Flight Dynamics Bible

Volume I:
Everything about 6 Degree of Freedom Modeling

Hudson Reynolds
Author

Preston Wright
Co-Author

May 14, 2025

Part of a

Flight Dynamics

Series

This document is dedicated to Purdue Space Program Liquids. PSP Liquids has given us a vast amount of experience and means to explore the realm of Flight Dynamics. We hope that we can contribute to the wealth of information provided by PSP with this document.

Contents

Code Listings

Preface

Preston and I began working on the Flight Dynamics sub-team in the fall of 2023. From May 2024 to January 2025, we focused our efforts on creating a 6-Degree of Freedom (DoF) model from scratch. During this time, we learned a lot about the mathematics and physics involved in creating such a model. We were often faced with frustration with different conventions and poor explanations of certain topics, all of which were exacerbated by the difficulty of what we were learning. This made much of the material intractable. In this document, we hope to provide detailed explanations of all the necessary concepts in one place to make this journey easier for those who may need it in the future.

Furthermore, we hope that this document can be used by those outside the realm of flight dynamics to better understand the methodologies and inner workings of a 6-DoF model. Often, we have seen that a 6-DoF model is treated as a 'black box', where inputs go in and outputs come out. We hope that this document can provide a level of understanding adequate enough to allow greater collaboration between those working within and outside of flight dynamics.

In this document, we will be using excerpts of MATLAB script in some examples (code less than a page is included inline and longer code is in the Appendix) but our hope is that our explanations are thorough enough to facilitate the creation of a 6-DoF in any coding language. MATLAB facilitates the use of quite short code because of the number of built-in functions, but we include derivations for most of these equations or sources describing these derivations.

To use this document to its fullest extent, we recommend that you write scripts as you learn and use this document for reference and as a guide as you build or improve your own models. Doing the practice problems in each chapter will also greatly aid in your understanding.

In this first volume, we cover primarily the translational dynamics and the basics of rotational dynamics needed to set up a 6-DoF model. We mainly focus on the implementation of these equations in MATLAB and best practices. The methods of deriving equations of motion are also heavily discussed for their importance. These are, in our opinion, the fundamental things that need to be understood before working

on other aspects of a flight dynamics model. As such, these subjects comprise the large majority of both the effort and the length of the first volume.

Because of the depth of the subjects described here, we have decided to cover most of attitude dynamics in a separate volume. In this volume, we mostly describe how attitude dynamics can be implemented in MATLAB. Volume II contains most of the theory about attitude dynamics. This second volume is especially useful for the mathematically inclined reader or those who want to implement attitude dynamics math in another programming language.

As a note, while our work on flight dynamics has mainly focused on modeling rockets, we hope that this document can be used more widely for a variety of 3-DoF and 6-DoF models. As such, we have attempted to include content that is useful for other systems, such as aircraft modeling and orbital dynamics, insofar as we have experience in these areas. That being said, to limit the scope of this document, we have decided to explore these concepts in the context of rocket modeling, and hope that our explanations provide enough clarity for the intrepid reader to explore modeling in other domains.

Some other areas are also covered in this document but are curtailed because Preston and I do not have the experience to give full authority on these subjects. However, as we learn more, this document is growing every day. In future volumes, we hope to expand and add some of these other sections, especially those related to aerodynamics, energy methods, and control theory, to better serve those in the future who may hope to recreate the work we have done here.

Acknowledgments

- We would like give special thanks to Prof. Cunningham and Prof. Frueh for their help in reviewing our 6-DoF Model. Professor Frueh's notes have been especially helpful in compiling the information present in this document. We would also like to thank everyone who has taken their time to read through this document and help us make corrections and edits.

Notation and Standards

Mathematical notation for the subjects that we discuss is largely formalized and commonly used, but there are some areas where we may use notation slightly unfamiliar or different to what you have seen in the past. You may also see different notation when looking at papers and literature for future work on the 6-DoF. To remove any doubt, we describe the notation that we use here at the beginning for quick reference.

We also choose to use notation that is used in the Purdue AAE classes. Some of these may be unfamiliar if you have not encountered this coursework yet, but we do our best to describe notation throughout this document and in this section.

For derivatives with respect to time, we will use the compact Newtonian notation. For example, $\dot{x} = \frac{dx}{dt}$, $\ddot{x} = \frac{d^2x}{dt^2}$, and so on. Generally, we avoid the prime notation such as x' as this notation does not make clear what we are taking the derivative with respect to. The prime notation is only used in specific circumstances where the other notation is easily applicable.

For vectors, any vector with a hat like $\hat{x}$ refers to a vector with a magnitude of 1 (normalized vector). These are often seen as basis vectors or unit vectors describing the direction of a force. Vectors with a 'arrow' hat like $\vec{x}$ refers to a general vector with a non-normalized magnitude. Vectors with a non-normalized magnitude may also be represented by a bold-faced symbol. When taking the time derivative of a vector, we prefer to write using this boldface notation, such as $\dot{\mathbf{x}}$ instead of $\dot{\vec{x}}$.

Vectors and Matrices

The whole of the vector quantities used in this document are too great to enumerate in full, but we include the most important here. Notably, we stray from the typical notation of $\vec{r}_{P/O}$ in favor of $\vec{r}^{\,op}$ because of its usage in Purdue AAE.

Common usages are documented here:

$\vec{F}$	force
$\vec{a}$	acceleration
$\vec{V}$ or $\vec{v}$	velocity
$\vec{r}^{\,op}$	position vector from o to p
$\vec{X}$	state vector
$\vec{M}^o$	moment about point o
${}^i\vec{x}$	inertial vector quantity
$\vec{\omega}$	angular velocity
$\vec{\alpha}$	angular acceleration

Scalar Quantities and Symbols

J	functional
δ	variation
S	action
L	Lagrangian
q	generalized coordinate
$\dot{q}$	generalized position
k	spring constant
$\mathcal{M}$	number of degrees of freedom
T	kinetic energy
U	potential energy
$\mathcal{H}$	Hamiltonian
Re	Reynolds number
M	Mach number
α	angle of attack
C_x	aerodynamic force coefficient
Δt	timestep

Coding Standards

An ideal 6-DoF not only models the motion of a vehicle through the atmosphere as accurately as possible but is also well-structured and easily readable. Strict adherence to an agreed upon and appropriate set of coding standards is essential for a successful

program. These cover everything from naming conventions to heading structure, and make the integration of codes from multiple contributors more seamless.

For short pieces of code written inline, we use a mono-spaced font with the appearance of `code snippet` as shown. Longer code is included in listings with line numbers using the same mono-spaced font.

Generally, we use the following conventions in our coding practice:

- Variables utilize camelCase (symbolic variables are the only exception, where an underscore may be used)
- File names and functions utilize PascalCase

MATLAB script file headers are of the following format:

Code Listing 1: Script Header

```
% PSP FLIGHT DYNAMICS:
%
% Title:
% Author: __ - Created: 9/21/2024
% Last Modified:
%
% Description:
%
% Inputs:
%
% Outputs:
%
```

CHAPTER 1

Overview

"If you wish to make an apple pie from scratch, you must first invent the universe."

– Carl Sagan

In much the same way, we must construct all of the mathematical background required to make a 6-DoF from the ground up. This chapter details the general requirements and an outline of the process we will be following.

1.1 Document Outline

A general overview of the scope of this document is given in the preface. Here, we will go into more detail about the layout of each section. Our goal is to present this document in a similar manner to a textbook while still remaining approachable and interesting to read.

Our pedagogical approach is strongly reliant on examples and trying problems. We find that this is the best method for readers to learn the content.

In each chapter (aside from this overview chapter), there will be some additional content at the end that may be especially useful for quick reference. For quick reference, we include a 'Key Ideas' section that may be used to quickly reference the big takeaways from each chapter.

Additionally, a 'Further Reading and Notes' is included which the curious reader may benefit from. Many topics are covered in this document; each of which deserve their own book-length discussion. Because we have neither the desire nor capability

to write such a long document, we have linked these beneficial resources whenever possible. It is certainly not necessary to read any/all of these sources to comprehend the content here, but they may prove useful for deeper research into topic areas that interest the reader.

The end of some sections also provides several practice problems to ensure the key ideas of each topic are understood by the reader. These problems are mainly to validate the readers understanding and provide an example of using the presented content.

For readers new to this content, it is expected that some problem may take some consultation with other sources to complete. However, we have attempted to make these problems challenging while not being out of reach. These problems are designed to make the reader apply the knowledge and truly learn. Generally, each subsequent problem in a section is more challenging than the last.

1.2 Necessary Background

The goal of this document is to provide a comprehensive overview of the process and means of creating, maintaining, and building on a 6 Degree of Freedom model. Although we strive to make the document as easy as possible to understand, there are some instances where technical jargon is necessary and a strong understanding of mathematical concepts is required.

The first time an important technical jargon is used, we will *italicize* the word for recognition. It is important to learn these terms to effectively and concisely describe your work. That being said, we attempt to keep the usage of jargon to a minimum to make this text approachable. A glossary with these terms has also been added so that all of these words are included in a single location.

To aid in comprehensibility, we have added sources that we find especially useful and avoided the use of superfluous sources. These sources are hyperlinked for ease of access. In addition, useful links and footnotes have been highlighted in red boxes to be easily located.

To fully comprehend and successfully use this document, it is recommended that you have a good understanding of calculus, linear algebra, and calculus-based physics. A rudimentary understanding of differential equations and dynamics will also prove

useful but is not strictly required. Topics covered in AAE 203 / Introduction Dynamics will be used.

Some concepts will arise in this document that are derived from graduate-level coursework and complex mathematics. In these instances, we may not attempt to derive the background and full understanding, but rather only what is necessary for building a 6-DoF model. In these cases, we also try to link to source material if you are inclined to learn more about those topics.

1.3 Why 6-DoF?

Let's start off with some terminology in this section. A Degree of Freedom (DoF) refers to the number of ways an object can move in space. A more technically precise and rigorous definition defines this as the number of independent parameters that define the motion of a system. This concept is explored more deeply in subsection 3.3.1.

When modeling rigid body dynamics, it is common to see 1-DoF, 3-DoF, and 6-DoF models all used in aerospace applications. Each of these are useful constructs, and sometimes a simulation with more degrees of freedom is not necessarily better, depending on the application.

1.3.1 1-DoF

The first modeling that is often done for the sizing of rockets may use 1-DoF modeling. In this type of model, there is only one axis along which an object may translate or rotate about. In this case of rocket modeling, the only degree of freedom is the vertical altitude of the rocket. The goal of this type of model is mainly to provide a heuristic for determining rocket performance in the very preliminary stages of sizing by estimating the maximum altitude they can achieve.

Another example of this type of model is a simple mass-spring system along one axis. Such models may be useful for estimating the vibrations of bodies or as a first-order approximation of fuel slosh.

We will use the example of a 1-DoF in Section 2.1 to explain the creation of equations of motion in the simplest way. However, these models are often too constrained

to give the most valuable information about a system, which is why 3-DoF and 6-DoF models are more often used, especially in the context of flight dynamics.

1.3.2 3-DoF

After 1-DoF, the next step up is often a 3-DoF model. This type of model should be familiar to all flight dynamics members, as it is the subject of our onboarding project. In a 3-DoF model, there may be any combination of translational degrees of freedom and rotational degrees of freedom.

Often, 3-DoF modeling is utilized when either the rotational dynamics or translational dynamics of an object are considered unimportant. One example of this may be the calculation of an interplanetary transfer. The primary concern of the simulation is not the orientation of the satellite, but rather the location and velocity of the satellite and planets at various points in 3D space.

Another example of this type of model is planar motion. Often, problems in engineering are constrained in some way. Very often, we can consider one of the translational degrees of freedom and two of the rotational degrees of freedom to be non-important to the problem. This simplification allows us to greatly reduce the scope of the problem. One consequence of this action is that rotations are only permitted in a single axis. With two translational and one rotational degree of freedom, we can create useful models of many systems without great complexity.

Lastly, we see 3-DoF models used when we are considering only the rotational dynamics of a system. For objects such as gyroscopes and other spinning bodies, we may choose to model rotational motion in this way.

Although the rest of this document will mainly discuss 6-DoF models, really consider if you need that full complexity for your model! Often, a 3-DoF or 4-DoF model is perfectly adequate for a wide variety of problems without the hassle incurred by a 6-DoF model.

1.3.3 6-DoF

6-DoF models are the most complex representation of a single rigid body mechanical system. In 3D-space, there are only 6 possible degrees of freedom, and we

simulate the dynamics in all of them.

Doing so is quite a difficult challenge, but it is not insurmountable. With careful planning, it is possible to gain an appreciation and understanding of the dynamics of a full 3D system. Doing so allows unrivaled modeling potential. In the case of a rocket, the primary reason that modeling in 6-DoF is so important is because it allows us to analyze the aerodynamic stability of the system. In both 1-DoF and 3-DoF modeling, this important aspect of the system is largely neglected.

1.3.4 Structure Hierarchy of a 6-DoF

The structure of a 6-DoF consists of three main, large "blocks" of code that are executed in sequence. The first is the initialization section. This is where all known, fixed parameters of the vehicle, atmosphere, and physics are initialized to be called later in the program. The second section is the integrator. This block utilizes a certain integration scheme to step through time, calculating the derivatives of the vehicle's *state vector*. Within the integrator, there are three major sub-blocks corresponding to major aspects of the dynamics: the forces, the moments, and the attitude dynamics. The forces section takes on the more familiar particle dynamics – regarding position, velocity, and acceleration - that are seen in a high school physics class. The moments section takes on the torques and rotational forces acting on the system. Lastly, there is attitude dynamics – the orientation of the vehicle in space. This can take the form of either Euler Angles or Parameters, which will be discussed further later. The final block is the visualizer block. This block writes pertinent information to matrices, animates the movement of the vehicle from launch to land, and outputs graphs of important metrics for the user to visualize.

Each of the previously mentioned sections will be discussed in more detail throughout this document.

CHAPTER 2

Newtonian Dynamics

"I can calculate the motion of heavenly bodies, but not the madness of people."

– Isaac Newton

The first step when approaching any physics problem of this type is to consider the Newtonian dynamics of the system. In our case, we define Newtonian dynamics as the translational dynamics of the system as well as the moments acting on the system.

Our definition strays from particle dynamics because we are considering a rigid body which can exhibit rotation. The exploration of this rotation will be explored in much more detail in Chapter 3, Chapter 4, and Volume II and III. Here, we will introduce the methods used in our future analysis.

2.1 The 1-DoF Case

To begin our exploration of Newtonian dynamics, we will return to the example of the 1-DoF which we mentioned earlier. Our goal with this is to use a simple example that you are familiar with from high school physics and bring it up to speed with the terminology and techniques we will use for more complex analysis.

In Figure 2.1 below, we show an example of a very simple 1-DoF system free body diagram. We will use this example to explain the process of calculating the Newtonian dynamics in 1-dimension, which we then generalize to the full 3D case.

In Figure 2.1, a total of three forces are shown, all of which act along the $\hat{y}$-

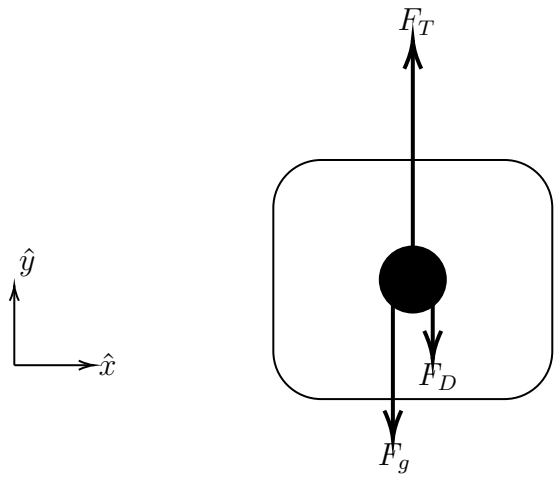


Figure 2.1: 1-DoF Forces on a body

direction. The colinearity of all these forces is what makes this a 1-DoF model. The three forces shown are the thrust force, the drag force, and the gravity force.

Following what you may do in an introductory physics class, we will describe each of these forces via their functional form. These are described in Figure 2.1:

Force	**Functional Form**
$\vec{F}_g$	$-mg$
$\vec{F}_D$	$-\frac{1}{2}\rho_\infty V_\infty^2 S C_D$
$\vec{F}_T$	$F_T \cdot [1 - u(t - t_b)], t \geq 0$

The notation here is important to understand. In future AAE classes at Purdue, this is likely the notation that you will see. Terms with the subscript ∞ denote *freestream* quantities, those that are freely flowing far away from the body of interest. Also of note is the unit step function, $u(t - t_b)$, which is a function which has value zero until reaching time t_b, where it thereafter equals one. The quantity ρ refers to the density of the fluid, and S to the *reference area*. The *drag coefficient* is C_D, a term which encapsulates the very complex nature of the fluid flow around an object. Calculating the value of C_D will be explored more deeply in later sections of this document. For now, we may assume a constant value.

It is helpful to think about these quantities as vectors, because we can add them together to achieve a resultant.[1] In the 1-dimensional case this is quite trivial since all vectors lie along the $\hat{y}$ unit vector, but this will become increasingly important as we move to higher dimensions.

2.1.1 Numerical Integration in the 1-DoF

Now we will move into something that you may have not seen before. Given all the forces, we normally apply Newton's 2nd Law, $\sum \vec{F} = m\vec{a}$, to arrive at an expression for the acceleration, known as an Equation of Motion (EOM), as shown in (2.1). Doing so might look something like this:

$$\begin{aligned} F_T \cdot [1 - u(t - t_b)] - \frac{1}{2}\rho_\infty V_\infty^2 S C_D - mg &= m\vec{a} \\ \vec{a} = \frac{F_T \cdot [1 - u(t - t_b)] - \frac{1}{2}\rho_\infty V_\infty^2 S C_D}{m} - g \end{aligned} \tag{2.1}$$

From here, we use integration to find the velocity and then position of the object with time. Formally, we might define this formally as:

$$\vec{V} = \int \vec{a} dt$$

However, you may notice something strange with our equation. In our expression for acceleration, we need to know the velocity. However, we do not know the velocity without integrating the equation first. Since this dependence is non-linear, this becomes even more complicated to solve. This puts us in a chicken v. egg situation, so we must take a different approach to the problem. [2]

The way we resolve this is to make an approximation of the solution in a process called *numerical integration*. Here, we will outline the simplest type of numerical integration, known as Euler's Method. This section aims to show how to implement Euler's method in code, and the mathematical reasoning will be given a more formal treatment in Section 6.1. We will describe more complex numerical integration schemes and why you might want to use them, but it's good to see the code for a simple case first.

Euler's Method involves discretizing our time interval. You may remember the concept of a Riemann sum from calculus, where you approximate an integral in discrete time steps. We will take a similar approach with Euler's Method. Euler's method is applicable to solving first order ordinary differential equations (ODE's). However, in our problem, we have a second order differential equation, because $\vec{a} = \ddot{y}$.

To resolve this issue, we will make this one differential equation into a system of first-order differential equations. In general, we can convert an nth-order ODE into a system of n first-order ODEs.

In this case, we define a variable, $v = \dot{y}$. Now, we can express equation (2.1) as a system of two differential equations:

$$\begin{aligned} \dot{v} &= \frac{F_T \cdot [1 - u(t - t_b)] - \frac{1}{2}\rho_\infty V_\infty^2 S C_D}{m} - g \\ \dot{y} &= v \end{aligned} \tag{2.2}$$

Now, we will apply Euler's method to solve the problem. We will use the following steps to do so, using equation (2.2) as our example:

1. Rewrite the system in Leibniz notation, where $\dot{x} = \frac{dx}{dt}$.
2. Perform separation of variables, to arrive at $dx = v \cdot dt$.
3. Discretize the equation by 'converting' dx and dt to Δx and Δt.[3]
4. Rewrite Δx as $x_{new} - x_{old}$. We can rearrange the equation as $x_{new} = v \cdot \Delta t + x_{old}$
5. To start, we define an initial state. This initial state will be the first value of x_{old}.
6. Iterate through values of Δt until the desired time of simulation is achieved.

We follow the same process for the integration of acceleration to find the velocity. Of note, we will use the previous value of $\vec{v}$ in the computation of the new acceleration. Note that a more formal definition of Euler's method is given in Section 6.1.

We also show this Euler Integrator example in MATLAB Script 3 below. Note that $\frac{1}{2}\rho_\infty S C_D$ is called k in the script for simplicity. This simplification assumes that ρ_∞ and the C_D are constant, which we will later see is a quite poor assumption but is okay for a first approximation.

Code Listing 2.1: 1 DoF

```
% Euler Method 1DoF

%constants:
m = 1;     % mass [kg]
g = 9.8;   % gravity [m/s^2]
k = 1e-3; % drag const [kg/m]
tb = 5;    % thrust time [s]
Ft = 100; % force of thrust [N]

```

```
% define constant forces
Fg = -m*g;

% pos and vel init:
x = 0;
v = 0;
t = 0;

% array init:
i = 1;

% timestep definition:
dt = 0.01;

while v >= 0
    % variables updated every loop
    Fd = -k*v(i)^2;
    Ft = Ft * (1-heaviside(t(i)-tb));

    % Euler Integration:
    a(i+1) = Ft+Fg+Fd;
    v(i+1) = a(i)*dt + v(i);
    x(i+1) = v(i+1)*dt + x(i);

    % Iteration update
    i = i+1;
    t(i) = i*dt;
end

% plot result:
figure(1)
plot(t,x)

figure(2)
plot(t,v)
```

There are a few important things to note about the MATLAB implementation of the script. Firstly, the only force that is defined outside of the loop is gravity, because the magnitude and direction of this force is independent of the current state of the system. We contrast this with the drag force and the thrust, which must be calculated on every iteration through the algorithm to find an updated value for the force.

We also note the use of the `heaviside` function. This is functionally identical to a unit step function, just a different notation.

2.2 Frame of Reference

Before attempting to describe movement in multiple dimensions, we should first discuss how we can describe important quantities through different lenses. Later, it will become apparent that a vector or problem is easier described or completed through one lens than another. We call these different lenses reference frames.

A frame of reference is an *orthonormal* set within three-dimensional space. The orthonormal nature of the basis vectors that create every frame of reference allows for any other vector in 3-dimensional space to be described as a linear combination of those three basis vectors (hence why we call it a basis for 3-space). This tool becomes incredibly useful within the study of Newtonian and Attitude Dynamics, as every arbitrary vector can be broken down into three set "scalar" directions. Naturally, the question arises of how we determine which set directions to utilize, and so we will be discussing a few frames of reference critical to the creation of a 6-DoF and aerospace engineering as a whole.

2.2.1 The Inertial Frame

The first and most important frame of reference is the *inertial frame*. Typically denoted by e or i, or in our case $(\hat{x}, \hat{y}, \hat{z})$, this motionless frame is the one that allows for the use of Newton's second law in the first place, as the frame itself has "zero" translational and rotational movement.[4] We say "zero" as this motion must be negligible in relation to the vehicle of interest. For example, our specific 6-DoF sets the inertial frame to be anchored within the Earth on the launch pad. And while we are well aware the Earth is moving and rotating through space at a non-zero rate, this motion is negligible relative to the motion of our vehicle which remains close to Earth within the atmosphere for a short period of time. Therefore, we can assume that the Earth acts appropriately as an inertial frame of reference, and Newton's second law is valid for this frame.

Returning to the specific example of a 6-DoF, the directions you set your basis vectors to point in are mostly left to the coder's discretion. If your chosen directions retain the orthonormal nature of a frame of reference, as well as obeying the right hand rule (RHR), the choice is all yours. Within the example we are following, our inertial frame's origin is anchored within the launch pad as previously stated. The first inertial basis vector points up perpendicular to the Earth's surface, the second

inertial basis vector points due East, while the third inertial basis vector follows the right-hand rule and points due North. An image for quick reference is provided in Figure 2.2:

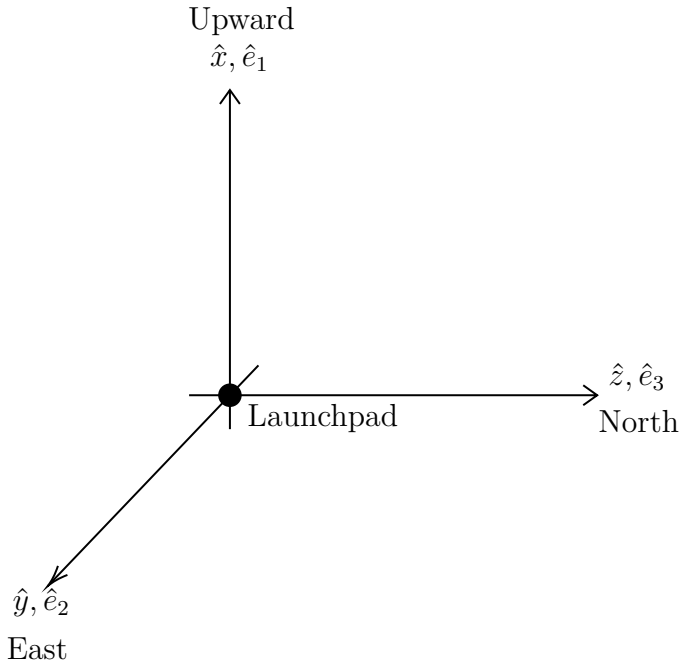


Figure 2.2: Inertial Frame Convention

2.2.2 The Body Frame

The second most important frame of reference is the *body frame*. Typically denoted by b, or in our case $(\hat{X}, \hat{Y}, \hat{Z})$, this frame of reference is always anchored at some point in the vehicle and allows us to describe the state vector in relation to the vehicle itself. This frame is moving and rotating with respect to the inertial frame. For every body frame you create – from a frame for the entire vehicle to a frame for a small electronic part – you want simplicity. This means that in aerospace, it is often the case for a general body frame to have a basis vector pointing along the longitudinal axis of the vehicle, as well as one out a wing. A generic image of such a convention is provided in Figure 2.3 In addition, it means at a zero angle of rotation, every basis vector of the body frame is parallel to the corresponding basis vector in the inertial frame. This is imperative to rotational dynamics, as the offset of each basis vector in the body frame will correspond to the pitch, yaw, and roll of the vehicle. But more on that in

Volume II.

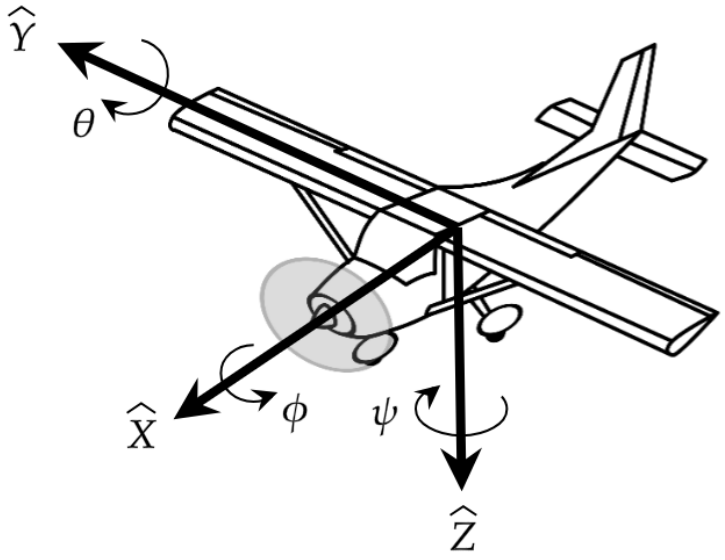


Figure 2.3: Typical Body Frame Orientation

For our current model, the body frame of our 6-DoF has its origin in the nose of the vehicle. It has the first body basis vector, $\hat{X}$, pointing up along the longitudinal axis of the rocket following standard convention. The second basis vector, $\hat{Y}$, points right due East when on the launch pad, and the third, $\hat{Z}$, points due North on the launch pad. We use the cardinal directions of the launchpad as not only are there no conventional wings on the rocket, but it allows the body frame to be defined parallel to the corresponding inertial frame basis vectors. Note that they won't remain pointing in these directions as the rocket moves through space, and they may not even begin parallel to the inertial axis to begin with if the rocket launches from a tilt.

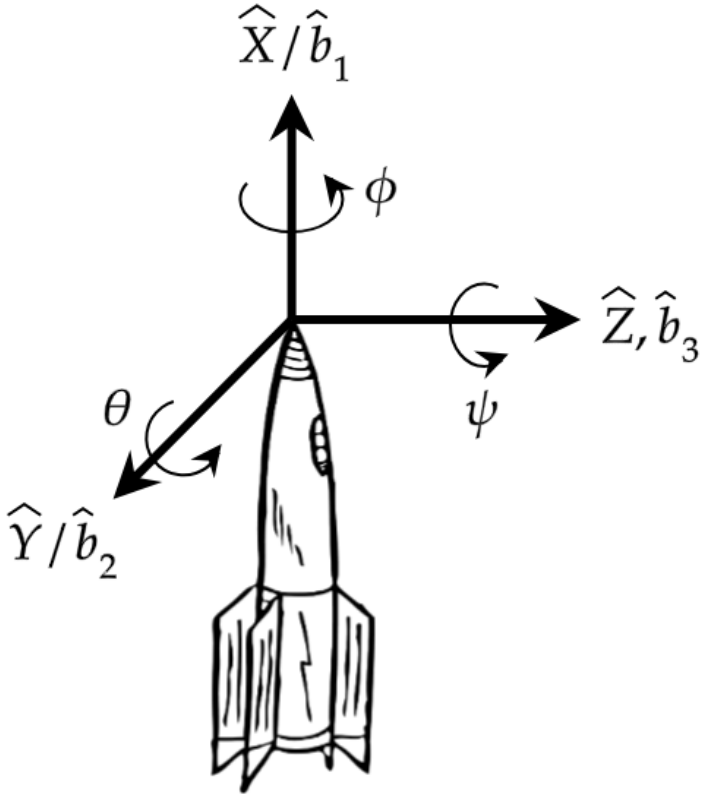


Figure 2.4: Body Frame Convention of our Rocket

2.2.3 The Wind Frame

The last and by far least important frame of reference is the *wind frame*. This frame of reference, like the body frame, is always anchored at some point in the vehicle. Also like the body frame, it moves and rotates with the vehicle and allows for a new way of describing the state vector. Unlike the body frame, the most important definition is not along the longitudinal axis of the vehicle. As indicated by its name, the necessary orientation of a basis vector points in the direction of the free stream velocity. Relating this to the body frame, this translates to an offset from the body frame by the *angle of attack* and sideslip. This frame is often useful for aircraft, but we do not use it explicitly in our 6-DoF. The wind frame itself is in fact skipped over during wind calculations, as the angle of attack and imparted forces/moments are calculated directly onto the body frame. So as this isn't as useful as the other two reference frames, we won't elaborate on this frame any further, and it won't be shown

as much in our example 6-DoF.

2.3 The 3-DoF Case

2.3.1 Vectors in the 3-DoF Case

For particle dynamics, the most complex case is full 3D translational motion. Luckily, nothing very fundamental changes as compared to the 1-DoF case. The most important difference is the use of vector notation for compactness and clarity of mathematics and code. For position ($\vec{r}$), velocity ($\vec{v}$), and acceleration ($\vec{a}$), we represent them as tri-dimensional column vectors:

$$\vec{r} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \vec{v} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{bmatrix}, \vec{a} = \begin{bmatrix} \ddot{x}_1 \\ \ddot{x}_2 \\ \ddot{x}_3 \end{bmatrix}$$

This is done not only for the sake of compactness, but also to facilitate the use of matrix operations for translation between reference frames. This concept is further explored in Section 2.2. For this reason, expressing these elements in vector form will become crucial as we move forward. In MATLAB, we express a tri-dimensional column vector as:

```
pos=[0;0;0];
```

Where each semicolon represents a new row of the vector. We define a row vector by using a comma instead of a semicolon:

```
pos=[0,0,0];
```

Often, it is useful to convert between a row and column vector in MATLAB because some functions will expect the input to be in a different format. We can do so with a transposition. In MATLAB, this looks like:

```
pos=[0,0,0]';
```

Expressed in vector notation, we write Newton's 2nd Law in the 3D case as $\vec{F} = m^i\ddot{\vec{r}}^{\,op}$. The presence of i indicates that this acceleration vector must be in the

inertial frame. The quantity ${}^{i}\ddot{\vec{r}}^{\,op}$ can be deduced from the use of the Basic Kinematic Equation (BKE) in simple cases.

2.3.2 Vector directions in the 3-DoF

We present below a modified version of the 1-DoF below in full 3D in Figure 2.5.

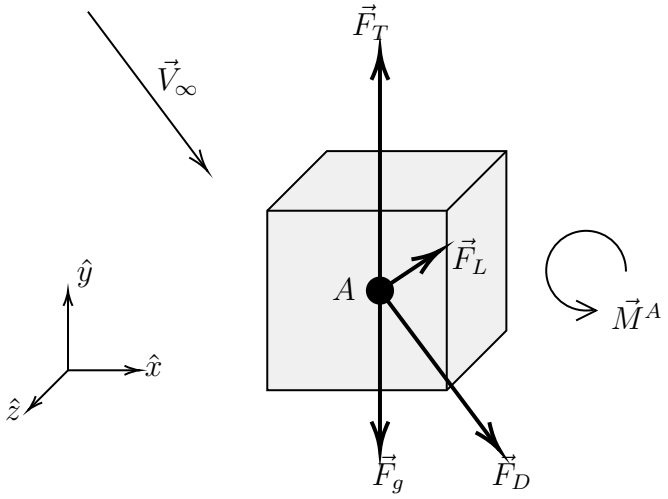


Figure 2.5: 3-DoF Forces on a Body

As compared to 1-DoF case, we have three new quantities that are present. The first of these is $\vec{V}_\infty$, the freestream airflow vector.[5] This is represented diagrammatically because it is no longer constrained along the $\hat{y}$ direction. When we have an angle between the nose (the direction through which the thrust force points in Figure 2.5) and the freestream velocity, we refer to this as an angle of attack. This angle of attack is often denoted with the letter α. We will discuss angle of attack and its effects more in Chapter 5. For now, we simply need to understand how to calculate α given the vector through the nose and the freestream velocity vector.

To do this, we use the fact that the dot product is related to the cosine of the angle between the vectors. Denoting the vector through the nose of the rocket $\hat{X}$, we can find the angle of attack as:

$$\alpha = cos^{-1}\left(\frac{\hat{X}\cdot\vec{V}_\infty}{||\hat{X}||\ ||\vec{V}_\infty||}\right) \tag{2.3}$$

The next new quantity is $\vec{F}_L$, the force of lift. The force of lift is always perpendicular in direction to $\vec{V}_\infty$. Knowing this, we can find the direction of the force of lift as:

$$\hat{L} = \frac{(\vec{V}_\infty \times \hat{X}) \times \vec{V}_\infty}{||(\vec{V}_\infty \times \hat{X}) \times \vec{V}_\infty||} \tag{2.4}$$

Here, the cross product is used because it generates a vector orthogonal to the two input vectors. This is exactly the property that we need when defining lift.

For the sake of completeness, we should also note that the force of drag lies opposite the direction of $\vec{V}_\infty$. This is much simpler to calculate, and looks like:

$$\hat{D} = \frac{-\vec{V}_\infty}{||\vec{V}_\infty||} \tag{2.5}$$

The last force directions to define are the direction of thrust and the direction of gravity. Luckily, these are easily defined because they point directly along basis vectors.[6] The force of gravity is defined with respect to the inertial reference frame and is defined in (2.6). The force of thrust is defined with respect to the vector pointing through the nose, $\hat{X}$, and is shown in (2.7):

$$\vec{F}_g = -mg \cdot \hat{y} \tag{2.6}$$

$$\vec{F}_T = F_T \cdot \hat{X} \tag{2.7}$$

2.3.3 Moments

The last new quantity is the moment about point A, denoted $\vec{M}^A$. For now, we will simply note that for an arbitrary selection of point A, we will have a rotational moment that is generated because the forces on the body do not necessarily act through point A, even if it is the center of mass (see section 4 for more details). In Figure 2.5, we have drawn all of the forces acting through a single point for simplicity, but this is not generally the case.

For our rocket modeling, it is generally assumed that gravity and thrust act about the center of gravity, and the aerodynamic forces about the center of pressure.

We will show how to compute this moment if the location of the center of gravity and the center of pressure is known. To show this, we will briefly return to the 2D

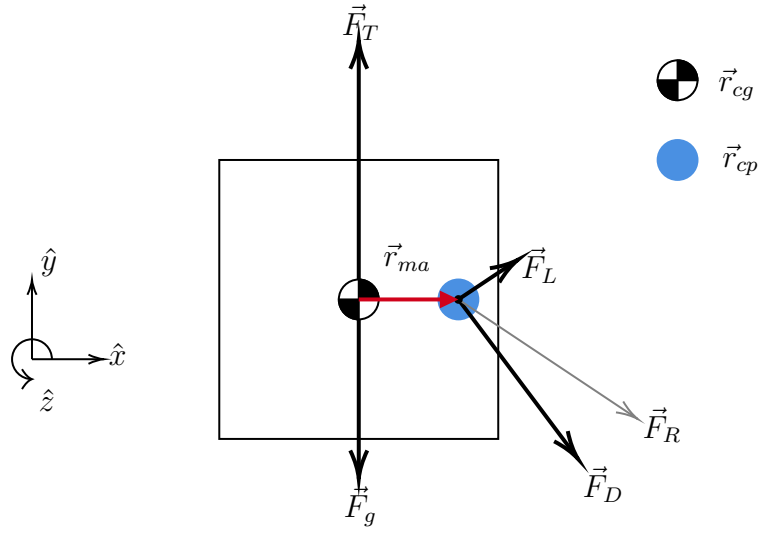


Figure 2.6: Moment Demonstration

case for illustration, but keep in mind that this concept is extensible in 3D. We show this in Figure 2.6.

In Figure 2.6 we include a few new symbols. All force magnitudes, however, are equivalent to what we show in Figure 2 (with the assumption that they lie in the x-y plane for this example). We denote the location of the center of gravity as $\vec{r}_{cg}$ and the location of the center of pressure as $\vec{r}_{cp}$.[7] The difference between $\vec{r}_{cg}$ and $\vec{r}_{cp}$ is denoted as $\vec{r}_{ma}$. We refer to this as the moment arm of the aerodynamic forces. It should be noted that all of locations are tridimensional vectors.

We also define a new vector, $\vec{F}_R$, the resultant aerodynamic vector, which is the vector sum of $\vec{F}_D$ and $\vec{F}_L$. Because free rotations occur about the center of mass, it is most helpful to calculate our final moment from this location. To do so, we use the moment equation:

$$\boxed{\vec{M}^{cg} = \vec{r}_{ma} \times \vec{F}_R}$$

We note that the direction of this moment is in the $-\hat{z}$ direction from the RHR. If we assume that our forces are planar, this moment will generate a rotation about a single axis. This makes our model into a 3-DoF.

One thing to note is that some quantities are more easily defined with respect to vectors that are defined with respect to the vehicle (which we will call the body

frame), such as the vector $\hat{X}$. As seen in Equation (2.3) and (2.4), we often want to use these vectors in the body frame for computation. As such, we will need a method in which we can convert between the body and inertial frames. This conversion between frames is explored in Section 2.2. We should also note that our moments will be in this body frame as well, which turns out to be especially useful in Section 2.4.

The last thing that we note in this section is that our forces are moments are coupled. This means that the aerodynamic forces are a function of the orientation of the vehicle, so the moments and forces must be solved as a system.

2.3.4 State Vector

As a last note for this section, we will briefly discuss the concept of a state vector. The state vector is a column vector that contains information on the current state of our system. In the example of our 3-DoF, we will include the position, velocity, rotation, and rotation rate of the system in the state vector. This would look like[8]:

$$\vec{X} = \begin{bmatrix} x \\ y \\ z \\ \theta \\ \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\theta} \end{bmatrix}$$

It is also useful to define the derivative of our state vector, which would just be the derivative of each of its elements. This is expressed as

$$\dot{\mathbf{X}} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\theta} \\ \ddot{x} \\ \ddot{y} \\ \ddot{z} \\ \ddot{\theta} \end{bmatrix}$$

For the sake of compactness, these vectors are generally not written in their full form. In this text, we will often refer to a state vector as just $\vec{X}$. Note that we also use

the vector $\hat{X}$ to denote a unit vector in the body frame, so the difference in the hat becomes an important distinction here.

2.4 Euler Rotation Equations

In subsection 2.3.3 we have explored the moments that are created when a force acts at a point that is not through the center of mass. Here, we will make the jump from particle dynamics into *rigid body* dynamics.

In analogy to Newton's 2nd Law of motion, there is an equally important relationship for rotational dynamics. This equation is known as Euler's 2nd Law:

$$\boxed{\vec{M}^o = \frac{{}^i d\vec{H}^o}{dt}} \tag{2.8}$$

Euler′s 2nd Law

This equation reads as "The moment about point O is the inertial time derivative of the angular momentum, H, about point O." The superscript i refers to the fact that this derivative must be taken in the inertial frame.

We also have the relationship between angular momentum, $\vec{H}$, and the *moment of inertia*, I, for an rigid body that you have likely seen in physics classes:

$${}^i\vec{H} = {}^i(I\vec{\omega}) \tag{2.9}$$

You will notice that our moment of inertia here is defined with respect to the inertial frame. As a result, when our body changes orientation, the moment of inertia may also change!

Often however, it is more helpful to define this moment of inertia in terms of our body frame. We will go through this derivation of the moment of inertia below.

2.4.1 Moment of Inertia and Inertia Tensor

The moment of inertia of a body is a measure of its resistance to rotational acceleration. It is in essence, the rotational analogue of mass (mass is a measure of resistance to linear acceleration, by the same line of reasoning).

Since we need the moment of inertia to describe our relationship for angular momentum in (2.9), it is useful for us to see its derivation here.

We can start by showing that the mass of a region can be found by integrating the density over its volume. Mathematically, we show this as:

$$\iiint_V \rho dV$$

Similarly, we know that the moment of inertia of a particle is mr^2, which is it's mass multiplied with the length of the moment arm, r, squared. This is often referred to as the 'second moment' of mass. By integrating this second moment, we have:

$$\iiint_V \rho r^2 dV$$

It is useful to define the moment of inertia along each direction, though. We can rewrite r in terms of x, y, and z for each of the component directions:

$$\begin{aligned} I_{xx} &= \iiint_V \rho\left(y^2 + z^2\right) dV \\ I_{yy} &= \iiint_V \rho\left(x^2 + z^2\right) dV \\ I_{zz} &= \iiint_V \rho\left(x^2 + y^2\right) dV \end{aligned} \tag{2.10}$$

For our rocket, we will find that these are the only quantities that we need to define. We also take define these quantities with respect to the center of mass since this is the point about which free rotations occur.

However, in general, we also have three more quantities, which are called the product of inertia. This is defined by the product of two of our elements, such as xy. This will have the same dimensions as the quantity r^2, so it can be applied in a similar way to the moment of inertia. The product of inertia is defined as follows:

$$\begin{aligned} I_{xy} = I_{yx} &= \iiint_V -xy\rho dV \\ I_{xz} = I_{zx} &= \iiint_V -xz\rho dV \\ I_{yz} = I_{zy} &= \iiint_V -yz\rho dV \end{aligned}$$

For our rocket, because of the symmetry about the $\hat{X}$ axis, these quantities go to zero. However, for future implementations that may include fuel slosh and asymmetries, the products of inertia may be non-zero! We express our *inertia tensor*, denoted I, as the 3×3 matrix which contains the moments and products of inertia.

$$I = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix}$$

For our rocket, we get the simplification that:

$$I = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix}$$

In fact, we can always find axes such that the products of inertia are zero. These are called the *principle axes* and are important in other applications, such as structures [2]. For our rocket, it just happens that these principle axes are the ones we have chosen to define as our body frame. The moment of inertia along these principle axes are solutions to the eigenvalue problem $I\vec{\omega} = I^*\vec{\omega}$, where I^* is a scalar. Essentially, we are finding a matrix I such that the output is just a scalar multiple of the input, ω. We can see that having this property will greatly simplify our results. This is another reason why we like to perform many calculations in the body frame.

Using the inertia tensor, a matrix form of (2.9) is written as:

$$\vec{H} = I\vec{\omega} \tag{2.11}$$

In the simplification using the principle axes, we often refer to the terms I_{xx}, I_{yy}, and I_{zz} simply as I_x, I_y, and I_z. There are called the principle moment of inertias (PMOIs).

2.4.2 Parallel Axis Theorem

Similar to the way in which we described the moment of inertia of a particle to be mr^2, we can describe the change in moment of inertia of a body due to the offset of its axis.

We accomplish this with the *parallel axis theorem*, which relates the moment and products of inertia through different rotation points. To use parallel axis theorem,

we simply take the known moment of inertia and add mr^2 along that component direction. This looks like:

$$I_{x'x'} = I_{xx} + m\left(y^2 + z^2\right)$$

This is similarly computed for the other axes, with the r always being the axis that is not the one we are computing the moment of inertia of. We can think of this intuitively because we are translating the center along the two axes perpendicular to the one describing the moment of inertia.

We also must do the same for the products of inertia. We note that even if we select body axes with no products of inertia, we may still have them in the translated center!

The parallel axis theorem for the products of inertia is given as:

$$I_{x'y'} = I_{xy} - mxy$$

This is similarly computed for the other axes.

2.4.3 Rotations in the Body Frame

Now that we have defined our inertia tensor in the body frame, it is useful to reexamine (2.8) and (2.9) to express them in terms of the body frame.

We can express equation (2.8) in terms of the body frame using the BKE. We write this as:

$$^i\frac{dH^o}{dt} = {}^b\frac{dH^o}{dt} + {}^e\vec{\omega}^b \times H^o \tag{2.12}$$

In the case of principle axes, (2.11) can be simplified to yield:

$$^iH^o = I_x\omega_x\hat{b}_1 + I_y\omega_y\hat{b}_2 + I_z\omega_z\hat{b}_3$$

Plugging this result into (2.12), we get the following:

$$^i\frac{dH^o}{dt} = {}^b\frac{d}{dt}\left(I_x\omega_x\hat{b}_1 + I_y\omega_y\hat{b}_2 + I_z\omega_z\hat{b}_3\right) + {}^e\vec{\omega}^b \times \left(I_x\omega_x\hat{b}_1 + I_y\omega_y\hat{b}_2 + I_z\omega_z\hat{b}_3\right)$$

Where ${}^e\vec{\omega}^b$ is $\omega_x\hat{b}_1 + \omega_y\hat{b}_2 + \omega_z\hat{b}_3$. The algebraic manipulation and vector math here is left as an exercise for the reader. Collecting the final terms, our final expression is:

$$^iH^o = \left[I_x\dot{\omega}_x + (I_z - I_y)\right]\hat{b}_1 + \left[I_y\dot{\omega}_y + (I_x - I_z)\right]\hat{b}_2 + \left[I_z\dot{\omega}_z + (I_y - I_x)\right]\hat{b}_3$$

Equating this with the moments in the body frame that were found in subsection 2.3.3 as ${}^{b}M^{o} = M_x\hat{b}_1 + M_y\hat{b}_2 + M_z\hat{b}_3$, we arrive at the final form of our equations for Euler rotations[9]:

$$\begin{aligned} M_x &= I_x\dot{\omega}_x + (I_z - I_y)\,\omega_y\omega_z \\ M_y &= I_y\dot{\omega}_y + (I_x - I_z)\,\omega_x\omega_z \\ M_z &= I_z\dot{\omega}_z + (I_y - I_x)\,\omega_y\omega_x \end{aligned}$$

Euler Rigid Body Equations

We can rearrange this to a form that is more useful for us (although somewhat less sightly), putting the unknown $\dot{\omega}$ (we also refer to this quantity as the angular acceleration, $\vec{\alpha}$) quantities alone:

$$\begin{aligned} \dot{\omega}_x &= \frac{M_x - (I_z - I_y)\,\omega_y\omega_z}{I_x} \\ \dot{\omega}_y &= \frac{M_y - (I_x - I_z)\,\omega_z\omega_x}{I_y} \\ \dot{\omega}_z &= \frac{M_z - (I_y - I_x)\,\omega_y\omega_x}{I_z} \end{aligned} \tag{2.13}$$

These sets of equations are especially powerful because they allows us to numerically integrate the angular velocity. Our moments can be found using Newtonian dynamics and the PMOIs are quantities that are easily found using the integrals described above in subsection 2.4.1. We will use these angular velocity derivatives extensively in Volume II to describe the time rate of change of the attitude of the rocket.

2.5 Examples

We've just laid out a large amount of material, so it is best to show a few examples to get the gist of things.

Example 2.1: Explorer 1 Satellite

NASA asks you to help analyze the rotational dynamics of the Explorer 1 satellite for your job interview. The satellite can be modeled approximately as a cylinder with a length of 2 m and a diameter of 16 cm. Assume the mass is 16kg with a uniform mass distribution.

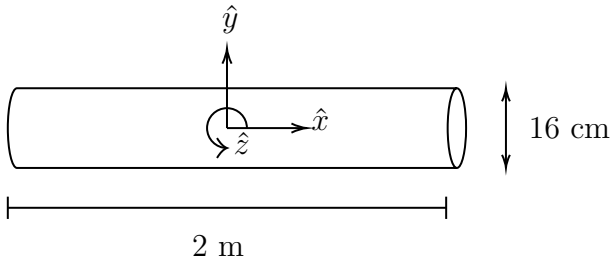


Figure 2.7: Explorer 1 Satellite

Moments of Inertia

To start analyzing the rotations, we'll first need to compute the moments of inertia, as these will define the 'rotational mass' of the satellite. Since the satellite is cylindrical, we have symmetry about the $\hat{x}$-direction. So, we will convert into polar coordinates, which gives the following relations:

$$\begin{aligned} x &= x \\ y &= r\cos\theta \\ z &= r\sin\theta \end{aligned}$$

We can use these relations in our moment of inertia integrals (2.10). We'll start by computing the moment in the x-direction, substituting in our polar coordinates:

$$I_{xx} = \iiint_V \rho\big(y^2+z^2\big)dV \rightarrow I_{xx} = \iiint_V \rho\Big[(r\cos\theta)^2+(r\sin\theta)^2\Big]dV$$

Next, we change the integral bounds. These integration bounds will be the same for all I_{xx}, I_{yy}, and I_{zz} since the bounded areas are identical. We use the volume of the cylinder as the integration bounds, and use the jacobian associated with polar coordinates for dV (which is $rdrd\theta dx$).

$$I_{xx} = \int_0^r \int_0^{2\pi} \int_{-\frac{h}{2}}^{\frac{h}{2}} \rho\Big[(r\cos\theta)^2+(r\sin\theta)^2\Big] r dr d\theta dx$$

With the integral set up, it is simply a matter of computation to complete it. Some trig relations may be useful in completing the integral. The only other thing to note is that we can rewrite ρ as $\frac{m}{V}$, where V for the cylinder is $\pi r^2 h$.

Doing these integrals for all of the axes, we get:

$$\begin{bmatrix} \frac{1}{2}mr^2 & 0 & 0 \\ 0 & \frac{1}{12}m(3r^2+h^2) & 0 \\ 0 & 0 & \frac{1}{12}m(3r^2+h^2) \end{bmatrix}$$

Which is valid for all uniform density cylinders.

Plugging in the given values for Explorer 1, we have:

$$\begin{bmatrix} 0.0512 & 0 & 0 \\ 0 & 5.3590 & 0 \\ 0 & 0 & 5.3590 \end{bmatrix} \text{ kg·m}^2$$

Spin Stabilization

NASA wants to spin-stabilize the satellite along the $\hat{x}$-direction with a final rotation rate of 2π rad/s. We will assume that the impulse is provided by two motors acting as a couple. The spin motors will also separate the satellite from the launch vehicle, so they are inclined 10° from horizontal to give the satellite a small forward momentum. So, we will calculate:

- The total impulse needed to bring the satellite to a final rotation rate of 2π rad/s
- The ΔV between the launch vehicle and the satellite after the burn, assuming the total impulse requirement is met.
- The average thrust if a 0.5 s burn is desired.

The first thing to do is calculate the total impulse needed, which can be found easily. The impulse is defined as begin the same as H using the impulse-momentum equality:

$$H = I_{xx}\Delta\omega_x = M_x\Delta t$$

Since Δt is unknown, we consider the left-hand side of the equality. Taking $\Delta\omega_x = \omega_f - \omega_i = 2\pi - 0$ and I_{xx} from before, we have a total impulse requirement of 0.3217 N·m·s.

Despin

After the spin stabilization maneuver is done to inject the satellite into its final orbit, NASA wants to slow the satellite down by using a yo-yo de-spin. Suppose the satellite has two weights, each of mass 0.5 kg, that are tied to massless strings on the satellite. Assuming that these weights can be approximated as point masses, find the necessary length of the strings so that the final rotation rate of the satellite is 5 rpm.

2.6 Key Ideas

1. Newton's 2$^{\text{nd}}$ Law can be used to create 2$^{\text{nd}}$ order differential equations that describe particle motion.
2. An n^{th} order differential equation can be reduced into n first order differential equations.
3. First order differential equations can be easily numerically integrated by a computer, allowing us to get approximate solutions to a huge number of problems.
4. We can describe the properties of a bodies rotation in an analogous way to Newton's 2$^{\text{nd}}$ Law.
5. We can create equations for the angular acceleration and angular velocity using Euler rotation equations.
6. We can use different frames of reference to more easily express quantities. The body frame and inertial frame are the most important in our context.

2.7 Further Reading

We assume that most readers are broadly familiar with the topics discussed in AAE 203. Understanding the application of the Basic Kinematic Equation (BKE) (otherwise known as derivative transport theorem) is fundamental to the understanding of dynamics. More information about basic dynamics in general can be found in [7]. For Newtonian dynamics, refer to Chapter 1 and 2. Euler rotation equations are

discussed in Chapter 4. This book uses different notation (notably, $\vec{r}^{\,op}$ is normally denoted as $\vec{r}_{p/o}$) but can help solidify this knowledge.

2.8 Practice Problems

1. On a nice Wednesday evening during one of your strolls through campus, you are kidnapped by a business major, who will only let you free if you can fix the vibration of the squeaky door at their frat house.

   You can consider the door hinge as a mass-spring system constrained to motion along the $\hat{x}$-direction. The spring has an equilibrium length of L_0 and a spring constant of $k = 5$ N/m with a mass of 1 kg. At time t_0, an instantaneous impulse of 10 N $\cdot$ s is applied to the hinge in the $-\hat{x}$-direction by a misguided ping-pong ball hitting the door. This is shown diagrammatically in Figure 2.8. Assume that the system is at rest in the equilibrium position at t_0.

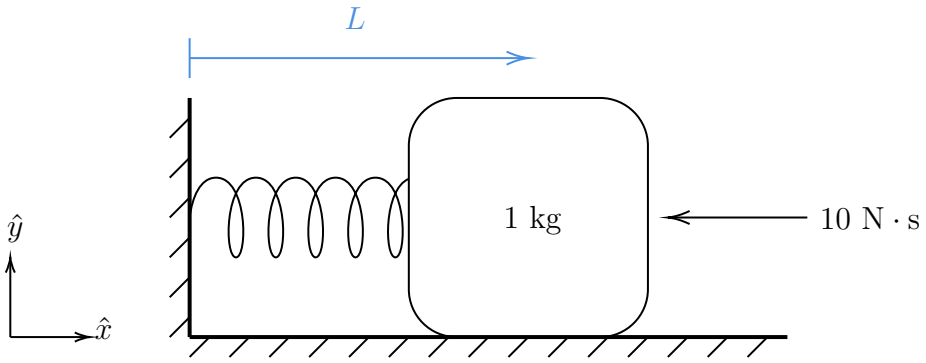


Figure 2.8: Mass-Spring System Problem Set-Up

   (a) Consider the system to be frictionless. Derive the EOMs for the system in both of the component directions.

   (b) Find the analytical solution of the system from the EOM you derived. Graph the solution.

   (c) Numerically integrate the EOMs ascertain the position and velocity as a function of time. Graph the solution. How does it compare to the analytical solution? You may want to refer to Chapter 6 for other numerical integration methods.

(d) Consider the system with some frictional force that is proportional to the velocity, with a proportionality constant of $c = 1\frac{\text{kg}}{\text{s}}$. This frictional force opposes the motion of the system. Repeat step c with this new system. How does the friction change the system?

2. After your escape from the frat house, you run into your friend, Buckminster, who is a big fan of circular-shaped objects. He asks you if you can model the orbit of a satellite to see if it looks anything like his *geodesic* domes. Not wanting to let him down, you write a MATLAB script to do so. You can assume the earth has a radius $R_\oplus$ =6380 km and $\mu_\oplus = 3.986 \cdot 10^5 \frac{\text{km}^3}{\text{s}^2}$.

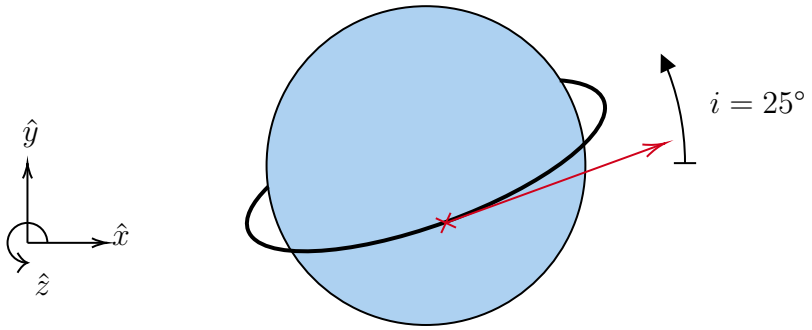


Figure 2.9: Satellite in Orbit Problem Set-Up

(a) Assuming 3-DoF, derive the EOM for the satellite motion assuming that gravity is the only force acting on the satellite and the earth is a point mass. *Hint: You may want to use a different coordinate system.*

(b) You find a satellite in orbit with a height of 400 km. You also measure the inclination of the orbit as it passes overhead as 25°. Assuming a circular orbit, what is the magnitude of the velocity and the component velocities in $\hat{x}$ and $\hat{y}$ as it passes overhead?

Notes

1. More formally, we define these vectors as existing a linear space, where multiplicative and additive properties are preserved. This formal definition allows us to apply this to any arbitrarily number of dimensions. For our uses, this will usually be up to 3 dimensions.

2. Sometimes systems like this can be solved with traditional ODE methods. The most easy identifier is if the system is linear or not. However, most complex systems like our 6-DoF have no known analytical solutions. We will focus on numerical integration because it will be more useful in general.

3. Note that we are not 'converting' in the sense of an equality, since the equations are no longer equivalent when we convert a differential element into a discrete one. We are just making an approximation of the true solution. Luckily, numerical integration schemes can get us quite close to true solutions.

4. An inertial frame can have some constant velocity and still be inertial. Of course, the earth has some velocity around the sun and with respect to the distant stars which we can approximate as constant for our time scales and thus ignore.

5. freestream velocity is drawn in the direction of incoming air by convention. However, for calculations, we will use the convention that $\vec{V}$ is the direction of the velocity of the vehicle (opposite sign to drawing). These statements are identical depending on the reference frame of the observer.

6. We assume here that there is no misalignment in the thrust vector and it is coincident with the vector. A more complex formula can be used when thrust misalignment is present, but this geometrical argument is not shown here.

7. The different notation of a subscript is used because this is not a standard position vector from one point to another, they rather denote the center of mass and center of pressure.

8. The order of the state vector does not particularly matter, so long as we are consistent with the location of the elements. Typically, it is common to write all vectors first and then their derivatives, but there is no hard rule. We follow this convention in our discussion.

9. Very observant readers may notice that this form of the equation is technically incorrect, since our center of rotation, O, will change throughout the flight as the center of mass changes position.

Moreover, the moments will change as fuel is drained, so another derivative should exist. Thus, we will have some small effects that are not accounted for in this equation. For our purposes here, these are ignored for simplicity of presentation. However, a more general approach that implements this may be used in the future if slosh modeling is implemented.

CHAPTER 3

Energy Methods of Mechanics

> *"Mathematics compares the most diverse phenomena and discovers the secret analogies that unite them."*
>
> – Joseph Fourier

Energy methods of mechanics employ the concepts of the energy of a system to solve a problem in mechanics. This is in contrast to the Newtonian method, which considers the forces of a system. The good news is that both will result in the same answer (see proof in subsection 8.1.2). However, having both in your set of tools as an engineer can prove especially useful.

To grasp energy methods, it is best to have an intuition for Newtonian dynamics first. A thorough reading of Chapter 2 and other sources will help prepare the reader for this section.

There are two main types of energy mechanics, referred to as Hamiltonian and *Lagrangian* mechanics. In this chapter, we will mostly focus on the Lagrangian version of these mechanics. Hamiltonian mechanics are arguably more fundamental, but will only be briefly discussed.

Although the Hamiltonian and Lagrangian are not directly used in the case of our 6-DoF, they are fundamental to other areas of mechanics and make derivations far simpler in many cases. The Hamiltonian and Lagrangian descriptions of mechanics are more often taught to physics students rather than engineers, but we feel that they deserve inclusion among the topics presented here. When we discuss numerical integration, we can use *symplectic integrator*s when our problem has a Hamiltonian description. This will be further discussed in Section 6.4. You may find that certain EOM derivations, such as the spinning top and problems in orbital mechanics are more easily accomplished with energy methods.

Another reason to motivate these systems is that these energy methods are rotationally invariant because energy methods can employ a generalized coordinate system. These generalized coordinates are very powerful and allow us to describe a huge variety of systems. We will see this later in some examples.

A brief overview of the differences between the two schemes of EOM derivation is shown in Figure 3.1.

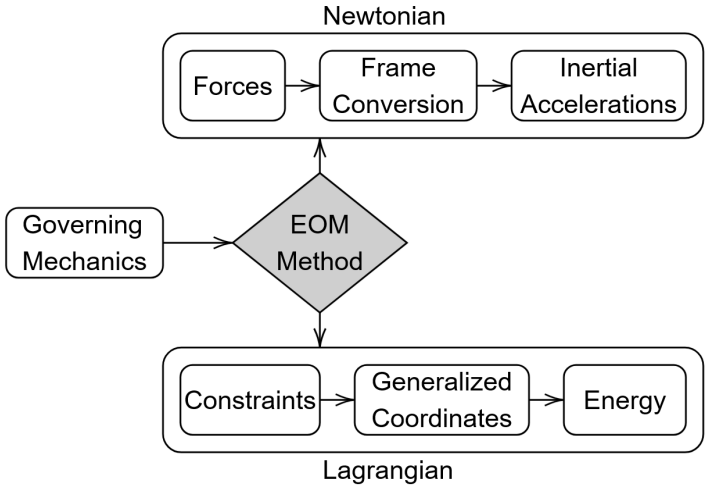


Figure 3.1: Newtonian vs. Lagrangian Mechanics Approach

3.1 Calculus of Variations

The calculus of variations underpins the ideas of Lagrangian and Hamiltonian mechanics, so it is important to understand the principles behind it before moving on to the actual mechanics. In analogy to how single variable calculus is fundamental to Newtonian mechanics, the calculus of variations is fundamental to energy mechanics.

The fundamental idea of the calculus of variations is simple. We want to find the maximums and minimums of some quantity in our problem. Why? In physics, we often see that quantities are minimized or maximized for some reason. For example, a soap surface will minimize its surface area, or a beam of photons will follow a *geodesic* of spacetime. Light will also refract through surfaces to minimize time as seen in Snell's Law.

Because so many physical phenomena are governed by this behavior to minimize certain quantities, Hamilton (the same mathematical hero we will see later with attitude dynamics), decided to formulate a mathematical description of this.

Before we get to Hamilton's formulation, we need to understand what the calculus of variations is trying to do. To find a minima means that naturally, we should look for the extrema of some quantity. We call this quantity a *functional*, which we define as:

$$J = \int_{x_1}^{x_2} f\left[y(x), y'(x); x\right] dx \tag{3.1}$$

The functional, J, is a quantity which maps a function onto the real number line. This means that it takes the function as an input and outputs a single real number. We are attempting to find an input function, $y(x)$, that yields an extrema of J.

To be precise about this definition of an extrema - we are attempting to find a function $y(x)$ such that any other function added to $y(x)$, no matter how close it is to $y(x)$, must change the value of J such that it is no longer an extrema. By convention, this other function added to $y(x)$ is called the *neighborhood function*, and is denoted $\eta(x)$.

We can compactly write this $y(x) + \eta(x)$ as $y(\varepsilon, x)$. We define the condition $y(0, x) \equiv y(x)$, where $y(x)$ is the function that gives the extrema of J. So, we can write $y(\varepsilon, x)$ as

$$y(\varepsilon, x) = y(0, x) + \varepsilon\eta(x) \tag{3.2}$$

The important restriction that we impose on $\eta(x)$ is that it must vanish at the endpoints, x_1 and x_2. This will prove important later so we can apply integration rules on our function. We also want a continuous first derivative so that $y'(\varepsilon, x)$ can be defined as

$$y'(\varepsilon, x) = \frac{dy(0, x)}{dx} + \varepsilon\frac{d\eta}{dx} \tag{3.3}$$

Using this definition of the neighborhood function, we can rewrite (3.1) as

$$J(\varepsilon) = \int_{x_1}^{x_2} f\left[y(\varepsilon, x), y'(\varepsilon, x); x\right] dx$$

So, with this redefined version of the function, we can finally find the extrema using the familiar calculus rule, setting the derivative equal to zero. We also evaluate at zero since this is where we defined the extrema to exist:

$$\left.\frac{\partial J(\varepsilon)}{\partial \varepsilon}\right|_{\varepsilon=0} = 0$$

Applying this to (3.1), we arrive at

$$\frac{\partial J(\varepsilon)}{\partial \varepsilon} = \frac{\partial}{\partial \varepsilon} \int_{x_1}^{x_2} f\left[y(0,x), y'(0,x); x\right] dx$$

The ε terms drop out from the RHS from the evaluation of $\varepsilon = 0$. Applying the Leibniz Integral Rule, the partial derivative can be moved inside the RHS integral:

$$\frac{\partial J(\varepsilon)}{\partial \varepsilon} = \int_{x_1}^{x_2} \frac{\partial}{\partial \varepsilon} f\left[y(x), y'(x); x\right] dx$$

Applying the rules of implicit integration on the integrand, we get:

$$\frac{\partial J(\varepsilon)}{\partial \varepsilon} = \int_{x_1}^{x_2} f\left(\frac{\partial f}{\partial y}\frac{\partial y}{\partial \varepsilon} + \frac{\partial f}{\partial y'}\frac{\partial y'}{\partial \varepsilon}\right) dx \tag{3.4}$$

Note that we still must take the partial derivative with respect to ε on the RHS, even though we have plugged in 0 for ε.

Next, we can find expressions for $\frac{\partial y}{\partial \varepsilon}$ and $\frac{\partial y'}{\partial \varepsilon}$. This is done with the expressions (3.2) and (3.3). Starting with the first, we differentiate (3.2) with respect to ε:

$$\frac{\partial}{\partial \varepsilon} y(\varepsilon, x) = \frac{\partial}{\partial \varepsilon}\left[y(0,x) + \varepsilon\eta(x)\right] \tag{3.5}$$

Here, $y(0,x)$ is independent of ε since we define it to occur at value $\varepsilon = 0$. So this leaves us with

$$\frac{\partial y}{\partial \varepsilon} = \eta(x)$$

Doing the same for (3.5), we arrive at

$$\frac{\partial y'}{\partial \varepsilon} = \frac{\partial \eta}{\partial x}$$

Finally, plugging these back into the original expression for (3.4), we get the expression

$$\frac{\partial J(\varepsilon)}{\partial \varepsilon} = \int_{x_1}^{x_2} \left(\frac{\partial f}{\partial y}\eta(x) + \frac{\partial f}{\partial y'}\frac{\partial \eta}{\partial x}\right) dx$$

Finally, performing some integration by parts, we arrive at the final integral expression of the maximum:

$$\frac{\partial J(\varepsilon)}{\partial \varepsilon} = \int_{x_1}^{x_2} \left(\frac{\partial f}{\partial y} - \frac{d}{dx}\frac{\partial f}{\partial y'}\right) \eta(x) dx \tag{3.6}$$

Since we are evaluating the maximum, $\frac{\partial J(\varepsilon)}{\partial \varepsilon}$ evaluates to 0. Since $\eta(x)$ is an arbitrary function, the only way to guarantee an equality is to ensure that the part of the integrand in parenthesis is 0.

This gives us the fundamental equation of the calculus of variations, called *Euler's equation*:

$$\frac{\partial f}{\partial y} - \frac{d}{dx}\frac{\partial f}{\partial y'} = 0 \tag{3.7}$$

3.1.1 Delta Notation

In later sections, it is useful to have a shorthand for these partial derivative quantities. For a quantity χ, the variation, δ is given as:

$$\delta \equiv \frac{\partial \chi}{\partial \varepsilon} d\varepsilon$$

Using this notation, we can rewrite (3.6) after some algebraic manipulation as:

$$\delta J = \int_{x_1}^{x_2} \left(\frac{\partial f}{\partial y} - \frac{d}{dx}\frac{\partial f}{\partial y'} \right) \delta y dx$$

And thus, (3.7) can be written concisely as:

$$\boxed{\delta J = 0}$$

This is a very general and far reaching conclusion that can be applied to many systems. For our use cases in dynamics, we have more useful forms of this equation which will be explored in the following section.

3.2 Hamilton's Principle

Following directly from the ideas of the calculus of variations, Hamilton (alongside many other mathematicians) wanted to apply this to physical systems. Hamilton's Principle is as follows:

Theorem 1 (Hamilton's Principle)**.** *A system which may travel from one point to another within a specific time interval, the path that is followed is the path which minimizes the time integral of the difference between the kinetic and potential energies.*

$$\delta \int_{t_1}^{t_2} T - U dt = 0$$

The quantity $T - U$, the kinetic energy minus the potential energy, is a quantity called the *Lagrangian*, denoted L.

$$L \equiv T - U$$

The time integral of the Lagrangian is the *action*, denoted S. This is often why Hamilton's Principle is also called the *principle of least action*.

$$S = \int_{t_1}^{t_2} L dt$$

So, our action takes the form of the functional from the calculus of variations. From Hamilton's principle, this functional, the action, must be minimized. So, we often see the compact form of Hamilton's Principle as:

$$\delta S = 0$$

Expressed using Euler's differential equation, (3.7), we get the *Euler-Lagrange Equations*:

$$\boxed{\frac{\partial L}{\partial x_i} - \frac{d}{dt}\frac{\partial L}{\partial \dot{x}_i} = 0} \tag{3.8}$$

Euler – Lagrange Equations

Before we look into some examples of how to utilize these equations in practice, we should consider in what situations these will be better than the Newtonian methods.

We note that in this chapter, we will only consider potential functions which are *conservative*, or path independent. This does mean that frictional forces and aerodynamics are not modeled with the Lagrangian we describe here (although they can be with the utilization of Rayleigh's dissipation function and other techniques, such as those discussed in [13]). This is the reason we do not directly use these equations in the 6-DoF.[10]

So, for systems that are non-conservative, it is best to stick with the Newtonian method of mechanics, as we do for our 6-DoF. However, for those that are frictionless, the methods described here will work well. While it may seem that frictionless systems are very idealistic, we do see systems that can be approximated as frictionless quite often! Orbits, of course, are one case where this is especially true. For many other systems, such as a spinning top or other rotating bodies, we can often ignore the small

amount of friction and arrive at analytical solutions without much hassle using the Lagrangian. These methods also become extraordinarily important for our derivations of rotational equations.

With that noted, let us use the Lagrangian in a few examples to see how it can turn some complex Newtonian systems into fairly trivial problems.

3.2.1 Mass-Spring System

As a very basic first system, we will consider a mass spring system as shown in Figure 3.2. We consider the length x to be referenced from the rest length of the spring.

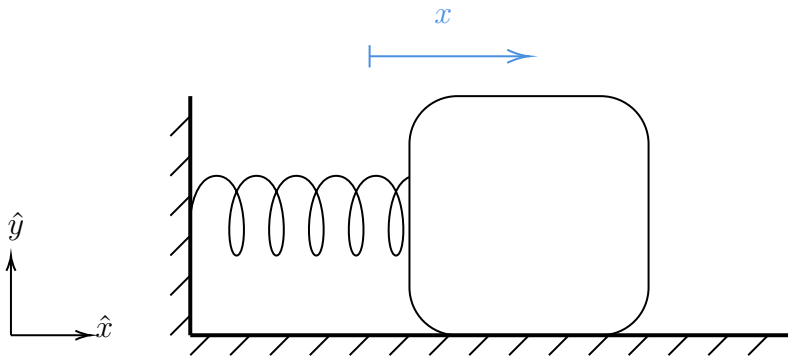


Figure 3.2: Mass-Spring System for Lagrangian

We know the analytical form of the EOM from Newtonian Dynamics:

$$m\ddot{x} + kx = 0$$

So, in essence, we are using the Lagrangian *a posteriori* to match this solution.

First, we will find the Lagrangian of the system. The kinetic energy of the particle is given by the linear translation only (there is no rotational motion for a point particle), so we have $T = \frac{1}{2}m\dot{x}^2$. The potential energy is the energy of the spring, given as $U = \frac{1}{2}kx^2$. So, the Lagrangian is:

$$L = \frac{1}{2}m\dot{x}^2 - \frac{1}{2}kx^2$$

Now, applying (3.8) in the $\hat{x}$-direction, as this is the only degree of freedom:

$$\frac{\partial L}{\partial x} - \frac{d}{dt}\frac{\partial L}{\partial \dot{x}} = 0$$

Taking each term individually,

$$\frac{\partial L}{\partial x} = -kx$$
$$\frac{\partial L}{\partial \dot{x}} = m\dot{x}$$
$$\frac{d}{dt}(m\dot{x}) = m\ddot{x}$$

So, the (3.8) equation gives us:

$$-kx - m\ddot{x} = 0$$

A simple change of signs gives the exact same result as the Newtonian approach:

$$m\ddot{x} + kx = 0$$

In this problem, we used a simple case where the dynamics were constrained along a single axis of motion. Before we move to a move complex problem, we will introduce the method by which Lagrangian mechanics deals with these reference frames.

3.3 Generalized Coordinates

We recall that in Newtonian mechanics, we describe reference frames for an object. We represent the position, velocity, and acceleration of a particle in terms of these coordinate systems as a tridimensional vector. Generally, we may write the $\vec{r}^{\,op}$ vector in a frame a as:

$$x\hat{a}_1 + y\hat{a}_2 + z\hat{a}_3$$

Because any tridimensional set of coordinates that we choose that are linearly independent of each other will span $\mathbb{R}^3$, we can generalize this to any reference frame, q, where the terms of a can be described by the terms of q:

$$x(q_1, q_2, q_3) + y(q_1, q_2, q_3) + z(q_1, q_2, q_3)$$

As a shorthand, we will write this full set of q, $(q_1, q_2, \cdots q_n)$, as q_j.

We can also generalize this to the velocity and the acceleration. The components of velocity can be written in terms of $q_j, \dot{q}_j$ and those for acceleration in terms of $q_j, \dot{q}_j, \ddot{q}_j$. When we do Newtonian dynamics, we need to use the BKE to transform these quantities if they are not in the inertial frame.

Note that this is what this looks like for a single particle. When multiple particles are involved, we may need to increase this number of q coordinates. As a very general statement, we can say for N particles, we will need n generalized coordinates in q. Expressed vectorially, given a particle of the i^{th} index:

$$\vec{r}^{\,oi}(q_j)$$

In other words, the position of an arbitrary particle is given in terms of n generalized coordinates.

The acceleration of the particles, when Newton's Laws are applied, are a function of the q_j as well as their derivatives, $\dot{q}_j$. So, the evolution of the system exists in a $2n$-dimensional state space. However, we can use our previously mentioned Lagrangian to circumvent the need for Newton's Law and the BKE in our derivation of the EOM.

3.3.1 Constraints

Constraint equations are fundamental to Lagrangian mechanics. In the previous section, we very vaguely stated the need for n generalized coordinates. To find a precise number, we turn our attention to constraints.

For each particle, we need 3 values to define the position, so we need $3N$ generalized coordinates.

However, we often have constraints that reduce this number. Often, we see a problem that is planar or where the particles are connected in a rigid body. These constraints will reduce the number of generalized coordinates needed. Here, we are only considering constraints on the positions of particles. Constraints on the positions of particles are known as *holonomic constraints*.

A holonomic constraint relates generalized coordinates to each other. In the commonly used notation, the holonomic constraint is expressed in terms of the q_j's as:

$$\Phi_k\left(q_j, t\right) = 0,\ k = 1 \cdots K$$

or in terms of the position vectors:

$$f_k\left(\vec{r}^{\,op_1}, \vec{r}^{\,op_2} \cdots \vec{r}^{\,op_N}, t\right) = 0$$

The number of holonomic constraints, K, will change the number of generalized coordinates as:

$$\boxed{\mathcal{M} = 3N - K} \tag{3.9}$$

The quantity $\mathcal{M}$ is the number of DoF. However, this is the number of DoF for the whole system, which is why it can exceed 6 in some cases!

Example 3.1: Particle on a path

Suppose we have a single particle that is constrained to motion along a path in the xy-plane. This path is given by $y = g(x)$.

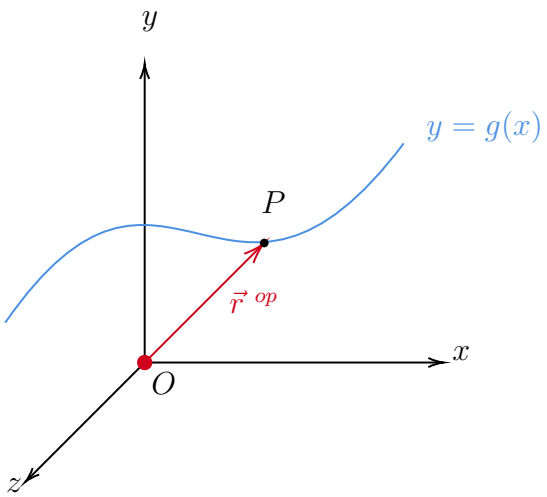


Figure 3.3: Particle Moving on Constrained Path

We can select our generalized coordinates to be the same as the cartesian coordinates, so that:

$$q_1 = x$$
$$q_2 = y$$
$$q_3 = z$$

So, to write these as holonomic constraints, we can write:

$$q_3 = 0$$
$$q_2 - g(x) = 0$$

With these two constraints, we have reduced the number of degrees of freedom, $\mathcal{M}$, to 1.

This makes sense intuitively, as the particle can only move along the path specified by the equation $g(x)$, therefore only having one degree of freedom.

The power of using generalized coordinates is that we could define a coordinate that more easily defines this system. For example, our one coordinate could be the arc length along the path, s defined from some starting point.

In general, these generalized coordinates do not even need to have units of length! We can select any parameters that are most convenient for defining the state, even those that are energies, so long as we can define a continuous derivative of that state and use the Euler-Lagrange Equation.

Rigid Body Constraints

For most of the systems that we will deal with, including our rocket, we make the assumption that particles are rigidly attached to each other, meaning that there is no change in distance between particles. Using the notation of constraints described above, we will show how this is done for a two particle system, which can then naturally be extended with more particles.

Given two particles that are connected together by a massless rigid rod of length L, we have a constraint on the particles distance:

$$\|\vec{r}^{\,op_1} - \vec{r}^{\,op_2}\| = L$$

Written in the more standard notation:

$$f\left(\vec{r}^{\,op_1}, \vec{r}^{\,op_2}\right) = \|\vec{r}^{\,op_1} - \vec{r}^{\,op_2}\| - L = 0$$

So this constraint takes the 6-DoF for the two particles and reduces it to 5-DoF.

An example of these constraints would be the position of the first particle in Cartesian coordinates (giving 3 of the generalized coordinates, x, y, z, and the azimuthal and elevation angle, ϕ and θ. The roll angle does not change the system in any measurable way, hence we only need 5 parameters to define a two particle system.

In the general case of N particles composing a rigid body, we have $3N$-DoF in a completely unconstrained system. However, we must constrain each pair of particles. An example of this for the 3 particle system in shown in Figure 3.4.

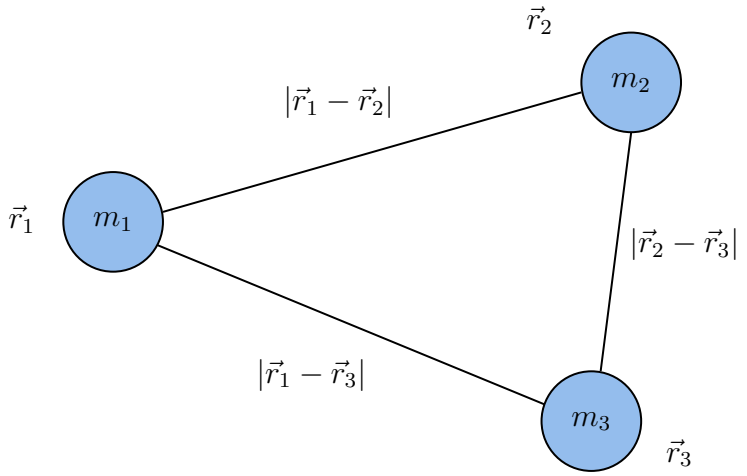


Figure 3.4: 3 Particle Rigid Body

So, we have 3 constraint equations in a system with 9 particles. This gives a total of $3N - 3 = 6$-DoF. When any additional particles are added, they must be fully constrained in all 3 translational degrees of freedom by constraints between r_1,r_2, and r_3. So, we always have constraints of the form $3N - 3(N - 2)$ for a rigid body. This is why we always see a rigid body displaying 6-DoF. We can then, of course, parametrize this in general coordinates with the cartesian coordinates and 3 angles.

Minimal Set of Constraints

So far, we have only discussed holonomic constraints, those that are related to position or on the generalized parameters, and possibly have some relation in time. When a system is a *holonomic system*, that is, purely defined by holonomic constraints, it is always possible to find a set of $\mathcal{M}$ generalized parameters to describe the particle motion. In other words, the minimum number of parameters is equivalent to the number of degrees of freedom in the system.

That said, we often choose to use more parameters, particularly in the case of rotations. As we will see in Volume II, we often find it more useful to use a non-minimal set of constraints to encode rotations for reasons to avoid singularities in the solution.

It is also common to see a non-minimal set of constraints used when the minimal set is cumbersome and does not give an intuitive feel to the problem.

Non-Holonomic Constraints

The other class of constraints are those constraints that are not merely a function of the generalized coordinates. These constraints, called *non-holonomic constraint*s, define the set of all constraints that depend on more than just the position of particles. This is a huge class of constraints so we will not cover every type. These are also less important to our derivations so we will only briefly discuss them.

The most common among these, and where we will focus our attention here, is constraints on the time derivative of the generalized coordinates. These are called *generalized velocities*, denoted $\dot{q}_j$. Such an example of a constraint on generalized velocities is a roller skate. A roller stake only allows a velocity in the direction of the skate and restricts velocity along the direction normal to the roller stake.

One important thing about non-holonomic constraints is that they do not reduce the number of parameters needed to define the system.

3.3.2 Lagrange's Equation in Generalized Coordinates

The good thing about the Lagrangian method is that it is invariant to changes in our coordinate system. Because the Lagrangian is an equation of energy, and energy is independent of coordinate transformations, we can easily express the Euler-Lagrange Equation in generalized coordinates with little hassle.

Using the generalized coordinates, the kinetic energy is a function of the generalized positions, velocities, and time. The potential is a function of the generalized position and time, so the Lagrangian becomes:

$$L \equiv T - U = T(q_j, \dot{q}_j, t) - U(q_j, t)$$

Stated another way, the Lagrangian is a function of the generalized positions, velocities, and time, as $L = L(q_j, \dot{q}_j, t)$. The Euler-Lagrange Equation then becomes:

$$\boxed{\frac{\partial L}{\partial q_j} - \frac{d}{dt}\frac{\partial L}{\partial \dot{q}_j} = 0, \quad j = 1, 2, \cdots M} \tag{3.10}$$

Generalized Euler – Lagrange Equations

Next we will show a problem where the rotational invariance and generalized coordinates proves especially useful in simplifying our derivation:

3.3.3 Pendulum EOM Example

Problem Set-Up

Suppose we have a pendulum of length L. The mass is rigidly attached to the pendulum. A reference frame a that moves with the pendulum is shown in Figure 3.5. The inertial frame is also shown. The pendulum moves in only the xy plane.

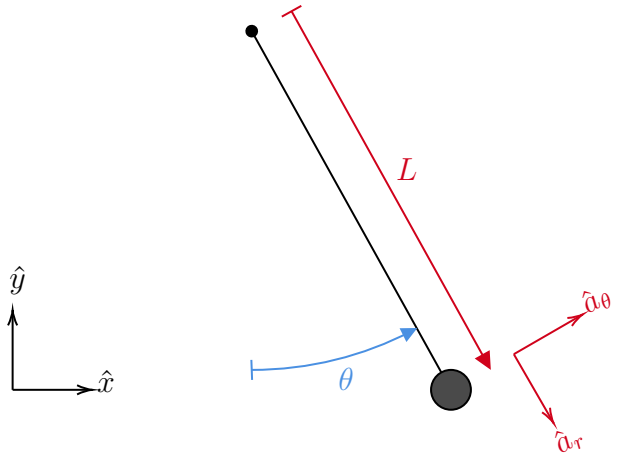


Figure 3.5: Pendulum Set-Up

Newtonian Approach

You have likely seen the Newtonian approach many times, but we show it here for completeness and to contrast the approach with the Lagrangian more directly.

In the Newtonian Approach to deriving EOM's, we always start with the forces governing the system. In this case, we have two forces. The gravitational force, mg, acts in the $-\hat{y}$-direction and the tension force acts in the $-\hat{a}_r$-direction. From the constraint on length, the acceleration in the $\hat{a}_r$ must be zero. Thus, the component of gravity in the $\hat{a}_r$ direction must equal the tension.

We express the gravity in the a-frame to find this:

$$\hat{y} = -\cos\theta\hat{a}_r + \sin\theta\hat{a}_\theta$$

So, by balancing the forces in the $\hat{a}_r$-direction:

$$m\ddot{a}_r = mg\cos\theta\hat{a}_r - \vec{F}_T$$

In the $\hat{a}_\theta$-direction, we have unconstrained motion, so $\sum\vec{F} = m\vec{a}$. Our force equation is thus:

$$F = -mg\sin\theta\hat{a}_\theta$$

Next, we need to find our acceleration in the inertial frame using the BKE to solve the RHS of the force balance. Starting with the position vector:

$$\vec{r}^{\,op} = L\hat{a}_r$$

We can apply the BKE to find our inertial velocity:

$${}^i\vec{v}^p = {}^a\frac{d}{dt}\vec{r}^{\,op} + {}^i\omega^a \times \vec{r}^{\,op}$$

Plugging in appropriate quantities:

$${}^i\vec{v}^p = {}^a\frac{d}{dt}(L\hat{a}_r) + \dot{\theta}\hat{a}_3 \times (L\hat{a}_r)$$

$${}^i\vec{v}^p = \dot{\theta}L\hat{a}_\theta$$

We do this again to find the inertial acceleration:

$${}^i\vec{a}^p = {}^a\frac{d}{dt}{}^i\vec{v}^p + {}^i\omega^a \times {}^i\vec{v}^p$$

$${}^i\vec{a}^p = {}^a\frac{d}{dt}\dot{\theta}L\hat{a}_\theta + \dot{\theta}\hat{a}_3 \times \dot{\theta}L\hat{a}_\theta$$

$${}^i\vec{a}^p = \ddot{\theta}L\hat{a}_\theta - \dot{\theta}^2L\hat{a}_r$$

Now, we equate the forces in each direction:

$$-mg\sin\theta\hat{a}_\theta = \ddot{\theta}L\hat{a}_\theta$$

Rearranging, we arrive at our EOM:

$$\ddot{\theta} + \frac{g}{L}\sin\theta = 0$$

Lagrangian Approach

In the Lagrangian approach, we will start by defining our constraints and coordinates. Some preliminary analysis shows that using polar coordinates for this problem will be useful. If we do not recognize this, we can always convert later (a distinct advantage of the Lagrangian approach, since it is coordinate frame invariant). Here we will choose a cylindrical coordinate systsem:

$$q_1 = r$$
$$q_2 = \theta$$
$$q_3 = z$$

Next, we define constraints. Since there is no motion in the z-axis:

$$z = 0$$

And the constraint on the length gives us:

$$q_1 - L = 0$$

So, we have 1 DoF from (3.9). Since both z and r are constrained, we can express our EOM in terms of one generalized coordinate, θ.

Next, we define the Lagrangian. Knowing that the linear velocity of the particle is $\dot{\theta}L$, we get:

$$T = \frac{1}{2}m\dot{\theta}^2L^2$$

For the potential, we choose the origin as our reference, giving us:

$$U = -mgy$$

And a coordinate transformation into polar using $y = r\cos\theta$ yields:

$$U = -mgL\cos\theta$$

This gives the Lagrangian:

$$L = \frac{1}{2}m\dot{\theta}^2L^2 + mgL\cos\theta$$

Applying the Euler-Lagrange formula for generalized coordinates, (3.10) on the single generalized coordinate:

$$\frac{\partial L}{\partial \theta} = -mgL\sin\theta$$
$$\frac{\partial L}{\partial \dot{\theta}} = mL^2\dot{\theta}$$
$$\frac{d}{dt}\left(mL^2\dot{\theta}\right) = mL^2\ddot{\theta}$$

So, the final system is:

$$-mgL\cos\theta - mL^2\ddot{\theta} = 0$$

With some rearrangement:

$$\ddot{\theta} + \frac{g}{L}\sin\theta = 0$$

We notice that with the Lagrangian approach, we did not need to have an understanding of the vector directions, frame translations, or BKE to find our EOM. This, in addition to the flexibility of the generalized coordinates, is the real power of this approach.

3.4 Hamiltonian Mechanics

After the formulation of Lagrangian Mechanics, another formulation was created by Hamilton later on. The Hamiltonian formulation, like the Lagrangian, does not introduce any new concepts, because it is simply an alternate approach to dynamics. However, like the Lagrangian, it makes some types of problems more simple. We present the Hamiltonian here briefly for its use case in symplectic integrators.

3.4.1 Derivation of the Hamiltonian

The exact derivation of the Hamiltonian using Legendre transforms is beyond the scope of our present discussion, so we will mostly discuss the consequences of this derivation.

The Hamiltonian method replaces the generalized velocities with *generalized momenta*. The generalized momenta are a function of $q, \dot{q}$, and t. We express the

generalized momenta as p. p is defined to be:

$$p(q, \dot{q}, t) \equiv \frac{\partial L}{\partial \dot{q}_i}$$

Thus, we express our Hamiltonian, $\mathcal{H}$, in terms of the generalized positions, generalized momenta, and time. This looks like:

$$\mathcal{H} = \mathcal{H}(q_j, p_j, t)$$

In a conservative system, we can also write:

$$\mathcal{H} = T + U$$

Using relations with the Lagrangian, we can write two first order differential equations for the EOM with the Hamiltonian.

$$\dot{q}_k = \frac{\partial \mathcal{H}}{\partial p_k}$$

$$-\dot{p}_k = \frac{\partial \mathcal{H}}{\partial q_k}$$

Hamiltonian EOM's

This is all we will note about Hamiltonian Mechanics for now. We will note here that these are more useful in the case of numerical integration because these are first order differential equations, as opposed to the second order differential equations of Lagrangian Mechanics.

3.5 Lagrangian Derivation of Rigid Body Dynamics

In Section 2.4, we explored the Newtonian method of deriving the Euler rotation equations using the BKE. Here, we present an alternative derivation that utilizes the Lagrangian approach and is ultimately a more fundemental and simple derivation.

We showed earlier in Figure 3.3.1 that we can describe a rigid body system with 6 generalized parameters, three of which are angles.

3.5.1 Torque Free Equations

In the case of no applied torque, the derivations of Euler's equations become surprisingly simple. We can assume that the change in height of the rigid body is irrelevant for pure rotation, so the Lagrangian is simply the kinetic energy of the object. We can suppose that we are also in a coordinate system where the translational kinetic energy is 0. This gives:

$$L = \frac{1}{2} I \omega^2 \tag{3.11}$$

We can select our generalized coordinates as the three Euler angles, ψ, θ, and ϕ (see Section 4.1).

Selecting the generalized coordinate, ψ, as an example, the Euler-Lagrange equation gives:

$$\frac{\partial L}{\partial \psi} - \frac{d}{dt}\frac{\partial L}{\partial \dot{\psi}} = 0$$

Using the derivation of the components of angular velocity using Euler Angles, we have:

$$\begin{aligned} \omega_1 &= \dot{\phi} \sin\theta \sin\psi + \dot{\theta} \cos\psi \\ \omega_2 &= \dot{\phi} \sin\theta \cos\psi - \dot{\theta} \sin\psi \\ \omega_3 &= \dot{\phi} \cos\theta + \dot{\psi} \end{aligned}$$

So, we can also express our Lagrangian in terms of the angular acceleration:

$$\frac{\partial L}{\partial \omega_i}\frac{\partial \omega_i}{\partial \psi} - \frac{d}{dt}\frac{\partial L}{\partial \omega_i}\frac{\partial \omega_i}{\partial \dot{\psi}} = 0$$

At this point, the derivation is simply a matter of 'plug and chug'. Calculating the derivatives with respect to ψ:

$$\begin{aligned} \frac{\partial \omega_1}{\partial \psi} &= -\dot{\phi} \sin\theta \cos\psi - \dot{\theta} \sin\psi \\ \frac{\partial \omega_2}{\partial \psi} &= -\dot{\phi} \sin\theta \sin\psi - \dot{\theta} \cos\psi \\ \frac{\partial \omega_3}{\partial \psi} &= 0 \end{aligned}$$

We notice here that $\frac{\partial \omega_1}{\partial \psi} = \omega_2$ and $\frac{\partial \omega_2}{\partial \psi} = -\omega_1$. Next, taking our derivatives with

respect to $\dot{\psi}$:

$$\frac{\partial \omega_1}{\partial \dot{\psi}} = 0$$
$$\frac{\partial \omega_2}{\partial \dot{\psi}} = 0$$
$$\frac{\partial \omega_3}{\partial \dot{\psi}} = 1$$

Last, we can make a substitution for the $\frac{\partial L}{\partial \omega_i}$ term by taking a derivative of the kinetic energy(3.11):

$$\frac{\partial L}{\partial \omega_i} = I_i \omega_i$$

Plugging everything into the Euler-Lagrange formula, we have:

$$I_1 \omega_1 \omega_2 - I_2 \omega_1 \omega_2 - \frac{d}{dt} I_3 \omega_3 = 0$$

Some simplification yields:

$$(I_1 - I_2)\, \omega_1 \omega_2 = I_3 \dot{\omega_3}$$

We can perform the analogous analysis for the other axes, yielding the full set of torque free rigid body equations:

$$\begin{aligned} (I_y - I_z)\, \omega_y \omega_z &= I_x \dot{\omega_x} \\ (I_z - I_x)\, \omega_x \omega_z &= I_y \dot{\omega_y} \\ (I_x - I_y)\, \omega_x \omega_y &= I_z \dot{\omega_z} \end{aligned}$$

Torque – Free Rotations

3.5.2 Forced Euler Rigid Body Dynamics

3.6 Key Ideas

1. We can find the EOM of a system using either the Newtonian approach or the Lagrangian approach, and it is often easier to find an EOM using the Lagrangian for a conservative system.

2. The Lagrangian is scalar, so it is invariant to coordinate transforms. This means we can choose any variables we want for our generalized coordinates when performing derivations.

3. The methodology of Newtonian mechanics and Lagrangian mechanics are fundementally different. Newtonian mechanics deals with the influence of external actions (forces), while Lagrangian Mechanics emphasizes the internal state of the system.

4.

3.7 Further Reading

The following section borrows heavily from the derivations of Chapter 6 and 7 of [19] and Chapter 5 of [21]. A good video overview of this topic can be found at [22].

There are an abundance of sources for classical mechanics. We can personally recommend the video series at [16], which covers this material in a manner more suitable to engineering contexts.

3.8 Practice Problems

1. Arthur keeps finding bugs on the apples that he is feeding his horse. Arthur heard that you have been learning about generalized coordinates. He wants a way to describe the position of the bugs on his apples using a minimal set of coordinates.

   Find a suitable set of generalized coordinates to describe this system. Write out any equations of constraints of the system. *Hint: A apple is topologically similar to a sphere!*

2. You are a prominent mathematician in the year 1696. Bernoulli proposes a problem to you to find a curve from a point (x_0, y_0) to another point (x_f, y_f). When a particle is placed on this path, it should be a faster path than any other path the particle could take down the slope under the influence of only gravity.

(a) Find the velocity as a function of the distance from the initial height, h, for the particle using energy conservation.

(b) Write a general expression for the amount of time it takes a particle to descent a path using an integral. *Hint: you can write* $t = \frac{ds}{v}$.

(c) Use the Euler-Lagrange equation to solve for the form of this curve (You may find the integral form useful!). Plot the result.

3. Théoden, king of the Rohirrim, is riding his armies north through the Emyn Muil. His horse, Snowmane, trips on a small rock and Théoden begins to lose his balance.

   You can model Théoden as a mass on the end of a massless rod and his horse as a cart constrained to motion along the $\hat{x}$-axis. The angle of Théoden from the vertical is denoted as θ. Derive the EOM of the system using energy methods. The set-up is shown in Figure 3.6.

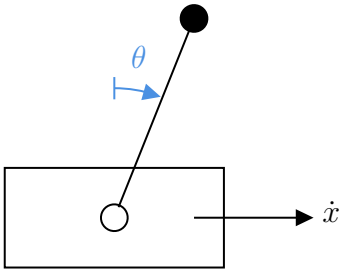


Figure 3.6: Inverted Pendulum Set-Up

4. After his daring stunt with the rock, Théoden runs into Frodo, who is carrying the Phial of Galadriel, which holds the light of Eärendil's star, which has some of the primordial light from the two trees of Valinor.

   The Phial is hanging from a chain. The dynamics of the system can be modelled as a double pendulum with two rigid arms, both of length L, which are free to rotate in the xy-plane. Find the EOM for the system as shown in Figure 3.7.

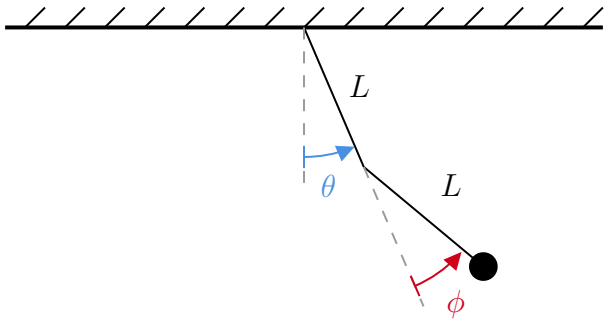


Figure 3.7: Double Pendulum Set-Up

Notes

10. Much of the rotational dynamics we describe can be represented equivalently using the Lagrangian approach, so we do use these equations in a roundabout manner. We will explore this more extensively in Volume III

CHAPTER 4

Computational Attitude Dynamics

"Quaternions came from Hamilton after his really good work had been done; and, though beautifully ingenious, have been an unmixed evil to those who have touched them in any way, including Clerk Maxwell."

– Lord Kelvin, 1892

Attitude Dynamics is the math describing the orientation of a vehicle in space as the result of applied moments. Relying heavily on advanced mathematics, this concept is naturally more difficult to grasp. In fact, we have dedicated the entirety of Volume II to the theory behind attitude determination and dynamics. Derivations for everything from simple, two-dimensional problems and single-axis rotations to *quaternion*s are included in that book. Following the core purpose of this volume, we will only be describing implementation of attitude dynamics using MATLAB functions. However, the entirety of Volume II contains the theory necessary to gain a more definitive understanding of what, and more importantly why, everything is happening with the code in this section.[11]

4.1 Euler Angles

The first method to express the orientation of a vehicle in space is through the use of *Euler angles*. These form the basis for our understanding of attitude determination, being the more intuitive method. As stated previously, the theory and derivations for everything you will use here is included in Volume II! If you haven't read this yet, don't worry, we will provide some basic definitions here to give you the bare bones of what is needed in the function calls to help you and your code be successful.

First and foremost, we use the intrinsic (body-fixed) 3-2-1 rotation sequence to

determine Euler angles. This means we have an angle ϕ to describe the rotation about the x-axis, an angle θ to describe the rotation about the y-axis, and an angle ψ to describe the rotation about the z-axis. The order in which we complete these rotations is Z-Y-X, or ψ-θ-ϕ.[12] Remembering this rotation sequence is very important, as every function call within your MATLAB script will depend in part on this chosen sequence!

4.1.1 The Direction Cosine Matrix

To start, we want a way to easily convert any arbitrary vector from being described using one reference frame to another. We do this through the use of the Direction Cosine Matrix, or DCM. This allows us to take this matrix and multiply it with our vector in the inertial frame and obtain an identical vector expressed using the body frame as shown:

$$^{b}\vec{r} = A_{3,2,1}\ ^{i}\vec{r}$$

In MATLAB, there exists a function - `angle2dcm` - that creates this for us![13] By inputting the scalar values of our Euler angles (in radians!) as well as the chosen rotation sequence, MATLAB creates the DCM as an output, allowing us to pre-multiply this matrix with our inertial vector we want to express in the body frame. Note the function allows an input of any number of rotations to find DCMs for. This is a fun little aspect of the function; however, it is irrelevant for our use as we input our new Euler angles every time-step during the `ode45` integration scheme. One additional thing to consider: the `angle2dcm` function returns the DCM as a transform from the inertial frame to the body frame. If we were to transform a vector from the body to inertial frame, we would need to take the inverse - in this case transpose - of the given matrix.[14] Further reading on this function can be seen in Mathwork's own documentation at Source [9].

4.1.2 Relating Euler Rates with Angular Velocity

One thing you might recall if you have read Volume II is that integrating the angular velocities in a 3-dimensional case DOES NOT return the Euler angles. Giving a quick reason as to why: angular velocity is expressed exclusively with respect to the body frame, while our Euler angles are defined about their corresponding intermediate frame in the rotation sequence. Meaning this job for integration is left to *Euler rates*,

something else we need to include in the code. All you brilliant readers will also recall the existence of what we affectionately call the "B" matrix that allows us to use Euler angles to express a relation between Euler rates and angular velocities. Unfortunately, there doesn't exist a MATLAB function that can calculate the "B" matrix, leaving us to our own devices to construct one from scratch. Luckily, we already did the heavy lifting, meaning we can copy the matrix derived in Volume II straight into our MATLAB script. That matrix and equation is as follows, for quick reference:

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = B \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}$$

Euler Rates

$$B = \begin{bmatrix} 1 & tan(\theta)sin(\phi) & tan(\theta)cos(\phi) \\ 0 & cos(\phi) & -sin(\phi) \\ 0 & \frac{sin(\phi)}{cos(\theta)} & \frac{cos(\phi)}{cos(\theta)} \end{bmatrix}$$

The "B" Matrix for $3-2-1$ Rotations

4.1.3 Euler Angle MATLAB Function Example

Putting it all together, we're going to write a simple code using Euler angles to practice using MATLAB's built in functions. Say we have a very important vector we want to express in the body frame and another important vector we want to express in the inertial frame: gravity and thrust respectively. If our rocket weighs 100 Newtons and outputs a thrust of 1000 Newtons, we know this appears in vector notation as: $-100\hat{x}$ and $-1000\hat{X}$ respectively. Consider a moment in time, when - using the 3-2-1 rotation sequence - our vehicle is at a roll angle of 2.71 radians and a yaw and pitch angles of 0.2 radians, where we want to convert our gravity vector from the inertial frame to the body frame. In addition, with knowledge that the angular velocity at that instant is $\begin{bmatrix} 2.5 & 0.1 & 0.1 \end{bmatrix}^T$, we can also find the Euler rates of the system. The following code listing quickly runs through the process, with sufficient commentary within the script explaining every action taken.

Code Listing 4.1: Euler Angle Dynamics Example

```
1 % Title: GravityToBodyEuler
```

```
% Author: Preston Wright
% Example converting a gravity force from the inertial to body frame and thrust force from body to inertial frame using Euler angles

% Start with your initializations. Every angle should be in radians, and our forces are in Newtons
phi = 2.71;                     % x-axis angle [rad]
theta = 0.2;                    % y-axis angle [rad]
psi = 0.2;                      % z-axis angle [rad]
gravityInertial = [-100;0;0];   % inertailly expressed gravity force [N]
thrustBody = [-1000;0;0];       % body expressed thrust force [N]
omega = [2.5;0.1;0.1];          % angular velocity [rad/s]
bMatrix = [1, tan(theta)*sin(phi), tan(theta)*cos(phi); ...
           0,       cos(phi),         -sin(phi); ...
           0, sin(phi)/cos(theta), cos(phi)/cos(theta)];

% Calculate the DCM for the instantaneous orientation of the vehicle. Remember we're using a 3-2-1 rotation sequence for this problem; therefore, we will input the corresponding z, then y, then x angle and define our order at the end
DCM = angle2dcm(psi,theta,phi,"ZYX");

% Matrix multiply the DCM with the inertial gravity vector to find the body frame representation of the gravity vector. Multiply the thrust vector by the transpose (inverse) of the DCM to obtian the thrust vector in the inertial frame
gravityBody = DCM*gravityInertial;
thrustInertial = (DCM')*thrustBody;

% We now want to multiply our b matrix by the angular velocity vector to get the Euler rates
eulerRate = bMatrix*omega;
```

Our results come out to be the following, with the gravity force expressed in the body frame and thrust expressed in the inertial frame as:

$$ {}^{b}\vec{F}_g = \begin{bmatrix} -96.0530 \\ -26.1902 \\ 9.3748 \end{bmatrix} ; \; {}^{i}\vec{F}_T = \begin{bmatrix} -960.5305 \\ -194.7092 \\ 198.6693 \end{bmatrix} $$

At this point, we would use our numerical integration scheme to approximate the next timestep's Euler angles using the Euler rates we just found. We won't do that here, but you can see a similar implementation with quaternions in the 6DoF itself or the Dzhanibekov example!

4.2 Quaternions

To match industry/academia standards and improve the robustness of our simulation, our 6DoF model uses *quaternion*s. The use of hyper-imaginary numbers in a normalized vector allows us to avoid losing a degree of freedom at a specific orientation of our rocket (called gimbal lock), which is what makes the simulation more robust. Just as with Euler angles, MATLAB has an extensive suite of functions that makes the implementation of quaternions just as easy.

4.2.1 Quaternion Rotation Matrix

We're going to combine a few initial steps of the code surrounding quaternion attitude dynamics. To start, we want to convert from Euler angles to quaternions, as Euler angles are more intuitive to define and initialize in the simulation. MATLAB's function `eul2quat` does just that. By inputting the Euler angles in the appropriate sequence, along with the chosen rotation sequence (make sure to match the Euler angles with the order), the function returns the corresponding quaternion vector. Mathworks documentation for this function can be found at Source [9].

Having initialized our quaternion vector, we can now use another MATLAB function to construct our DCM. The function `quat2dcm` takes the quaternion as an input, and returns the DCM at that instant. Incredibly simple and incredibly useful; however, we must once again consider that this DCM converts only from the inertial frame to the body frame. Meaning just as we had done with Euler angles, it is necessary to take the transpose (inverse) of the matrix if we want to transform from the body frame to the inertial frame. The Mathworks documentation for the `quat2dcm` function can be found at Source [10].

4.2.2 Quaternion Rates

Just as we had done for Euler angles with Euler rates, we want to find a vector of quaternion rates that allows us to integrate and obtain a new quaternion vector. Also just as with Euler angles, there is no MATLAB function allowing us to relate quaternion rates with angular velocity. But once again, just as with Euler angles, we've already found the necessary matrix to do so! This equation is shown in (4.1).

So plugging it into our script is an easy fix to make this operation possible. The calculation is included here for quick reference:

$$\dot{\mathbf{q}} = \frac{1}{2}\begin{bmatrix} 0 & -\omega_x & -\omega_y & -\omega_z \\ \omega_x & 0 & \omega_z & -\omega_y \\ \omega_y & -\omega_z & 0 & \omega_x \\ \omega_z & \omega_y & -\omega_x & 0 \end{bmatrix} \vec{q} \tag{4.1}$$

Quaternion Rates

4.2.3 Quaternion MATLAB Function Example

Let's take the exact same example as before. We have the force of gravity equal to $-100\hat{x}$ and the thrust force equal to $-1000\hat{X}$. We have a roll angle of 2.71 radians, yaw and pitch angles of 0.2 radians, and we have chosen a 3-2-1 rotation sequence. We finally have an instantaneous angular velocity of $\begin{bmatrix} 2.5 & 0.1 & 0.1 \end{bmatrix}^T$. Wanting to obtain the gravity vector expressed in the body frame instead of the inertial frame, and vice versa for the thrust vector, this time we're going to use quaternions to do so. In addition, we're now going to find the quaternion rate at the given moment. The listing below does this, and is sufficiently explained in the comments between operations to hopefully give you as the reader a sufficient understanding of what is happening. Note: you should see numerous similarities, with some sections of the code identical to the previous.

Code Listing 4.2: Quaternion Dynamics Example

```
% Title: GravityToBodyQuat
% Author: Preston Wright
% Example converting a gravity force from the inertial to body frame
    and thrust force from body to inertial frame using quaternions

% Start with your initializations. Every angle should be in radians,
    and our forces are in Newtons
phi = 2.71;                     % x-axis Euler angle [rad]
theta = 0.2;                    % y-axis Euler angle [rad]
psi = 0.2;                      % z-axis Euler angle [rad]
gravityInertial = [-100;0;0];   % inertailly expressed gravity force
    [N]
thrustBody = [-1000;0;0];       % body expressed thrust force [N]
omega = [2.5;0.1;0.1];          % angular velocity [rad/s]
```

```
bMatrixQuat = [0, -omega(1), -omega(2), -omega(3); ...
               omega(1), 0, omega(3), -omega(2); ...
               omega(2), -omega(3), 0, omega(1); ...
               omega(3), omega(2), -omega(1), 0];

% Before moving on to calculating our rotation matrix, we want to
   convert the Euler angles to quaternions like so (remember our
   rotation order!):
quat = eul2quat([psi,theta,phi],"ZYX");

% Here we will also get into the habit of normalizing any
   quaternions we compute, as normalized quaternions (versors) are
   the only way to express quaternion encoded rotations
quat = quat/norm(quat);

% We can now call the funciton to find our rotation matrix
quatDCM = quat2dcm(quat);

% Multiplying our DCM by the gravity vector will give us the gravity
    vector expressed in the body frame. Multiplying the transpose (
   inverse) of the DCM by the thrust vector will give us the thrust
   vector in the inertial frame
gravityBodyQuat = quatDCM*gravityInertial;
thrustInertialQuat = (quatDCM')*thrustBody;

% To prepare for matrix multiplication in the following step, we
   need to transpose the given quaternion vector to make it a column
    vector
quat = quat';

% Calculating our quaternion rate using the known formula appears as
    follows:
quatRate = (1/2)*bMatrixQuat*quat;
```

Taking a look at the outputs this generates, we see our results come out to be the following, with the gravity force expressed in the body frame and thrust expressed in the inertial frame as:

$$ {}^{b}\vec{F}_g = \begin{bmatrix} -96.0530 \\ -26.1902 \\ 9.3748 \end{bmatrix} ; \; {}^{i}\vec{F}_T = \begin{bmatrix} -960.5305 \\ -194.7092 \\ 198.6693 \end{bmatrix} $$

A quick note about this script in relation to the other: the results shown above and the results shown in subsection 4.1.3 are identical. This gives confidence that we've implemented MATLAB's functions correctly. And once agian, at this point, we would use our numerical integration scheme to approximate the next timestep's quaternion

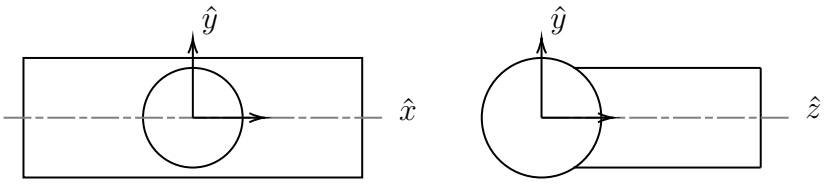
using the quaternion rates we just found. We won't do that here, but as a reminder, you can see the implementation in the 6DoF itself or the Dzhanibekov example!

4.3 3D Rotation Example: Dzhanibekov effect

To help digest the complex nature of attitude dynamics, a full example is especially useful. One of the consequences of Euler's Equations described above is that rotation about an intermediate axis is unstable. This effect is known as the Intermediate Axis Theorem or the *Dzhanibekov Effect*.

This is a good phenomenon to describe 3D rotations because it requires us to put together lots of knowledge throughout this section. Because our object rotates about all three axes, we must parametrize our rotation with quaternions to avoid possible singularities in the solution.

Before we can describe our system dynamics, we need to establish our coordinate system for this problem.



$$I_x = 1.202 \cdot 10^5 \ g/mm^2$$
$$I_y = 4.615 \cdot 10^5 \ g/mm^2$$
$$I_z = 4.010 \cdot 10^5 \ g/mm^2$$

Figure 4.1: Non-Symmetric Body for Demonstrating Dzhanibekov Effect

4.3.1 State Vector

We'll start by describing our state vector and initial conditions for this problem:

We define our original position is at the origin, giving us a position vector:

$$\vec{r_0} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

We also define the initial velocity to be zero, giving a velocity vector:

$$\vec{v_0} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

We also need an initial orientation. We can either directly initialize a quaternion or write our initial orientation with Euler angles and then convert to quaternions. Often, Euler angles are more intuitive, so we will use these for our initialization. We can define an initial angle of all zeros, meaning that our body frame axes are initially coincident with the inertial frame. This gives Euler angles of:

$$\begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Using the command `eul2quat` with a specification of `ZYX` for the frame, the initial quaternion vector is given as:

$$\vec{q_0} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Lastly, we need to describe the initial angular velocity. To properly demonstrate the effect, we need a small perturbation on our non-intermediate axes. We can write our omega vector as:

$$\vec{\omega_0} = \begin{bmatrix} 0.05 \\ 0.05 \\ \pi \end{bmatrix} rad/s$$

Putting all of the together, we arrive at our state vector:

$$\vec{X_0} = \begin{bmatrix} \vec{r_0} \\ \vec{v_0} \\ \vec{\omega_0} \\ \vec{q_0} \end{bmatrix}$$

Which we can expand to the full form as a 13-element column vector:

$$\vec{X}_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0.05 \\ 0.05 \\ \pi \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

4.3.2 Forces and Moments

We can assume that we are operating in a 0-g environment (as this is where the effect is most prevalent) and that no body forces are acting on the object. We will also assume that air resistance in negligible and there are no losses. These assumptions mean that there are no forces that the acting on our body. Additionally, there are no moments on the body either. We have chosen this case for our example so that we can focus solely on the attitude dynamics.

4.3.3 Attitude Dynamics

Our attitude dynamics are fairly simple since there are no moments acting on the body. Our equations for the angular velocity rates in (2.13) reduce to the torque free equations:

$$\dot{\omega}_x = \frac{(I_y - I_z)\,\omega_y\omega_z}{I_x}$$
$$\dot{\omega}_y = \frac{(I_z - I_x)\,\omega_z\omega_x}{I_y}$$
$$\dot{\omega}_z = \frac{(I_x - I_y)\,\omega_y\omega_x}{I_z}$$

To find our quaternion rates, we simply use the same methods as in subsection 4.2.2. This gives us:

$$\dot{\mathbf{q}} = \frac{1}{2}\begin{bmatrix} 0 & -\omega_x & -\omega_y & -\omega_z \\ \omega_x & 0 & \omega_z & -\omega_y \\ \omega_y & -\omega_z & 0 & \omega_x \\ \omega_z & \omega_y & -\omega_x & 0 \end{bmatrix} \vec{q}$$

4.3.4 Translating to Code

With all of these defined, we can write the derivative of our state vector as:

$$\dot{\mathbf{X}} = \begin{bmatrix} \vec{v} \\ \vec{a} \\ \vec{\alpha} \\ \dot{\mathbf{q}} \end{bmatrix}$$

From here, we can implement this in `ode45`. We show all of the code for this model in the appendix.

Our initialization of the state vector and all of the basic setup in contained in the main file, shown in Script 8.2.2. The code that gets the derivative of the state vector is the function passed into ode45. This code is shown in listing Script 8.2.2. There is additionally a file to display the rotation of the body in an animation, which is shown in Script 8.2.2.

4.4 Key Ideas

1. Read Volume II to learn more about the theory behind MATLAB's functions
2. Rotations can be encoded in a variety of ways, but the most useful, compact, and robust method of encoding rotations involves the use of complex numbers.
3. The use of MATLAB's built in suite of functions is incredibly useful in the implementation of attitude dynamics in 6DoF modeling, and rotational simulations in general.

4. Be sure to keep your chosen rotation sequence straight when utilizing MATLAB's functions, and be sure to have a thorough understanding of how these functions work!

5. No matter which style of describing rotations we choose - Euler angles or quaternions - they both result in identical answers barring any singularities generated with the former.

4.5 Further Reading

Volume II of this text contains most of the information pertaining to attitude dynamics. For reference until Volume II is released, we find the website [12] especially useful for understanding quaternions.

4.6 Practice Problems

1. Your friend Jimothy, impressed by your MATLAB skills, has challenged you to model a spinning top. The top is axially symmetric about its $\hat{X}$-axis and has an initial spin rate of 1200 rpm and has an initial angle to the vertical of 20°. The top spins down at a rate governed by an exponential decay of form $\omega(t) = \omega_0 e^{\frac{1}{10}\ln(2)t}$ such that the spin rate is halved every 10 seconds. Gravity acts on the top in the negative $\hat{x}$-direction. The top has a mass of 20 grams. Relevant details are given in Figure 4.2. Define any additional quantities that you deem necessary to solve the problem.

   (a) Write the state vector for the system.

   (b) Write the EOMs for the system. If you choose the Newtonian approach, write the forces and moments of the system. If you choose the Lagrangian Approach, write the energy equations and constraint equations.

   (c) Numerically integrate the system. For this problem, you may be able to use either Euler angles or quaternions if you choose your singularity point carefully.

   (d) Create a visualization of the system. Create any other plots that you deem useful.

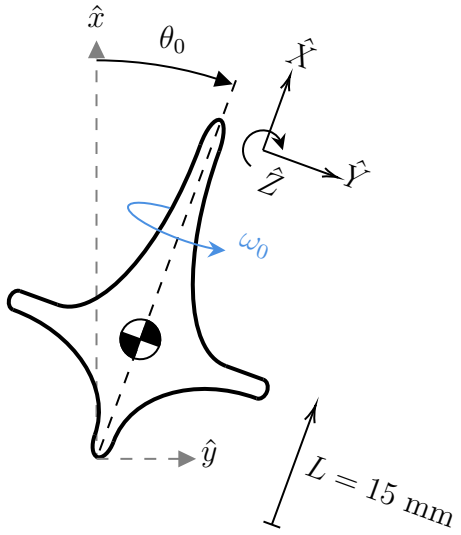


Figure 4.2: Spinning Top Set-Up

2. After your stint in modeling the Explorer 1 Satellite, NASA has asked you to come back to model a new satellite. NASA wants the satellite to be nadir-pointing at all times, and has decided to use gravity gradient stabilization to do so.

Notes

11. Volume II is yet to be released, and is currently scheduled to do so late Summer 2025.

12. These intermediate rotations about the body frame are sequentially and individually applied to our system as we move from frame to frame in our rotation order.

13. Note there exists another similar function - eul2rotm - that also creates a DCM; however, this function converts a vector from the body frame to the inertial frame.

14. We are allowed to use the transpose here as the DCM is an orthogonal matrix.

CHAPTER 5

Aerodynamic Modeling

"When I meet God, I am going to ask him two questions: Why relativity? And why turbulence? I really believe he will have an answer for the first."

– Werner Heisenberg

In Section 2.3 we discussed the 4 fundamental forces on a vehicle and how we define their directions. In that section we did not define the magnitude of the aerodynamic forces, $\vec{F}_L$ and $\vec{F}_D$. The magnitude of these forces are quite difficult to calculate and require the most approximation to do correctly. The governing equation for calculating the magnitude of our aerodynamics is $\frac{1}{2}\rho_\infty V_\infty^2 S C_X$, where C_X is either C_L, C_D, C_N, or C_A.

Despite the innocuous appearance of this equation, calculating the aerodynamics of our system is deceptively hard because the calculation of C_X is very complicated. In fact, without real data, any calculation that we made of these values will just be an approximation.[15] That being said, there are approaches we can take which will get us close enough to true values to prove useful.

5.1 Non-Dimensionalization

It is useful to understand why the equations for the aerodynamic forces take the form that they do, with the $\frac{1}{2}\rho V^2 S$ term. Along the way, we will also discover some other important properties that influence the airflow. We will start by enumerating the various factors that could impact the drag experienced by an object. Some intuition will tell us that drag is likely affected by the following factors:

1. Viscosity of the fluid that is being traveled through - higher viscosity creates

more 'stickiness' in the air that needs to be countered.

2. Velocity - higher velocity creates more interactions with the fluid at any given time, so it will contribute to our drag force.
3. Density of fluid - higher density will create more interactions per time.
4. Characteristic size - the size of the object traveling through the fluid will increase the particle interactions per time and drag.
5. Speed of sound - the properties of the fluid will depend on the speed of sound in the fluid.
6. Angle of attack - the angle of the object will change the apparent size in the airflow.

Already, we can see that some of the terms that we see in the lift and drag equations are present above. However, we do not know in what may they are correlated and what else they depend on.

To determine these dependencies, we can invoke *dimensional analysis* and the *Buckingham Pi Theorem*. Both of these will allow us to derive dimensionless quantities that prove to be particularly useful.

Theorem 2 (Buckingham Pi Theorem)**.** *Given a system with N parameters and K physical dimensions, it is possible to express the result of this system with P dimensionless Π groups, where $P = N - K$.*

Corollary 2.1. *Additionally, each Π group is independent of each other. Thus, each Π group is a function of all the other Π groups:*

$$\Pi_1 = f\left(\Pi_2, \Pi_3, \cdots \Pi_{N-K}\right)$$

and so on for the other Π groups.

For more information on Buckingham Pi Theorem, refer to [23]

Example 5.1: Aerodynamic Forces

Using the system characteristics described before, let us use Buckingham Pi Theorem to write the non-dimensional groups for our aerodynamic system. We start by writing the dimensions for each of the variables we enumerated:

Variable	Dimension
ρ	$\frac{M}{L^3}$
V	$\frac{L}{T}$
S	L
μ	$\frac{M}{LT}$
a	$\frac{L}{T}$
α	$-$

Applying Buckingham Pi Theorem, we can create 4 Π groups from the 7 variables.

The solving of these Π groups is shown in the Appendix in subsection 8.1.1 because it is quite lengthy.

Solving the Π groups, we get:

$$\Pi_1 = \frac{\rho V S}{\mu}$$
$$\Pi_2 = \frac{V}{a}$$
$$\Pi_3 = \frac{F}{\rho V^2 S}$$
$$\Pi_4 = \alpha$$

Π_1 is a quantity known as the Reynolds number, the ratio of inertial forces to viscous forces. Π_2 is the Mach number, the ratio of velocity to the speed of sound of the fluid. Π_3 is the force coefficient that we have described earlier. Π_4 is the relation for the angle of attack, which is just itself since α is unitless.

From Corollary 2.1, we can also write

$$\boxed{C_x = f\left(Re, M, \alpha\right)}$$

This is a very important condition of the theorem, showing that the force coefficient is a function of the Reynolds number, Mach number, and angle of attack. Thus, whenever we do modeling of aerodynamics, we only need to consider Reynolds number, Mach number, and angle of attack. This will become a cornerstone of how we model aerodynamics in our 6-DoF modeling.

5.2 Aerodynamics Terminology

Aerodynamics has a lot of terminology associated with it that is important to understand before we can approach modeling the aerodynamics of the system. We will mostly cover aerodynamics terminology for rockets, but the modeling done here should be transferable to aspects of aircraft systems as well.[16]

While some of the theory here is not directly used in our 6-DoF, having an understanding of this will be greatly beneficial so we have chosen to include some background here.

5.2.1 Lift and Drag

We start this section by noting that lift and drag are just conventions that are commonly used to describe the aerodynamic force. It should be noted that both the lift and the drag arise from the same source - the air acting on the vehicle to create a force. It is simply convenient to define this resultant force with respect to the direction of the freestream velocity, which we call lift and drag.

When modeling aerodynamic forces, we can imagine that two differential force elements act at each point along the body. At each point, there is a pressure which acts normal to the surface and a shear stress (you can think of it as a friction) that acts tangentially to the surface. This shear stress arises from the viscosity of the air. These are shown in Figure 5.1.

Since a force is a pressure multiplied by an area, $F = pA$, we can integrate our pressure distribution over an area to understand the total forces that are acting on a geometry. Often, we see this expressed in the form of an integral over the length of the geometry. This length is denoted by c, the chord length, in the example of the airfoil. These integrals are shown below:

$$N' = \int_0^c p_l u - p_u dx + \int_0^c \left[\tau_u \frac{dy_u}{dx} + \tau_l \frac{dy_l}{dx} \right] dx$$

$$A' = \int_0^c \left[p_u \frac{dy_u}{dx} + p_l \frac{dy_l}{dx} \right] dx + \int_0^c (\tau_u + \tau_l) dx$$

Where the N' and A' are the normal and axial force on the aerodynamic surface.

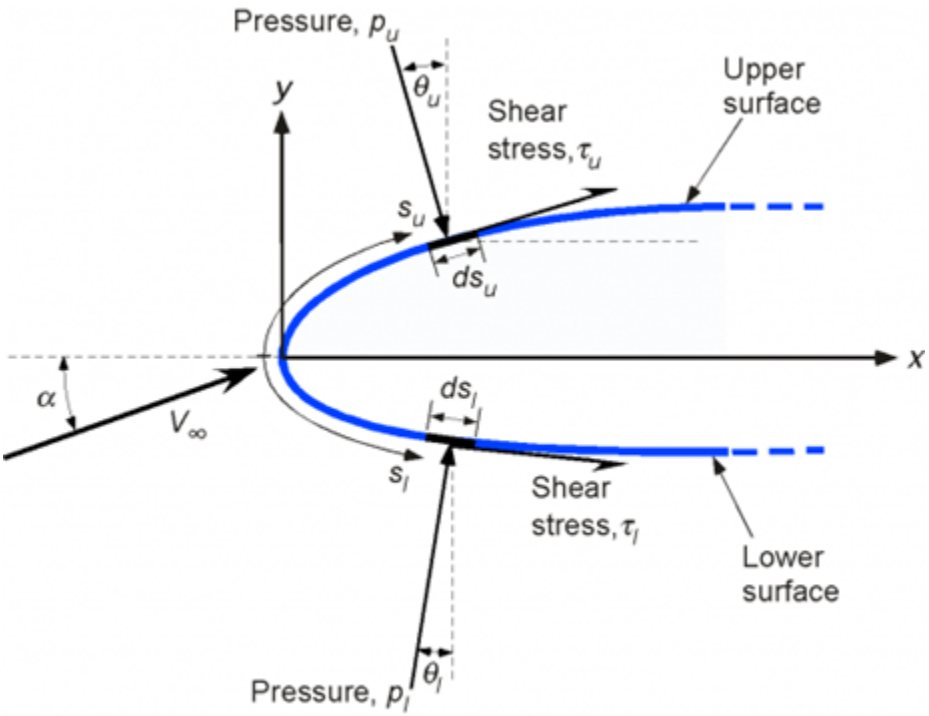


Figure 5.1: Pressure and Shear Stress on an Aerodynamic Surface

Here, the apostrophe (We would say 'N prime') refers to the fact that the force acts over a unit length in the $\hat{z}$ direction. We are assuming a unit length here since we are considering the 2-D case in this example. More generally, we would perform this calculation in 3D, but that is beyond the scope of this example.

The normal force acts normal in the $\hat{y}$ direction and the axial force acts in the $\hat{x}$ direction as shown in Figure 5.1. Normal and axial forces are not commonly used in our modeling outside of these calculations. More often, we see the quantities lift and drag, which are defined with respect to the freestream velocity. We can easily transform our normal and axial forces into a lift and drag using a rotation matrix with our angle of attack, α. This rotation matrix takes the form:

$$\begin{bmatrix} D' \\ L' \end{bmatrix} = \begin{bmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} A' \\ N' \end{bmatrix} \tag{5.1}$$

We can expand (5.1) to give two scalar equations:

$$L' = N'\cos\alpha - A'\sin\alpha$$
$$D' = A'\cos\alpha + N'\sin\alpha$$

So, if we know the distribution of pressure over our body, we have shown that we can find the resultant aerodynamic forces acting on our body. As briefly described in subsection 2.3.3, another important parameter to find is our center of pressure.

5.2.2 Center of Pressure

Simply stated, the center of pressure is the location at which all the aerodynamic forces acting on a body produce no resultant moment. This location is particularly useful for our case because we can assume that the lift and drag forces act through the center of pressure rather than assuming that lift and drag act as force distributions. This will make our future calculations much simpler, so we would like to know the center of pressure and how to calculate it.

To find the location where the center of pressure occurs, we can take the moment about any point first. Often, we choose to take the moment about the nose of the rocket (called the leading edge when referring to a wing), since we define the origin here. This integral has a similar form to the normal and axial force integrals except that we multiply the force by the distance from the nose. For more information on why this is done, refer to [6]. Writing out this integral, we get:

$$M'_{LE} = \int_0^c p_u - p_l dx + \int_0^c \left(\tau_u \frac{dy_u}{dx} + \tau_l \frac{dy_l}{dx}\right) x dx$$
$$+ \int_0^c \left(p_u \frac{dy_u}{dx} + \tau_u\right) y_u dx + \int_0^c \left(-p_l \frac{dy_l}{dx} + \tau_l\right) y_l dx$$

Knowing the normal force and the moment about the leading edge, we can find the center of pressure location as:

$$\boxed{x_{cp} = \frac{-M'_{LE}}{N'}}$$

We note that the convention for the moment about the nose is for a positive moment to denote a pitch up (this is why the negative sign is present). This is opposite the convention established by the RHR. For all other moments used in our 6-DoF models, we use the standard RHR convention for moments.

For more information on the derivation of the equations in this section, refer to Chapters 1 and 2 of [1].

For a rocket, we generally consider the body to be *axisymettric* about the longitudinal axis, when the whole structure can be rotated about the axis which we define as $\hat{X}$.

Because our rocket is axisymmetric, we assume that the center of pressure always lies along the longitudinal axis. Although this assumption is not necessarily true, as we will see later, the largest effects on the aerodynamic moments will occur from changes along the longitudinal direction rather than changes along the transverse plane. We can think of this intuitively because of the center of pressure is separated from the center of mass by more than the diameter of the rocket, so changes in this direction have a greater effect.

5.3 Atmospheric Modeling

Another hurdle to overcome when describing the aerodynamics of our system is to find values for our freestream quantities. We explore the calculation of V_∞ in Chapter 2. The other parameters we need to calculate are ρ_∞ and a_∞. Fortunately, the standard density and Mach number of the atmosphere on Earth is well defined. In MATLAB, we can find this density at any altitude with `atmosisa` as:

```
1     [T, a, P, rho] = atmosisa(height);
```

This gives us a standard value of ρ_∞ and a_∞ to use. However, the value of can vary by up to 10% depending on the weather conditions [18]. Similarly, the Mach number has a strong dependence on temperature $\left(M = \sqrt{\gamma RT}\right)$, so the free-stream Mach number may also be different by 10-20% of standard values. Further derivation of this mach relation can be found in [5].

All of this is to say that using `atmosisa` gives us a good baseline model of the atmosphere, but is not the final say in atmospheric modeling. For now, we use `atmosisa` in our modeling because we can live with this variation in our model. However, it may be useful in the future to use other sources of atmospheric data that give location- and time-specific data.

5.4 Computational Aerodynamics

In the case of our rocket, we encounter a variety of flight regimes from subsonic to high supersonic. Because of this variety, hand calculations of our aerodynamics can become very complicated. As a result, we choose to use computational tools to compute the aerodynamics of our rocket.

5.4.1 RasAero

The main tool that we use presently is RasAero. This tool allows us to input the geometry of the rocket and will output the drag coefficient as a function of the Mach number. This is less than perfect, as we lose some of the dependence on Reynolds number. Additionally, the dependence on angle of attack is quite course (only measurements every 2°. That being said, RasAero has been used to model many rocket systems around the same scale as our rocket with an average error in apogee of 3.47% and 80% of the flights within ±10% of the true apogee. For us, this is an acceptable compromise so that we can focus on flight dynamics modeling rather than aerodynamics.

More information on RasAero can be found at [15]. The documentation on this page does a better job explaining the software than we could here in a short summary. Our implementation uses the .csv export files from RasAero in MATLAB.

5.4.2 Other Computational Models

It is worth noting that other computational models exist. RasAero does have shortcomings in modeling some of the aspects of the rocket geometry, so other methods such as Computational Fluid Dynamics (CFD) are also options to be explored. However, in the current state of our model, we have not looked into CFD methods very heavily for direct use in MATLAB. These are in the preliminary stages of investigation and will likely be explored more heavily in future editions of this text.

5.5 Key Ideas

1. When we model a system such as our rocket, we can capture most of the phenomena by capturing data at different Re, M, and α. This is true for both CFD and wind tunnel modeling.
2. While exact analytical solutions are hard to attain in aerodynamics, we can use computational tools to estimate aerodynamics

5.6 Further Reading

There are many good resources for aerodynamics modeling. The first among these, which serves as a good introduction to the topic of fluid dynamics, is [1].

For more complex studies of aerodynamics, we recommend Modern Compressible Flow by Anderson.

5.7 Practice Problems

1. You are flying home one day in your P-51 Mustang (pictured in Figure 5.2 for your viewing pleasure). At cruise, your aircraft is 10,000 ft above sea level.

Figure 5.2: P-51D-5NA Aircraft

(a) What is the density of air at this altitude?

(b) Your wings provide a C_L of 0.3 at cruising speed. Your plane has a mass of 4,000 kg and a planform area of 21.8 m^2. Find the speed necessary to maintain level light at this altitude.

(c) Your airplane is equipped with unguided HVAR rockets. As a daring pilot, you want to model the aerodynamics of this rocket so you can launch it at your friend's house with astounding accuracy (*that* will teach them to never steal your mini corn dogs again).

Using your best engineering judgment of the nose cone shape and any other parameters you cannot find, model the HVAR rocket in RasAero.

(d) As it turns out, shooting rockets at Canadian civilians is illegal (I know, right?). The Royal Canadian Air Force F/A-18's are on your tail now and you will be shot down in 15 minutes unless you can make it to US airspace. You are 160 km from the airspace boundary. You push your throttles to max to book it back home.

At full throttle, your Merlin V12 produces 1676 hp (1250 kw). Your aircraft has a C_D of 0.017. At the same altitude of 10,000 ft, find the maximum flight speed. Will you make it back to American airspace in time?

Notes

15. Even with wind tunnel data, it is still likely that some approximations need to be made. Aerodynamics in general has no definite answers.

16. The principles we describe are the same, but much of the modelling is simpler for rockets since they have axial symmetry in most cases. We will utilize these assumptions early on to simplify this section.

CHAPTER 6

Numerical Integration Schemes

> *"The best that most of us can hope to achieve in physics is simply to misunderstand at a deeper level."*
>
> – Wolfgang Pauli

In our exploration of numerical integration schemes, it will be useful to have a good understanding of ordinary differential equations. We highly recommend utilizing Source [4] for reviewing differential equations.

We note that this section does not include practice problems since most of the other practice problems include numerical integration as part of the process.

6.1 Euler Integration

We have previously discussed the simplest integration scheme, the explicit Euler method, which we used in subsection 2.1.1. Understanding the Euler method is key to understanding and using more complex integration schemes, so we recommend writing a few of your own Euler integration scripts in MATLAB or a coding language of your choice to really solidify your knowledge before moving on to more complex integration schemes.

We formalize the explicit Euler integration below. Given a linear ODE:

$$\frac{dy}{dt} = f\left(x, y\right)$$

With initial condition:

$$y(x_0) = y_0$$

We perform Euler integration first by defining an interval over which to perform the numerical integration. For Euler integration, we equally space these points by our timestep, dt. We can parameterize these points as:

$$x_i = x_0 + i\Delta t$$

Our next step is to find the value of the tangent line at each point. This is trivial, since our differential equation tells us what $\frac{dy}{dt}$ looks like at any value. All we have to do is evaluate:

$$\frac{dy}{dt} = f\left(x_i, y(x_i)\right)$$

The problem here is that $y(x_i)$ is an unknown quantity (if we knew it, we wouldn't have to do integration). So, to find the value of $y(x_i)$ we take an iterative approach. We know the starting value, $y(x_0) = y_0$, so we can find an approximation of the new value by multiplying the timestep by the current slope. For the first timestep, this will look like:

$$y_1 = y_0 + \Delta t \cdot f(x_0, y_0)$$

Now that we have found an approximation for y_1, we can extend this to find approximations for y_2, y_3, etc. This formula is generally extensible as:

$$\boxed{y_{n+1} = y_i + \Delta t \cdot f(x_i, y_i)}$$

Euler's Method

We represent this Euler integration scheme graphically with a simple example. Suppose our differential equation is $\frac{dy}{dt} = 2t$ with initial condition $y(0) = 0$. In this example, we can see from inspection that the solution curve is $y = t^2$. Performing Euler integration with a Δt of 0.5 yields the black approximation in Figure 6.1.

As can be seen from Figure 6.1, large timesteps quickly lead to error in the solution. For a strictly convex curve, our Euler approximation will always give us an underapproximation of the solution. In some cases, however, our solution may diverge so greatly that it becomes unstable and values tend toward infinity.[17]

6.1.1 Errors in the Euler Method

We will use the example of the Euler Method to demonstrate error in numerical integrators. These same concepts will apply when we move on to higher order integrators.

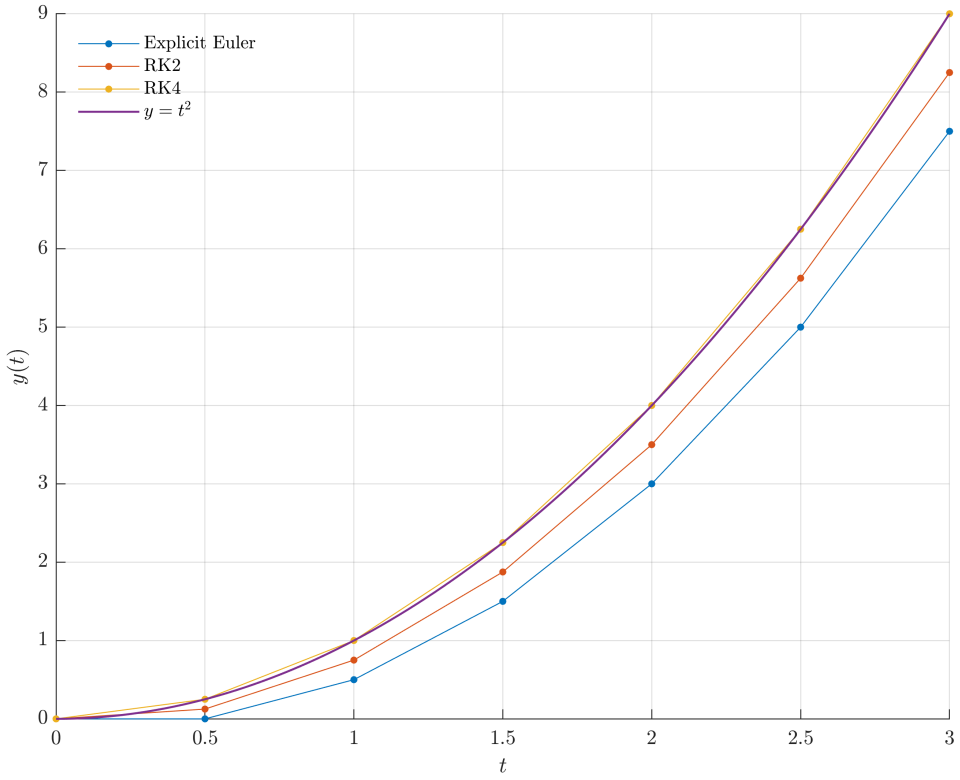


Figure 6.1: Explicit Euler Method Numerical Approximation

Errors in the integration scheme come from two different sources:

1. Euler integration schemes only account for linear terms because it only considers the slope of the tangent line at a point. Thus, any quadratic order terms or higher will not be accounted for. Thus, the error is proportional to the timestep squared, Δt^2.[18] We call this error *truncation error*.

2. Another problem is the fact that computers don't have infinite precision. The loss of precision during computing is called *roundoff error*. Computing more timesteps in total will increase this roundoff error.

It is worth noting that these errors are per timestep! So over the course of a full simulation, our errors will grow much larger than the error bounds that we've calculated above. In fact, for the Euler Method described above, the global error is proportional to the timestep, Δt.

We can of course, reduce errors in the simulation by reducing our timestep, Δt. However, this will also increase the computation time. Additionally, doing so will increase the roundoff error in the simulation. With more calculations being done, the roundoff error will have a larger impact. Thus, it is worthwhile to investigate other integration schemes.

6.2 Improved Euler / RK2

Given the limitations of the Explicit Euler method, we will start by investigating the improved Euler Method. This improved Euler method is the first of a family of integration schemes known as Runge-Kutta (RK) integration schemes. The number, 2, represents the order of the integration scheme, which we will explore more later.

The main area that led to inaccuracies in the explicit Euler method was the slope that we calculated. We can improve on this by taking two points and finding the average slope. We choose to take a point at our start and our end point, x_i and x_{i+1}, and evaluate the slope at both of those points and average them. From the previous section, we can remember that:

$$\frac{dy}{dt} = f\left(x_i, y(x_i)\right)$$

So, we can write the average slope as

$$m = \frac{\dot{y}_1 + \dot{y}_2}{2}$$

Where

$$\dot{y}_1 = \frac{dy_1}{dt} = f\left(x_i, y(x_i)\right)$$
$$\dot{y}_2 = \frac{dy_2}{dt} = f\left(x_{i+1}, y(x_{i+1})\right)$$

Now that we have the slope, we follow the same process as we did for the Explicit Euler method to find the next value. We take the slope and multiply it by the timestep, Δt, and add the current value, $y(x_i)$. This gives us our approximation as:

$$y = y(x_i) + \frac{\dot{y}_1 + \dot{y}_2}{2}\Delta t$$

We can write this then as:

$$y_{i+1} = y_i + \frac{\Delta t}{2}\left[f(x_i, y(x_i)) + f(x_{i+1}, y(x_{i+1}))\right]$$

To rewrite this equation in a way that is better suited for code:

$$y_{i+1} = y_i + \frac{\Delta t}{2}(k_1 + k_2) \tag{6.1}$$

$$k_1 = f(x_i, y_i)$$
$$k_2 = f(x + \Delta t, y_i + \Delta t k_1)$$

We can note here that this method requires at least two evaluations since k_2 is dependent on k_1. However, this is often still more computationally efficient than the Explicit Euler method because larger time steps can be used with lesser error.

6.2.1 RK2 in Code

Implementing the RK2 algorithm in code is slightly more complicated than implementing the explicit Euler method. To properly implement this, we will use *function handles* in MATLAB. Function handles act as pointers to an instance of a function. To explain what that means, we will examine the following code snippet:

```
f = @computeSquare;
a = 4;
b = f(a)
```

where `computeSquare` is defined as:

```
function y = computeSquare(x)
y = x.^2;
end
```

The function handle allows us to assign the function to a variable, which we are calling `f`. We call this a pointer because this variable 'points' to the function `computeSquare`.

In this example, when we call this operation, we get `b=16` in the command window. The power of this is that we could instead pass in a vector of values for `a` and get outputs from the function for all of them. If we instead input:

```
f = @computeSquare;
a = [4,6,10];
b = f(a)
```

We get an output of `b = 16  36  100`. The usefulness of this is immediately apparent for numerical integration. We can pass in a vector of many different elements and get outputs for each of them.

Another useful feature of function handles for numerical integration is that they allow us to pass in only the variables needed for numerical integration and pass other variables in the expression as constants. In general, we want our numerical scheme to only accept our state vector and the time vector. This is the same thing that `ode45` does, so it is best practice to do the same thing here.

Our RK2 algorithm is a direct translation of (6.1) into code. We present this below in Script 6.2.1:

Code Listing 6.1: RK2 Integrator

```
%% RK2 Integrator
% numerically integrates a first order initial value
   problem using Runge-Kutta 2nd order.
% inputs:
% fun - function of integration
% dt - time step [s]
% tIn - input time [s]
% xIn - initial value input vector
% outputs:
% out - final value output vector
function out = rk2(fun, dt, tIn, xIn)
    f1 = fun(tIn,xIn);
    f2 = fun(tIn + dt/2, xIn + dt .* f1);

    out = xIn + (dt / 2)*(f1 + f2);
end
```

6.3 RK4 Integration

RK4 integration is the last major integration scheme that we will explore. RK4 integration is often the best balance between the precision of the algorithm and the speed of computation.

To explain the RK4, we will go back to the RK2 and explain it in a slightly different way that will shed light on what we are doing with RK4. In RK2, we described taking the average slope of the function. We can also imagine taking the Taylor expansion of our function:

$$y_{i+1} = y_i + \dot{y}_i \Delta t + \ddot{y}_i \frac{\Delta t}{2} + O\left(\Delta t^3\right)$$

From this Taylor expansion of the solution, we can that our approximation accounts for linear and quadratic terms. Thus, our error is proportional to terms to the 3rd power, represented by $O(\Delta t^3)$. We know y_i and $\dot{y}_i$, but the 2nd order derivative of our function is unknown. We can approximate this 2nd order derivative as:

$$y_i = \frac{\dot{y}_{i+1} - \dot{y}_i}{\Delta t}$$

Upon rearranging this equation, we can see that we arrive at the same expression as we did before. We can apply the same approach for the RK4 integration, except we consider terms up to the 4th order expansion of the Taylor series:

$$y_{i+1} = y_i + \dot{y}_i \Delta t + \ddot{y}_i \frac{\Delta t}{2} + \dddot{y}_i \frac{\Delta t}{6} + \ddddot{y}_i \frac{\Delta t}{24} O\left(\Delta t^5\right)$$

Immediately, we can see that our error has shrunk to be proportional to terms of the 5th order, a dramatic reduction as compared to RK2 or Explicit Euler method.

6.3.1 RK4 Integration in Code

The RK4 Integration scheme can be expressed as a set of equations very similar to what we did with (6.1). We will not derive these equations here, because the process is very similar to that used for the RK2 method.

The form of these equations is shown in (6.2).

$$y_{i+1} = y_i + \frac{\Delta t}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \tag{6.2}$$

$$\begin{aligned} k_1 &= f(x_i, y_i) \\ k_2 &= \left(x_i + \frac{\Delta t}{2}, y_i + \frac{\Delta t}{2} k_1\right) \\ k_3 &= \left(x_i + \frac{\Delta t}{2}, y_i + \frac{\Delta t}{2} k_2\right) \\ k_4 &= f(x_i + \Delta t, y_i + \Delta t k_3) \end{aligned}$$

An implementation of this algorithm into code is shown below.

Code Listing 6.2: RK4 Integrator

```
%% RK4 Integrator
% numerically integrates a first order initial value
   problem
% inputs:
% fun - function of integration
% dt - time step [s]
% tIn - input time [s]
% xIn - initial value input vector
% outputs:
% out - final value output vector
function out = rk4(fun, dt, tIn, xIn)
    f1 = fun(tIn,xIn);
    f2 = fun(tIn + dt/2, xIn + (dt/2) .* f1);
    f3 = fun(tIn + dt/2, xIn + (dt/2) .* f2);
    f4 = fun(tIn + dt, xIn + dt*f3);

    out = xIn + (dt / 6)*(f1 + 2*f2 + 2*f3+f4);
end
```

Often, we use a function in MATLAB called `ode45` to perform RK4 integration. This function has the benefit that it will change the step size to produce high accuracy when needed and decrease computation time otherwise.

An understanding of the operation of `ode45` is best motivated by an example. We include an example of a Lorenz Attractor in the Appendix subsection 8.2.1 as an

example of ode45. We recommend reading through this code and running it, as well as changing parameters to see how it affects the outputs.

6.3.2 ODE45 Usage

While examples are helpful, it is useful to see the inner workings of `ode45` for a new user.

`ode45` relies on the function handles that we mentioned earlier. `ode45` takes as inputs the integration function, time span, and initial state vector.

`ode45` only takes integration functions that are a function of time and the state variable, or `(t,x)` in MATLAB notation. If we have an integration function that is a function of more than just these two variables, we need to pass them through using function handles. We can do this in the following manner:

```
1 @(t,x) integrator(t,x,a,b)
```

In this example, we have a function that performs our numerical integration, called `integrator`, the output of which will be the derivative of our state vector. We want `ode45` to see this function and only input `(t,x)`, so we put that in a function handle at the front. After the function handle, we are free to pass in any other variables, such as `a` and `b`. These will simply be read in as constants in the `ode45` integration scheme.

The benefit of adding these in as a function pass through instead of being computed in `ode45` will become apparent when we discuss code optimization in Section 6.6. This also keeps our code hierarchy cleaner and more easily modifiable.

6.4 Other Numerical Integration Considerations

6.4.1 More on Error

We briefly discuss the error of the integration schemes in each of the previous sections. However, it is useful to see examples of these to solidify understanding.

Using the same example differential equation from subsection 6.1.1 as $\frac{dy}{dt} = 2t$, we can show the approximation for Explicit Euler, RK2, and RK4. This is shown in Figure 6.2. The code that runs this example is shown in Script 8.2.3 in the Appendix.

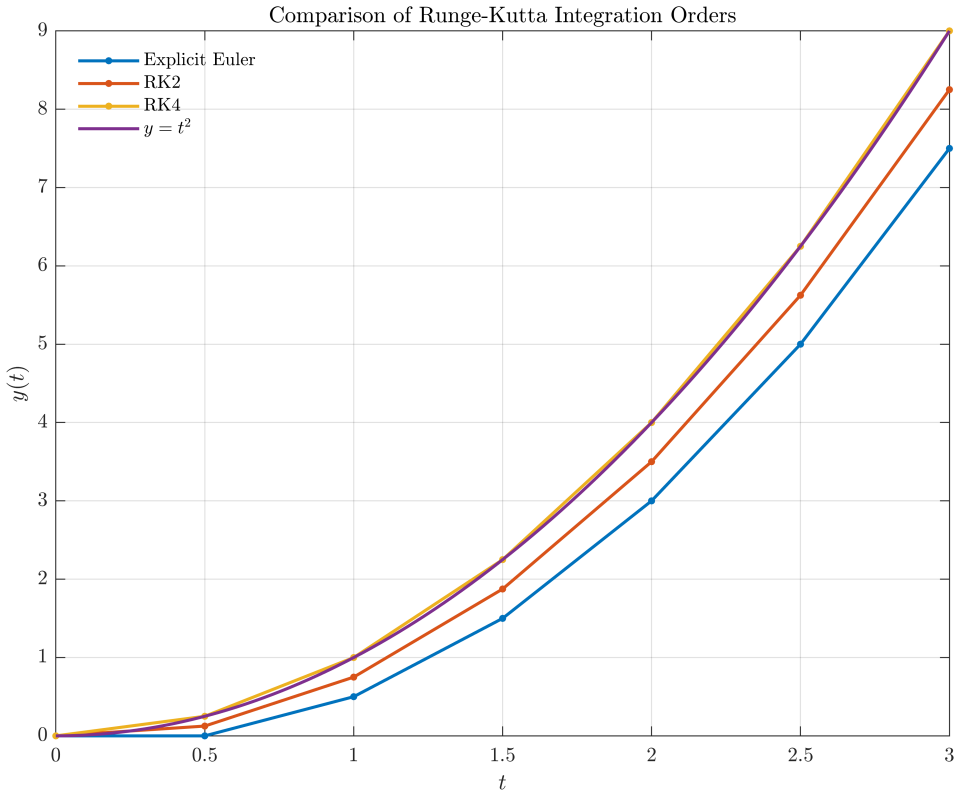


Figure 6.2: Comparison of Numerical Integration Schemes for $y' = 2t$

This code is a good example for understanding the syntax for using the integration code snippets in Listing Script 6.2.1 and Script 6.3.1.

As we can see from this example, even with a fairly large timestep of $\Delta t = 0.5s$, the RK4 perfectly matches the true curve. For this curve, there are no terms of higher order than quadratic, so the RK4 will perfectly match the solution curve.

The difference in these numerical integration schemes will become more apparent when modeling the behavior of different differential equation systems. Systems containing higher order terms will be better modeled by the higher order integration schemes, unsurprisingly. The true use of these methods is when analytical solutions

to the differential equations are hard or impossible to obtain. That being said, we still want to know characteristics of our system so we can appropriately select integration parameters.

6.4.2 Sampling and Stiffness

To begin our discussion of sampling and stiffness, we can take the differential equation $\frac{dy}{dt} = \cos(4\pi t)$with the initial condition of $y(0) = 0$. This function is periodic in nature, so it will be good for our demonstration here. In this first example, we will keep the same timestep of $\Delta t = 0.5s$. As is seen in Figure 6.3, this timestep is the same as the period of the sinusoidal solution of this differential equation. When this is the case, we can see that the solutions diverge from the original periodic function greatly for all of the integration schemes (aside from the RK2, although this is only the case when the period is exactly the same as the timestep). This occurs because we are not sampling the function at a high enough frequency to capture the nature of the periodic function.

So, what timestep is required to accurately model a periodic function? For our example, we will use RK4 since we have shown it to be the most accurate of the integration schemes. We choose to do this to highlight effects aside from error in the numerical approximation. We can vary the timestep for this same periodic function and see what the response looks like. This is shown in Figure 6.4.

In this figure, we note several important things. When our timestep is slightly less than the period of the function, we infer a lower frequency. This is seen when we use a timestep of $\Delta t = 0.4s$. We can intuitively explain this as a *beat frequency*. We know that the solution curve has a frequency of 2 Hz, and a wave with a period of $0.4s$ would have a frequency of 2.5 Hz. Thus, a beat frequency of 0.5 Hz would be produced. This is exactly what is seen, as the resultant solution appears as a sine wave with a frequency of 0.5 Hz (2 second period).

The same is seen when Δt is close to but not quite double the frequency of the wave (called *bandwidth* of the signal). The perceived signal has a changing amplitude with a superimposed sinusoid. This again, does not match the true signal. It is only when we sample at the *Nyquist rate* that a true approximation of the function is found. In general, we must sample our function at twice the highest frequency present to ensure a good approximation!

In this example, this criterion does not greatly affect our numerical integration.

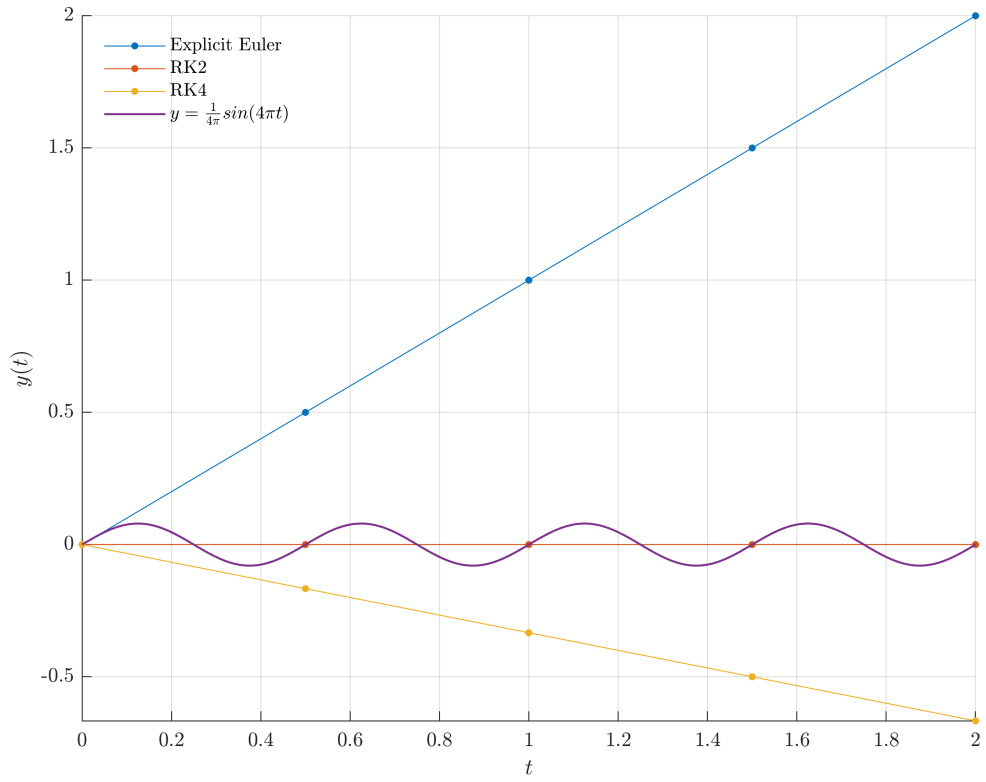


Figure 6.3: Numerical Integration of periodic function

However, when we have other dynamics in the system, the need to sample at double the highest frequency of the system may be detrimental to the performance of our simulation. We call such systems that are difficult to numerically integrate *stiff*.

Stiff systems make numerical integration challenging not only because of increased simulation time, but also because of aforementioned truncation error. In a 6-DoF model such as ours, this is important if vibrational modeling is considered. As an example, slosh modeling and parachute modeling (which are not deeply discussed here, but may be important aspects of a rocket 6-DoF) are often based on mass-spring-damper systems.

It is best practice to try and decouple the simulation of high-frequency vibrations from the flight dynamics of the system. Sometimes this may be unavoidable, such as aircraft with large amounts of wing flexure where aerodynamics and elastic defor-

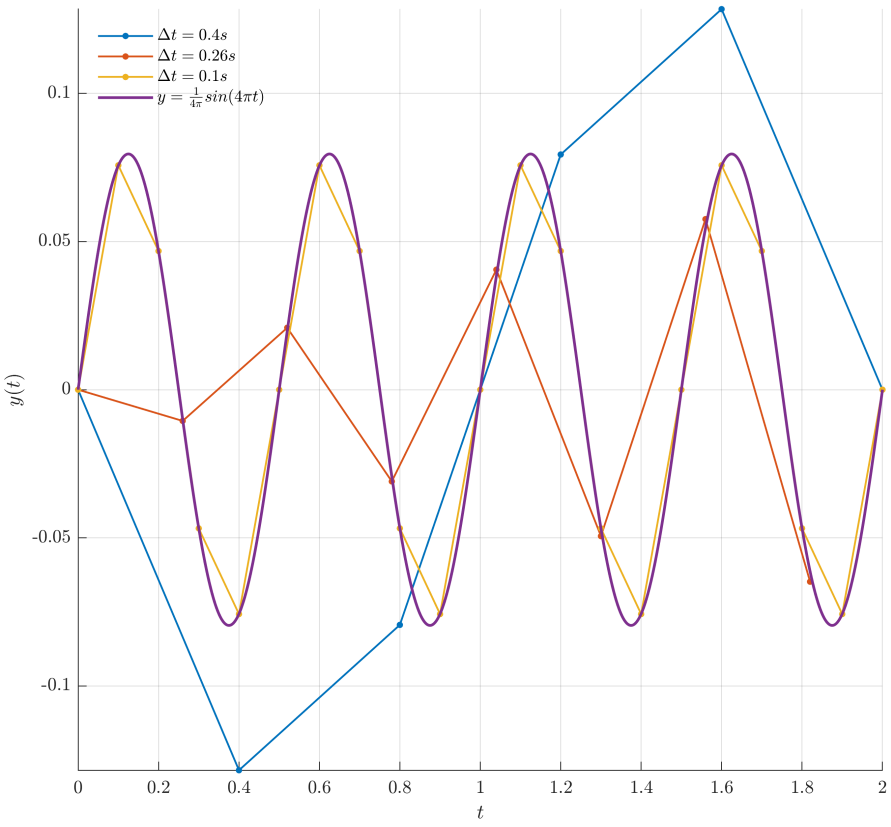


Figure 6.4: Numerical Integration of Periodic Function with varying timestep

mation are heavily coupled. However, this should be attempted whenever possible. This problem is very complicated and is the subject of much ongoing research! More information about such dynamics can be found in [17].

6.5 Other Integration Schemes

While `ode45` works very well for most use cases with numerical integration, there are cases where other integration schemes should be considered. We will mainly discuss the implementations that exist in MATLAB. More information about such systems can be found at [8] and [11].

We previously discuss the concept of stiff systems in subsection 6.4.2.

For systems that have moderate stiffness, we can use `ode23` instead of the more typical `ode45` in MATLAB. Using a lower order integration leads to more course results, but it also allows very stiff parts of the system to be computed faster.

On the converse side, for systems that require very high precision, such as orbits, algorithms of the 8th or 9th order are often used. In MATLAB, such algorithms are used in the `ode78` and `ode89` functions. Using a higher order, the truncation error can be significantly reduced and fewer timesteps are needed when very high tolerances are required. However, in most scenarios, there algorithms are about 300-500% slower than `ode45` while providing few accuracy benefits. Thus, these algorithms are mostly used for only very high precision use cases.

6.5.1 Symplectic and Variational Integrators

In addition to these numerical integration schemes, there are a variety of integration schemes that use energy conservation as the basis of their integration schemes. These types of integrators are called symplectic integrator. These types of integrators directly solve Hamilton's equations for a Hamiltonian system. The form of these equations is discussed briefly in Section 3.4.

Because the Hamiltonian equations are first order already, they are ready for numerical integration without the need for order reduction.

A similar operation can be done to reduce the Lagrangian to two first order equations, in a system known as a variational integrator.

MATLAB has no built in functions for variational or symplectic integrators, so we do not discuss them further here for now.

Use Case of Symplectic Integrators

Although MATLAB does not have built-in functions for these integrators, scripts to do so can be written or found online.

The best use case for these algorithms is for systems with a Hamiltonian or Lagrangian description that need to be integrated over a long time span. Because these

integrators conserve energy, they will better preserve the characteristics of these systems over long times as compared to RK integrators.

6.6 Numerical Integration Code Optimization

6.6.1 Monte Carlo Simulation

Often, we want to run many simulations with various parameters to gain a better understanding of our system. We call this method of running many simulations the Monte Carlo method. Using the Monte Carlo method, we can ascertain information about the likelihood of certain events or approximate quantities that are hard to find analytically by running a large number of simulations and performing statistical analysis on the results. In this document, we will not dive into the methods of statistical analysis and the best ways to perform Monte Carlo simulations. This topic is very deep and is the focus of many peoples' whole careers.

However, even with only rudimentary analysis, Monte Carlo can prove very useful. We will use the example of Monte Carlo to motivate the need for fast simulations. Often, we run upwards of 1000 simulations and fast computation is necessary to perform this large number of simulations. In this section, we will dive into the various methods to optimize MATLAB code and best practices for flight dynamics.

6.6.2 Array and Function Optimizations

MATLAB is generally quite computationally efficient at computing arrays and matrices. After all, MATLAB is 'MATrix LABoratory'. However, we still need to be mindful of optimizations to our code.

It is best practice to only import what you need for a function or subroutine. For example, if you have wind data for all of the months out of the year, it is desirable to cull that down to the specific month before running the simulation.

Additionally, it is also beneficial for any imported data or long matrices to be passed into a function rather than called inside a function. This is especially important inside the numerical integration scheme! Running `readmatrix` inside of ode45

or whichever other numerical integration scheme you are using can slow down code upwards of 500% depending on the length of the matrices being read.

Instead, it is best to pass these arrays into the function. Performing `readmatrix` can be slow because it must check if the values are numeric and organize them into tables. By passing only MATLAB arrays, computations are much faster.

6.6.3 Flame Graphs for Optimization

One important method of optimization is the MATLAB profiler. The primary capability of the profiler is the flame graph. An example of the flame graph is shown in Figure 6.5

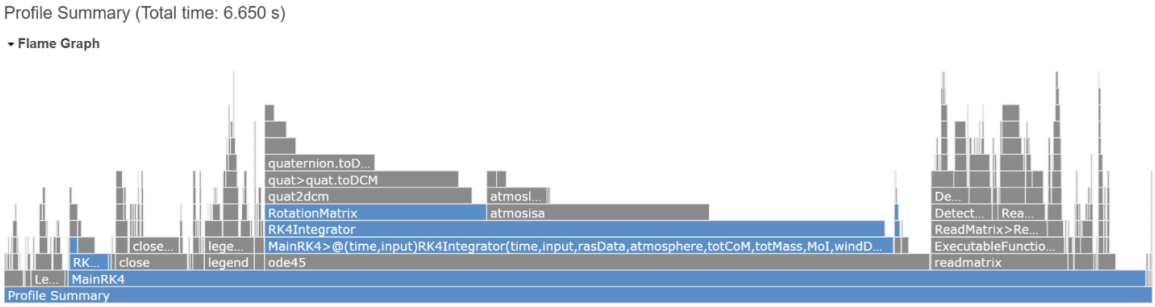


Figure 6.5: Flame Graph Showing Runtime of Individual Code Elements in MATLAB

The flame graph gives a visual representation of how long the code spends on each function. Functions in blue are those defined by the user. Those in grey are built in MATLAB functions.

Poorly optimized code will spend a long time running an excessive number of operations. In some cases, this is unavoidable. For example, ode45, which runs the RK4 integration scheme we discussed in Section 6.3, must run for every time step and is likely to comprise a large part of simulation time.

However, other operations, such as `atmosisa`, should not comprise a large portion of simulation time. As seen in Figure 15, calling the atmosisa function on every loop takes about 40% of the total time spent in 'RK4Integrator' function. By instead creating a table of the atmosisa data before running the simulation, we can reduce the runtime quite drastically. Implementing this simple change, we can see the reduction in simulation time as shown in Figure 6.6

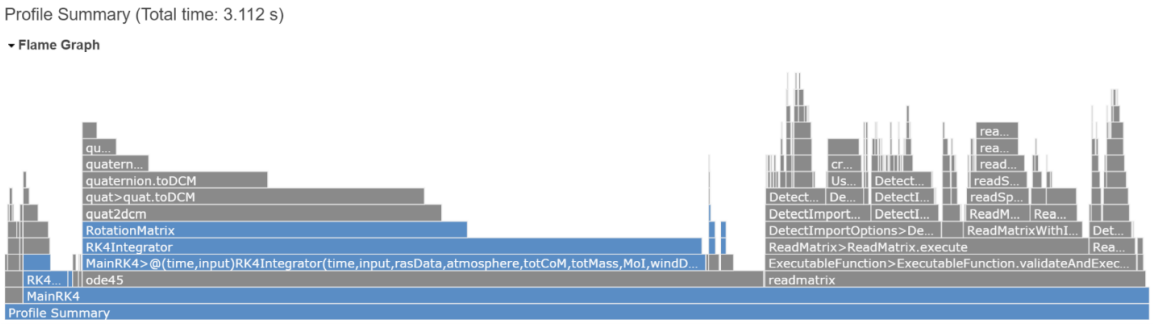


Figure 6.6: Flame Graph After Optimizations

6.7 Key Ideas

1. Numerical integration schemes of higher order can better approximate systems. Generally, a higher order will require more up front computation but will be more accurate and will take fewer steps to achieve a given accuracy

2. When numerically integrating functions with high frequencies or rapid changes, we need to be mindful of stiffness of the system and ensure to sample at at least the bandwidth of the signal.

3. Keep in mind what you need to calculate inside the integrator and what should be passed through. Calculations inside the integral are very time expensive!

6.8 Notes and Further Reading

For more information and derivations on this section, refer to [20]. This source includes some more information than included here and some useful examples.

The MATLAB documentation for numerical integration schemes is quite good. For non-`ode45` integration, we recoommend the following two documentation sites: [8] and [11]. Other documentation exists within MATLAB for other functions.

Notes

17. This is especially true for periodic functions. When working with periodic functions, make sure to sample well below the Nyquist limit.

18. This intuitive explanation is not entirely correct, see [20] for a full derivation using the Taylor Series. We also note that the technically correct way to denote this is $O(\Delta t^2)$, meaning the error is of order Δt^2

CHAPTER 7

The 6-DoF

In this section we will combine everything from the previous sections and explain how our 6-DoF model works. We will follow a similar set up to Section 4.3 here, describing the elements of the model in detail and then attaching relevant code either in-line or in the appendix.

Our definitions of our frames follow from those discussed in Section 2.2. For the earth-centered inertial frame, we follow the conventions discussed in Figure 2.2. Recall that the $\hat{x}$ axis points upward toward the zenith, the $\hat{y}$ axis points east along the surface, and the $\hat{z}$ axis finishes the right-handed coordinate system, pointing north.

For the body frame, we follow the convention in Figure 2.4. This has $\hat{X}$ pointing through the nose of the rocket and $\hat{Y}$and $\hat{Z}$ along the transverse plane. The directionality of $\hat{Y}$ and $\hat{Z}$ is not particularly important since the rocket is axially symmetric, so we arbitrarily select the $\hat{Y}$ axis to point through one of the fins and choose $\hat{Z}$ to complete the right-handed frame.

For our Euler angles, we use a 3-2-1 rotation sequence. We define the zero angles to be when the body frame and the earth-centered inertial frame are coincident.

We note here that this is the section where we have the most room for improvement. We hope that you can translate your knowledge into better features for the 6-DoF as we note some of the simplifications we make in this section.

7.1 State Vector

Our state vector is very similar to that shown in Section 4.3 section, but it bears repeating here because there are some important differences. Another thing to note is that some of the values of the state vector are altered for Monte Carlo simulations.

These possible changes, when applicable, are discussed.

We define our original position at the origin, giving us a position vector. It is also possible to define our initial position in another coordinate system, such as (lat, long, elevation). Generally it is good practice to set the origin to zero and add any offsets later to keep floating point errors in numerical integration small:

$$\vec{r}_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

We also define the initial velocity to be zero, giving a velocity vector. Since velocity is frame independent for an observer at rest with respect to our inertial frame, this is almost always zero. We may see an initial velocity if we are modeling a sounding rocket launched with an initial speed or similar for validation of our model:

$$\vec{v}_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

We also need the initial orientation. Here, we use Euler angles again. We often choose to launch at a slight angle to simulate adverse affects that may push the launch vehicle slightly off-course during the initial launch. We choose to add a small angle in the θ and ψ directions in this case:

$$\begin{bmatrix} \psi \\ \theta \\ \phi \end{bmatrix} = \begin{bmatrix} 0.1 \\ 0.1 \\ 0 \end{bmatrix} \tag{7.1}$$

We note that we could also parametrize this as a 45° rotation on the ϕ axis and then a 0.1 radian angle on either the θ or the ψ axis. For different scenarios one may be more useful than another. For example, if we want to simulate a 10° launch angle (from zenith), we can then use the ϕ angle to determine the azimuth at which we launch.

We use the command `eul2quat` with a specification of `ZYX` for the frame and chosen 3-2-1 rotation sequence. The specification is important now because we need to explicitly define our rotation sequence when two or more rotations are present. In the case of (7.1), this gives an the initial quaternion vector of :

$$\vec{q}_0 = \begin{bmatrix} 0.9975 \\ -0.0025 \\ 0.0499 \\ 0.0499 \end{bmatrix}$$

Lastly, we need to describe the initial angular velocity. We assume for most simulations that there is no angular velocity. For some cases, such as the simulation of *rail whip*, we may choose to put an angular velocity on the rocket.

$$\vec{\omega}_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} rad/s$$

Putting all of the together, we arrive at our state vector:

$$\vec{X}_0 = \begin{bmatrix} \vec{r}_0 \\ \vec{v}_0 \\ \vec{\omega}_0 \\ \vec{q}_0 \end{bmatrix}$$

7.2 Forces and Moments in the 6-DoF

The forces and moments on our rocket are what separate this case from the more simple case of the Dzhanibekov Effect in Section 4.3. We have discussed most of these forces in Figure 2.5, and in Chapter 5, but the exact implementation into the 6-DoF model bears repeating, specifically relating to frame conversion.

7.2.1 Forces

We define 4 main forces that act on the vehicle, those being the 4 forces described in Section 2.3. We will start with the most simple, which is gravity.

Gravity

Because we define the $\hat{x}$ direction as the zenith direction in the inertial earth frame, we define gravity as:

$$F_g = \begin{bmatrix} -mg \\ 0 \\ 0 \end{bmatrix}$$

For gravity, we note that because it acts through the center of mass of the rocket, we do not have a resultant moment to worry about.

Thrust

Thrust is also fairly simple to model. If we consider the thrust to be coincident to the longitudinal axis, the thrust is given by (2.7). For the magnitude, we consider a slightly more complex model than before. Because the rocket is changing altitude, the thrust of the engine will change as the pressure changes. The thrust of the engine, accounting for this pressure change, is given by:

$$F_T = \dot{m}V_e + (P_e - P_{atm}) \cdot A_e$$

Where $\dot{m}$ is the mass flow rate of the engine, V_e is the exit velocity of the engine, P_e is the exit pressure of the engine, P_{atm} is the atmospheric pressure, and A_e is the area of the nozzle exit. For a given propulsion system, these parameters are usually specified or can be easily derived, although we will not do so here. See [3] or the propulsion team for more details.

Here, we introduce the use of the rotation matrix to convert between different frames. MATLAB allows for the creation of a rotation matrix with the input of a quaternion using the `quat2dcm` function. Using this, we can perform frame conversion to and from the body and earth frame. We show this below in Script 7.2.1:

Code Listing 7.1: Rotation Matrix

```
function [vecOut] = RotationMatrix(vecIn, quatIn,
    bodyOrEarth)
% PSP FLIGHT DYNAMICS:
%
% Title: DCM
% Author: Hudson Reynolds - Created: 7/2/2024
% Last Modified: 7/8/2024
%
% Description: This program takes inputs from the body or
    earth frame and
% converts it to the alternate frame given the euler
    angles using a
% rotation matrix.
%
%
% Inputs:
% vecIn = 3x1 column vector [x;y;z]
% angleIn = 3x1 vector with euler angle inputs (x-angle, y
    -angle, z-angle)
```

```
% quatIn = 4x1 vector with euler parameters
% bodyOrEarth = 0 -> convert earth to body, 1 -> convert body to earth
%
% Outputs:
% vecOut = 3x1 vector in the converted frame

    % make the input real for edge cases in Monte Carlo:
    rotIn = real(quatIn.');

    % create the rotation matrix
    R = quat2dcm(rotIn);

    %perform transformation:
    if bodyOrEarth == 0
        vecOut = R * vecIn;

    elseif bodyOrEarth == 1
        vecOut = R.' * vecIn;
    else
        fprintf("Input for body or earth frame is invalid\n");
    end
end
```

Using this code, we can convert the thrust from the body frame into the earth frame through the function call `RotationMatrix(thrustForceBody, quat, 1)`. The input of 1 in this function means that the vector is transformed from the body frame into the earth frame. In our case, this means that we multiply by the transpose of the rotation matrix in the MATLAB notation convention.

Lift and Drag

Our other two forces follow in a similar manner. The quantities of lift and drag are heavily discussed in Chapter 5, so we will keep them succinct here. The directions of lift and drag in the earth frame are described by (2.4) and (2.5). To convert these forces into the body frame, we use the function call `RotationMatrix(force, quat, 0)`. The specification of `0` in the function converts a force in the earth frame into

one in the body frame. We need the body frame force for computing the moments in the body frame, as discussed in subsection 2.4.3.

Parachute Forces

The other force that sometimes acts on our rocket is a parachute force. This acts as another body with drag that is activated in the post-apogee regime. This drag has the same form as the aerodynamic drag, $\frac{1}{2}\rho_\infty V_\infty^2 S C_D$ with the drag acting opposite the freestream velocity. In code, this looks like:

```
forceMag = 0.5 * totCd * vertArea * rho * norm(vel)^2;
paraDragForce = forceMag .* (-vel ./ norm(vel));
```

This force is only active when the vertical velocity is negative, so this is controlled by an `if/else` structure as: `if vel(1) < 0`.

The parachute dynamics can be far more complicated that this, but currently we do not model this added complexity.

Currently, there are no other forces that act in our 6DoF model. Internal forces and vibrations are not currently modeled, as we assume a completely rigid vehicle.

7.2.2 Moments

The modeling of moments in the 6-DoF is fairly straightforward. In the previous subsection 7.2.1, we converted each of our forces into the body frame. Here, we will use these forces in the body frame to describe the moments acting on our vehicle.

The moment arm of the aerodynamic forces is equivalent to the distance from the center of mass to the center of pressure.

$$M^c = \vec{r}_{cm} - \vec{r}_{cp}$$

Assuming that both lie along the longitudinal axis, we can express this moment as:

$$M^c = \hat{X}(x_{cm} - x_{cp})$$

Since gravity acts through the center of mass, the moment arm has length of 0 and no resultant moment exists. For thrust, since it is applied through the longitudinal axis, the $\vec{r} \times \vec{F_T}$ term is zero because $\vec{r}$ and $\vec{F_T}$ are parallel.

Thus, the only moments that are present are from the aerodynamic forces. The moment generated by the aerodynamic forces is:

$$M^c = \hat{X}(x_{cm} - x_{cp}) \times \vec{R}$$

Where $\vec{R}$ is the resultant aerodynamic force (the combination of both lift and drag).

7.2.3 Attitude Dynamics

These moments are used to find angular acceleration, $\vec{\alpha}$ using the Euler rigid body equations described in (2.13).

Lastly, the quaternion rates are described in the same as in Section 4.3.

$$\dot{\mathbf{q}} = \frac{1}{2}\begin{bmatrix} 0 & -\omega_x & -\omega_y & -\omega_z \\ \omega_x & 0 & \omega_z & -\omega_y \\ \omega_y & -\omega_z & 0 & \omega_x \\ \omega_z & \omega_y & -\omega_x & 0 \end{bmatrix} \vec{q}$$

So, taking all of these together, we have the derivative of our state vector needed for numerical integration.

$$\dot{\mathbf{X}} = \begin{bmatrix} \vec{v} \\ \vec{a} \\ \vec{\alpha} \\ \dot{\mathbf{q}} \end{bmatrix}$$

7.2.4 6-DoF Code

We take this integrated state vector along with the initial conditions and integrate them using `ode45`. The two most important scripts in this process are the `MainRK4.m` and `RK4Integrator.m` MATLAB scripts. Because these MATLAB scripts are quite long, we have chosen not to include them in the appendix. Instead, a link to the GitHub is given where the scripts can be found.

The full code can be found at: [14] or in the appendix in subsection 8.2.4.

7.3 Validation of 6-DoF

The validation of our 6-DoF model is achieved through modeling other rocket systems that have flown to check the validity of our model and modeling other physical phenomena. Validation can range from very simple to far more complex depending on the system that is being validated. For us, validation falls under two main categories:

1. Sub-component validation - validation of parts of the 6-DoF by modeling certain types of systems or doing hand calculations to verify correct behavior
2. Full model validation - validation of the whole 6-DoF by modeling a different rocket system and comparing 6-DoF outputs to real outputs.

Sub-component validation may look like the Dzhanibekov modeling in Section 4.3, where we test the attitude dynamics.

Full model validation is usually more complex. One such example is our validation of our model against the V-2 rocket launch on March 7, 1949. This comparison is shown in Figure 7.1

7.4 Data Visualization

Here, we will briefly discuss data visualization and best practices for MATLAB. No matter how good your simulation is, you must be able to appropriately share data in a way that is not only easily conveyed to a non-expert audience, but also visually interesting and descriptive.

At the most basic level, your MATLAB plots should have proper formatting. The default MATLAB plot formatting isn't up to snuff with the standards of scientific publications or presentation of your work professionally. Luckily, MATLAB has many tools to improve the presentation of figures.

MATLAB has the ability for plots to utilize LaTeX formatting for figures. We highly recommend to enable this option for all plots as it allows for much better formatting on figures. This can be done with the following commands:

```
1 set(groot, 'defaultAxesTickLabelInterpreter','latex');
```

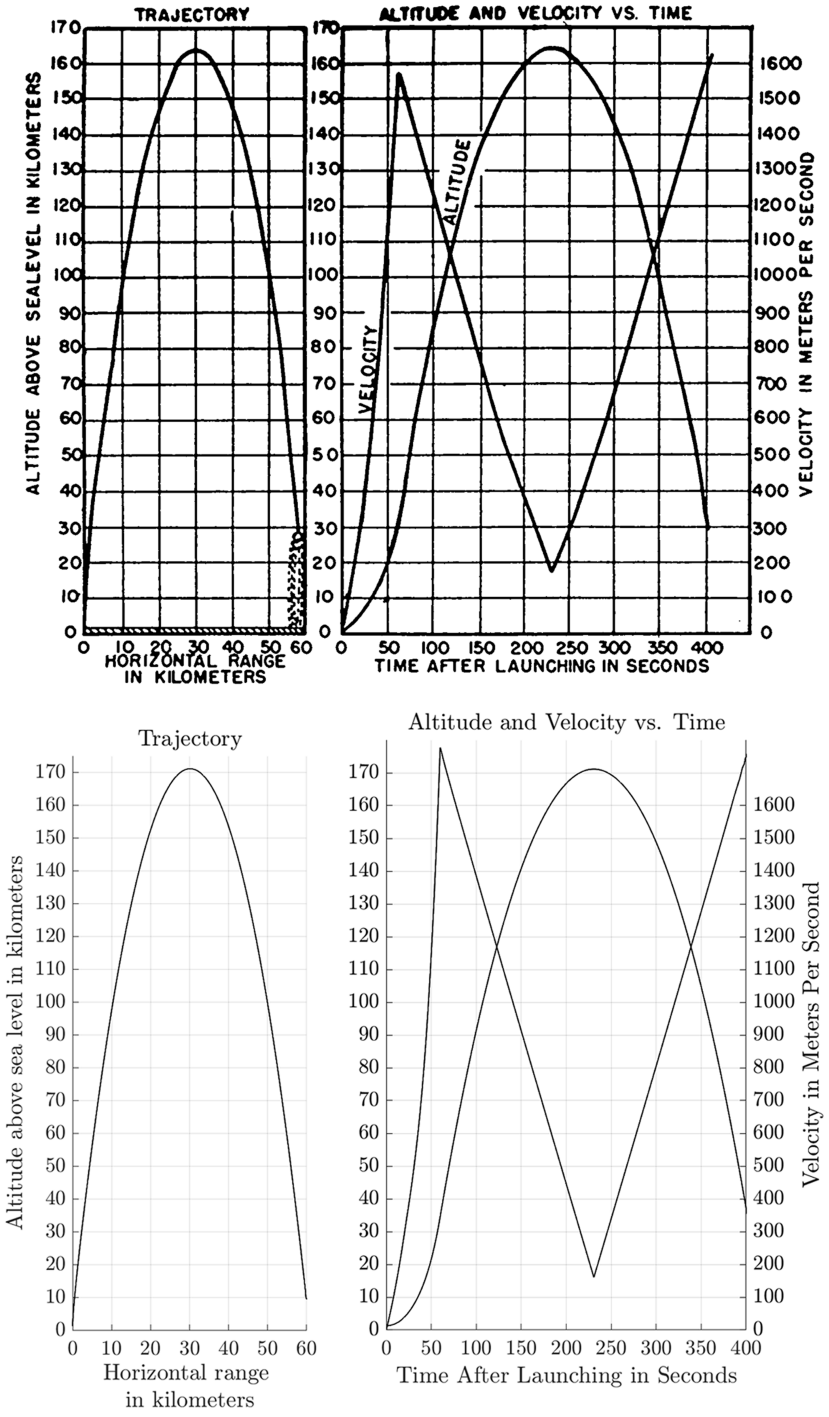


Figure 7.1: Comparison of Real V-2 Rocket Launch Data against 6-DoF Modeling

```
set(groot, 'defaultLegendInterpreter','latex');
set(groot, 'defaultTextInterpreter', 'latex')
```

7.4.1 Conventions for Plots

There are several conventions for plots which are generally followed in engineering practices. These will change depending on where your figures are to be published, but for our purposes, we use the following conventions:

1. All plots should have clear axis labels. Units are typically included in brackets as $[m]$, in example of a length. Units should be SI base units wherever possible.
2. Be knowledgeable of the use of color. Overuse of color may distract the viewer from the information being presented. Additionally, if your content is to be presented to a large audience, be mindful of those with color vision deficiencies. If your content may be printed, plots should be easily readable in black and white. Using different markers and line types is a good strategy.
3. Always include a legend when plotting multiple data on a single graph.
4. When the magnitudes of data are vastly different, a logarithmic axis or two y-axes are best practice.

7.4.2 Changing Default MATLAB Plotting

If you want to have these changes to figures universally applied in MATLAB, you can create a file in MATLAB called 'startup.m' that is run when MATLAB is opened. This file must exist in the user path for MATLAB. To find the user path for MATLAB, the command `userpath` can be typed into the command line. This will output something like:
`C:\Users\YourName\OneDrive\Documents\MATLAB`

The code in Script 7.4.2 is the d efault plotting used in this document, which should be a good baseline for professional plots.

Code Listing 7.2: Startup File for Plotting

```
% set linewidth
```

```
set(groot,'defaultLineLineWidth',1.5)

% set the interpreter to latex
set(groot, 'defaultAxesTickLabelInterpreter','latex');
set(groot, 'defaultLegendInterpreter','latex');
set(groot, 'defaultTextInterpreter', 'latex');

%change font sizes
set(groot, 'defaultLegendFontSize', 12)
set(groot, 'defaultTextFontSize', 12)
set(groot, 'defaultAxesFontSize', 10)

% grid on and box off
set(groot, 'defaultLegendBox', 'off');
set(groot,'defaultAxesXGrid','on');
set(groot,'defaultAxesYGrid','on');
```

And with that, we have completed the first Volume of the Flight Dynamics Bible. Additional information and code can be found in the Appendix. If you have any questions, concerns, or suggestions, feel free to reach out to Hudson Reynolds on Slack or via email at Hudsonj.reynolds@gmail.com.

CHAPTER 8

Appendix

8.1 Proofs and Derivations

8.1.1 Buckingham Pi

Variable	Dimension
ρ	$\frac{M}{L^3}$
V	$\frac{L}{T}$
S	L
μ	$\frac{M}{LT}$
a	$\frac{L}{T}$
α	$-$

For Buckingham Pi Theorem, we want to select repeating variables that span the base dimensions and do not want to select two variables that have the same dimensions. Somewhat arbitrarily, we select p, V, and S as the 3 repeating variables.

With each Π group, we solve by equating the exponents of the expressions to zero. For example:

$$\Pi_1 = \rho^a V^b S^c \mu = M^0 L^0 T^0$$

Plugging in the dimension of each of these:

$$\Pi_1 = \left[\frac{M}{L^3}\right]^a \left[\frac{L}{T}\right]^b [L]^c \left[\frac{M}{LT}\right] = M^0 L^0 T^0$$

This yields a system of equations with three equations:

$$M : a + 1 = 0$$
$$L : -3a + b + c - 1 = 0$$
$$T : -b - 1 = 0$$

Solving this system gives $a = b = c = -1$. Plugging in these values for the coefficients, we arrive at:

$$\Pi_1 = \frac{\mu}{\rho V S}$$

Which is $\frac{1}{Re}$. The same analysis is done for the rest of the Π groups:

$$\Pi_1 = \frac{\rho V S}{\mu}$$
$$\Pi_2 = \frac{V}{a}$$
$$\Pi_3 = \frac{F}{\rho V^2 S}$$
$$\Pi_4 = \alpha$$

8.1.2 Equivalence of Newtonian and Lagrangian Mechanics

Proof. We start with Lagrange's equations in rectilinear coordinates $x_i = x, y, z$:

$$\frac{\partial L}{\partial x_i} - \frac{d}{dt}\frac{\partial L}{\partial \dot{x}_i} = 0$$

We know that the Lagrangian is given as $L = T - U$, so we can express the Euler-Lagrange equations as:

$$\frac{\partial (T - U)}{\partial x_i} - \frac{d}{dt}\frac{\partial (T - U)}{\partial \dot{x}_i} = 0$$

We know that the translational kinetic energy is a function of the velocity only, so $\frac{\partial T}{\partial x_i} = 0$. A conservative potential is a function of the position only, so $\frac{\partial U}{\partial \dot{x}_i} = 0$. This simplifies the Euler-Lagrange Equation to:

$$-\frac{\partial U}{\partial x_i} = \frac{d}{dt}\frac{\partial T}{\partial \dot{x}_i}$$

For a conservative system, $-\frac{\partial U}{\partial x_i} = F_i$, because the force is the gradient of the potential function.

We know the kinetic energy is given as $\frac{1}{2}m\dot{x}^2$. The derivative with respect to the velocity $\dot{x}$ is $m\dot{x}_i$. Taking the time derivative, we have $\frac{d}{dt}(m\dot{x}_i) = m\ddot{x}_i$ in the case of constant mass (or more generally, $\frac{d}{dt}(m\dot{x}_i) = \dot{p}_i$).

So, we have $F = m\ddot{x}_i$.

□

Corollary 2.2. *From the proof, we can also see that the validity is dependent on the choice of rectangular coordinates.*

It can further be shown that the Newtonian derivation must be performed in an inertial frame.

8.2 Code

This section contains lengthier segments of example code from either the 6-DoF itself or a proof/example relating to it. Refer to the section numbers to find the appropriate implementation of each code segment.

8.2.1 Lorenz Attractor RK4 Example

Code Listing 8.1: Lorenz Attractor

```
1 %clear everything out
2 clear; close all; clc
3
4 % change whether to output to video file or not
5 output = 0;
6
7 %initialize the parameters of the lorenz attractor
8 sigma = 10;
9 rho = 28;
10 beta = 8/3;
11
```

```
%put these parameters into a vector called beta
beta = [sigma, rho, beta];

% initialize the timestep
dt = 1e-3;

% create a timestep to integrate this over. Read this as timespan from 0 to
% 20 with a step of dt between each step
tspan = 0:dt:20;

%set the options for ode45. Because the Lorenz attractor is a chaotic
%system set this to have high accuracy
options = odeset('RelTol',1e-12, 'AbsTol',1e-12*ones(1,3));


for j = 1:3:10
    % set a delta for each particle to see this diverge over time
    delta = rand()*(1e-2);

    %set the initial vector for each particle
    x0 = [0+delta;1+delta;20+delta];

    %run ode45. Use a function handle to pass through the time and the x
    %and pass through the beta value seperately as a constant.
    [t,x(:,j:j+2)] = ode45(@(t,x)lorenz(t,x,beta), tspan, x0, options);

end

%plot the solution
figure(1)
hold on
plot3(x(:,1),x(:,2),x(:,3));
plot3(x(:,4),x(:,5),x(:,6));
```

```
plot3(x(:,7),x(:,8),x(:,9));

xl = xlim;
yl = ylim;
zl = zlim;

lims = [xl,yl,zl];

%draw the solution in an animation
LorenzAnimation(x, lims, tspan, t, output)

%% Lorenz
% Differential equation for the Lorenz Attraction
% inputs:
% t - timespan [s]
% x - state vector
% outputs:
% dx - derivative of state vector
function dx = lorenz(t, x, beta)

dx = [
beta(1)*(x(2)-x(1));
x(1)*(beta(2)-x(3))-x(2);
x(1)*x(2) - beta(3)*x(3);
];

end
```

We also have a script to display the results of this in a nice animation:

Code Listing 8.2: Lorenz Animation

```
%% Lorenz Animation
% Draw the Lorenz Attractor
% inputs:
% x - output vector
% lims - bounds on the camera, xl,yl,zl
% tspan - time span of values
% t - time value
% output - turn on / off video output
% outputs:
```

```
% video output

function[] = LorenzAnimation(x, lims, tspan, t, output)

writerObj= VideoWriter('LorenzAnimation', 'MPEG-4');
writerObj.FrameRate = 30;
writerObj.Quality = 85;
open(writerObj)

lineNumber = size(x, 2)/3;

for i = 1:lineNumber
    colors(i, :) = [0, .5, .5] + ones(1,3)*(0.5)*i /
   lineNumber;
end

figure(3)
figure('units','pixels','position',[0 0 1920 1080])

curve = animatedline('LineWidth', 1.5, 'Color',colors(i,:)
   , MaximumNumPoints= 400);
curve1 = animatedline('LineWidth', 1.5, 'Color',colors
   (2,:), MaximumNumPoints= 400);
curve2 = animatedline('LineWidth', 1.5, 'Color', colors
   (3,:), MaximumNumPoints= 400);
curve3 = animatedline('LineWidth', 1.5, 'Color', colors
   (4,:), MaximumNumPoints= 400);



set(gca,'XColor', 'none','YColor','none', 'ZColor', 'none'
   )

set(gca, 'color', 'none');
axis equal
g = gcf;

%g.WindowState = 'maximized';
set(gcf,'Color','k');
speed = 5;
```

```
44
45 xlim(lims(1:2));
46 ylim(lims(3:4));
47 zlim(lims(5:6));
48 view(43,24);
49
50
51 for i=1:speed:length(tspan)
52     camorbit(1/20, 0)
53     camzoom(1 + (1e-4))
54     addpoints(curve, x(i,1), x(i,2), x(i,3));
55     addpoints(curve1, x(i,4), x(i,5), x(i,6));
56     addpoints(curve2, x(i,7), x(i,8), x(i,9));
57     addpoints(curve3, x(i,10), x(i,11), x(i,12));
58
59
60
61     title(sprintf("Trajectory at time %.2f s", t(i)));
62     drawnow;
63     %pause(dt / playbackSpeed);
64
65     % if output == 1
66     %     frame = getframe(gcf);
67     %     img =  frame2im(frame);
68     %     [img,cmap] = rgb2ind(img,256);
69     %     if i == 1
70     %         imwrite(img,cmap,'TrajAnimation.gif','gif','
   LoopCount',Inf, 'DelayTime', 1/60);
71     %     else
72     %         imwrite(img,cmap,'TrajAnimation.gif','gif','
   WriteMode','append', 'DelayTime', 1/60);
73     %     end
74     % end
75
76     if output == 1
77         frame = getframe(gcf);
78         writeVideo(writerObj,frame)
79     end
80
81 end
```

```
82
83 close(writerObj)
84 disp('Video File Written')
```

8.2.2 Dzhanibekov Effect Model

Code Listing 8.3: Dzhanibekov Main

```
1 % PSP FLIGHT DYNAMICS:
2 %
3 % Title: DzhanibekovMain
4 % Author: Hudson Reynolds - Created: 2/24/2024
5 % Last Modified: 2-24-2025
6 %
7 % Description: This is the overarching function that runs
     a 6-DoF model of
8 % the Dzhanibekov effect, calling all neccesary functions
     to run the
9 % simulation. The overarching structure uses ODE45.
10 %
11 % Inputs: N/A
12 %
13 % Outputs:
14 % Graph and value outputs. See subfunctions for specific
     outputs
15
16 %% Initialization:
17 % clear the console and figures before running the code:
18 clear
19 clc
20 close all
21
22 %% Simulation Settings:
23 % create a time array to span the simulation time.
24
25 % set the simulation time parameters
26 dt = 0.025;
27 time = 17.95;
28 tspan = 0:dt:time;
29
```

```
% initialize the state vector
% position (x,y,z)
pos = [0;0;0];
% velocity (xdot,ydot,zdot)
vel = [0;0;0];
% initial angle(z angle, y angle, x angle)
angleVector = [0;0;0];
% initial rotation rate(x rate, y rate, z rate)
omega = [0.02;0.02;pi];
%initalize the quaternion based on the euler angle input:
quatVector = eul2quat(angleVector.', "ZYX").';
% initial state vector
Init = [pos;vel;omega;quatVector];


% run the RK4:
[timeArray, out] = ode45(@(time,input)
   DzhanibekovIntegrator(time,input), tspan, Init);

% parse rk4 outputs:
posArray = [out(:,1), out(:,2), out(:,3)];
velArray = [out(:,4), out(:,5), out(:,6)];
omega = [out(:,7), out(:,8), out(:,9)];
quatArray = [out(:,10), out(:,11), out(:,12), out(:,13)];


%% Plotting:
% Euler Parameters:
figure(1)
plot(timeArray, quatArray);
xlim([0,time]);
title("Euler Parameters")
xlabel("Time (s)")
ylabel("Euler Parameters")
legend('q0', 'q1', 'q2', 'q3');

% run the rotation visualizer script
playbackSpeed = 1;
quatArray = quatArray';
posArray = posArray';
```

```
69 RotationsVisualizer(posArray, quatArray, timeArray, time,
       dt, playbackSpeed, 1);
```

Code Listing 8.4: Dzhanibekov Integrator

```
1  function [out] = DzhanibekovIntegrator(time, input)
2  % PSP FLIGHT DYNAMICS:
3  %
4  % Title: RK4Integrator
5  % Author: Hudson Reynolds - Created: 2/24/2025
6  %
7  % Description: This is the integration function to be used
       in ode45. This
8  % computes all funciton derivatives and differential
       equations for the
9  % translational and rotational dynamics.
10 %
11 % Inputs:
12 % time - current simulation time [s]
13 % input - Array of position, velocity, rotational velocity
       , and quaternions
14 %          [m|m/s|rad/s|unitless]
15 %
16 % Outputs:
17 % out = derivative of state vector [m/s|m/s^2|rad/s^2|
       unitless^2]
18
19 pos = [input(1);input(2);input(3)];
20
21 vel = [input(4);input(5);input(6)];
22
23 omega = [input(7); input(8); input(9)];
24
25 quat = [input(10); input(11); input(12); input(13)];
26
27 %bodyVectorEarth = RotationMatrix(bodyVector, quat, 1); %
       Body vector in inertial frame
28
29 accel = zeros(3,1);
30
31
```

```
%% Moments:
% just putting in values based on a cylinder moment of
   inertia for now.
Jxx = 0.09;
Jyy = 0.01;
Jzz = 0.03;

J = [Jxx,0,0;0,Jyy,0;0,0,Jzz];

momentVector = zeros(3,1);
% use euler equations to find the final moments:

wx = omega(1);
wy = omega(2);
wz = omega(3);

alpha(1) = (Jyy-Jzz)/(Jxx)*wy*wz;
alpha(2)= (Jzz-Jxx)/(Jyy)*wx*wz;
alpha(3)= (Jxx-Jyy)/(Jzz)*wx*wy;

alpha = alpha';

B = [0, -wx, -wy, -wz;
     wx, 0, wz, -wy;
     wy, -wz, 0, wx;
     wz, wy, -wx, 0];

quatRates = 0.5 * B * quat;

out = [vel;accel;alpha;quatRates];

end
```

Code Listing 8.5: Dzhanibekov Rotation Visualizer

```
function RotationsVisualizer(posArray, quatArray,
   timeArray, endTime, dt, playbackSpeed, output)
% PSP FLIGHT DYNAMICS:
%
% Title: RotationsVisualizer
% Author: Hudson Reynolds - Created: 7/12/2024
```

```
%
% Description: This function takes the euler angle and position outputs of
% the rocket and graphs them in 3D to get a better grasp of the rocket
% orientation
%
% Inputs:
% angleArray = array of all the angles generated by the 6 DoF
% posArray = array of all the positions generated by the 6 DoF
% timeArray = array of time values
% endTime = ending time of animation
% dt = time step
% playbackSpeed = speed up factor
% output = 0 -> no output, 1 -> output GIF file
%
% Outputs:
% figure output

% playback speed rate
playbackSpeed = 2;

posArrayTrans = transpose(posArray);

figure(6)

qs = quaternion([45,0,0],'eulerd','ZYX','frame');
qf = quaternion([-45,0,0],'eulerd','ZYX','frame');

ps = [0 0 0];
pf = [0 0 0];

patch = poseplot(qs,ps,'ENU', MeshFileName="Model.stl", ScaleFactor= 1, PatchFaceColor='r');
%xlim([0,1000]);
%ylim([0,max(posArray(2,:))]);
%zlim([0,max(posArray(1,:))]);
view(45,25);
```

```
axis square
%camroll(113)


xlabel("x")
ylabel("y")
zlabel("z");


for i = 1:5:length(posArray)
    q = quaternion(quatArray(:,i)');
    %position = [posArrayTrans(i,3),posArrayTrans(i,2),
   posArrayTrans(i,1)];
    position = posArrayTrans(i,:);
    set(patch,Orientation=q,Position=position);

    title(sprintf('Orientation at time %.1f s',timeArray(i
   )))
    drawnow
    pause(1/100);

    if output == 1
        frame = getframe(gcf);
        img =  frame2im(frame);
        [img,cmap] = rgb2ind(img,256);
        if i == 1
            imwrite(img,cmap,'RotationAnimation.gif','gif'
   ,'LoopCount',Inf,'DelayTime',1/24);
        else
            imwrite(img,cmap,'RotationAnimation.gif','gif'
   ,'WriteMode','append','DelayTime',1/24);
        end
    end
end
```

8.2.3 Numerical Integration Error Examples

Code Listing 8.6: Numerical Integration Error Comparison

```
% Numerical Integrator Error Showcase:
close all; clear; clc;

% initial condition, y(0) = 0
y = 0;
yRK2 = y;
yRK4 = y;

% timespan over which to integrate:
dt = 0.5;
tEnd = 3;
t = 0:dt:tEnd;

%integration loop:
for i=1:length(t)-1
    % numerically integrate with explicit euler
    y(i+1) = y(i) + YDot(t(i),y(i)) * dt;
    % numerically integrate with RK2
    yRK2(i+1) = rk2(@(t,y)YDot(t(i),y), dt, t, yRK2(i));
    % numerically integrate with RK4
    yRK4(i+1) = rk4(@(t,y)YDot(t(i),y), dt, t, yRK4(i));
end

% tspan for the original function:
tspanCurve = linspace(0,tEnd, 100);

%figure plotting:
hfig = figure;  % save the figure handle in a variable
fname = 'Numerical Integrator Comparison Figure';
picturewidth = 20;
hw_ratio = 0.75;


plot(t, y, '.', LineStyle='-', MarkerSize= 12)
hold on
plot(t, yRK2, '.', LineStyle='-', MarkerSize= 12)
plot(t, yRK4, '.', LineStyle='-', MarkerSize= 12)
plot(tspanCurve, tspanCurve.^2, LineWidth=1.5)

legend('Explicit Euler', 'RK2', 'RK4', '$y=t^2$', '
```

```
   Location', 'northwest')

%title('Explicit Euler Numerical Approximation')
xlabel('$t$')
ylabel('$y(t)$')
title('Comparison of Runge-Kutta Integration Orders')

grid on
axis tight

set(hfig,'Units','centimeters','Position',[3 3
   picturewidth hw_ratio*picturewidth])
pos = get(hfig,'Position');
set(hfig,'PaperPositionMode','Auto','PaperUnits','
   centimeters','PaperSize',[pos(3), pos(4)])
print(hfig,fname,'-dpng','-r600')

% functions

%euler
function[out] = YDot(t,Y)
out = 2 * t;
end

%rk2
function out = rk2(fun, dt, tIn, xIn)
    f1 = fun(tIn,xIn);
    f2 = fun(tIn + dt/2, xIn + dt .* f1);

    out = xIn + (dt / 2)*(f1 + f2);
end

%rk4
function out = rk4(fun, dt, tIn, xIn)
    f1 = fun(tIn,xIn);
    f2 = fun(tIn + dt/2, xIn + (dt/2) .* f1);
    f3 = fun(tIn + dt/2, xIn + (dt/2) .* f2);
    f4 = fun(tIn + dt, xIn + dt*f3);

    out = xIn + (dt / 6)*(f1 + 2*f2 + 2*f3+f4);
```

```
78 end
```

Code Listing 8.7: Numerical Integration of Periodic Function

```
1 % Numerical Integrator Error Showcase:
2 close all; clear; clc;
3
4 % initial condition, y(0) = 0
5 y = 0;
6 yRK2 = y;
7 yRK4 = y;
8
9 % timespan over which to integrate:
10 dt = 0.5;
11 tEnd = 2;
12 t = 0:dt:tEnd;
13
14 %integration loop:
15 for i=1:length(t)-1
16     % numerically integrate with explicit euler
17     y(i+1) = y(i) + YDot(t(i),y(i)) * dt;
18     % numerically integrate with RK2
19     yRK2(i+1) = rk2(@(t,y)YDot(t(i),y), dt, t, yRK2(i));
20     % numerically integrate with RK4
21     yRK4(i+1) = rk4(@(t,y)YDot(t(i),y), dt, t, yRK4(i));
22 end
23
24 % tspan for the original function:
25 tspanCurve = linspace(0,tEnd, 1000);
26
27 %figure plotting:
28 hfig = figure;  % save the figure handle in a variable
29 fname = 'Numerical Integrator Nyquist 1';
30
31 picturewidth = 20; % set this parameter and keep it
      forever
32 hw_ratio = 0.75; % feel free to play with this ratio
33 set(findall(hfig,'-property','FontSize'),'FontSize',18) %
      adjust fontsize to your document
34
35 plot(t, y, '.', LineStyle='-', MarkerSize= 12)
```

```
36 hold on
37 plot(t, yRK2, '.', LineStyle='-', MarkerSize= 12)
38 plot(t, yRK4, '.', LineStyle='-', MarkerSize= 12)
39 plot(tspanCurve, 1/(4*pi)*sin(4*pi*tspanCurve), LineWidth
   =1)
40
41 legend('Explicit Euler', 'RK2', 'RK4', '$y=\frac{1}{4\pi}
   sin(4\pi t)$', 'Location', 'northwest')
42
43 %title('Explicit Euler Numerical Approximation')
44 xlabel('$t$')
45 ylabel('$y(t)$')
46
47 grid on
48 axis tight
49
50 set(findall(hfig,'-property','Box'),'Box','off') %
   optional
51 set(findall(hfig,'-property','Interpreter'),'Interpreter',
   'latex')
52 set(findall(hfig,'-property','TickLabelInterpreter'),'
   TickLabelInterpreter','latex')
53 set(hfig,'Units','centimeters','Position',[3 3
   picturewidth hw_ratio*picturewidth])
54 pos = get(hfig,'Position');
55 set(hfig,'PaperPositionMode','Auto','PaperUnits','
   centimeters','PaperSize',[pos(3), pos(4)])
56 print(hfig,fname,'-dpng','-r400')
57
58 % Part II:
59
60 dtList = [0.4,0.26,0.1];
61
62 yRK42 = zeros(4, 32);
63 %tList = zeros(4,32);
64
65 forLength = tEnd ./ dtList;
66
67 tList1 = 0:dtList(1):tEnd;
68 tList2 = 0:dtList(2):tEnd;
```

```
tList3 = 0:dtList(3):tEnd;

int1 = rk4Integrate(dtList(1), tList1);
int2 = rk4Integrate(dtList(2), tList2);
int3 = rk4Integrate(dtList(3), tList3);

%figure plotting:
hfig = figure;  % save the figure handle in a variable
fname = 'Numerical Integrator Nyquist 2';

picturewidth = 20; % set this parameter and keep it forever
hw_ratio = 0.9; % feel free to play with this ratio
set(findall(hfig,'-property','FontSize'),'FontSize',18) % adjust fontsize to your document

plot(tList1, int1, '.', LineStyle='-', MarkerSize= 12, LineWidth=1)
hold on
plot(tList2, int2, '.', LineStyle='-', MarkerSize= 12, LineWidth=1)
plot(tList3, int3, '.', LineStyle='-', MarkerSize= 12, LineWidth=1)

plot(tspanCurve, 1/(4*pi)*sin(4*pi*tspanCurve), LineWidth=1.5)

legend('$\Delta t=0.4s$', '$\Delta t=0.26s$', '$\Delta t=0.1s$','$y=\frac{1}{4\pi}sin(4\pi t)$', 'Location', 'northwest')

%title('Explicit Euler Numerical Approximation')
xlabel('$t$')
ylabel('$y(t)$')

grid on
axis tight

set(findall(hfig,'-property','Box'),'Box','off') % optional
```

```
set(findall(hfig,'-property','Interpreter'),'Interpreter',
   'latex')
set(findall(hfig,'-property','TickLabelInterpreter'),'
   TickLabelInterpreter','latex')
set(hfig,'Units','centimeters','Position',[3 3
   picturewidth hw_ratio*picturewidth])
pos = get(hfig,'Position');
set(hfig,'PaperPositionMode','Auto','PaperUnits','
   centimeters','PaperSize',[pos(3), pos(4)])
print(hfig,fname,'-dpng','-r400')


% functions

%euler
function[out] = YDot(t,Y)
out = cos(2*pi/0.5*t);
end

%rk2
function out = rk2(fun, dt, tIn, xIn)
    f1 = fun(tIn,xIn);
    f2 = fun(tIn + dt/2, xIn + dt .* f1);

    out = xIn + (dt / 2)*(f1 + f2);
end

%rk4
function out = rk4(fun, dt, tIn, xIn)
    f1 = fun(tIn,xIn);
    f2 = fun(tIn + dt/2, xIn + (dt/2) .* f1);
    f3 = fun(tIn + dt/2, xIn + (dt/2) .* f2);
    f4 = fun(tIn + dt, xIn + dt*f3);

    out = xIn + (dt / 6)*(f1 + 2*f2 + 2*f3+f4);
end

% rk4 integrator
function out = rk4Integrate(dt, t)
Y = 0;
```

```
136
137     for i=1:length(t)-1
138         % numerically integrate with RK4
139         Y(i+1) = rk4(@(t,y)YDot(t(i),y), dt, t, Y(i));
140     end
141     out = Y;
142 end
```

8.2.4 6-DoF Code

The code for the 6-DoF can be found at [14] or accessed via GitHub with the QR code. This GitHub also contains the most up-to-date version of this .pdf and .tex document:

Special Terms

action

The time integral of energy is known as the action, which is often denoted S. 37, 52, 58

angle of attack

The angle of offset of the freestream velocity with the $\hat{X}/\hat{b}_1$, or longitudinal axis, of the vehicle. This is often denoted with the Greek letter α. 14, 16, 71, 72, 74, 77

axisymmetric

An object in which there is rotational symmetry about one or more of the axes. 76

bandwidth

The bandwidth of a signal is the difference between its upper and lower frequencies. In our context of numerical integration, this bandwidth refers to twice the highest frequency of a function. 91, 97

beat frequency

A beat frequency is the result of constructive and destructive interference between two waves of a similar frequency. The similarity of the frequencies creates variations in the amplitude of the resultant wave. 91

BKE

Basic Kinematic Equation. 16, 23, 27, 40, 46, 48, 49

body frame

A reference frame that is fixed to the body of an object and follows the translations and rotations of the vehicle. 12–14, 18–20, 22–24, 27, 57–62, 64, 99, 103, 104

Buckingham Pi Theorem

The Buckingham Pi Theorem is a method to non-dimensionalize parameters of a system. In a system with N parameters and K physical dimensions, it is possible to express $P = N - K$ dimensionless groups. 71, 72, 110

CFD

Computational Fluid Dynamics. 77, 78

conservative

A system or potential which is independent of the path. A closed integral in a conservative field is 0. 37, 49, 51, 111, 112

dimensional analysis

The process of determining information from a physical equation by considering the dimensions, or units, of quantities in the equation. 71

DoF

Degree of Freedom. vii–x, 1, 3–7, 11, 13–16, 18, 19, 29, 30, 32, 41–43, 47, 72, 73, 75, 92, 99, 104, 106, 112, 129

drag coefficient

A non dimensional parameter that describes the amount of drag acting on an object. This non-dimensionalized value is given by $\frac{F_D}{q_\infty S}$, where q_∞ is the dynamic pressure, $\frac{1}{2}\rho_\infty V_\infty^2$. 7, 77

Dzhanibekov Effect

A phenomenon in which rotation about an intermediate axis results in rotation about another axis. This phenomenon occurs with any body where the three PMOIs are not equal. It is easily seen with a tennis racket or a cell phone. 63, 101, *see* PMOI

EOM

Equation of Motion. 8, 28, 29, 32, 33, 38, 40, 46–49, 51, 53, 54, 67

Euler angles

A group of three scalar values used to describe the orientation of a body in space. 56–60, 64, 67, 99, 100

Euler rates

A way to describe rotational rates with orientation for 3-dimensional rotations, they are denoted by the dot of the corresponding Euler angle: $(\dot{\psi}, \dot{\theta}, \dot{\phi})$. 57–60

Euler's equation

The fundemental equation in the calculus of variations describing the maximum or minimum conditions of a functional. 36, 50, *see* functional

Euler-Lagrange Equation

The fundemental equations describing the solutions that give the extrema of the action in a Lagrangian system. 37, 42, 44, 45, 111, *see* action

freestream

A freestream quantity is one that is far away from the body, such that the influences of the body are negligible. For example, freestream velocity is the velocity that would be seen outside the influence of the body at the same conditions. 7, 16, 30, 73, 74, 76, 104, 132

function handle

A pointer to an instance of a function in MATLAB. A function handle essentially stores a function call in a variable in MATLAB. 85, 86, 89

functional

A quantity that maps a function onto the real number line. Often, this is called a 'function of a function'. 34, 37

generalized momenta

A generalized coordinate that is defined as $\frac{\partial L}{\partial \dot{q}_i}$. 48, 49

generalized velocity

A free direction of movement for a vehicle. These can be translational or rotational directions. 44

geodesic

The shortest path between two points in an arbitrarily curved space. 29, 33

holonomic constraint

A constraint on a Lagrangian system which is a function of only the position of the particles. 40–43, *see* Lagrangian

holonomic system

A system that is entirely composed of holonomic constraints. 43, *see* holonomic constraint

inertia tensor

A matrix that describes the moment of inertia along each of the axes of a body. This matrix, often denoted I, transforms the angular velocity, ω into an angular momentum, H. 22, 23, *see* moment of inertia

inertial frame

A reference frame that is static and experiences no accelerations. We use the Earth's surface as the reference frame for our rocket. 11–13, 16, 19, 20, 27, 30, 40, 45, 46, 57–62, 64, 99, 100

Lagrangian

A description of the energy of a system given by $L \equiv T - U$. Denoted with the letter L. 32, 33, 37–40, 44, 45, 47–52, 55, 67, 94, 111

moment of inertia

A measure of its resistance to rotational acceleration. It is in essence, the rotational analogue of mass. Often denoted with the letter I. 20–23

neighborhood function

A small function added to another function in the calculus of variations. Analogous to a differential element in single variate calculus. 34

non-holonomic constraint

A constraint that depends on more than the position of the particles in the system. 44, *see* holonomic constraint

numerical integration

The process of making a numerical approximation of a differential equation through discretization of the independent variable (usually time for most physical phenomenon). 8, 28, 30, 32, 49, 59, 62, 81, 82, 86, 89–97, 100, 105

Nyquist rate

The rate at which a signal must be sampled to be free of distortions and aliasing. This rate is typically twice the highest frequency present in the signal. 91

orthonormal

A set of basis vectors that are mutually orthogonal to each other in a vector space and have unit length. We mostly refer to orthonormal vectors in $\mathbb{R}^3$ space. 11

parallel axis theorem

A theorem relating the moment of inertia and product of inertia about different rotation points. 22, 23, *see* moment of inertia

PMOI

principle moment of inertia. 22, 24

principle axes

The axes defined such that the products of inertia for a body are zero. These axes are the values that solve the eigenvalue problem $I\vec{\omega} = I^*\vec{\omega}$. 22, 23

principle of least action

A form of Hamilton's Principle that states that the action of a system should be minimized in a physical mechanical system. 37, *see* action

quaternion

An extension of the complex numbers with elements j and k added. The common form is expressed $q_1 + q_2 i + q_3 j + q_4 k$ (our notation) or $q_1 i + q_2 j + q_3 k + q_4$. The algebra of quaternions is generally denoted $\mathbb{H}$. 56, 60–64, 66, 67, 100, 102, 105

rail whip

A phenomenon where the flexibility of the launch rail causes a large moment to be exerted on the rocket during launch. 101

reference area

An area that is used to define a characteristic area of an object in aerodynamics. This is used to normalize quantities such as lift and drag. 7

RHR

right hand rule. 11, 18, 75

rigid body

A rigid body is a system of particles whose positions with respect to each other is fixed. Rigid body dynamics are the basis of our analysis of rotational dynamics. 20, 40, 43, 49–51, 105

roundoff error

Error in numerical integration that comes from inaccuracies in the floating point precision of computers. This is worsened by a larger number of timesteps, as this error will compound. 83, 84

state vector

A vector containing information about the position and velocity of a body at a given time. This vector can also include more information, such as the angular velocity and quaternion. Generally, this vector includes all quantities needed to be integrated to find the new state. 5, 12, 14, 19, 63, 64, 66, 67, 86, 89, 99, 101, 105, *see* quaternion

stiff

A differential equation that is hard to numerically integrate as a result of instability when larger timesteps are used. Stiffness is a property of the differential equation itself, and not the solution. Stiff systems are not completely well defined, but generally follow from this definition. 92, 94

symplectic integrator

A class of numerical integration schemes that are energy preserving because they preserve certain geometric properties of a differential equation. 32, 48, 94

truncation error

Error in numerical integration that comes from the inaccuracies of the integration scheme in modeling the true behavior of the function. This error is related to the number of higher order terms ignored. 83, 92, 94

wind frame

A reference frame in reference to the direction of freestream velocity, denoted V_∞. 14, *see* freestream

References

[1] John Anderson. *Fundamentals Of Aerodynamics 6 Edition*. 6th ed. Mc-Graw Hill, 2017. 1154 pp. ISBN: 978-1-259-12991-9. URL: http://archive.org/details/fundamentals-of-aerodynamics-6-edition (visited on 02/23/2025).

[2] Daniel W. Baker and William Haynes. *Statics: Products of Inertia*. 2020. 419 pp. URL: https://engineeringstatics.org/products-of-inertia.html (visited on 02/25/2025).

[3] Marcelo Dasilva. *Chemical Equilibrium with Applications*. Glenn Research Center | NASA. Apr. 17, 2023. URL: https://www1.grc.nasa.gov/research-and-engineering/ceaweb/ (visited on 03/04/2025).

[4] Paul Dawkins. *Differential Equations*. Paul's Online Math Notes. June 26, 2023. URL: https://tutorial.math.lamar.edu/Classes/DE/DE.aspx (visited on 03/06/2025).

[5] Hugo Filmer. *Fluid Systems Bible*. URL: https://www.overleaf.com/project/67e4548eec2d615a2e90fce1 (visited on 04/12/2025).

[6] Gregory Gundersen. *Understanding Moments*. Understanding Moments. Apr. 11, 2020. URL: https://gregorygundersen.com/blog/2020/04/11/moments/ (visited on 02/23/2025).

[7] N. Jeremy Kasdin and Derek A. Paley. *Engineering dynamics: a comprehensive introduction*. Princeton, NJ: Princeton University Press, 2011. 1 p. ISBN: 978-0-691-13537-3. URL: https://students.aiu.edu/submissions/profiles/resources/onlineBook/g2B5P5_Engineering_Dynamics_A_Comprehensive_Introduction.pdf.

[8] Mathworks. *Choose an ODE Solver*. 2024. URL: https://www.mathworks.com/help/matlab/math/choose-an-ode-solver.html (visited on 03/08/2025).

[9] Mathworks. *eul2quat*. eul2quat. URL: https://www.mathworks.com/help/robotics/ref/eul2quat.html (visited on 03/31/2025).

[10] Mathworks. *quat2dcm*. URL: https://www.mathworks.com/help/aerotbx/ug/quat2dcm.html (visited on 03/31/2025).

[11] Mathworks. *Summary of ODE Options*. 2024. URL: https://www.mathworks.com/help/matlab/math/summary-of-ode-options.html (visited on 03/08/2025).

[12] Ashwin Narayan. *How to Integrate Quaternions*. Ashwin Narayan. Sept. 10, 2017. URL: https://www.ashwinnarayan.com/post/how-to-integrate-quaternions/ (visited on 03/31/2025).

[13] Eqab M Rabei, Tareq S Alhalholy, and A A Taani. "On Hamiltonian Formulation of Non-Conservative Systems". In: (2004), p. 10. URL: https://journals.tubitak.gov.tr/cgi/viewcontent.cgi?article=1763&context=physics (visited on 03/18/2025).

[14] Hudson Reynolds. *HudsonReynolds/Flight-Dynamics-Bible*. URL: https://github.com/HudsonReynolds/Flight-Dynamics-Bible (visited on 03/30/2025).

[15] Charles Rogers and David Cooper. *RasAero*. Rogers Aeroscience RASAero Aerodynamic Analysis and Flight Simulation Software. 2019. URL: https://www.rasaero.com/ (visited on 03/22/2025).

[16] Shane Ross. *Lagrange's Equations Intro, Generalized Coordinates, Constraints, Degrees of Freedom, Lecture 17*. URL: https://www.youtube.com/watch?v=_g9QD43P3v8&list=PLUeHTafWecAUl2DuWWdRU1MckJv7M5LEH&index=17 (visited on 03/22/2025).

[17] W. Shyy et al. "Recent progress in flapping wing aerodynamics and aeroelasticity". In: *Progress in Aerospace Sciences* 46.7 (Oct. 1, 2010), pp. 284–327. ISSN: 0376-0421. DOI: 10.1016/j.paerosci.2010.01.001. URL: https://www.sciencedirect.com/science/article/pii/S0376042110000023 (visited on 03/07/2025).

[18] Petra Svickova. *Air Density Changes Due to Weather*. BARANI. Mar. 1, 2020. URL: https://www.baranidesign.com/faq-articles/2020/2/20/air-density-changes-due-to-weather (visited on 02/23/2025).

[19] Stephen Thornton and Jerry Marion. *Classical Dynamics Of Particles And Systems Thornton*. 5th ed. URL: http://archive.org/details/ClassicalDynamicsOfParticlesAndSystemsThornton (visited on 03/18/2025).

[20] William Trench. *3.1: Euler's Method*. Mathematics LibreTexts. July 20, 2020. URL: https://math.libretexts.org/Courses/Community_College_of_Denver/MAT_2562_Differential_Equations_with_Linear_Algebra/03%3A_Numerical_Methods/3.01%3A_Euler's_Method (visited on 02/23/2025).

[21] *Variational Principles in Classical Mechanics (Cline)*. Physics LibreTexts. Nov. 11, 2017. URL: https://phys.libretexts.org/Bookshelves/Classical_Mechanics/Variational_Principles_in_Classical_Mechanics_(Cline) (visited on 03/18/2025).

[22] Veritasium. *The Closest We've Come to a Theory of Everything*. Oct. 29, 2024. URL: https://www.youtube.com/watch?v=Q10_srZ-pbs (visited on 03/18/2025).

[23] Carl Wassgren. *Buckingham Pi*. URL: https://engineering.purdue.edu/~wassgren/teaching/ME30800/NotesAndReading/DimensionalAnalysis_BuckinghamPi_MethodOfRepeatingVariables_Reading.pdf (visited on 03/16/2025).