Multistage 2-DOF rocket trajectory simulation program for freshmen level engineering students

By

TITLE PAGE

Zachery Doucet

A Thesis

Submitted to the Faculty of

Mississippi State University

in Partial Fulfillment of the Requirements

for the Degree of Master of Science

in Aerospace Engineering

in the Department of Aerospace Engineering

Mississippi State, Mississippi  
August 2019

Copyright by

COPYRIGHT PAGE

Zachery Doucet

2019

Multistage 2-DOF rocket trajectory simulation program for freshmen level engineering students

By

APPROVAL PAGE

Zachery Doucet

Approved:

Keith Koenig

(Major Professor)

Yang Cheng

(Committee Member)

Calvin R. Walker

(Committee Member)

David S. Thompson

(Graduate Coordinator)

Jason M. Keith

Dean

Bagley College of Engineering

Name: Zachery Doucet

ABSTRACT

Date of Degree: August 9, 2019

Institution: Mississippi State University

Major Field: Aerospace Engineering

Major Professors: Keith M. Koenig

Title of Study: Multistage 2-DOF rocket trajectory simulation program for freshmen level engineering students

Pages in Study: 108

Candidate for Degree of Master of Science

With the growing interest in space exploration, the need for professionals to be prepared for the job is ever more present. This project aims to satisfy that goal by presenting a MATLAB program for use by freshmen level engineering students that provides results for rocket trajectories within acceptable margins of accuracy. The initial stages of development were shaped by the decision to keep the program accessible to someone within the target demographic. As a result, a three-degree of freedom system was chosen, with focus put on amateur rocketry including high power rockets. Various methods were used to provide outputs for rocket flight paths that could be compared to field tests. The solution methods available are Euler and fourth-order Runge-Kutta that utilize the equations of motion for a perpendicular-parallel or x-y coordinate frame. Trajectory simulations and field tests were conducted to have reference cases to confirm the accuracy of the program.

DEDICATION

This thesis is dedicated to the memory of my beloved grandfather, Raymond Vallimont. Although one of the greatest inspirations for my academic pursuit, he did not live to see me complete my education. This one is for you Gramps.

ACKNOWLEDGEMENTS

Throughout the writing of this thesis I have received a great deal of support and assistance. I would first like to thank my major professor, Dr. Keith Koenig, whose expertise and patience were invaluable in the formulating of the research topic and the methodology used. In addition, I would like to thank the fellow members of my committee, Dr. Yang Cheng and Mr. Calvin Walker for their patience and support throughout the project. I would also like to thank my professors and instructors here at Mississippi State University, who have provided me with the tools and knowledge to complete this thesis. The Mississippi State University Space Cowboys rocket team also has my thanks for providing me with data that was beneficial to completing this project.

Additionally, there are my friends who offered great support and provided countless happy distractions over the years, such as my colleagues Tyler Howell, Jonathon Lee and Sepehr Zangeneh, and my dear friends William Orrin Cole and William Streaker. Finally, I would like to thank my parents, Jean and Joseph Doucet, and grandmother, Mary Ann Vallimont, who have provided me with invaluable advice, love and support over the years. Whether it be the counsel of my father, the sympathetic ear of grandmother, or the strong push to do and be the best I can be from my mother, I know you are always there for me.

TABLE OF CONTENTS

[DEDICATION ii](#_Toc9960033)

[ACKNOWLEDGEMENTS iii](#_Toc9960034)

[LIST OF TABLES vii](#_Toc9960035)

[LIST OF FIGURES viii](#_Toc9960036)

[NOMENCLATURE xi](#_Toc9960037)

[CHAPTER](#_Toc9960038)

1. [INTRODUCTION 1](#_Toc9960039)
2. [DEVELOPMENT PROCESS 3](#_Toc9960040)

[2.1 Goal of Development 3](#_Toc9960041)

[2.1.1 MATLAB as the program of choice 4](#_Toc9960042)

[2.1.2 OpenRocket as a reference case 4](#_Toc9960043)

[2.2 The Briggs Multistage Supersonic Rocket Flight Simulator 5](#_Toc9960044)

[2.2.1 Development of *Vcr* 8](#_Toc9960045)

[2.2.2 The Exclusion of Lift 10](#_Toc9960046)

[2.3 The Coordinate Frames 10](#_Toc9960047)

[2.3.1 The Parallel-Perpendicular Coordinate Frame 10](#_Toc9960048)

[2.3.1.2 1st Order Euler Equations 11](#_Toc9960049)

[2.3.1.3 4th Order Runge-Kutta Equations 13](#_Toc9960050)

[2.3.2 The X-Y Coordinate Frame 13](#_Toc9960051)

[2.3.2.2 1st Order Euler Equations 15](#_Toc9960052)

[2.3.2.3 4th Order Runge-Kutta Equations 16](#_Toc9960053)

[2.4 Drag Profiles 16](#_Toc9960054)

[2.4.2 No Fins Model 21](#_Toc9960055)

[2.4.3 Small Fins Model 21](#_Toc9960056)

[2.4.4 Large Fins Model 22](#_Toc9960057)

[2.4.5 Delta Fins Model 24](#_Toc9960058)

1. [PROGRAM BREAKDOWN 26](#_Toc9960059)

[3.1 Startup Scripts 28](#_Toc9960060)

[3.1.1 Simulator.m 29](#_Toc9960061)

[3.1.2 Menu scripts 30](#_Toc9960062)

[3.1.2.2 Parameters Menu 31](#_Toc9960063)

[3.1.2.3 Program Menu & Program Editor 32](#_Toc9960064)

[3.2 Initialization Scripts 33](#_Toc9960065)

[3.2.1 Rocket and Motor Parameters 34](#_Toc9960066)

[3.2.2 Initial Launch Conditions 37](#_Toc9960067)

[3.3 Thrust Selection 40](#_Toc9960068)

[3.3.1 Average Thrust Values 41](#_Toc9960069)

[3.3.2 Interpolated Thrust and Motor Data 41](#_Toc9960070)

[3.4 Drag Selection 43](#_Toc9960071)

[3.4.1 Actual Drag 44](#_Toc9960072)

[3.4.2 Constant Drag 44](#_Toc9960073)

[3.4.3 Fin Type Profiles 45](#_Toc9960074)

[3.5 Solution Scripts 47](#_Toc9960075)

[3.5.1 1st Order Euler Solution Scripts 47](#_Toc9960076)

[3.5.2 4th Order Runge-Kutta Solution Scripts 52](#_Toc9960077)

[3.6 Additional Sub-Functions 54](#_Toc9960078)

[3.6.1 Mass.m 54](#_Toc9960079)

[3.6.2 Gravity.m 55](#_Toc9960080)

[3.6.3 Area.m 55](#_Toc9960081)

[3.6.4 SIAtm.m 55](#_Toc9960082)

1. [TESTING AND RESULTS 56](#_Toc9960083)

[4.1 Comparison to Reference Case 56](#_Toc9960084)

[4.2 Field Testing Results 58](#_Toc9960085)

[4.2.1 Small Model Rocket 58](#_Toc9960086)

[4.2.2 Hand-made Rockets 59](#_Toc9960087)

1. [CONCLUSION 62](#_Toc9960088)

[REFERENCES 64](#_Toc9960089)

[APPENDIX](#_Toc9960090)

* + 1. [MATLAB SCRIPTS 66](#_Toc9960091)

[A.1 Startup Page and Menus 67](#_Toc9960092)

[A.1.1 Simulator Primary File 67](#_Toc9960093)

[A.1.2 Main Menu 67](#_Toc9960094)

[A.1.3 Parameters Menu 68](#_Toc9960095)

[A.1.4 Program Menu 69](#_Toc9960096)

[A.1.5 Program Editor Menu 70](#_Toc9960097)

[A.2 Initialization Scripts 71](#_Toc9960098)

[A.2.1 Initial Conditions Compiler 71](#_Toc9960099)

[A.2.2 Rocket and Motor Parameters 72](#_Toc9960100)

[A.2.2.1 Custom Rocket and Motor Data Input 74](#_Toc9960101)

[A.2.3 Initial Launch Conditions 75](#_Toc9960102)

[A.2.3.1 Deflection Scripts 76](#_Toc9960103)

[A.2.3.1.1 Deflection Screener 76](#_Toc9960104)

[A.2.3.1.2 Deflection Calculation 76](#_Toc9960105)

[A.2.4 Thrust Selection 77](#_Toc9960106)

[A.2.4.1 Thrust Data Interpolation 77](#_Toc9960107)

[A.2.4.1 Interpolation Function 79](#_Toc9960108)

[A.2.5 Drag Selection 80](#_Toc9960109)

[A.2.5.1 Drag Profiles 81](#_Toc9960110)

[A.2.5.1.1 No Fins Drag Model 81](#_Toc9960111)

[A.2.5.1.2 Small Fins Model 82](#_Toc9960112)

[A.2.5.1.3 Large Fins Model 83](#_Toc9960113)

[A.2.5.1.4 Delta Fins Model 83](#_Toc9960114)

[A.3 Solution Scripts 84](#_Toc9960115)

[A.3.1 Euler Methods 84](#_Toc9960116)

[A.3.1.1 Euler X-Y Solution 84](#_Toc9960117)

[A.3.1.2 Euler Parallel-Perpendicular 86](#_Toc9960118)

[A.3.2 4th Order Runge-Kutta Methods 89](#_Toc9960119)

[A.3.2.1 4th Order X-Y 89](#_Toc9960120)

[A.3.2.1.1 X-Y ODE Function 90](#_Toc9960121)

[A.3.2.2 4th Order Parallel-Perpendicular 91](#_Toc9960122)

[A.3.2.2.1 ODE Function Parallel-Perpendicular 92](#_Toc9960123)

[A.3.2.3 Additional Files Used for 4th Order Scripts 93](#_Toc9960124)

[A.3.2.3.1 Stop ODE Function 93](#_Toc9960125)

[A.3.2.3.2 Thrust ODE Sub-Function 93](#_Toc9960126)

[A.3.2.3.3 ODE Drag Coefficient – File Version 94](#_Toc9960127)

[A.3.2.3.4 ODE Drag Coefficient – Profile Version 94](#_Toc9960128)

[A.3.3 Sub-Functions Used in Solution Scripts 95](#_Toc9960129)

[A.3.3.1 Profile Drag Coefficient 95](#_Toc9960130)

[A.3.3.2 Average Thrust Profile 95](#_Toc9960131)

[A.3.3.3 Area Calculation 97](#_Toc9960132)

[A.3.3.4 Mass Calculation 98](#_Toc9960133)

[A.3.3.5 Gravity Calculation 99](#_Toc9960134)

[A.3.3.6 Standard Atmosphere 100](#_Toc9960135)

* + 1. [ADDITIONAL FIGURES 102](#_Toc9960136)

LIST OF TABLES

[Table 2.1 No fins drag profile based on Refs. 2, 5, 6, 10, 11 and 13. 21](#_Toc9960137)

[Table 2.2 Small fins drag profile based on Refs. 11 and 13. 22](#_Toc9960138)

[Table 2.3 Large fins drag profile based on Refs. 6, 10 and 11. 24](#_Toc9960139)

[Table 2.4 Delta fins drag profile based on Ref. 10 and 12. 24](#_Toc9960140)

[Table 4.1 Average difference percentile for all scripts 57](#_Toc9960141)

[Table 4.2 Average difference percentile of 4th order parallel-perpendicular case 57](#_Toc9960142)

[Table 4.3 Field testing vs. simulation, units in meters 58](#_Toc9960143)

LIST OF FIGURES

[Figure 2.1 OpenRocket output vs. Briggs MATLAB output 5](#_Toc9960144)

[Figure 2.2 Modified Briggs model vs. OpenRocket 6](#_Toc9960145)

[Figure 2.3 Rocket flight in 6-DOF system vs. point mass system. 7](#_Toc9960146)

[Figure 2.4 MATLAB output with *Vcr* vs. OpenRocket 9](#_Toc9960147)

[Figure 2.5 Forces of parallel-perpendicular coordinate frame 11](#_Toc9960148)

[Figure 2.6 Forces of x-y coordinate frame 14](#_Toc9960149)

[Figure 2.7 Nose and fin designs (all dimensions in centimeters), Ref. 7 17](#_Toc9960150)

[Figure 2.8 Axial force coefficient plot, edited for clarity, Ref. 10 18](#_Toc9960151)

[Figure 2.9 Base axial coefficient plot, Ref. 10 19](#_Toc9960152)

[Figure 2.10 Long and short test models, dimensions in terms of diameter (5.72 cm), Ref. 5 19](#_Toc9960153)

[Figure 2.11 Drag coefficient profiles for several body and fin configurations. 20](#_Toc9960154)

[Figure 2.12 Blue Scout, Ref. 13. 22](#_Toc9960155)

[Figure 2.13 Blue Scout fin design, Ref. 13. 22](#_Toc9960156)

[Figure 2.14 Two and three stage rocket, dimensions in centimeters, Ref. 6 23](#_Toc9960157)

[Figure 2.15 Four stage rocket, dimensions in centimeters, Ref. 11. 23](#_Toc9960158)

[Figure 2.16 Delta planform fin, dimensions in centimeters, Ref. 10 24](#_Toc9960159)

[Figure 2.17 Drag profiles 25](#_Toc9960160)

[Figure 3.1 Nomenclature for program flowchart 26](#_Toc9960161)

[Figure 3.2 A simplistic sequence of events showing the overall process of the program. 27](#_Toc9960162)

[Figure 3.3 Startup sequence that proceeds from the main file to the menu options 28](#_Toc9960163)

[Figure 3.4 Simulator startup message 29](#_Toc9960164)

[Figure 3.5 Startup dialogue box 30](#_Toc9960165)

[Figure 3.6 Main Menu display 30](#_Toc9960166)

[Figure 3.7 Parameters Menu 31](#_Toc9960167)

[Figure 3.8 Program selection menu 32](#_Toc9960168)

[Figure 3.9 Program Editor Menu 33](#_Toc9960169)

[Figure 3.10 Initialization sequence 34](#_Toc9960170)

[Figure 3.11 Preloaded rocket recommendation 35](#_Toc9960171)

[Figure 3.12 Custom input box 35](#_Toc9960172)

[Figure 3.13 Time step and flight angle prompt 37](#_Toc9960173)

[Figure 3.14 Rod deflection box 38](#_Toc9960174)

[Figure 3.15 Deflection display 38](#_Toc9960175)

[Figure 3.16 Thrust Selection flowchart. 40](#_Toc9960176)

[Figure 3.17 Thrust Selection dialogue box 41](#_Toc9960177)

[Figure 3.18 File selection window 42](#_Toc9960178)

[Figure 3.19 Drag Selection flowchart 43](#_Toc9960179)

[Figure 3.20 Drag Selection dialogue box 44](#_Toc9960180)

[Figure 3.21 Profile selection box. 45](#_Toc9960181)

[Figure 3.22 No Fins warning message 45](#_Toc9960182)

[Figure 3.23 Fin Type selection box 46](#_Toc9960183)

[Figure 3.24 Euler method solution scripts flowchart 48](#_Toc9960184)

[Figure 3.25 Drag coefficient prompt 49](#_Toc9960185)

[Figure 3.26 Final Command Window display 51](#_Toc9960186)

[Figure 3.27 Flight path plot 51](#_Toc9960187)

[Figure 3.28 ODE Solution Flowchart 52](#_Toc9960188)

[Figure 4.1 Enterprise rocket, Ref. 14 56](#_Toc9960189)

[Figure 4.2 *Mostly Harmless* rocket, Ref. 15 57](#_Toc9960190)

[Figure 4.3 Freshman class test plot 59](#_Toc9960191)

[Figure 4.4 Personal rocket resting 60](#_Toc9960192)

[Figure 4.5 Combined data plots 61](#_Toc9960193)

[Figure B.1 Example of Chart used to find *CN* Values 103](#_Toc9960194)

NOMENCLATURE

*A* = area

*a* = acceleration

*α* = angle of attack

*c* = speed of sound

*CA* = axial force coefficient

*CA,b* = base axial force coefficient

*CA,f* = forward axial force coefficient

*CD*= drag coefficient

*CN* = normal force coefficient

*D* = drag force

*Df* = diameter of fuselage

*dt* = time differential

*δ* = angle of deflection

*E* = Young’s modulus of elasticity

*F* = thrust

*g* = gravitational acceleration

*g0* = sea-level gravitational acceleration

*γ* = specific hear ratio

*I* = moment of inertia

*kd* = ballistic coefficient

*kt* = ‘thrust coefficient’

*L* = lapse rate

*Lr* = length of launch rod

*Ls* = length of rocket structure

*M* = Mach number

*Mend* = final Mach value in drag profile

*m* = mass during flight

*md* = mass flow rate

*mf* = final stage mass

*mi* = initial stage mass

*mp* = stage propellant mass

*ms* = stage structural mass

*mt* = total mass of rocket

*n* = number of steps

*nos* = total number of stages

*ODE*  = ordinary differential equations

*P* = pressure

*q* = dynamic pressure

*Re* = Reynolds number

*RE* = radius of Earth

*Rg* = gas constant

*Rr* = radius of launch rod

*ρ* = density

*ρ0* = sea-level density

*T* = temperature

*TF* = thrust force

*Tstage* = stage thrust

*t* = time

*tb* = burn time

*tbo* = total burnout time

*tbt*= total burn time for rocket

*ts*= time span

*twait* = wait time

*tx*= time to reach distance

*ty*= time to reach apogee in x-y frame

*tz*= time to reach apogee in par-per frame

*θ* = flight angle

*θd* = flight angle after deflection

*θi* = initial flight angle

*u* = horizontal component of velocity

*v* = vertical component of velocity

*V* = velocity

*Vcr*= critical velocity

*Vx* = horizontal component of velocity

*Vy* = vertical component of velocity

*W* = weight

*x* = horizontal displacement

*Xx* = landing distance

*y* = vertical displacement in x-y frame

*Yx* = apogee in x-y frame

*z* = altitude in par-per frame

*Zx* = apogee in par-per frame

*i,j,k* = index terms



INTRODUCTION

Programing is something that most, if not all, engineers must be acquainted with at some point in their career. With the evolution of technology, the speed and power of computers has improved with it, allowing us the benefit of being able to predict certain outcomes before conducting field testing. A field where this is evident is flight test engineering, specifically rocket flight tests. While in years past field tests or wind tunnel analysis would need to be conducted to determine how a certain design or configuration would perform, the option to simulate rocket performance via software programs has become a key resource in the industry of rocket design. However, programs that allow for this acquire a heavy amount of knowledge of rocket design and flight dynamics. In order to provide a resource for those not as well versed in these fields, a program was developed here for use by entry level users.

Having a software that can simulate rocket flight allows for numerous hours to be saved during the design process. Additionally, the option to change certain variables and parameters to simulate different models awards the user with the option to experiment with different ideas and design philosophies. But how would someone with little knowledge of designing a rocket be able to perform such a task? This paper seeks to provide that answer with an accessible program that allows a user with little knowledge the same ability to predict the flight path of a conceptually designed rocket as those with more experience. “Accessible” meaning a program that is not only easy to use, but which allows the users to learn the actual solution techniques. It is this aspect of the work here which makes it particularly valuable in an educational setting.

Before the code is explored, the development process will be discussed, highlighting some of the key decisions and reasoning behind each of them respectively. This will be followed by a detailed breakdown of the program, including looks at the main scripts and the equations within. Additionally, breakdowns of sub-functions will be included to flesh out the overview of the entire program, and to give a user a complete understanding of how it all works collectively. Finally, the results of the program will be analyzed to determine their validity and will then be compared to reference cases, other simulation software, and field test data to determine if the accuracy of the program falls within the acceptable margin set prior to development.

DEVELOPMENT PROCESS

The beginnings of this project started from an interest in rocket and missile trajectory analysis. Although there are several software packages available for this analysis, one thing became clear: that the bar of entry to use these programs effectively was high. Coupled with this realization was also the need for a program that could work well with the entry level students within the Introduction to Aerospace Engineering course at Mississippi State University. With those two ideas in mind, the basis of the program became clear and will be discussed in the first section of this chapter.

* 1. Goal of Development

Keeping in mind the target user base for the program, the first condition that needed to be met was keeping the program accessible. For the case here, the goal was for anyone within the target demographic, to be able to generate this program after receiving the appropriate instruction. This influenced how the program was designed, being constructed in a way that carries users through the program step by step and allowing them to input their own conditions as to not force them to manually sort through countless files and functions to change one variable. Additionally, the method in which the program executed its solution had to be understandable to a freshmen level student in aerospace engineering. To fulfill this condition, more advanced equations and solution methodology were simplified This made it clear a two-degree-of-freedom (2-DOF) system approach would be needed as opposed to the more correct equations for a six-degree-of-freedom (6-DOF) system. However, this presented another goal for the project: to keep the same level of accuracy provided by a 6-DOF system while using 2-DOF equations.

* + 1. MATLAB as the program of choice

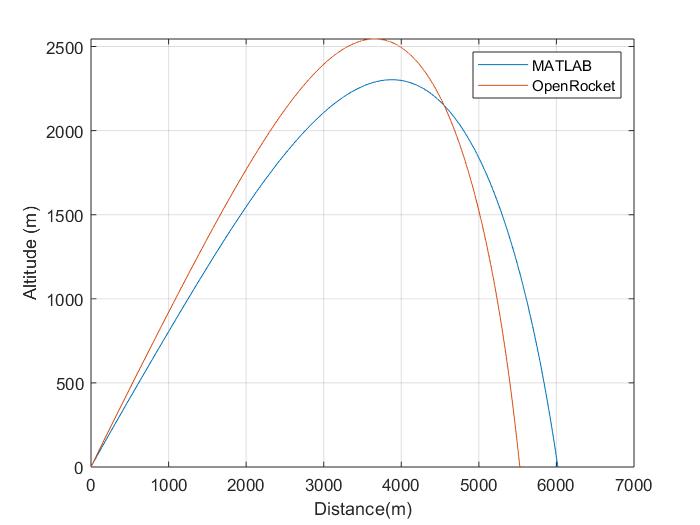
With the overall goal of the project set, a choice had to be made on what software package would be used to construct it in. Many different programs were considered including Python, C++, Maple, MathCAD, and MATLAB. MATLAB was chosen because of the familiarity that the freshmen class had with it as well as its more user-friendly layout and design.

* + 1. OpenRocket as a reference case

Since the set of equations that would be used were for a 2-DOF, a 6-DOF analysis tool was needed to verify accuracy. Two commonly used programs for this are RASaero and OpenRocket [References 17 and 18]. OpenRocket was ultimately decided upon due to it being an open source program and for its ability to output values for various variables throughout the simulation flight. However, there was an important factor to be kept in mind while utilizing OpenRocket, that is how it calculates and applies drag. While its breakdown of drag components is quite thorough, it does not account for wave drag due to it being geared more towards creation and simulation of model rockets, in which case high Mach numbers are more than likely not present [Ref. 17]. This means that when simulating high powered model rockets, it tended to overpredict apogee and as a result, underpredict the max distance travelled in a standard no parachute launch. Due to this being the only downside to using the program, it was determined that its data would still be useful in the development of the current program. Another goal of development was to create a program that could accurately predict the launch trajectories of low angle flights. While OpenRocket was used as the reference cases for 90 to 30 degree launches, physical testing results were needed to determine the accuracy of the program for those angles below 30 degrees.

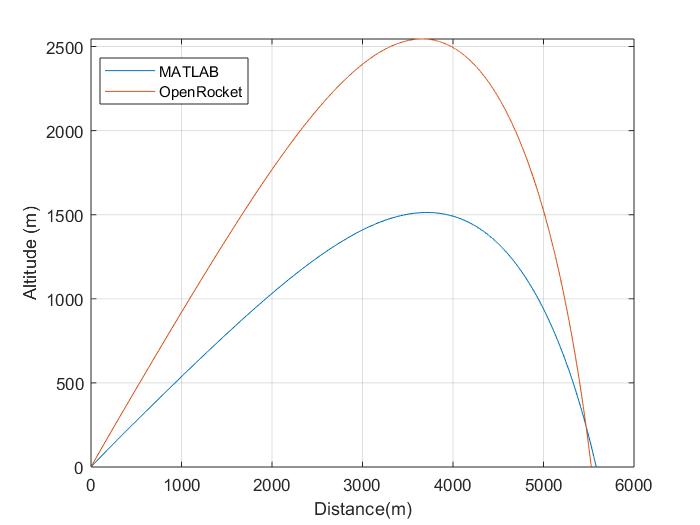
* 1. The Briggs Multistage Supersonic Rocket Flight Simulator

A key building block for the program was the program written in MATLAB by Nicholas Briggs as found in Ref. 3. The simulator he built was for a senior seminar project in the aerospace department and utilized a set if 2-DOF equations in the parallel-perpendicular frame. Using this software, some preliminary tests were performed to see how they compared to the reference case of OpenRocket. The resulting plot shows the initial test in comparison to the range and apogee of OpenRocket vs. MATLAB, and the conditions were a forty-five-degree launch angle for a single stage rocket.



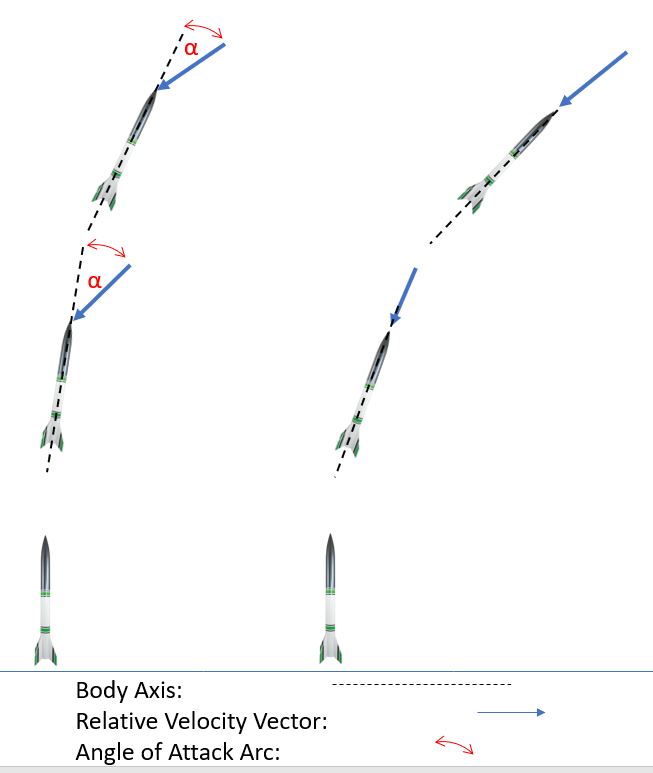
OpenRocket output vs. Briggs MATLAB output

By looking at the plot we can see that the landing distances and apogee have large discrepancies in the values predicted by the respective programs. Upon inspection, the Briggs model uses the average thrust of the motors instead of interpolating the thrust values over time. Additionally, the drag profile used is a custom generated profile defined as the ‘general small fins design’ that gives a constant value between certain values [Ref. 3]. To provide the MATLAB version with the same data as the OpenRocket, values for *CD*and thrust were extracted from the output file of OpenRocket and read into the MATLAB version. The resulting plot can be seen in Fig. 2.2.



Modified Briggs model vs. OpenRocket

While the reading in the output data helps correct the shape and landing distance of the MATLAB output, the overall apogee difference is still too much. To determine the cause of this, the initial stages of flight were examined in order to find the cause. After a few more trial runs, the problem was identified: the rockets being simulated in MATLAB were experiencing a large drop in angle in the first few seconds of flight, sometimes as much as ten degrees. This turned out to be a result of how the MATLAB program defined the rocket in space. In programs like OpenRocket, the rocket is seen as a three-dimensional body. However, in the MATLAB program developed by Briggs, the equations treat the rocket as a point mass system Therefore, several aerodynamic factors are not accounted for. Figure 2.3 gives a better look at what is happening to the rocket during its flight.



Rocket flight in 6-DOF system vs. point mass system.

In the figure, the rocket on the left represents how a rocket behaves during flight, such that the thrust vector that is aligned with the body axis, is prevented from aligning with the relative velocity vector. However, in a point mass system, no aerodynamic forces exist to prevent this from happening, and the flight path is impacted as seen in the right-hand side of the figure.

In order to counter this, the use of a launch rod would need to be implemented to keep the rocket from dropping at the beginning of its flight. This raises another concern; how would someone know how long to make the launch rod? To counter this, an assumption would need to be made to hold the rocket in place at its initial angle until a certain point to give the user an estimate for how long to have their launch rod be.

* + 1. Development of *Vcr*

Rocket’s must reach a certain velocity in order to have stable flight. It was decided that holding the angle constant until a “critical” velocity, Vcr, was met might be a reasonable solution. To obtain an estimate for the critical velocity, Newton’s 2nd law was applied to the initial motion of the rocket. Here drag is negligible and, if the launch angle is low, gravity is nearly perpendicular to the velocity vector.

(2.1)

Rearranging we get eqn. 2.2.

(2.2)

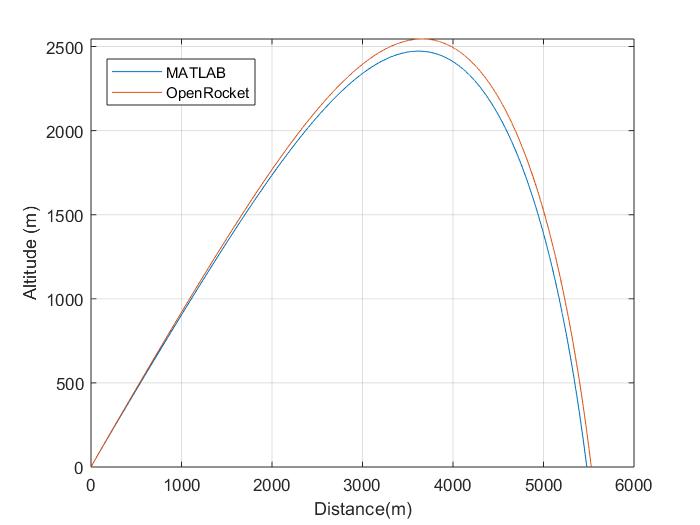
From here we can perform the following integration to provide us with an equation to use for our velocity condition.

(2.3)

Where our velocity is integrated from zero to the current velocity, and the thrust to mass ratio is integrated over the length of some distance, in this case zero to L, where L is the length of our rocket’s body. Also, mass is assumed constant during this very short period. Performing the integration leaves us with the final form of the equation.

(2.4)

With this equation derived, it was implemented into the program to see how the result compared with OpenRocket. The same conditions used in the figure below are the same from Fig. 2.1 and 2.2 but utilizing the hold on the angle as a result of *Vcr*.



MATLAB output with *Vcr* vs. OpenRocket

With this hold implemented, it provides us with a good result in comparison with OpenRocket. The differences between apogee and range are within five percent and thus acceptable for our case. Realizing that OpenRocket tends to overpredict the apogee, this result may be a bit more accurate when compared to field tests.

* + 1. The Exclusion of Lift

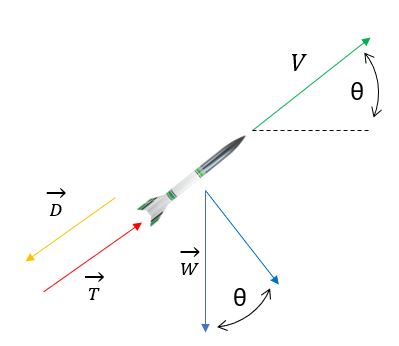
OpenRocket accounts for lift and its effects if a lift coefficient is provided, but the Briggs simulator neglects lift. Checks were made with OpenRocket to see the influence that lift might have. The results with and without lift were close enough to warrant not including lift in the analysis.

* 1. The Coordinate Frames

The simulation file provided by the Briggs program served as a nice starting point for the project, with its coordinate system being that of axes parallel to and perpendicular to the velocity vector. However, to give the user various options for solution methods, three alternate solution scripts were created along with a heavily modified version of the parallel-perpendicular frame provided by Briggs. The other coordinate frame used is for the x-y coordinate system. Additionally, both frames of reference allow the user two alternate solution methods, these being a Euler method and a 4th Order Runge-Kutta method. All equations that follow are taken from Ref. 8.

* + 1. The Parallel-Perpendicular Coordinate Frame

While the Brigg’s simulator uses this coordinate frame, that program provided more of a reference point rather than being fully incorporated to this new program. The main way this coordinate frame distinguishes itself is how it simplifies the velocity vector relative to the body of the rocket based on its current angle of orientation. Where the thrust, drag and velocity vectors are in line with the body and do not need to be broken up into components. Only weight must be decomposed into its components. A visual representation of this can be seen if Fig. 2.5 below.



Forces of parallel-perpendicular coordinate frame

The equations of motion used in this system are the same for both solution methods, however the form that they are in changes due to the way each solution method proceeds. Parallel to the flight direction Newton’s second law is

(2.5)

Perpendicular to the flight direction, the second law yields

(2.6)

Drag is expressed in the usual way

(2.7)

* + - 1. 1st Order Euler Equations

The Euler method, also known as a 1st order Runge-Kutta method, is the more simplistic method of the two, using a ‘new is old plus change’ method of calculating the values throughout the simulation. To keep the equations simple and easy to follow, some shorthand notation was created as shown in Eqs. 2.5 And 2.6.

(2.8)

This variable is a form of ballistic coefficient. In the spirit of this, a thrust coefficient was created to fill a similar purpose.

(2.9)

Implementing both into the velocity equation, and formatting for use within a For loop, we get the following expression in Eq, 2.7.

(2.10)

Note that the angle being used in this expression is for the previous value. This is because the value at the current time step uses the value for velocity to determine this as seen in Eq. 2.8. It is also important to note that the parallel-perpendicular solution script utilizes trigonometry functions such that the angle is being calculated from relative north instead of from horizontal.

(2.11)

Where the equation divides the gravitational acceleration by the new velocity, multiplies by the time step and adds it to the previous value. However, the program is written to account for the correction of angle drop within the initial stages of flight, such that the angle of flight will not change if the certain conditions occur. These conditions are: the velocity value is equal to or less than zero during the burn time of a motor, the distance travelled is less than the length of the launch rod, or if the velocity value has not exceeded the value of the *Vcr* variable.

The script will then take these two variables and use them to find the horizontal and vertical distances travelled, seen below in Eqs. 2.9 and 2.10.

(2.12)

(2.13)

Where the current distances are the product of the current velocity and angle added to the previous values for each value respectively.

* + - 1. 4th Order Runge-Kutta Equations

As stated previously, the same equations are utilized for the 4th order method as in the Euler method. However, the equations are put in their differential forms for the ordinary differential equations (ODE) solver to read and use them. The differential forms of the equations are listed below.

(2.14)

(2.15)

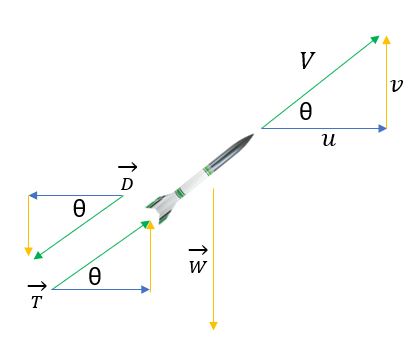
(2.16)

(2.17)

These equations are used to fill an array with the corresponding values, the process will be highlighted within chapter 3. The benefit that the 4th order method has is its increase in accuracy, as it breaks down the processes into several smaller sets to evaluate them. While the values come out relatively close to the 1st order ones, they are still the more accurate out of the two. It should also be noted that the same conditions used in the Euler method are programed into the ODE function to prevent the premature drop in altitude.

* + 1. The X-Y Coordinate Frame

The second coordinate frame used was one that utilized the equations for x and y components for velocity and distance travelled. The reason for using this system is that while using the parallel-perpendicular system the angle of orientation, θ, is calculated at each time step. This means that it has a direct influence over both the distance travelled in the x and y directions. Whereas in the x-y coordinate frame, there exist separate equations that calculate these distances based on the velocity in each direction over the angle. This can be seen by looking at Fig. 2.6, which is like Fig. 2.5, but with some of the force vectors divided up into their x and y components.



Forces of x-y coordinate frame

Here the weight force is no longer needed to be divided since it is only acting in the y-direction. The velocity, thrust and drag vectors are divided into their respective components based on the angle of orientation at that moment during the flight. As with the parallel-perpendicular frame, the same sets of equations are used, but the differential form is used for the Runge-Kutta method.

In this system the equations of motion are

(2.18)

in the x direction and

(2.19)

in the y direction. Drag is expressed as before. The cosine and sine terms can be written in terms of velocities as

(2.20)

where V is the magnitude of the velocity vector. Alternatively

(2.21)

* + - 1. 1st Order Euler Equations

Utilizing Eqs. 2.5 and 2.6 we can define the equations for the velocity components as the following expressions.

(2.22)

(2.23)

As with the parallel-perpendicular frame, there are certain conditions that will keep the rocket from dropping in velocity during the initial stages of flight. Due to the velocities being broken into components, there are set of conditions for each. For the y-component of velocity the only condition that must be met is for the velocity to be greater than zero during the first stage motor burn time for it to start gaining distance and increasing velocity. For the x-component of velocity, there must be some velocity value greater than zero for velocity to start increasing. Additionally, the rocket must reach the end of the launch rod and reach the required *Vcr* speed to start using Eq. 2.15 to calculate velocity. Before that the following equation is used to determine the x-component of velocity.

(2.24)

Which is the y-component of velocity multiplied by tangent of the flight angle. After calculating the components of velocity, the distance in each direction can be found with the following expressions.

(2.25)

(2.26)

* + - 1. 4th Order Runge-Kutta Equations

Like the 4th order method for the parallel-perpendicular, the differential form of the equations is used for the ODE solver to process. The differential forms can be seen below.

(2.27)

(2.28)

(2.29)

(2.30)

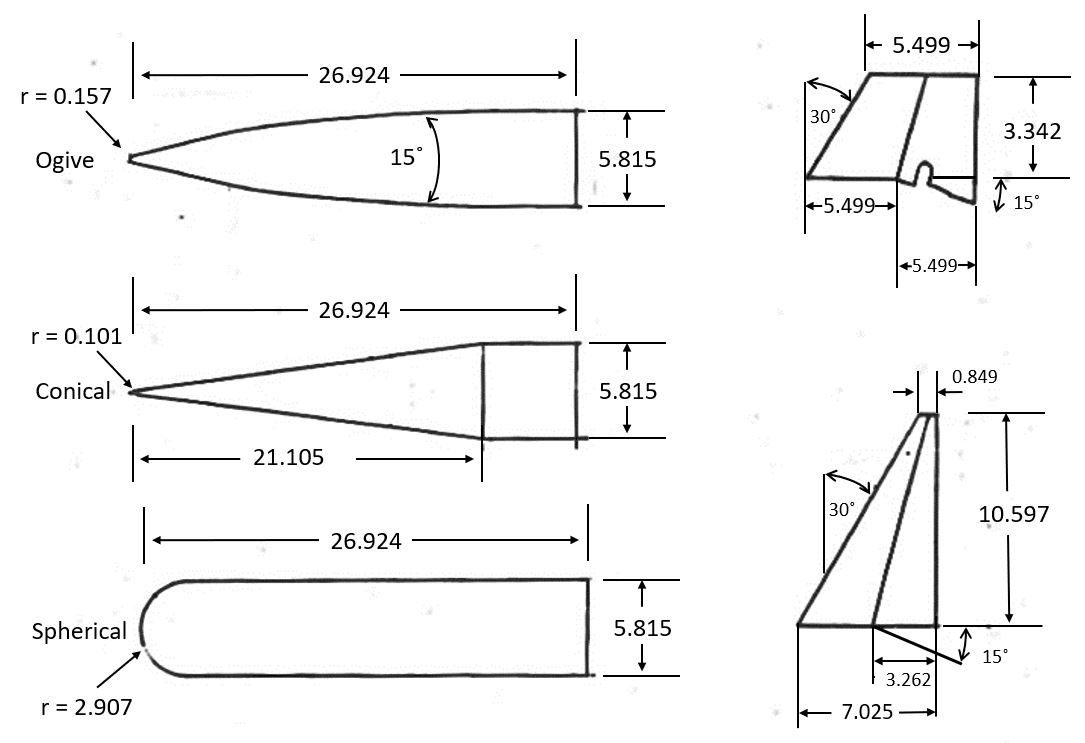
The same conditions must also be met in order to use Eqs. 2.27 and 2.28. The differential form of Eq. 2.24 is as follows.

(2.31)

* 1. Drag Profiles

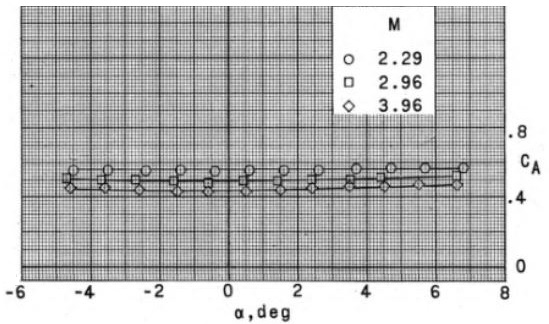
Within the 2-DOF system with negligible lift, the only aerodynamic coefficient that is needed is that for drag. In the Brigg’s model, a general small fin design drag profile was created by gathering data from relevant NASA and AEDC documents [Ref. 3]. Following the process in which that profile was created, two additional profiles were created in addition to a modified version of the small fin’s design. In the Briggs model, the *CD* is determined based on where the current Mach number falls within an array of drag values, but no interpolation is performed. In the program developed here, the profile will be interpolated to give a more accurate answer over the course of the rocket flight.

Reference 1 contains test results for several different configurations of rocket-like bodies. The most helpful of these bodies were the ones that utilized ogive shaped noses and swept wings. These two features are common among many smaller rockets. Ref. 7 provides additional data on the effects of different nose and fin configurations on normal and axial force coefficients. Examples of these are presented in Fig. 2.7.



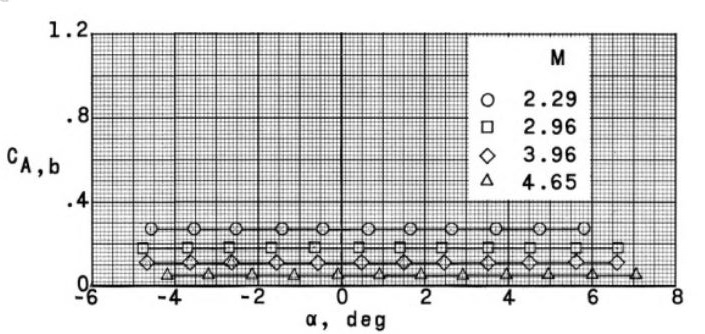
Nose and fin designs (all dimensions in centimeters), Ref. 7

For the drag profiles created, the ogive nose data set was selected. Each of the fin designs provided data for either small, large or no fins profiles. The data in question had to be gathered from the plots within the reference documents. An example of these plots can be seen below.



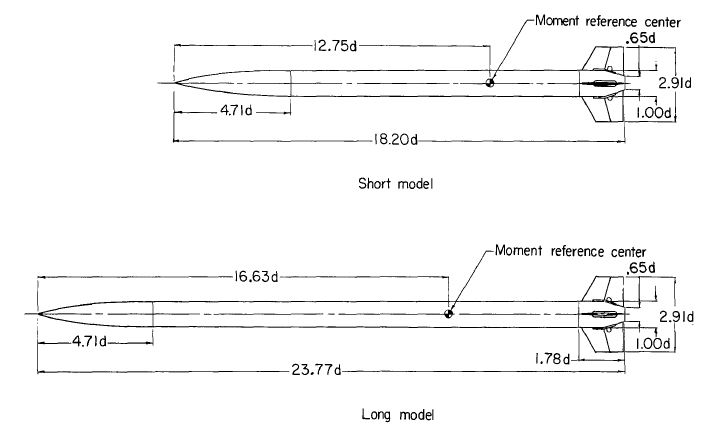
Axial force coefficient plot, edited for clarity, Ref. 10

The plot shows the axial coefficient, *CA*, versus angle of attack while each line represents a certain value for Mach. Where *CA* at zero angle of attack is the drag coefficient *CD*.In some references the axial coefficient is broken down into forebody (*CA,f*) and base axial drag (*CA,b*)*.* Where the plot for *CA*as seen above is used to show the forebody axial drag, and the base axial drag can be found in a plot like that of Fig. 2.9.



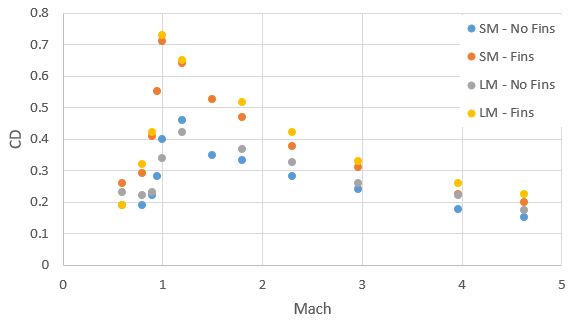
Base axial coefficient plot, Ref. 10

Given the range of sizes of the body section of the rocket, Ref. 5 was used to determine how to develop a useful model for a range of rocket body lengths. The tests were conducted from Mach values 0.6 to 1.2. The models used are as follows.



Long and short test models, dimensions in terms of diameter (5.72 cm), Ref. 5

Additionally, Ref. 2. provided further testing with the same models for higher Mach values ranging from 1.5 to 4.65. These values were added to the ones from the prior testing and used to get the following plot.



Drag coefficient profiles for several body and fin configurations.

By examining the plot, the values for the short and long models are closer in value as the Mach number increases past the transonic region. Following this study, it was determined to have the small fins model use values more closely related to the shorter model, given that the small fins drag profile is being redesigned to be used with mostly single stage rockets. The longer model data gathered was used when determining the drag profile for large fins design, since rockets with more stages are generally longer in length and more applicable to those values. The subsections will review the four drag profiles that can be used and where the values used are taken from. It should be noted that the initial drag coefficient in each profile is an assumption that higher drag occurs now of liftoff due to a rapid increase in thrust force.

* + 1. No Fins Model

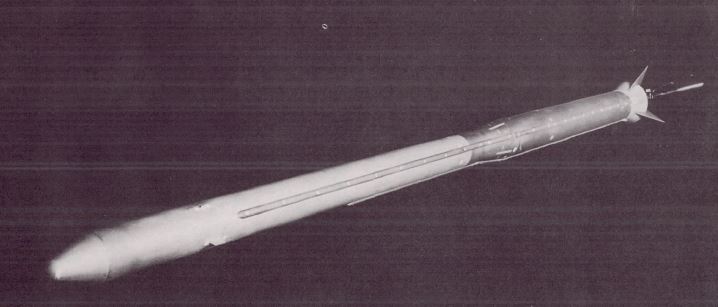
. The values for this drag profile were taken from Refs. 2, 5, 6, 10, 11 and 13. From these six references a wide range of Mach values were tested on several bodies ranging from single to multistage rockets. The overall trend from these references was that the drag coefficients do not change as the length increased. Since the fins have a large impact on the drag force acting on the rocket, removing them would eliminate a lot of the deviation in the different bodies. Even so, this model is intended to get a general idea of how a rocket will perform before the user has decided on what type of fins to use. This model is recommended for use only in the development stage or to check the performance on unique designs. The profile is presented in Table 2.1.

No fins drag profile based on Refs. 2, 5, 6, 10, 11 and 13.

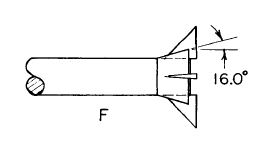
|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Mach | 0 | 0.4 | 0.6 | 0.8 | 1 | 1.03 | 2.29 | 2.96 | 3.96 | 4.65 |
| *CD* | 0.33 | 0.3 | 0.38 | 0.42 | 0.57 | 0.66 | 0.44 | 0.34 | 0.27 | 0.24 |

* + 1. Small Fins Model

The small fins model used is one created for use in Ref. 3. The values used are taken from Ref. 11 and 13 which explore the aerodynamic characteristics of the Blue Scout sounding rocket. The initial program from Ref. 3 was built with this model in mind and has remained unchanged given that the Blue Scout is an excellent example of a single stage sounding rocket. The Blue Scout rocket is shown below, along with the fin configuration used to determine the small fins drag profile.



Blue Scout, Ref. 13.



Blue Scout fin design, Ref. 13.

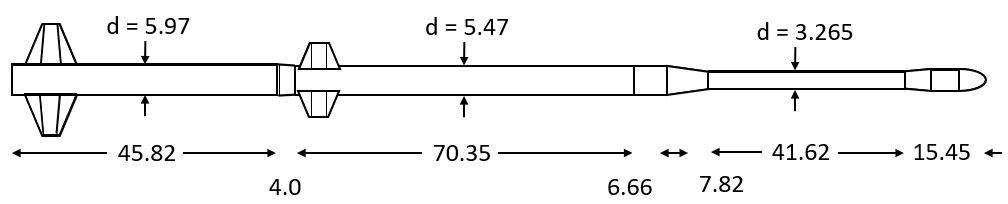
Taking all the values from these two tests, the following profile was found for a general small fins design.

Small fins drag profile based on Refs. 11 and 13.

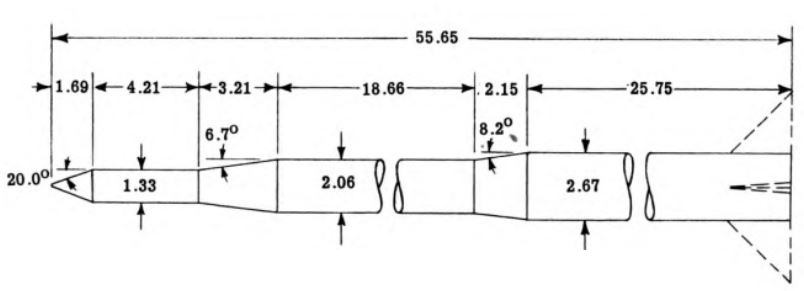
|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Mach | 0 | 0.6 | 0.8 | 0.9 | 1 | 1.03 | 2.29 | 3.96 | 3.96 | 4.65 |
| *CD* | 0.55 | 0.5 | 0.59 | 0.7 | 1.15 | 1.15 | 0.55 | 0.44 | 0.34 | 0.31 |

* + 1. Large Fins Model

Serving as the companion to the small fins profile, the large fins profile was designed for use with multistage rockets. To determine this, several documents testing the performance of several stage rockets were used to develop a general average set of drag values for large fins. References 10 and 11 detail the testing of three and four stage scale model rockets respectively, at various supersonic Mach values. Reference 6 also explores the aerodynamic characteristics of two and three stage scale model rockets from subsonic to supersonic Mach values. For the lower Mach values, Ref. 6 was used to determine a general value for the drag coefficients at the subsonic transonic range, while Refs. 10 and 11 were used along with Ref. 6 to get an average set of drag coefficients for the supersonic range from 2.29 to 4.65. The rockets used in this test are shown below, along with the table that shows the values for the general large fins drag profile, designed from averaging values of the two to four stage rockets, given the data available from the references used.



Two and three stage rocket, dimensions in centimeters, Ref. 6



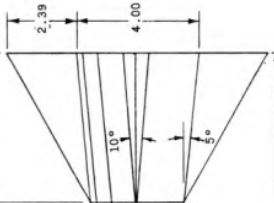
Four stage rocket, dimensions in centimeters, Ref. 11.

Large fins drag profile based on Refs. 6, 10 and 11.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Mach | 0 | 0.4 | 0.6 | 0.8 | 0.9 | 0.95 | 1 | 2 | 2.29 | 2.36 | 2.86 | 2.96 | 3.96 | 4.65 |
| *CD* | 0.5 | 0.47 | 0.58 | 0.6 | 0.75 | 0.8 | 1.2 | 1.37 | 0.62 | 0.75 | 0.68 | 0.47 | 0.36 | 0.32 |

* + 1. Delta Fins Model

Data for delta fins, such as shown in Fig. 2.15 are given in Ref. 10 and 12.



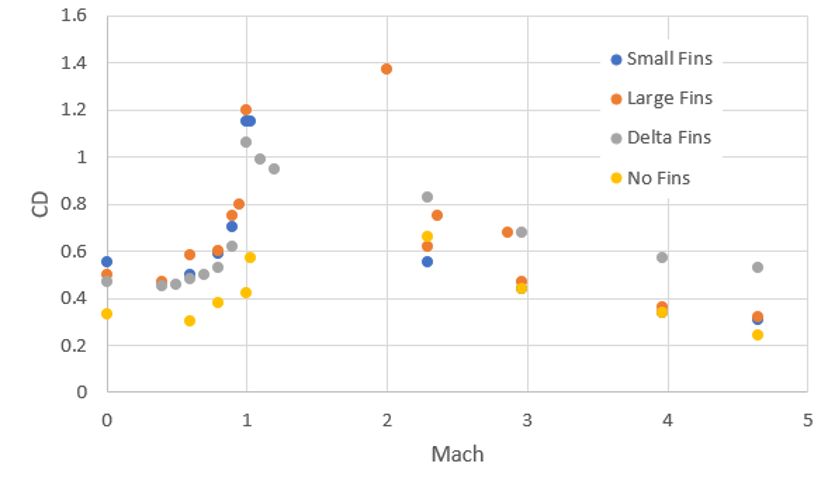
Delta planform fin, dimensions in centimeters, Ref. 10

Reference 10 provided a range of Mach values tested from 0.4 to 1.2, and Ref. 12 used delta wings in its tests from 2.29 to 4.65. Since these values provided a nice range, there was no need to average any values. It should be noted that the delta profile should only be used if using triangular wings as seen in Fig. 2.15, and small and large fins profiles should be considered before utilizing this one.

Delta fins drag profile based on Ref. 10 and 12.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Mach | 0 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1 | 1.1 | 1.2 | 2.29 | 2.96 | 3.96 | 4.65 |
| *CD* | 0.47 | 0.45 | 0.46 | 0.48 | 0.5 | 0.53 | 0.62 | 1.06 | 0.99 | 0.95 | 0.83 | 0.68 | 0.57 | 0.53 |

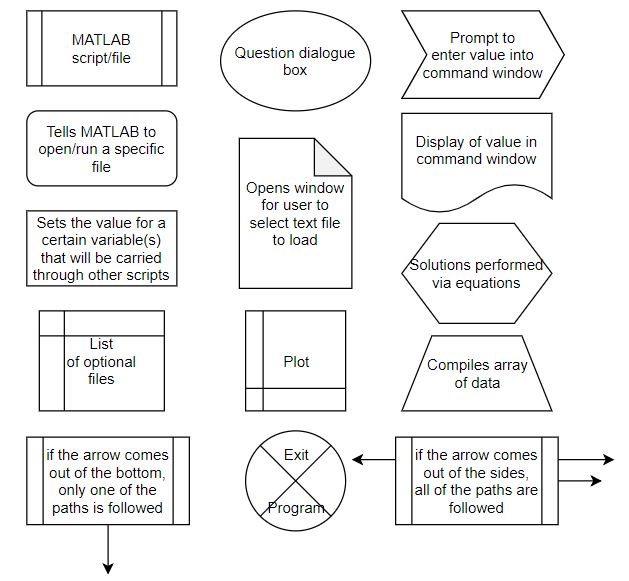
The following shows the plot of the four profiles created with their respective drag coefficients versus the corresponding Mach value.



Drag profiles

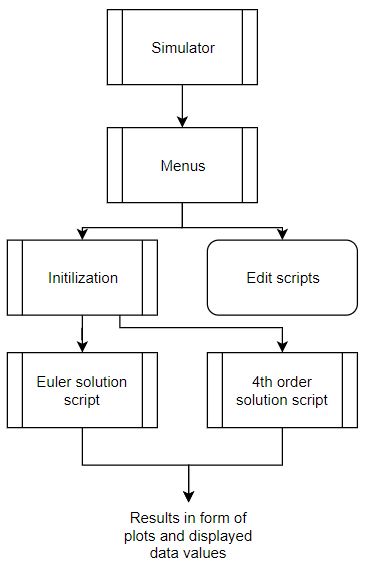
PROGRAM BREAKDOWN

In order to better describe how the program works, this chapter will break it down into smaller parts while highlighting the key functions and order of operations in each section. Figure 3.1 displays the nomenclature of the flowcharts used to illustrate the program chain of events.



Nomenclature for program flowchart

Figure 3.2 shows a simplified chain of events of how the program runs. As stated in the nomenclature figure, if the arrow for the next operation comes from the bottom of the MATLAB script box, then only one of the options is followed.

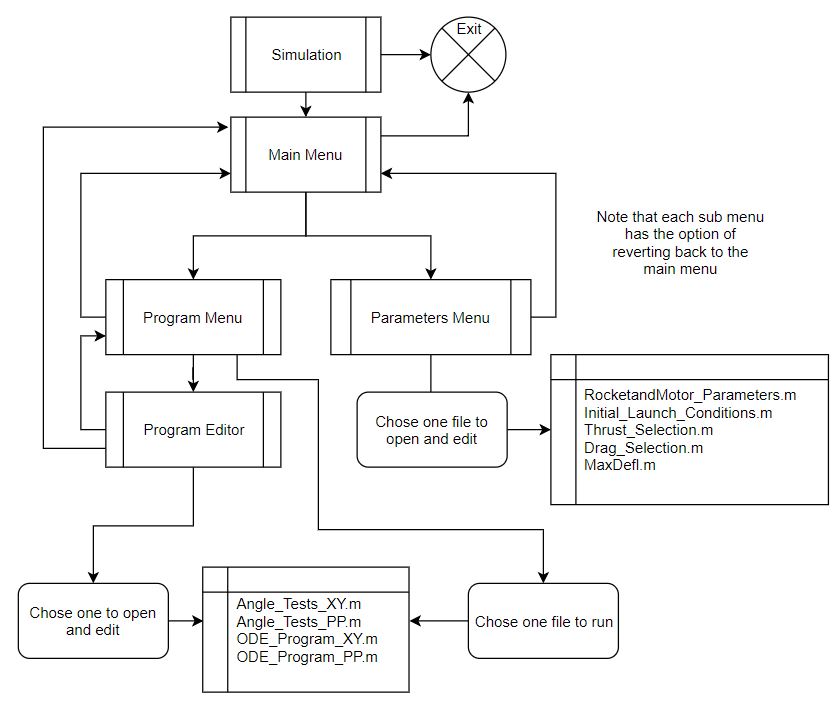


A simplistic sequence of events showing the overall process of the program.

Looking at the figure we see the general breakdown of events. The main script used is the ‘Simulator.m’ file, running this sends the user to through the menu files that allows them to either open or edit certain parameter files, or to run one out of four different solution method scripts. Notice that in the figure that the initialization script comes before the solution. This is for the sake of understanding the flow of events, the initialization file is listed as a function in each of the solution files. This means that while the user may pick a solution method before the initialization occurs, the initialization function runs as the first process in every one of the solution files. The following sections will highlight this overview in greater detail starting with the main file used in this program. All scripts can be found in the Appendix of this paper.

* 1. Startup Scripts

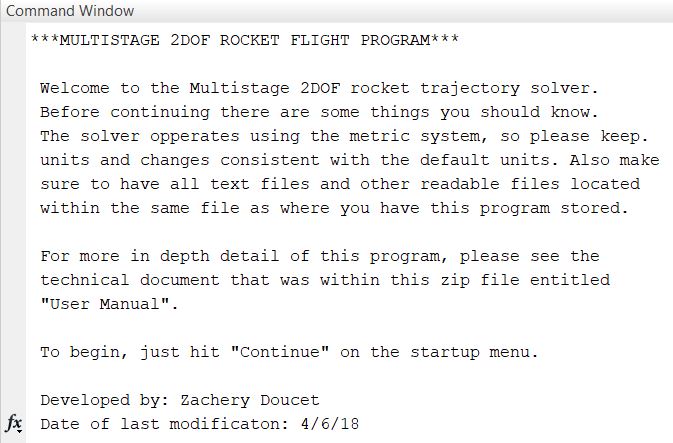
The first set of scripts that execute are classified as ‘startup’ scripts. This includes the main file that needs to be run, as well as the menu scripts that walk the user through the initial stages of the program. These first steps will allow them to either edit existing parameters or to go into the solution method selection. A breakdown of the startup sequence is displayed below.



Startup sequence that proceeds from the main file to the menu options

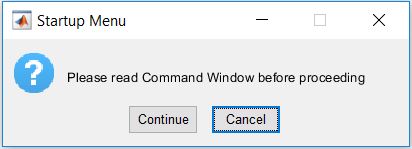
* + 1. Simulator.m

The only file that needs to be opened by a user to use the program is the ‘Simulator.m’ file. Upon opening the file in MATLAB, the user should immediately hit the run button, this will display a message in the Command Window giving a brief introduction to the program with some key details to keep in mind.



Simulator startup message

These details include stating the unit system as metric for the program, reminding the user to keep all the files provided in the program package in the same place, and directing them to the user’s manual (which this chapter doubles as). Once the entire message is read, the user will be prompted via dialogue box on whether they want to continue, as seen below.

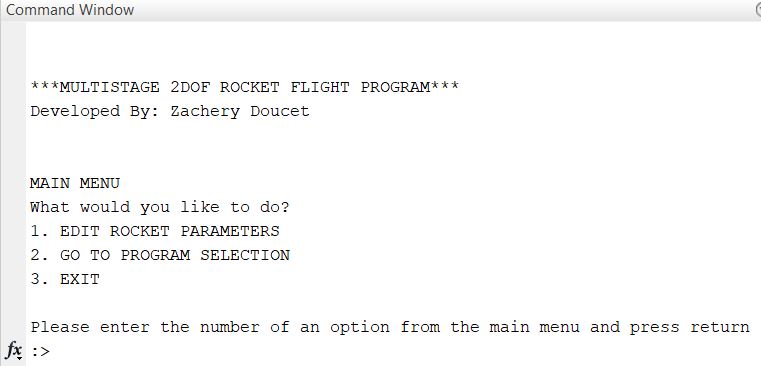


Startup dialogue box

If the user selects ‘Cancel’ then the program will cease, and the Command Window will be cleared. Should the user select continue, then the next script in the sequence will load.

* + 1. Menu scripts

When the user selects to continue to the ‘Main\_Menu.m’ file, they will be greeted with a prompt in the Command Window.

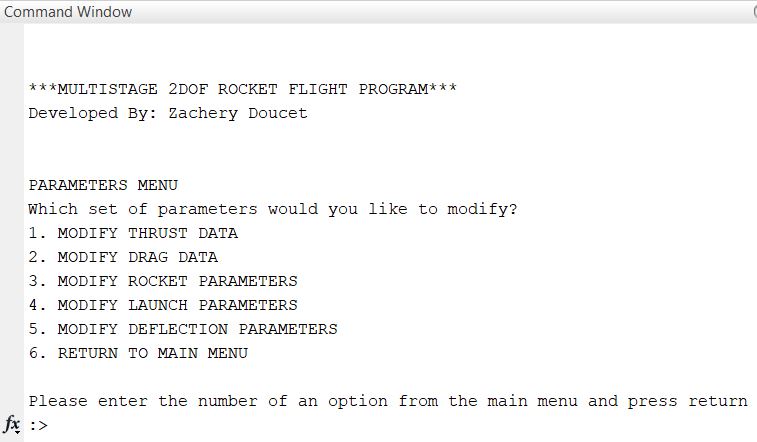


Main Menu display

The prompt will allow them to proceed to either the program or parameters menu. The two options are discussed below. It should also be noted that the option to exit the program entirely is also present in the main menu script. The template used for the menus was taken from the Cambridge University Rocketry Toolbox, a free package that can be downloaded from their website for use with MATLAB [Ref. 4].

* + - 1. Parameters Menu

The ‘Parameters\_Menu.m’ script allows for the user to open and edit scripts that pertain to certain parameters that may change depending on the type of tests being performed, as shown in in Fig. 3.7.

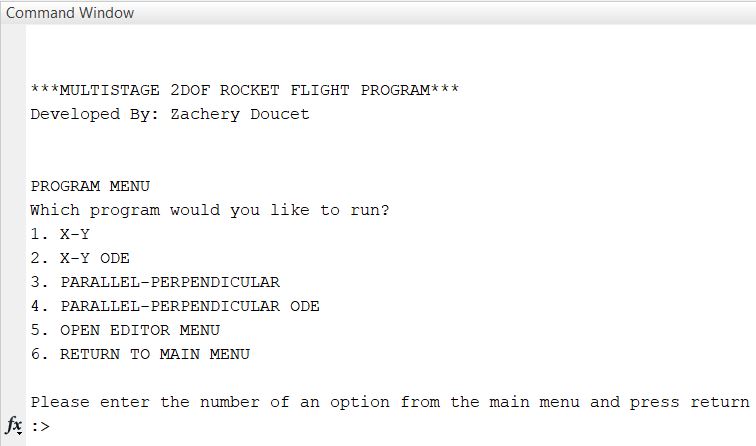


Parameters Menu

An example would be the script that contains the rocket parameter data. While this script has some preloaded rocket data, having the ability to open and insert a new rocket for a certain test flight saves the user some time and allows them to bypass manual entry of data for each time they want to run the program. Another script that is accessible from this menu is the one that contains the deflection calculations for the launch rod. Currently the launch rod has values associated with a certain material and modulus of elasticity (which will be discussed in a later section) and having the ability to edit that allows for an ease of access of modifying the dimensions and stiffness of the launch rod.

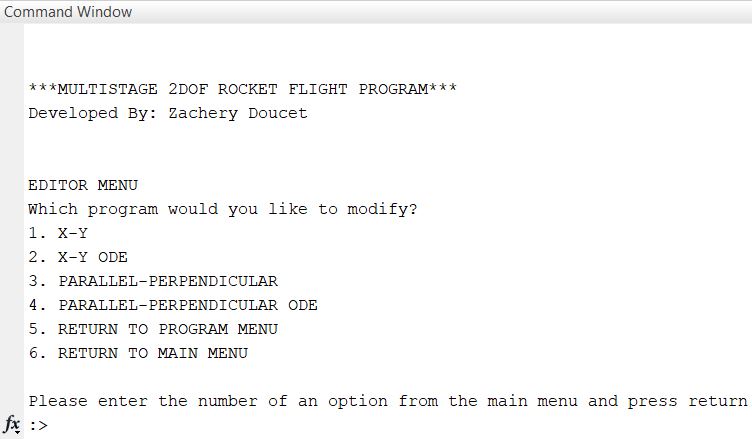
* + - 1. Program Menu & Program Editor

The ‘Program\_Menu.m’ script will allow the user to select from one of the four solution scripts, two Euler and two 4th order Runge-Kutta, each solution method allowing for the choice between the x-y or parallel-perpendicular coordinate frame as shown in Fig. 3.8.



Program selection menu

Additionally, there is an option on the program menu that allows for the user to access the ‘Program\_Editor.m’ script. This script works similarly to the program menu, but instead of running one of the four solution scripts, it opens it for modification by the user much like the parameter’s menu does.

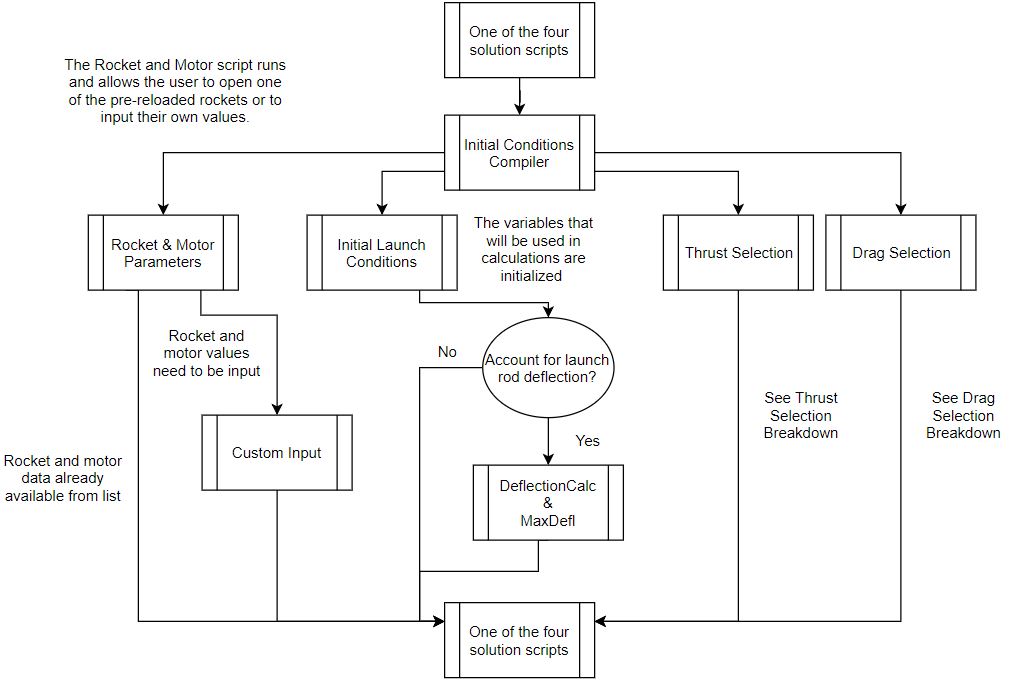


Program Editor Menu

The reason for this option is to give the user the ability to add in other values to be calculated if necessary. For example, if the user wanted to add in an equation to calculate dynamic viscosity, they could open the solution script and insert it into the corresponding program loop. Both scripts have the option to return to the main menu in case of a key stroke error, and the program editor menu has the option of returning to the program menu for the same reasons. If there is no need to edit any of the programs or parameters, and the user finds the solution script suited for their needs, the next stage of the program will begin.

* 1. Initialization Scripts

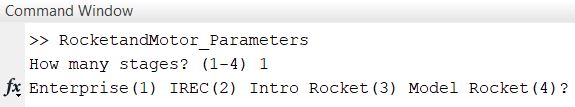
Once the user has made their way through the menus, they will have selected one of the four solution scripts to run. No matter which solution is selected, the same initialization process will run. At the start of each of the solution scripts, the ‘Initial\_Conditions\_Compiler.m’ script is called, and it will execute four more scripts that flesh out the initial values before any calculations can be performed. For this program, the term ‘compiler’ is used for the script that loads the initial conditions for the rocket parameters and the launch conditions. The sequence of events can be seen in the following figure. As stated in the nomenclature figure, if the arrows come from the side of the script symbol, all the paths are followed.



Initialization sequence

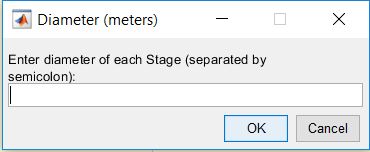
* + 1. Rocket and Motor Parameters

The first script to run in the initialization step is the one that sets the rocket and motor parameters. The main variables here are rocket diameter, structure and propellant mass, structure length, launch rod length and radius, and the average thrust and burn time per stage. This program allows for one to four stage rockets and has several preloaded rockets within the file which are recommended once the user selects the number of stages from the Command Window.



Preloaded rocket recommendation

However, if the user wants to use a custom input, they can opt out of using one of the preloaded rockets and the program will run the ‘Custom\_Input.m’ script. This will allow the user to input values for the eight parameters stated above. An example of the dialogue box used by the script can be seen in Fig. 3.12. For long term use, the user may use the parameters menu covered in the last section to permanently add in the new rocket.



Custom input box

Following the loading of those parameters, the script will then calculate the reference area for the rocket and its total mass using Eqs. 3.1 and 3.2,

(3.1)

where *A* is the reference area and *Df*is the fuselage diameter.

(3.2)

where *mt*is the total rocket weight and *ms* and *mp* are the structure and propellant mass respectively. Following these calculations, a nested For loop then calculates *mi* and *tbo* for each stage using Eqs. 3.3 and 3.4.

(3.3)

(3.4)

The initial iteration runs from i = 1 to *nos*, the total number of stages in the rocket configuration and the array for initial mass and total burnout time are initialized. The first nested For loop runs Eq. 3.3 from *j* = *i* to *nos*, so *ms* and *mp* can be added without interfering with *mi* as it stacks with each iteration. A second nested For loop calculates Eq. 3.4 from *k* = 1 to *k* = i. Burn time and an additional wait time, *twait*, are added to *tbo* which then stacks for each successive iteration. The additional three seconds added to each stage are to account for the time that the rocket’s onboard computer takes to process that the stage has stopped burning, and the next three seconds are to account for the time it takes for the first stage to detach from the rocket and clear the area for the second stage motor to fire. After the For loop has completed and created its corresponding arrays, Eqs. 3.5 and 3.6 are used to determine final mass and mass flow rate for each stage.

(3.5)

(3.6)

Note that the flow rate equation used here is an assumption made by dividing the propellant mass by the burn time, and while not exact, it provides a logical value that can be used to track the decrease in propellant mass. Following the calculations of burn out times, Eq. 3.7 gives the expression that determines the total burn out time of the rocket.

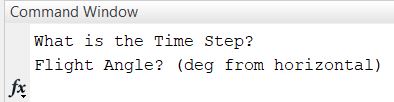
(3.7)

Where the burnout time of the previous stage is added to the burn time of the current stage’s motor. Following the completion of these calculations, the script is complete and exports the data into the compiler, which then proceeds to the next script.

* + 1. Initial Launch Conditions

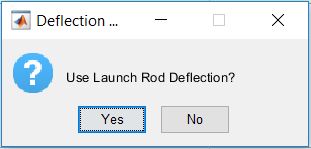
The main function of this script is to initialize certain variables to be used with the For loop in the Euler solution scripts, or the 4th order Runge-Kutta solver within that set of scripts. The variables that are initialized at zero are *a, t, x, y, z,* and *M*. Variables initialized to certain values are *T, P, ρ,* and *c*, which are set to the standard sea level values as found in the U.S. Standard Atmosphere model [Ref. 19], as well as *Rg, Re,* and *go* which are the gas constant, radius of Earth, and standard gravity value. Additionally, the initial values for *m, g,* and *A*, are set by accessing specific sub-functions that will be explored in more detail in a following section.

There are two prompts that are used when this script loads, the first asking the user to input what time step they want to use for the flight, and the second asking what the initial flight angle is. The time step can range from being one second intervals, to using thousandths of a second intervals, although it is recommended that no time step larger than ten seconds or smaller than a thousandth of a second are used. The initial angle is input in degrees before being converted to radians, since that is the format in which MATLAB’s trigonometry functions work, and the range that can be used in the program ranges from 0-90 degrees. The display can be seen below in Fig. 3.13.

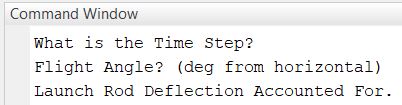


Time step and flight angle prompt

After the initial angle is entered, the program will open a dialogue box asking the user if they want to account for the angle deflection. As mentioned in chapter 2, a launch rod was needed in order to prevent the harsh angle drops at the initial stages of flight; however, depending on what the material and dimensions of the launch rod are, it may be subject to bending.



Rod deflection box



Deflection display

If the user choses to not to account for this, the message will show that the deflection was not accounted for and the program will continue without performing the calculation.

Following the selection to account for deflection, the program will load the deflection sub-function. This sub-function takes the initial angle and uses the following sequence of equations to return the launch angle once it leaves the launch rod.

(3.8)

The equation for moment of inertia for the launch rod, which uses the value for the rod’s radius from the previous script covering the motor and rocket parameters. This equation is used for a thin rod that is undergoing a loading as taken from Ref. 9. Going back to section 3.1, it can be seen in the flowchart for the menus, that the parameter menu has the option of opening the deflection sub-function to make these changes. Because the rocket is being launched at an initial angle, the component of weight acting on the downward direction is needed. Equation 3.9 provides this component.

(3.9)

The angle the rod deflects is given by Eq. 3.10.

(3.10)

The default material in the program for the rod is stainless steel (value of 1.96 x 1011 Pascals). However, it be modified by accessing the sub-function through the parameter’s menu. Finally, the deflection angle is subtracted from the initial flight angle to give the angle of flight when the rocket leaves the rod.

(3.11)

The sub-function sends this value back to the initial flight script where it becomes the new initial flight angle.

The final calculations that are done by this script are the calculations of the initial values of the x and y components of velocity. During testing an error occurred when a value of zero was used for the initial velocity. This error resulted in the rockets trajectory being reduced to miniscule distances instead of a proper flight path. To prevent this, a small velocity value needs to be input as the initial value. Taking that small initial *V* value, we use Eqs. 3.12 and 3.13 to find the components of the initial velocity.

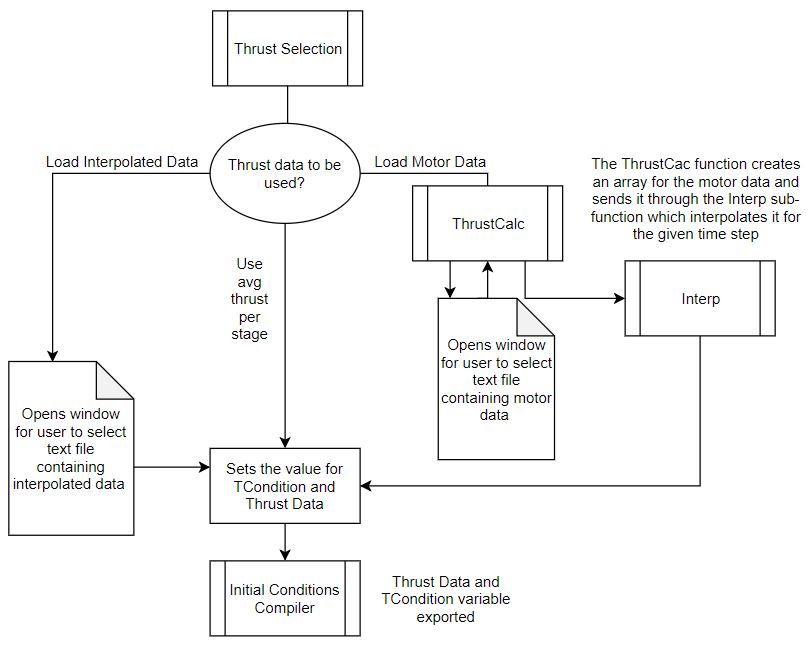
(3.12)

(3.13)

Where the theta used is the flight angle, whether the deformation calculation was performed or not. After the all values are initialized, they values are transferred back into the compiler which then proceeds to the next script in the list.

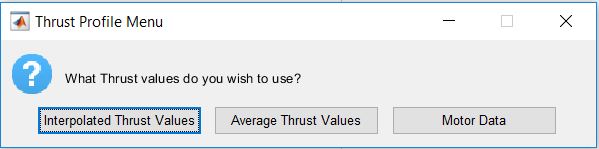
* 1. Thrust Selection

The thrust selection script exists to give the user a wide variety of what thrust values to use. Looking back at Fig. 10, the branch for this script has a message to look at its own individual breakdown, which is presented in Fig. 3.16.



Thrust Selection flowchart.

As scene in the Fig. 3.16, the two values that are exported are the thrust condition, which is the term used by the program to determine how to load the thrust data variables, and data variables, which will be used in whichever solution script they have chosen. Additionally, when the compiler script accesses this file, it displays a dialogue box to be answered. The dialogue box determines what conditions are set and what actions are performed as seen in Fig. 3.17.



Thrust Selection dialogue box

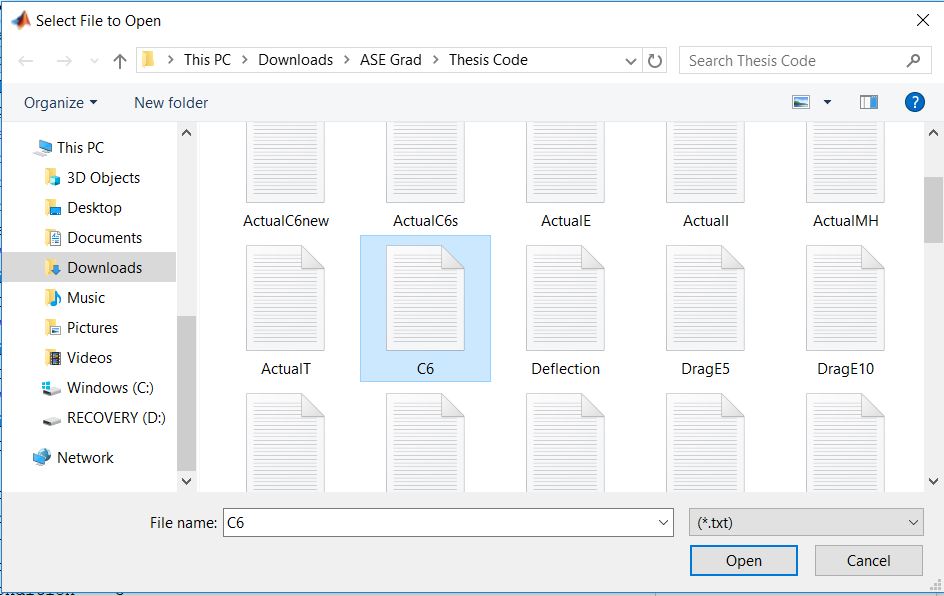
The dialogue box provides the user with three options, to use an interpolated thrust data set, to load the motor data for the rocket, or to use the average thrust per stage. Each option will be detailed below.

* + 1. Average Thrust Values

When selecting the average thrust values, the script will set the condition and then set the drag data value to zero, as there is another sub-function that will create an array for the average values within the solution scripts. The basic function of the script sets the thrust value as constant through the duration of the burn time. This script will be discussed more in section 3.5.

* + 1. Interpolated Thrust and Motor Data

These remaining two options that are available both require the loading of a set of data from a text file. Upon selecting either interpolated or motor data, a window will open allowing the user to access their files and find the file they wish to load. The script is written to find and read only text files. This can be edited from the parameters menu to fit the user’s need should they want to load from a different type of file. Upon selecting the corresponding options, a window will open to the current operations file and let the user select their file to be read in, in this case of Fig. 3.18 a file containing the C6 motor data is being selected.

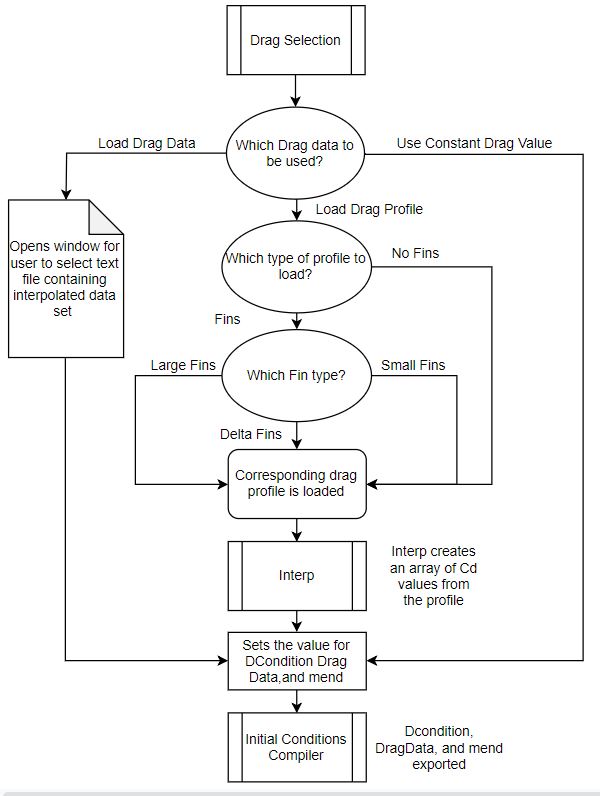


File selection window

If the option they chose was for a set of already interpolated data, then the data is loaded into the thrust data array. It is important to keep in mind that doing so will not edit the data in any way, so it is important for the user to make sure that the data set is compatible to their chosen time set. However, if the motor data option was selected instead, then the data will be sent to a pair of scripts that will perform a linear interpolation between input data points. The sub-functions are named ‘ThrustCalc’, which handles the organizing of the motor data into an array and finds the slope and y-intercept values associated with those values, and ‘Interp.m’, which takes the array of slopes and intercepts and performs the interpolation at specified times. The interpolation creates an array of thrust values corresponding to the time step of the program chosen by the user. The values for thrust are then loaded into an array, sent back to the thrust selection script and loaded into the thrust data variable. Once the thrust script has completed loading the thrust data, it exports the it with the condition variable back into the compiler script.

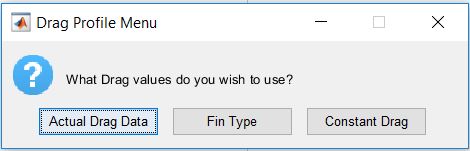
* 1. Drag Selection

Much like the script for thrust selection, the script responsible for loading the drag data has its own breakdown with a similar flow. Like the thrust script, the drag selection script will allow the user to select between three different ways of loading a drag profile. The breakdown and is displayed in Fig. 3.19.



Drag Selection flowchart

Once the script is loaded, the dialogue box shown in Fig. 3.20 is opened allowing for the user to select which option they want for the drag coefficient data.



Drag Selection dialogue box

* + 1. Actual Drag

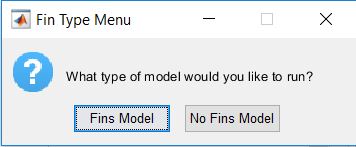
The first option that the script allows for is that of actual drag data. This allows for the user to open a text file containing a series of drag coefficients to be used as the drag data array. The window that opens identical to the one seen in Fig. 3.18. The user should make sure that the set of data they load is compatible with their current time step and that the drag coefficients are consistent with the range of Mach numbers that their rocket experiences during the simulation.

* + 1. Constant Drag

The constant drag option is not recommended if the user is trying to determine an accurate flight; however, it can be helpful in trying to determine what sort of drag coefficient the user should try to design their rocket to have if they need it to go a certain distance. When this option is selected, the drag data array is set to zero, and the drag condition value will tell the solution script to ask for the drag coefficient value when it gets to that point in the program. That input will be discussed in section 3.5.

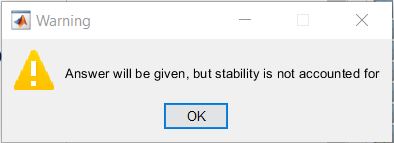
* + 1. Fin Type Profiles

The final option for loading a drag data array is to utilize one of the preloaded drag profiles that are detailed in section 2.4. Choosing this option will queue a second dialogue box to open asking if the user wants to load a fin profile, or use the profile built with no fins in mind.



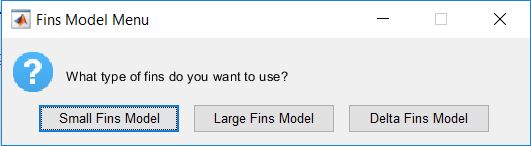
Profile selection box.

If the user choses to use the no fin model, that array will be loaded. Additionally, a warning message will be displayed informing the user that the use of no fins means that their rocket will more than likely be unstable, and a smooth trajectory is not expected. The option is still provided in case the user wants to know how well the rocket body performs before choosing how to design their fins.



No Fins warning message

If the user choses to use a fin type model, a final dialogue box will be opened, allowing them to choose from one of three fin types. The user should refer to the details provided for each drag profile within section 2.4 in order to determine which profile is best for their case.



Fin Type selection box

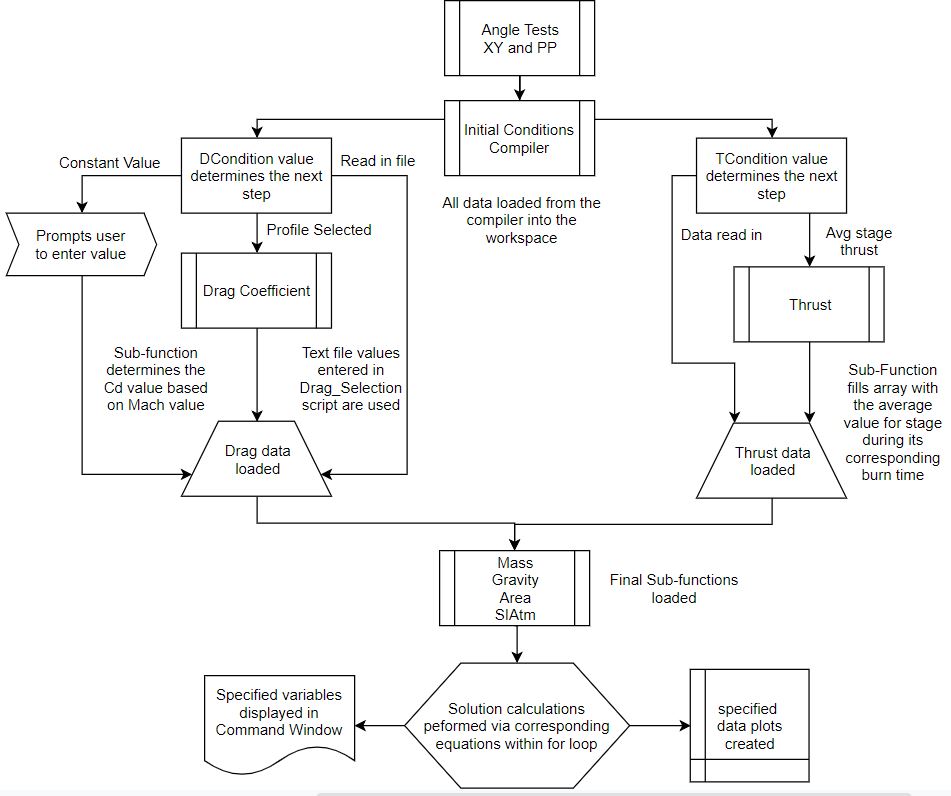
Once one of the profiles is selected, a fin type variable will be set, and the drag selection script will then load the corresponding sub-function respective to that fin type. Each sub-function has a set of drag coefficient and Mach values, as detailed in section 2.4. The sub-function that is selected performs the organization of data into an array and finds the slope and y-intercepts like ‘ThrustCalc.m’ for the motor data interpolation. That array is then exported into the same function that interpolates the motor data from the previous section, ‘Interp.m’. After the profile is interpolated, it is exported back to the drag selection script along with the highest Mach value available from that drag profile, this will be helpful when the drag data is loaded by the solution script. Once the drag data is loaded into the selection script, it is the exported with the final Mach value, *Mend*, and the drag condition variable, back into the compiler. With this script complete, the compiler has finished running and will export all the data it has collected into whichever solution script the user has selected, which will then use that data with its corresponding equations to perform the final calculations of the program.

* 1. Solution Scripts

The final step in the overall program process is the execution of a solution script which will calculate the trajectory of the users chosen rocket using the equations corresponding to the coordinate frame and method of solution as defined in chapter 2. As stated in the previous sections, all the solution scripts contain a call to the initial condition’s compiler in their first line. All values found during that script are exported into the workspace of the corresponding file before any further calculations are made. Aside from the equations used in the different frames, the method of solution is relatively the same, thus both Euler and Runge-Kutta methods will be discussed in the same subsection with any differences being noted.

* + 1. 1st Order Euler Solution Scripts

The figure below shows the flow of events for the Euler method solution, where both the coordinate frames follow the same sequence.

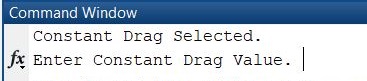


Euler method solution scripts flowchart

After the initial condition’s compiler brings in all the values it calculated, the first thing that is calculated is the number of steps to be used, in this case represented by *n*. This value is set by dividing the total burn out time by the time step variable and multiplying by some scalar to make sure that there are enough data points to plot the entire flight path.

(3.14)

Following this, if the user chose to have a constant drag value instead of an array, they will be prompted to enter that value into the Command Window.



Drag coefficient prompt

If they chose one of the other options, then drag data from the corresponding array will be used inside of the For loop.

Once the script gets to the For loop, it will first interpret what values to use for thrust and drag based on what condition values have been loaded. In the case of drag, the constant drag condition will set all the *CD* values equal to that for the entire flight, and if they chose a fin type profile, a sub-function will be loaded to find the corresponding *CD* value at each step based on the Mach number at that time. If the user chose to use the stage thrust for the motors in the rocket, the same process is followed as the drag, having a sub-function find the thrust value at a given time step based on what the time variable is at that given step. If the user chose to read in a text file for drag data or thrust data, the interpolated values that were found in the drag selection or thrust selection script will be loaded here to be used. For thrust however, the values for that array will be used until the total burn time is reached, after that thrust will be set to zero.

Following the loading of the drag and thrust data, the corresponding equations for each frame will be used to find the values of velocity, angle, and distances. Additionally, several sub-functions are used to find the values of gravity, area, atmospheric conditions and mass during the flight, but these will be covered more thoroughly in section 3.6. At the end of each iteration, there are a few equations that are needed for certain sub-functions to work.

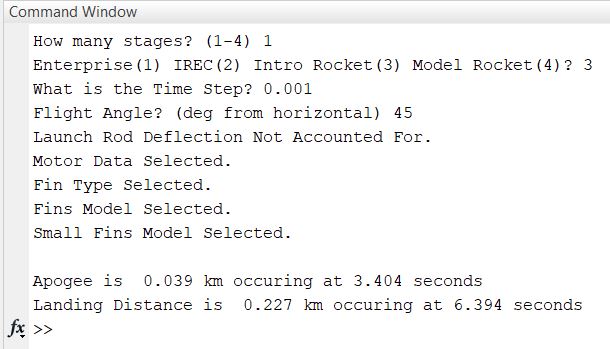
(3.15)

(3.16)

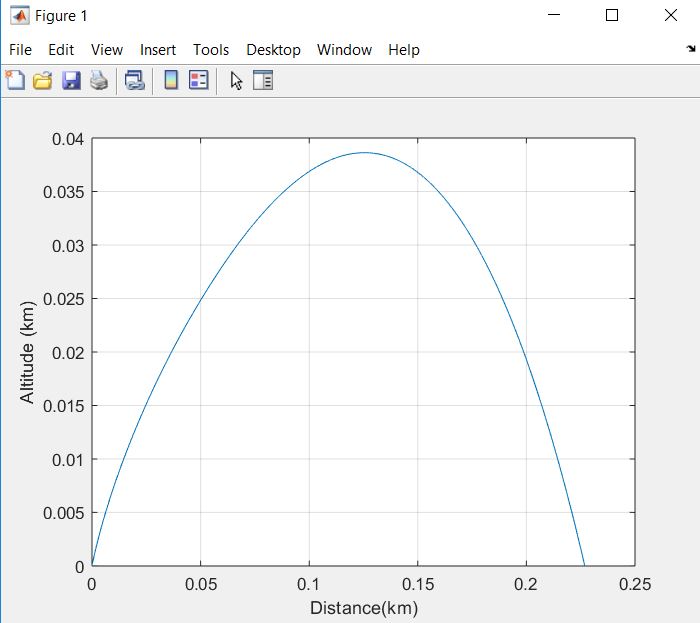
(3.17)

Which are the equations for speed of sound, Mach number and dynamic pressure. The speed of sound is needed to find the Mach value which determines what value for the drag coefficient to load, and while dynamic pressure is not required, it is still nice to have as a check. There is a condition at the end of the For loop that stops the loop before it runs its max number of iterations if the *y* value calculated is less than zero and is implemented because at this point the rocket has hit the ground.

After the For loop is done running, conversions for units are applied to the values found in the loop. This is the reason why these scripts are available to be edited from the editor menu. The user may wish to convert to a different unit that is currently in place or may wish to add further calculations to the For loop depending on what they need. After the conversions, maximum values are found for specified variables, and the index at which they occur are found to find the time at which the occur. The result from the end of the solution script is a single plot that shows the x and y distances covered, and a message in the Command Window that shows the value of apogee and landing distance and how much time it took to get to those two points. An example of the final display of the Command Window and the figure created upon completion can be seen in Figs. 3.26 and 3.27 respectively.



Final Command Window display

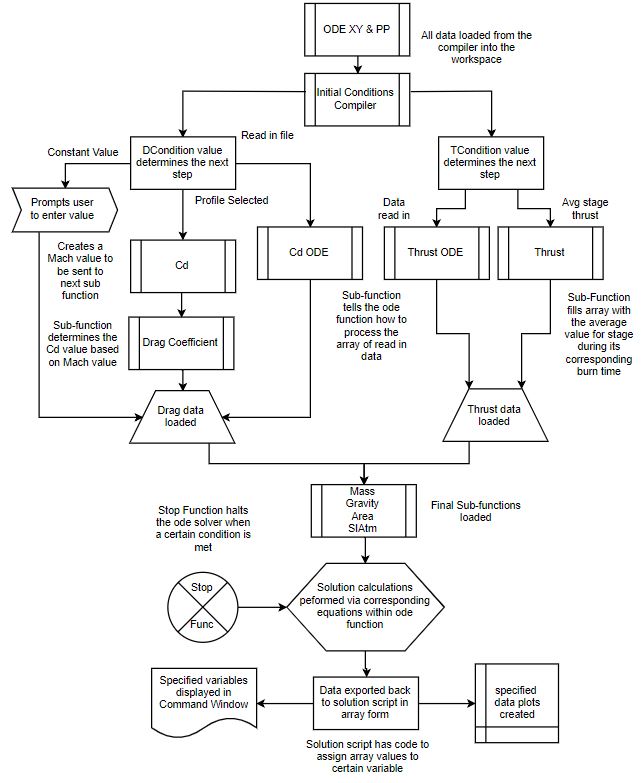


Flight path plot

After the plot and values are displayed, the program has finished running. Additional plots can be added into the script using the editor menu, as well as modifying the axes and units used for plotting. If the user wishes to display additional variables upon completion of the solution script, the option to add those displays is available from the program editor menu.

* + 1. 4th Order Runge-Kutta Solution Scripts

The sequence of events for the 4th order solution method is like that of the Euler, but with added events that are needed for the user of the ODE solver used in MATLAB.



ODE Solution Flowchart

After the script receives the values from the compiler, a similar process occurs to that of the Euler method scripts. The *n* variable is still calculated but this time it is used to provide a range of variables for the ODE solver called *ts*. If the constant drag option was chosen, the same message will be displayed in the Command Window as appears in Fig. 3.25. After this, the ODE scripts then diverge to their own process, first setting up an initial set of values for the differential equations in the scripts corresponding ODE function. For the parallel-perpendicular frame the initial values are for *V, θ, x, z,* and *m*, whereas in the x-y frame the values are *u, v, x, y* and *m.* These values form the first row of an array that will be filled in by the ODE solver utilizing the differential equations for the corresponding frame found in chapter 2. The differential equation for mass in both systems is simply the output of the time based ‘Mass.m’ sub-function to account for the decrease of mass over time. This sub-function will be explored in the following section. The call to the ODE solver built into MATLAB appears as such.

(3.18)

Where the *t* and *h* variables used tell the ODE function to use time as the differential factor, and have *h* be the values to load and build into an array. The function is then called, ‘ode45()’ being the 4th order solver for Runge-Kutta in MATLAB language, to be evaluated at the two established variables, using the function of the users choosing. Inside the function that is chosen are the differential equations for the respective coordinate frame, i.e. ‘ode\_func\_xy.m’ is the function that contains the equations for the x-y coordinate frame. That function will then load the time and initial values into it and perform the calculations across *ts*starting with the initial array of values called *h0*. The ‘Opt’ in the expression represents an additional function in the ‘options’ slot of the solver call. In this case, ‘Opt’ fills the same role as the condition at the end of the Euler For loop. The variable is set to run ‘StopFunc.m’, which contains the required expression to stop the ODE solver from running once thevalue for altitude becomes less than zero.

After the ODE solver completes its calculations for the initial variables over the time span or until the stop condition is met, it returns the full array of values. The script then takes those variables and separates them into columns in order to find the max values for each. Those max values are found along with the corresponding time index for each variable, and the results are plotted and displayed into the Command Window much like is seen in Figs. 3.26 and 3.27. Like the Euler scripts, the option to open the ODE scripts is made available if the user desires to use a different set of differential equations in the solver, or if they want to add in any variables for additional calculations. Once the solver is finished, and the values and plot are displayed, the program had completed its run.

* 1. Additional Sub-Functions

Not touched on in the previous sections are the smaller sub-functions that serve the role of providing values within the For loops and ODE functions of the solution scripts. These are key functions and are used to calculate certain variables within the loop based on what the input for them are. Due to their short length, they have been collected here.

* + 1. Mass.m

The mass sub-function utilizes *md* calculated in the rocket and motor parameters script, to find the current mass of the rocket at a given time. It uses a system of If loops and the current value for time to determine the amount of propellant mass remaining within the rocket. For single stages, once the motor stops burning the value for *m* remains constant. If the rocket is a multistage vehicle, it will use the values for *tbo* to determine when the next motor starts to burn, using the corresponding mass flow rate for each motor as it proceeds through the flight.

* + 1. Gravity.m

The gravity sub-function utilizes the input of *z* or *y* to determine altitude. It then uses the following equation to determine what the gravitation acceleration is at the given altitude.

(3.19)

Note that Eq. 3.19 shows the parallel-perpendicular version of this equation, the x-y frame utilizes the *y* variable in place of *z*.

* + 1. Area.m

The sub-function for area uses the reference area(s) that are calculated in the rocket and motor parameter script. If the rocket is a single stage, the sub-function will always return the same value for area. However, in a multistage rocket, the sub-function will use the input of time to determine what current stage of the rocket is based on the values for *tbo.* So that when the stage is separated from the body, the reference area changes and the correct value is returned.

* + 1. SIAtm.m

This script uses the input of the *z* or *y* variable to determine what the values a for *T, P* and *ρ* are. This script is taken from the breakdown of the standard atmosphere in metric units as found in Ref. 19.

TESTING AND RESULTS

Following the completion of the program, several tests were performed to determine the accuracy of the system with the goal being to have the software be within a five-percentile margin of accuracy to the reference case. The following subsections will detail these comparisons and the findings of each case.

* 1. Comparison to Reference Case

The first step was to compare the results of the software to the reference software found in Ref. 17. To do this, the reference program was tested at five degree increments from 85-30, 30 degrees being the lowest the reference case would allow for a simulated launch. By doing this the program could be compared to the higher angle tests to make sure that it was valid for those types of launches. After collecting the values for *tx, tz/ty, Xx,* and *Zx/Yx* from the reference case, the four different solution scripts were tested at the same range of launch angles. The rocket in question was the one found in Ref. 14 pictured below.



Enterprise rocket, Ref. 14

The following table highlights the average difference in percentile form that each solution method produced for this single stage rocket.

Average difference percentile for all scripts

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | X-Y Euler | Par-Per Euler | XY 4th order | Par-Per 4th order |
| *Xx* | 3.99 | 1.67 | 1.64 | 1.49 |
| *tx* | 1.87 | 0.30 | 0.78 | 0.66 |
| *Yx* or *Zx* | 4.53 | 1.21 | 1.02 | 0.86 |
| *tx* or *tx* | 2.51 | 0.74 | 0.56 | 0.56 |

By observing the average differences and comparing them to one another, the 4th order Runge-Kutta method in the parallel-perpendicular frame is the most accurate to the reference case.

To verify its accuracy, a second test was conducted over the same range of launch angles but using a second stage rocket as the test subject.



*Mostly Harmless* rocket, Ref. 15

Using the required parameters for the rocket within the program, the overall average difference for each of the variable is shown below for the 4th order Runge-Kutta script in the parallel-perpendicular reference frame.

Average difference percentile of 4th order parallel-perpendicular case

|  |  |
| --- | --- |
|  | Par-Per 4th order |
| *Xx* | 2.83 |
| *tx* | 1.38 |
| *Yx* or *Zx* | 2.44 |
| *tx* or *tx* | 1.45 |

By looking at the data as the number of stages goes up, the increase in difference is about one percent, this is suitable for our case and it can be assumed that using three and four stage rockets will be within a five percent margin of accuracy as well.

* 1. Field Testing Results

With the range of 30-90 degrees covered by simulation comparison, field tests would be needed to verify the validity of low angle tests. In order to do this, two separate tests were conceived, one that utilized a small model rocket kit at low angle launch, and a second test that would utilize several handmade rockets constructed by the freshmen aerospace class for testing.

* + 1. Small Model Rocket

The small model rocket used was under a third of a meter in length and had forward swept fins. The rocket as seen below was then launched at an initial angle of thirty-five degrees, along a thin stainless-steel launch rod. The use of the deflection analysis script was used to account for this due to the rod being so thin. The rocket in question travelled approximately 250 meters in distance. Using all four scripts, these are the results.

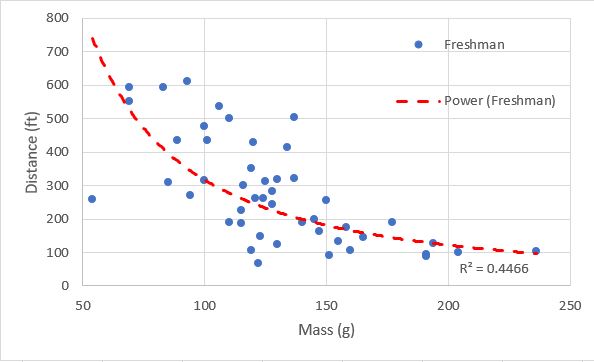
Field testing vs. simulation, units in meters

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Actual | XY Euler | Par-Per Euler | X-Y 4th order | Par-Per 4th order |
| 250 | 216 | 221 | 233 | 250 |

As with the higher angle tests, the 4th order solver for the parallel-perpendicular coordinate frame is the most accurate in this case as well. However, due to environmental and measuring errors, it is quite possible that any of these values could have been correct. This reinforces the need for multiple solution methods for the user to get a range of values prior to testing so that plans can be made accordingly.

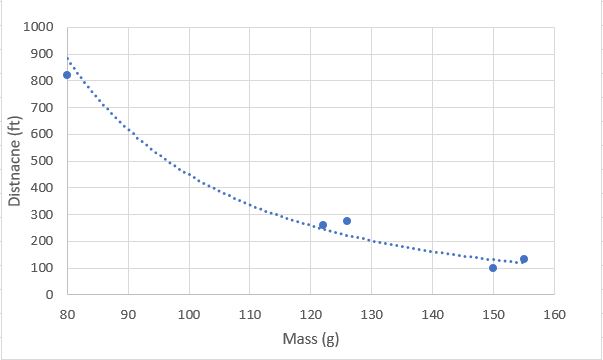
* + 1. Hand-made Rockets

In addition to the small model rocket, the freshman aerospace class was tasked with creating their own rockets out of paper towel roll tubes. In the build up to this they were instructed on how to write a simplistic version of this system, using only the x-y coordinate frame, with the Euler method of solving.



Freshman class test plot

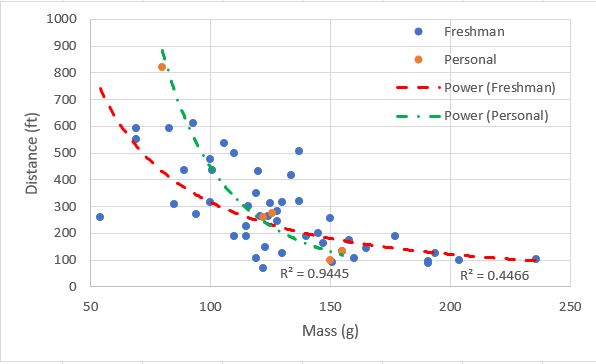
While a tad erratic, the overall curve shows a general decline in mass versus distance, due not only to the effects of gravity, but to how much the rod deflected under the weight of the rocket. There are also environmental conditions that should be kept in mind for some of the higher and lower performing rockets, stability and wind speeds being the biggest factors that could affect the rocket’s flight path. The following plot shows a more contained test, including four of these self-made rockets tested personally to get a feel for how they would perform, along with the data point for the small model rocket from the previous section.



Personal rocket resting

Since all the rockets that were built in this case were constructed following the same design mentality, they performed very similar to each other. Still environmental conditions had effects on these tests, due to crosswind speeds and deflections of the launch rod.

Following these tests, the freshmen were tasked to right a paper explaining how their rocket performed and how much they thought the launch angle had deflected. Most predicted that the launch rod deflected more than it did when compared to the deflection accounted for by this program. This lines up with the results of the x-y Euler method underpredicting the trajectory of the rockets. The final plot below is a combined version of Figs. 4.3 and 4.4.



Combined data plots

CONCLUSION

To summarize the process of developing this program, the work completed here provides the user an accurate resource for students entering the aerospace profession. While not the most accurate system, the program that was developed gives a user options for conditions such as thrust and drag profiles, rocket design, and launch conditions. Along with the ability to edit various scripts within the program for further customization. Given the time and patience, the program could be made fully customizable by any and all users who end up utilizing it.

If future development were to take place, there are a few things that could be done to improve or add to this program. The first being additional methods for calculating thrust and drag profiles. While the options presented work well, being able to determine the values of thrust and drag in flight without the use of outside data would be beneficial. Additionally, a more sophisticated deflection program could help in predicting the deflection of various materials and sizes of launch rods. Integrating the program with a higher order solver such as OpenRocket might provide for more accurate results as well. Finally, the biggest addition that could be made, would be to integrate a recovery program, such as the one developed by the MSU Space Cowboys to track the landing patterns of rockets such as the one in Ref. 15., would go a long way in rounding out the overall simulation.

Finally, it should be noted that this software can be used by anyone with some knowledge of programing in MATLAB. Additionally, the program was created for academic purposes and exists to be used by students and instructors alike to learn and teach the material and methods used within it. Evaluating the results in compression to the reference case and field-testing results, the program allows for an accurate flight simulation even with the downside of the simplicity in its methods. With all development goals met, and the program being as accurate as it is, the project was a success.

REFERENCES

1Anderson, C. F., and Henson, J. R., “Aerodynamic Characteristics of Several Bluff Bodies of Revolution at Mach numbers from 0.6 to 1.5,” Propulsion Wind Tunnel Facility, Arnold Engineering Development Center, Air Force Systems Command, Arnold Air Force Station, Tn., AEDC-TR-71-130, July 1971.

2Babb, C. Donald, and Fuller, Dennis E., “Static Stability Investigation of a Sounding Rocket Vehicle at Mach Numbers from 1.5 to 4.63,” Langley Research Center, Langley Station, Hampton, Va., NASA TN D-4014, June 1967.

3Briggs, Nicholas, “Multistage Supersonic Rocket Flight Simulator,” Mississippi State University, Starkville, Mississippi, May 2016.

4“Cambridge Rocketry Simulator.” *CambridgeRocketry:Home*, Cambridge University, cambridgerocket.sourceforge.net/.

5Ferris, James C., “Static Stability Investigation of a Single-Stage Sounding Rocket at Mach Numbers from 0.60 to 1.20,” Langley Research Center, Langley Station, Hampton, Va., NASA TN D-4013, June 1967.

6Fournier, Roger H., Hinson, William F,, and Langhans, Richard A.,” Aerodynamic Characteristics In Pitch of a 1/7-Scale Model of a Two- and Three-Stage Rocket Configuration at Mach Numbers of 0.4 to 4.63,” Langley Research Center, Langley Station, Hampton, Va., NASA TN D-5378, August 1969.

7Fuller, Dennis E., “Effects of Nose Shape and Fin Geometry on Static Stability of a High-Fineness-Ratio Sounding Rocket,” Langley Research Center, Langley Station, Hampton, Va., NASA TM X-1661, October 1968.

8Hibbeler, R. C. *Engineering Mechanics: Dynamics*. 12th ed., Prentice Hall, 2010.

9Hibbeler, R. C. *Engineering Mechanics: Statics*. 12th ed., Prentice Hall, 2010.

10Jernell, Lloyd S., “Investigation of the Static Longitudinal and Lateral Stability Characteristics of a 0.10-Scale Model of a Three-Stage Configuration of the Scout Research Vehicle at Mach Numbers of 2.29, 2.96, 3.96, and 4.65,” Langley Research Center, Langley Field, Va., NASA TN D-711, March 1961.

11Jernell, Lloyd S., and Wong, Norman, “Investigation of the Static Longitudinal Stability Characteristics of a 0.067-Scale Model of a Four-Stage Configuration of the Scout Research Vehicle at Mach Numbers of 2.29, 2.96, 3.96, and 4.65,” Langley Research Center, Langley Field, Va., NASA TN D-554, September 1960.

12Kelly, Thomas C.,” Transonic Wind-Tunnel Investigation of the Static Longitudinal Aerodynamic Characteristics of Several Configurations of the Scout Vehicle and of a Number of Related Models,” Langley Research Center, Langley Station, Hampton, Va., NASA TN D-794, May 1961.

13Kelly, Thomas C., and Keyton, Robert, J., “Transonic Wind-Tunnel Investigation of the Static Aerodynamic Characteristics of Several Configurations of the Blue Scout Launch Vehicle,” Langley Research Center, Langley Station, Hampton, Va., NASA TN D-1958, September 1963.

14MSU Space Cowboys, “Enterprise Launch Vehicle Technical Write-up,” Mississippi State University, Starkville, Ms., April 2017.

15MSU Space Cowboys, “Harmless Launch Vehicle Family Technical Write-up,” Mississippi State University, Starkville, Ms., March 2018.

17Niskanen, Sampo, “Development of an Open Source model rocket simulation software,” Helsinki University of Technology, Espoo, Finland, May 2009.

18Rogers, Charles E., “Rodgers Aeroscience RASAero II Aerodynamic Analysis and Flight Simulation Program – User’s Manual,” Rogers Aeroscience, Lancaster, Ca., 2016.

19“U.S. Standard Atmosphere, 1976,” Washington, D.C., NASA TM X-74335, October 1976



MATLAB SCRIPTS

* 1. Startup Page and Menus

The MATLAB scripts used as the main file and menus of the program

* + 1. Simulator Primary File

The only file that the user sees when running the program, contains a welcome display and prompt to either continue or exit.

clc;clear all  
fprintf('\*\*\*MULTISTAGE 2DOF ROCKET FLIGHT PROGRAM\*\*\*');  
fprintf('\n\n Welcome to the Multistage 2DOF rocket trajectory solver.');  
fprintf('\n Before continuing there are some things you should know.');  
fprintf('\n The solver opperates using the metric system, so please keep.');  
fprintf('\n units and changes consistent with the default units. Also make');  
fprintf('\n sure to have all text files and other readable files located ');  
fprintf('\n within the same file as where you have this program stored.');  
fprintf('\n\n For more in depth detail of this program, please see the');  
fprintf('\n technical document that was within this zip file entitled')  
fprintf('\n "User Manual".');  
  
fprintf('\n\n To begin, just hit "Continue" on the startup menu.');  
  
fprintf('\n\n Developed by: Zachery Doucet');  
fprintf('\n Date of last modificaton: 4/6/18');  
  
startup = questdlg('Please read Command Window before proceeding',...  
 'Startup Menu','Continue','Cancel','Cancel');  
switch startup  
 case 'Continue'  
 fprintf('\n\n Opening main menu, please wait...');  
 pause(5)  
 Main\_Menu()  
 case 'Cancel'  
 fprintf('\n\n Program cancelled. Resetting Command Window...')  
 pause(5)  
 clc  
end

* + 1. Main Menu

The main menu where the user is given the option to go to the parameter’s menu or the program menu.

function Main\_Menu()

%% Master Control Panel  
clc;  
disp(sprintf('\n\n\*\*\*MULTISTAGE 2DOF ROCKET FLIGHT PROGRAM\*\*\*'));  
disp(sprintf('Developed By: Zachery Doucet'));  
disp(sprintf('\n\nMAIN MENU'));  
disp(sprintf('What would you like to do?'));  
disp(sprintf('1. EDIT ROCKET PARAMETERS'));  
disp(sprintf('2. GO TO PROGRAM SELECTION'));  
disp(sprintf('3. EXIT'));  
  
stop = false;  
while stop == false  
 disp(sprintf('\nPlease enter the number of an option from the main menu and press return'));  
  
 MMs = input(':>','s');  
 MM = str2num(MMs);  
 MM = uint8(MM);  
  
 if (~isempty(MM) && MM>=1 && MM<=3)  
 stop = true;  
 else  
 disp(sprintf(['!! Sorry ',MMs,' is not a recognised menu item please try again']));  
 end  
end  
  
if MM==1  
 Parameters\_Menu();  
elseif MM==2  
 Program\_Menu();  
elseif MM==3  
 fprintf('\n\n Exiting Program. Resetting Command Window...')  
 pause(5)  
 clc;  
else  
 disp(sprintf('Something went wrong with the main menu selection'));  
end

end

* + 1. Parameters Menu

The parameters menu allows for the user to open and edit certain scripts that contain initial data, such as rocket and motor parameters and flight conditions.

function Parameters\_Menu()

%% Parameters Control Panel  
clc;  
disp(sprintf('\n\n\*\*\*MULTISTAGE 2DOF ROCKET FLIGHT PROGRAM\*\*\*'));  
disp(sprintf('Developed By: Zachery Doucet'));  
disp(sprintf('\n\nPARAMETERS MENU'));  
disp(sprintf('Which set of parameters would you like to modify?'));  
disp(sprintf('1. MODIFY THRUST DATA'));  
disp(sprintf('2. MODIFY DRAG DATA'));  
disp(sprintf('3. MODIFY ROCKET PARAMETERS'));  
disp(sprintf('4. MODIFY LAUNCH PARAMETERS'));  
disp(sprintf('5. MODIFY DEFLECTION PARAMETERS'));  
disp(sprintf('6. RETURN TO MAIN MENU'));  
  
  
stop = false;  
while stop == false  
 disp(sprintf('\nPlease enter the number of an option from the main menu and press return'));  
  
 MMs = input(':>','s');  
 MM = str2num(MMs);  
 MM = uint8(MM);  
  
 if (~isempty(MM) && MM>=1 && MM<=6)  
 stop = true;  
 else  
 disp(sprintf(['!! Sorry ',MMs,' is not a recognised menu item please try again']));  
 end  
end  
  
if MM==1  
 open Thrust\_Selection.m;  
elseif MM==2  
 open Drag\_Selection.m;  
elseif MM==3  
 open RocketandMotor\_Parameters.m;  
elseif MM==4  
 open Initial\_Launch\_Conditions.m;  
elseif MM==5  
 open MaxDefl.m  
elseif MM==6  
 Main\_Menu();  
else  
 disp(sprintf('Something went wrong with the main menu selection'));  
end

end

* + 1. Program Menu

The script that allows the user to either run one of the solution scripts or edit them.

function Program\_Menu()

%% Program Control Panel  
clc;  
disp(sprintf('\n\n\*\*\*MULTISTAGE 2DOF ROCKET FLIGHT PROGRAM\*\*\*'));  
disp(sprintf('Developed By: Zachery Doucet'));  
disp(sprintf('\n\nPROGRAM MENU'));  
disp(sprintf('Which program would you like to run?'));  
disp(sprintf('1. X-Y'));  
disp(sprintf('2. X-Y ODE'));  
disp(sprintf('3. PARALLEL-PERPENDICULAR'));  
disp(sprintf('4. PARALLEL-PERPENDICULAR ODE'));  
disp(sprintf('5. OPEN EDITOR MENU'));  
disp(sprintf('6. RETURN TO MAIN MENU'));  
  
  
stop = false;  
while stop == false  
 disp(sprintf('\nPlease enter the number of an option from the main menu and press return'));  
  
 MMs = input(':>','s');  
 MM = str2num(MMs);  
 MM = uint8(MM);  
  
 if (~isempty(MM) && MM>=1 && MM<=6)  
 stop = true;  
 else  
 disp(sprintf(['!! Sorry ',MMs,' is not a recognised menu item please try again']));  
 end  
end  
  
if MM==1  
 Angle\_Tests\_XY();  
elseif MM==2  
 ODE\_Program\_XY();  
elseif MM==3  
 Angle\_Tests\_PP();  
elseif MM==4  
 ODE\_Program\_PP();  
elseif MM==5  
 Program\_Editor();  
elseif MM==6  
 Main\_Menu();  
else  
 disp(sprintf('Something went wrong with the main menu selection'));  
end

end

* + 1. Program Editor Menu

The menu that allows the user to edit the solution scripts.

function Program\_Editor()

%% Editor Control Panel  
clc;  
disp(sprintf('\n\n\*\*\*MULTISTAGE 2DOF ROCKET FLIGHT PROGRAM\*\*\*'));  
disp(sprintf('Developed By: Zachery Doucet'));  
disp(sprintf('\n\nEDITOR MENU'));  
disp(sprintf('Which program would you like to modify?'));  
disp(sprintf('1. X-Y'));  
disp(sprintf('2. X-Y ODE'));  
disp(sprintf('3. PARALLEL-PERPENDICULAR'));  
disp(sprintf('4. PARALLEL-PERPENDICULAR ODE'));  
disp(sprintf('5. RETURN TO PROGRAM MENU'));  
disp(sprintf('6. RETURN TO MAIN MENU'));  
  
stop = false;  
while stop == false  
 disp(sprintf('\nPlease enter the number of an option from the main menu and press return'));  
  
 MMs = input(':>','s');  
 MM = str2num(MMs);  
 MM = uint8(MM);  
  
 if (~isempty(MM) && MM>=1 && MM<=6)  
 stop = true;  
 else  
 disp(sprintf(['!! Sorry ',MMs,' is not a recognised menu item please try again']));  
 end  
end  
  
if MM==1  
 open Angle\_Tests\_XY.m;  
elseif MM==2  
 open ODE\_Program\_XY.m;  
elseif MM==3  
 open Angle\_Tests\_PP.m;  
elseif MM==4  
 open ODE\_Program\_PP.m;  
elseif MM==5  
 Program\_Menu();  
elseif MM==6  
 Main\_Menu();  
else  
 disp(sprintf('Something went wrong with the main menu selection'));  
end

end

* 1. Initialization Scripts

The MATLAB scripts that handle loading of the initial conditions.Hidden text to allow template to find last page in document

* + 1. Initial Conditions Compiler

%% Initial Conditions Compiler  
global DragData ThrustData TCondition DCondition mend  
RocketandMotor\_Parameters  
Initial\_Launch\_Conditions  
[TCondition, ThrustData] = Thrust\_Selection();  
[DCondition, DragData, mend] = Drag\_Selection();

* + 1. Rocket and Motor Parameters

The script that loads the rocket and motor data based on user selection. The script contains preloaded options for one and two stage rockets, as well as the ability to utilize a custom input for adding new parameters.

%% INITIAL LAUNCH VEHICLE/MOTOR DATA  
global mi mf mdot tbo twait tb nos A Ls Lr Tstage tadd total\_burn\_time mt Rr  
prompt = 'How many stages? (1-4) '; % Number of Stages  
nos = input(prompt);  
if nos ==1  
 prompt1 = 'Enterprise(1) IREC(2) Intro Rocket(3) Model Rocket(4)? ';  
 nos1 = input(prompt1);  
 if nos1 ==1  
 % Enterprise  
 Df = 0.0578; % Diamter of fuselage (meters)  
 Ls = 1.47; % Length of structure (meters)  
 Lr = 2; % Length of launch rod (meters)  
 ms = 2.441; % Mass of structure (kg)  
 mp = 1.4; % Mass of propellant (kg)  
 Tstage = 1422; % Stage thrust (N)  
 tb = 2; % burn time (sec)  
 elseif nos1 ==2  
 % IREC  
 Df = 0.132;  
 Ls = 2.77;  
 Lr = 3;  
 ms = 17.592;  
 mp = 9.425;  
 Tstage = 5772.1;  
 tb = 3.95;  
 elseif nos1 == 3  
 % Intro Rocket  
 Df = 0.045;  
 Ls = 0.33655;  
 Lr = 0.758825;  
 Rr = 0.001651;  
 ms = 0.1142;  
 mp = 0.0108;  
 Tstage = 4.7;  
 tb = 1.86;  
 elseif nos1 == 4  
 % Intro Rocket 2  
 Df = 0.018;  
 Ls = 0.2921;  
 Lr = 0.7493;  
 Rr = 0.001651;  
 ms = 0.0692;  
 mp = 0.0108;  
 Tstage = 4.7;  
 tb = 1.86;  
 else  
 [Df,Ls,Lr,ms,mp,Tstage,tb] = Custom\_Input();  
 end  
elseif nos ==2  
 prompt2 = 'Mostly Harmless(1) or JoT&AT(2)? ';  
 nos2 = input(prompt2);  
 if nos2 == 1  
 % MH  
 Df = [0.108;0.108];  
 Ls = 4.3;  
 Lr = 5;  
 ms = [8.362;11.013];  
 mp = [4.452; 4.349];  
 Tstage = [3421.1; 2021.9];  
 tb = [2.99; 4.301];  
 elseif nos2 ==2  
 % Thing  
 Df = [0.157;0.108];  
 Ls = 4.88;  
 Lr = 5;  
 ms = [9.551;10.429];  
 mp = [3.950; 5.309];  
 Tstage = [2587; 2356];  
 tb = [2.14; 4.371];  
 else  
 [Df,Ls,Lr,ms,mp,Tstage,tb] = Custom\_Input();  
 end  
else  
[Df,Ls,Lr,ms,mp,Tstage,tb] = Custom\_Input();  
end

A = (pi/4)\*(Df.^2); % Cross-Sectional Area of Stages - m^2  
tadd = [3;3;6;6]; % Wait Time - s  
twait = 3;

%% CALCULATION OF STAGE MASSES, FLOW RATES, AND BURNOUT TIMES  
for i = 1:nos  
 mt(i) = mp(i) + ms(i);  
 mi(i,1) = 0;  
 for j = i:nos  
 mi(i,1) = mi(i,1) + ms(j,1) + mp(j,1);  
 end  
 tbo(i,1) = 0;  
 for k = 1:i  
 tbo(i,1) = tbo(i,1) + tb(k,1) + tadd(k,1);  
 end  
end  
  
if nos == 1  
 total\_burn\_time = tb(1);  
elseif nos == 2  
 total\_burn\_time = tbo(1) + tb(2);  
elseif nos == 3  
 total\_burn\_time = tbo(2) + tb(3);  
elseif nos == 4  
 total\_burn\_time = tbo(3) + tb(4);  
end  
  
mt = sum(mt) % Total Mass (kg)  
mf = mi - mp; % End of Stage Mass (kg)  
mdot = mp./tb; % Mass Flow Rate - kg/s

* + - 1. Custom Rocket and Motor Data Input

The script that loads if a preloaded rocket model isn’t selected, prompting the user to enter the dimensions and values for certain variables for the rocket they want to test.

function [Df,Ls,Lr,Rr,ms,mp,Tstage,tb] = Custom\_Input  
%% Custom Rocket Input

% Diameter of Stages - m increased diameter for structure around motors   
Dfstr = inputdlg('Enter diameter of each Stage (separated by semicolon):',...  
 'Diameter (meters) ', [1 50]);  
  
% Length of Structure - m  
Lsstr = inputdlg('Enter total length of the structure',...  
 'Structure Length (meters)',[1 60]);  
Ls = str2num(Lsstr{1});  
  
% Length of Launch Rod - m  
Lrstr = inputdlg('Enter length of launch rod','Launch Rod Length (meters)'...  
 ,[1 60]);  
Lr = str2num(Lrstr{1});  
  
% Radius of Launch Rod - m  
Rrstr = inputdlg('Enter radius of launch rod','Launch Rod Radius (meters)'...  
 ,[1 60]);  
Lr = str2num(Lrstr{1});  
  
% Mass of Stages - kg  
msstr = inputdlg('Enter structural mass of each motor (separated by semicolon):',...  
 'Mass of Structure (kilograms)', [1 60]);  
ms = str2num(msstr{1});  
  
% Mass of Stage Propellants - kg  
mpstr = inputdlg('Enter propellant mass of each motor (separated by semicolon):',...  
 'Mass of Propellant (kilograms)', [1 60]);  
mp = str2num(mpstr{1});  
  
% Thrust of Stages - N  
Tstr = inputdlg('Enter Avg thrust of each motor (separated by semicolon):',...  
 'Average Thrust (Newtons)', [1 60]);  
Tstage = str2num(Tstr{1});  
  
% Stage Burn Time - s  
tbstr = inputdlg('Enter burn time of each motor(separated by semicolon):',...  
 'Burn Time (Seconds)', [1 50]);  
tb = str2num(tbstr{1});

end

* + 1. Initial Launch Conditions

The script that loads the initial conditions for the key variables that are used in the calculation of the rocket’s flight path.

%% INITIAL CONDITIONS  
global go RE TimeStep Rg  
Rg = 286.9; % Gas Constant - J/kg\*K  
go = 9.80665; % Sea Level Gravity - m/s^2  
RE = 6376000; % Radius of the Earth - m

%% INITIAL FLIGHT CONDITIONS  
% Time Step  
TimeStep = input('What is the Time Step? ');  
  
% Angle of Launch  
ask = input('Flight Angle? (deg from horizontal) ');  
theta\_i = ask; % Angle of Flight - deg  
Disp = questdlg('Use Launch Rod Deflection?','Deflection Menu',...  
 'Yes','No','Yes');  
switch Disp  
 case 'Yes'  
 disp(['Launch Rod Deflection Accounted For.']);  
 theta\_r = DeflectionCalc(theta\_i);  
 case 'No'  
 disp(['Launch Rod Deflection Not Accounted For.']);  
 theta\_r = deg2rad(theta\_i);  
end  
theta(1) = theta\_r;  
  
% Initial Values  
V(1) = 0.447; % Velocity - m/s  
u(1) = V(1)\*cos(theta(1)); % x-component of velocity  
v(1) = V(1)\*sin(theta(1)); % y-component of velocity  
a(1) = 0; % Acceleration - m/s^2  
y(1) = 0; % Altitude - m (XY)  
z(1) = 0; % Altitude - m (PP)  
x(1) = 0; % Range - m  
t(1) = 0; % Flight Time - s  
T(1) = 288.16; % Atmospheric Temperature - K  
M(1) = 0; % Mach Number  
c(1) = 340.3961; % Speed of Sound - m/s^2  
rhoe(1) = 1.23; % Sea Level Density - kg/m^3  
  
% Calls functions for initial values  
m(1) = Mass(t(1));  
g(1) = Gravity(z(1));  
area(1) = Area(t(1));

* + - 1. Deflection Scripts

The scripts that are used to determine the deflection of the launch rod.

* + - * 1. Deflection Screener

The script that determines if the launch rod is at a position to deflect, if the rod is at perfect horizontal or vertical position, it will send back the same angle value that was input.

function [theta\_rad] = DeflectionCalc(theta)  
% NOTE: This deflection analysis is done with a specific stainless steel  
% rod, other launch rods will have different deflection values  
if theta == 90  
 theta\_new = 90;  
elseif theta <= 89 && theta >= 1  
 theta\_new = MaxDefl(theta);  
elseif theta == 0  
 theta\_new = 0;  
else  
 error('Angle must be between 0-90 deg');  
end  
theta\_rad = deg2rad(theta\_new);

* + - * 1. Deflection Calculation

The script that determines the angle of deflection and new launch angle for the launch.

function [angle] = MaxDefl(theta)  
global mt Lr Rr go  
  
% Calculate Deflection Angle  
I = 0.25\*pi\*(Rr)^4;  
E = 1.93\*10^11; % Modulus of Elasticity of Stainless Steel (Pa)  
P = cosd(theta)\*mt\*go;  
DefAngle = (P\*Lr^2)/(2\*E\*I);  
angle = theta-rad2deg(DefAngle);  
  
% Negative Angle Failsafe  
if angle < 0  
 error('Deflection Causes a Negative Angle');  
end  
end

* + 1. Thrust Selection

The script that allows the user to decide which method to use to load the drag profile.

function [TCondition,ThrustData] = Thrust\_Selection  
answer = questdlg('What Thrust values do you wish to use?', ...  
 'Thrust Profile Menu', ...  
 'Interpolated Thrust Values','Average Thrust Values','Motor Data',...  
 'Interpolated Thrust Values');  
% Handle response  
switch answer  
 case 'Interpolated Thrust Values'  
 disp([answer ' Selected.'])  
 TCondition = 1;  
 case 'Average Thrust Values'  
 disp([answer ' Selected.'])  
 TCondition = 2;  
 case 'Motor Data'  
 disp([answer ' Selected.'])  
 TCondition = 3;  
end  
if TCondition == 1  
 filter = {'\*.txt'};  
 [filenameT] = uigetfile(filter);  
 ThrustData = dlmread(filenameT);  
elseif TCondition == 3  
 [ThrustData] = ThrustCalc();  
else  
 ThrustData = 0;  
end  
end

* + - 1. Thrust Data Interpolation

The script that handles the reading in and prepping of the text file containing the motor data for interpolation.

function [thrust] = ThrustCalc()  
global TimeStep tb nos tadd tbo  
filter = {'\*.txt'};  
[file] = uigetfile(filter);  
data = dlmread(file);  
f = data(:,2);  
t = data(:,1);  
n = length(f)-1;  
dt = TimeStep;  
b = tb(1)/dt;  
  
for i = 1:n  
 c1(i) = (f(i+1) - f(i)) / (t(i+1) - t(i));  
 c0(i) = f(i) - c1(i)\*t(i);  
 C(i,1) = c0(i);  
 C(i,2) = c1(i);  
end  
  
for j = 1:b  
 time(j) = (j-1)\*dt;  
 Force(j) = Interp(C,t,time(j));  
end  
  
if nos >= 2  
 filter = {'\*.txt'};  
 [file2] = uigetfile(filter);  
 data2 = dlmread(file2);  
 f2 = data2(:,2);  
 t2 = data2(:,1);  
 n2 = length(f2)-1;  
 b2 = tb(2)/dt;  
  
 for i = 1:n2  
 d1(i) = (f2(i+1) - f2(i)) / (t2(i+1) - t2(i));  
 d0(i) = f2(i) - d1(i)\*t2(i);  
 D(i,1) = d0(i);  
 D(i,2) = d1(i);  
 end  
  
 for j = 1:b2  
 time2(j) = (j-1)\*dt;  
 Force2(j) = Interp(D,t2,time2(j));  
 end  
 tfill = tadd(1)/dt;  
 thfill = zeros(1,tfill);  
end  
  
if nos >= 3  
 filter = {'\*.txt'};  
 [file3] = uigetfile(filter);  
 data3 = dlmread(file3);  
 f3 = data3(:,2);  
 t3 = data3(:,1);  
 n3 = length(f3)-1;  
 b3 = tb(3)/dt;  
  
 for i = 1:n3  
 e1(i) = (f3(i+1) - f3(i)) / (t3(i+1) - t3(i));  
 e0(i) = f3(i) - e1(i)\*t3(i);  
 E(i,1) = e0(i);  
 E(i,2) = e1(i);  
 end  
  
 for j = 1:b3  
 time3(j) = (j-1)\*dt;  
 Force3(j) = Interp(E,t3,time3(j));  
 end  
 tfill2 = tadd(3)/dt;  
 thfill2 = zeros(1,tfill2);  
end  
  
if nos >= 4  
 filter = {'\*.txt'};  
 [file4] = uigetfile(filter);  
 data4 = dlmread(file4);  
 f4 = data4(:,2);  
 t4 = data4(:,1);  
 n4 = length(f4)-1;  
 b4 = tb(4)/dt;  
  
 for i = 1:n4  
 g1(i) = (f4(i+1) - f4(i)) / (t4(i+1) - t4(i));  
 g0(i) = f4(i) - g1(i)\*t4(i);  
 G(i,1) = g0(i);  
 G(i,2) = g1(i);  
 end  
  
 for j = 1:b4  
 time4(j) = (j-1)\*dt;  
 Force4(j) = Interp(G,t4,time4(j));  
 end  
 tfill3 = tadd(4)/dt;  
 thfill3 = zeros(1,tfill3);  
end  
  
if nos == 1  
 thrust = Force;  
elseif nos == 2  
 thrust = horzcat(Force,thfill,Force2);  
elseif nos == 3  
 thrust = horzcat(Force,thfill,Force2,thfill2,Force3);  
elseif nos == 4  
 thrust = horzcat(Force,thfill,Force2,thfill2,Force3,thfill3,Force4);  
end

* + - 1. Interpolation Function

The script that handles the linear interpolation of the motor data, also used in the drag profiles below.

function [F] = Interp(C, t, tau)  
  
j = 1;  
while tau >= t(j)  
 j = j + 1;  
end  
k = j - 1;  
F = C(k,1) + C(k,2) \* tau;  
end

* + 1. Drag Selection

The script that allows the user to decide which method to load the drag profile.

function [DCondition,DragData,mend] = Drag\_Selection  
answer2 = questdlg('What Drag values do you wish to use?', ...  
 'Drag Profile Menu', ...  
 'Actual Drag Data','Fin Type','Constant Drag','Actual Drag Data');  
% Handle response  
switch answer2  
 case 'Actual Drag Data'  
 disp([answer2 ' Selected.'])  
 DCondition = 1;  
 case 'Fin Type'  
 disp([answer2 ' Selected.'])  
 DCondition = 2;  
answer3 = questdlg('What type of model would you like to run?', ...  
 'Fin Type Menu', ...  
 'Fins Model','No Fins Model','Fins Model');  
 switch answer3  
 case 'Fins Model'  
 disp([answer3 ' Selected.'])  
answer4 = questdlg('What type of fins do you want to use?', ...  
 'Fins Model Menu', ...  
 'Small Fins Model','Large Fins Model','Delta Fins Model',...  
 'Small Fins Model');  
 switch answer4  
 case 'Small Fins Model'  
 disp([answer4 ' Selected.'])  
 FinType = 1;  
 case 'Large Fins Model'  
 disp([answer4 ' Selected.'])  
 FinType = 2;  
 case 'Delta Fins Model'  
 disp([answer4 ' Selected.'])  
 FinType = 3;  
 end  
 case 'No Fins Model'  
 disp([answer3 ' Selected.'])  
 FinType = 4;  
warn = warndlg('Answer will be given, but stability is not accounted for',...  
 'Warning');  
pause(5)  
 end  
 case 'Constant Drag'  
 disp([answer2 ' Selected.'])  
 DCondition = 3;  
end  
if DCondition ==1  
 filter = {'\*.txt'};  
 [filenameD] = uigetfile(filter);  
 DragData = dlmread(filenameD);  
 mend = 0;  
elseif DCondition == 2  
 if FinType == 4  
 [DragData,mend] = NoFins();  
 elseif FinType == 1  
 [DragData,mend] = SmallFins();  
 elseif FinType == 2  
 [DragData,mend] = LargeFins();  
 elseif FinType == 3  
 [DragData,mend] = DeltaFins();  
 end  
elseif DCondition == 3  
 DragData = 0;  
 mend = 0;  
end  
end

* + - 1. Drag Profiles

The four different drag profiles that can be selected by the user if they do not have a drag profile to read in or want to use a constant drag value.

* + - * 1. No Fins Drag Model

The no fins model recommended to be used for only initial development stages.

function [Cd,mend] = NoFins()  
global TimeStep  
% General No Fins Design  
% Drag Coefficient (y-axis)  
cdy = [0.33;0.3; 0.38; 0.42; 0.57; 0.66; 0.44; 0.34; 0.27; 0.24];  
% Mach (x-axis)  
machx = [0;0.6; 0.8; 0.9; 1; 1.03; 2.29; 2.96; 3.96; 4.65];  
  
% Set up for linear interpolation  
f = cdy;  
t = machx;  
n = length(cdy)-1;  
dt = TimeStep;  
b = machx(end)/dt;  
  
for i = 1:n  
 c1(i) = (f(i+1) - f(i)) / (t(i+1) - t(i));  
 c0(i) = f(i) - c1(i)\*t(i);  
 C(i,1) = c0(i);  
 C(i,2) = c1(i);  
end  
  
for j = 1:b  
 speed(j) = (j-1)\*dt;  
 coeff(j) = Interp(C,t,speed(j));  
end  
mend = machx(end);  
Cd = coeff(:);  
end

* + - * 1. Small Fins Model

The model for small fins, recommended for use with single stage or smaller multistate rockets.

function [Cd,mend] = SmallFins()  
global TimeStep  
% General Small Fins Design  
% Drag Coefficient (y-axis)  
cdy = [0.55;0.5; 0.59; 0.7; 1.15; 1.15; 0.55; 0.44; 0.34; 0.31];  
% Mach (x-axis)  
machx = [0;0.6; 0.8; 0.9; 1; 1.03; 2.29; 2.96; 3.96; 4.65];  
  
% Set up for linear interpolation  
f = cdy;  
t = machx;  
n = length(cdy)-1;  
dt = TimeStep;  
b = machx(end)/dt;  
  
for i = 1:n  
 c1(i) = (f(i+1) - f(i)) / (t(i+1) - t(i));  
 c0(i) = f(i) - c1(i)\*t(i);  
 C(i,1) = c0(i);  
 C(i,2) = c1(i);  
end  
  
for j = 1:b  
 speed(j) = (j-1)\*dt;  
 coeff(j) = Interp(C,t,speed(j));  
end  
mend = machx(end);  
Cd = coeff(:);  
end

* + - * 1. Large Fins Model

The large fins model, recommended for use with multistage rockets, or substantially long single stage rockets.

function [Cd,mend] = LargeFins()  
global TimeStep  
% General Large Fins Design  
% Drag Coefficient (y-axis)  
cdy = [0.5;0.47;0.58;0.6;0.75;0.8;1.2;1.37;0.62;0.75;0.68;0.47;0.36;0.32];  
% Mach (x-axis)  
machx = [0;0.4;0.6;0.8;0.9;0.95;1;2;2.29;2.36;2.86;2.96;3.96;4.65];  
% Set up for linear interpolation  
f = cdy;  
t = machx;  
n = length(cdy)-1;  
dt = TimeStep;  
b = machx(end)/dt;  
  
for i = 1:n  
 c1(i) = (f(i+1) - f(i)) / (t(i+1) - t(i));  
 c0(i) = f(i) - c1(i)\*t(i);  
 C(i,1) = c0(i);  
 C(i,2) = c1(i);  
end  
  
for j = 1:b  
 speed(j) = (j-1)\*dt;  
 coeff(j) = Interp(C,t,speed(j));  
end  
mend = machx(end);  
Cd = coeff(:);  
end

* + - * 1. Delta Fins Model

The delta fins model, only to be used with rockets using full delta planform fins.

function [Cd,mend] = DeltaFins()  
global TimeStep  
% General Delta Fins Design  
% Drag Coefficient (y-axis)  
cdy = [0.47;0.45;0.46;0.48;0.5;0.53;0.62;1.06;0.99;0.95;0.83;0.68;0.57;0.53];  
% Mach (x-axis)  
machx = [0;0.4;0.5; 0.6; 0.7; 0.8; 0.9; 1; 1.1; 1.2; 2.29; 2.96; 3.96;4.65];  
  
% Set up for linear interpolation  
f = cdy;  
t = machx;  
n = length(cdy)-1;  
dt = TimeStep;  
b = machx(end)/dt;  
  
for i = 1:n  
 c1(i) = (f(i+1) - f(i)) / (t(i+1) - t(i));  
 c0(i) = f(i) - c1(i)\*t(i);  
 C(i,1) = c0(i);  
 C(i,2) = c1(i);  
end  
  
for j = 1:b  
 speed(j) = (j-1)\*dt;  
 coeff(j) = Interp(C,t,speed(j));  
end  
mend = machx(end);  
Cd = coeff(:);  
end

* 1. Solution Scripts

The scripts that perform the desired solution method of choice, being either the Euler or 4th order Runge-Kutta solution method for the x-y or parallel-perpendicular coordinate frame.

* + 1. Euler Methods
       1. Euler X-Y Solution

clc;clear all;

%% Call Initial Conditions  
Initial\_Conditions\_Compiler

%% Simulation Calculations  
if nos == 1  
 dt = TimeStep; % Time Differential (Sampling Rate - Hz)  
 n = tbo/dt; % Time of Flight  
else  
 dt = TimeStep;  
 n = tbo(nos)/dt;  
end  
  
if DCondition == 3  
 Co\_Drag = input('Enter Constant Drag Value. ');  
end  
  
for i = 2:n\*100  
 % Calculate Drag Coefficient for current Mach  
 if DCondition == 1  
 Cd(i) = DragData(i);  
 elseif DCondition == 2  
 Cd(i) = DragCoefficient(M(i-1),mend);  
 elseif DCondition == 3  
 Cd(i) = Co\_Drag;  
 end  
  
 t(i) = t(i-1) + dt;  
 if (TCondition == 1 || TCondition == 3)  
 if t(i) < total\_burn\_time  
 thrust(i) = ThrustData(i-1);  
 else  
 thrust(i) = 0;  
 end  
 else  
 thrust(i) = Thrust(t(i-1));  
 end  
  
 m(i) = Mass(t(i));  
 Vcr(i) = sqrt((thrust(i)/m(i))\*Ls\*2);  
  
 % X Y Velocity Component Equations  
 kd(i) = (rhoe(i-1)\*Cd(i)\*area(i-1))/(2\*m(i));  
 kt(i) = thrust(i)/m(i);  
 V(i) = sqrt(u(i-1)^2 + v(i-1)^2);  
 DvDt(i) = kt(i)\*v(i-1)/V(i) - kd(i)\*v(i-1)\*V(i) - g(i-1);  
 if DvDt(i) < 0 && t(i-1) <= tb(1)  
 dvdt(i) = 0;  
 else  
 dvdt(i) = DvDt(i);  
 end  
 v(i) = v(i-1)+dvdt(i)\*dt;  
  
 utheta(i) = v(i)\*u(i-1)/v(i-1);  
 ux(i) = u(i-1)+((kt(i)\*u(i-1)/V(i-1))-kd(i)\*u(i-1)\*V(i-1))\*dt;  
  
 if DvDt(i) < 0 && t(i-1) <= tb(1)  
 u(i) = u(i-1);  
 elseif x(i-1) <= Lr\*cos(theta(1))  
 u(i) = utheta(i);  
 elseif V(i) < Vcr(i)  
 u(i) = utheta(i);  
 else  
 u(i) = ux(i);  
 end  
  
 theta(i) = atan(v(i)/u(i));  
 x(i) = x(i-1) + u(i)\*dt;  
 y(i) = y(i-1) + v(i)\*dt;  
 q(i) = 0.5\*rhoe(i-1)\*(V(i-1)^2);  
  
 % Calls Functions  
 g(i) = Gravity(y(i-1));  
 area(i) = Area(t(i-1));  
  
 % Atmospheric Profile  
 [T(i),P(i),rhoe(i)] = SIAtm(y(i));  
 c(i) = sqrt(1.4\*Rg\*T(i-1)); % Speed of Sound - m/s  
 M(i) = abs(V(i-1))/c(i-1); % Mach Number  
  
 if y(i) < 0  
 break  
 end  
end  
  
theta = theta\*(180/pi);  
y = y;  
x = x;  
q = q./1000;

%% PREDICTED PERFORMANCE  
Vx = max(V);  
ax = max(a);  
Mx = max(M);  
Yx = max(y);  
Xx = max(x);  
tx = max(t);  
Qx = max(q);  
mx = max(mi);  
  
[valx,idxx] = max(x);  
[valy,idxy] = max(y);  
tX = t(idxx);  
tY = t(idxy);  
  
disp(' ');  
formatSpec = 'Apogee is %6.3f m occuring at %5.3f seconds \n';  
fprintf(formatSpec,Yx,tY)  
formatSpec = 'Landing Distance is %6.3f m occuring at %5.3f seconds \n';  
fprintf(formatSpec,Xx,tX)  
  
figure(1)  
plot(x,y);  
hold on  
xlabel('Distance (m)');  
ylabel('Altitude (m)');  
grid on

* + - 1. Euler Parallel-Perpendicular

Modified version of script found in Ref. 3.

clc;clear all;

%% Call Initial Conditions  
Initial\_Conditions\_Compiler

%% Simulation Calculations  
if nos == 1  
 dt = TimeStep; % Time Differential (Sampling Rate - Hz)  
 n = tbo/dt; % Time of Flight  
else  
 dt = TimeStep;  
 n = tbo(nos)/dt;  
end  
theta(1) = (90\*pi/180)-theta(1);  
if DCondition == 3  
 Co\_Drag = input('Enter Constant Drag Value. ');  
end  
  
for i = 2:n\*100  
 % Calculate Drag Coefficient for current Mach  
 if DCondition == 1  
 Cd(i) = DragData(i);  
 elseif DCondition == 2  
 Cd(i) = DragCoefficient(M(i-1),mend);  
 elseif DCondition == 3  
 Cd(i) = Co\_Drag;  
 end  
  
 t(i) = t(i-1) + dt;  
 if (TCondition == 1 || TCondition == 3)  
 if t(i) < total\_burn\_time  
 thrust(i) = ThrustData(i-1);  
 else  
 thrust(i) = 0;  
 end  
 else  
 thrust(i) = Thrust(t(i-1));  
 end  
  
  
 m(i) = Mass(t(i));  
 Vcr(i) = sqrt((thrust(i)/m(i))\*Ls\*2);  
  
 % 1st Order Runga-Kutta or Euler's Equation  
 V(i) = (thrust(i)/m(i) - g(i-1)\*cos(theta(i-1))-...  
 (rhoe(i-1)\*Cd(i-1)\*area(i-1)/(2\*m(i)))\*V(i-1)^2)\*dt + V(i-1);  
  
 if V(i) < 0 && t(i) < tb(1)  
 V(i) = 0;  
 end  
  
 a(i) = (thrust(i)/m(i) - g(i-1)\*cos(theta(i-1))-...  
 (rhoe(i-1)\*Cd(i-1)\*area(i-1)/(2\*m(i)))\*V(i-1)^2) + (V(i)-V(i-1))/dt;  
  
 if V(i) <= 0 && t(i) < tb(1)  
 theta(i) = theta(i-1);  
 elseif x(i-1) <= Lr\*sin(theta(1))/1000  
 theta(i) = theta(i-1);  
 elseif V(i) <= Vcr(i)  
 theta(i) = theta(i-1);  
 else  
 theta(i) = (g(i-1)/V(i))\*sin(theta(i-1))\*dt + theta(i-1);  
 end  
  
 z(i) = V(i)\*cos(theta(i))\*dt + z(i-1);  
 q(i) = 0.5\*rhoe(i-1)\*(V(i-1)^2);  
 x(i) = V(i)\*sin(theta(i))\*dt + x(i-1);  
  
 % Components of V  
 V1(i) = thrust(i)/m(i);  
 V2(i) = g(i-1)\*cos(theta(i-1));  
 V3(i) = ((rhoe(i-1)\*Cd(i-1)\*area(i-1)/(2\*m(i)))\*V(i-1)^2);  
 Vtot(i) = V1(i)-V2(i)-V3(i);  
  
 % Calls Functions  
 g(i) = Gravity(z(i));  
 area(i) = Area(t(i));  
  
 % Atmospheric Profile  
 [T(i),P(i),rhoe(i)] = SIAtm(z(i));  
 c(i) = sqrt(1.4\*Rg\*T(i)); % Speed of Sound - m/s  
 M(i) = abs(V(i-1))/c(i); % Mach Number  
 if z(i) <= 0  
 break  
 end  
end  
  
a = a./g;  
theta = theta\*(180/pi);  
z = z;  
x = x;  
q = q./1000;

%% PREDICTED PERFORMANCE  
Vx = max(V);  
ax = max(a);  
Mx = max(M);  
Zx = max(z);  
Xx = max(x);  
tx = max(t);  
Qx = max(q);  
mx = max(mi);  
  
[valx,idxx] = max(x);  
[valz,idxz] = max(z);  
tX = t(idxx);  
tZ = t(idxz);  
  
disp(' ');  
formatSpec = 'Apogee is %6.3f m occuring at %5.3f seconds \n';  
fprintf(formatSpec,Zx,tZ)  
formatSpec = 'Landing Distance is %6.3f m occuring at %5.3f seconds \n';  
fprintf(formatSpec,Xx,tX)  
  
  
figure(1)  
plot(x,z);  
hold on  
xlabel('Distance (m)');  
ylabel('Altitude (m)');  
grid on

* + 1. 4th Order Runge-Kutta Methods
       1. 4th Order X-Y

clc; clear all  
global thetao dt Co\_Drag

%% Call Initial Conditions  
Initial\_Conditions\_Compiler

%% Set Addition Parameters   
dt = TimeStep;  
n = (tbo(end)\*100);  
tspan = [0:dt:n];  
thetao = theta(1);  
if DCondition == 3  
 Co\_Drag = input('Enter Constant Drag Value. ');  
end  
  
% Set Initial Values for ode function  
xo = 0;  
yo = 0;  
mo = m(1);  
uo = u(1);  
vo = v(1);  
y0 = [uo vo xo yo mo];  
  
% Set the limit for the ode function  
Opt = odeset('Events', @StopFunc);  
  
[t,y] = ode45(@(t,y) ode\_func\_x\_y(t,y),tspan,y0,Opt);

%% Extract Data from arrays  
u = y(:,1);  
v = y(:,2);  
xpos = y(:,3);  
ypos = y(:,4);  
mass = y(:,5);  
  
% Find Max X and Y positions  
[valx,idxx] = max(xpos);  
[valy,idxy] = max(ypos);  
Yx = valy;  
Xx = valx;  
tX = t(idxx);  
tY = t(idxy);  
disp(' ');  
  
formatSpec = 'Apogee is %6.3f m occuring at %5.3f seconds \n';  
fprintf(formatSpec,Yx,tY)  
formatSpec = 'Landing Distance is %6.3f m occuring at %5.3f seconds \n';  
fprintf(formatSpec,Xx,tX)  
  
figure(1)  
hold on  
plot(xpos,ypos)  
xlabel('Distance (m)');  
ylabel('Altitude (m)');  
grid on

* + - * 1. X-Y ODE Function

The sub-function that contains the bulk of equations used for the ODE solver

function dydt = ode\_func\_x\_y(t,y)  
global TCondition DCondition thetao Ls tb Lr Co\_Drag  
dydt = zeros(5,1);  
  
if (TCondition == 1 || TCondition == 3)  
 Vcr = sqrt((thrustode(t)/Mass(t))\*Ls\*2);  
 kt = thrustode(t)/Mass(t);  
else  
 Vcr = sqrt((Thrust(t)/Mass(t))\*Ls\*2);  
 kt = Thrust(t)/Mass(t);  
end  
  
Ve = sqrt(y(1)^2 + y(2)^2);  
  
% Determine kd  
if DCondition == 1  
 kd = (rhoe(y(4))\*Cdode(t,Ve)\*Area(t))/(2\*Mass(t));  
elseif DCondition == 2  
 kd = (rhoe(y(4))\*Cd(Ve,y(4))\*Area(t))/(2\*Mass(t));  
else  
 CD = Co\_Drag;  
 kd = (rhoe(y(4))\*CD\*Area(t))/(2\*Mass(t));  
end  
% Determine Diff Eqs  
DvDt = kt\*y(2)/Ve - kd\*y(2)\*Ve - Gravity(y(4));  
 if DvDt < 0 && t <= tb(1)  
 dvdt = 0;  
 else  
 dvdt = DvDt;  
 end  
dydt(2) = dvdt;  
  
dutdt = DvDt\*1/tan(thetao);  
duxdt = (kt\*y(1)/Ve)-kd\*y(1)\*Ve;  
 if DvDt < 0 && t <= tb(1)  
 dudt = 0;  
 elseif y(3) <= Lr\*cos(thetao)  
 dudt = dutdt;  
 elseif Ve <= Vcr  
 dudt = dutdt;  
 else  
 dudt = duxdt;  
 end  
dydt(1) = dudt;  
  
dydt(3) = y(1);  
dydt(4) = y(2);  
dydt(5) = -Mass(t);

* + - 1. 4th Order Parallel-Perpendicular

clc; clear all  
global thetao dt Co\_Drag

%% Call Initial Conditions  
Initial\_Conditions\_Compiler

%% Set Addition Variables  
dt = TimeStep;  
n = (tbo(end))\*100;  
tspan = [0:dt:n];  
thetao = theta(1);  
if DCondition == 3  
 Co\_Drag = input('Enter Constant Drag Value. ');  
end  
  
% Set Initial Conditions for ode function  
xo = 0;  
yo = 0;  
mo = m(1);  
Vo = 0.447;  
y0 = [Vo thetao xo yo mo];  
  
% Set the limit for the ode function  
Opt = odeset('Events', @StopFunc);  
  
[t,y] = ode45(@(t,y) ode\_func\_pa\_per(t,y),tspan,y0,Opt);

%% Extract Data From Arrays  
V = y(:,1);  
theta = y(:,2)./0.0174533;  
xpos = y(:,3);  
zpos = y(:,4);  
mass = y(:,5);  
  
% Find X and Z max values  
[valx,idxx] = max(xpos);  
[valz,idxz] = max(zpos);  
Zx = valz;  
Xx = valx;  
tX = t(idxx);  
tZ = t(idxz);  
disp(' ');  
  
formatSpec = 'Apogee is %6.3f km occuring at %5.3f seconds \n';  
fprintf(formatSpec,Zx,tZ)  
formatSpec = 'Landing Distance is %6.3f km occuring at %5.3f seconds \n';  
fprintf(formatSpec,Xx,tX)  
  
figure(1)  
hold on  
plot(xpos,zpos)  
xlabel('Distance(km)');  
ylabel('Altitude (km)');  
grid on

* + - * 1. ODE Function Parallel-Perpendicular

The function that contains the differential equations used in the ODE solver.

function dydt = ode\_func\_pa\_per(t,y)  
global TCondition DCondition thetao Ls Lr Co\_Drag  
  
dydt = zeros(5,1);  
if (TCondition == 1 || TCondition == 3)  
 Vcr = sqrt((thrustode(t)/Mass(t))\*Ls\*2);  
else  
 Vcr = sqrt((Thrust(t)/Mass(t))\*Ls\*2);  
end  
  
if (TCondition == 1 || TCondition == 3) && DCondition ==1 % Thrust and Drag Data Case  
dydt(1) = thrustode(t)/Mass(t) - ((rhoe(y(4))\*Cdode(t,y(1))\*Area(t))/...  
 (2\*Mass(t)))\*(y(1)^2) - Gravity(y(4))\*sin(y(2)); % dV/dt  
elseif (TCondition == 1 || TCondition == 3) && DCondition ==2 % Thrust Data, Sim Drag Case  
dydt(1) = thrustode(t)/Mass(t) - ((rhoe(y(4))\*Cd(y(1),y(4))\*Area(t))/...  
 (2\*Mass(t)))\*(y(1)^2) - Gravity(y(4))\*sin(y(2)); % dV/dt  
elseif (TCondition == 1 || TCondition == 3) && DCondition ==3  
CD = Co\_Drag;  
dydt(1) = thrustode(t)/Mass(t) - ((rhoe(y(4))\*CD\*Area(t))/...  
 (2\*Mass(t)))\*(y(1)^2) - Gravity(y(4))\*sin(y(2)); % dV/dt  
elseif TCondition ==2 && DCondition ==1 % Avg Thrust, Drag Drag Data Case  
dydt(1) = Thrust(t)/Mass(t) - ((rhoe(y(4))\*Cdode(t)\*Area(t))/...  
 (2\*Mass(t)))\*(y(1)^2) - Gravity(y(4))\*sin(y(2)); % dV/dt  
elseif TCondition ==2 && DCondition ==2 % Avg Thrust, Sim Drag Case  
dydt(1) = Thrust(t)/Mass(t) - ((rhoe(y(4))\*Cd(y(1),y(4))\*Area(t))/...  
 (2\*Mass(t)))\*(y(1)^2) - Gravity(y(4))\*sin(y(2)); % dV/dt  
elseif TCondition ==2 && DCondition ==3  
CD = Co\_Drag;  
dydt(1) = Thrust(t)/Mass(t) - ((rhoe(y(4))\*CD\*Area(t))/...  
 (2\*Mass(t)))\*(y(1)^2) - Gravity(y(4))\*sin(y(2)); % dV/dt  
end  
  
if y(3) <= Lr\*cos(thetao)  
 dydt(2) = 0;  
elseif y(1) <= Vcr  
 dydt(2) = 0;  
else  
 dydt(2) = -(Gravity(y(4))/y(1))\*cos(y(2)); % dtheta/dt  
end  
  
dydt(3) = y(1)\*cos(y(2)); % dx/dt  
dydt(4) = y(1)\*sin(y(2)); % dz/dt  
dydt(5) = -Mass(t);

* + - 1. Additional Files Used for 4th Order Scripts

The files designed specifically for use with the 4th order scripts, mostly to aid the ODE solver in utilizing established data arrays.

* + - * 1. Stop ODE Function

The function that is used as an ‘options’ operator in the solver, the condition being to stop the solver from running if the value for altitude goes below zero.

function [value, isterminal, direction] = StopFunc(t,y)  
value = (y(4) <= 0);  
isterminal = 1; % Stop the integration  
direction = 0;

* + - * 1. Thrust ODE Sub-Function

The sub-function that tells the ODE solver which value for thrust to use.

function [thrust] = thrustode(tau)  
global ThrustData dt total\_burn\_time  
if tau <= total\_burn\_time  
 n = length(ThrustData)-1;  
 trange = (0:dt:(length(ThrustData)-1)\*dt)';  
 for j = 1:n  
 if tau == trange(j)  
 thrust = ThrustData(j);  
 elseif tau > trange(j) && tau < trange(j+1)  
 thrust = ThrustData(j);  
 elseif tau >= trange(j+1)  
 thrust = ThrustData(j+1);  
 end  
 end  
else  
 thrust = 0;  
end  
  
% n = length(trange);  
% for i= 1:n  
% timeclosest(i) = min(abs(tau - trange(i)));  
% end  
% result = find(timeclosest == 0);  
% thrust = ThrustData(result);  
% end

* + - * 1. ODE Drag Coefficient – File Version

Sub-function that tells the solver which drag coefficient value to access from a read in text file.

function [Cd] = Cdode(tau,v)  
global DragData dt  
n = length(DragData)-1;  
trange = (0:dt:(length(DragData)-1)\*dt)';  
for j = 1:n  
 if tau == trange(j)  
 Cd = DragData(j);  
 elseif tau > trange(j) && tau < trange(j+1)  
 Cd = DragData(j);  
 elseif tau >= trange(j+1)  
 Cd = DragData(j+1);  
 end  
end  
Cd = sign(v)\*Cd;  
end

* + - * 1. ODE Drag Coefficient – Profile Version

Sub-Function that tells the solver which drag coefficient value to access from the selected drag profile,

function [Cd] = Cd(v,z)  
global Rg mend  
[T,~,~] = SIAtm(z);  
c = sqrt(1.4\*Rg\*T);  
M = abs(v)/c;  
Cd = DragCoefficient(M,mend);  
end

* + 1. Sub-Functions Used in Solution Scripts

The scripts used for calculating certain variables over time and position during flight.

* + - 1. Profile Drag Coefficient

The script that takes the current Mach value and determines what the value for the drag coefficient is at that given speed. Adapted from script found in Ref. 3.

function Cd = DragCoefficient(M,mend)  
global DragData TimeStep  
  
n = length(DragData)-1;  
Mrange = (0:TimeStep:mend);  
for j = 1:n  
 if M == Mrange(j)  
 Cd = DragData(j);  
 elseif M > Mrange(j) && M < Mrange(j+1)  
 Cd = DragData(j);  
 elseif M >= Mrange(j+1)  
 Cd = DragData(j+1);  
 end  
end  
  
end

* + - 1. Average Thrust Profile

The script that builds the array for current thrust, based off stage thrust values and burn out times. Taken from Ref. 3.

function thrust = Thrust(tau)  
  
global tbo Tstage twait tb nos  
  
%thrust throughout flight of rocket  
if nos == 1  
 if tau >= 0 & tau <= tb  
 thrust = Tstage;  
 else  
 thrust= 0;  
 end  
elseif nos == 2  
 if tau >= 0 && tau <= tb(1)  
 thrust = Tstage(1);  
 elseif tau > tb(1) && tau <= tbo(1)  
 thrust = 0;  
 elseif tau > tbo(1) && tau <= tbo(1)+twait  
 thrust = 0;  
 elseif tau > tbo(1)+twait && tau <= tbo(2)  
 thrust = Tstage(2);  
 else  
 thrust= 0;  
 end  
elseif nos == 3  
 if tau >= 0 && tau <= tb(1)  
 thrust = Tstage(1);  
 elseif tau > tb(1) && tau <= tbo(1)  
 thrust = 0;  
 elseif tau > tbo(1) && tau <= tbo(1)+twait  
 thrust = 0;  
 elseif tau > tbo(1)+twait && tau <= tbo(2)  
 thrust = Tstage(2);  
 elseif tau > tbo(2) && tau <= tbo(2)+twait  
 thrust = 0;  
 elseif tau > tbo(2)+twait && tau <= tbo(2)+2\*twait  
 thrust = 0;  
 elseif tau > tbo(2)+2\*twait && tau <= tbo(3)  
 thrust = Tstage(3);  
 else  
 thrust = 0;  
 end  
elseif nos == 4  
 if tau >= 0 && tau <= tb(1)  
 thrust = Tstage(1);  
 elseif tau > tb(1) && tau <= tbo(1)  
 thrust = 0;  
 elseif tau > tbo(1) && tau <= tbo(1)+twait  
 thrust = 0;  
 elseif tau > tbo(1)+twait && tau <= tbo(2)  
 thrust = Tstage(2);  
 elseif tau > tbo(2) && tau <= tbo(2)+twait  
 thrust = 0;  
 elseif tau > tbo(2)+twait && tau <= tbo(2)+2\*twait  
 thrust = 0;  
 elseif tau > tbo(2)+2\*twait && tau <= tbo(3)  
 thrust = Tstage(3);  
 elseif tau > tbo(3) && tau <= tbo(3)+twait  
 thrust = 0;  
 elseif tau > tbo(3)+twait && tau <= tbo(3)+2\*twait  
 thrust = 0;  
 elseif tau > tbo(3)+2\*twait && tau <= tbo(4)  
 thrust = Tstage(4);  
 else  
 thrust= 0;  
 end  
else  
 disp('ERROR! Use a number of stages between 1-4.');  
 return  
end

* + - 1. Area Calculation

Sub-Function that loads the reference area over time based on the stage diameters and burn out times. Taken from Ref. 3.

function area = Area(tau)  
  
global tbo A twait tb nos  
  
%area throughout flight of rocket  
if nos == 1  
 area = A(1);  
elseif nos == 2  
 if tau >= 0 && tau <= tb(1)  
 area = A(1);  
 elseif tau > tb(1) && tau <= tbo(1)  
 area = A(1);  
 elseif tau > tbo(1) && tau <= tbo(1)+twait  
 area = A(2);  
 elseif tau > tbo(1)+twait && tau <= tbo(2)  
 area = A(2);  
 else  
 area= A(2);  
 end  
elseif nos == 3  
 if tau >= 0 && tau <= tb(1)  
 area = A(1);  
 elseif tau > tb(1) && tau <= tbo(1)  
 area = A(1);  
 elseif tau > tbo(1) && tau <= tbo(1)+twait  
 area = A(2);  
 elseif tau > tbo(1)+twait && tau <= tbo(2)  
 area = A(2);  
 elseif tau > tbo(2) && tau <= tbo(2)+twait  
 area = A(2);  
 elseif tau > tbo(2)+twait && tau <= tbo(2)+2\*twait  
 area = A(3);  
 elseif tau > tbo(2)+2\*twait && tau <= tbo(3)  
 area = A(3);  
 else  
 area = A(3);  
 end  
elseif nos == 4  
 if tau >= 0 && tau <= tb(1)  
 area = A(1);  
 elseif tau > tb(1) && tau <= tbo(1)  
 area = A(1);  
 elseif tau > tbo(1) && tau <= tbo(1)+twait  
 area = A(2);  
 elseif tau > tbo(1)+twait && tau <= tbo(2)  
 area = A(2);  
 elseif tau > tbo(2) && tau <= tbo(2)+twait  
 area = A(2);  
 elseif tau > tbo(2)+twait && tau <= tbo(2)+2\*twait  
 area = A(3);  
 elseif tau > tbo(2)+2\*twait && tau <= tbo(3)  
 area = A(3);  
 elseif tau > tbo(3) && tau <= tbo(3)+twait  
 area = A(3);  
 elseif tau > tbo(3)+twait && tau <= tbo(3)+2\*twait  
 area = A(4);  
 elseif tau > tbo(3)+2\*twait && tau <= tbo(4)  
 area = A(4);  
 else  
 area= A(4);  
 end  
else  
 disp('ERROR! Use a number of stages between 1-4.');  
 return  
end

* + - 1. Mass Calculation

Sub-Function that determines the total weight based on mass flow rates per stage and current flight time. Taken from Ref. 3.

function [m] = Mass(tau)  
  
global mi mf mdot tbo twait tb nos  
  
%mass inbetween stages  
if nos == 1  
 if tau >= 0 & tau <= tb  
 m = mi - mdot\*tau;  
 else  
 m = mf;  
 end  
elseif nos == 2  
 if tau >= 0 && tau <= tb(1)  
 m = mi(1) - mdot(1)\*tau;  
 elseif tau > tb(1) && tau <= tbo(1)  
 m = mf(1);  
 elseif tau > tbo(1) && tau <= tbo(1)+twait  
 m = mi(2);  
 elseif tau > tbo(1)+twait && tau <= tbo(2)  
 m = mi(2) - mdot(2)\*(tau-(tbo(1)+twait));  
 else  
 m = mf(2);  
 end  
elseif nos == 3  
 if tau >= 0 && tau <= tb(1)  
 m = mi(1) - mdot(1)\*tau;  
 elseif tau > tb(1) && tau <= tbo(1)  
 m = mf(1);  
 elseif tau > tbo(1) && tau <= tbo(1)+twait  
 m = mi(2);  
 elseif tau > tbo(1)+twait && tau <= tbo(2)  
 m = mi(2) - mdot(2)\*(tau-(tbo(1)+twait));  
 elseif tau > tbo(2) && tau <= tbo(2)+twait  
 m = mf(2);  
 elseif tau > tbo(2)+twait && tau <= tbo(2)+2\*twait  
 m = mi(3);  
 elseif tau > tbo(2)+2\*twait && tau <= tbo(3)  
 m = mi(3) - mdot(3)\*(tau-(tbo(2)+2\*twait));  
 else  
 m = mf(3);  
 end  
elseif nos == 4  
 if tau >= 0 && tau <= tb(1)  
 m = mi(1) - mdot(1)\*tau;  
 elseif tau > tb(1) && tau <= tbo(1)  
 m = mf(1);  
 elseif tau > tbo(1) && tau <= tbo(1)+twait  
 m = mi(2);  
 elseif tau > tbo(1)+twait && tau <= tbo(2)  
 m = mi(2) - mdot(2)\*(tau-(tbo(1)+twait));  
 elseif tau > tbo(2) && tau <= tbo(2)+twait  
 m = mf(2);  
 elseif tau > tbo(2)+twait && tau <= tbo(2)+2\*twait  
 m = mi(3);  
 elseif tau > tbo(2)+2\*twait && tau <= tbo(3)  
 m = mi(3) - mdot(3)\*(tau-(tbo(2)+2\*twait));  
 elseif tau > tbo(3) && tau <= tbo(3)+twait  
 m = mf(3);  
 elseif tau > tbo(3)+twait && tau <= tbo(3)+2\*twait  
 m = mi(4);  
 elseif tau > tbo(3)+2\*twait && tau <= tbo(4)  
 m = mi(4) - mdot(4)\*(tau-(tbo(3)+2\*twait));  
 else  
 m = mf(4);  
 end  
else  
 disp('ERROR! Use a number of stages between 1-4.');  
 return  
end

* + - 1. Gravity Calculation

The sub-function that calculates gravity based on the current altitude. Taken from Ref. 3.

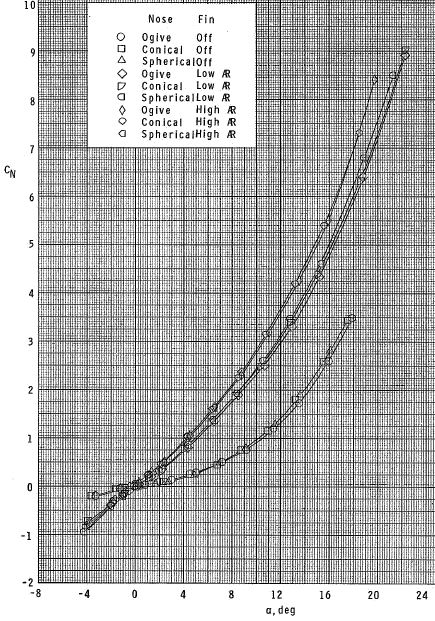
function g = Gravity(z)  
  
global go RE  
  
%varying gravity as altitude varies  
  
g = go\*(RE/(RE+z))^2;

* + - 1. Standard Atmosphere

The sub-function that determines temperature, density, and pressure based on the standard atmosphere found in Ref. 19. The sub-function ‘rhoe.m’ uses the same code but only outputs density for use with the ODE solvers.

function [t,p,rho]= SIAtm(altitude)  
%%givens  
hg=altitude; %geometric alt in meters  
R=287; %Gas Constant  
re=6.356766e6; %radius of earth  
g0=9.8066; %gravity  
h = re/(re+hg)\*hg;  
  
% Altitude in meters  
h0 = 0.0;  
h1 = 11000.0;  
h2 = 25000.0;  
h3 = 47000;  
h4 = 53000;  
h5 = 79000;  
h6 = 90000;  
h7 = 105000;  
  
% Temperature in Kelvin.  
t0 = 288.16;  
t1 = 216.66;  
t2 = 216.66;  
t3 = 282.66;  
t4 = 282.66;  
t5 = 165.66;  
t6 = 165.66;  
t7 = 225.66;  
  
%%equations  
  
a01 = (t1-t0)/(h1-h0); % slope of T/h in K/m for 0 to 11 km  
a23 = (t3-t2)/(h3-h2); % slope of T/h in K/m for 25 to 47 km  
a45 = (t5-t4)/(h5-h4); % slope of T/h in K/m for 53 to 79 km  
a67 = (t7-t6)/(h7-h6); % slope of T/h in K/m for 90 to 105 km  
  
% Pressure in N/m^2  
p0 = 101325.0;  
p1 = p0\*(t1/t0)^(-g0/(a01\*R));  
p2 = p1\*exp(-(g0/(R\*t1))\*(h2-h1));  
p3 = p2\*(t3/t2)^(-g0/(a23\*R));  
p4 = p3\*exp(-(g0/(R\*t3))\*(h4-h3));  
p5 = p4\*(t5/t4)^(-g0/(a45\*R));  
p6 = p5\*exp(-(g0/(R\*t5))\*(h6-h5));  
p7 = p6\*(t7/t6)^(-g0/(a67\*R));  
  
if h < h1  
 t = t0+a01\*h;  
 p = p0\*(t/t0)^(-g0/(a01\*R));  
elseif h < h2  
 t = t1;  
 p= p1\*exp(-(g0/(R\*t))\*(h-h1));  
elseif h < h3  
 t = t2+a23\*(h-h2);  
 p = p2\*(t/t2)^(-g0/(a23\*R));  
elseif h < h4  
 t = t3;  
 p = p3\*exp(-(g0/(R\*t))\*(h-h3));  
elseif h < h5  
 t = t4+a45\*(h-h4);  
 p = p4\*(t/t4)^(-g0/(a45\*R));  
elseif h < h6  
 t = t5;  
 p = p5\*exp(-(g0/(R\*t))\*(h-h5));  
elseif h < h7  
 t = t6+a67\*(h-h6);  
 p = p6\*(t/t6)^(-g0/(a67\*R));  
else  
 t= t7;  
 p = p7\*exp(-(g0/(R\*t))\*(h-h7));  
end  
rho=p/(R\*t); % Density in kg/m^3  
end

ADDITIONAL FIGURES



Example of Chart used to find *CN* Values

An example of the charts used to find *CN* and *CA*values, this one taken from Ref. 7. Was used to determine the best nose and fin shape, as noted in section 2.4.