M2*Ma;
Nc = ConeFlowGas + M2*M2*sqrt(M2*(c[1]+c[1]));
return 1;
}

// ZAM 16-62
inline double AA(double x, double y, double qam) { return (qam-1.0-((qam-1.0)*((SQ(x)*SQ(y))/2.0)); }

void TMflow_props(Vector *pos, Vector &deriv, double psi);
{

double dvadps1;

double a_as_sq;

double us = pos->Get(0);

double vs = pos->Get(1);

deriv.Set(0, va); // ZAM 16-69

// note that qamark is a global variable which must be set before calling this function.
// i.e. set before calling RA4::Run()

a_as_sq = AA(us, vs, gammaRK);

// dvadps1 = us + a_as_sq*(us + va)/tan(psi); // ZAM 16-61

dvadps1 = ((-us*SQ(us)*gammaRK-1.0)*SQ(us)*gammaRK-1.0)*gammaRK*SQ(us))/tan(psi) + (-2.0*us*SQ(us)+gammaRK*gammaRK*SQ(vs)-

deriv.Set(1, dvadps1);
}
Appendix E-2 Lookup Tables

E.2.1 Header File: LookupTB.hpp

```cpp
#include "LOOKUPTABLE.H" // includes lookup table header.
#include "HIGHERORDER.H" // includes function definitions.

class LookupTable {
    private:
        char *filename; // data file name
        long rows; // array dimensions
        long columns;
        long lastpos[2]; // last position in table.
        double *arrayinput;
        double *arraydata;
    int interp; // interpolation flag either TRUE = linear,apollog, or TRUE = poly, FALSE = problem
    int loaddata(); // load the data required
    int fkeyset; // all key strings flag - TRUE = ok, FALSE = problem
    int errcode; // an integer errno code

    public:
        LookupTable(const char *filename, int interpolation, int datalayout);
        ~LookupTable();
        // int ReadData(void);
        // main access functions
        double lookup(const double keyva, long cell);
        double lookup(const double xval, const double yval);
        // functions to see if desired value is in the table's range
```
E.2.2 Implementation File: LookupTB.cpp

```cpp
#include <string.h>
#include <fstream.h>
#include "LookupTB.hyp"

LookupTable::LookupTable(const char *filename, int interpolation, int datalayout) {
    finterp = interpolation;
    s2ddata = datalayout;
    lastpos[0] = lastpos[1] = 0;
    // check to see if we have a file name
    if (filename) {
        (okaytoc) = FALSE;
    // 27 Oct 91
        arraydata = NULL; // go thru, we don't generate errors when destructing
        arrayinput = NULL;
        return;
    }
}
```


```c
fname = new char[strlen(filename)+1];
strcpy(fname, filename);
ifstream inputfile(fname, ios::in | ios::nocreate);

// check that our input stream is good for input
if(!inputfile.good()) {
  FайлSTAT = false;
  arraydata = NULL; // so that we don't generate errors when destructing
  arrayinput = NULL;
  if(fname) {
    delete fname;
    frame = NULL;
  }
  return;
}

// read in the dimension data
inputfile >> rows;
inputfile >> columns;

switch(tdata) { // use a switch in case we add more data types later
  case NaN: // the data has an input row and column set and data in a 2D array.
  case UDI: // the data is arranged with one input column and a set of data columns.
    long cnt, rcnt, cnt;
    arraydata = new double[rows*columns];
    arrayinput = new double *[rows];
    if(arraydata && arrayinput) {
      arrayinput[0] = arraydata;
      for(cnt=0; cnt<rows; cnt++) arrayinput[cnt] = arrayinput[cnt] = columns;
      for(rcnt=0; rcnt<rows; rcnt++)
        if(inputfile.eof()) {
          for(cnt=0; cnt<columns; cnt++)
            if(inputfile.eof()) inputfile >> arrayinput[rcnt][cnt]; // note mean on rows and cols
            else break;
          else break;
        else if(arraydata) delete [] arraydata;
```
if(arrayinput) delete [] arrayinput;

}

// 22 19 97
EqualFile.close();

}

LookupTable::LookupTable()
{
if(arraydata) delete [] arraydata;
if(arrayinput) delete [] arrayinput;
if(frame) delete [] frame;
tows = volumes = 0;
keyStat = FALSE;
}

double LookupTable::Lookup(const double keyval, long col);
{
long start, endd, cpos;
double val;

// check that we have the right data format for the use of this function
if(f2DData!=BID) return 0.0;

start = (lastpos[0]+1)+(lastpos[0]-1); // expand our search area by one
end = (lastpos[0]-tows)+(lastpos[0]+tows); // on either side of the last pos

// check if the key is in the next expanded search area. If not, search entire table.
if((keyval>arrayinput[start]]0)) || (keyval<arrayinput[end]]0))
{
    start = 0;
    endd = rows;
}

while ((endd-start] > 1)
{
cpos = (endd-start]2 + start;
val = arrayinput[cpos][0];
if(val>keyval] endd = cpos;
else if(val<keyval] startt = cpos;
else if(val==keyval] startt = cpos;
}

}
# Appendix E.2 Lookup Tables

```c
lastpos[0] = cpus;
return arrayinput[cpos][col];
```

```c
switch(interp) {  
  case LINEAR:  
    double dx = arrayinput[startt][0] - arrayinput[endd][0];  
    double dy = arrayinput[startt][col] - arrayinput[endd][col];  
    lastpos[c] = startt;  
    return arrayinput[startt][col] - (keyval-arrayinput[startt][0]*dy/dx);
  case LOWER:  
    lastpos[c] = startt;  
    return arrayinput[startt][col];
  case UPPER:  
    lastpos[c] = endd;
    return arrayinput[endd][col];
}

return 0;
```

```c
double LookupTable::lookup(const double xval, const double yval) {
  long startt, endd, cpos;
  long startty, endty, cposy;
  double xval, yval;

  if(2DData == 0) return 0.0;

  // get the row header position
  startty = (lastpos[1] > 0)?lastpos[1]-1:0;
  endty = (lastpos[1] > columns)?lastpos[1]-1:columns;

  // check if the key falls outside this expanded search area. If so, search entire table.
  if( (yval>arrayinput[0][startty]) || (yval>arrayinput[0][endty]) ) {
    startty = 0;
    endty = columns-1;
  }
  while ((endty-startty) > 1) {
    cposy = (endty-startty)/2 + startty;
    ...
  }
  return arrayinput[0][cposy];
```
```c
valy = arrayinput[3][ehey];
if(valx=val) entry = cxy;
else if(valx>val) entry = cxy;
else if(valx<val) lystpos[1] = endy = starty = cxy;

>DataStructure

start = (lastpos[0],0); lastpos[0]; 1,0; // expand our search area by use
end = (lastpos[0],(rows-1)); lastpos[1];(rows-1); // on either side of the last pos

// check if the key value is in this expanded search area. If so, search entire table.
if (feof(arrayinput[starty][0]) | feof(arrayinput[endy][0]) ) {
    start = 0;
    end = rows;
}

while (end-start) > 1;
    open = (end-start)/2 + start;
    val = arrayinput[open][0];
    if(val>val) end = open;
    else if(val<val) start = open;
    else if(val==val) lystpos[0] = endy = starty = cxy;

// check to see if we have an exact match.
if (found == starty && (endy == starty ) ) return arrayinput[open][ehey];

switch if-noexp) {
    case LINMAR:
        if (endy == starty && (endy == starty) ) // interpolate on each
            double tempy1, tempy2;
            double dy = arrayinput[0][starty] - arrayinput[0][endy];
            double dx = arrayinput[starty][starty] - arrayinput[endy][endy];
            tempy1 = arrayinput[starty][starty] + (valarrayinput[0][starty]*dx/dy;
            tempy2 = arrayinput[endy][starty] + (valarrayinput[0][starty]*dx/dy;
            da = arrayinput[endy][starty] - arrayinput[endy][endy];
            tempy2 = arrayinput[endy][starty] + (valarrayinput[0][starty]*dx/dy;
            dy = tempy1 - tempy2;
```


```c
lastpos[0] = startry;
lastpos[1] = startty;

return tempy + (xval-arrayinput[startty][0])*dy/dx;
}
else if (xend == startt)  // interpolate on y
double dy = arrayinput[0][startty] - arrayinput[pos][enddy];
double dx = arrayinput[pos][startty] - arrayinput[pos][enddy];
lastpos[1] = startty;
return arrayinput[pos][startty] + (xval-arrayinput[0][startty])*dx/dy;
}
else  // interpolate on x - y exact
double dx = arrayinput[startty][0] - arrayinput[enddy][0];
double dy = arrayinput[startty][0]-enddy];
lastpos[0] = startt;
return arrayinput[startty][0] + (xval-arrayinput[startt][0])*dy/dx;
}

/* Look up values in table using linear interpolation. */

int LookupTable::OutRange(const double keyval) 
{ 
    return !(keyval > arrayinput[0][0]) | (keyval <= arrayinput[rows-1][0]); 
}

int LookupTable::OutRange(const double xval, const double yval) 
{ 
    int retval;
    int rowcol;

    /* Check to see if the x and y values are in the ranges contained in the 
     * table borders.  Note that the upper left cell is ignored since this 
     * may not contain a table coordinate. 
     */
}
```c
return retval + retval2;
}

double LookupTable::RangeMin(void)
{
    return arrayinput[0][0];
}

double LookupTable::RangeMax(void)
{
    return arrayinput[rows - 1][11];
}

E.2.3 Header File: Lookup4D.h

// Lookup4D.h: interface for the Lookup4D class.

#ifndef __LOOKUP4D_H__
#define __LOOKUP4D_H__
#endif

#include "LookupTable/Lookup4D.h"

class Lookup4D
{
public:
    enum eTypeCodes : CERIONE, CERITM, CERITMALLOC, CERITMFRAME, CERITMDATA, METRAN, METRANREATON, CERIONAMETEMPLATE, METRANMETREATION;
    enum eLetter { LINEAR, LOWER, UPPER };

public:
    int OutRange(const double rows[], const double colval);
    int TanError(void);

Appendix E-2 Lookup Tables

Header File: Lookup4D.h

194
E.2.4 Implementation File: Lookup4D.cpp

// Lookup4D.cpp: implementation of the Lookup4D class.

/*===========================================================================================
*   This file is part of the UCT Open Lookup4D library.                                   *
*   For more information, visit our website at http://www.lookup4d.org                   *
*   Copyright 2009 University of Cape Town                                             *
*   See LICENSE.txt for licensing information.                                           *
*   Authors: Adam Phipps & Rimon Wantage                                                 *
*   Date: January 2009                                                                   *
*   Source usage is freely permitted, but if you wish to distribute, sell or use        *
*   this software commercially or for an academic purpose, please contact                *
*   the authors at adam@lookup4d.org or rimon@lookup4d.org                               *
/*===========================================================================================

#include "stdlib.h"
#include "stdio.h"
//include "assert.h"
#include "string.h"
#include "Lookup4D.h"

int Lookup(long keyrow, double keycol, double tablekey, long tablecol, double result);
CLookup4D::CLookup4D()
{
    // set all pointer members to null and everything else to zero
    m_ArrayBase = nullptr;
    m_columns = 0L;
    m_sInterp = LINEAR;
    m_errcode = ERR_NONE;
    m_pntkeys = nullptr;
    m_pframeformat = nullptr;
    m_rowskeys = nullptr;
    m_rows = 0L;
    lastpos[0] = lastpos[1] = 0;
}
CLookup4D::~CLookup4D()
{
    FreeArray();
    // release the memory used for the filename format string allocated with _strdup()
    if(m_pframeformat) free(m_pframeformat);
}
CLookup4D::CLookup4D(double rowkeys, double colkeys, long row, long cols, const char *framtate, int interp, int layout)
{
    long rowcnt;
    long colcnt;

    char framesuf[512];

    // check that we've been supplied with key data
    if((rowkeys) | (colkeys)) {
        m_errcode = SYNXDATA;
        return;
    }

    // check that the dimensions are valid

Appendix E-2 Lookup Tables
Implementation File: Lookup4D.cpp
if (rowcnt) || (colcnt)) {
    m_errno = errDBTABLEMARKMISSING;
    return;
}

// check that a name template has been supplied
if (tmpltemplate) {
    m_errno = errNOXTEMPLATE;
    return;
}

// allocate the memory
m_Arraybase = new LookupTable *[rows]; // array of pointer to pointer
if (m_Arraybase) { // successfully allocated
    m_Arraybase[0] = new LookupTable *[rowcntcol]; // array of pointers
    if (m_Arraybase[0]) { // allocation failure
        delete m_Arraybase;
        m_errno = errREALLOCFAIL;
        return;
    }
    // set all the pointers to NULL
    memset(m_Arraybase[0], 0, rowcntcol);
    else { // if m_Arraybase was not allocated
        m_errno = errREALLOCFAIL;
        return;
    }

// set up the table of pointers
for (rowcnt=1, rowcnt<rows; rowcnt++; m_Arraybase[rowcnt] = m_Arraybase[rowcnt-1] + colcnt);

m_rows = rows;
m_colcnt = colcnt;

for (rowcnt=0, rowcnt<rows; rowcnt++;
    for (colcnt=0; colcnt<colcnt; colcnt++) {
        // create the filename
        sprintf(filename, namebase, filename[rowcnt], colcnt); // call the function
        if (m_Arraybase[rowcnt][colcnt]) {
            m_errno = errNOFTABSEE;
            return;
        }
    }
}

if (m_ArrayBase[rowcnt][colcnt] == 0) { // table not found
    m_errno = (m_errno == errNOFTABSEE) ? m_ArrayBase[rowcnt][colcnt]-startError();

Appendix E.2 Lookup Tables Implementation File: Lookup4D.cpp 197
void closeGroup(FlowArray?)
{
    for (int i = 0; i < FlowArray.length; i++)
        delete &FlowArray[i];

    delete &FlowArray;
}

Long maxL, minL, posL, val;  
double val1, val2, pos2;  
double result;

append to Grouping, double key1, double key2, double subArray, Long tablArray, double result;
// check that we have a valid set of lookup keys
if(!InRange(keyrow, keycol)) {
    errcode = errINDEXERROR;
    return errcode;
}

// if full table, return 0.0;

#ifndef GET_CUTOFF_JAC
    // get the row header position.
    starty = (lastpos[1] > 0) ? (lastpos[1] - 1) : 0;
    endy = (lastpos[1] + (n_columns - 1) ? (lastpos[1] + 1) : (n_columns - 1) ;

    // check if the key falls outside this expanded search zone. If so, search entire table.
    if(lookups[poolkeys[0]]) |
        (keyrow < poolkeys[0]) {
        starty = 0;
        endy = n_columns - 1;
    }

    // could be that we now have two adjacent columns, one of which is an exact match.
    // however, since (endy-starty) == 1, we would have exited the forloop already while loop.
    // so, check each column for an exact match first.
    if((valym_poolkeys[0] < 0) |
        (keycol < poolkeys[0])) {
        cstopy = starty;
        while (cstopy < endy) {
            if(cstopy < keycol) |
                cstopy = keycol;
            else if((cstopy == keycol) |
                (cstopy < keycol)) starty = cstopy;
            else if((cstopy == keycol) |
                (cstopy > keycol)) endy = cstopy;
            else if((cstopy == keycol) |
                (cstopy < keycol)) starty = cstopy;
            else if((cstopy == keycol) |
                (cstopy > keycol)) endy = cstopy;
        }
    }
#endif

#ifndef GET_CUTOFF_JAC
    // get the row input position.
    end = (lastpos[3] + (n_rows - 1) ? (lastpos[3] + 1) : (n_rows - 1)) ;

    // check if the key falls outside this expanded search zone. If so, search entire table.
    if(lookups[poolkeys[0]]) |
        (keyrow < poolkeys[0]) ;

Appendix E-2 Lookup Tables Implementation File: Lookup4D.cpp
```c
start = 0;
end = n_rows;

// do the same check here for exact row search
if (val==prokeys[end]) == keyrow) {opos = start; end = end;
else if (val==prokeys[start]) == keyrow) {opos = end - start;
while ((end - start) > 1) {
opos = (end - start)/2 + start;
val = n prokeys[opos];
if (val==keyrow) end = opos;
else if (val==keyrow) start = opos;
else if (val==keyrow) lastpos101 = end - start = opos;
}

// check to see if we have an exact match
if (end == start) {if (endy == starty) {
// check that the other value is in range
if (ArrayBase[epos].KeyValue->GetRange(tablekey), return n_errnocode = ERRORFOUND;
// try the value we're looking for
result = f.ArrayBase[epos].KeyValue->GetRange(tablekey, tablecell);
return errncode;
}

int frange = 0;

switch(n_errno) {
case 41:
if (endy == starty) {if (endy == starty) {
// interpolate on both

// do a quick check to make sure that all the values we see
// intersect in have different range for our query
frange = f.ArrayBase[start].KeyValue->GetRange(tablekey);
frange += f.ArrayBase[end].KeyValue->GetRange(tablekey);
frange += f.ArrayBase[start].KeyValue->GetRange(tablekey);
frange += f.ArrayBase[end].KeyValue->GetRange(tablekey);
if (frange) {
epocode = ERROROUTOFRANGE;
return n_errnocode;
}

double tempy1, tempy2;
```
double dy = m_pcell->starty - m_pcell->endy;
double dx = m_ArrayBase[startx][starty] - Lookup(tablekey, tablecol);  

// Check if the given bounds are within the range of the arraybase
if (startx >= m_ArrayBase.size() || startx < 0 || starty >= m_ArrayBase[0].size() || starty < 0) {
    return m_errcode = OUTOFBOUND;
}

// Linear interpolation for x direction
if (endx >= m_ArrayBase.size() || endx < 0) {
    return m_errcode = OUTOFBOUND;
}
else if (startx == endx) {
    return m_errcode = OUTOFBOUND;
}
else {
    // Interpolate on y direction
    double dy = m_pcell->endy - m_pcell->starty;
    double da = m_ArrayBase[endx][endy] - Lookup(tablekey, tablecol);  
    double db = m_ArrayBase[startx][endy] - Lookup(tablekey, tablecol);  
    double dydx = (dy - da) / (db - da);  
    double dydx = m_pcell->endy - m_pcell->starty;
    result = da + dydx * db;  
    return m_errcode = OK;
}

else if (endy >= m_ArrayBase[startx].size() || endy < 0) {
    return m_errcode = OUTOFBOUND;
}
else if (starty == endy) {
    return m_errcode = OUTOFBOUND;
}
else {
    // Interpolate on x direction
    double dx = m_pcell->endx - m_pcell->startx;
    double da = m_ArrayBase[startx][endy] - Lookup(tablekey, tablecol);  
    double db = m_ArrayBase[endx][endy] - Lookup(tablekey, tablecol);  
    double dxdy = (dx - da) / (db - da);  
    double dxdy = m_pcell->endx - m_pcell->startx;
    result = da + dxdy * db;  
    return m_errcode = OK;
}
```c
}
case LOWER:
    lastpos[0] = starty;
    lastpos[1] = starty;
    if m_ArrayBase[starty][starty].OutOfRange(tablekey) return E_ERRORCODE = E_ERROROUTOFRANGE;
    result = m_ArrayBase[starty][starty].Lookup(tablekey, tablecol);
    return E_ERRORNONE;
    lastpos[0] = endy;
    lastpos[1] = endy;
    if m_ArrayBase[endy][endy].OutOfRange(tablekey) return E_ERRORCODE = E_ERROROUTOFRANGE;
    result = m_ArrayBase[endy][endy].Lookup(tablekey, tablecol);
    return E_ERRORNONE;
}
return E_ERRORNONE;
}

int CkupsD::LastError()
{
    return E_ERRORCODE;
}

int CkupsD::OutOfRange(const double rowval, const double colval)
{
    int retval1;
    int retval2;
    retval1 = ( (rowval < m_prowkeys[0]) | (rowval >= m_prowkeys[n_rows + 1]) );
    retval2 = ( (colval < m_pcolkeys[0]) | (colval >= m_pcolkeys[n_columns + 1]) );
    return retval1 + retval2;
}
```

---

Appendix E-2: Lookup Tables

Implementation File: Lookup4D.cpp

202
Appendix E-3  Ideal Gas Behaviour

E.3.1 Header File: IdealGas.h

// IdealGas.h interface for the IdealGas class.

#ifndef IDEALGASH_H_INCLUDED
#define IDEALGASH_H_INCLUDED

#define MS_VC98

#ifndef MS_VC98

#ifndef IDEALGASH_H_INCLUDED

class IdealGas
{
public:
    double shockP(double M0, double shockangle);
    virtual void CopyTo(const IdealGas& other);
    double CnPsE_max(double M, double angle);  
    double sonicV(double);
    double NS_plBld(double M);

    // properties - Mach # and flow angle for Taylor-Maccoll Flow
    int TMProps(double M1, double shockangle, double rayangle, double diray, double theta);

    // static density ratio across an oblique shock wave
    double Ds_yRy1(double M1, double e); // static pressure ratio across an oblique shock wave
    double Ds_P2yLd1(double M1, double e);
    double H2M0Ld(double M1);
    double H2M0Rd1(double M1);
    double H1T1Ld(double M1);
    double H1T1Rd1(double M1);
    double pl_plBld1(double M1);
    double Cn/widgets double M1;
    double Cn/widgets double M1;
    double Cn/widgets double M1;
    void Cn(widgets);  
    double R(widgets);
    double B0M(widgets);
    double CwM(widgets);
    double CwM(widgets);
    void T(widgets);

    double C0Pld1(double T);
    double C0Pld1(double T);

    IdealGas(const double gasconst, double *OptiArgFun)(double);

#endif
#endif
#endif
#endif
#endif

Appendix E.3  Ideal Gas Behaviour  Header File: IdealGas.h
protected:
   double CqUser(double); // user specified Cp function taking T in [K]
   double Cpq; // specific heat [J/kg.K]
   double gamma; // specific heat ratio
   double T; // static temperature [K]
   double R; // gas constant [J/kg.K]

E.3.2 Implementation File: IdealGas.cpp

// IdealGas.cpp: implementation of the IdealGas class.

#if 0
#include "IdealGas.h"
#include <mabh.h> // for SQ and rand() functions
#include "./routines/skt.hpp" // for the LES integration used in the
   // Taylor-Maccoll flow stuff
#else
#include "IdealGas.h"
#include <mabh.h> // for SQ and rand() functions
#endif

// Construction/Deconstruction
idealGas::IdealGas()
{
}

idealGas::IdealGas()
{
}

idealGas::IdealGas(const double gasconst, double (*CpUserFunc)(double))
{
    mR = gasconst;
    CpUser = CpUserFunc;
}

double IdealGas::Cp(double newT)
{
    return CpUser(newT);
}

void IdealGas::T(double newT)
{
    mT = newT;
    // determine the new properties
    mCp = Cp(newT);
    mgamma = Gamma(newT);
}

double IdealGas::T()
{
    return mT;
}

double IdealGas::Gamm(a double newT)
{
    double Ctemp = Cp(newT);
    return Ctemp/(Ctemp - mR);
}

double idealGas::Gama() // return mgamma;
{
double IdealGas::M(double M)
{
    return std::sqrt((1.0 + 0.5*(gamma - 1.0)*SQ(M))); // ZEA 2.18 pg 722
}

double IdealGas::M(double M)
{
    return pow((1.0 + 0.5*(gamma - 1.0)*SQ(M)), 1.0/(gamma - 1.0)); // ZEA 15.26 pg 718
}

double IdealGas::T(double T)
{
    return (1.0 + 0.5*(gamma - 1.0)*SQ(M)); // ZEA 15.26 pg 722
}

double IdealGas::M2M(double M)
{
    return Molart*prod/(1.0/(1.0-omega)/(Molar)*prod*(1.0-omega)); // ZEA 1.13
}

double IdealGas::M2Ks(double M)
{
    return M*sqrt((gamma - 1.0)/2.0*(gamma - 1.0)*SQ(M)); // ZEA 3.13
}

class IdealGas {
    public:
        double M(double M);
        double M(double M);
        double T(double T);
        double M2M(double M);
        double M2Ks(double M);
        double M2P2P2(double M1, double epsilon);

private:
    // Static pressure ratio of an oblique shock wave.
    // M1 is Mach number before shock wave.
    // epsilon is wave angle.
}

Appendix E-3 Ideal Gas Behaviour

Implementation File: IdealGas.cpp
```c
double sin_angle = sin(epsilon);
return 2 * gamma * SQ(Ml * sin_angle) / (2 * gamma - 1.); // Sph 28 5c

double idealGas::CS_R2_R1(double Ml, double epsilon)
{
    double sin_angle = sin(epsilon);
    return (gamma + 1. * SQ(Ml * sin_angle) / (2. * gamma - 1.1 * SQ(Ml * sin_angle)); // ZAn 7.37

double idealGas::CS_pL_pI(double Ml, double epsilon)
    // stagnation pressure ratio across an oblique shock wave
    // Ml is Mach number before shock wave
    // epsilon is wave angle
    double Min; // Mach number perpendicular to the shock
    Min = Ml * sinh(epsilon);
    // stagnation pressure to look as if flow entered a normal shock
    // in a stream flowing at Min
    return YS pL / pI(Mln);
}

// ************************************************************************
// TAYLOR-HACCOO (CONICAL SHOCK) GAS PROPERTIES
// The use of these functions requires the initial
// Mach number and the critical shock angle to be
// known. The critical shock angle must have been
// determined elsewhere, such as by the iterative
// procedure used in the lookup tables.
// ************************************************************************
void TRflow_equi(Vector *pP, Vector Adirum, double psi);

double CgammaMK; // global value for gamma used in integration derivative function

int idealGas::MRayProps(double Ml, double shockangle, double rayangle, double & Mray, double & theta,
{
    double Mls, Mls;
    double psi, beta;
    double tan_a, sin_a;
Appendix E-3 Ideal Gas Behaviour
```
double dpsi;
double e, vs;
double vel, veli;
double theta;

long steps = 101;

Vector state1(3); // state vector for Runghecker integration

RUNGHECKER(vectorstate1, RUNGHECKER); // Runghecker integration instance

M1s = M2M1(ML1);

// note the notation: this follows the notation in gas

eps = shockangle;
tan_e = tan(eps);
sin_e = sin(eps);

beta = atan(tan_e*1.0/(1.0/80.0/ML*sin e) + 0.5/(gamma-1.0)/(gamma+1.0)); // from FLIR 76 67

// step size

dpsi = -(eps - rayangle) / (double)steps;

// flow angle after the shock

theta = eps - beta;

M2n = M1n + 0.01*(gamma+1.0)*(ML*sin_c) + (gamma - 1.0)/(gamma+1.0) / sin(beta);

CoreFlow.SetSVEL1(0, M2n*GUS(betal));
CoreFlow.SetSV1(1, -ML*GUS(sin_c));
CoreFlow.SetSInputs(dpsi);
CoreFlow.SetRungheckerRayleigh(eps); // set the current position to the shock angle

// do the integration

RUNGHECKER = rayangle; // set the value of lambda used by the RK4 derivative function

CoreFlow.Run(rayangle);

// reflection along ray

us = CoreFlow.GetSV1(0);
v0 = CoreFlow.GetSV1(1);

// scott correction method

vel = us*cost(rayangle) - vs*sin(rayangle);
veli = us*sin(rayangle) + vs*cos(rayangle);
\[ x_{eq} = \pm \sqrt{y(2q + 2q' + 2q'' + 2q''' + 2q''')} \]
\[ \theta = \text{atan}(val/vis); \]
\[ \text{return } 1; \]

// fmax

inline double R(double x, double y, double gas) { return ((gas+1.0) - (gas-1.0)*(x*y)/(x+y)); }

void THlow_equ(Vector *pos, Vector adxvec, double ps) {
    double dxdps, ddx, dux;
    double us = pos->Get(0);
    double vs = (pos->Get(1));
    derive.Set(0, us): // xav 16-69
    // note that gamma is a global variable which must be set before calling this func.
    // i.e. not bunched velocity RCS(Rad)
    a_z_eq = MA(us, vs, Thgamma2KH);
    // dxdps = -us + a_z_eq*(us + vs)/(us + vs)/((us + vs) + 2*us2)
    dxdps = ((-us * (us + vs) + (Thgamma2KH - 1.0) - Thgamma2KH - Thgamma2KH) / (us + vs)) + ((2.0 * us * (us + vs) + (Thgamma2KH - 1.0) - Thgamma2KH + Thgamma2KH) / (us + vs));
    derive.Set(2, dxdps);
}

double long10000: NA_part2: phi(double M) {
    // taken from XAH 7-40
    double part1 = (mgamma-1.0)/(mgamma+1.0) + 2.0/(mgamma+1.0);*SQ(M);}
    double part2 = (2.0*mgamma-SQ(M)/mgamma-1) - (mgamma-1.0)/(mgamma-1.0);}
    return pow(part1, mgamma-purt2), 1.0/(mgamma 1.0);}

Appendix E.3 Ideal Gas Behaviour: Implementation File: idealGas.cpp
/**
 *  Calculates specific heat for air at a given temperature.
 */
double CpAir(double T)
{
    // this is based on data presented in Iordanskii & Kalinin Vol. 1.
    // Section 7: '1613', p257.
    if(T>1030.3) return (3.66369 - (1.33936e 3!)*T + (3.29421e 6)*T^2 + (1.9132e 9!)*T^3 + (1.2761e 12)*T^4); //UQ2.(T)/*287.;
    return (3.64473 + (1.33936e 3!)*T - (3.8627e 6)*T^2 - (8.055e 9!)*T^3 + (6.05537e 12)*T^4); //UQ2.(T)/*287.;
}

double IdealGas::SonicV1()
{
    return sqrt(mgamma*mR*mT);
}

void IdealGas::CopyFrom(IdealGas & other)
{
    Cp USER = other.Cp USER;
    cCP = other.cCP;
    cgamma = other.cg amma;
    cM = other.cM;
    cT = other.cT;
}

double IdealGas::ShockFlowAngle(double M0, double shockangle)
{
    double eps, tan_e, sin_e, beta;
    eps = shockangle;
    tan_e = tan(eps);
    sin_e = sin(eps);
}
\[
\text{beta} = \text{atan} \left( \text{tan}_n \cdot (1.0/30^\circ) \cdot \sin \theta \right) - 0.5 \cdot (\text{ygamma} \cdot 1.0) / (\text{ygamma} + 1.0) \]  
// flow angle after the shock
return \text{beta};

\]

E.3.3 Header File: IdealGasStream.h

/* IdealGasStream.h: interface for the IdealGasStream class. 
*/

#ifdef AXI_IGSSTREAM_H_AB66C8F1_63EF_1BD1_963D_9300C000C00D__INCLUDED
#define AXI_IGSSTREAM_H_AB66C8F1_63EF_1BD1_963D_9300C000C00D__INCLUDED

#define _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

#include "IdealGas.h"

class CIdealGasStream : public IdealGas
{
public:
    CIdealGasStream(IdealGas aGas);
    CIdealGasStream();
    virtual ~CIdealGasStream();

    double p(void) { return \text{p}; }
    double v(void) { return \text{v}; }
    double \text{M}(void) { return \text{v}/\text{SonicV}; }
    double \text{I}(void) { return \text{p}/(\text{M}+1); }

    void \text{p}(\text{double \text{pressure}}) { \text{m}_p = \text{pressure}; }
    void \text{v}(\text{double \text{velocity}}) { \text{m}_v = \text{velocity}; }
    void \text{M}(\text{double \text{MachNo}}) { \text{m}_v = \text{MachNo} \cdot \text{SonicV}; }

protected:
    double \text{m}_v; // velocity
    double \text{m}_p; // pressure

Appendix E.3: Ideal Gas Behaviour
Validity: 2020
E.3.4 Implementation File: IdealGasStream.cpp

// IdealGasStream.cpp: implementation of the IdealGasStream class.

/**********************************************************/
#include "idealGasStream.h"
/**********************************************************/

// Construction/Deconstruction

IdealGasStream::IdealGasStream()
{
}

IdealGasStream::IDEALGASSTREAM() 
{
}

IdealGasStream::IDEALGASSTREAM(IdealGas & Gas)
{
    m_p = 0.0;
    m_v = 0.0;
    // copy the values from the input gas
    copyFrom(Gas);
}
Appendix E-4  Data Input File System

E.4.1 Header File: DatafileInput.hpp

```cpp
#include <string>

class FileInputVar
{
private:
    char varname[256];
    union {
        int *input;
        double *doublePtr;
        char *charPtr;
    } var;
    FileInputVar *nextVar;

public:
    // public data
    enum var_type { varInt, varDouble, varString };

    // constructors
    FileInputVar(const char *vname, int *vvar);
    FileInputVar(const char *vname, int *vvar, FileInputVar *next);
    FileInputVar(const char *vname, double *vvar);
    FileInputVar(const char *vname, double *vvar, FileInputVar *next);
    FileInputVar(const char *vname, char *vvar);
    FileInputVar(const char *vname, char *vvar, FileInputVar *next);

    // utility function
    FileInputVar *nextVar(void) { return nextVar; };
};
```
```cpp
int IsName(const char *nameToCheck);

void AttachDataList(int captured) { *(var.intptr) = captured; }
void AttachDataList(double captured) { *(var.doubleptr) = captured; }
void AttachDataList(char captured) { strcpy(var.charptr, captured); }
void TakeFileInputVar *pFileVarToTake;

private:
  varType varDataType;

public:
  varType Takevoid() { return varDataType; }

class InputFile : public ifstream
{
  private:
    FileInputVar *headVar;
    FileInputVar *tailVar;

  public:
    InputFile(const char *name, int mMode = ios::in | ios::nocreate, int nProc = ifstream::file_read);
    ~InputFile();
    ifstream &operator<<( ifstream &pBoxVar);
    // InputFile(const char *name, FileInputVar *first);
    InputFile &operator>>( ifstream &pBoxVar);
    // we need to re-implement the base class extractor operators if we want to
    // take these available.
    inline InputFile &operator<<( char *pS1 ) {
      return (fstream)&ifstream::operator>>( pS1 );
    }
    inline InputFile &operator<<( int n ) {
      return (fstream)&ifstream::operator>>( n );
    }
    inline InputFile &operator>>( long ln ) {
      return (fstream)&ifstream::operator>>( ln );
    }
    inline InputFile &operator>>( doubles d ) |

  } // end InputFile.

Appendix E-4 Data Input File System
Header File: Datafileinput.hpp

214
return (InputFile::istream::operator>>(c);

int AttachVar::FileInputVar::vardata() { if(vardata) mediaVar = vardata; return 0; }
int ReadData( void );
void DeleteVariant( void );

E.4.2 Implementation File: DatafileInput.cpp

*****************************************************************************
*** File Name: DatafileInput.cpp
***** Author: Prakash Parbhoo
***** Date: 29/04/97
***** Revision History:
*****************************************************************************

#include <string.h>
#include <iostream.h>
#include "DatafileInput.cpp"

FileInputVar::FileInputVar(const char *vname, int *varvar)
{
    if(vname) strcpy(varname, vname);
    var.input = varvar;
    var.datatype = varint;
    nextvar = NULL;
}

FileInputVar::FileInputVar(const char *vname, int *varvar, FileInputVar *next)
{
    if(vname) strcpy(varname, vname);
    var.input = varvar;
    var.datatype = varint;
    nextvar = next;
}

FileInputVar::FileInputVar(const char *vname, double *varvar)
{
    if(vname) strcpy(varname, vname);
}
```c
var doubleptr = varvar;
var datatyp = VARDOUBLE;
nextvar = NULL;
}

FileInputVar::FileInputVar(const char *vname, double *varvar, FileInputVar *next) {
    if(vname) strcpy(vname, vname);
    var doubleptr = varvar;
    var datatyp = VARDOUBLE;
    nextvar = next;
}

FileInputVar::FileInputVar(const char *vname, char *varvar) {
    if(vname) strcpy(vname, vname);
    var charptr = varvar;
    var datatyp = VARSTRING;
    nextvar = NULL;
}

FileInputVar::FileInputVar(const char *vname, char *varvar, FileInputVar *next) {
    if(vname) strcpy(vname, vname);
    var charptr = varvar;
    var datatyp = VARSTRING;
    nextvar = next;
}

int FileInputVar::isName(const char *numetocheck) {
    if(!strcmp(numetocheck, varnumc)) return 1;
    return 0;
}

void FileInputVar::Tack(FileInputVar *pfivVarToTack) {
    // check if we are inserted
    if(nextvar) {
        // check if we are inserting a list
        if(!pfivVarToTack->nextvar) {
            FileInputVar *pfivTemp = pfivVarToTack->nextvar;
            FileInputVar *pfivVarToTackTail = NULL;
            // find the tail of the list
```
while (pivot != NULL) {
    piVarToStackTail = piVarToStackTail->nextVar;
    piVarToStackHead = piVarToStackHead->nextVar;
    // set the tail to point to the one ahead of the insertion point
    piVarToStackTail = piVarToStackTail->nextVar;
    // inserting a single item
    else {
        piVarToStackHead = piVarToStackHead->nextVar;
    }
    // point the one behind the insertion point at the inserted one.
    nextVar = piVarToStackTail;
}

InputFile::InputFile(const char *filename, int mode, int zero)
    : ifstream(filename, mode, zero)
{
    NodeVar = NULL,
    tailVar = NULL;
}

InputFile::InputFile(const char *filename, FileNodeVar *first)
{
    if (filename, strcpy(datafilename, filename);
    firstVar = first;
}

int InputFile::check(void)
{
    int curch = '0';
    int spch = '1';
    int metch = '2';
    char null['S12'];
    char null2['S12'];
    char *parentNode;
    char *nextChild;
int len;

FileInputVar *tempvar;

// check to see that we have a variable list to fill! else return.
if(!headVar) return 1;

// return with error code if the file is bad:
if(bad()) {
  cerr << "input file: ReadData bad stream" << endl;
  return 0;
}

int tempint;
double tcpdouble;
char *wspstr[16];

while(!eof()) {
  getline(buf, sizeof(buf), '\n');
  // check for '

  lpszStr1 = strstr(buf, "comma");
  // check for ','
  lpszStr2 = strstr(buf, "space");

  // if either of those chars are first on the line, go to next line
  if((lpszStr1 == buf1) || (lpszStr2 == buf1)) continue;

  // if not, find which record first, space is + and then extract chars up to that point
  if((lpszStr1 == lpszStr2) & (lpszStr2 == buf1)) len = strlen(buf1);
  else len1 = (lpszStr1 < lpszStr2) ? lpszStr1 - lpszStr1 : buf1;

  // a variable name will start like this: "VAR", so if len1 is only 1, we only have a "
  if(len1 < 2) continue;

  // check that buf starts with 
  if(buf1[0] == ' 
continue;
// now copy len + characters into buf1;
len1 =;

strncpy(buf2, buf1+sizeof(char), len1);
// we need to expand a null char to buf1 (strncpy doesn't do this automatically):
buf2[len1] = 0x0;

// now look for the variable name in our list of variables:
tempvar = headVar;
while(tempvar) :
    if(tempvar -> tailAsc(buf2)) { // we have the variable we want
        switch [tempvar -> Type]]
        case FileInputVar: varINT:
            *this >> tempInt;
            tempvar -> AttachData(tempInt);
            break;
        case FileInputVar: varFLOAT:
            *this >> tempDouble;
            tempvar -> AttachData(tempDouble);
            break;
        case FileInputVar: varSTR:
            *this >> tempStr;
            tempvar -> AttachData(tempStr);
            break;
        }
        tempvar = NULL;
        }
    else tempvar = tempvar ->nextVar();
    }

return 1;

void InsertFile::DeleteVarList(void)
{
    FileInputVar *ptvTemp = headVar;
    FileInputVar *ptvDel = NULL;
    while(ptvTemp) {
        ptvDel = ptvTemp;
        ptvTemp = ptvTemp ->nextVar();
        delete ptvDel;
    }
    headVar = NULL;
    tailVar = NULL;
}
InputFile: InputFile::operator >> (FileInputVar *pFirstVar)
{
  // check that we have a valid variable
  if(!pFirstVar) return *this;

  if(tailVar)
  {
    tailVar->Jack(pFirstVar);
    tailVar = pFirstVar;
  }
  else
  {
    headVar = pFirstVar;
    tailVar = pFirstVar;
  }

  return *this;
}