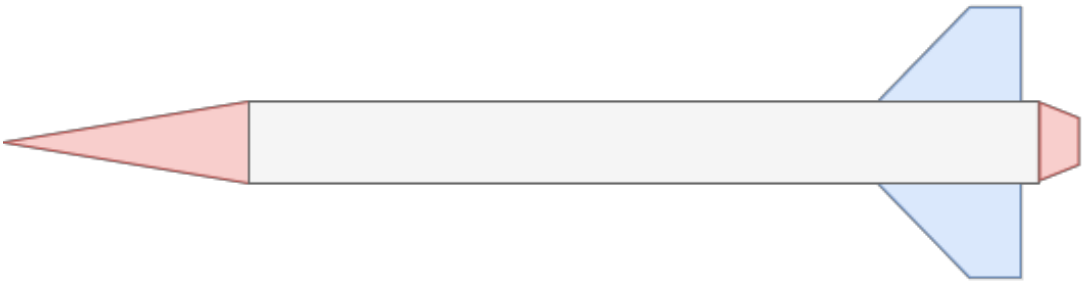
TÉCNICO LISBOA



Development of a Multidisciplinary Framework for Hybrid Rockets

Miguel Filipe Faia Morgado

Thesis to obtain the Master of Science Degree in

Integrated MSc. in Aerospace Engineering

Supervisors: Prof. Afzal Suleman
Prof. Frederico José Prata Rente Reis Afonso

Examination Committee

Chairperson: Prof. Filipe Szolnoky Ramos Pinto Cunha
Supervisor: Prof. Afzal Suleman
Member of the Committee: Prof. Edgar Caetano Fernandes

July 2022

To my parents, for setting a high standard for me to grow beyond.

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Abstract

Hybrid rocket propulsion is characterised by one propellant being a liquid and the other one a solid and it is becoming increasingly prevalent among *New Space* launch vehicles. In fact, companies like HyImpulse or Gilmour Space are developing sounding rockets or even small orbital launch vehicles that use hybrid rocket motors. However, the technology is still not as developed as liquid or solid rockets are. It is known that hybrid rockets can be an environmentally friendly and cost-effective option, which could explain the recent trend to use them as the industry transitions into an ecosystem of affordable access to space.

In this context, this work intends to provide a tool for the development and optimization of hybrid-propelled launch vehicle concepts, using custom, adjustable models developed in a MATLAB environment to suit a wide range of requirements and mission types. Four different disciplines – Propulsion, Sizing, Aerodynamics and Trajectory – are iterated on an MDF optimization loop. The program uses inputs from DATCOM and NASA's CEA code for the aerodynamic coefficients and for the combustion chamber properties, respectively. The trajectory module was upgraded to a 3-DOF with rotation model, allowing the optimization to support constraints and multi-objective function variables such as apogee and burnout velocity. These combined methods grant this tool a multidisciplinary approach that is not very common in hybrid propulsion design optimization. The tool was validated and developed for rockets using the nitrous oxide-paraffin propellant combination, but other propellant choices are possible. The results are consistent with flight-proven rockets.

Keywords: Hybrid Propulsion, MDO, Suborbital Rocket, MATLAB, Trajectory

Resumo

A propulsão de foguete híbrida caracteriza-se por utilizar um propelente no estado líquido e outro no estado sólido e está a ganhar notoriedade na indústria de lançadores do *New Space*. Empresas como a HyImpulse ou a Gilmour Space estão a desenvolver pequenos lançadores orbitais que fazem uso desta tecnologia. Porém, esta não está ainda a par da propulsão sólida ou líquida em termos de desenvolvimento. À medida que a indústria transita para a sustentabilidade e baixos custos no acesso ao espaço, esta tendência recente pode ser explicada pelos propelentes híbridos serem mais amigos do ambiente e relativamente baratos.

Neste contexto, o objetivo deste trabalho é fornecer uma ferramenta para o desenvolvimento conceptual e otimização de lançadores híbridos, com recurso a modelos MATLAB customizáveis e ajustáveis a uma vasta gama de requisitos e tipos de missão. São executadas quatro disciplinas - Propulsão, Dimensionamento, Aerodinâmica e Trajetória - num esquema de otimização MDF. O programa utiliza inputs dos programas CEA, da NASA, e DATCOM para obter as propriedades da câmara de combustão e os coeficientes aerodinâmicos, respetivamente. A disciplina da Trajetória foi atualizada para um modelo com 3 graus de liberdade, permitindo o uso de constrangimentos como o apogeu ou a velocidade. A combinação destas disciplinas confere à ferramenta um cariz multidisciplinar pouco comum na otimização de propulsão híbrida. A ferramenta foi validada e desenvolvida com a combinação óxido nitroso-parafina, mas outras escolhas de propelente também são possíveis. Os resultados mostram-se coerentes com dados de foguetes reais.

Palavras-Chave: Propulsão Híbrida, MDO, Foguete Suborbital, MATLAB, Trajetória

Contents

List of Figures

List of Tables

Abbreviations

AAO	All at Once
ABS	Acrylonitrile Butadiene Styrene
AMROC	American Rocket Company
CEA	Chemical Equilibrium with Applications
COESA	Committee on Extension to the Standard Atmosphere
DOF	Degree(s) of Freedom
DLR	German Aerospace Center
EuRoC	European Rocketry Challenge
GA	Genetic Algorithm
GLOW	Gross Lift-off Weight
H_2O_2	Hydrogen Peroxide
HRE	Hybrid Rocket Engine
HTP	High-Test Peroxide
HTPB	Hydroxyl-terminated polybutadiene
IDF	Individual Discipline Feasible
LEO	Low Earth Orbit
LOX	Liquid Oxygen
MDA	Multidisciplinary Analysis
MDF	Multidisciplinary Feasible
MDO	Multidisciplinary Design Optimization
N_2O	Nitrous Oxide
N_2O_4	Nitrogen Tetroxide
NASA	National Aeronautics and Space Administration
PBAN	Polybutadiene Acrylonitrile
RP-1	Rocket Propellant-1
SAND	Simultaneous Analysis and Design
SRM	Solid Rocket Motor
SQP	Sequential Quadratic Programming
TRL	Technology Readiness Level
TVC	Thrust Vectoring Control

XDSM Extended Design Structure Matrix

Nomenclature

c Effective Exhaust Velocity or Constraint Function

c^c Consistency Constraint Function

C_{inj} Effective Injection Area

d_{ext} External Diameter

f Objective Function

g Gravitational Acceleration

G Mass Flux

I_{sp} Specific Impulse

I_{tot} Total Impulse

L_f Fuel Grain Length

m Mass

$\dot{m}$ Mass Flow Rate

O/F Oxidizer-to-Fuel ratio

p_{CC} Combustion Chamber Pressure

p_{feed} Pressure Loss through the Injector and Feed System

p_{OT} Oxidizer Tank Pressure

r_{port} Fuel Port Radius

$\dot{r}$ Fuel Regression Rate

$\boldsymbol{R}$ Discipline Residuals Vector

Δv Change in Velocity

x Downrange Distance

$\boldsymbol{x}$ Design Variables Vector

$\boldsymbol{y}$ Coupling Variables Vector

$\hat{\boldsymbol{y}}$ Coupling Variable Copies Vector

$\bar{\boldsymbol{y}}$ State Variables Vector

z Altitude

γ Flight Path Angle

ρ Density

Chapter 1

Introduction

1.1 Topic Overview

Space travel has been a reality for more than sixty years. However, rockets and rocket-powered spaceplanes remain the only transportation vehicles capable of reaching space and achieving orbit of the Earth. Thus, further developing rocket technology is key in increasing accessibility to outer space. The state of affairs currently allows many different countries and companies to place objects in orbit for varying purposes, from military applications to space tourism. But the so-called *democratization of space* [1] is still an ongoing process.

The technology has proliferated not only on the space segment, but also on the launch segment as well. The launch vehicle market can only grow sustainably, however, if there is demand for it on the payload side. In fact, luckily, the satellite market has been steadily growing, especially the CubeSat market [2, 3]. Small satellites can be launched individually or using some sort of rideshare solution. However, the latter is not ideal for many applications, like, for example, if the customer requires a specific deployment orbit or if the launch is requested on short notice; either of these scenarios would render a combined launch of many such payloads impractical.

Hence, there will always be a niche for dedicated small launchers for small payloads and many launch vehicle start-ups are betting on this prospect [4, 5]. This is being regarded as part of the *New Space* era, a significant wave of trends that are shaping the space sector into being more liberal and agile [6]. For example, the first successful launch from Rocket Lab, one of those *New Space* small launcher companies, was completed in January 2018. Rocket Lab is probably the most successful private small launcher company to date, but a few more have had their maiden launches since and even more are expected to do so in the coming years.

Environmental concerns, paramount in the 21st century, and profitability are two variables that must be taken into account when designing a modern launch vehicle. One common approach that addresses these issues is designing the vehicle to be cheap to mass produce and efficient. To accomplish this, an increasing number of companies is employing hybrid rockets as the main propulsion devices in their small launch vehicles - Gilmour Space Technologies, from Australia, HyImpulse Technologies [4], from Germany,

and HyPr Space [5], from France, just to name a few. In fact, combining liquid and solid propellants give hybrid rockets some competitive advantages, which are further explored in chapter 2.

The hybrid rocket is not a new concept. The first rocket launch using such technologies is thought to have occurred in 1933 [7], in the former Soviet Union. The vehicle, two and a half meters tall, burned liquid oxygen as the oxidizer, and gelled gasoline as the fuel. Despite its early debut, the technology would remain sparsely used until the formation of the American Rocket Company (AMROC), which in the late 20th century tried to develop large hybrid-propelled rocket boosters. It was only at the turn of the millennium that hybrid rocket technology had its resurgence, certainly thanks to AMROC and to research centers such as the Standford University [7]. As new fuel technologies began to be explored and developed, hybrid rockets became a noteworthy technology, culminating in the hybrid-propelled SpaceShipOne's flight to the Kármán Line, in 2004.

Finally, designing a rocket is a hard skill and necessarily a multidisciplinary endeavour, which directly correlates to the objectives of this work. A bad design can be inefficient or, worse, not really useful. Optimization of a vehicle's design can not only lower the needed resources (specially, propellant) for its operation and accompanying pollutant emissions, but is also vital for one intended to be economically competitive. Multidisciplinary Design Optimization, which is a relatively recent field of knowledge, enabled by the always improving computer technology, provides tools to solve the problem of designing not just to meet requirements, but to provide an optimal result.

1.2 Motivation

Over the years, many aerospace students have the opportunity to participate in university clubs and extra-curricular projects linked with space technology. Rocketry teams, for example, are one of the most complete experiences an engineering student can partake. Teams such as Rocket Experiment Division (Figure 1.1), from Técnico Lisbon, are competing among each other annually in rocketry engineering competitions. The most relevant two are perhaps the Spaceport America Cup, held in the United States, and the European Rocketry Challenge (EuRoC), held in Portugal.

The goal of the competitions is to develop a sounding rocket that delivers a standard-mass payload to a very specific target altitude, usually 10000 ft or 30000 ft. Many of the teams choose to develop hybrid rocket engines to propel their rockets up to the target altitude. Furthermore, mass budgets are tightly controlled and some vehicles even feature aerodynamic altitude control mechanisms (such as airbrakes) to reach as close as possible to the target apogee, without surpassing it.

Having said that, it's not too far fetched to say that the engineering challenge offered by this competition format closely mimics the challenge of designing and optimizing a small launch vehicle to put a payload into low Earth orbit - albeit, in a very different scale, of course. Hence, these competition-level sounding rockets can provide a simpler, more forgiving test-bed to develop and validate design and optimization tools, namely software, before attempting to use them for suborbital flights or orbital-class rocket vehicles. This is precisely the philosophy behind this work.

This thesis is also being developed in the context of a booming launch industry, with new launch

Figure 1.1: The Rocket Experiment Division team at the European Rocketry Challenge, in October 2021, with their solid rocket *Blimunda*.

vehicles being announced every year and the reusability paradigm shift still ongoing. There's also a growing interest around sounding rockets *per se*, as they can be very useful for incremental technology testing. For this purpose, in Portugal, for example, an enterprise consortium is studying Viriato [8], a reusable suborbital launcher.

In addition to all of this, there's interest in flying these vehicles from Europe and from Portuguese soil, namely from the Azores, making this thesis subject very much current in the grand scheme of international and national space policy and technology development.

1.3 Objectives

In the current landscape of new emerging space launch solutions, this thesis is intended to continue the development of an engineering software tool to help in the conceptual and preliminary optimal design of hybrid propulsion sounding rockets or even small orbital launch vehicles. It aims to support as broad a design range as possible, so it remains relevant and useful for a large variety of mission requirements and rocket configurations.

To do so, the tool shall model a vehicle using a multidisciplinary approach in which each discipline is addressed sequentially in an optimization loop. The main objectives are:

- Explore the conceptual design optimization of hybrid propulsion systems and hybrid rockets;
- Improve the performance of the existing MATLAB multidisciplinary design optimization tool;
- Expand the tool's optimization envelope and capability;
- Validate the tool by comparing it with test flight data.

Since the working assumption is that the scope of the project lies in conceptual design, no high-fidelity tools like computational fluid dynamics or finite element analysis will be used. Quicker, less resource-intensive models shall be chosen.

1.4 Previous Work

This work follows on the footsteps of Gustavo Yamada, who worked on this software in his master thesis, in 2019 [9]. He developed an optimization program for hybrid sounding rockets in MATLAB using a modified Individual Discipline Feasible Multidisciplinary Design Optimization architecture, to which a gradient-based search algorithm was applied.

The program was tailored for the optimization of university student hybrid rockets competing in the Spaceport America Cup competition. Hence, the design envelope contemplated low altitude (3 km apogee) sounding rockets with student-developed propulsion. The propulsion discipline is the heart of the program and was developed first by the University of Victoria's UVic Rocketry team and by Benjamin Klammer [10].

Those Propulsion models were adapted by Yamada, to which he added a heat transfer model for the oxidizer tank. However, given the usually short burn times (a few seconds), the heat transfer model has little effect on the final performance and, as such, it was removed from the present work to improve performance and simplify the optimization architecture.

1.5 Thesis Outline

After the brief introductory Chapter 1, Chapter 2 will introduce the topic of Hybrid Rocket Propulsion, along with some general space propulsion concepts as well. The different components of a hybrid rocket will be presented and explained. On section 2.1.4, the regression rate problem will be discussed and the new developments that partially solved that problem, allowing for the current resurgence of hybrid propulsion to emerge, are shown.

Then, on Chapter 3, the thesis reviews the classic multidisciplinary design optimization approaches and how they apply to rocket design, followed by the methodology employed on this work to solve that optimization problem in Chapter 4.

Some results are briefly presented in Chapter 5 before the conclusions and future work are laid out in the final part, Chapter 6.

The source code for the MATLAB tool can be found in the Appendix.

Chapter 2

Hybrid Rocket Technology Overview and Applications

2.1 Hybrid Rocket Propulsion

Chemical bi-propellant rocket propulsion, the most common type of rocket propulsion, is characterised by burning two propellants - a fuel and an oxidizer - in a rocket engine, with or without the need for a catalyst or an igniter to begin the reaction. The two propellants are most often physically separated until they are burned. In a liquid propulsion system, the fuel and the oxidizer are stored in separate tanks before they are fed into the combustion chamber, using pressure or pumps. Conversely, in a solid rocket motor, the fuel and the oxidizer are mixed and bonded together in a solid state fuel grain which burns when ignited.

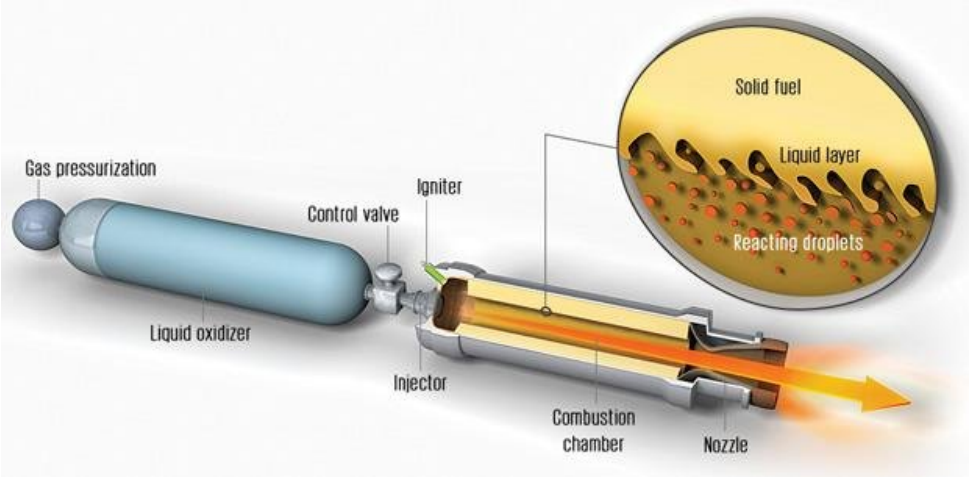


Figure 2.1: Schematic of a Hybrid Rocket Engine. This example features a pressurizing gas tank. Source: [11].

Hybrid rocket propulsion concepts, on the other hand, as the one illustrated on Figure 2.1, make use of these two propellants in different states of matter. One, usually the fuel, is a solid and the other, usually the oxidizer, is in liquid form. These roles can, however, be switched, in which case the rocket is said to

be a reverse hybrid [12]. When one refers to a "hybrid rocket", usually that is a reference to a vehicle that uses a hybrid rocket engine (HRE), sometimes called a hybrid motor, as its main propulsion device. That is the meaning of the expression "hybrid rocket" in the context of this work, although it may be used to refer to just the motor system.

Hybrid rockets are a blend of their solid and liquid counterparts not only in relation to the propellants themselves, but also in the techniques and technology used. Many developments in solid and liquid rocket technology can be applied to hybrids as well, since most of the components and processes of hybrids have an equivalent in at least one of the other two types [13].

2.1.1 Similarities between hybrid and solid propulsion

HREs feature a solid "block" of propellant, called *grain*, stored inside the combustion chamber, much like a solid rocket motor. This is usually the fuel. Logically, greater quantities of fuel require a more voluminous combustion chamber.

A simple solid rocket motor (SRM) is comprised of a nozzle and a combustion chamber. The chamber itself is comprised typically of an outer casing (commonly made of materials such as steel, aluminum or fiber composites [13]), a thermal insulation layer and the propellant grain. The casing is not generally made of a single piece, but includes segments and/or a top bulkhead, sealed off using o-rings (the component that famously and unfortunately failed on the Space Shuttle Challenger, in 1986) and other mechanical devices. Evidently, an SRM can have no moving parts and that can help explain why they are considered relatively simple machines. The same cannot be said of HREs, which require at the very least an actuated oxidizer valve.

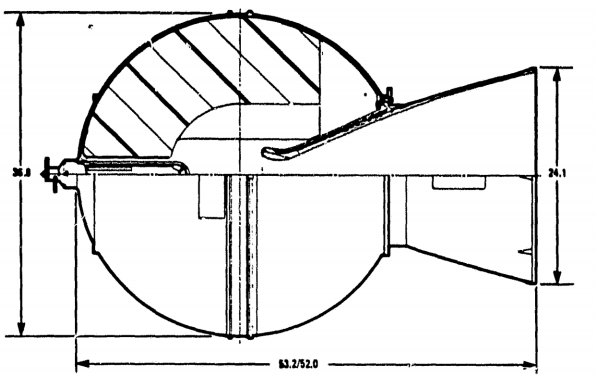


Figure 2.2: A diagram of the Star 37N, a spherical upper stage solid motor. Source: Thiokol [14].

Solid motors are produced in varying sizes and formats. Figure 2.2 shows the Star 37N, a spherical SRM intended to be used outside the atmosphere. However, the most common shape for a solid rocket, especially when addressing atmospheric launch vehicle applications, is a cylindrical tube, containing the propellant grain, which, in turn, features a built-in hollow core. This empty core is where the combustion will take place, as the grain is consumed from its inside walls while its external surface is protected from the flame, placed against the motor's casing. The propellant grain core cross-section can be given any shape, with the simpler option being a circle. The shape is directly related to the size of the internal

burning surface, which directly influences the pressure and thrust generated by the motor. Some core shape examples are shown in Figure 2.3.

A star-shaped core on a propellant grain can ensure the burn is initially quicker, meaning more mass will be ejected through the nozzle and more thrust will be produced at the start of the burn, than for example a circle-shaped core. This is called a regressive burn pattern, since burning surface area, pressure and thrust decrease with time. Neutral or progressive burn patterns can also be achieved to suit vehicle or mission requirements. The same technique can be applied to the solid fuel grains on hybrid rockets to manipulate thrust in a programmed manner.

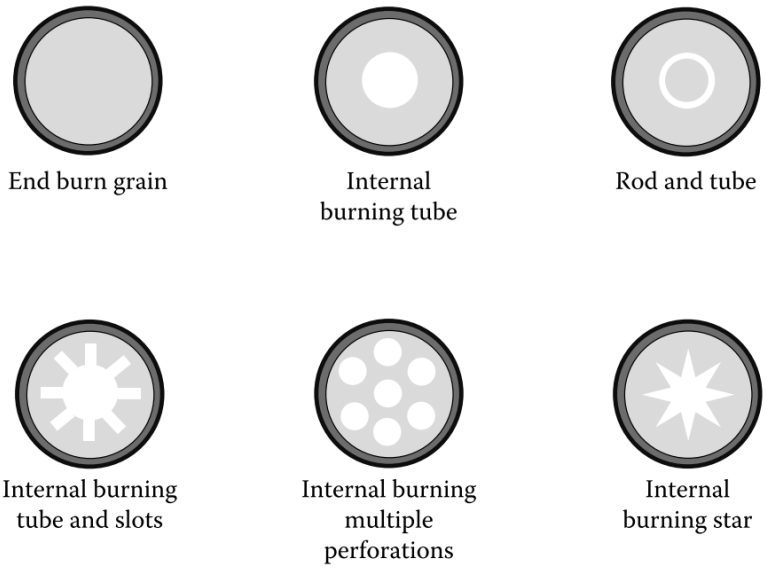


Figure 2.3: Six examples of propellant grain perforation configurations, viewed as cross-sections of the combustion chamber. Source: [12].

The thermal insulation layer is essential in the structural design of many rocket motors. Made from ablative cooling materials, it protects the casing from the intense heat of combustion inside the chamber. Although the propellant grain inside the motor is most often bonded to the chamber walls, which separates and protects them from the hot combustion products until the final moments of motor operation, parts of the chamber remain without propellant attached and, thus, without protection.

Hybrid rockets commonly feature two sections unprotected by the fuel grain - the pre-combustion and the aft combustion chambers, situated on the forward and aft sections of the chamber, respectively. These sections need to be protected by an insulating layer, otherwise the interior temperatures, higher than the melting point of metals, would destroy the casing [15].

Thermal insulation layers are designed to prevent casings from absorbing too much heat during the motor operation. This makes casings typically the only part of a solid motor that can be made reusable. Consider, however, that time is critical here. An optimal design would have the insulation layer just thick enough to ablate fully when the motor burns out. This, however, does not mean the casing remains cool *after* the motor is no longer operating, as this is not really a requirement if the casing is not meant to be reused in another flight. In HREs, this approach might not be useful, either because the casing is meant to be reused or because the mission profile requires it to be restarted later in flight.

Other processes and components of HREs similar to those found in solid rocket motors may be the casting of the solid propellant and the concept of regression rate, but those are subjects of section 2.1.3.

2.1.2 Similarities between hybrid and liquid propulsion

Processes and components of hybrid rockets similar to those found in liquid rockets include, for example, the combustion chamber injectors. These are a combination of perforations, tubes and manifolds that connect the liquid propellant feed lines to the inside of the combustion chamber. These perforations, or holes, on a bulkhead-type wall form the injector assembly, which main purpose is to mix the propellants efficiently inside the combustion chamber [12]. This is usually achieved by splitting the liquid into small droplets (atomization), making sure the gases mix in the correct stoichiometry and disperse as intended throughout the combustion chamber [13]. Injector design varies wildly, since it must consider the characteristics of different propellant combinations and different engine operating states [15]. Some propellants require pre-mixing before being sprayed into the chamber, while most are mixed in the chamber after being sprayed out of the injector holes, in a meticulous pattern of impinging high pressure streams.

In the case of HREs, since there is only one liquid propellant being injected, the injector design can be simpler, as there is no mixing involved, just atomization and dispersion. The mixing happens along the length of the solid fuel grain, which sits inside the elongated combustion chamber.

Perhaps the most easily interchangeable major component between liquid and hybrid rockets is the liquid propellant tank. As stated before, that propellant is usually the oxidizer, while the fuel assumes a solid state and is stored in the combustion chamber instead of in a tank. First of all, the tank is where the oxidizer needed for the thrust-generating combustion process is stored. The more propellant a rocket vehicle carries, the more thrust it can generate from expelling that propellant through a nozzle. On the other hand, the propellant has mass and the more massive a rocket, the more inertia it has and the more propellant it needs to consume to move by a certain amount. In fact, one of the best ways to increase performance of a rocket (e.g. increase *delta-V*) may not be to increase its propellant mass, but to decrease its inert mass. This can be understood by analysing the Tsiolkovsky Rocket Equation, equation (2.1) [16].

$$\Delta v = c \ln \frac{m_0}{m} \tag{2.1}$$

where Δv represents the change in velocity of the vehicle, c is the effective exhaust velocity of the propellants coming out of the nozzle and m and m_0 represent the current and initial mass of the vehicle. When the vehicle depletes its propellant, the difference between m_0 and m is the inert (or dry) mass of the vehicle.

So, it is of paramount importance for the tanks to be lightweight, in order to decrease the total dry mass of the rocket. This is usually achieved by making the tanks out of materials such as aluminum, steel alloys or fiber composites. These materials are both light and strong enough to hold the pressure inside the tank. In fact, another aspect crucial for many space vehicles is the pressurized manner in which liquid propellants are stored. The reasons for this are varied and include [13]:

- In pressure-fed engines, a high-pressure propellant feed is required at the injectors and since fluids

moves from high to low pressure zones, the propellant tank needs an operating pressure higher than that of the combustion chamber of the engine;

- In pump-fed engines, tank pressurization, although at lower pressures, is still important to push the liquid propellant into the pump, mitigating cavitation;
- Some propellant tanks double as structural elements of the launch vehicle and a pressurized vessel can withstand greater structural loads without deforming (the phenomenon can be observed with drink cans, which are easily crushed once opened).

Hence, the necessity of a tank pressurization system is also a commonality between liquid and hybrid propulsion systems. Some tanks are pressurized using mechanical systems such as pistons or flexible bladders, but the most common approach, at least in space applications, is to use direct gas pressurization [13]. Gas pressurization systems are as old as rockets themselves. In fact, the first liquid engine rocket ever to be flown, developed and launched by the rocketry pioneer Robert H. Goddard, in 1926, used a gas pressurization system to push its gasoline and liquid oxygen out of the tanks and into the combustion chamber [13][12].

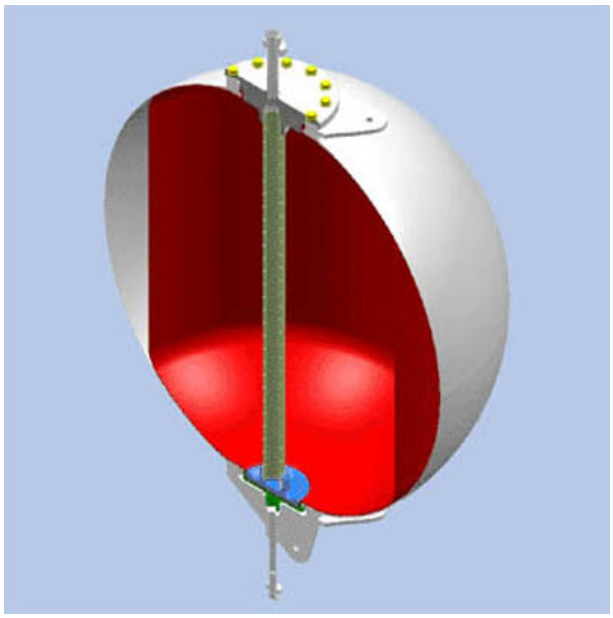


Figure 2.4: Computer model of a spherical hydrazine tank, featuring a bladder pressurization system. Hydrazine is stored inside the bladder (in red) and pushed out of the tank through the central perforated pipe by a pressurizing gas that fills the volume between the rigid tank walls (in white) and the bladder [17].

The pressurizing gas may be sourced from a number of different alternatives. Some options rely on a heat source (for example, the engine) to heat and vaporize a small fraction of the propellant, expanding it, and then flowing it back into the propellant tank. Cryogenic propellants could potentially achieve this effect just by evaporating with time, although at a slower, uncontrolled pace. Another alternative is to create gas from a chemical reaction. This can be achieved by taking a small portion of propellant from the tanks and running it through a gas generator, using it then to pressurize the propellant tanks. However, this option is not easily applicable to HREs, since there are not two different accessible propellants to

feed a reaction, but it could be done with a dedicated gas generator using, for example, a small quantity of solid propellant.

Excluding other exotic methods, two approaches to tank pressurization remain. The most common, not only on hybrid rockets but in general, consists of using an inert gas pressurization system. The other relies on the use of self-pressurizing propellants and is commonly used when nitrous oxide is the oxidizer. Section 2.1.3 explores these propellants in greater detail.

The aforementioned most common approach of using an inert gas (usually Nitrogen or Helium [13]) pressurization system can be further classified according to where in the rocket the gas is stored. If it is stored in the empty ullage volume of the propellant tank, the method is known as a Blowdown system. If the pressurizing gas is stored on a separate high-pressure tank, then it is called a Regulated Pressure system. The stored gas is inserted into the lower pressure propellant tank using feed lines running through a pressure regulator and valves that assure the tank remains at a constant pressure. Regulated Pressure systems can be engineered to use heated gas, just like the propellant pressurization systems described before. This would increase the complexity of the feed system, but larger vehicles could benefit from a reduction in the inert gas mass, since, at higher temperatures, a smaller amount of gas could provide the pressure needed.

Blowdown systems are lighter and less complex than Regulated Pressure systems. No additional gas tank is required, nor extra plumbing, though the tank should be larger to hold the pressurizing gas together with the propellant. The major drawback of this configuration is that the pressure inside the tank decays as the propellant is consumed which translates to decaying pressure downstream of the tank as well.

Finally, hybrid and liquid propellant rockets share a functional upper hand over conventional solid rocket architectures which is the ability to produce variable thrust [18]. To be precise, solid rocket motors do produce variable thrust - in any motor, the thrust is not completely constant over time. There is an initial phase of thrust build up, while the propellant is being ignited, and a wind down phase as it is consumed. Also, as mentioned before, the geometry of solid propellant grains can be tailored, during design and production, to manipulate the way the thrust varies over time and obtain specific thrust profiles. Several examples of this are shown in Figure 2.3.

The actual difference is that liquid and hybrid engines can produce variable thrust in a non-prescribed random manner as needed during flight, using a feature called throttling. On HREs, this is achieved by adjusting the oxidizer valve aperture, feeding the combustion chamber injector assembly with a lower mass flow than optimal. Because thrust is almost proportional to the propellant mass flow, this actuation of the valve results in an almost proportional control of the thrust. While pressure-fed engines use valves, pump-fed engines can throttle by changing the rotary speed of the pumps [13]. This ability comes at a cost, however, since injector flow dynamics change as the overall mass flow through the engine is reduced, stripping away efficiency from atomization and mixing. All in all, when lowering the throttle in an engine to a sub-optimal mass flow, the chamber pressure and specific impulse decrease.

Still, there are glaring differences between the throttle capability of some engines and others. Most throttleable engines allow only for a slight reduction of the generated thrust; up to about 40%. This is true for launch vehicles like the Falcon 9 and the Space Shuttle Orbiter, which reduce thrust during the ascent

phase to minimse aerodynamic loads. The Falcon 9 actually lands thanks to that throtttling capability, even though it is not enough to allow the rocket to "hover".

Reducing the throttle further, on the other hand, something called deep throttling, is a much rarer capability and usually requires a more intricate design of the injectors to avoid combustion instabilities. A good example is the variable geometry pintle injector design from the Apollo Lunar Module landing engine (Figure 2.5), which allowed the astronauts to soft land on the lunar surface and is still the only fully human-rated engine for landings [15].

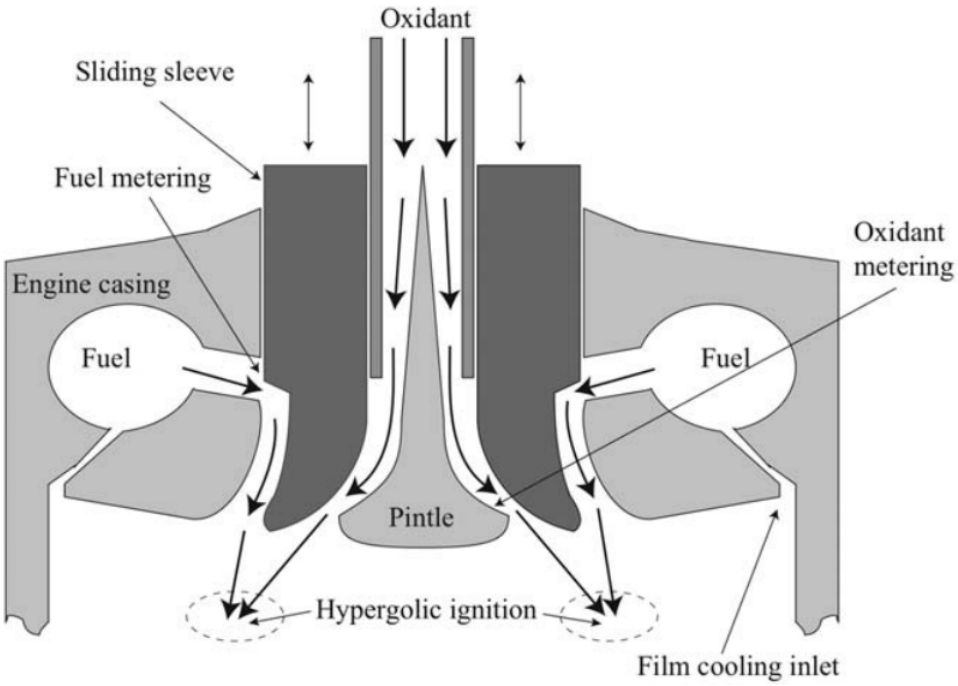


Figure 2.5: Cross-section cut illustration of the pintle injector mechanism which was successfully used in the Apollo Lunar Module Descent Engine. Notice the only moving part is the sliding sleeve, whose movement shapes the two concentric annular openings from where the two propellants are injected. Source: [15].

2.1.3 Propellants

As mentioned before, hybrid rocket engines are characterized by the two propellants being in different states of matter. Thus, an analysis of the different propellants available for these engines can be divided into two sections, the oxidizers and the fuels.

Starting with the liquid oxidizers, right away the two most common choices are liquid oxygen (LOX) and nitrous oxide (N_2O) [15]. LOX is a well-known propellant and provides great performance overall. It's deeply cryogenic, so it is adequate for large hybrid boosters, like the ones intended to be used by the German spaceflight start-up HyImpulse Technologies [4], but not for use-cases requiring a storable chemical.

Nitrous oxide is a more niche propellant than LOX, with its use being more famous in hybrid rocketry than elsewhere. They share as advantages their lower toxicity and cost when compared with other oxidizers such as N_2O_4, for example, a typical storable, hypergolic propellant. Virgin Galactic's SpaceShipTwo,

which propelled four passengers plus a crew of two to the edge of space (86 km in altitude, to be precise) in July 2021 [19], uses nitrous oxide, making this probably the most high-profile use of the oxidizer on a hybrid propulsion system. On the other hand, nitrous oxide is a common choice for small hybrid rocket engine, often times amateur in nature, in part thanks to its self-pressurizing property and relatively safety. Even so, it can decompose quickly if heated or in the presence of impurities, which poses a storage risk.

The oxidizer tank sometimes does not need to be pressurized by a dedicated pressurization system, such as those discussed in section 2.1.2. This is mostly dependent on the oxidizer itself. If the oxidizer is *self-pressurizing*, external pressurizing equipment may not be needed. Self-pressurizing propellants are liquid substances or mixtures that evaporate in such a way that the gaseous phase provides high enough internal pressure to the propellant tank to feed the oxidizer injector (and, hence, the combustion) effectively. This is one of the advantageous properties of nitrous oxide, thanks to its high vapor pressure at room temperature of around 50 bar.

Some cryogenic propellants, such as LOX, can also be evaporated while inside the tank to support self-pressurization, but the experience with that approach is quite limited [13]. However, the mixture of LOX and N_2O has been studied as a viable, self-pressurizing, refrigerated oxidizer, where the more volatile LOX is dissolved in cold nitrous oxide and, as it evaporates, creates pressures of up to 120 bar, which is enough for a wide range of applications [7]. Due to the nitrous oxide being refrigerated to negative temperatures, this mixture is denser than pure N_2O but it still retains the self-pressurizing property and is less prone to decomposition.

Another common oxidizer choice is hydrogen peroxide (H_2O_2). Peroxide has a great heritage as a rocket fuel in Europe, most notably in Great Britain. The 60s British orbital launcher Black Arrow burned peroxide and RP-1 in its liquid rocket engines [20]. Peroxide also burns adequately with solid fuels, making a very volume-efficient combination, as high concentration H_2O_2, called High-Test Peroxide (HTP), has a density almost 50% greater than water. Like nitrous oxide, it also suffers from stability issues and should be stored and handled with great care [7].

As for the fuel in a hybrid rocket motor, it corresponds generally to the solid propellant and is oftentimes considered the bottleneck when it comes to making large hybrid motors viable, due to the difficulty of getting traditional solid fuels to evaporate quickly enough to obtain relevant thrust outputs [7]. That is the case of hydroxyl-terminated polybutadiene (HTPB), a low-energy chemical frequently used in solid rocket propellants as a binder, turning an otherwise incohesive mixture of fuel and oxidizing agents into a homogeneous propellant. Alone, it can be cast into a hybrid combustion chamber and will achieve a respectable performance with various oxidizers.

Polybutadiene acrylonitrile (PBAN) has a similar role and properties to HTPB. It is being used as a constituent of the solid propellant on NASA's Space Launch System boosters and has also seen use as a hybrid fuel. According to Calabro [21], one of the highest values of specific impulse ever recorded from a hybrid engine test, around 380 seconds, used PBAN with lithium and lithium hydride as a solid fuel. The oxidizer was a mixture of liquid oxygen and liquid fluorine.

Metal additives have been added to solid fuels to try to increase its performance, either by increasing the regression rate of the fuel grain or by shifting the oxidizer to fuel ratio. Aluminum powders and metal

hydrides like the lithium hydride mentioned before are common examples.

3D-printing plastics such as ABS have also been tested on small engines and are easily accessible and safe for amateurs or university laboratories. Polyethelene also saw extensive testing as a hybrid fuel by General Electric [21]. In fact, almost any combustible material can be burned in a hybrid combustion chamber and generate thrust, albeit few will give useful performance.

2.1.4 Increasing the regression rate

The combustion process and internal ballistics of a hybrid motor are considerably different from both solid and liquid engines [15]. For adequate burning to occur, the oxidizer and the fuel must be thoroughly mixed. In a solid motor, they are mixed in the solid state before being burned. In a liquid engine, the mixing happens shortly after injection into the chamber, with the two propellants mixing as they vaporize. In a hybrid, the two propellants aren't mixed as soon as they become gaseous. The hot combustion inside the chamber heats the outer layers of the fuel grain core, causing them to evaporate, as drawn in Figure 2.6. This will form a film of fuel vapor covering the rest of the still solid fuel. Eventually, this boundary layer of gas mixes with the also gaseous oxidizer stream flowing through the center of the combustion chamber, from the injector. It is between these two layers of pure oxidizer and pure fuel that the mixing and combustion process actually takes place.

This implies that, in a hybrid motor, heat transfer to the solid fuel is ruled by the behaviour of the boundary layer. This process is different from a solid motor, in which the combustion happens closer to the surface of the grain, immediately after vaporization of the solid propellant. In addition to this, the combustion in a solid is evidently premixed, while a hybrid motor's combustion is diffusive.

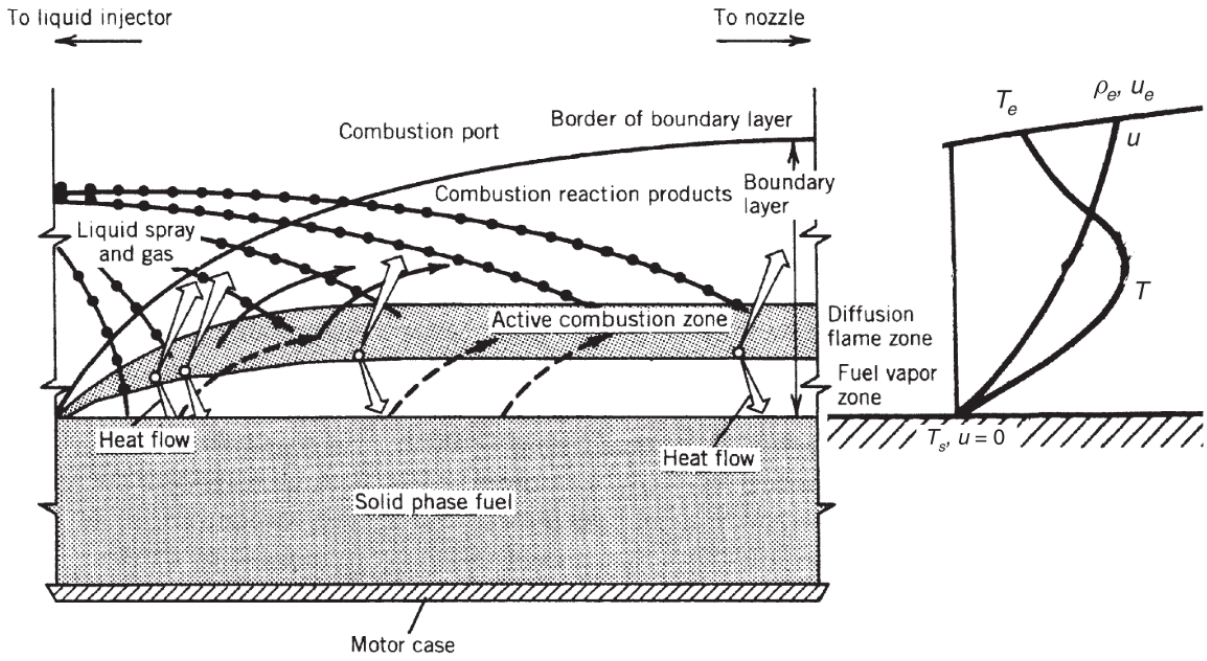


Figure 2.6: Illustration of the three boundary layer model of the hybrid combustion process. Notice the temperature increase in the middle diffusion flame layer. Source: [13].

Because of the boundary layer effect, the complete mixing and combustion of the last portion of the

fuel grain only happens further downstream from the grain end - hence, an aft-combustion chamber, about as long as the chamber is wide [15], is usually needed before the nozzle to improve the combustion efficiency of the motor. This unique combustion process also has implications in the solid fuel regression rate - the rate, usually measured in mm/s, at which the solid fuel surface is consumed. As the fuel surface absorbs heat from convection and radiation, its vaporization creates the fuel boundary layer and, thus, a "blowing" effect that counters the heat transfer. Without an effective heat transfer mechanism, fuel regression rates are low and insufficient to generate the levels of thrust needed for large hybrid rocket boosters and many other applications.

Several approaches have been pursued to solve this issue, most of them intended to facilitate the heat transfer to the fuel [7]. One is called the *mixed hybrid* approach, wherein the solid fuel is doped with a small amount of solid oxidizer. A similar approach is to use metal additives in the fuel, as mentioned in section 2.1.3, to increase the heat radiation mechanism to the fuel [13]. On the other hand, the problem can be addressed by increasing the surface area of the fuel, instead of increasing the regression rate, but this does not lead to satisfactory solutions in most cases. To increase the surface area, the chamber volume also needs to be increased or a multi-port design is employed. Each option has their own disadvantages, but they both lead to a lower filling factor (empty volume) and consequent higher structural mass fraction of the rocket vehicle.

While increasing the regression rate has been historically difficult, a new class of liquefying (or "melting") fuels gained traction in the late 1990s that could solve this issue. When heated, these fuels turn from solid to a low viscosity liquid state before being vaporized, unlike HTPB or PBAN. From this group of fuels, paraffin wax stands out as ubiquitous. Stanford University pioneered the research into paraffin at the start of the millennium. It was first fired at Stanford University in 1998 with oxygen as the oxidizer [7]. Paraffin in HREs burns at a regression rate several times higher than common polymeric fuels, thanks to a process called *entrainment*, in which fuel droplets from the melting surface of the fuel are released into the flow, vaporizing away from the blowing effect of the fuel boundary layer - effectively increasing the fuel mass flow rate. The dominant constraint on the regression rate ceases to be the rate of heat transfer to the fuel and becomes the viscosity and surface tension of the liquid fuel layer. Entrainment is the process represented on the magnified detail of Figure 2.1.

2.1.5 Performance

In terms of performance, hybrids also occupy the middle space between solids and liquids, although they are much less mature and less tested. One of the better developed hybrid propellant combination so far is liquid oxygen with HTPB fuel, which gives acceptable performance (its nominal I_{sp} of 280 seconds (see Equation 3.1) and low O/F ratio are comparable to the main orbital launch vehicles operating in the commercial market). However, HTPB is being dropped in favor of higher regression rate fuels, like paraffin, for high-thrust cryogenic booster stage applications without significant losses in efficiency, as shown by Figure 2.7. All-in-all, the paraffin and LOX combination can become a serious candidate to see widespread commercial use, as exposed further in section 2.2.

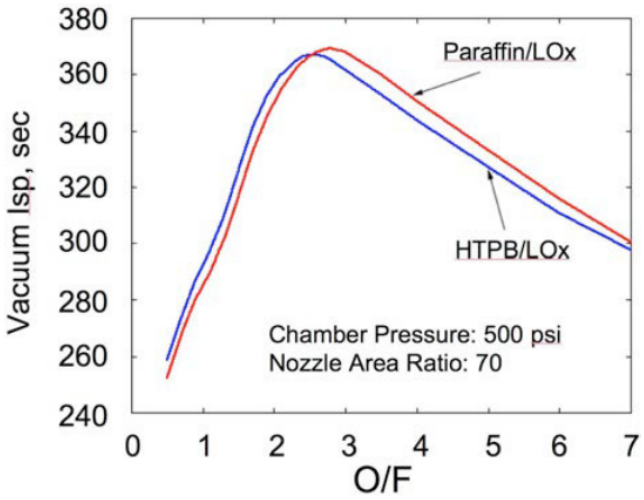


Figure 2.7: Depiction of the theoretical performance of liquid oxygen burning with paraffin and HTPB under a 500 psi chamber pressure. Source: [7].

N_2O and paraffin wax, not necessarily together, have seen widespread use in recent years among amateurs and universities due to the ease of handling and operation. The better performing pairs are metallized fuels with fluorine oxidizers, but these are expensive and/or dangerous, in many cases defeating the point of using hybrid propellants in the first place. The two together produce an I_{sp} of 248 seconds, with an oxidizer-to-fuel ratio of 8, as shown on the table of Figure 2.8.

2.2 Use Cases and Competitiveness

In section 2.1, the many peculiarities of HREs were discussed and compared to both liquid and solid propulsion. This section provides a more system-level overview of how those characteristics can come together to generate a useful product in the form of a hybrid-propelled rocket vehicle.

Figure 2.9 does a good job of stating the main strong points of hybrid propulsion in a visual manner. As evidenced by the gray area of the Figure, rockets and spaceplanes using HREs are well suited for missions requiring throttling and restart capabilities. This can be the case with in-space operations, for which hybrid upper stages with a storable oxidizer would be appropriate and better performing than solid kick motors like the Star 37 [14]. Another case, which some players in the industry are already looking into [5], is reusable first stages using hybrid propellants. The propulsive recovery of a rocket booster requires an in-flight re-ignition right before landing and fine throttle control, which a solid motor cannot do.

One may argue that reusable rocket vehicles have been landing propulsively for the past years with liquid engines, so HREs do not bring anything new to that area. Nevertheless, when safety and simplicity (of operation and development) are more important than performance, hybrids are an interesting solution. Launch start-ups, for example, might leverage the safety aspect of HREs to develop their technology quicker and with a lower initial investment, eventually even using HREs as a stepping stone to liquid propulsion.

Some hybrid propellant combinations have a higher volumetric specific impulse than most liquid or

Performance of hybrid propellants				
Fuel	Oxidizer	Optimum O/F	I_{sp}, s	c^*, m/s
HTPB	LOx	1.9	280	1820
PMM($C_5H_8O_2$)	LOx	1.5	259	1661
HTPB	N_2O	7.1	247	1604
HTPB	N_2O_4	3.5	258	1663
HTPB	RFNA	4.3	247	1591
HTPB	FLOx(OF_2)	3.3	314	2042
Li/LiH/HTPB	FLOx(OF_2)	2.8	326	2118
PE	LOx	2.5	279	1791
PE	N_2O	8	247	1600
Paraffin	LOx	2.5	281	1804
Paraffin	N_2O	8	248	1606
Paraffin	N_2O_4	4	248	1667
HTPB/Al (40%)	LOx	1.1	274	1757
HTPB/Al (40%)	N_2O	3.5	252	1637
HTPB/Al (40%)	N_2O_4	1.7	261	1679
HTPB/Al (60%)	FLOx (OF_2)	2.5	312	2006
Cellulose($C_6H_{10}O_5$)	GOx	1	247	1572
Carbon	Air	11.3	184	1224
Carbon	LOx	1.9	249	1599
Carbon	N_2O	6.3	236	1522
Cryogenic hybrids				
Pentane (s)	LOx	2.7	279	1789
CH_4 (s)	LOx	3	291	1871
CH_4 (s)/Be (36%)	LOx	1.3	306	1918
NH_3 (s)/Be (36%)	LOx	0.47	307	1967
Reverse hybrids				
JP-4	AN	17	216	1418
JP-4	AP	9.1	235	1526
JP-4	NP	3.6	259	1669

Note: JP-4 is kerosene and nearly all of these combinations in the table have been tested at least at laboratory scale.

Figure 2.8: Comparison of performance of several hybrid propellant combinations for a chamber pressure of 3.5 MPa and sea-level exit pressure. Source: [21].

solid alternatives, making hybrid rockets more compact for the same mission than alternative designs (even if, in terms of gravimetric specific impulse, the performance of liquid propellants is usually higher). This could have potential military applications, for example. Nitrous oxide is a particular good fit for low-mass, small sounding rockets, since the self-pressurization allows tight mass budgets to be feasible and it can be stored for some time.

However, HREs have a few drawbacks worth mentioning. First of all, the overall Technology Readiness Level (TRL) of these systems is not exactly on par with their liquid and solid counterparts, which have been studied a lot more in the past. Fortunately, the recent renewed interest in hybrids, driven also by environmental and safety concerns [22], is starting to make up for it. Still, new HREs designs often suffer from combustion instability problems and their complicated internal ballistics make designing larger motors difficult.

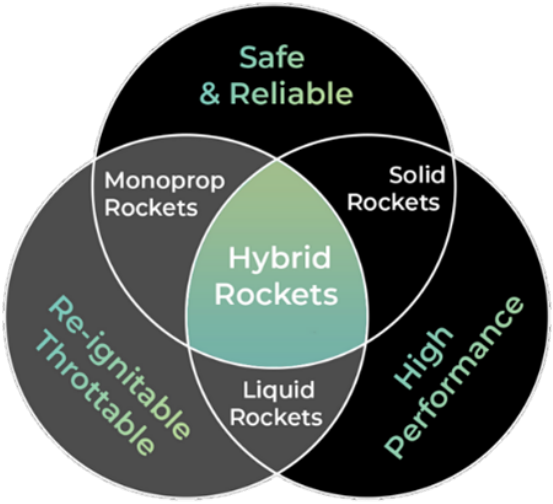


Figure 2.9: HyImpulse's visualisation of the combined strong points of solid and liquid propellant rockets (and monopropellant configurations as well) in hybrid rockets, as presented on their website [4].

Furthermore, controlling the mixture ratio of the propellants is not straightforward. As the solid propellant grain is consumed, the burning area changes and, in order to maintain the fuel to oxidizer ratio, the injector valve would have to be adjusted in accordance. A similar perturbation of the mixture also happens when throttling, but in this case, adapting the burning area to the changing liquid propellant flow is not possible. The issue is that any less-than-optimal mixture ratio degrades the specific impulse of the engine, making it less efficient, as becomes evident when looking at Figure 2.7.

Looking forward, as the TRL of hybrids improve, these disadvantages are dissipating and the advantages are starting to make hybrid propulsion a relevant and cost-effective alternative, especially with respect to solid motors. The German company HyImpulse Technologies [4] is evidence of this trend.

This start-up was formed in 2018, stemming from a research group at the DLR (German Aerospace Center), whose founders' first experience with hybrid rockets had been in student sounding rocket projects [23, 18]. Their aim is to differentiate themselves from the rest of the European small launcher start-ups by using hybrid propulsion in their vehicles. They are developing a modular three-stage small launch vehicle which they claim will be able to put 500 kg payloads into a dedicated orbit.

The rocket will use a proprietary paraffin-based fuel, to which a minor percentage of additives is added, and liquid oxygen as the oxidizer. This promises to be an environmentally friendly combination, as well as relatively easy and safe to be handled during the manufacturing process and launch operations, one of the major hurdles being the handling of liquid oxygen at cryogenic temperatures and its loading into the vehicle [24].

Also born in the 21st century, Virgin Galactic is another spaceflight company that is exploiting the benefits of hybrid propulsion in the main thruster of their suborbital manned spaceplanes. Virgin Galactic is currently conducting tourism flights, like the one depicted on Figure 2.10, to the edge of space using passenger spaceplanes whose design evolved from SpaceShipOne, the craft that flew the first privately funded manned spaceflight, back in 2004 [19].

Figure 2.10: Virgin Galactic's SpaceShipTwo soaring upwards under thrust from its powerful hybrid motor. Source: *marsscientific.com*

In fact, human spaceflight is another realm where hybrid propulsion can thrive, thanks to its overall safety aspects, especially for suborbital "hops" that don't require too high performance. SpaceShipTwo is propelled by nitrous oxide but the oxidizer tank does not rely on self-pressurization [25].

Chapter 3

Launch Vehicle Multidisciplinary Design Optimization

Conceptualizing a rocket vehicle is a complex design challenge that benefits from breaking down the full system into smaller, less complex parts, modelling each one separately and analyzing their interactions. In this spirit, rocket design is usually split into various interdependent subsystems, like the propulsion system, structure, aerodynamic and control surfaces, etc.

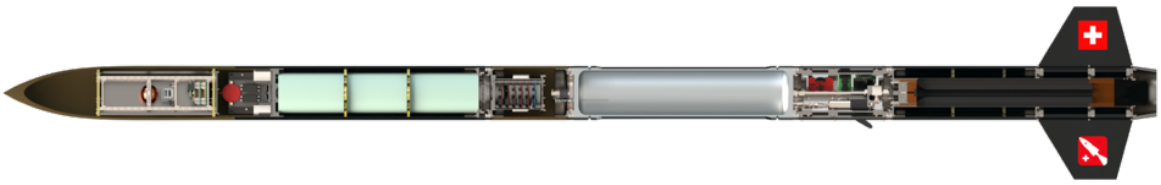


Figure 3.1: Sliced view of the sounding rocket Bella Lui II, from the Swiss EPFL University team. The team won an award at the EuRoC 2021 competition with this vehicle. Source: [26].

For optimization (or simply design improvement) purposes, all these subsystems can be improved independently, but, given the great level of mutual interdependence in an aerospace system such as this, a multidisciplinary optimization approach can leverage the analysis of subsystem interactions to provide better results [27].

3.1 Mission Requirements

There's another subsystem that deserves a mention - the payload. No transportation vehicle is made without a purpose and a rocket's goal is usually to deliver a payload to a destination. Like luggage on an automobile, the payload is not needed for the rocket to function, except when it is also playing a structural and/or aerodynamic role. However, the payload is instrumental in defining the mission requirements for a launch vehicle [16], together with its destination - an altitude, for a suborbital flight, or an orbit.

The capability of a rocket vehicle is usually measured in terms of how much payload mass it can place at a given altitude, at a given velocity. For orbital launch vehicles, this can translate to being capable of putting a 500kg payload into Low Earth Orbit (LEO), as is the case with HyImpulse's SL1 rocket [4], for example. For suborbital vehicles, it can translate to reaching a given apogee (with null vertical velocity). In either situation, the payload must be accelerated to a precise velocity vector at a precise point in space (or in the atmosphere).

For the purposes of conceptual design validation and optimization, a full set of orbital elements isn't necessarily required or relevant. Thus, it is possible to model a launch vehicle's mission objective in a simplified way, using a target scalar velocity, with the appropriate flight path angle, at the target altitude at which that velocity must be verified. These three variables are, thus, constraints on the optimization algorithm being defined here.

Additionally, a commercial rocket designer and manufacturer is not interested in developing a vehicle that fulfils the mission requirements whatever the cost. For it to be worth an investment, it must result in an economically viable product. Accurately estimating the cost of rocket launchers is not a straightforward subject and many of the available models are ill-suited to the twenty-first century reality of commercial small launch vehicles, as evidenced by Drenthe [28]. However, the gross lift-off weight (GLOW) of the vehicle is a central variable for estimating cost, and as such it can be minimized in the design as a simpler alternative to estimating and optimizing the cost.

Cost is an important factor, but environmental and operational concerns also drive the need for minimizing the mass of the vehicle. Likewise, minimizing the amount of propellant consumed directly minimizes pollutant emissions generated from using the propellant or from producing it. The best indicator of propellant consumption efficiency is the specific impulse,

$$I_{sp} = \frac{I_{tot}}{m_p\, g_0} = \frac{c}{g_0} \tag{3.1}$$

which measures, in seconds, the amount of time a rocket can produce thrust of magnitude equal to the total weight of its propellant under standard Earth gravity, g_0 [12]. c is the effective exhaust velocity of the engine. Thus, maximizing the specific impulse and minimizing the GLOW should be objectives of any rocket design optimization process.

3.2 Multidisciplinary Design Optimization Problems

According to Martins and Ning [29], optimization is the concept of "finding the *best* possible solution by changing variables that can be controlled, often subject to constraints". Multidisciplinary Design Optimization (MDO), then, is the engineering discipline of combining numerical optimization methods with complex system design, such as aircraft wings, chemical reactors or whole vehicles, which due to their nature need the contribution of various disciplines.

While design in engineering is usually an iterative process, MDO requires less time to complete, while increasing the final design performance by ensuring its optimality [29]. Formulating an optimization

problem begins by assessing which are the most relevant design variables, that is, the ones which have a stronger influence on the design outcome. One way of evaluating this is through sensitivity analysis.

Sensitivity analysis examines the impact on the outputs of small variations, whether deliberate changes or uncertainties, in the input (design) variables. One method of sensitivity analysis, finite differencing, is used to compute gradients in the design space by introducing small perturbations on each design variable while maintaining all the other variables constant [30]. The computation of these gradients is often referred to, in an engineering optimization context, simply as "sensitivity analysis".

With this technique, it is possible to assess which variables have the greatest impact on a given area of the design space and, as a consequence, which are the most relevant to choose as design variables for the optimization problem.

3.2.1 Design Variables

The design variables are the inputs to the optimizer and they, in conjunction with any constant system parameters, describe the system undergoing optimization. Together, the design variables form the design vector $\boldsymbol{x} = [x_1, x_2, x_3, ..., x_n]$. These variables are handled by the optimizer, although some optimization architectures require the user to input an initial design vector, as a starting point for the optimization process. In practice, this can be an existing design that is intended to be optimized or just a feasible placeholder, from which the optimizer may deviate considerably [29].

The design variables need to be independent from one another [29]. For example, when choosing a design variable, the designer should make sure: that variable is not a linear combination of other variables; that it is not determined by simulation parameters; or that it is not directly related to the value of other design variables. Otherwise, the optimizer will not be able to iterate on all components of $\boldsymbol{x}$ freely, resulting in a poorly defined problem.

In optimization problems with physical significance, as is the case with rocket vehicle design, setting boundaries on the variables values is sometimes necessary. Mass, length and cost are just some examples of properties that loose meaning if, during the optimization process, they become negative. Thus, a hard-coded lower bound with the value zero (or some positive real number close to zero) is customary for such properties. Setting a lower and upper bound for each variable may also be instrumental in the prevention of computational errors in the models, which depend, of course, on the specific problem definition and implementation.

If a user is looking for an optimal design in a known region of the design space, it can make sense to constrain the bounds as much as possible around the target region. However, if the user is too conservative on the bounds definition, it can have the opposite effect, hindering the optimizer from exploring the design space further for better solutions [29].

3.2.2 Objective Function

The *design outcome* mentioned in the beginning of section 3.2 must be translated into a mathematical formulation in order to be quantified and optimized. That formulation is called the objective function,

$f(x)$, and, by convention, its value is to be minimized in the optimization process. When the intent is to maximize the objective function (e.g., maximize the mission duration of a satellite in a decaying Low Earth Orbit (LEO)), the problem can be formulated, equivalently, as

$$max[f(x)] = -min[f(-x)],$$

as stated by Martins and Ning [29]. The objective function can be computed, assuming it is defined, in every point of the design space. The point with the lowest $f(x)$ value is the optimum point.

The objective function choice is not as trivial as it is vital. Its simplest form is a scalar value, whose reduction should denote an improvement or optimization of the system, even though that simple scalar may be the result of a complex sequence of equations or algorithms instead of a single, simple variable. There may also be more than one relevant variable to minimize in a given problem, resulting in a trade-off situation where the designer may need to decide which is the best variable to minimize. Alternatively, the optimization designer may resort to multi-objective optimization, which returns a set of possible solutions instead of a single one [29]. This may be preferable to obtaining a single solution in cases, such as this work, in which the models being analysed are not very precise and a group of preliminary solutions are chosen to later be analysed by more advanced tools. This is usually the case with feasibility studies or concept generation.

More methods exist (see Martins and Ning [29]), but perhaps the most common approach to multi-objective optimization is to use a weighted sum function as the objective - in essence, a linear combination of the n objectives such that the new and only objective function is given by

$$\bar{f} = \sum_{i}^{n} w_i \, f_i(x) \tag{3.2}$$

where w_i is a value between 0 and 1 and the sum of all the n weights is equal to one. If there are only two objectives, then the weighted sum can be simply written as

$$\bar{f} = w f_1(x) + (1 - w) f_2(x). \tag{3.3}$$

The weighted sum is, in fact, not the best method by any reasonable metric other than being widely used [29]. Another widespread method is the epsilon-constraint method, which is as simple to use and provides better results in the presence of non-convex Pareto sets [31]. It consists of turning all of the objectives except one into optimization constraints, which are discussed in section 3.2.3, and then computing the solution for various different constraint values.

In multi-objective optimization, since the objective function is no longer a single variable but actually a linear combination of multiple variables, this means there are now infinite optimal solutions, assuming all variables are continuous. A new optimality concept is used, the Pareto optimality, based on the principle of *domination*. A solution design point is said to dominate another if it is better than the other on all objective functions [31]. To better visualize this, it is usual to plot a so called Pareto front, like the one on Figure 3.2. The figure shows the values of the two objective functions plotted for each design point.

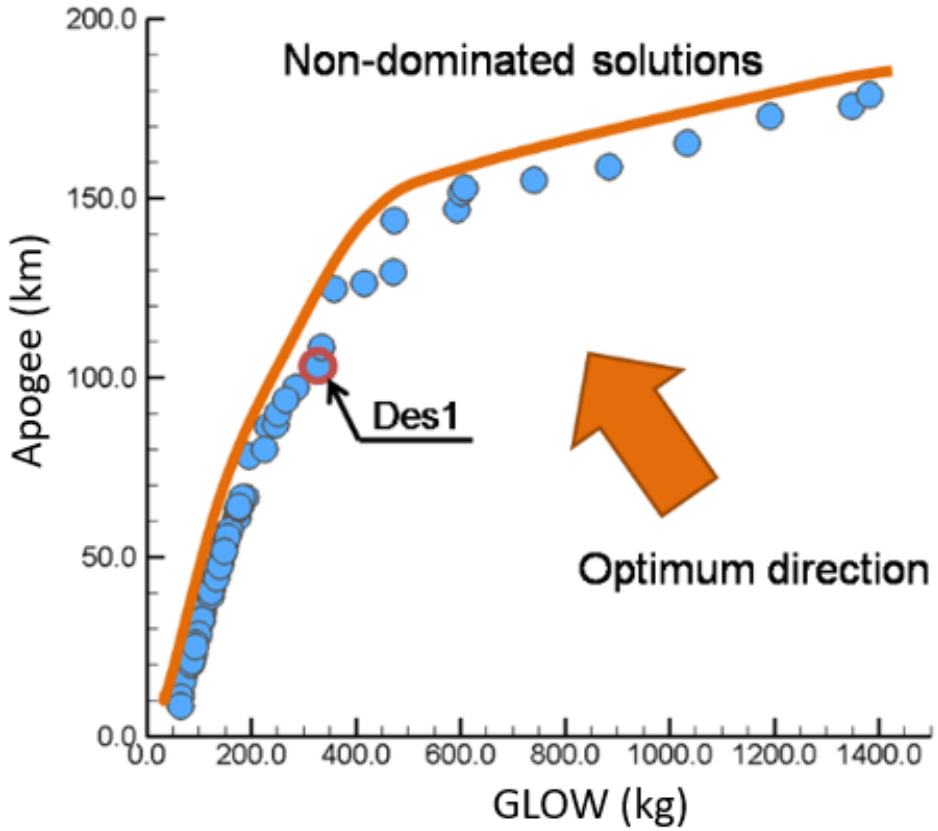


Figure 3.2: Example of a Pareto front plot from a rocket system optimization problem, where apogee (vertical axis) and GLOW (horizontal axis) are the two objectives. Source: [32].

In this case, the problem was to optimize a sounding rocket, where the GLOW and the apogee were the two objectives. We can see that the non-dominated design points follow a more or less asymptotic behavior, meaning that a trade-off relationship between altitude and mass is present. Dominated design points are not represented, only non-dominated ones - these form the Pareto front. In case of more than two objectives optimization, the visualization of the Pareto front is not as straightforward as a 2D plot.

3.2.3 Constraints

As discussed in section 3.2.1, the design variables are usually bounded to ensure they maintain physical meaning. However, this alone does not grant physical feasibility to the solution. To make sure the optimized solution conforms to the problem specifications, physical realism and user requirements, a set of constraints ought to be implemented within an MDO architecture.

Just like with defining variables and objectives, constraint formulation is an art [27] that requires some level of experience with optimization. Constraints are computed using variables in algorithms with varying degrees of complexity, just like with objective function computation. Constraints also play an important role in most MDO architectures by ensuring the convergence of different simulation variables. To this end, there are equality constraints, of the form $g(x) = 0$, where the constraint function g is required to be equal (within a certain numerical tolerance) to a fixed value; and inequality constraints, of the form $g(x) \leq 0$, where the constraint function can not go over (or below, depending on the inequality direction convention)

a certain value [29].

If a solution satisfies all constraints, it is said to be in the feasible region of the design space. On par with optimality, feasibility is the parameter that guides the optimizer in the search for a solution. Solution A may have a lower objective function value than solution B, but if A does not comply with all the constraints, then the optimizer will opt for solution B if it sits in the feasible region. Hence, adding more constraints to an optimization problem will necessarily decrease the optimality of the solutions, or, at best, produce no change [29]. Over-constraining a problem is, then, a real possibility if the designer is too conservative or if the problem, under the considered assumptions and models, is unfeasible.

3.2.4 State and Coupling Variables

In multidisciplinary optimization, the optimizer runs more than one discipline analysis, which translate to mathematical models that calculate some output response variables, usually known as state variables. State variables originating from discipline i are represented by the vector $\bar{\boldsymbol{y}}_i$ [33]. The usefulness of MDO is that it simulates discipline interaction, through the exchange of information between the different disciplines, resulting in the manipulation of the same variables by different models in a way that every discipline agrees with each other in the end - a process called convergence.

The exchange of information can be done in multiple different ways, but it usually relies on coupling variables, as illustrated on the example Figure 3.3. Coupling variables, $\boldsymbol{y}$, are a transformed subset of the state variables (not necessarily all variables computed by a discipline are used to model interdisciplinary interactions). For MDO architectures running disciplines in parallel, multiple copies of the same coupling variables may be needed. These copies sometimes act in the same way as design variables, being input into the disciplines and controlled by the optimizer, and can be considered "target" variables, $\hat{\boldsymbol{y}}$.

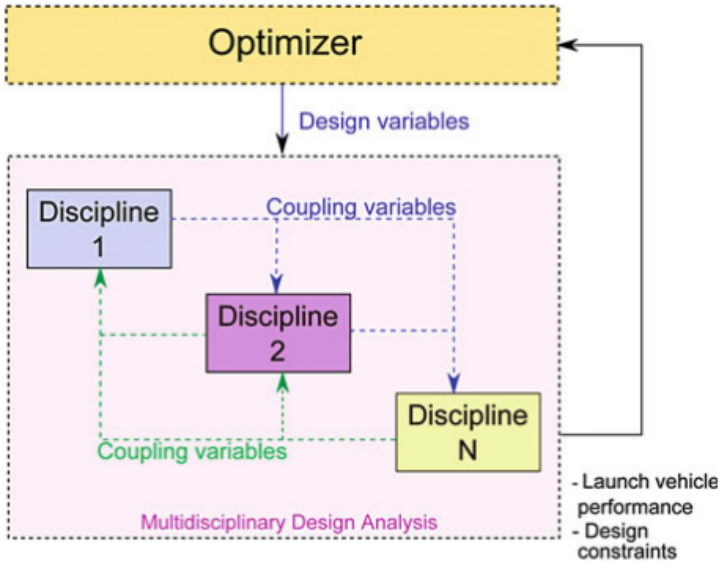


Figure 3.3: Example of an MDO design process with coupling variables in feed-forward and feedback coupling. Source: [34].

3.2.5 Optimization Algorithms

When solving a multidisciplinary design optimization problem, a designer has to go through a multi-step process. First, it's the problem definition, as previously described on the beginning of section 3.2. Choosing variables, objectives, and so on. Secondly, classifying the problem is important, because not all optimization problems are created equal and, to be solved efficiently (or solved at all), different problems require different strategies [29].

Variable definition, being the a start point for problem definition, can also be the start point for problem classification. Whether variables are discrete or continuous greatly influences the difficulty of the problem and the solving strategies at our disposal, with discrete being the more restrictive case [29]. Mixed (some continuous, some discrete variables) problems are also possible.

Like continuity, convexity has a similar influence on the difficulty of a problem and the possible paths to solving it. The most well-known optimization methods can solve convex and non-convex problems alike. However, if a problem is known to be convex, it is wise to use methods that are specially adapted to solve such types of problems. Convex problems require the objective function and inequality constraints to be convex as well, resulting in problems that are, generally-speaking, simpler than most aerospace optimization problems. Linear and quadratic programming techniques are two examples of algorithms that take advantage of the convexity of a problem to solve it [29].

The third property to consider, after convexity and continuity, is differentiability. Whether it is possible to compute the derivatives of the functions of interest or not, the choice of optimization strategies becomes more or less limited. Optimization algorithms supported on derivative computation and analysis are usually called gradient-based, as opposed to gradient-free algorithms [29].

Under the right circumstances - objective function smoothness, for example -, gradient-based algorithms will converge to an optimum point using less computational resources (in other words, quicker) than gradient-free methods [30]. However, gradient-based methods are not good tools to find the global optimum solution of a problem, even when paired with techniques like *multistart*, where a local search method is run several times from different points in the design space, in an attempt to find better solutions [29]. Note, though, the distinction between the gradient-based or gradient-free classification and the local or global classification of optimization strategies. Even though gradient-based methods usually obtain local solutions, these concepts are not interchangeable.

Gradient-free methods, sometimes called zeroth order methods, are better suited for when there are few design variables, when the design space is multimodal (non-convex), or when gradients cannot be obtained, either because the functions have plenty of discontinuities or because the models under optimization are "black-boxes", which means the optimizer only has access to the model's outputs, not its inner workings or governing equations [29][30].

Gradient-based methods generally use some type of sensitivity analysis to model the problem, approximately, in the vicinity of the current point. This is the strategy the Sequential Quadratic Programming (SQP) method employs to solve constrained nonlinear C^2 continuous problems [27]. Using finite differences, SQP estimates and solves a local quadratic programming problem to find the next point in the optimization process. Thus, SQP is, as its designation implies, an iterative, sequential resolution of local quadratic

programming problems. Finite differencing introduces errors due to the size of the step used to compute derivatives, but it can be used to compute derivatives of black box models [29].

While SQP is one of the most useful gradient-based algorithms for aerospace applications, genetic algorithms are the most well-known evolutionary algorithms, which are a subset of gradient-free optimization methods characterized by a global search using heuristics [29]. Genetic algorithms (GA) are based on a population of initial points. The larger the initial population, the better the design space will be covered. Each iteration of a GA is called a generation - the nomenclature, like the algorithms themselves, are inspired by natural reproduction and selection processes [29].

The population in a GA, from generation to generation, changes, or evolves, using three different processes. The exact details of how these processes are implemented vary widely between different GA. First, *selection*: the best design points, according to some sort of heuristic or evaluation, are selected to reproduce, to the detriment of the non-selected. Then, there's *crossover* between "parent" design points, producing new points that combine characteristics from the parents. Finally, the third process is *mutation*, which translates into a more or less random introduction of new characteristics to diversify the population of design points.

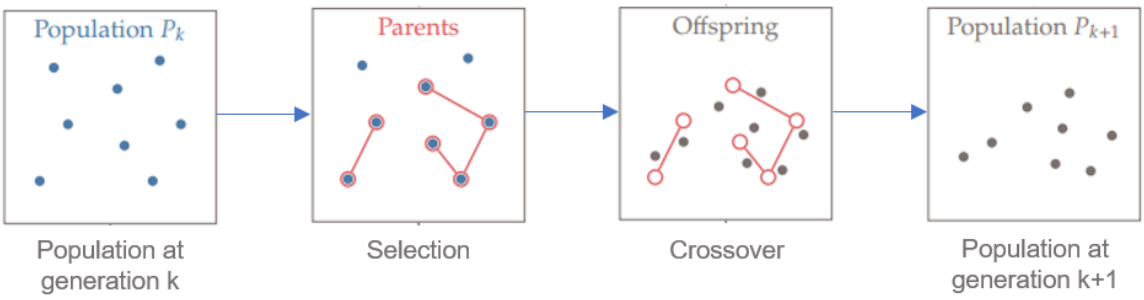


Figure 3.4: Visual interpretation of the evolution of a population from one generation to the next. The mutation process is not represented. Adapted from [29].

Genetic algorithms are able to search effectively in non-convex spaces, where the objective function is not linear nor simple, a clear advantage with relation to gradient based alternatives. They are not hard to implement [27], although they require a large population and many generations to be effective, rendering their use impractical in many engineering scenarios.

Finally, the last step in the design of an MDO problem framework is the choice of an architecture to support the solver algorithm. This will be the focus of section 3.2.6.

3.2.6 MDO Architectures

The MDO architecture is simply the way the optimization problem is structured and solved. This section focuses on monolithic architectures only, meaning a single optimization is being solved instead of distributing it into various smaller problems. To easily visualize MDO architectures, along with discipline interdependency and process flow, a type of diagram named the Extended Design Structure Matrix (XDSM), proposed by Lambe and Martins [35], is used.

Monolithic architectures are very similar to one another, with the differences being related to the

handling of design and coupling variables by the optimizer. In fact, every MDO formulation has its roots in the fundamental multidisciplinary optimization architecture, called *All at Once* (AAO) [33]. Optimization problems can be specified and formulated in a formal mathematical declaration. For example, an AAO problem can be mathematically formulated as

$$
\begin{aligned}
\text{minimize} \quad & f(\boldsymbol{x}, \boldsymbol{y}) \\
\text{with respect to} \quad & \boldsymbol{x}, \boldsymbol{y}, \bar{\boldsymbol{y}}, \hat{\boldsymbol{y}} \\
\text{subject to} \quad & \boldsymbol{c}_0(\boldsymbol{x}, \boldsymbol{y}) \leq 0 \\
& \boldsymbol{c}_i(\boldsymbol{x}_0, \boldsymbol{x}_i, \boldsymbol{y}_i) \leq 0 \quad \text{for } i = 1,..., N \\
& \boldsymbol{c}_i^c = \hat{\boldsymbol{y}}_i - \boldsymbol{y}_i = 0 \quad \text{for } i = 1,..., N \\
& \boldsymbol{R}_i(\boldsymbol{x}_0, \boldsymbol{x}_i, \boldsymbol{y}_i, \bar{\boldsymbol{y}}_i, \hat{\boldsymbol{y}}_{j \neq i}) = 0 \quad \text{for } i = 1,..., N,
\end{aligned}
\tag{3.4}
$$

where $\boldsymbol{c}$ represent constraint functions, $\boldsymbol{c}^c$ are consistency constraint functions and $\boldsymbol{R}_i$ correspond to discipline i governing equations' residuals. Its XDSM representation can be seen of Figure 3.5.

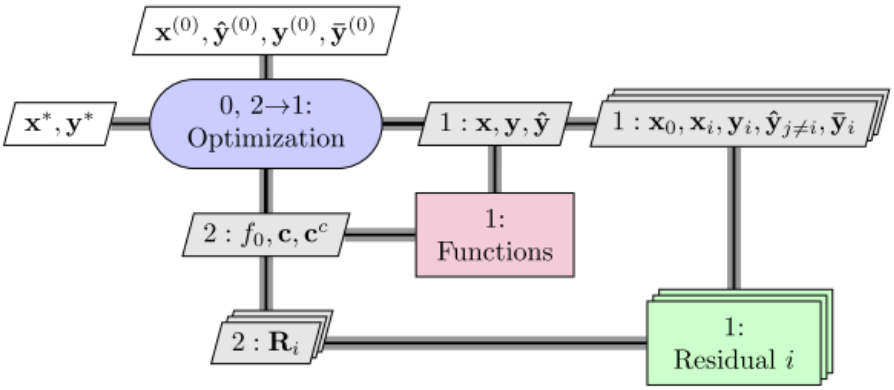


Figure 3.5: XDSM of an All at Once MDO architecture. All functions and discipline residuals can be evaluated in parallel. Source: [33].

In reality, AAO is a primordial monolithic formulation and, despite being the root of all others, isn't really used in practice [33]. It can be very easily simplified by removing the copies of the state variables and, thus, the consistency constraints, which are not really necessary because the optimizer is handling all the variable modifications. This simplified version of AAO is called Simultaneous Analysis and Design (SAND).

The main issue with SAND (and AAO) is the computation of the disciplines' governing equations residuals, $\boldsymbol{R}_i$. These methods use the discipline analysis equations as constraints, making the residuals of these equations and their derivatives necessary and accessible. This fact precludes the use of "black box"-type models in the disciplinary analysis, because these only reveal the coupling variables [33].

The Individual Discipline Feasible (IDF) architecture is answering this concern. IDF is obtained from AAO by introducing discipline solvers instead of burdening the optimizer with discipline analysis constraints. This removes the need for residuals knowledge. Instead, the method makes use of the optimizer-controlled coupling variable copies $\hat{\boldsymbol{y}}$, which were eliminated in SAND. State and actual coupling variables become a function of design variables and coupling variable copies and are determined in and by the discipline

solvers. This way, after discipline evaluation, the comparison between the optimizer's "guesses" for the coupling variables and the actual modeled coupling variables can be made. Mathematically, to assess their convergence, IDF resorts to the consistency constraints $\boldsymbol{c}^c$.

The optimization problem formulation under an IDF architecture is, thus,

$$\begin{aligned}
&\text{minimize} && f_0(\boldsymbol{x}, \boldsymbol{y}(\boldsymbol{x}, \hat{\boldsymbol{y}})) \\
&\text{with respect to} && \boldsymbol{x}, \hat{\boldsymbol{y}} \\
&\text{subject to} && \boldsymbol{c}_0(\boldsymbol{x}, \boldsymbol{y}(\boldsymbol{x}, \hat{\boldsymbol{y}})) \leq 0 \\
& && \boldsymbol{c}_i(\boldsymbol{x}_0, \boldsymbol{x}_i, \boldsymbol{y}_i(\boldsymbol{x}_0, \boldsymbol{x}_i, \hat{\boldsymbol{y}}_{j\neq i})) \leq 0 \quad \text{for } i = 1,\ldots, N \\
& && \boldsymbol{c}_i^c = \hat{\boldsymbol{y}}_i - \boldsymbol{y}_i(\boldsymbol{x}_0, \boldsymbol{x}_i, \hat{\boldsymbol{y}}_{j\neq i})) = 0 \quad \text{for } i = 1,\ldots, N.
\end{aligned} \tag{3.5}$$

The discipline analysis constraints, $\boldsymbol{R}_i$, present in the AAO formulation, have been removed, along with all state variables $\bar{\boldsymbol{y}}$. As observable in its XDSM representation, which can be seen on Figure 3.6, the discipline analysis can be run in parallel, considering they only depend on variables $\boldsymbol{x}$ and $\hat{\boldsymbol{y}}$, determined by the optimizer in the loop's initial step [33]. On the other hand, the consistency constraints that ensure coupling variable convergence need to be evaluated after model analysis, alongside with the other constraints and objective function, thereupon closing the optimization loop.

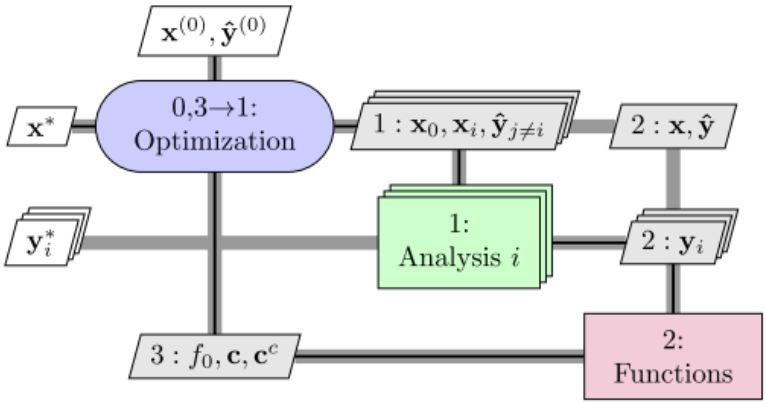


Figure 3.6: XDSM of an Individual Discipline Feasible MDO architecture. Source: [33].

Compared to AAO/SAND, IDF problems are smaller and simpler, from the optimizer's point of view. However, this does not say much, because AAO/SAND problems handle all the variables, in an intrusive manner, making it the largest MDO architecture [29]. If the number of coupling variables is high and this cannot be mitigated by changing the position of certain models in between disciplines, to minimize variable transfers, then the size of the optimization may hamper its efficiency.

Another consequence of the IDF approach is that disciplines are solved independently from one another in a given iteration, meaning they could be, in theory, modelling completely dissimilar systems at a given time, before the optimization algorithm assures convergence of the equality constraints [29]. This stands in stark contrast to the Multidisciplinary Feasible (MDF) architecture.

Of all classic monolithic architectures, Multidisciplinary Feasible generates the smallest optimization architectures. It is also the most suitable for heavily coupled interdependent systems such as those developed in the aerospace industry [27].

MDF, formulated as

$$\begin{aligned} \text{minimize} \quad & f_0(\boldsymbol{x}, \boldsymbol{y}(\boldsymbol{x}, \boldsymbol{y})) \\ \text{with respect to} \quad & \boldsymbol{x} \\ \text{subject to} \quad & \boldsymbol{c}_0(\boldsymbol{x}, \boldsymbol{y}(\boldsymbol{x}, \boldsymbol{y})) \leq 0 \\ & \boldsymbol{c}_i(\boldsymbol{x}_0, \boldsymbol{x}_i, \boldsymbol{y}_i(\boldsymbol{x}_0, \boldsymbol{x}_i, \boldsymbol{y}_{j\neq i})) \leq 0 \quad \text{for } i = 1,\ldots, N, \end{aligned} \tag{3.6}$$

does not use consistency constraints, like IDF, nor does it need access to the discipline state variables and residuals, like SAND. Figure 3.7 presents a representation of the MDF architecture, which includes a Gauss-Seidel Multidisciplinary Analysis (MDA) loop, initiated by a driver component. The MDA is responsible for making the coupling variables consistent between the various disciplines on each iteration. Instead of feeding copies of the coupling variables to the different disciplines and running them in parallel, the MDF runs disciplines sequentially, feeding the component 1 output into component 2, as shown in Figure 3.7. This process is computationally demanding, converging slowly, and harder to implement than just a simple IDF [29].

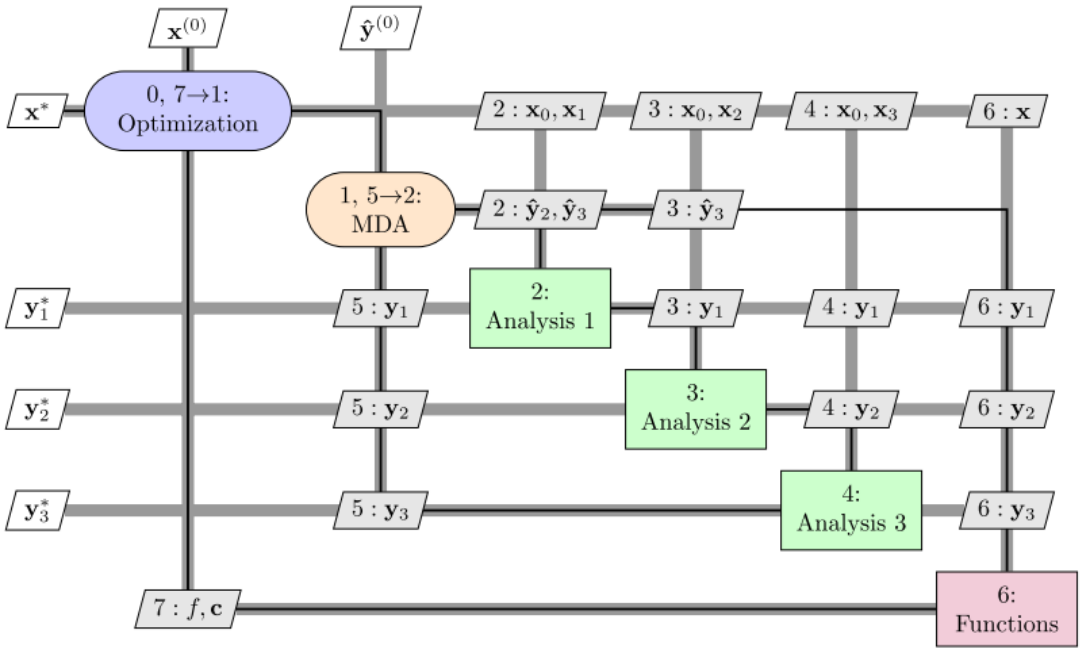


Figure 3.7: XDSM of a Multidisciplinary Feasible MDO architecture with three disciplines. Source: [33].

On the other hand, the optimization loop is very simple, with only the design vector, objective function and design constraints being controlled by the optimizer [33]. A clear advantage of this strategy is that every iteration of an MDF is physically meaningful, that is, if an optimization run were to stop prematurely, the resulting design point would be realistic, albeit not necessarily feasible with regards to the design constraints and, of course, not optimal.

Chapter 4

Hybrid Rocket Optimization Tool Implementation

4.1 Launch Vehicle Modelling

4.1.1 Vehicle Subsystems

The vehicle models weren't too much changed from the work of Yamada [9]. The Mass and Sizing discipline depends on semi-empirical relationships to compute the masses of all subsystems. Some, like the oxidizer tank and the combustion chamber masses are computed analytically, resorting to simple geometric models.

Although avionics and recovery subsystem models were implemented by Yamada [9] in his work, it was decided to abandon those detailed specifications and substitute the two subsystems by a single one, referred to as *components*. There were some reasons for this:

- In an attempt to reduce the computational resources needed by the program and its execution time, merging those two subsystems into one meant that two less coupling variables had to be handled by the optimizer and functions (two mass and two length variables turned into one of each).
- Users may not intend to include a parachute recovery system on their design.
- Avionics systems are hard to model empirically or analytically, both in mass and volume. Accurately modelling electronic components goes beyond the scope of a conceptual level analysis, which is this work intention.
- It is significantly easier to build accurate empirical models for non-structural *components* as a whole in a rocket than to develop several different models for each component or subsystem imaginable.

4.1.2 Propulsion System

The hybrid propulsion system is simulated using the framework developed by Klammer [10] and Yamada [9], wherein three control volumes are modelled to represent the oxidizer tank, the combustion chamber and the nozzle, as illustrated by Figure 4.1. Mass and energy fluxes are simulated from the oxidizer tank, through the chamber and to the nozzle. In addition to these three components, only one other is modelled - the injector. Otherwise, the tool does not model ignition devices or cooling systems. There is also no tank pressurization system being modelled because the tool was developed under the assumption that nitrous oxide would be the oxidizer of choice, which is self-pressurizing. This does not mean the tool cannot function with other oxidizers, but some hard-coded parameters need to be changed accordingly and the user has to design an adequate pressurization system separately, if desired.

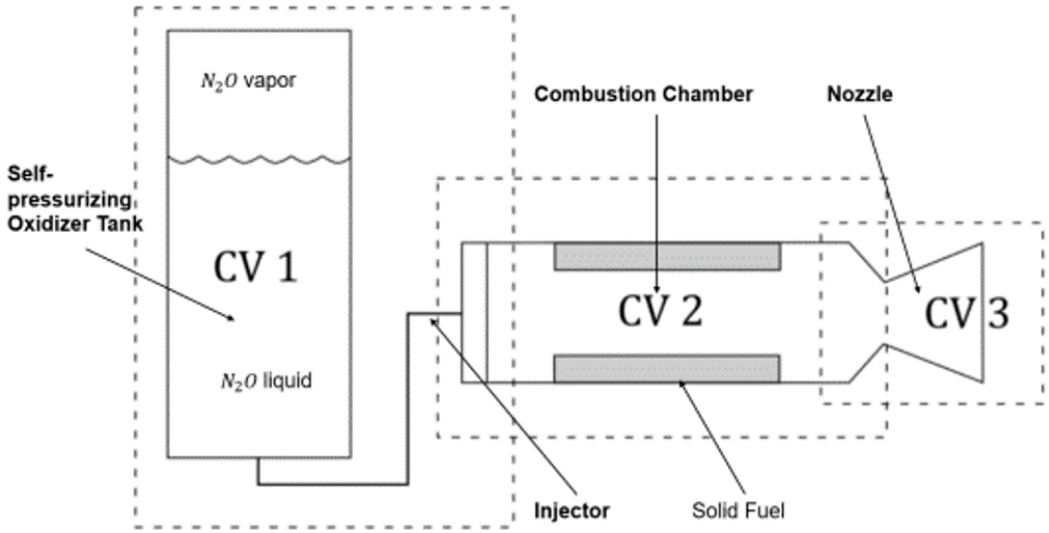


Figure 4.1: The internal conditions and mass flows of each of the three control volumes are modelled at each time step in the MATLAB framework. Source: [10].

The oxidizer is stored in both liquid and gaseous phases inside the tank. Its two-phase equilibrium is modelled by the sub-function *thermoSat*, using a table of thermodynamic properties of the oxidizer at saturated conditions, from the melting point up to the critical temperature. This way, knowing the oxidizer vapor pressure (which corresponds to the tank pressure, p_{OT}) and the pressure loss through the injector and feed system, p_{feed}, the possibility of oxidizer flow to the combustion chamber is verified. If the combustion chamber pressure, p_{CC}, is such that

$$p_{OT} - p_{feed} - p_{CC} > 0 \tag{4.1}$$

then the pressure in the tank is enough to sustain the oxidizer flowing to the chamber. This is quantified by the oxidizer mass flow rate, $\dot{m}_{ox}$, and calculated by means of equation 4.2, under the assumption of incompressible single-phase fluid flow.

$$\dot{m}_{ox} = C_{inj}\sqrt{2\rho_d(p_{OT} - p_{feed} - p_{CC})} \tag{4.2}$$

where C_{inj} is the effective injection area, obtained by multiplying the injector area by a discharge coefficient, and ρ_d is the discharge fluid density.

The fluid in the combustion chamber is assumed to be homogeneous, with the combustion process modelled as instantaneous and complete. To assure this is a reasonable assumption, the combustion chamber is sized so that the fuel grain sits between two empty portions of the chamber of combined length greater than the external radius of the vehicle, d_{ext} - the pre-combustion and the aft combustion chambers. They improve oxidizer atomization and combustion completeness, respectively.

One limitation of the software tool is how it only simulates cylindrical grain port shapes. This facilitates the simulation of the fuel regression, since it only depends on the regression rate as shown on Figure 4.2. Although this rate can vary axially along the length of the grain, the fuel is assumed to regress uniformly in the code as the uncertainty associated with modelling those differences would make the additional accuracy not be worth the added computational cost [10].

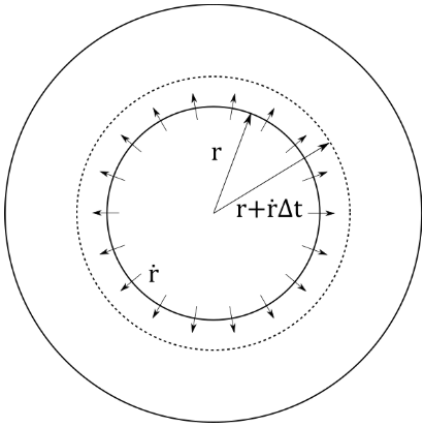


Figure 4.2: The fuel grain regression model implemented considers the regression rate is uniform on both the axial and the radial directions. $r = r_{port}$ is the radius of the fuel port, while $\dot{r}$ is the regression rate of the fuel surface. Source: [10].

Usually, in the literature [13], the fuel regression rate is given by

$$\dot{r} = aG_{ox}^{n} \tag{4.3}$$

where a and n are empirically-fitted parameters highly dependent on the propellant choices and G_{ox} is the oxidizer mass flux through the port, obtained iteratively by dividing the oxidizer mass flow rate by the port section area, which gradually increases as the fuel is consumed. The fuel mass flow rate, $\dot{m}_f$ is also given iteratively by multiplying the fuel port surface area by the regression rate and the fuel density:

$$\dot{m}_f = 2\pi r_{port} L_f \dot{r} \rho_f \tag{4.4}$$

As explained by Klammer [10], the regression rate's a and n empirical parameters originally used in the simulation tool were determined considering the full average mass flux through the port, G (calculated including $\dot{m}_f$), in the equation instead of G_{ox}, so that is still implemented and parameters are set accordingly. It is fairly simple to switch the code to use the oxidizer flux only, if necessary.

The oxidizer mass flow rate, $\dot{m}_{ox}$, is one of the most relevant parameters in hybrid rocketry, because G_{ox} depends on it and fuel flow, in turn, depends on G_{ox}. As explained in section 2.1.2, this allows the thrust of a hybrid rocket to be controlled by limiting oxidizer flow by means of a valve, for example. Although this feature is not modelled in the present work, it is something that could be added in the future.

Finally, the flow behaviour in the nozzle section is simulated using ideal nozzle theory, under the assumption of isentropic, ideal gas flow. The nozzle's throat diameter and area ratio are design variables supplied by the optimizer. The combustion chamber pressure and the oxidizer-to-fuel ratio are used as inputs to search for the other corresponding chamber properties in look-up tables, which data is compiled from NASA's Chemical Equilibrium with Applications (CEA) program [36] for the appropriate propellant combination. CEA does not have many of the hybrid propellants on its default library, but they can be manually introduced if their composition and formation enthalpy are known.

4.2 Trajectory and Control

Klammer [10] and Yamada [9] have previously included in their hybrid rocket optimization codes a 1-degree of freedom (DOF) trajectory simulation function, first developed in 2016 by Michael Pearson [10]. However, when the scope of this work was defined, it became clear that a 1-DOF model would soon be insufficient as the program evolves. Hence, developing a new three-dimensional model was made a priority.

Section 3.1 explored how the mission requirements under consideration for the purposes of booster stage and sounding rocket conceptual design optimization are the payload mass, the apogee (or booster separation altitude), and the velocity vector, $\boldsymbol{v}$ - or, simpler still, the speed, v, and the flight path angle, γ, which is illustrated on Figure 4.3 and defined as

$$\gamma = \arctan \frac{\dot{z}}{\dot{x}} \tag{4.5}$$

where $\dot{x}$ and $\dot{z}$ are, respectively, the horizontal and vertical components of the velocity vector [37].

In fact, the new 3-DOF trajectory discipline provides the needed outputs to optimize a vehicle for those mission requirements. The payload mass is a design parameter, while the vehicle's position and velocity vectors as a function of time, in cartesian coordinates, are calculated and placed in the state structure of the simulation program to be passed along to other functions. As inputs, the trajectory module needs, besides the design variables and general program parameters, the thrust and propellant mass data as a function of time, the launch altitude and angle, and general geometric and mass values for the computation of aerodynamic forces and moments of inertia.

The vehicle's motion variables are computed on a *while* loop. At the end of each iteration, the variables

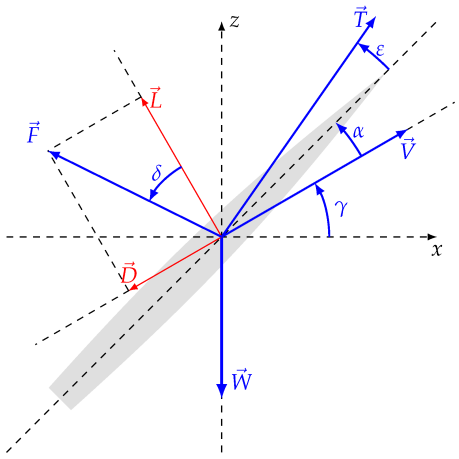


Figure 4.3: A diagram of a rocket flying in the positive x and z directions. The forces acting on the vehicle are shown (thrust, $\boldsymbol{T}$, weight, $\boldsymbol{W}$, aerodynamic force, $\boldsymbol{F}$) as well as its velocity vector, $\boldsymbol{v}$, angle of attack α, flight path angle, γ, and thrust vectoring angle, ϵ. $\boldsymbol{F}$ is split between its two components, drag and lift, illustrated in red. Source: [37].

are stored, and the simulation time is incremented. The loop breaks when the vehicle returns to ground altitude, after following a ballistic trajectory from wherever propulsion ceased.

The vehicle altitude $z(t)$ is used at each time step to calculate atmospheric properties, namely air pressure, density and local speed of sound, through the 1976 COESA model MATLAB function. The model is extrapolated up to an altitude of 150 km, above which the program considers a vacuum. Likewise, the altitude, Mach number and angle of attack are used to search for the corresponding lift and drag coefficients, C_L and C_D, from look-up tables estimated for a sounding rocket by the Digital DATCOM script, an implementation of the United States Air Force DATCOM [38].

The following phase is the pitch guidance, which relevance is dependent on the type of vehicle and mission. A sounding rocket should not require pitch guidance, nor is it likely to have thrust vectoring control (TVC) or aerodynamic control surfaces to follow any active guidance. But for orbital-class vehicles, an adequate trajectory profile, usually called a gravity turn, is an optimum way of reaching the desired altitudes and speeds. For these vehicles, until the propulsion system cuts-off, a guidance algorithm is run to determine the desired pitch, θ_d, for the vehicle.

This is achieved in three phases. First, once the vehicle clears the launch tower, θ_d is increased linearly, with the linear rate being handled by the optimizer. Then, after a certain time, the desired pitch decreases exponentially, to allow for a smooth transition into the third phase, the gravity turn itself, during which the pitch guidance law is simply $\theta_d = \gamma$.

θ_d is then introduced as a reference into a simple proportional feedback controller, where it is subtracted by the simulated pitch angle, θ, obtained from the dynamics equations in the previous time step. This difference is multiplied by a simple negative gain, ranging from 0 to -1, to obtain ϵ, the TVC angle. This goes through a saturation condition to ensure it does not exceed the maximum allowed TVC angle magnitude, which is a simulation parameter.

Finally, the equations of motion, through which the accelerations in x and z are calculated, are adapted from Equations (21a-b) of Campos and Gil [37].

4.3 Optimization Problem Setup

In this section, the suborbital hybrid rocket optimization problem is formulated and discussed. When following the optimization problem solving process previously discussed in section 3.2, the first step is to define the problem, its variables, objectives and constraints.

The scope of the problem led to the division of the rocket system modelling into four different disciplines: Propulsion, Mass and Sizing, Aerodynamics, and Trajectory and Control. Each discipline is represented by a different component in the MDO architecture, as illustrated by Figure 4.5.

4.3.1 Problem Variables

The four disciplines share data between them in the form of coupling variables. Table 4.1 shows the four disciplines and their input and output variables, in no particular order. The order of the four disciplines, on the other hand, is not arbitrary. The order in which the MDO components are executed determines how the exchange of data between them is done. Notice how all coupling variable are fed forward to the next disciplines in the loop.

Table 4.1: Input and output variables of each discipline, according to the chosen problem formulation. Non-variable inputs (simulation parameters) have been omitted. State variables not used outside the function they are computed in are also not represented.

Propulsion		Mass and Sizing		Aerodynamics		Trajectory and Control	
Inputs	Outputs	Inputs	Outputs	Inputs	Outputs	Inputs	Outputs
V_{OT}	T	m_f	m_{rocket}	L_{cone}	x_{OT}	d_{ext}	h
C_{inj}	m_f	m_{ox}	L_{body}	$m_{components}$	CG	m_{rocket}	v
L_f	m_{ox}	$r_{CC_{in}}$	L_{rocket}	m_{fins}	SM	SM	$accel$
$d_{port,0}$	$r_{CC_{in}}$	r_{OT}	d_{ext}	m_{noz}		CG	
d_{th}	I_{sp}	L_{CC}	m_{CC}	m_{cone}		L_{rocket}	
$A/A*$	p_{CC}	D_{fin}	m_{OT}	L_{body}		T	
	p_{OT}	B_{fin}	m_{body}	d_{ext}		$A/A*$	
	G	b_{fin}	L_{OT}	m_{body}		d_{th}	
		m_{add}	$L_{components}$	m_{OT}		$gain$	
		$m_{addcone}$	$m_{components}$	m_{CC}		Δt_{po}	
		V_{OT}	m_{fins}	m_{rocket}		$\Delta\theta$	
			m_{noz}	L_{OT}			
			m_{cone}	$L_{components}$			
			V_{cone}	D_{fin}			
			L_{cone}	B_{fin}			
				b_{fin}			
				L_{CC}			
				m_{ox}			
				m_f			

From all variables included in Table 4.1, some are chosen as system design variables. For example, all six Propulsion inputs are design variables, whose values are selected iteratively by the optimizer. This is intentional. This way, Propulsion does not depend on the other disciplines directly, which makes it adequate to be the first discipline to be executed in the optimization loop. Table 4.2 lists all design variables and their purpose, while Figure 4.4 showcases the relationship between each one and different parts of a rocket.

Table 4.2: Description of the problem's sixteen design variables. The last column indicates which optimization disciplines take which design variables as input.

x	Design Variable	Description	Discipline(s)
1	V_{OT}	Oxidizer Tank Volume, in m^3	Propulsion, Mass and Sizing
2	C_{inj}	Effective Injection Area, in m^2	Propulsion
3	L_f	Fuel Grain Length, in m	Propulsion
4	$d_{port,0}$	Initial Fuel Port Diameter, in m	Propulsion
5	d_{th}	Nozzle Throat Diameter, in m	Propulsion, Trajectory and Control
6	$A/A*$	Nozzle Area Ratio, dimensionless	Propulsion, Trajectory and Control
7	L_{CC}	Combustion Chamber Length, in m	Mass and Sizing, Aerodynamics
8	D_{fin}	Fin Span, in m	Mass and Sizing, Aerodynamics
9	R_{OT}	Oxidizer Tank Radius, in m	Mass and Sizing
10	B_{fin}	Fin Root Chord, in m	Mass and Sizing, Aerodynamics
11	b_{fin}	Fin Tip Chord, in m	Mass and Sizing, Aerodynamics
12	m_{add}	Ballast Mass in the Fuselage, in kg	Mass and Sizing
13	$m_{addcone}$	Ballast Mass in the Nosecone, in kg	Mass and Sizing
14	$gain$	Pitch Controller Gain, dimensionless	Trajectory and Control
15	Δt_{po}	Pitch-Over Maneuver Duration, in s	Trajectory and Control
16	$\Delta\theta$	Pitch-Over Angle, in rad	Trajectory and Control

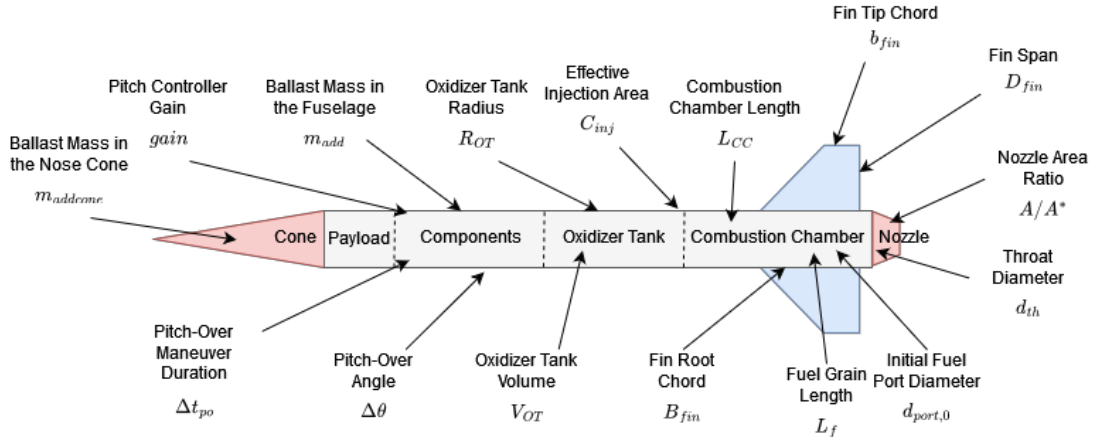


Figure 4.4: A diagram of a rocket showing the optimization problem's sixteen design variables. These are directly controlled by the optimizer.

Observing all the variables, it is easy to conclude the problem is continuous. There are some discrete quantities in the simulation, such as the number of fins, for example, but this is considered a constant simulation parameter ($n_{fins} = 4$) that can be altered by the user.

All design variables are constrained by upper and lower bounds, to ensure they maintain their physical significance. All masses and lengths are lower-bounded by small positive values to ensure a null length or mass does not generate errors on the code. In fact, on most variables, a null length would not make sense. An exception could be the fins' dimensions, with $D_{fin} = 0$ meaning the vehicle has no fins, which could be compensated by the thrust vectoring control system. However, that possibility has not been tested on the code and the stability constraints (see section 4.3.3) would need to be omitted.

4.3.2 Multi-objective Function

The rationale behind the choice of an objective function for the optimization of a rocket launch vehicle has been layed out previously in section 3.1, reaching the conclusion that the vehicle's mass, m_{rocket}, and the motor's specific impulse, I_{sp}, are two interesting parameters to optimize since they convey cost. These two variables were, then, chosen as the constituents of the optimization problem multi-objective function.

Note, however, that a lower mass does not always translate to a lower cost in engineering, as sometimes the difficulty of miniaturization and the manufacturing or procuring of small components inverts the cost relationship [29]. Even so, the author does not expect the mass-cost relationship of this system to invert itself given the design envelope of this problem. Nevertheless, this objective function should be viewed for what it is - an approximated model for cost - and an actual cost model should be employed in the future, if viable.

Since the intention is to maximize I_{sp} and minimize m_{rocket}, introducing these variables into the weighted sum multi-objective function from equation 3.3 will result in

$$\bar{f} = -w\frac{I_{sp}}{E_1} + (1-w)\frac{m_{rocket}}{E_2}. \tag{4.6}$$

The minus sign on the first operand is to comply with the minimization convention, under which the optimizer will try to minimize I_{sp} if we do not change its sign. E_1 and E_2 are two optional constants that can be introduced to normalize the two objectives. Choosing E_1 and E_2 as the expected value of each corresponding variable can be a good choice, as it will balance the influence of each term.

The value of the weight w can be chosen by the user. Higher values will grant more importance to the specific impulse, resulting in a higher engine performance but, perhaps, a heavier vehicle. Lower w values will have the tendency to achieve the opposite.

4.3.3 Constraints

Constraints must be put in place to ensure the simulation model corresponds has a correspondence with physical reality. Furthermore, the definition of constraints is particularly dependent on the architecture choice. If the MDO architecture employed is an IDF, which was briefly explained in section 3.2.6, then the program will need to include a set of consistency constraints, c^c, one for each coupling variable copy. Likewise, it is not always clear whether an objective should actually be a constraint and vice-versa. In the case of ϵ-constraint methods, for example, all objectives become constraints at some point.

For this particular architecture, which is explained in further detail in section 4.3.4, there are no consistency constraints. However, the mission requirements pointed out in section 3.1 - target altitude and target velocity at apogee - are introduced as equality constraints. The rationale behind this decision is that the vehicle should be design to achieve a very precise apogee and/or payload insertion point. Having an equality constraint for the altitude is also practical for the design of sounding rockets for the university-level student competitions, such as Spaceport America Cup or EuRoC, which require very precise apogees.

Thus, the equality constraints are given by

$$\begin{aligned} c_{eq,1} &= h(t^t) - h^t \\ c_{eq,2} &= v(t^t) - v^t \\ c_{eq,3} &= \gamma(t^t) - \gamma^t \end{aligned} \tag{4.7}$$

where h^t, v^t and γ^t are the target altitude, speed and flight path angle, respectively, and t^t is the time instant at which the target altitude is obtained. In practice, $c_{eq}(1)$ is self-evident and could be left out. However, for simple sounding rocket problems, whose goal is to reach a given altitude and come back down again, the equality constraints simplify to just

$$c_{eq} = h(t_{apogee}) - h^t. \tag{4.8}$$

Regarding the inequality constraints, it is possible to link each one of them to a physical limitation or phenomenon that isn't otherwise modelled. They are:

$$
\begin{aligned}
c_1 &= p_{cc} - 0.8\,p_{OT} - p_{feed} \leq 0 \\
c_2 &= G - 500 \leq 0 \\
c_3 &= accel - 100 \leq 0 \\
c_4 &= L_f + d_{ext} - L_{CC} \leq 0 \\
c_5 &= b_{fin} - 0.9\,B_{fin} \leq 0 \\
c_6 &= \frac{D_{fin}}{\frac{B_{fin}+b_{fin}}{2}} - 3 \\
c_7 &= m_{addcone} - V_{cone}\,\rho_{Pb} \leq 0 \\
c_8 &= -m_{components} \leq 0 \\
c_9 &= 1 - SM \leq 0 \\
c_{10} &= SM - 3 \leq 0 \\
c_{11} &= 27 - v_{off\ rail} \leq 0.
\end{aligned}
\tag{4.9}
$$

c_1 is a more conservative version of Equation 4.1, introduced to guarantee the combustion backflow failure mode does not happen with some margin of safety. Also to do with the propulsion system, c_2 puts the limit of 500 $kg/s.m^2$ on the mass flux through the grain port, since higher values are associated with combustion instability issues, according to Fraters and Cervone [39]. In a similar way, the third constraint holds the maximum acceleration of the vehicle below a safe value (in this case, approximately 10G), to prevent component or structural failure. The second-order effect of this constraint is to limit the thrust produced by the engine. The last constraint related to the propulsion system is c_4, which ensures the aft and pre-combustion chambers are well sized, using the vehicle external diameter as a reference.

c_5 and c_6 ensure the fins maintain a short trapezoidal shape that is both aerodynamically and structurally sound, while c_9 and c_{10} bound the static margin, SM, that they grant to the vehicle. The last constraint is also related to stability. It guarantees that passively stabilized vehicles launch out of their rail guide with enough speed to maintain stable flight.

c_8 is essential to prevent the mass of components, $m_{components}$, from being negative, which can happen, given its empirical determination model, if the variable is not controlled and bounded by the optimizer. Finally, c_7 is responsibly for making sure the added ballast mass on the nose cone, given by the design variable $m_{addcone}$, fits inside the nosecone volume if it were made of lead. It acts, then, as a density check.

4.3.4 MDO Architecture

The program was originally designed to use the gradient-based Sequential Quadratic Programming algorithm on an IDF architecture. Despite its multimodality, the models are generally smooth and SQP will converge to the local minimum rather quickly. SQP also has the great advantage of being able to

recover from function errors, which are very common in the Propulsion discipline. These errors happen when a variable, like the O/F ratio, for example, goes outside the bounds of the provided look-up tables, resulting in a NaN, the termination of that iteration and the choice of a new design point.

Now, the MDO archtecture resembles an MDF structure, as can be seen of Figure 4.5. The number of variables handled by the optimizer decreased substantially, from more than 30 (design plus state variables) to the current 16.

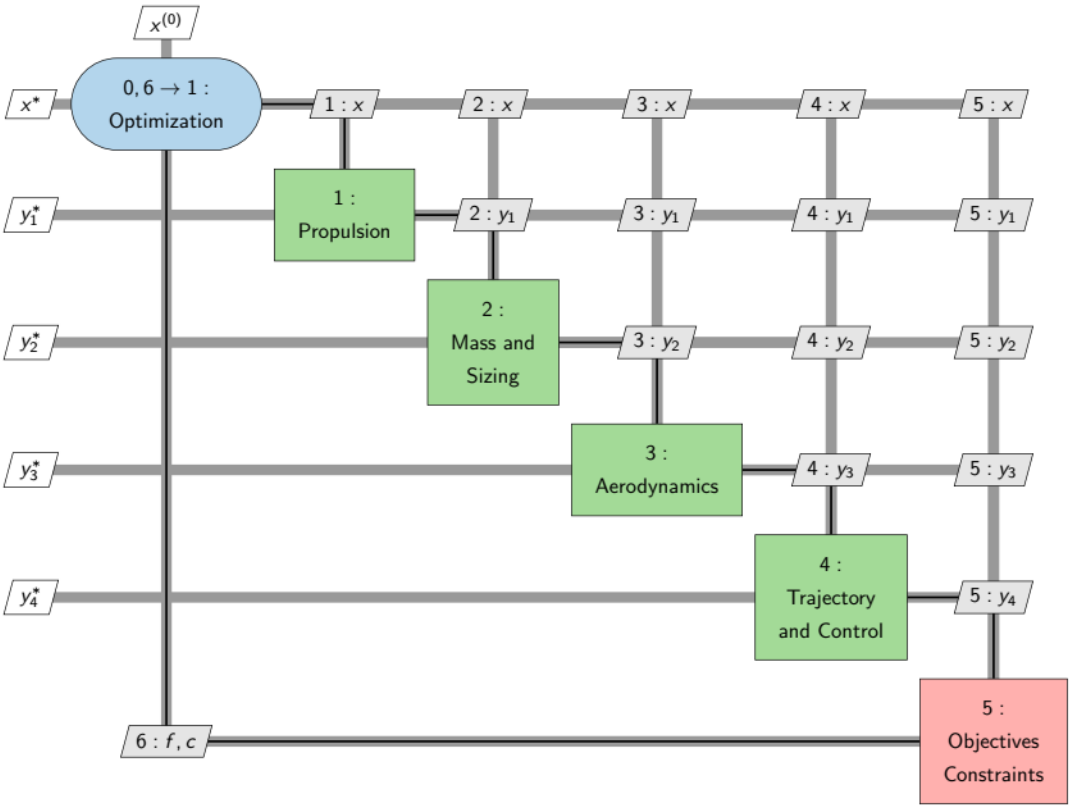


Figure 4.5: The XDSM representation of the program's simplified MDF architecture.

4.4 Numerical Methods

Throughout the MATLAB program, but especially in the Propulsion and Trajectory disciplines, some numerical approximations must be done. At each time step in the Trajectory and Propulsion function loops, ordinary differential equations are integrated using the Forward Euler method, an explicit first-order Runge-Kutta algorithm. The error of this method is proportional to the step size, so a compromise between speed and accuracy has to be arranged. The time step size, Δt, is an easily changeable parameter in the program.

As an example, the horizontal position, x, of the vehicle is determined, at each time step i, by using the Euler method such that

$$x(i+1) = x(i) + \dot{x}(i)h \tag{4.10}$$

where h is the step size, which is a fixed discipline parameter.

Linear interpolation is used to extract values from the look-up tables for the N2O saturation and CEA combustion chamber thermodynamic properties. Cubic spline interpolation is used on the trajectory discipline to get continuous thrust data. For the aerodynamic force coefficients, no interpolation is done. Rather, the nearest value is chosen to impede small numerical variations in the angle of attack to introduce a lift component on the vehicle. The root-finding bisection algorithm is employed to determine the temperature of the oxidizer tank and the Mach number of the nozzle flow. The secant method was previously used, but this was changed because the program would sometimes crash when the secant method tried to calculate the internal energy of a negative temperature. Using the bisection method guarantees these variables stay within bounds and converge.

Chapter 5

Results and Discussion

5.1 Validation

The program validation can be done at both the system and subsystem levels, by comparing the output results of each model with a known vehicle design or another tool with the same purpose. For the exercise to make sense, the inputs to the model should also correspond to that vehicle's characteristics. Some uncertainties will be present in the inputs, since the information for a vehicle may be hard to obtain precisely and some estimation work may be necessary.

5.1.1 Propulsion

The Propulsion system was not substantially altered since Yamada [9] analysed and validated its results against experimental data from three different small hybrid sounding rockets from the Spaceport America Cup competition, using the N_2O and paraffin-based fuel propellant combination. His simulations showed the thrust, pressure and tank temperature profiles agreed with the test data with an acceptable error for conceptual design purposes, given the uncertainties associated with the data. Table 5.1 shows the summary of the model errors obtained on those three comparisons with test data. Figure 5.1 shows the simulated thrust curve of Phoenix 1A compared to its experimental data, as an example. Table 5.1 and Figure 5.1 were extracted from [9].

Table 5.1: Relative errors between simulated variables and test data from three hybrid engines [9].

Rocket Name	Deliverance II	Boundless	Phoenix 1A
p_{OT} error	1.55 %	15 %	-
p_{CC} error	10.7 %	25.7 %	12.3 %
T error	7.55 %	32.8 %	12.2 %

Hence, the validation of the propulsion discipline was, this time, done with a different propellant combination. The HRE chosen for the comparison exercise was the one powering the Bella Lui II rocket (Figure 3.1), built by the EPFL Rocket Team, a student team from Switzerland that competed in the 2021 edition of EuRoC. The liquid oxidizer is still Nitrous Oxide, but its fuel is ABS plastic. The engine has a star-shaped center hole, which is not a shape modelled by the MATLAB program.

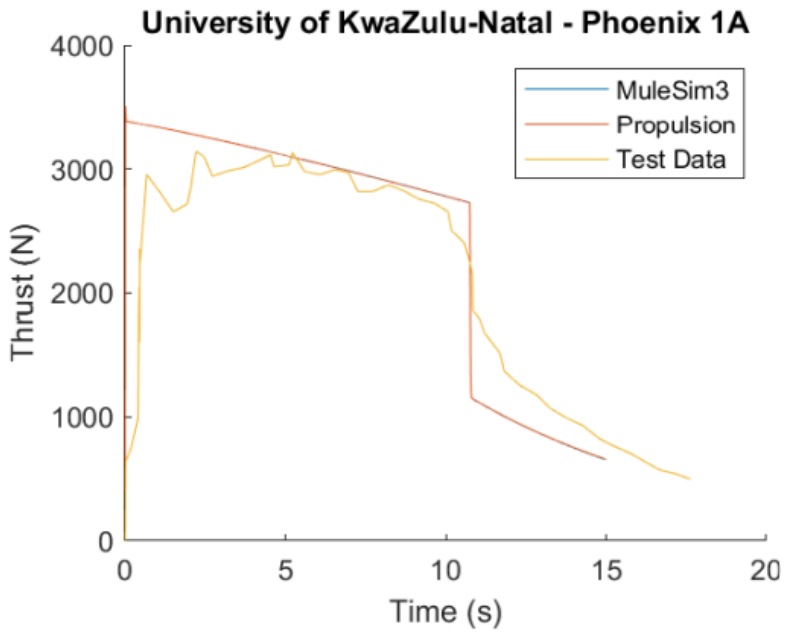
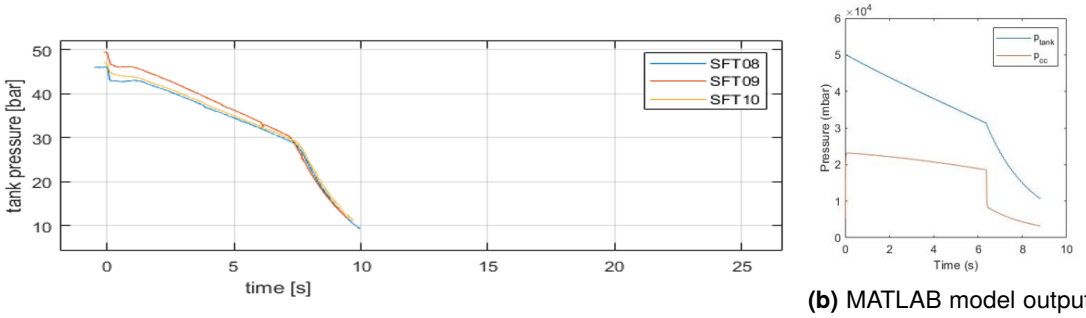


Figure 5.1: Comparison between the experimental data, in yellow, and the simulated data from the hybrid rocket Propulsion function, in orange. Source: [9].



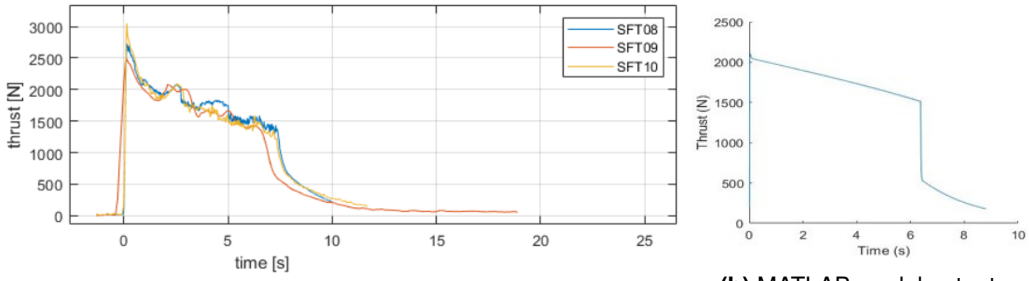
(a) Experimental data.

(b) MATLAB model output. Tank pressure is the blue curve.

Figure 5.2: Oxidizer tank pressure comparison between the experimental data from the EPFL Rocket Team static fire tests 8 through 10 and the simulated data from the hybrid rocket Propulsion function. Source of experimental data: [26].

On Figures 5.2 and 5.3, it's possible to observe very clearly the two phases of engine burn: during the first phase, the tank pressure drops slowly as liquid oxidizer exits the tank until it runs out; during the second phase, only gaseous oxidizer remains in the tank and both pressure and thrust drop as the combustion chamber is fed with lower pressure gaseous oxidizer. The most relevant propulsion parameters obtained from static fire tests of the Bella Lui II engine are listed on Table 5.2 and compared to the same parameters obtained from the MATLAB simulation.

As evidenced by Figure 5.3, there's a thrust peak at the beginning of the burn which isn't being modelled properly by the MATLAB tool. The initial peak in thrust is characteristic of star-shaped ports, which should be the main reason for disparity here, since the MATLAB tool can only simulate a single circular-shaped hole in the grain. This fact can help explain the larger error on the maximum thrust parameter. Otherwise, the errors are smaller and compatible with the scope of the tool. Another source of input uncertainty is the ABS plastic composition, since the exact mixture ratio of the three ingredients that make up the material



(a) Experimental data.

(b) MATLAB model output.

Figure 5.3: Thrust comparison between the experimental data from the EPFL Rocket Team static fire tests 8 through 10 and the simulated data from the hybrid rocket Propulsion function. Source of experimental data: [26].

Table 5.2: Relative errors between Bella Lui II's engine static fire test number 9 (SFT09) results and simulated data. Source of experimental data: [26].

Parameter	Relative Error
Average Thrust (liquid phase) [N]	+2.62 %
Avg. Thrust (liquid phase) [N]	-15.14 %
I_{sp} [s]	-0.18 %
Burn Duration (liquid phase) [s]	-15.99 %
Max. Thrust [N]	-23.38 %

used by the experimental tests may not coincide with the mixture introduced in the CEA code.

Nonetheless, the Propulsion discipline is considered still adequate for the end purposes of the software tool after the small changes and bug fixes implemented and using a fuel other than paraffin.

5.1.2 Mass & Sizing

The same Bella Lui II rocket, from a university team competing at the EuRoC competition, was used as baseline for the Mass & Sizing discipline validation. Like before, the component masses and rocket body diameter are taken from the team's report and webpage [26] and compared to the outputs generated by the Mass & Sizing discipline on the MATLAB program. The inputs used to run the model, namely the design and propulsion variables, were introduced manually and correspond to the Bella Lui II design. Table 5.3 collects and compares the variable values.

Table 5.3: Relative errors between Bella Lui II's component masses and external diameter and the same variables modelled in the MATLAB software. Source of experimental data: [26].

Parameter	Bella Lui II	MATLAB Simulation	Relative Error
Oxidizer Tank Mass [kg]	5.7	4.831	-17.6 %
Combustion Chamber Mass [kg]	3.3	2.487	-33.9 %
Diameter [m]	0.16	0.161	+2.5 %
Total Mass [kg]	42	44.161	+5.7 %

Again, some discrepancies are present. This was to be expected, since the Mass & Sizing is the lower fidelity model in the program, being defined partially by empirical multivariate regressions. The combustion chamber mass, however, is computed analytically, albeit that computation is dependent on a number of hard-coded parameters such as wall thickness and material density. These two alone, if

incorrect, can be enough to justify the mass difference. Furthermore, inner coatings or thermal insulation, structural elements, valves and other objects are not modelled by the tool but may be included in the mass value presented by the rocketry team.

5.1.3 Trajectory

Regarding the newly developed Trajectory discipline, it was first validated by checking that the flight simulations matched the results obtained on other similar tools, such as OpenRocket, for the same inputs. The maximum error found on single variable (apogee, maximum velocity, maximum acceleration, time to apogee) comparison was 5% when doing a results comparison exercise with the Open Rocket tool with low-altitude sounding rocket designs. The new 3-DOF code was also compared with the former 1-DOF script on sounding rocket (up and down) flights and the errors were negligible ($\leq$1%), indicating the tool retained its accuracy in vertical flight.

Validation was performed by comparison with experimental flight data as well. By running the trajectory script with experimental thrust and inertial data from a given test-flown rocket, it is possible to compare the results obtained with the data measured in flight. Table 5.4 shows the comparison between experimental data recorded from the flight computer of the Xi-16 rocket, launched on February 16, 2020, by Richard Nakka [40] and the corresponding simulated flight. Some uncertainties can help explain part of the error. The thrust curve of the engine corresponded to static test fire data, not from the flight itself. There is no guarantee the solid rocket motor from Xi-16 performed equally on both occasions. Additionally, the aerodynamic coefficients were estimated, using DATCOM, for another rocket, similar enough in shape and size so as the resulting coefficients to be similar, but certainly not identical.

Table 5.4: Relative errors between simulated flight and test flight data from the Xi-16 flight.

Variable	Apogee	Maximum Acceleration	Time to Apogee
MATLAB program	1034 m	185 m/s^2	14.15 s
Xi-16 flight data	1150 m	220 m/s^2	14.3 s
Relative error	10.1 %	15.9 %	1.1 %

To validate the trajectory script on a 3-DOF use-case, the same exercise was done, but comparing to the flight profile of SpaceX's Falcon 9 first stage, up until second stage separation, on the Crew Dragon Demo-1 mission, flown on March 2nd, 2019. Once more, input uncertainties are not negligible. The aerodynamic coefficients are difficult to estimate for the altitude and speed regimes in which the Falcon 9 flies. Thrust and trajectory data for the launch vehicle was extracted from internet sources [41] that recorded the flight telemetry from the company's live launch webcast.

The outputs from the simulation were compared with the flight data at two moments in time - 100 and 157 seconds after lift-off, as the latter was moments before second stage separation. The results are shown on Table 5.5 and a velocity and altitude comparison graph can be found in Figure 5.5. Once more, the results look favorable, showing small errors and demonstrating the tool's ability to simulate 3-DOF *gravity turns*, a maneuver essential to larger Launch Vehicle missions that insert payloads or second stages into orbit.

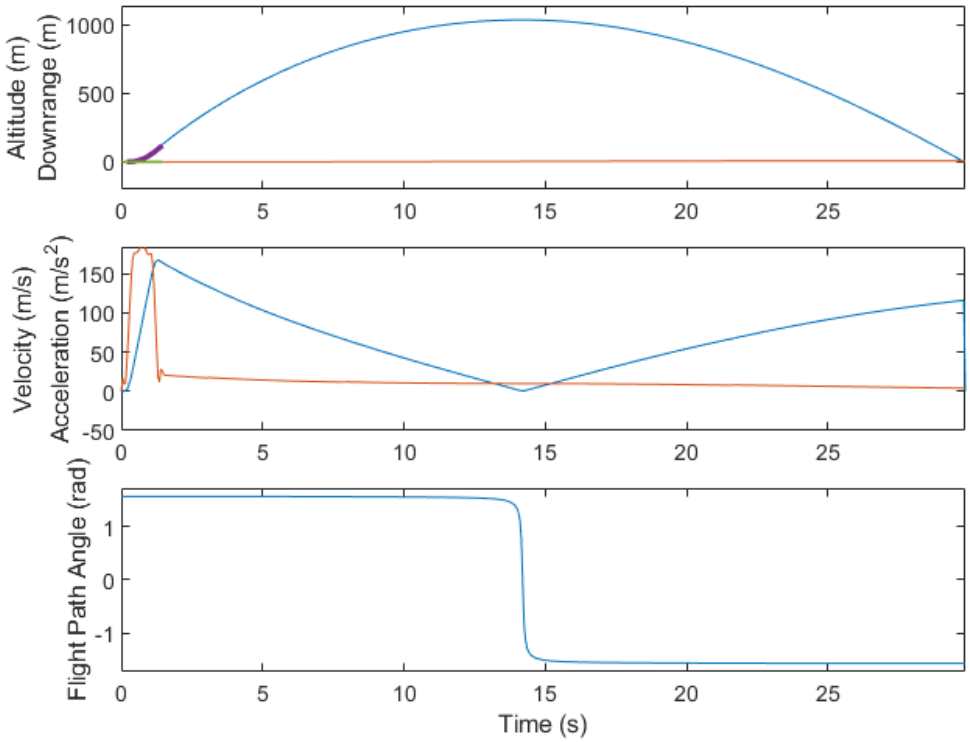


Figure 5.4: Simulation of the Xi-16 flight on the MATLAB trajectory code.

Table 5.5: Relative errors between the simulated flight and flight telemetry data from the Falcon 9 Crew Dragon DM-1 mission.

	Altitude (t =100s)	Velocity (t =100s)	Altitude (t =157s)	Velocity (t =157s)
MATLAB Simulation	25.76 km	0.68 km/s	81.82 km	1.79 km/s
Falcon 9 flight telemetry	25.80 km	0.70 km/s	84.00 km	1.88 km/s
Relative error	0.16 %	2.86 %	2.60 %	4.79 %

Regarding the multidisciplinary design optimization results, Yamada [9] conducted optimization studies for different objective function weights, for a low altitude sounding rocket. Recent results have confirmed the tool remains useful in that range, after the implementation of the new changes. Optimization studies for other flight envelopes and vehicles sizes are being compiled and preliminary results look favorable.

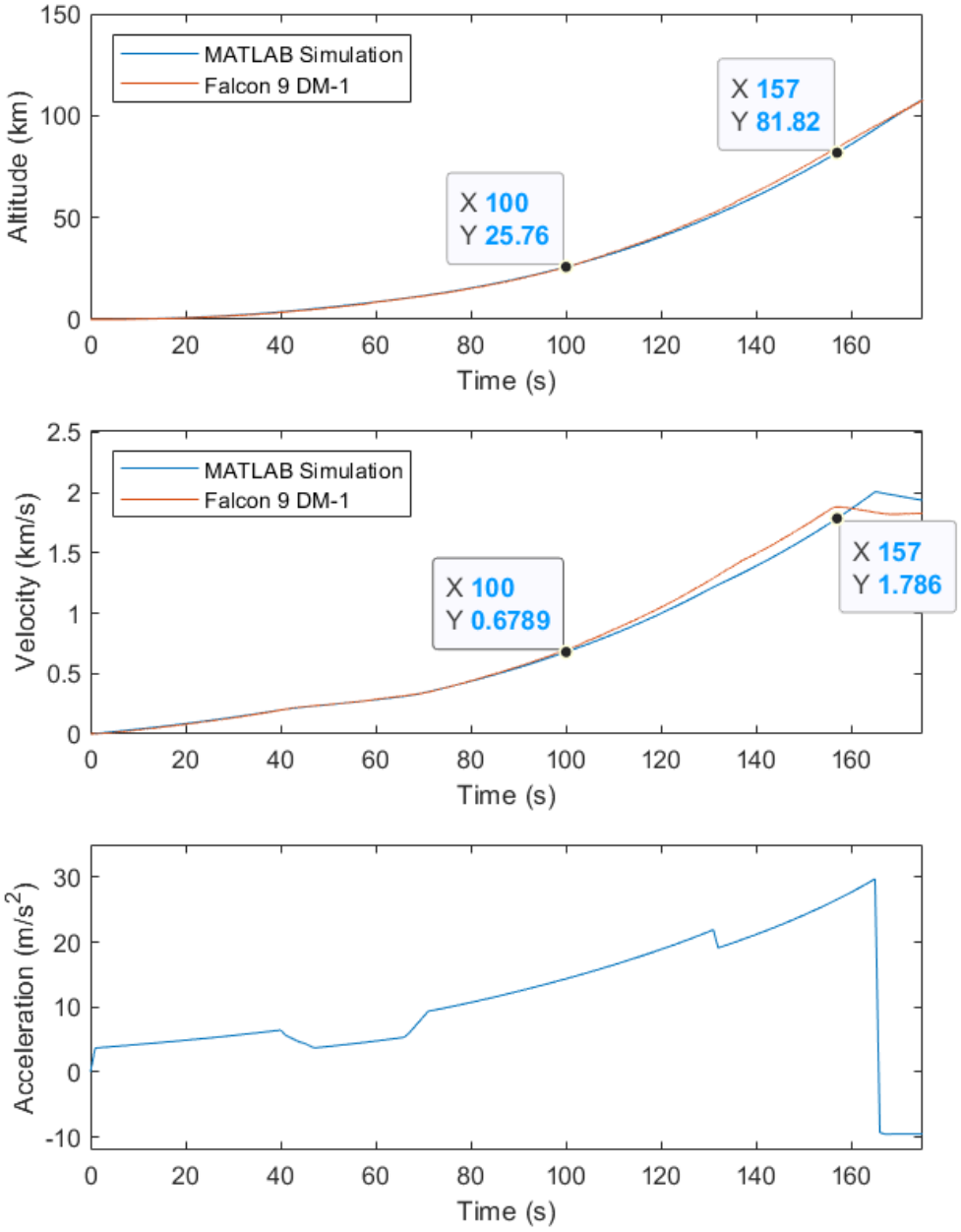


Figure 5.5: Falcon 9 DM-1 mission flight altitude and velocity over time, as recorded by the flight telemetry and as simulated by the MATLAB tool. The bottom plot is the acceleration/propulsion profile introduced into the Trajectory discipline to simulate the Falcon 9 propulsion system.

Chapter 6

Conclusions and Future Work

6.1 Concluding Remarks

The proposed MATLAB multidisciplinary design optimization framework is a useful tool for the conceptual design (Phase 0, Phase A) of new hybrid propellant rocket vehicles. Given the increasing prevalence of hybrid propulsion and new use cases emerging for it, it's predicted this tool will continue to be expanded and developed and it has the potential to be useful for designing experimental suborbital vehicles and student-built hybrid sounding rockets.

The changes implemented in this work have granted the software a much-needed versatility, allowing it to simulate trajectories in a two-dimensional orbital plane, a requirement for modelling gravity turns of orbital-class booster stages, and establishing the ground works for the introduction of new features, such as throttle control, and reusability.

The trajectory and propulsion disciplines have been updated and validated by comparison with other simulating software and experimental data, with acceptable error rates. On the other disciplines, more work needs to be done in the following weeks and improvements are required to allow the tool to design higher mass and higher apogee vehicles with acceptable errors.

Looking back at the objectives set in Chapter 1, the work has met all of them to varying degrees. The completion of the first objective is somewhat subjective, meaning the objective itself was poorly written. Although, in fact, the design optimization of hybrid launch vehicles was explored, much was left untouched. For example, even though the problem is relatively simple and does not need any sophisticated MDO architecture, only the most classic architectures (IDF, MDF, SAND and AAO) were discussed and only IDF and MDF were experimented with. A gradient-free optimization algorithm could have been tested and evaluated against the SQP method. On the rocket subsystems modelling, though, very little was accomplished outside the Trajectory discipline.

Speaking of which, the new 3-DOF model was implemented successfully and did expand the scope and capabilities of the software, although more thorough testing is needed with regards to the optimizer handling of the pitch control design variables.

6.2 Future Work

As is expected with a software tool of this kind, there's an endless sea of possibilities for expansion of capabilities and performance optimization. That being said, the area probably needing more attention is the Mass and Sizimg discipline, where a complete overhaul of the empirical models, specially the mass regression laws, should be done. An attempt was made at building new empirical models, again resorting to multivariate regressions of the mass of some components with relation to the propellant mass, apogee or payload mass. However, a lot of data needs to be collected on existing rocket systems to build a high confidence model, which is a labour intensive task, not to mention there isn't a lot of published reports of detailed mass budgets for rocket vehicles.

Besides the masses and lengths calculated with regression data, there are a lot of variables whose values are hard-coded or just linearly dependent on others. For example, the nozzle length and mass.

Under the new MDF architecture, a Genetic Algorithm may be better suited to solve the problem, since it should be able to explore the non-convex parts of the design space better than SQP. Under the SQP, the initial design point guess is very important.

And finally, a true cost model for the objective function would be a great addition to the software, especially for more commercial applications.

Bibliography

[1] Portugal Space. *Democratization of Space - the full integration of space*. URL: https://ptspace.pt/democratization-of-space/ (visited on 05/30/2022).

[2] Thyrso Villela et al. "Towards the Thousandth CubeSat: A Statistical Overview". In: *International Journal of Aerospace Engineering* 2019 (Jan. 2019), pp. 1–13. DOI: 10.1155/2019/5063145.

[3] Inês Bernardino. "Electrical Design of Radio Antenna for the TT&C Subsystem of the ORCASat Nanosatellite". MA thesis. Instituto Superior Técnico, Jan. 2021.

[4] HyImpulse Technologies. *HyImpulse*. URL: https://hyimpulse.de/en/ (visited on 06/30/2021).

[5] HyPr Space. *Hybrid Propulsion for Space - OB-1 / Mk2*. 2022. URL: https://hybrid-propulsion.space/vehicle_orb2.html#vehicle (visited on 05/11/2022).

[6] Gil Denis et al. "From new space to big space: How commercial space dream is becoming a reality". In: *Acta Astronautica* 166 (Jan. 2020), pp. 431–443. DOI: 10.1016/j.actaastro.2019.08.031.

[7] Brian J. Cantwell, Arif M. Karabeyoglu, and David G. Altman. "Recent Advances in Hybrid Propulsion". In: *International Journal of Energetic Materials and Chemical Propulsion* 9 (2010), pp. 305–326. DOI: 10.1615/IntJEnergeticMaterialsChemProp.v9.i4.20.

[8] Omnidea Lda. *Viriato*. URL: https://www.omnidea.net/viriato.html (visited on 03/30/2022).

[9] Gustavo Hideki Yamada. "Multidisplinary Design Optimization of a Hybrid Rocket". MA thesis. Instituto Superior Técnico, Nov. 2019.

[10] Benjamin Klammer. "Hybrid Rocket Modeling and Design Optimization". Honours thesis. University of Victoria, Apr. 2019.

[11] Hammad Shah et al. "Design and Prototyping of a Solid Fuel/Gaseous Oxidizer Hybrid Rocket Engine". PhD thesis. Dec. 2016. DOI: 10.13140/RG.2.2.12066.22728.

[12] Travis S. Taylor. *Introduction to Rocket Science and Engineering*. CRC Press, 2009. ISBN: 978-1-4200-7529-8.

[13] George Sutton and Oscar Biblarz. *Rocket Propulsion Elements*. 8th ed. Wiley, 2010. ISBN: 978-0-470-08024-5.

[14] Hadron Inc. *Ayame/PAM-D apogee kick motor nozzle failure analysis*. 1981.

[15] Martin J. L. Turner. *Rocket and Spacecraft Propulsion*. 3rd ed. Springer-Praxis, 2009. ISBN: 978-3-540-69202-7.

[16] Ashish Tewari. *Atmospheric and Space Flight Dynamics*. Birkhäuser Boston, 2007. ISBN: 978-0-8176-4438-3.

[17] Orbital Propulsion Centre - ArianeGroup. *Hydrazine Propellant Tanks Overview*. URL: https://hyimpulse.de/en/https://www.space-propulsion.com/spacecraft-propulsion/hydrazine-tanks/hydrazine-tank-overview.html (visited on 02/16/2022).

[18] M. Kobald et al. "Hybrid Experimental Rocket Stuttgart: A Low-Cost Technology Demonstrator". In: *Journal of Spacecraft and Rockets* 55.2 (2018), pp. 484–500. DOI: 10.2514/1.A34035.

[19] Chelsea Gohd. *Virgin Galactic launches Richard Branson to space in 1st fully crewed flight of VSS Unity*. URL: https://www.space.com/virgin-galactic-unity-22-branson-flight-success (visited on 03/17/2022).

[20] Alec Marsh. *Black Arrow was the radical British rocket that was doomed before it flew. But it flew anyway*. URL: https://www.nationalgeographic.co.uk/science-and-technology/2021/06/black-arrow-was-the-radical-british-rocket-that-was-doomed-before-it-flew-but-it-flew-anyway (visited on 03/26/2022).

[21] M. Calabro. "Overview on hybrid propulsion". In: *EUCASS Proceedings Series - Advances in AeroSpace Sciences*. Ed. by Array. Vol. 2. 2011, pp. 353–374. DOI: 10.1051/eucass/201102353.

[22] Alessandro Mazzetti, Laura Merotto, and Giordano Pinarello. "Paraffin-based hybrid rocket engines applications: A review and a market perspective". In: *Acta Astronautica* 126 (2016). Space Flight Safety, pp. 286–297. ISSN: 0094-5765. DOI: 10.1016/j.actaastro.2016.04.036.

[23] Caleb Henry. *DLR spinoff HyImpulse plans small launcher debut in 2022 - SpaceNews*. URL: https://spacenews.com/dlr-spinoff-hyimpulse-plans-small-launcher-debut-in-2022/ (visited on 06/30/2021).

[24] Christian Schmierer et al. "Low cost small-satellite access to space using hybrid rocket propulsion". In: *Acta Astronautica* 159 (2019), pp. 578–583. ISSN: 0094-5765. DOI: 10.1016/j.actaastro.2019.02.018.

[25] Guy Norris. "Galactic Quest". In: *Aviation Week & Space Technology* (Sept. 2009), pp. 54–58.

[26] EPFL Rocket Team. *EPFL Rocket Team Wiki*. URL: https://rocket-team.epfl.ch (visited on 02/24/2022).

[27] Walter E. Hammond. *Design Methodologies for Space Transportation Systems*. American Institute of Aeronautics & Astronautics, 2001. ISBN: 978-1-56347-472-9. DOI: 10.2514/4.861734.

[28] Nigel T. Drenthe. "SOLSTICE: Small Orbital Launch Systems, a Tentative Initial Cost Estimate". MA thesis. Delft University of Technology, Nov. 2016. URL: http://resolver.tudelft.nl/uuid:d93f5983-9fd4-483f-9801-6d99824866c6.

[29] Joaquim R. R. A. Martins and Andrew Ning. *Engineering Design Optimization*. Cambridge University Press, 2021. ISBN: 9781108980647. DOI: 10.1017/9781108980647.

[30] Joaquim R.R.A. Martins, Juan J. Alonso, and James J. Reuther. “A Coupled-Adjoint Sensitivity Analysis Method for High-Fidelity Aero-Structural Design”. In: *Optimization and Engineering* 6.33-62 (2005). DOI: 10.1023/B:OPTE.0000048536.47956.62.

[31] Pedro J. Copado-Méndez et al. “Enhancing the ϵ-constraint method through the use of objective reduction and random sequences: Application to environmental problems”. In: *Computers & Chemical Engineering* 87 (2016), pp. 36–48. ISSN: 0098-1354. DOI: 10.1016/j.compchemeng.2015.12.016.

[32] Yukihiro Kosugi et al. “Multidisciplinary and Multi-objective Design Exploration Methodology for Conceptual Design of a Hybrid Rocket”. In: *Infotech@Aerospace 2011*. AIAA 2011-1634. 2011. DOI: 10.2514/6.2011-1634.

[33] Joaquim R.R.A. Martins and Andrew B. Lambe. “Multidisciplinary design optimization: A survey of architectures”. In: *AIAA Journal* 51.9 (2013), pp. 2049–2075. DOI: 10.2514/1.J051895.

[34] Mathieu Balesdent et al. “Advanced Space Vehicle Design Taking into Account Multidisciplinary Couplings and Mixed Epistemic/Aleatory Uncertainties”. In: *Space Engineering: Modeling and Optimization with Case Studies*. Springer International Publishing, 2016. ISBN: 978-3-319-41508-6. DOI: 10.1007/978-3-319-41508-6_1.

[35] Andrew B. Lambe and Joaquim R.R.A. Martins. “Extensions to the design structure matrix for the description of multidisciplinary design, analysis, and optimization processes”. In: *Structural and Multidisciplinary Optimization* 46.2 (2012), pp. 273–284. DOI: 10.1007/s00158-012-0763-y.

[36] S. Gordon and B. J. McBride. *Computer Program for Calculation of Complex Chemical Equilibrium Compositions and Applications*. NASA RP-1311. 1994.

[37] Luís M. B. C. Campos and Paulo J. S. Gil. “On Four New Methods of Analytical Calculation of Rocket Trajectories”. In: *Aerospace* 5.3 (2018). ISSN: 2226-4310. DOI: 10.3390/aerospace5030088.

[38] John E. Williams and Steven R. Vukelich. *The USAF Stability and Control Digital DATCOM*. McDonnell Douglas Astronautics Company - St Louis Division, 1979.

[39] Arjan Fraters and Angelo Cervone. “Experimental Characterization of Combustion Instabilities in High-Mass-Flux Hybrid Rocket Engines”. In: *Journal of Propulsion and Power* 32.4 (2016), pp. 958–966. DOI: 10.2514/1.B35485.

[40] Richard Nakka. *Flight Xi-16*. 2020. URL: https://nakka-rocketry.net/Xi-16.html (visited on 09/15/2021).

[41] Anonymous. *Telemetry Data DM-1 GitHub*. URL: https://github.com/shahar603/Telemetry-Data/tree/master/DM-1/Excel (visited on 03/03/2019).

Appendix A

MATLAB Source Code

A.1 RocketOptimization.m

```
%% Main Function for the Rocket Optimization Program

% Main function input
fun = @(x)Multidisciplinar7(x);
nonlincon = @(x)Constraints6(x);

%% Boundaries definition
%FULL ENVELOPE (MAXIMUM AND MINIMUM BOUNDS) - ADAPT TO YOUR CASE
%lb = [0.004, 5e-6, 0.2, 0.030, 0.02, 2, 4, 1, 0.1, 0.4, 0.4, 0.7, 0.15,
    0.3, 0.01, 0.04, 0.01, 0.01, 0.25, 0.1, 0.3, 0.25, 0.01, 0.1, 0.05, 0.5,
     0.039, 0.1, 0.5, 0.04,0,0,-1,0,-8*pi/180];
%ub = [6, 1e-3, 5, 3.5, 0.30, 12, 70000, 30, 4, 4000, 4000, 15000, 10, 9,
    2, 3.9, 3, 2, 6, 2, 500, 8000, 500, 1000, 1000, 25, 3.89, 30000, 50000,
    3.9,1500,1000,0,6,8*pi/180];

%For generic 10km-50km rocket
lb = [0.003, 1e-6, 0.1, 0.030, 0.01, 2, 0.3, 0.01, 0.035, 0.01, 0.01, 0, 0,
     -1, 0, -8*pi/180];
ub = [0.3, 5e-3, 5.0, 3.5, 0.35, 12, 5, 1, 4, 1, 0.75, 50, 30, 0, 6, 8*pi
    /180];

lb=lb';
ub=ub';

%% Initial guess x0
```

```
x0 = [0.02,4.379e-05,0.2,0.05,0.0276,6.07,0.712,0.175,0.07835,0.28,0.125,0,0,0,0,0];

%% MultiStart
%rng default
opts = optimoptions(@fmincon,'Algorithm','sqp','Display','iter','tolcon',1e-3,'MaxFunctionEvaluations',5000,'FunValCheck','off'); %,'PlotFcn',['optimplotx','optimplotfvalconstr']
problem = createOptimProblem('fmincon','objective',fun,'x0',x0,'lb',lb,'ub',ub,'nonlcon',nonlincon,'options',opts);
ms = MultiStart('Display','iter','StartPointsToRun','bounds');
%ms.UseParallel = true;
[x_sqp,f_sqp,exitflag,outpt,solutions] = run(ms, problem,10);
```

A.2 Multidisciplinar7.m

```
function [obj,state] = Multidisciplinar7(x)
% Multidisciplinar7 is the main function that uses
% the individual disciplines to create an optimal
% design of a hybrid rocket.

% DESIGN VARIABLES + STATE VARIABLES

% PROPULSION
% Inputs:
% 'V_tank' - oxidizer tank volume (m^3) [x0]
% 'C_inj' - effective injector area (m^2) [x1]
% 'L' - fuel grain length (m) [x1]
% 'd_port_init' - initial fuel grain port diameter(m) [x1]
% 'd_th' - nozzle throat diameter (m) [x0]
% 'A_ratio_nozzle' - nozzle area ratio as a fraction [x0]
% Outputs:
% 'RCC_in' - combustion chamber internal radius (m) [y1]
% 'Mfuel' - fuel mass (kg) [y1]
% 'Mox' - fuel mass (kg) [y1]

% MASS&SIZING
% 'V_tank' - oxidizer tank volume (m^3) [x0]
% 'ROT' - oxidizer tank radius (m) [x0]
```

```
% 'LCC' - combustion chamber length (m) [x0]
% 'Dfin' - fin span (m) [x0]
% 'Bfin' - fin root chord (m) [x0]
% 'bfin' - fin tip chord (m) [x0]
% 'Madd' - additional mass in the recovery bay(kg) [x2]
% 'Madd_cone' - additional mass in the nose cone(kg) [x2]
% 'mass' - rocket's mass (kg) [y2]
% 'Lfuel' - fuel length (m) [?]
% 'De' - rocket's external diameter (m) [y2]
% 'MCC' - combustion chamber mass (kg)
% 'MOT' - oxidizer tank mass (kg)
% 'Mtube' - external structure mass (kg)
% 'LOT' - oxidizer tank length (m)
% 'Lcomponents' - other components length (m)
% 'Mcomponents' - other components mass (kg)
% 'Mfins' - fins mass (kg)
% 'Mnozzle' - nozzle mass (kg)
% 'Mcone' - cone mass (kg)
% 'RCC' - combustion chamber radius (m)

% AERODYNAMICS
% 'A_ratio_nozzle' - nozzle area ratio as a fraction [x0]
% 'Dfin' - fin span (m) [x0]
% 'Bfin' - fin root chord (m) [x0]
% 'bfin' - fin tip chord (m) [x0]
% 'LCC' - combustion chamber length (m) [x0]
% 'd_OT_i' - distance between nose cone tip and oxidizer tank (m) [y3]
% 'CP' - center of pressure [y3]
% 'CG' - center of gravity [y3]
% 'SM' - static margin [y3]

% TRAJECTORY
% 'd_th' - nozzle throat diameter (m) [x0]
% 'A_ratio_nozzle' - nozzle area ratio as a fraction [x0]
% 'vel' - velocity of the rocket over time vector (m/s) [y4]
% 'gain' - %pitch controller gain [x]
% 'deltat_po' - seconds, pitch-over manoeuvre duration [x]
% 'delta_theta' - rad, pitch-over angle [x]

```

```
% CONSTANTS
% 'p_feed' - feed system pressure drop (Pa)
% 'rho_f' - solid fuel density (kg/m^3)
% 'a' - regression rate coefficient (m/s)
% 'n' - regression rate exponent
% 'launchAlt' - altitude of launch site (m)
% 'p_amb' - ambient pressure at launch site (Pa)
% 'T_amb' - ambient temperature at launch site (K)
% 'zeta_d' - discharge correction factor
% 'zeta_cstar' - characteristic velocity correction factor
% 'zeta_CF' - thrust coefficient correction factor

%% INPUTS %%
% Load input files
state.CEA = load('n2o-paraffin_20211026_21h30.mat');% Combustion product
    lookup table for ABS fuel
state.N2Osat = load('N2Osat.mat'); % Nitrous oxide saturation properties

% Declare additional non-design model input
del_time = 0.01; % (s) Time step
time_max = 70; % (s) Maximum model run time
fill_level = 0.70; % (%) Initial liquid volume fraction in tank
p_tank_init = 6000000; % (Pa) Initial tank pressure
p_feed = 100000; % (Pa) Feed system pressure drop
rho_f = 930; % (kg/m^3) Fuel density. For ABS = 1040. For paraffin = 930.
a = 0.000155; % Regression rate constant for paraffin&N2O
n = 0.5; % Regression rate exponent for paraffin&N2O
%a = 0.0001267; % Regression rate constant for ABS&N2O
%n = 0.3728; % Regression rate exponent for ABS&N2O
launchAlt = 1400; % (m) Altitude above mean sea level of the launch site or
    launch pad
launchAngle = 86; % (degrees) Inclination of the launch rail. If no launch
    rail is used, set this to 90.
rail_length = 12; % (m) Launch rail length. If there is no rail, set to
    zero.
T_amb = 301; % (K) Ambient Temperature at Launch Site
p_amb = 85600; % (Pa) Pressure at 1400m ASL
zeta_d = 1.05; % Nozzle discharge correction factor
zeta_cstar = 0.90; % Characteristic velocity correction factor
```

```
98  zeta_CF = 0.90; % Nozzle coefficient correction factor
99  N_fins = 4; %Number of fins
100 TVC_enabled = 0; %Thrust vectoring system. Disabled = 0, Enabled = 1
101
102 %% Inputs arrangement
103 % Place values into struct for function handling
104  state.parameters = [del_time, time_max, fill_level, p_tank_init, p_feed,
        rho_f, a, n, T_amb, p_amb, zeta_d, zeta_cstar, zeta_CF];
105  state.N_fins = N_fins;
106  state.TVC = TVC_enabled;
107  state.launchAlt = launchAlt;
108  state.launchAngle = launchAngle*pi/180; %converts deg to rad
109  state.rail_length = rail_length;
110  state.p_amb = p_amb;
111  state.design = x(1:6); %[V_tank, C_inj, L, d_port_init, d_th,
        A_ratio_nozzle]
112  state.LCC = x(7)  ;
113  state.Dfin = x(8)  ;
114  state.ROT = x(9)  ;
115  state.Bfin = x(10)  ;
116  state.bfin = x(11)  ;
117  state.Madd = x(12) ;
118  state.Madd_cone = x(13) ;
119  state.gain = x(14) ;
120  state.deltat_po = x(15) ;
121  state.delta_theta = x(16) ;
122 %% Create Constants
123 %Length
124 Lpay = 0.45;
125 Lnozzle = 0.2; %average value from reference
126 state.length = [Lpay,Lnozzle];
127
128 %Mass payload
129 state.payload = 15; % (kg) payload mass
130
131 %% Model Calculations
132 %Propulsion
133 try
134     state = MMPropulsion2(state);
```

```
135 catch ME
136     state.ME = ME; % If error gets thrown, return NaN
137     state.I_sp = NaN;
138     state.apogee = NaN;
139     state.pcc_max = NaN;
140     state.Thrust_max = NaN;
141     obj = NaN;
142     return
143 end
144 state = MMMass_sizing2(state);
145 state = MMAerodynamics(state);
146 state = MMTrajectory(state);
147
148 if max(isnan(state.alt)) == 0 && max(state.alt) > state.rail_length
149     si = find(state.alt ~= 0, 1, 'first'); % First index that is not zero
150     ei = find(state.alt > state.rail_length, 1, 'first'); % First index
        where alt is greater than nine meters
151     state.vel_off_rail = interp1(state.alt(si:ei), state.vel(si:ei), state.
        rail_length); % Calculate off-the-rail velocity
152 else
153     state.vel_off_rail = 0;
154 end
155
156 % Calculate Objective Function
157 if ~isnan(state.I_sp)
158     obj_1 = real((1000-state.I_sp)/(1000-180));
159     obj_2 = real(state.mass/100);
160     weight = 0.85;
161     obj = weight*obj_1 + (1-weight)*obj_2;
162 else
163     obj = NaN;
164 end
165
166 end
```

A.3 Constraints6.m

```
1 function [c,ceq] = Constraints6(x)
2 c = zeros(11,1);
```

```
ceq = zeros(1,1);

%% Model Inputs
 % Load input files
  state.CEA = load('n2o-paraffin_20211026_21h30.mat');% Combustion product
      lookup table for paraffin fuel
  state.N2Osat = load('N2Osat.mat'); % Nitrous oxide saturation properties

% Declare additional non-design model input
  del_time = 0.01; % (s) Time step
  time_max = 70; % (s) Maximum model run time
  fill_level = 0.70; % (%) Initial liquid volume fraction in tank
  p_tank_init = 6000000; % (Pa) Initial tank pressure
  p_feed = 100000; % (Pa) Feed system pressure drop
  rho_f = 930; % (kg/m^3) Fuel density. For ABS = 1040. For paraffin = 930.
  a = 0.000155; % Regression rate constant for paraffin&N2O
  n = 0.5; % Regression rate exponent for paraffin&N2O
 %a = 0.0001267; % Regression rate constant for ABS&N2O
 %n = 0.3728; % Regression rate exponent for ABS&N2O
  launchAlt = 1400; % (m) Altitude above mean sea level of the launch site
      or launch pad
  launchAngle = 86; % (degrees) Inclination of the launch rail. If no launch
       rail is used, set this to 90.
  rail_length = 12; % (m) Launch rail length. If there is no rail, set to
      zero.
  T_amb = 301; % (K) Ambient Temperature at Launch Site
  p_amb = 85600; % (Pa) Pressure at 1400m ASL
  zeta_d = 1.05; % Nozzle discharge correction factor
  zeta_cstar = 0.90; % Characteristic velocity correction factor
  zeta_CF = 0.90; % Nozzle coefficient correction factor
  N_fins = 4; %Number of fins
  TVC_enabled = 0; %Thrust vectoring system. Disabled = 0, Enabled = 1

 %% Inputs arrangement
 % Place values into struct for function handling
  state.parameters = [del_time, time_max, fill_level, p_tank_init, p_feed,
      rho_f, a, n, T_amb, p_amb, zeta_d, zeta_cstar, zeta_CF];
  state.N_fins = N_fins;
  state.TVC = TVC_enabled;
```

```
37   state.launchAlt = launchAlt;
38   state.launchAngle = launchAngle*pi/180; %converts deg to rad
39   state.rail_length = rail_length;
40   state.p_amb = p_amb;
41   state.design = x(1:6); %[V_tank, C_inj, L, d_port_init, d_th,
         A_ratio_nozzle]
42   state.LCC = x(7)  ;
43   state.Dfin = x(8)  ;
44   state.ROT = x(9)  ;
45   state.Bfin = x(10)  ;
46   state.bfin = x(11)  ;
47   state.Madd = x(12) ;
48   state.Madd_cone = x(13) ;
49   state.gain = x(14) ;
50   state.deltat_po = x(15) ;
51   state.delta_theta = x(16) ;
52
53  %% Create Constants
54  %Length
55  Lpay = 0.45;
56  Lnozzle = 0.2; %average value from reference
57  state.length = [Lpay,Lnozzle];
58
59  %Mass payload
60   state.payload = 15; % (kg) payload mass
61
62   %% Model Calculations
63   try
64       state = MMPropulsion2(state);
65  catch ME
66       state.ME = ME; % If error gets thrown, return NaN
67       state.I_sp = NaN;
68       state.apogee = NaN;
69       state.pcc_max = NaN;
70       state.Thrust_max = NaN;
71       obj = NaN;
72       return
73  end
74   state = MMMass_sizing2(state);
```

```
75  state = MMAerodynamics(state);
76  state = MMTrajectory(state);
77
78  if max(isnan(state.alt)) == 0 && max(state.alt) > state.rail_length
79      si = find(state.alt ~= 0, 1, 'first'); % First index that is not zero
80      ei = find(state.alt > state.rail_length, 1, 'first'); % First index
            that is higher than rail
81      state.vel_off_rail = interp1(state.alt(si:ei), state.vel(si:ei), state.
            rail_length); % Calculate off-the-rail velocity
82  else
83       state.vel_off_rail = 0;
84  end
85
86  %% Variables for Constraints
87  pcc = state.pcc;
88  ptank = state.ptank;
89  pfeed = state.pfeed;
90  G = state.G;
91  accel = state.accel;
92
93  if max(isnan(pcc)) == 0
94  for count=1:length(pcc)
95  CC_P_constraint(count) = pcc(count) - 0.8*(ptank(count)-pfeed);
96  end
97  end
98
99  if max(isnan(G)) == 0
100 for count=1:length(G)
101 Fuel_grain_const(count) = G(count)-500;
102 end
103 end
104
105  %% Constraints EQ
106 if ~isnan(state.De)
107 ceq(1) = real(state.apogee - 47500)/100;
108 %ceq(2) = real(state.vel(state.target_time) - 450)/10;
109 %ceq(3) = real(state.fpa(state.target_time) - (20*pi/180));
110
111 else
```

```
112         ceq(:) = NaN;
113     end
114
115     %% Constraints
116     if ~isnan(state.vel_off_rail)
117     c(1) = real(max(CC_P_constraint)/1000);
118     c(2) = real(max(Fuel_grain_const)/10);
119     c(3) = real(max(accel(1:state.time_to_apogee)) - 100);
120     c(4) = real(x(3)+state.De-state.LCC);
121     c(5) = real((state.bfin - 0.9*state.Bfin)*10);
122     c(6) = real(state.Dfin/((state.Bfin + state.bfin)/2) - 3); %Fin Aspect
            Ratio Cap
123     c(7) = real((state.Madd_cone - state.Vcone*11340)/10); %Added mass inside
            cone must be lower than lead density
124     c(8) = real(-state.Mcomponents/10);
125     c(9) = real( 1 -(state.SM))*5;
126     c(10) = real(state.SM - 3)*5;
127     c(11) = real(27 - state.vel_off_rail);
128     else
129         c(:) = NaN;
130     end
131 end
```

A.4 Propulsion2.m

```
1  function [state] = Propulsion2(state)
2  % PROPULSION
3  %   Propulsion2 is a function that simulates the hybrid
4  %   rocket motor. It is used for design optimization
5  %   using "Multidisciplinar7" as the main function.
6  %
7  % All calculations performed in metric units
8
9  %% INPUTS %%
10 CEA = state.CEA; % Combustion product lookup table
11 N2Osat = state.N2Osat; % Nitrous oxide saturation properties
12
13 design = num2cell(state.design); % Distribute design vector
14 [V_tank, C_inj, L, d_port_init, d_th, A_ratio_nozzle] = design{:};
```

```
15
16 parameters = num2cell(state.parameters); % Distribute parameter vector
17 [del_time, time_max, fill_level, p_tank_init, p_feed, rho_f, a, ...
18     n, T_amb, p_amb, zeta_d, zeta_cstar, zeta_CF] = parameters{:};
19
20 iteration_tolerance = 0.01; % Fraction tolerances on iterative solvers
21
22 tic; % Start timing function
23
24 %% INITIAL CALCULATIONS %%
25 % Calculate initial thermodynamic properties from tank pressure
26 thermo_sat_init = thermoSat(p_tank_init, 'p', {'T', 'rho_liq', 'rho_vap', '
       u_liq', 'u_vap'});
27 thermo_sat_init = num2cell(thermo_sat_init);
28 [T_tank, rho_liq, rho_vap, u_liq, u_vap] = thermo_sat_init{:};
29
30 if ~exist('m_ox_tank_init', 'var') % If oxidizer mass is not given,
       calculate it using fill level
31     m_ox_tank_init = V_tank*(rho_liq*fill_level + (1-fill_level)*rho_vap);
           % Calculate initial oxidizer mass based on fill level
32 end
33
34 x_tank = (V_tank/m_ox_tank_init - 1/rho_liq)/(1/rho_vap - 1/rho_liq); %
       Calculate vapour mass fraction
35 u_tank = (x_tank*u_vap + (1-x_tank)*u_liq); % (J/kg*K) Calculate specific
       internal energy
36 U_tank = m_ox_tank_init*u_tank; % (J) Calculate total internal energy
37
38 if x_tank <= 0
39     error('Propulsion2:too high oxidizer mass. The initial oxidizer mass is
           too high for the considered maximum pressure (x_tank <= 0)')
40 end
41
42 % Calculation of constants
43 R_const = 8.3144598; % (J/mol*K) Gas constant
44 A_th = pi()*(d_th/2)^2; % (m^2) Nozzle Throat Area
45 V_tank_eps = iteration_tolerance*V_tank; % (m^3) Set acceptable tank volume
       error to 1% of tank volume
46 u_tank_eps = iteration_tolerance*u_tank; % (m^3) Set acceptable tank
```

```
    specific internal energy error to 1% of initial tank specific internal
    energy
47  A_ratio_nozzle_eps = iteration_tolerance*A_ratio_nozzle; % (m^3) Set
    acceptable nozzle area ratio error to 1% of nozzle area ratio
48
49  % Setting initial conditions
50  m_tot = 0; % (kg) Initialize total mass of propellant
51  m_ox_tank = m_ox_tank_init; % (kg) Mass of oxidizer in tank
52  r_cc = d_port_init/2; % (m) Combustion chamber port radius
53  p_cc = p_amb; % (Pa) Initial Combustion chamber pressure
54  T_cc = T_amb; % (K) Combustion chamber temperature
55  T_stag = T_amb; % (K) Combustion chamber stagnation temperature
56  R_cc = R_const/0.02897; % (J/kg*K) Molar mass of air at 298K
57  k_cc = 1.4; % Specific heat ratio of air at 298K
58  Ma = 3; % Initialize the mach number near solution to ensure convergance
59  burn_time = time_max; % Initialize burn time to max time
60
61  %% MODEL CALCULATIONS %%
62  t = 1;
63  try % Start try block to catch errors that occur in the main loop (for
    debugging)
64      while 1 % Run until break
65          if isnan(T_stag(t))
66              T_stag(t+1) = NaN;
67              break
68          else
69              if x_tank(t) < 1
70                  % OXIDIZER TANK CALCULATIONS FOR SATURATED LIQUID AND
                        VAPOUR %
71                  Vin.U = U_tank(t);
72                  Vin.m = m_ox_tank(t);
73                  Vin.V = V_tank;
74                  if abs(Verror(T_tank(t), Vin)) > V_tank_eps
75                      T_tank(t) = secant(@Verror, T_tank(t), Vin);
76                  end
77
78                  % Calculation of tank thermo properties
79                  thermo_sat_tank = thermoSat(T_tank(t), 'T', {'p', 'h_liq',
                        ...
```

```
80                  'h_vap', 'rho_liq', 'rho_vap', 'u_liq', 'u_vap'});
81              thermo_sat_tank = num2cell(thermo_sat_tank);
82              [p_tank(t), h_liq, h_vap, rho_liq, rho_vap, u_liq, u_vap] =
                    thermo_sat_tank{:};
83              x_tank(t) = (U_tank(t)/m_ox_tank(t) - u_liq)/(u_vap - u_liq
                    );
84              u_tank(t) = x_tank(t)*u_vap + (1-x_tank(t))*u_liq;
85              h_tank(t) = x_tank(t)*h_vap + (1-x_tank(t))*h_liq;
86              rho_tank(t) = 1/(x_tank(t)/rho_vap + (1-x_tank(t))/rho_liq)
                    ;
87
88              rho_discharge = rho_liq;
89              h_discharge = h_liq;
90
91              % Record burn time to be the time at which the tank runs
                    out of oxidizer
92              if x_tank(t) > 1
93                  burn_time = t*del_time;
94              end
95          else
96              % OXIDIZER TANK CALCULATIONS FOR VAPOUR ONLY %
97              rho_tank(t) = m_ox_tank(t)./V_tank;
98              u_tank(t) = U_tank(t)/m_ox_tank(t);
99              uin.rho = rho_tank(t);
100             uin.u = u_tank(t);
101             if abs(uerror(T_tank(t), uin)) > u_tank_eps
102                 T_tank(t) = bisection(@uerror, 1, 1000, uin, T_tank(t)*
                        iteration_tolerance);
103             end
104
105             thermo_span_tank = thermoSpanWagner(rho_tank(t), T_tank(t),
                    {'p','h'});
106             p_tank(t) = thermo_span_tank(1);
107             h_tank(t) = thermo_span_tank(2) + 7.3397e+05; % Convert
                    from Span-Wagner enthalpy convention to NIST
108
109             h_discharge = h_tank(t);
110             rho_discharge = rho_tank(t);
111
```

```
112             end
113
114             if p_tank(t)-p_feed-p_cc(t) < 0 % Check that the chamber
                    pressure is sufficiently lower than the tank pressure
115                 error('Propulsion2:highPressure', 'The combustion chamber
                        pressure is too high (p_cc=%7.0f Pa, p_tank=%7.0f Pa, )
                        \n', p_cc(t), p_tank(t))
116             end
117             m_dot_ox_in(t) = C_inj*sqrt(2*rho_discharge*(p_tank(t)-p_cc(t)-
                    p_feed)); % Incompressible fluid assumption (better than
                    nothing)
118             if t == 1  % Take average over last two time periods to
                    attenuate numerical instability
119                 m_dot_ox_in = 1/2*m_dot_ox_in;
120             else
121                 m_dot_ox_in(t) = 1/2*m_dot_ox_in(t) + 1/2*m_dot_ox_in(t-1);
122             end
123
124             del_m_ox_tank = -m_dot_ox_in(t)*del_time; % Oxidizer tank mass
                    differential equation
125             %Q_tank = HeatOT(De,ROT,T_tank(t),LOT,RCC,d_OT_i,vel(t)); %
                    Oxidizer tank heat transfer
126             %del_U_tank = (-m_dot_ox_in(t)*h_discharge - Q_tank)*del_time;
                    % Oxidizer tank energy differential equation (original)
127             del_U_tank = -m_dot_ox_in(t)*h_discharge*del_time; % Oxidizer
                    tank energy differential equation (altered)
128
129             % COMBUSTION CHAMBER CALCULATIONS %
130             A_cc(t) = pi()*r_cc(t)^2; % Port geometry
131             G(t) = m_dot_ox_in(t)/A_cc(t); % Oxidizer mass flux
132             del_r_cc(t) = a*G(t)^n * del_time; % Regression rate law
133             m_dot_f(t) = 2*pi()*r_cc(t)*L * rho_f * (del_r_cc(t)/del_time);
                    % Fuel mass flow rate from geometry
134
135             % Iteratively solve for m_dot_f
136             k = 0;
137             m_dot_f_temp = 0;
138             while abs(m_dot_f_temp - m_dot_f(t)) > 0.01*m_dot_f(t) && k <
                    100 % until m_dot_f converges
```

```
139             m_dot_f_temp = m_dot_f(t);
140             m_dot_cc(t) = m_dot_f(t) + m_dot_ox_in(t); % Total mass
                    flow in
141             G(t) = (m_dot_ox_in(t) + m_dot_cc(t))/(2*A_cc(t)); %
                    Average mass flux accross entire port
142             del_r_cc(t) = a*G(t)^n * del_time; % Regression rate law
                    using updated average mass flux
143             m_dot_f(t) = 2*pi()*r_cc(t)*L * rho_f * (del_r_cc(t)/
                    del_time); % Fuel mass flow rate from geometry
144             k = k+1;
145         end
146
147         OF(t) = m_dot_ox_in(t)./m_dot_f(t); % Calculate Oxidizer-Fuel
                Ratio
148         p_stag(t) = m_dot_cc(t)/(zeta_d*A_th)*sqrt(T_stag(t)*R_cc(t)/
                k_cc(t)*((k_cc(t)+1)/2)^((k_cc(t)+1)/(k_cc(t)-1))); % Steady
                 state choked flow expression for pressure
149
150         if t > 1 % First time step is too cold for accurate velocity
                correction
151             vel_cc(t) = m_dot_cc(t)/(rho_cc(t)*A_cc(t));
152             vel_cc(t) = 1/2*vel_cc(t) + 1/2*vel_cc(t-1); % Average
                    velocity over last time step for numerical stability
153             T_cc(t) = T_stag(t) - vel_cc(t)^2/(2*cp_cc(t));
154         else
155             T_cc(t) = T_stag(t);
156         end
157
158         p_cc(t) = p_stag(t)*((T_cc(t)/T_stag(t)).^(k_cc(t)/(k_cc(t)-1))
                ); % Velocity correction for stagnation pressure
159         thermo_comb = CEAProp(OF(t), p_cc(t), {'T', 'rho', 'cp', 'k', '
                M'});
160         thermo_comb = num2cell(thermo_comb);
161         [T_stag(t+1), rho_cc(t+1), cp_cc(t+1), k_cc(t+1), M_cc(t+1)] =
                thermo_comb{:}; % Calculate chamber properties at next time
                step
162         T_stag(t+1) = T_stag(t+1)*zeta_cstar^2; % Correct temperature
                with cstar efficiency
163         R_cc(t+1) = R_const/M_cc(t+1);
```

```
164
165             % NOZZLE AND THRUST CALCULATIONS %
166             Ain.k = k_cc(t);
167             Ain.A = A_ratio_nozzle;
168             if abs(Aerror(Ma(t), Ain)) > A_ratio_nozzle_eps % Nozzle mach
                    number solver
169                 Ma(t) = secant(@Aerror, Ma(t), Ain); % Supersonic solution
170             end
171
172             p_exit(t) = p_stag(t)./(1+(k_cc(t)-1)./2.*Ma(t).^2).^(k_cc(t)/(
                    k_cc(t)-1)); % Nozzle exit pressure
173             if x_tank(t) >= 1 % Physically wrong, but the condition is used
                     to hide the transition that results from overexpansion
174                 p_exit(t) = p_stag(t)*(2/(k_cc(t)+1))^(k_cc(t)/(k_cc(t)-1))
                        ; % Throat pressure
175                 vel_exit(t) = sqrt(2*k_cc(t)*R_cc(t)*T_stag(t)/(k_cc(t)+1))
                        ; % Throat velocity
176                 A_ratio_nozzle_eff = 1; % Throat area ratio
177             else
178                 T_exit(t) = T_stag(t)./(1+(k_cc(t)-1)./2.*Ma(t).^2); % (K)
                        Temperature of gas at exit of nozzle
179                 vel_exit(t) = Ma(t).*sqrt(k_cc(t).*R_cc(t).*T_exit(t)); % (
                       m/s) Velocity of gas at exit of nozzle
180                 A_ratio_nozzle_eff = A_ratio_nozzle; % Effective nozzle
                        area ratio is actual nozzle area ratio
181             end
182             F_thrust(t) = zeta_CF*(m_dot_cc(t).*vel_exit(t) + (p_exit(t)-
                    p_amb).*A_th.*A_ratio_nozzle_eff); % (N) Rocket Motor Thrust
                     !!!
183
184             % ITERATE FORWARD IN TIME %
185             if t < time_max/del_time && m_ox_tank(t)/m_ox_tank_init > 0.05
                    % If less than max burn time and more than 5% of oxidizer is
                     left, step forward in time
186                 % Oxidizer Tank Values
187                 m_ox_tank(t+1) = m_ox_tank(t) + del_m_ox_tank;
188                 U_tank(t+1) = U_tank(t) + del_U_tank;
189                 x_tank(t+1) = x_tank(t);
190                 T_tank(t+1) = T_tank(t);
```

```
191                     m_dot_ox_in(t+1) = m_dot_ox_in(t);
192
193                     % Combustion Chamber Values
194                     m_tot(t+1) = m_tot(t) - m_dot_cc(t)*del_time;
195                     r_cc(t+1) = r_cc(t) + del_r_cc(t);
196                     p_cc(t+1) = p_cc(t);
197                     Ma(t+1) = Ma(t);
198                     t = t+1;
199                 else
200                     break % Exit loop if no more time or oxidizer
201                 end
202             end
203         end
204     catch ME % Catch any errors
205         if isa(state, 'double') && (length(state)==6)
206             state = NaN;
207             return
208         end
209         rethrow(ME)
210     end
211     time = 0:del_time:(del_time*(t-1)); % Create a time vector
212
213
214
215     %% OUTPUTS %%
216     if max(isnan(T_stag))== 0 && isreal(m_dot_ox_in)
217         % Major output calculations
218         m_out = trapz(m_dot_cc)*del_time; % (kg)
219         m_tot = m_tot + m_out;
220         m_f = (trapz(m_dot_f)*del_time); % (kg)
221         I_tot = (trapz(F_thrust)*del_time); % (N*s) Just total impulse during
222         v_e = I_tot/m_out; % Effective exhaust velocity (m/s)
223         I_sp = (v_e/9.81); % (s)
224         %c_star = mean(p_cc)*A_th/mean(m_dot_cc); % (m/s)
225
226         % Save metadata of simulation
227         time_sim = toc; % Save how long the simulation took (minus loading data
                and plotting figures)
228         time_stamp = datestr(datetime); % Save the current time and date
```

```
229         computer_name = computer; % Save which computer the code is running on
230         function_name = mfilename; % Save the name of the function that is
                being called
231
232         % Output thrust and mass vs time for trajectory analysis
233         TCurve = [I_tot, mean(F_thrust), m_out];
234         TCurve = vertcat(TCurve, [time; F_thrust; m_tot]');
235
236         % Save all pertinent data to the state variable
237         state.TC = TCurve;
238         state.burn = burn_time/del_time;
239         state.mass_prop = m_out;
240         state.Mfuel = m_f;
241         state.Mox = m_ox_tank_init;
242         state.OF = OF;
243         state.RCC_in = r_cc(end);
244         state.I_sp = I_sp;
245         state.Ath = A_th;
246         state.AR = A_ratio_nozzle;
247         state.Thrust_max = max(F_thrust);
248         state.pcc_max = max(p_cc);
249         state.meta = [time_sim, time_stamp, computer_name, function_name];
250         state.ptank = p_tank;
251         state.pfeed = p_feed;
252         state.pcc = p_cc;
253         state.G = G;
254         state.I_tot = I_tot;
255         state.ptank_max = max(p_tank);
256         state.time_prop = time;
257         state.Thrust =  F_thrust;
258         state.Tcc = T_cc;
259     else
260         state.TC = NaN;
261         state.burn = NaN;
262         state.mass_prop = NaN;
263         state.Mfuel = NaN;
264         state.Mox = NaN;
265         state.OF = NaN;
266         state.RCC_in = NaN;
```

```
267         state.I_sp = NaN;
268         state.Ath = NaN;
269         state.AR = NaN;
270         state.Thrust_max = NaN;
271         state.pcc_max = NaN;
272         state.meta = [NaN, NaN, NaN, NaN];
273         state.ptank = NaN;
274         state.pfeed = NaN;
275         state.pcc = NaN;
276         state.G = [NaN,NaN];
277     end
278
279
280     %% SUBFUNCTIONS %%
281         function [out] = CEAProp(OF, p, in)
282             %MASSFRAC Calculates the mass fractions of products for a given
                    reaction
283             %   'OF' is the oxidizer to fuel weight ratio
284             %   'p' is the pressure of the combustion chamber
285             %   'alpha' is an array of weight fractions
286             if OF < CEA.OF(1) % Check that query is inside bounds
287                 error('CEAProp:lowOF', 'Outside O/F range: \n OF = %f', OF)
288             elseif OF > CEA.OF(end)
289                 error('CEAProp:highOF', 'Outside O/F range: \n OF = %f', OF)
290             elseif p < CEA.p(1)
291                 error('CEAProp:lowP', 'Outside pressure range: \n p = %f Pa', p
                        )
292             elseif p > CEA.p(end)
293                 error('CEAProp:highP', 'Outside pressure range: \n p = %f Pa',
                        p)
294             end
295
296             if ~isnan(OF)
297                 for mm = 1:length(CEA.OF)
298                     if OF < CEA.OF(mm) % Find first index that is larger than
                            OF
299                         break
300                     end
301                 end
```

```
302         end
303
304         if ~isnan(p)
305             for nn = 1:length(CEA.p)
306                 if p < CEA.p(nn) % Find first index that is larger than p
307                     break
308                 end
309             end
310         end
311
312         out = zeros(size(in));
313
314         if isnan(OF) || isnan(p)
315             %OF_int = NaN;
316             %p_int = NaN;
317             %C_int = NaN;
318             %out_int = NaN;
319             for kk = 1:length(in)
320                 out(kk) = NaN;
321             end
322         else
323             for kk = 1:length(in) % Iterate through all products
324                 % Do some bilinear interpolation (equations from wikipedia)
325                 OF_int = [CEA.OF(mm)-OF, OF-CEA.OF(mm-1)];
326                 p_int = [CEA.p(nn)-p; p-CEA.p(nn-1)];
327                 C_int = 1/((CEA.OF(mm)-CEA.OF(mm-1))*(CEA.p(nn)-CEA.p(nn-1)
                        ));
328                 out_int = CEA.(in{kk})(mm-1:mm,nn-1:nn);
329                 out(kk) = C_int.* OF_int*out_int*p_int;
330             end
331         end
332     end
333
334     function [val_out] = thermoSat(val_in, in, out)
335         %THERMOSAT returns thermodynamic properties at saturation
336         %   'val_in' is a double specifying the value of thermodynamic
                property inputted
337         %   'in' is a string specifying the given input thermodynamic
                property
```

```
338         %   'out' as a cell array specifying the desired output
                thermodynamic property
339         %   'val_out' is a double specifying the value of thermodynamic
                property returned
340         val_out = zeros(size(out));
341         for mm = 1:length(out) % For each desired output
342             % Locate which variable column is input, and which is output
343             ii = find( (in == N2Osat.meta(1,:)), 1, 'first');
344             jj = find( (out(mm) == N2Osat.meta(1,:)), 1, 'first');
345             % Interpolate between the two values
346             for kk = 1:length(N2Osat.data(:,ii))
347                 if val_in < N2Osat.data(kk,ii)
348                     break
349                 end
350             end
351             val_out(mm) = (val_in - N2Osat.data(kk-1,ii))./(N2Osat.data(kk,
                    ii)-N2Osat.data(kk-1,ii)).*(N2Osat.data(kk,jj)-N2Osat.data(
                    kk-1,jj)) + N2Osat.data(kk-1,jj);
352         end
353     end
354
355     function [out] = thermoSpanWagner(rho, T, in)
356         %THERMOSPANWAGNER Calculates thermodynamic properties for N2O as a
                non-ideal gas
357         %   'in' is a cell array containing the desired output parameters
358         %   'rho' is the density in kg/m^3
359         %   'T' is the temperature in K
360         %   'out' is an array containing the output values in the order
                listed in 'in'
361
362         % Hardcode in data for N2O (from "Modeling Feed System Flow Physics
                for Self-Pressurizing Propellants)
363         R = 8.3144598/44.0128*1000; % (J/kg*K) Gas constant
364         T_c = 309.52; % (K) Critical Temperature
365         rho_c = 452.0115; % (kg/m^3) Critical Density
366         n0 = [0.88045, -2.4235, 0.38237, 0.068917, 0.00020367, 0.13122,
                0.46032, ...
367             -0.0036985, -0.23263, -0.00042859, -0.042810, -0.023038];
368         n1 = n0(1:5); n2 = n0(6:12);
```

```
369         a1 = 10.7927224829;
370         a2 = -8.2418318753;
371         c0 = 3.5;
372         v0 = [2.1769, 1.6145, 0.48393];
373         u0 = [879, 2372, 5447];
374         t0 = [0.25, 1.125, 1.5, 0.25, 0.875, 2.375, 2, 2.125, 3.5, 6.5,
                4.75, 12.5];
375         d0 = [1, 1, 1, 3, 7, 1, 2, 5, 1, 1, 4, 2];
376         P0 = [1, 1, 1, 2, 2, 2, 3];
377         t1 = t0(1:5); t2 = t0(6:12);
378         d1 = d0(1:5); d2 = d0(6:12);
379
380         % Calculate non-dimensional variables
381         tau = T_c/T;
382         delta = rho/rho_c;
383
384         % Calculate explicit helmholtz energy and derivatives (from
385         ao = a1 + a2*tau + log(delta) + (c0-1)*log(tau) + sum(v0.*log(1-exp
                (-u0.*tau./T_c)));
386         ar = sum(n1.*tau.^t1.*delta.^d1) + sum(n2.*tau.^t2.*delta.^d2.*exp
                (-delta.^P0));
387         ao_tau = a2 + (c0-1)/tau + sum(v0.*u0./T_c.*exp(-u0.*tau./T_c)./(1-
                exp(-u0.*tau./T_c)));
388         ao_tautau = -(c0-1)/tau.^2 + sum(-v0.*u0.^2./T_c.^2.*exp(-u0.*tau./
                T_c)./(1-exp(-u0.*tau./T_c)).^2);
389         ar_tau = sum(n1.*t1.*tau.^(t1-1).*delta.^d1) + sum(n2.*t2.*tau.^(t2
                -1).*delta.^d2.*exp(-delta.^P0));
390         ar_tautau = sum(n1.*t1.*(t1-1).*tau.^(t1-2).*delta.^d1) + sum(n2.*
                t2.*(t2-2).*tau.^(t2-2).*delta.^d2.*exp(-delta.^P0));
391         ar_delta = sum(n1.*d1.*delta.^(d1-1).*tau.^t1) + sum(n2.*tau.^t2.*
                delta.^(d2-1).*(d2-P0.*delta.^P0).*exp(-delta.^P0));
392         ar_deltadelta = sum(n1.*d1.*(d1-1).*delta.^(d1-2).*tau.^t1) + sum(
                n2.*tau.^t2.*delta.^(d2-2).*((d2-P0.*delta.^P0).*(d2-1-P0.*delta
                .^P0)-P0.^2.*delta.^P0).*exp(-delta.^P0));
393         ar_deltatau = sum(n1.*d1.*t1.*delta.^(d1-1).*tau.^(t1-1)) + sum(n2
                .*t2.*tau.^(t2-1).*delta.^(d2-1).*(d2-P0.*delta.^P0).*exp(-delta
                .^P0));
394
395         out = zeros(size(in));
```

```
396         for kk = 1:length(in)
397             switch in{kk}
398                 case 'p' % Pressure (Pa)
399                     out(kk) = rho*R*T*(1+delta*ar_delta);
400                 case 'u' % Specific internal energy (J/kg)
401                     out(kk) = R*T*tau*(ao_tau+ar_tau);
402                 case 's' % Specific entropy (J/kg*K)
403                     out(kk) = R*(tau*(ao_tau+ar_tau)-ao-ar);
404                 case 'h' % Specific enthalpy (J/kg)
405                     out(kk) = R*T*(1+tau*(ao_tau+ar_tau)+delta*ar_delta);
406                 case 'cv' % Specific heat constant pressure (J/kg*K)
407                     out(kk) = R*-tau^2*(ao_tautau+ar_tautau);
408                 case 'cp' % Specific heat constant pressure (J/kg*K)
409                     out(kk) = R*(-tau^2*(ao_tautau+ar_tautau) + (1+delta*
                            ar_delta-delta*tau*ar_deltatau)^2/(1+2*delta*
                            ar_delta+delta^2*ar_deltadelta));
410                 case 'a' % Speed of sound (m/s)
411                     out(kk) = sqrt(R*T*(1+2*delta*ar_delta+delta^2*
                            ar_deltadelta - (1+delta*ar_delta-delta*tau*
                            ar_deltatau)^2/(tau^2*(ao_tautau+ar_tautau))));
412                 otherwise
413                     error('Invalid input')
414             end
415         end
416     end
417
418     function [x] = secant(fun, x1, in)
419         %SECANT is a zero-finding function based on the secant method
420         %   'fun' is the function handle for which the zero is desired
421         %   'x1' is the initial guess
422         %   'in' is a struct containing any additional inputs 'fun' might
                require
423         %   'x' is the value of the zero
424         x_eps = x1*0.005; % Set the tolerance to be 0.5% of initial guess
425         x2 = x1-x1*0.01; % Set a second point 1% away from the original
                guess
426         F1 = fun(x1, in); % Evaluate function at x1
427         F2 = fun(x2, in); % Evaluate function at x2
428         kk = 1; % Set up counter
```

```
429         kk_max = 1000;
430         while abs(x2-x1)>=x_eps && kk<kk_max % While error is too large and
                counter is less than max
431             x3 = x2 - F2*(x2-x1)/(F2-F1);
432             x1 = x2; % Move everything forward
433             x2 = x3;
434             F1 = F2;
435             F2 = fun(x2, in);
436             kk = kk+1;
437         end
438         x = x2;
439     end
440
441     function [x] = bisection(fun, a, b, in, tol)
442         % BISECTION is a zero-finding function based on the bisection
                method
443         %   'fun' is the function handle for which the zero is desired
444         %   'a' is the lower value of the interval
445         %   'b' is the lower value of the interval
446         %   'in' is a struct containing any additional inputs 'fun' might
                require
447         %   'tol' is the error tolerance
448         %   'x' is the value of the zero
449
450         c=(a+b)/2;
451         kk = 1; % Set up counter
452         kk_max = 1000;
453
454         while(abs(fun(c, in))>=tol && kk<kk_max)
455             if (fun(c, in)*fun(a, in))<0
456                 b=c;
457                 c=(a+c)/2;
458             elseif (fun(c, in)*fun(b, in))<0
459                 a=c;
460                 c=(c+b)/2;
461             elseif f(c)==0
462                 break;
463             end
464             kk=kk+1;
```

```
465             end
466             x=c;
467         end
468
469
470     % SET UP FUNCTIONS TO BE ITERATIVELY SOLVED %
471         function V = Verror(T, in) % Finds the difference between the estimated
                and actual tank volume
472             thermo = thermoSat(T, 'T', {'rho_liq','rho_vap','u_liq','u_vap'});
473             thermo = num2cell(thermo);
474             [rho_l,rho_v,u_l,u_v] = thermo{:};
475             x = (in.U/in.m - u_l)/(u_v - u_l);
476             V = in.m*((1-x)/rho_l + x/rho_v) - in.V;
477         end
478
479         function U = uerror(T, in) % Finds the difference between the estimated
                and actual tank internal energy
480             U = (thermoSpanWagner(in.rho, T, {'u'}) + 7.3397e+5) - in.u;
481         end
482
483         function A = Aerror(M, in) % Finds the difference between the estimated
                and actual nozzle ratio
484             A = (1/M^2)*(2./(in.k+1).*(1+(in.k-1)./2.*M.^2)).^((in.k+1)./(in.k
                    -1)) - in.A^2;
485         end
486
487         function Q = HeatOT(De,ROT,TOT,LOT,RCC,d_OT_i,vel)
488             % Constants
489             Tair = 301; % (K)
490             K_cond = 0.0264704; % (W/m K) thermal condutivity air 30 degC
491             g = 9.81;
492             beta = 0.0033; % (1/K) expansion coefficient air 30 degC
493             nu = 15.9668e-6; % (m^2/s) kinematic viscosity air 30 degC
494             Pr = 0.71; % Prandtl number
495             re = De/2;
496             %alfa = 22.07e-6; %Thermal Diffusivity m^2/s
497
498             % For model validation
499             %       nu = 14.96e-6; % (m^2/s) kinematic viscosity air 19 degC
```

```
500         %       K_cond = 0.025618; % (W/m K) thermal condutivity air 19
                degC
501
502         %R convection internal
503
504         L_heat = 2*(log(RCC/ROT))^(4/3)/(RCC^(-3/5) + ROT^(-3/5))^(5/3);
505         Gr = g*beta*abs(TOT-Tair)*L_heat^3/nu^3;
506         Ra = Gr*Pr;
507         K_eff = K_cond*0.386*(Pr/(Pr+0.861))^0.25*Ra^0.25;
508
509         if K_eff < K_cond
510             K_eff = K_cond;
511         end
512
513         R_conv_in = log(RCC/ROT)/(2*pi*LOT*K_eff);
514
515         %R conduction
516
517         k_cond = 2; %(W/mK) thermal condutivity carbon fiber-epoxy
518
519         R_cond = log(re/RCC)/(2*pi*k_cond*LOT);
520
521         %R convection external
522
523         dx = LOT/1000;
524         x = d_OT_i;
525         %cont = 1;
526         iterations = ceil((x-LOT)/dx);
527         h  = zeros(iterations,1);
528
529         for i = 1:iterations
530             position = x + (i-1)*dx;
531             Re = vel*position/nu;
532             Nu = 0.0296*Re^(4/5)*Pr^(1/3);
533             h(i) = Nu*K_cond/position;
534         end
535
536         h_average = trapz(h)*dx/LOT;
537
```

```
538             R_conv_ex = 1/(h_average*2*pi*re*LOT);
539
540             %T = TOT;
541             Q = (TOT-Tair)/(R_conv_in+R_cond+R_conv_ex);
542
543         end
544     end
```

A.5 Mass_sizing2.m

```
1   function [state] = Mass_sizing2(state)
2   % MASS_SIZING2
3   %   Mass_sizing2 is a function that estimates the rocket
4   %   mass and size. It is used for design optimization
5   %   using "Multidisciplinar7" as the main function.
6
7
8   if ~isnan(state.RCC_in)
9       %% INPUTS %%
10      length = num2cell(state.length);
11      [Lpay,Lnozzle] = length{:}; %parameters
12      M_pay = state.payload; %parameter
13      N_fins = state.N_fins; %parameter
14      Mfuel = state.Mfuel; %from propulsion
15      Mox = state.Mox; %from propulsion
16      Rcc_in = state.RCC_in; %from propulsion
17      ROT = state.ROT; %from x
18      LCC = state.LCC; %from x
19      Dfin = state.Dfin; %from x
20      Bfin = state.Bfin; %from x
21      bfin = state.bfin; %from x
22      Madd = state.Madd; %from x
23      Madd_cone = state.Madd_cone; %from x
24
25      design = num2cell(state.design); % Distribute design vector
26      [V_tank, ~, ~, ~, ~, ~] = design{:};
27
28
29
```

```
%% Constants
N_nozzle = 0.3; %percentage of nozzle length outside of the tube
t_CC = 0.006;
t_OT = 0.005;
t_tube = 0.002;
t_cone = 0.0024;
t_fins = 0.002;
rho_CC = 2700; % Aluminum 6061 T6 density
%rho_CC = (3*2700+3*1329)/6; % Approximate density of Aluminum 6061 T6 casing with phenolic paper liner
%rho_OT = 2700;% Aluminum 6061 T6 density
rho_OT = (1.5*2700+2.6*1750)/4.1; % = 2097.56 kg/m^3 Approximate density of Aluminum 6061 T6 inner liner wrapped with carbon fibre composite
%rho_fins = 340; %Carbon fiber-foam sandwich (14 mm) density
rho_fins = 1750;
rho_tube = 1750; %Carbon fiber epoxy density
rho_cone = 1800; %Fiberglass epoxy density

%% Calculation
%Diameter
Rcc = Rcc_in + t_CC;
r_OT = ROT - t_OT;
if Rcc>ROT
    R_tube = Rcc + t_tube;
else
    R_tube = ROT + t_tube;
end
De = 2*R_tube;


%Length
LOT = (V_tank-(4/3)*pi*r_OT^3)/(pi*r_OT^2)+2*ROT; %Hemispherical Head Cylinder approximation
Lcone = 1.35*Lpay + De;
L_tube = LOT + LCC + Lnozzle*(1-N_nozzle);
L_rocket = (L_tube + Lcone + Lnozzle * N_nozzle)/0.59394036 - 1.32444; %from the linear regression
L_rec = 0.2404*L_rocket - 0.4193;
```

```
64      L_components = L_rec + (0.6891*L_rec - 0.0784);
65      L_tube = L_tube + L_components;
66      L_rocket = L_tube + Lcone + Lnozzle*N_nozzle;
67
68      %Transversal area
69      Atrans = pi*((De+Dfin)/2)^2;
70
71      %Volume
72      V_CC = LCC*pi*(Rcc^2-Rcc_in^2);
73      V_OT = LOT*pi*(ROT^2-r_OT^2)+(4/3)*pi*(ROT^3-r_OT^3); %Hemispherical
            Head Cylinder approximation
74      V_tube = L_tube*pi*(R_tube^2-((R_tube-t_tube)^2));
75
76      %Cylindrical components
77      M_CC = rho_CC*V_CC;
78      M_OT = rho_OT*V_OT;
79      M_tube = rho_tube*V_tube;
80
81      % Nose cone
82      g_cone = sqrt(Lcone^2 + R_tube^2);
83      A_cone = pi*R_tube*g_cone;
84      V_cone = (pi*R_tube^2*Lcone)/3; %zero wall thickness approximation
85      Factor_cone = 1.3;
86      M_cone = A_cone*t_cone*rho_cone*Factor_cone;
87      M_cone = M_cone + Madd_cone; %Add weight for stability
88
89      % Fins
90      A_fins = N_fins*(Bfin+bfin)*Dfin/2;
91      M_fins = A_fins*t_fins*rho_fins;
92
93      % Mass estimation
94      M_prop = Mfuel+Mox;
95      Mass_structural = 0.434866*M_prop + 259.0147*De - 13.2621; % Calculate
            structural mass from empirically fitted equation
96      M_struct_components = Mass_structural-M_tube-M_cone-M_fins-M_CC-M_OT+
            Madd;
97      M_nozzle = real(0.15*M_CC);
98      M_aero = M_tube+M_cone+M_fins+M_CC+M_OT+M_nozzle;
99      M_rocket = real(M_prop+M_aero+M_pay+M_struct_components);
```

```
100
101     %% outputs
102     state.mass = M_rocket;
103     state.Ltube = L_tube;
104     state.Lrocket = L_rocket;
105     state.De = De;
106     state.Atrans = Atrans;
107     state.MCC = M_CC;
108     state.MOT = M_OT;
109     state.Mtube = M_tube;
110     state.LOT = LOT;
111     state.Lcomponents = L_components;
112     state.Mcomponents = M_struct_components;
113     state.Mfins = M_fins;
114     state.Mnozzle = M_nozzle;
115     state.Mcone = M_cone;
116     state.Vcone = V_cone;
117     state.RCC = Rcc;
118     state.Lcone = Lcone;
119
120 else
121     state.mass = NaN;
122     state.Ltube = NaN;
123     state.Lrocket = NaN;
124     state.De = NaN;
125     state.Atrans = NaN;
126     state.MCC = NaN;
127     state.MOT = NaN;
128     state.Mtube = NaN;
129     state.LOT = NaN;
130     state.LCC = NaN;
131     state.Dfin = NaN;
132     state.ROT = NaN;
133     state.Bfin = NaN;
134     state.bfin = NaN;
135     state.Lcomponents = NaN;
136     state.Mcomponents = NaN;
137     state.Mfins = NaN;
138     state.Mnozzle = NaN;
```

```
        state.Mcone = NaN;
        state.RCC = NaN;
        state.Lcone = NaN;
    end
end
```

A.6 Aerodynamics.m

```
function [state] = Aerodynamics(state)
% AERODYNAMICS
%   Aerodynamics is a function that simulates the rocket
%   trajectory and stability. It is used for design
%   optimization using "Multidisciplinar7" as the main
%   function.
%% INPUTS

length_rocket = num2cell(state.length);
[Lpay,Lnozzle] = length_rocket{:}; %parameters
Mpay = state.payload; %parameter
n_fins = state.N_fins; % number of fins, parameter
Lcone = state.Lcone; %from sizing
Mcomponents = state.Mcomponents; %from sizing
Mfins = state.Mfins; %from sizing
Mnozzle = state.Mnozzle; %from sizing
Mcone = state.Mcone; %from sizing
Ltube = state.Ltube; %from sizing
De = state.De; %from sizing
Mtube = state.Mtube; %from sizing
MOT= state.MOT; %from sizing
MCC = state.MCC; %from sizing
Mtotal = state.mass; %from sizing
LOT = state.LOT; %from sizing
Lcomponents = state.Lcomponents; %from sizing
Dfin = state.Dfin; %from x
Bfin = state.Bfin; %from x
bfin = state.bfin; %from x
LCC = state.LCC; %from x
Mox = state.Mox; %from propulsion
Mfuel = state.Mfuel; %from propulsion
```

```
32
33
34 %% CG and CP
35 %Areas
36 A_fin_tri = (Bfin-bfin)*Dfin/2; %trapezoidal
37 A_fin_quad = bfin*Dfin;
38
39 %Distances (with the nose cone as reference)
40 d_cone = 2*Lcone/3;
41 d_tube = (Lcone + Ltube/2);
42 d_fin_tri = Lcone+Ltube-bfin-(Bfin-bfin)/3;
43 d_fin_quad = (Lcone+Ltube-bfin/2);
44
45 %Center of pressure Barrowman Equation
46 CN_nose = 2;
47
48 R = De/2;
49 Lf = (Dfin^2 + (Bfin/2 - bfin/2)^2)^0.5;
50 x_B = Lcone + Ltube - Bfin;
51 x_R = Bfin - bfin;
52
53 CN_fins = (1+ R/(Dfin + R))*(4*n_fins*(Dfin/De)^2)/(1+(1+(2*Lf/(Bfin+bfin))
       ^2)^0.5);
54 x_fins = x_B + x_R/3*(Bfin+2*bfin)/(Bfin+bfin) + (Bfin + bfin - Bfin*bfin/(
       Bfin+bfin))/6;
55
56 CP = real((CN_nose*d_cone + CN_fins*x_fins)/(CN_nose+CN_fins));
57
58 %Additiconal distances
59 d_pay = Lcone + Lpay/2;
60 d_components = Lcone + Lpay + Lcomponents/2;
61 d_OT = Lcone  + Lpay + Lcomponents + LOT/2;
62 d_CC = Lcone + Lpay + Lcomponents + LOT + LCC/2;
63 d_nozzle = Lcone + Lpay + Lcomponents + LOT + LCC + Lnozzle/2;
64 d_OT_i = d_OT - LOT/2;
65
66 %Fuel and oxidizer on CC and OT
67 M_OT = MOT + Mox;
68 M_CC = MCC + Mfuel;
```

```
69
70 %Center of gravity
71 CG = real((d_cone*Mcone + d_tube*Mtube + d_fin_tri*A_fin_tri*Mfins/(
       A_fin_tri+A_fin_quad) + d_fin_quad*A_fin_quad*Mfins/(A_fin_tri+
       A_fin_quad) + d_components*Mcomponents + d_pay*Mpay + d_OT*M_OT + d_CC*
       M_CC + d_nozzle*Mnozzle)/Mtotal);
72
73 %% Stability
74 SM = real((CP-CG)/De); %Static Margin, CP&CG measured according to the nose
        cone
75
76
77 %% OUTPUTS
78 state.CP = CP;
79 state.CG = CG;
80 state.d_OT_i = d_OT_i;
81 state.SM = SM;
82 end
```

A.7 Trajectory.m

```
1  function [state] = Trajectory(state)
2  % TRAJECTORY
3  %   Trajectory is a function that simulates the rocket
4  %   trajectory (position, velocity and acceleration). It is used for design
5  %   optimization using "Multidisciplinar7" as the main
6  %   function.
7  %
8  % Miguel Morgado (Instituto Superior Tecnico) - 2021
9  % Based on the work of Michael Pearson (2016) and Benjamin Klammer (2019)
10 %
11 %------------------------------------------------------------------------%
12 % Function Description
13 %------------------------------------------------------------------------%
14 % Input is struct 'state', which must contain fields:
15 %   'De' containing the outer diameter of the rocket in m
16 %   'mass' containing the total mass of the rocket in kg
17 %   'Lrocket' containing the lenght of the rocket in m
18 %   'launchAlt' containing the launch elevation in m
```

```
19 %    'p_exit' containing the nozzle design ambient pressure
20 %    'Ath' containing the nozzle throat area in m^2
21 %    'CG' containing the center of mass position from the nose cone in m
22 %    'AR' containing the nozzle area ratio
23 %    'SM' containing the rocket Static Margin
24 %    'TC' containing an array with burn time in column 1, thrust data in
25 %     column 2, mass in column 3, and with the first row containing total
26 %     impulse in column 1, average thrust in column 2, and total mass
27 %     including propellant and case in column 3
28 %-------------------------------------------------------------------------%
29 %
30
31 %% INPUTS
32
33 launchAlt = state.launchAlt; %parameter
34 launchAngle = state.launchAngle; %parameter
35 rail_length = state.rail_length; %parameter
36 De = state.De; %from sizing
37 rocket_mass = state.mass; %from sizing
38 p_exit = state.p_amb; % Pa, Ambient pressure used in Propulsion for thrust
       correction, parameter
39
40 area = pi*(De/2)^2;
41 t_po = 0; % seconds, pitch-over manoeuvre elapsed time
42 t_pd = 0; % seconds, pitch decay manoeuvre elapsed time
43
44 if isfield(state, 'SM')
45     SM = state.SM; %from aerodynamics
46     Lrocket = state.Lrocket; % m, from sizing
47     CG = state.CG; % m, from aerodynamics
48     TC = state.TC; %from propulsion
49     A_exit = state.AR*state.Ath; % Nozzle exit area for thrust correction,
          should come directly from x
50     gain = state.gain; %pitch controller gain %range: -0 -> -1
51     deltat_po = state.deltat_po; % seconds, pitch-over manoeuvre duration %
          range: 0 -> 6
52     delta_theta = state.delta_theta; % rad, pitch-over angle %range: -8 deg
           -> +8 deg
53     deltat_pd = 4*deltat_po; % seconds, pitch decay manoeuvre duration
```

```
54      if state.TVC == 0
55          gain = 0;
56      end
57  else % to run this function outside the Rocket Optimization program
58      SM = 0;
59      Lrocket = 70; % m
60      CG = 40; % m
61      gain = -0.05; %pitch controller gain %range: -0 -> -1
62      deltat_po = 1; % seconds, pitch-over manoeuvre duration %range: 0 -> 6
63      delta_theta = -3*pi/180; % rad, pitch-over angle %range: -8 deg -> +8
            deg
64      deltat_pd = 4*deltat_po; % seconds, pitch decay manoeuvre duration
65      TC = state.motor;
66      A_exit = pi*(0.92/2)^2; % Nozzle exit area for thrust correction
67  end
68
69  %DATCOM IMPORT FROM FILE
70  %aux=datcomimport("Datcom_output.out");
71  %datacom=aux{1,1};
72
73  %DIRECT IMPORT
74  load("datacom.mat", 'datacom');
75
76  %% POSITION AND ATTITUDE ESTIMATION
77
78  if ~isnan(TC)
79      [alt,vel,accel,time,~,~,~,~,~,fpa1,~] = motion(TC,datacom);
80      alt = alt-launchAlt; % Subtract launch altitude to get AGL altitude
81      [maximum_alt,time_apogee_index] = max(alt);
82
83      %%% PLOTS
84      %time_to_apogee = time(time_apogee_index);
85      %SOoutput(alt,x1,vel,accel,fpa1,time,maximum_alt,time_to_apogee);
86
87      %OUTPUTS
88      state.alt = alt;
89      state.vel = vel;
90      state.accel = accel;
91      state.apogee = maximum_alt;
```

```
92      state.time_aero = time(end);
93      state.time_to_apogee = time_apogee_index;
94      state.fpa = fpa1;
95
96  else
97      state.apogee = NaN;
98      state.vel = NaN;
99      state.alt = NaN;
100     state.accel = NaN;
101 end
102
103 %-------------------------------------------------------------------------%
104 %-------------------------------------------------------------------------%
105 %% Sub Functions                    /   |
106 %                            ,------'    '-------..
107 %                           (_____________________-+
108 %-------------------------------------------------------------------------%
109 %-------------------------------------------------------------------------%
110
111 %-------------------------------------------------------------------------%
112 %% Rocket motion function
113 %-------------------------------------------------------------------------%
114     function [z,vel,accel,time_vec,x,x_dot,z_dot,x_2dot,z_2dot,fpa,mass] =
            motion(TCurve,AC)
115
116         %%% Thrust and propellant mass curves interpolation %%%
117         T = griddedInterpolant(TCurve(2:end,1),TCurve(2:end,2),'spline'); %
                spline object that interpolates the propulsion thrust data
118         propellant_mass = griddedInterpolant(TCurve(2:end,1),TCurve(2:end
                ,3),'spline'); %spline object that interpolates the propellant
                mass data
119
120         %%% Preallocation %%%
121         x_dot = zeros(1,100); %m/s
122         z_dot = zeros(1,100); %m/s
123         vel = zeros(1,100); %m/s
124         x_2dot = zeros(1,100); %m/s^2
125         z_2dot = zeros(1,100); %m/s^2
126         accel = zeros(1,100); %m/s^2
```

```
127         x = zeros(1,100); %m
128         z = zeros(1,100); %m
129         theta_dot = zeros(1,100); %rad/s
130         theta = zeros(1,100); %rad
131         theta_2dot = zeros(1,100); %rad/s^2
132         fpa = zeros(1,100); %rad
133         AoA = zeros(1,100); %rad
134         I_rocket = zeros(1,100); %kgm^2
135         time_vec = zeros(1,100); %s
136         mass = zeros(1,100); %kg
137
138         %%% Initial condition values and Definitions %%%
139         z(1) = launchAlt; %m
140         theta(1)=launchAngle; %rad
141         fpa(1)=launchAngle; %rad
142         mass(1) = rocket_mass; %kg
143         dt=0.05; %seconds
144         terminate_flag = 0;
145         i=1;
146
147         %%% main loop %%%
148         while terminate_flag == 0
149             %%% calculate atmospheric conditions %%%
150             if z(i)<150000
151                 h = 6371000*z(i)/(6371000+z(i)); %h - geopotential height,
                        r0 = 6371000 m, average radius of Earth
152                 [~, a, p_act, rho] = atmoscoesa(h,'None');
153                 M = vel(i)/a; %Mach number
154             else
155                 M = 1000;
156                 p_act = 0;
157                 rho = 0;
158             end
159
160             I_rocket(i) = (mass(i)/12)*(3*(De/2)^2+Lrocket^2); %I_yy
161             % The above assumes the CG is in the center of the rocket
162
163             %%% lookup aerodynamic coefficients %%%
164             [~,indexA] = min(abs(AC.alpha(:)-(AoA(i)*180/pi))); % Find the
```

```
    index of the closest mach number in Cd table to actual mach
    number
165 [~,indexM] = min(abs(AC.mach(:)-M)); % Find the index of the
    closest mach number in CL table to actual mach number
166 [~,indexH] = min(abs(AC.alt(:)-z(i)));
167
168 Cl = AC.cl(indexA,indexM,indexH,7); %four selection parameters:
     alpha, mach number, altitude, configuration
169 Cd = AC.cd(indexA,indexM,indexH,7);
170
171 if abs(AoA(i))> pi/2
172     Cl = -Cl; %If the rocket "flips", the Lift force points in
        the opposite direction
173 end
174
175
176 %%% calculate gravitational acceleration %%%
177 g = grav(z(i)+6371000); % gravity decreases as you get farther
    from earth
178
179 %%% increments time %%%
180 time_vec(i+1) = time_vec(i) + dt;
181
182 %%% Thrust and mass update %%%
183 if time_vec(i)<=TCurve(end,1)
184     mass(i+1) = mass(i) - (propellant_mass(time_vec(i))-
        propellant_mass(time_vec(i+1))); %calculates the change
        in mass
185     Thrust = T(time_vec(i)) + (p_exit - p_act)*A_exit; %thrust
        correction with altitude
186 else
187     mass(i+1) = mass(i);
188     Thrust = 0;
189 end
190
191 if Thrust ~= 0 && gain ~= 0 %don't waste time running these
    calculations if there is no active TVC or thrust to vector
192     %%% TVC Pitch guidance %%%
193     if z(i)<launchAlt+rail_length
```

```
194                 thetad = launchAngle;
195             elseif deltat_po == 0 || delta_theta == 0
196                 thetad = launchAngle;
197             elseif t_po <= deltat_po %until the pitch-over phase
                    duration is exceeded
198                 t_po = t_po + dt;
199                 thetad = fpa(i) + (delta_theta*t_po/deltat_po);
200             elseif t_pd <= deltat_pd && deltat_pd ~= 0
201                 t_pd = t_pd + dt;
202                 thetad = fpa(i) + delta_theta*exp(-3*t_pd/deltat_pd);
203             else
204                 thetad = fpa(i);
205             end
206
207
208             if z(i)>launchAlt+120
209                 epsilon = gain*(thetad-theta(i)); %rad, TVC angle
210                 if epsilon > 2*pi/180 %max vectoring angle
211                     epsilon = 2*pi/180;
212                 elseif epsilon < -2*pi/180 %min vectoring angle
213                     epsilon = -2*pi/180;
214                 end
215             else
216                 epsilon = 0;
217             end
218         else
219             epsilon = 0;
220         end
221
222         %%% Auxiliary calculations and rotational dynamics %%%
223         q = 0.5*rho*area*(x_dot(i)^2+z_dot(i)^2); %dynamic pressure
                times reference area
224         torque = - q*De*SM*(Cl*cos(AoA(i))+Cd*sin(AoA(i)))-(Lrocket-CG)
                *Thrust*sin(epsilon);
225         theta_2dot(i+1) = torque/I_rocket(i);
226         theta_dot(i+1) = real(theta_2dot(i+1)*dt+theta_dot(i));
227
228
229         %%% calculate acceleration %%%
```

```
230             x_2dot(i+1) = (Thrust*cos(AoA(i)+epsilon+fpa(i)) - q*(Cd*cos(
                    AoA(i)+fpa(i))+Cl*sin(AoA(i)+fpa(i))))/mass(i);
231             z_2dot(i+1) = (Thrust*sin(AoA(i)+epsilon+fpa(i)) - q*(Cd*sin(
                    AoA(i)+fpa(i))-Cl*cos(AoA(i)+fpa(i))))/(mass(i)) - g;
232
233             %%% UPDATE VELOCITIES %%%
234             x_dot(i+1) = real(x_2dot(i+1)*dt+x_dot(i));
235             z_dot(i+1) = real(z_2dot(i+1)*dt+z_dot(i));
236
237             %%% UPDATE POSITIONS %%%
238             x(i+1) = real(x_dot(i+1)*dt+x(i));
239             z(i+1) = real(z_dot(i+1)*dt+z(i));
240
241             %%% check if rocket hits ground %%%
242             if z(i+1) < launchAlt
243                 z(i+1) = launchAlt;
244                 z_dot(i+1) = 0;
245                 x_dot(i+1) = 0;
246                 if Thrust == 0
247                     terminate_flag = 1;
248                 end
249             end
250
251
252             %%% Update Pitch %%%
253             theta(i+1) = real(theta_dot(i+1)*dt+theta(i)); %rad, pitch
                    angle
254
255             if theta(i+1) > pi
256                 theta(i+1)=theta(i+1)-(2*pi);
257             end
258             if theta(i+1) <= -pi
259                 theta(i+1)=theta(i+1)+(2*pi);
260             end
261
262
263             %%% Update Flight Path Angle %%%
264             fpa(i+1) = atan2(z_dot(i+1),x_dot(i+1)); %rad, flight path
                    angle
```

```
265
266             if x_dot(i+1) == 0 && z_dot(i+1) == 0
267                 if i==1
268                     fpa(i+1) = launchAngle; %The condition below should
                            already guarantee this
269                 else
270                     fpa(i+1) = fpa(i);
271                 end
272             end
273
274             %%% Update Angle of Attack %%%
275             AoA(i+1)=theta(i+1)-fpa(i+1); %rad, Angle of Attack
276
277             if AoA(i+1) > pi
278                 AoA(i+1)=AoA(i+1)-(2*pi);
279             end
280             if AoA(i+1) <= -pi
281                 AoA(i+1)=AoA(i+1)+(2*pi);
282             end
283
284
285             %%% Compute output vectors %%%
286             vel(i+1) = (x_dot(i+1)^2+z_dot(i+1)^2)^(1/2);
287             accel(i+1) = (x_2dot(i+1)^2+z_2dot(i+1)^2)^(1/2);
288             i=i+1;
289
290             %%% Error
291             if isnan(z(i)) || isinf(z(i)) || i> 120000
292                 terminate_flag = 2;
293             end
294         end
295     end % end motion function
296
297 end % end main program
298 %-------------------------------------------------------------------------%
299 %-------------------------------------------------------------------------%
300
301 %% Calculates gravitational acceleration at altitude
302 function g = grav(r)
```

```
303 g = ((6.67408E-11)*(5.972E24))/(r^2);
304 end
305
306 %% Outputs data and creates plots
307 function SOoutput(alt,x,vel,accel,fpa,time,max_alt,t_apogee)
308 data(1,:) = cellstr(['km    ';'km    ';'km/s  ';'m/s^2';'s      ']);
309 data(2,:) = num2cell([max_alt,max(abs(x)),max(vel),max(accel),t_apogee]);
310
311 % *** To output performance table, uncomment this section ***
312 array2table(data,'VariableNames'...
313         ,{'Apogee' 'Downrange' 'Max Vel' 'Max Accel' 'Time To Apogee'},'
            RowNames'...
314         ,{'Units','Value'})
315
316 % [~,time_apogee_index] = max(alt);
317 % t = linspace(0,t_apogee,length(alt(1:time_apogee_index)));
318 %
319 % figure(1);
320 % tiledlayout(3,2)
321 %
322 % top_left = nexttile;
323 % plot(t,alt(1:time_apogee_index))
324 % hold on
325 % plot([0,t(end)],[100000,100000],':')
326 % xlim([0,t(end)]);
327 % ylim([min(alt)*1.1,max(alt)*1.1]);
328 % ylabel(top_left,{'Altitude AGL (m)'});
329 %
330 % top_right = nexttile;
331 % plot(t,x(1:time_apogee_index))
332 % hold on
333 % plot([0,t(end)],[100000,100000],':')
334 % xlim([0,t(end)]);
335 % ylim([min(x)*1.1,max(x)*1.1]);
336 % ylabel(top_right,{'Downrange (m)'});
337 %
338 % middle_left = nexttile;
339 % plot(t,vel(1:time_apogee_index));
340 % xlim([0,t(end)]);
```

```
341 % ylim([0,max(vel)*1.1]);
342 % ylabel(middle_left,{'Velocity (m/s)'});
343 % xlabel(middle_left,'Time (s)');
344 %
345 % middle_right = nexttile;
346 % plot(t,accel(1:time_apogee_index));
347 % xlim([0,t(end)]);
348 % ylim([0,max(accel)*1.1]);
349 % ylabel(middle_right,{'Acceleration (m/s^2)'});
350 % xlabel(middle_right,'Time (s)');
351 %
352 % bottom_left = nexttile;
353 % plot(t,fpa(1:time_apogee_index));
354 % xlim([0,t(end)]);
355 % ylim([-pi,pi]);
356 % xlabel(bottom_left,'Time (s)');
357 % ylabel(bottom_left,{'Flight Path Angle (rad)'});
358 % hold off
359 %
360 % figure(2);
361 % plot(x(1:time_apogee_index),alt(1:time_apogee_index));
362 % xlabel('Downrange (m)');
363 % ylabel('Altitude (m)');
364
365 end
```