Implementation of a Modular 3D Rigid Body Motion Simulator

A Thesis

Submitted to the Faculty of Fachhochschule Aachen and Centro de Ingeniería y Desarrollo Industrial

BY

Ricardo Vega Ayora

In Partial Fulfillment of the requirements for the degree of Master of Science in Mechatronics

Santiago de Querétaro, Qro, México, February 2018

Declaration

I hereby declare that this thesis work has been conducted entirely on my own accord.

The data, the software employed and the information used have all been utilized in complete agreement to the copyright rules of concerned establishments.

Any reproduction of this report or the data and research results contained in it, either electronically or in publishing, may only be performed with prior sanction of the University of applied Sciences, Centro de Ingeniería y Desarrollo Industrial (CIDESI) and myself, the author.

The work developed for this project uses MATLAB version R2017a, which is licensed to the Fachhochschule Aachen. MATLAB is a trademark of The MathWorks, Inc., 1 Apple Hill Drive, Natick, Massachusetts, 01760.

Ricardo Vega Ayora

Santiago de Queretaro, Qro., Mexico, February 2018

Acknowledgments

I would like to use this section to express my gratitude to the following parties, for without them, this work would not have been possible.

- My family for all their support, guidance and patience.
- Dr.-Ing. Markus Czupalla for allowing me to start this project under his supervision, for sharing his ideas and for his suggestions.
- Dr.-Ing. Bernd Dachwald for his support on understanding rotations and for providing books with helpful information about motion simulation.
- Dr. Salomón Miguel Ángel Jiménez Zapata for his help with linear algebra and for his great guidance on the writing of this document.
- Consejo Nacional de Ciencia y Tecnología (CONACYT) for the scholarship.
- Centro de Investigación y Desarrollo Industrial (CIDESI) and the FH Aachen for the opportunity of being part of a double degree programme in an international environment.
- My colleagues Osvaldo, Jaime, Luis, Israel, Gerardo and Adrián for being outstanding teammates during the master’s programme.
- Chizuru, an admirable and motivating hard-working person who is always looking to the future.
- Lucía for her help at the beginning of the programme.

Table Contents

List of figures

List of tables

Abbreviations, units and sign conventions

RBT	Rigid Body Simulation Toolbox
IRF	Inertial Reference Frame
BRF	Body Reference Frame
DCM	Direct Cosine Matrix
MKS	Meter-Kilogram-Second unit system
CPU	Central Processing Unit
FPS	Frames Per Second
CoM	Centre of Mass
DOF	Degrees Of Freedom
ML	MATLAB®
MW	The MathWorks®
OOP	Object Oriented Programming
CT	Computational Thinking
FEA	Finite Element Analysis
API	Application Programming Interface
MDC	Mass Distribution Calculation Module
SMOD	Simulation Module
AMOD	Animation Module
ODE	Ordinary Differential Equation
$[T_a^b]$	Transformation from vector space ***a*** to vector space ***b***
$[A]$	Matrix A
$[\omega^b]$	Skew matrix of the vector ω^b
v^a	Vector in vector space ***a***
v^e	Vector in the IRF
v^b	Vector in the BRF
F	Force vector in newton
M	Torque vector in newton per meter
ω	Angular velocity vector in radians per second
H	Angular momentum vector in joules times second
L	Linear momentum vector in joules times second
E_k	Kinetic energy in joules
r	Position vector in meters
v	Linear velocity vector in meters per second
$[I]$	Moment of inertia tensor in kilogram times meter squared
m	Mass in kilograms
$\boldsymbol{X_e}$	X axis of the IRF
$\boldsymbol{Y_e}$	Y axis of the IRF
$\boldsymbol{Z_e}$	Z axis of the IRF
$\boldsymbol{x_b}$	X axis of the BRF
$\boldsymbol{y_b}$	Y axis of the BRF
$\boldsymbol{z_b}$	Z axis of the BRF
ϕ	Angle about x in radians
θ	Angle about y in radians
ψ	Angle about z in radians
$[w \quad i \quad j \quad k]$	Quaternion with real part w

The right-hand rule determines the sign of the rotations. The rotation is positive in the direction in which the first finger closes about the thumb and negative in the opposite direction.

All units are in the MKS unless otherwise noted.

Abstract

This work presents the development of a toolbox for MATLAB® for the simulation of rigid body dynamics. The toolbox includes a module for the simulation of the motion and a module for visualisation, the modules can be used together or independently. The motion simulator uses Newton-Euler mechanics to simulate the 6-DoF motion of a single rigid body as well as Euler angles and quaternions to simulate the attitude of the rigid body. The visualisation module is capable of creating animations of translating and rotating geometries in a three dimensional space.

Resumen

Este trabajo presenta el desarrollo de un sistema de simulación de dinámica de cuerpos rígidos para MATLAB®. El sistema está compuesto por un módulo para la simulación y otro para visualización. Los módulos pueden usarse en combinación o de manera independiente. El simulador de movimiento usa mecánica de Newton-Euler para simular el movimiento de un cuerpo rígido con 6 grados de libertad, así como la orientación del cuerpo con cuaterniones y ángulos de Euler. El módulo de visualización permite crear animaciones de geometrías que rotan y se trasladan en un espacio tridimensional.

Kurzfassung

Diese Masterarbeit befasst sich mit der Entwicklung einer Toolbox für MATLAB® zur Simulation der Starrkörperdynamik. Die Toolbox enthält ein Modul zur Simulation der Bewegung und ein Modul zur Visualisierung, die Module können zusammen oder unabhängig voneinander verwendet werden. Der Bewegungssimulator simuliert mithilfe der Newton-Euler-Mechanik die 6-DoF-Bewegung eines einzelnen starren Körpers, sowie mithilfe von Euler-Winkeln und Quaternionen, um die Lage des starren Körpers zu simulieren. Das Visualisierungsmodul kann Animationen von sich translatorisch bewegenden und rotierenden Geometrien in einem dreidimensionalen Raum erzeugen.

Keywords: simulation, rigid body dynamics, modular, object oriented programming.

Chapter 1: Introduction

Throughout history, understanding rigid body dynamics has enabled human-kind to create tools that avoid, prevent, or take advantage of events that would otherwise be harmful or represent a waste of potential resources. The wheel and the emerging reusable rockets are just two examples of a countless amount of inventions born from the study of dynamics, and which have helped human-kind with its scientific and technological progress.

In modern science and engineering, phenomena such as those that occur in dynamics are represented with sets of equations and variables, which together are called mathematical models and their analysis is a very helpful tool in the process of understanding a phenomenon. For instance, the combination of Newton's Three Laws and Euler's equations of rotational motion facilitate the development of tools like the robotic arms in a vehicle production line, or the momentum wheels that satellites use in space to control their orientation.

Recently, scientists and engineers of multiple disciplines have accelerated their understanding of the laws of physics with the help of electronic computers. Using programming languages, algorithms for numerical methods can be programmed into computers to numerically evaluate mathematical models and thus analyse the phenomenon the models describe. This kind of evaluation is called computer simulation, and it has a wide variety of applications such as analysis, training, and research [1].

Within the scope of the simulation of rigid body dynamics, there is a diverse range of existing tools, both open-source and commercial. On the open-source side, for example, standalone tools such as MBDyn [2] and Robotran [3] provide well-documented platforms to simulate and visualise multi-body dynamics. MBDyn is a general purpose, command-line driven, multi-body dynamics analysis software, capable of performing the integrated simulation and analysis of nonlinear mechanical, aeroelastic, hydraulic, electric and control problems by numerical integration. MBDyn also has a special input syntax that users need to learn so that they are able to describe simulation experiments.

Robotran is a platform that approaches multi-body dynamics with computational algebra, that is, symbolic mathematics. The platform features a graphical interface with which users sketch multi-body systems and run 3D animations, a symbolic generator that takes the sketched systems and returns corresponding symbolic models, and interfaces to MATLAB® (ML), Python and C/C++, with which users can later numerically analyse the symbolic models. It is important to notice that the symbolic generator runs on an external server to which users must have access credentials.

Among the open-source options, there are also libraries and physics engines that allow developers to run physics simulations within their own applications. Three important instances are PyBullet [4], PROJECTCHRONO®, and the Open Dynamics Engine™ (ODE), which as their standalone counterparts, are well-documented platforms that include visualisation modules. The first example, PyBullet, is a physics simulation engine developed in Python that features support for collision detection and simulation of both 6-DoF rigid bodies and generalised coordinate multibody systems. Additionally, PyBullet integrates interfaces to 3D modelling and animation tools such as Maya® and Blender™.

Similar to PyBullet, although with some differences in the features and in the programming language, there is PROJECTCHRONO®, a community project led by the University of Wisconsin-Madison and the University of Parma-Italy. The project provides a multi-physics simulation engine developed in C/C++ that features support for multibody dynamics, finite element analysis (FEA), vehicle dynamics, parallel computing and collision detection, and integrated interfaces with which PROJECTCHRONO® is capable of sharing or taking over the simulation of external platforms developed in Python, ML, Simulink® and Solidworks®.

Lastly, ODE is a library originally developed by Russel Smith in C/C++ for the simulation of rigid body dynamics. ODE features a fast integrator (or solver), collision detection, and rigid bodies with arbitrary mass distribution and multiple joint types. Due to its popularity, the library now has bindings for different programming languages such as Python, Java and JavaScript.

Moreover in the open-source software, although special cases, there are QuIRK [5] and the Spatial Vector and Rigid-Body Dynamics Software (just Spatial from here on) [6]. They are special cases because, although being open-source, they are toolboxes for ML, which is proprietary software. QuIRK consists of a set of simulation and visualisation tools for multi-body dynamics that uses quaternions to represent rotations, and the Udwadia-Kalaba pseudoinverse method to construct equations of motion of constrained systems. Being developed for ML, QuIRK provides an interactive command line interface for constructing systems of rigid bodies and joint constraints.

Spatial is a suite of functions that implement the spatial vector arithmetic and dynamics algorithms found in [7]. The suite includes functions for the calculation of forward, hybrid and inverse dynamics, and in addition to that, it computes the momentum and both the potential and kinetic energies of the system. Moreover, it includes a robust visualisation tool that allows users to view an animation at any speed, from any angle, and at any magnification.

In contrast to the so far mentioned simulation tools, there are also the commercial ones such as the standalone application Adams™, developed by MSC Software®. The platform's functionality allows the study of moving parts and how loads and forces are distributed throughout mechanical systems. Plugins such as Adams Mechatronics and Adams Control can be incorporated to allow the use of control systems in the simulation. One last feature, and probably the most important one, is that Adams™ is free for students.

The rigid body simulation market also includes tools such as COMSOL's Multibody Dynamics Module and ANSYS® Rigid Body Dynamics, they are extensions for COMSOL's Structural Mechanics Module and for the ANSYS® simulation platform respectively, which means that users must first acquire the corresponding base system in order to use the rigid body simulation tools. Both extensions feature a FEA approach to simulate dynamic problems involving mixed, flexible and/or rigid bodies. ANSYS®, however, has the advantage of holding an ISO 9001 certificate for the quality of its products.

All the simulation platforms mentioned so far give evidence of the fact that the computer analysis of rigid body dynamics is an important topic in science and engineering, a topic which is constantly improving and becoming more diverse. As a consequence of the growth in simulation methods and platforms for rigid body dynamics, fields such as aerospace engineering take advantage of the benefits of computer analysis and apply them into the development of specialised technology such as the simulation of a satellite's attitude in orbit. Aerospace simulation tools, as any other software category, can be classified in commercial and open-source/free-to-use. The latter term, free-to-use, means that while a piece of software can be obtained and used for free, one or more parts of the code are obfuscated, leaving them as black boxes. Two examples of such software are the Smart Nanosatellite Attitude Propagator (SNAP)

[8] and the Satellite Dynamics Toolbox [9]. SNAP is a 6-DoF satellite attitude propagator implemented in ML and Simulink®. SNAP features the analysis of environmental torques that affect a satellite with models for gravity gradient torque, magnetic torque due to permanent magnets, magnetic hysteresis torque and damping. Additionally, SNAP enables users to design and analyse passive attitude stabilisation techniques with plotting tools for the attitude's history and the possibility of exporting a file that the Systems Tool Kit® (STK) can use to create animations. The second free-to-use example, the Satellite Dynamics Toolbox, is a ML package that features rigid multi-body dynamics based on the Newton-Euler equations to compute the linear dynamic model of spacecraft with one or more flexible appendages, by that meaning antennae, robotic arms, solar panels, etc.

On the side of the open-source software, or completely free to obtain and modify in other terms, tools such as PROPAT [10] and 42 [11] are also capable of simulating the attitude of spacecraft in space. PROPAT, for instance, is a toolbox for ML that features a set of functions to solve Kepler equations as well as functions to simulate and propagate the attitude and orbit of satellites orbiting around the Earth. The other option, 42, is a standalone, general-purpose multi-body, multi-spacecraft simulation platform written in C/C++. The features include fast simulations of experiments that can be placed anywhere in the solar system, and the capability of being integrated with ML. As MBDyn, 42 has its own input syntax with which users write input text files that contain the description of the simulation experiment.

In parallel to the free software, there is also the commercial approach with the Spacecraft Control Toolbox [12] and the Systems Tool Kit® (STK). The former is a toolbox develop by Princeton Satellite Systems for ML that features tools to design, analyse and simulate spacecraft. Depending on the acquired licence, the functionality provided by the Spacecraft Control Toolbox ranges from a reduced feature set for the development of CubeSats, to a complete system that includes multiple spacecraft subsystems, models for sensors and actuators, and multiple control algorithms for the attitude dynamics and control simulation.

Finally, and probably the most robust spacecraft simulation platform, there is STK. A platform developed by AGI® for providing four-dimensional modelling, simulation, and analysis of objects from land, sea, air, and space. STK also features a complete suite for simulation and visualisation with an accurate model of the Earth in time and space.

In conclusion, the available options make it possible for researchers and developers to study and work with rigid body dynamics as their needs require, be it on the Earth, in space, or in a customised coordinate space. However, none of the mentioned tools targets the educational field, they can clearly be used to teach students how rigid body dynamics work, but before reaching that point students must first go over the respective learning curves, especially with software such as MBDyn or 42, which require users to learn the input syntax; or any of the commercially available options, which require training for the graphical interface. Therefore, the system presented in this document intends to provide a platform with rigid body simulation and visualisation tools that members of the education field, both students and educators, can use to simplify the learning process of rigid body dynamics.

1.1 Objectives

This project presents the Rigid Body Simulation Toolbox (RBT) for ML to simulate and animate the 6-DoF motion of rigid bodies. The design of the toolbox had to comply with three main goals:

1. Serve as an educational tool for professors and students to understand how rigid bodies behave in a 3D space by the effect of disturbances, namely internal and/or external forces and torques.
2. Be modular. The system's functionality must be well delimited in independent modules so that they can be used without the other modules.
3. Easy to use and extend. Requires the system to be written in a programming language common to most users so that modifications can be done without going over a new learning curve.

1.2 Motivation

The RBT aims to have economic and academic impacts. Before describing any of them, it is worth noticing that computational simulations reduce, and sometimes completely eliminate the need of physical tests. In addition, simulations can run repeatedly, subject to a variety of conditions without damaging the system of interest.

The economic impact is foreseen on, although not limited to, reducing the cost of satellite development programmes at the FH Aachen. The simulation capabilities for Attitude Determination and Control Systems (ADCS) will be implemented in future iterations, however, it is important to start considering the possible economic impacts so that the system is developed in a way that clearly helps in the cost reduction of satellite design. Two clear examples of how simulations have reduced the costs associated to a satellite project are [13] and [14], who describe in similar manners that simulation tools have the potential of reducing costs and risks of satellite missions. References [13, 14] discuss in similar terms the potential reductions on satellite costs and mission risks associated with the use of simulation tools. Hu and Li [13] classify satellites as 4-H products, that is high-tech, high-cost, high-benefit and high-risk. Therefore, they argue that simulation tools allow the reduction of costs and risks because they are reusable and non-destructive. Sarsfield [14], discusses the “Goal of a Risk-Based Process Improvement Plan”, shown schematically in Figure 1-1. The plan describes a past, present and future timeline of changes on satellite costs, reliability and performance directly connected to risk management philosophies. In the past, risk was avoided by following spacecraft-classification rules such as spacecraft complexity, launch constraints, etc. These rules set “hard reliability targets” [14] that implicated high development costs. Currently, funding cuts have led satellite designers and mission operators to relax reliability targets and hence to consider the risk a dependent variable rather than a prescribed value. Naturally, this “risk as a resource” [14] approach took risk assessment deeper into the general project management.

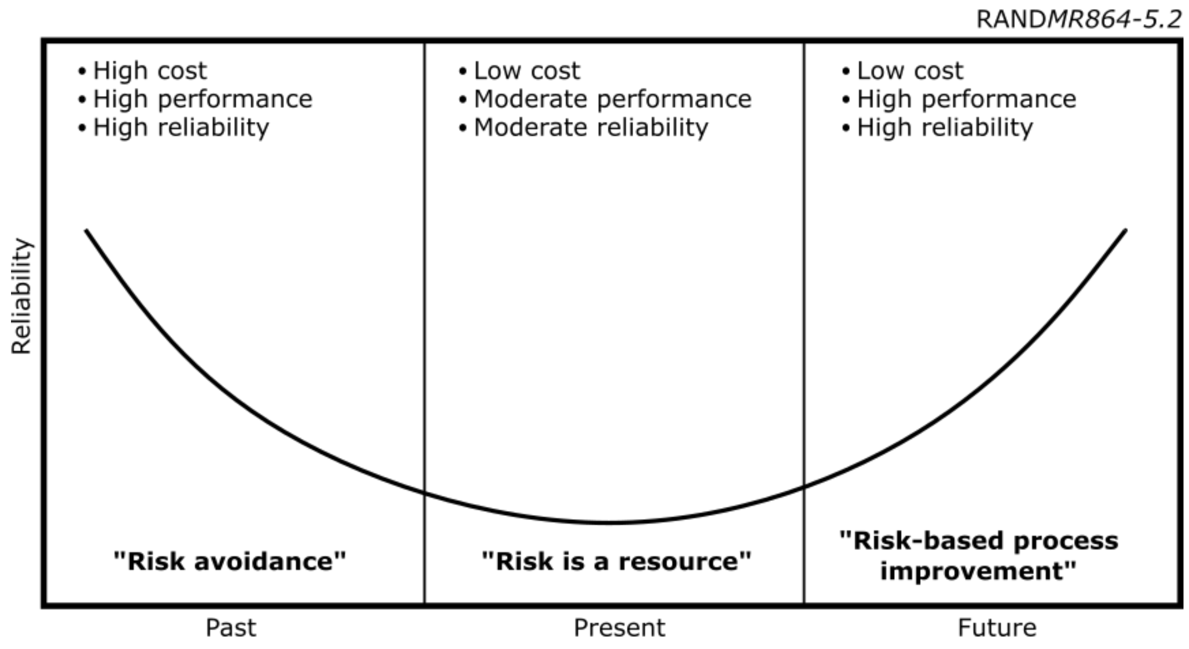


Figure 1-1 Goal of a Risk-Based Process Improvement Plan. Figure taken from [14].

Additionally, the author Sarsfield [14], argues that in the future, risks could be reduced and performance improved at lower costs with the appearance of improved tests processes, new insights into failures in space systems, the development of high-reliability components and subsystems, and advances in design processes; one of which is the adoption of computer simulations during the design stage.

Further on the economic impact, Koenigsmann and Gurevich [15] relates the cost of a satellite to the risk of a failure caused by a total or partial failure of a subsystem, such as the Attitude Control System (ACS). The authors gathered information about the costs of the ACS on different missions and showed that the cost of this subsystem varies between 3-18% of the cost of the bus. Moreover, the cost of ACS is the third greatest expense after launch and operations costs. They also report that the ACS is one of the subsystems with more design failures. The authors of [15] Koenigsmann and Gurevich, introduced the attitude simulation tool AttSim to provide virtual environments in which a satellite’s motion in orbit can be analysed before sending the satellite to space. In this manner developers are enabled to identify potential risks and deliver more reliable systems. Attitude simulation tools are therefore paramount in the development of control algorithms as well as the hardware used to control a satellite’s attitude.

The benefits of simulation tools go beyond the satellite industry, hence the academic impact. Being pieces of software capable of doing fast computations and that usually include visualisation modules, they also become computing tools that induce “Computational Thinking”. CT is a term that refers to a process of abstraction in which a person acquires an image of the essential details of an object or a situation in a way that an information-processing agent (a human, a machine, or a combination of both [16]) can automate such details and their relations. Using CT has proven to accelerate solutions for research and daily life problems. For instance, Wing [16] presents an example where the time to find LEGO® bricks is reduced by a factor of 10 when they are organised using a hashing function. Moreover, cases such as the use of the shotgun sequencing algorithm to accelerate the sequence of the human genome prove that research benefits from CT as well [17]. With such examples, Wing [17] discusses that CT is becoming a part of education at graduate and undergraduate levels and thereafter envisions that CT will also be integrated into childhood education. Furthermore, Wing [18] mentions that “CT will be a reality when it is so integral to human endeavours it disappears as an explicit philosophy”. Visions such as the ones exposed in [17, 18] have led people involved in the education field into the development of strategies to make people develop CT skills and habits. For example, in [19], a computational thinking framework called “Scratch” is developed to allow young people to programme stories, games and simulations. Additionally, some examples in [20] aim to help educators understand the concept of CT so that they can later teach it. One of the shown example sets is modelling and simulation, and they start by citing: “The underlying idea in computational thinking is developing models and simulations of problems that one is trying to study and solve” [21]. In consequence, simulation tools could target a larger audience if they are programmed to exploit the ideas behind CT and serve as tools that reinforce teaching.

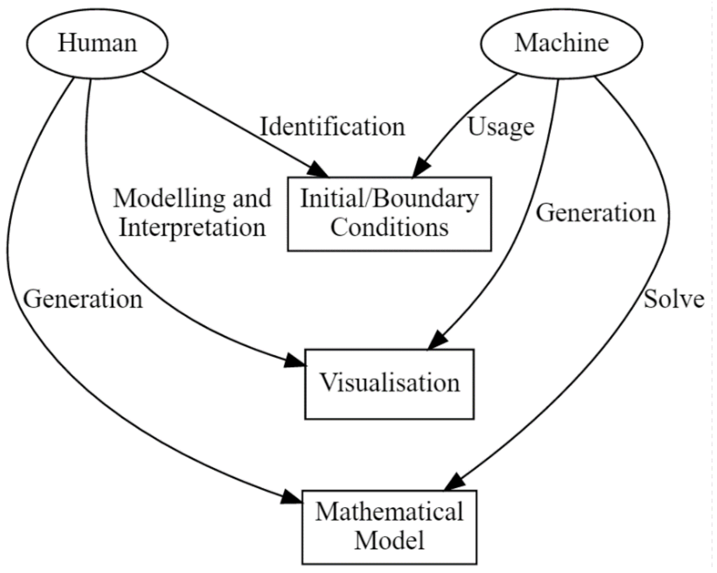


Figure 1-2 Representation of the abstract layers (boxes), automations (arrows) and the computers (ellipsoids) involved in a computational approach for rigid body motion.

In an attempt to illustrate the concepts of CT in terms of the simulation of rigid body dynamics, Figure 1-2 shows 3 abstraction layers obtained from a generic dynamics problem: initial and boundary conditions, mathematical model, and visualisation. Additionally, Figure 1-2 shows how each agent processes the layers using its strengths while the other covers for the weaknesses. For instance, humans are better at text and image processing, therefore the identification of initial and boundary conditions from a text or from a free body diagram is faster on a human brain, whereas machines are better for numerical solutions of mathematical models. In the same way, a human must first generate a virtual representation of the body, usually with a point cloud, so that the machine can create an animation with it, thereafter can a human understand the motion of the body by looking at the animation and relating it to the shape of the body together with the initial and boundary conditions.

As stated in [17], computing tools should provide a direct way into reinforcing the knowledge instead of blocking it or slowing it down. Therefore, within the scope of software computing tools, the complexity of the programming language they are written in is clearly an important factor. Programming languages that are easy to learn and use enable students to focus on understanding and carrying out their tasks instead of spending time debugging code. From an educational perspective, programming languages with simple syntaxes and high levels of abstraction are appropriate options. With the emergence of math-oriented programming languages such as MATLAB®'s scripting language, programmers can now directly code mathematical models and plot data without the need of installing additional libraries or packages, as languages like FORTRAN, C/C++ and Python require. In addition, ML is already being taught and used for research and education in multiple academic institutions, being the FH Aachen one of them. Apart from the mathematical syntax, MATLAB®'s scripting language is an interpreted, operating system-independent language that supports multiple paradigms such as object-oriented (OOP) and procedural programming, thus allowing programmers to develop modular and math-based software. A combination of the programming units of both the procedural and OOP approaches, that is, functions and classes respectively [22], allows projects to distribute its functionality in independent modules through classes for complex entities, and functions for utilities that extend or modify the functionality of such entities. Classes allow the reuse of code through inheritance, and the isolation of information through encapsulation. The final products of these two pillars of OOP are delimited entities with an intuitive operation: once a class is defined, the process of instantiation and method calling resembles the way in which objects or people are placed somewhere and told what to do. Such an operation complies with the CT goal of making a tool that is easy to use. An important detail concerning the inheritance in object-oriented development in ML is that the classes are always derivations of either the "value" or the "handle" classes. The basic difference between the two is that an instance of a "value" class is an instance to an object, whereas an instance of a "handle" class is a reference to an object. Hence, copying a "value" creates a clone, while copying a "handle" creates a reference, thus providing by-value and by-reference functionalities. Additionally, the "handle" base class provides event-driven functionality, which allows developers to create interactive programmes to give users real-time feedback about their actions. One last important factor to consider in the selection of a programming language is the confidence in the language's continued development or support. The growth figures reported for ML in [23] show that as of 2004, the number of employee and user counts grew by ratios of 1000 and 2000, respectively, in 20 years. From those quantities it is possible to assume that ML growth will still be maintained.

In another subject, reviewing how the simulation tools mentioned in Chapter 1: could meet the objectives of the project makes it possible to conclude that commercial tools cannot be used because, in contrast to ML, the FH Aachen does not provide licences for them. Moreover, in the case of ANSYS® and COSMOL®, users also require knowledge of FEA to setup simulations. On the other hand, being in a situation where a new application must be developed, the use of simulation libraries and engines in languages that are not math-oriented is inconvenient because, as explained before, they may complicate the educational approach. Similarly, a direct use of MBDyn and 42 is not practical because users must learn how to create the input files using the required syntax in order to run simulations. Even though the syntaxes are not complicated, they force the users to go over a learning curve they can avoid with a toolbox made for a platform they already know, in this case, ML. However, by taking advantage of the open-source licences of MBDyn, 42, and other mentioned tools, some of the algorithms could be ported to ML to extend the functionality of the application. For instance, when adding an attitude propagation module to the project, the code from PROPAT can be used and adapted to the project's needs to avoid rewriting the code.

For these reasons, and delimiting the RBT to three main modules: one to calculate the mass distribution properties of a rigid body (centre of mass, total mass and moment of inertia), another one to simulate the motion of such bodies, and a visualisation tool to animate the results of the simulation; the development of a new modular simulation platform could provide students, educators and researchers with tools that not only simulate rigid body dynamics, but also induce the trending CT, and allow future developers to easily incorporate custom functionality such as multibody dynamics and motion controllers, which in turn provide a robust platform for the simulation of a satellite's attitude.

1.3 Architecture

The RBT has three main modules upon which future development will be made:

1. A mass distribution calculator (MDC).
2. A 6-DoF motion simulator (SMOD).
3. An animation module (AMOD).

Figure 1-3 shows the interaction of the modules with each other and with the utility functions and classes. The architecture shown in Figure 1-3 allows the modules to be independent from each other and directly compatible at the same time. The independence means that users are able to run each module individually. The direct compatibility means that experiments can run in a pipeline of two or more modules, where the results of one module are directly passed to the next.

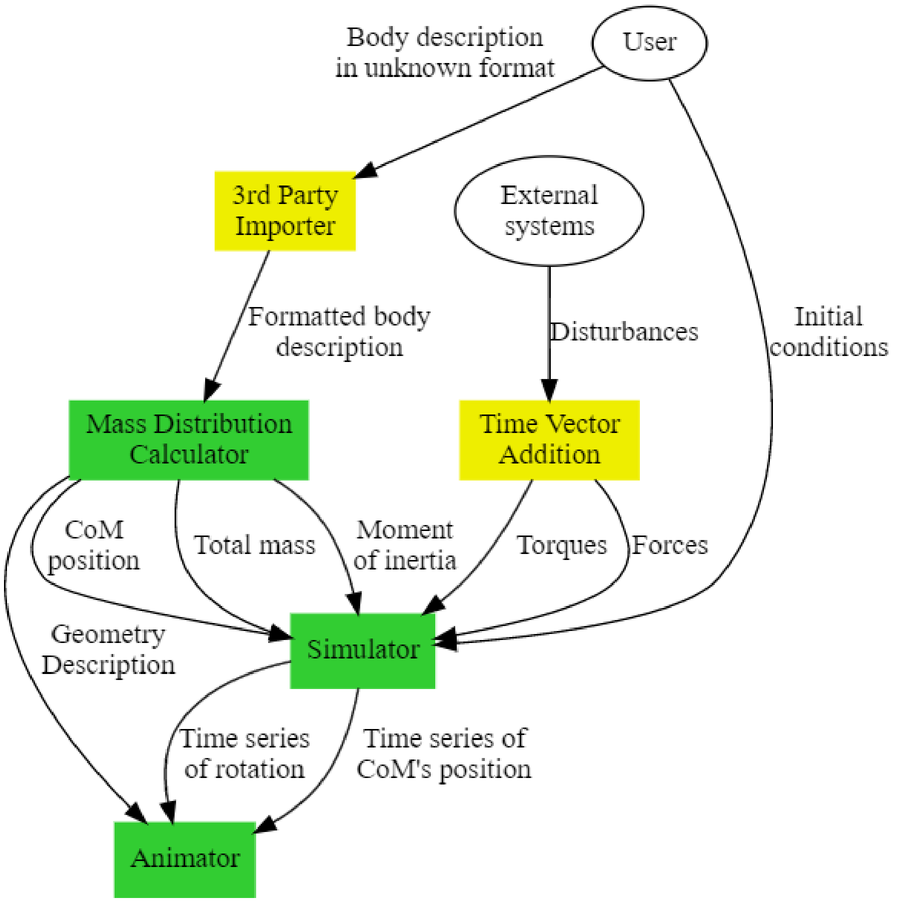


Figure 1-3 Flow diagram of how the modules interact with each other. The green rectangles are the core modules, yellow rectangles are the utilities, and the white ellipses represent external entities that interact with the system.

The following sections briefly describe the purpose of each module and how it interacts with the rest of the system, given that detailed descriptions are given in the corresponding chapters.

1.3.1 Mass Distribution Calculator (MDC).

A tool that takes a list of vertices, facets and material properties such as thickness and density to calculate the total mass of the body, as well as the location of the Centre of Mass (CoM) and the moment of inertia.

1.3.2 6-DoF Motion Simulator (SMOD)

A module that simulates the motion of a rigid body from a set of initial conditions, disturbances such as internal and external forces and torques, and the body's mass and moment of inertia.

1.3.3 Animation Module (AMOD)

A module that takes as input the geometry description of the body (vertices, facets and colour) and a time-spread vector with the position and orientation values so that it can move geometries in the scene and create an animation with it.

1.3.4 Utility Functions and Classes

The system includes a set of utility functions and classes to ease the data transfer from external and future entities to the core modules. The utilities shown in Figure 1-3 are a function to convert body description data from a specific format to the one that the MDC uses, and a class to add vectors of different types.

1.3.4.1 Body Description Converter

A function that converts a geometry description of a body from a third party format such as NASTRAN® to the format the MDC supports.

1.3.4.2 Time Vector Addition

A class that concatenates different vector sources such as constants and time changing functions and adds their values at a specific moment in time.

1.4 Deliverables

From the core modules the SMOD is delivered because it is the core of the entire system and it is the main reason for the development of the project. In addition to the SMOD, the AMOD is delivered because it displays the results of the simulation in a manner that clearly shows the rigid body motion. The AMOD is of great importance for the academic impact of the project.

From the system's utilities, the Time Vector Addition class was developed to provide a tool to concatenate multiple constant and time changing disturbances, thus enabling more advanced simulations.

The remaining modules and utilities that appear in Figure 1-3 were left for future development. In that way, feedback for the first iteration can also be applied on further iterations.

Chapter 2: Motion Simulation Module

2.1 Overview

The SMOD is a tool capable of simulating the 6-DoF motion of a rigid body, it is the core of the system and it is the main reason for the development of this project. The reason for the SMOD to be considered the core of the system is that it is what computes the motion of a rigid body.

2.1.1 Previous work

There are several ways to simulate the motion of rigid bodies in ML, examples of that are the use of existing simulators such as the ones discussed in Chapter 1:, writing the code for the required mathematical models with the standard ML language, or the use of Simulink®. The latter has a graphical interface in which users connect blocks to create models of dynamic systems. Simulink®'s standard blocks are enough to assemble the model for the motion of a rigid body. However, The MathWorks™ developed the Aerospace Blockset™ for Simulink®, which adds blocks that encapsulate operations such as the computation of the 6-DoF motion of a rigid body, coordinate transformations, and 3D visualisation. On a different approach, where ML's programming language is used by either writing a script or by using the ML console, the equations of motion for a moving rigid body can be solved with numerical methods or with symbolic mathematics. Gace [24, 25] constructed functions that simulate the 6-DoF motion of a single rigid body with fixed or variable mass by using ML's explicit Runge-Kutta (4,5) integrator (ODE45) to solve the Newton-Euler equations. The functions take as input a set of initial conditions for the position, orientation, linear and angular velocities, and in the case of the function that simulates the motion of bodies with variable mass, an initial condition for the mass is also required. It is important to notice that, although the functions consider the initial orientation of the body, they do not compute the change of orientation with respect to time. Another example that utilises numerical methods is the report by Ganapathi [26], who focuses on the study of the conservation of angular momentum of a symmetrical top and an asymmetrical top under torque and torque-free conditions. The code in the report implements a fourth order Runge-Kutta integrator to solve the Euler equations of rotational motion to find the angular velocity, and the quaternion's kinematic equation (see Eq. 1.18) to find the body's orientation.

In contrast to the numerical methods approach, there is also the use of symbolic mathematics. The book by Harper [27], for instance, has many examples of how to use ML's Symbolic Toolbox™ to obtain symbolic solutions to mechanics problems such as translation, rotation, impulse and momentum, work and energy, and vibrations. In addition to the examples, the book has an introduction to ML so that students can get familiar with the programming language, the console, the scripts, the graphics, and naturally, with ML's approach to symbolic mathematics.

Lastly, and as mentioned in Chapter 1:, there are also third-party toolboxes for ML that support the simulation of the motion of multibody systems. Examples of such toolboxes are the "Satellite Dynamics Toolbox" [9, 28], "QuIRK" [5], and the "Spatial Vector and Rigid-Body Dynamics Software" [6]. The Satellite Dynamics Toolbox (SDT) uses the Newton-Euler equations for the simulation of multi-body dynamics in space applications. In addition, the SDT is capable of simulating the motion of any multi-body system assembled as an open chain by a base, joints and appendages. QuIRK, is a toolbox for ML programmed in an object oriented fashion to improve modularity. QuIRK stands for Quaternion-state Interface for Rigid-body Kinetics, and as the name suggests, it uses quaternions to represent the attitude of the bodies. QuIRK uses the Udwadia-Kalaba pseudoinverse method, augmented for singular mass matrices [29], to calculate the motion of constrained multi-body systems. Additionally, QuIRK includes a visualisation tool to display the results of the simulations. The third and last example, Spatial [6], is a set of functions built to compute vector arithmetic and multi-body dynamics in ML. The algorithms implemented in the functions support both constrained and unconstrained motion.

In conclusion, the development of a motion simulator has multiple possible combinations when considering the number of bodies to simulate (single-body or multi-body), the solutions through symbolic mathematics or numerical methods, the chosen mechanics (Newtonian or a generalised-coordinates method such as the Udwadia-Kalaba), and the attitude simulation method (Euler angles or quaternions).

2.1.2 Module's Design

Mechanics. To decide which mechanics approach to use, the source code and documentation of [5], [6], [9] and [24] was inspected. Thereafter, it was concluded that using Newton-Euler mechanics is simpler in comparison to a generalised-coordinates method. Hence, and based on the statement of Zipfel [30] that says: “For any new dynamic theory to be acceptable, it must contain Newtonian dynamics as a limiting case”, the SMOD is limited to the use of Newton-Euler mechanics for the simulation, leaving the implementation of a generalised coordinates method for a future implementation.

Attitude. The SMOD is capable of simulating the attitude of a rigid body with Euler angles and quaternions. Both methods were chosen so that users have multiple options. From an academic point of view, being able to observe the differences between the two methods enables users to understand particular events such as how rigid bodies behave when a singularity occurs in the Euler angles simulation and how quaternions are used to avoid it.

Single body simulation. For the purposes of this thesis, the SMOD is limited to the simulation of a single rigid body. Support for multibody dynamics will be incorporated in future development, as well as attitude propagation and attitude control modules for satellites.

Numerical solvers. The SMOD is limited to the use of numeric solvers because the platform should be as generic as possible regarding the motion of a rigid body, and according to Harper [27], students should use numerical solutions only when the symbolic approach fails, meaning that the numerical solutions have a higher success rate.

Coordinate system. A three-dimensional Euclidean space is used to represent the position of points in the physical world, where the Z axis points to the zenith.

Units system. All the units in the module follow the MKS unit system.

2.2 Rigid Body Dynamics

This section gives a brief theoretical basis of the mathematical models used in the implementation of the SMOD. As mentioned in section 2.1.2, the module runs simulations using Newton-Euler mechanics, therefore, the rigid body dynamics described in this section are based on Newton’s Three Laws of motion and Euler’s equations for rotational motion.

Using the concepts of particle, rigid body and centre of mass from [31], a particle is a hypothetical object that has mass but no extension, a rigid body is a collection of such objects, and the centre of mass (CoM) is the point with the position vector defined by:

$$r_c^e = \frac{\sum_i m_i r_i^e}{m} \qquad \text{Eq. 1.1}$$

Where r_c^e is the position vector for the body's CoM with respect to the inertial frame e, m_i is the mass of the i_{th} particle, r_i^e is position vector referred to the inertial frame e of the i_{th} particle, and $m = \sum_i m_i$. Figure 2-1 illustrates a rigid body as a collection of particles and the particles' position vectors with respect to the IRF.

Moreover, "a collection of particles is called rigid body if there exists a reference frame in which each particle's position remains the same over time" [31]. Such a reference frame is called body-fixed reference frame (BRF). Figure 2-2 shows a rigid body with a BRF fixed to its CoM.

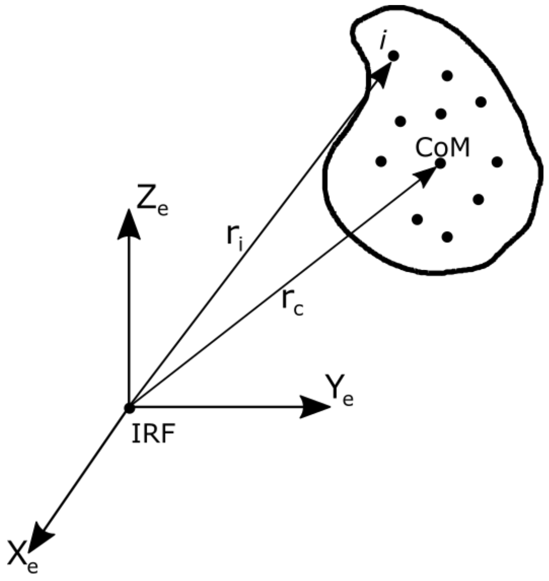


Figure 2-1 Collection of i particles with position vectors r_i representing a rigid body with centre of mass located at r_c.

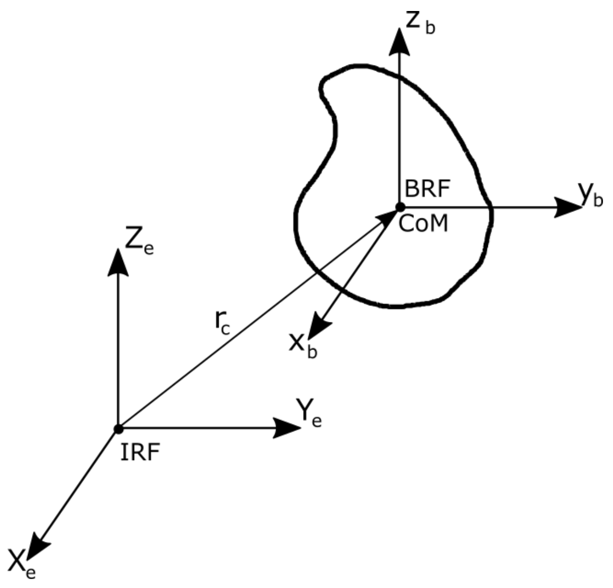


Figure 2-2 Relation between the IRF and a BRF centred at the body's CoM.

2.2.1 Translational Motion

The translational dynamics of a particle in Newtonian mechanics are represented by Newton's second law, which states that the rate of change of linear momentum of a particle is equal in magnitude and direction to the force applied to the particle [30]. Which for any particle i is represented by:

$$\dot{L}_i^b = f_i^b \qquad \text{Eq. 1.2}$$

Where $\dot{L}_i^b$ is the rate of change of linear momentum of the particle i with respect to time and with respect to the inertial frame b, and f_i^b is the force applied to the particle with respect to the reference frame b. Then, since the linear momentum is the product of the particle's mass and velocity, Eq. 1.2 can be rewritten as:

$$m_i \dot{v}_i^b = f_i^b \qquad \text{Eq. 1.3}$$

Where $\dot{v}_i^b$ is the rate of change with respect to time of linear velocity of the particle with respect to the reference frame b.

In general terms, the internal forces acting on the particle should also be considered. The internal forces are those that particles apply on each other. Then, the total force acting on the i_{th} particle is given by:

$$\sum_j f_{ij}^x + \sum_j f_{ij}^d = m_i \dot{v}_i^b \qquad \text{Eq. 1.4}$$

Where f_{ij}^x is the j_{th} external force applied to the i_{th} particle and f_{ij}^d is the j_{th} internal force applied to the i_{th} particle. However, when considering that in rigid bodies the particles keep their positions with respect to the BRF at any point in time, the internal forces cancel each other out by Newton's Third Law: "To every action there is always an opposed and equal reaction" [30]. So the equation for the total force applied to the body results:

$$F^b = L^b = \sum_i m_i \dot{v}_i^b \qquad \text{Eq. 1.5}$$

Where F^b is the total force applied to the collection of particles. Next, considering that the CoM of a rigid body is a particle with mass $m = \sum_i m_i$ and a position vector defined by Eq. 1.1, the rate of change of position with respect to time of the CoM with respect to the reference frame b is defined by:

$$v_c^b = \dot{r}_c^b = \frac{\sum_i m_i \dot{r}_i^b}{m} \qquad \text{Eq. 1.6}$$

Then, the following demonstration proves that analysing the motion of a body's CoM is equivalent to analysing the motion of the entire body:

$$F_c^b = m\dot{v}_c^b = m\ddot{r}_c^b = m\frac{\sum_i m_i \ddot{r}_i^b}{m} = \sum_i m_i \ddot{r}_i^b = \sum_i m_i \dot{v}_i^b = F^b \qquad \text{Eq. 1.7}$$

Where F_c^b is the force applied to the CoM with respect to the reference frame b. With the demonstration, it is possible to reformulate Eq. 1.5 so that it applies the total force to the CoM and thus to entire rigid body:

$$m\dot{v}^b = F^b \qquad \text{Eq. 1.8}$$

From this point on, consider the reference frame b the BRF and the inertial frame e the IRF. So, to observe the motion with respect to the IRF, a transformation $[T_b^e]$ is applied to v^b:

$$v^e = [T_b^e] v^b \qquad \text{Eq. 1.9}$$

Then the force becomes:

$$F^e = m\dot{v}^e = m\left(\frac{d([T_b^e]v^b)}{dt}\right) = m\left([\dot{T_b^e}]v^b + [T_b^e]\dot{v}^b\right) \qquad \text{Eq. 1.10}$$

As stated in [32], $\left[\dot{T}_b^e\right] = [T_b^e][\Omega^b]$, where $[\Omega^b]$ is the skew antisymmetric matrix of ω^b. Then:

$$F^e = m([T_b^e][\Omega^b]v^b + [T_b^e]\dot{v}^b) = m[T_b^e](\dot{v}^b + \omega^b \times v^b) \qquad \text{Eq. 1.11}$$

Thereafter, applying the inverse transformation $[T_e^b]$ to both sides of Eq. 1.11 and rearranging to leave $\dot{v}^b$ on the left side of the equation:

$$\dot{v}^b = \frac{F^b}{m} - \omega^b \times v^b \qquad \text{Eq. 1.12}$$

2.2.2 Rotational Motion

Before starting the analysis of the rotational motion, the concept of moment of inertia needs to be introduced. The moment of inertia represents the mass distribution of a rigid body with respect to a certain inertial frame [32]. In a three-dimensional space, the moment of inertia I^b with respect to the reference frame b is defined by:

$$I^b = \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{xy} & I_{yy} & -I_{yz} \\ -I_{xz} & -I_{yz} & I_{zz} \end{bmatrix} \qquad \text{Eq. 1.13}$$

Where the elements in the diagonal are called principal moments of inertia and rest of the terms are called products of inertia. The principal moments of inertia are the product of the mass of a particle i and the square of the shortest distance to one of the coordinate axes of the reference frame b. [32]

$$\begin{aligned} I_{xx} &= \sum_i m_i(y_i^2 + z_i^2) \\ I_{yy} &= \sum_i m_i(x_i^2 + z_i^2) \\ I_{zz} &= \sum_i m_i(x_i^2 + y_i^2) \end{aligned} \qquad \text{Eq. 1.14}$$

Similarly, the product of inertia of a particle i with respect to a set of two orthogonal planes is the product of the mass of the particle and the shortest distances from the planes to the particle:

$$\begin{aligned} I_{xy} &= \sum_i m_i x_i y_i \\ I_{yz} &= \sum_i m_i y_i z_i \\ I_{xz} &= \sum_i m_i x_i z_i \end{aligned} \qquad \text{Eq. 1.15}$$

Since the moment of inertia is calculated with the distances to the axes of the reference frame, there is a different moment of inertia matrix for each different reference frame. Therefore, bringing back the concept of a BRF where all the particles of a rigid body keep their original position over time, it is possible to conclude that a moment of inertia with respect to the BRF will remain constant. Therefore, the following analysis considers a BRF b fixed to the body's CoM as in Figure 2-2. And now, the angular momentum is defined by:

$$H^b = I^b \omega^b \qquad \text{Eq. 1.16}$$

Where H^b is the angular momentum of the rigid body with respect to the BRF. Then, to observe the motion with respect to the IRF, a process similar to the one for Eq. 1.12 with a transformation $[T_b^e]$ is carried out and results in:

$$\dot{\omega}^b = [\mathrm{I}^{-1}]^b \left(M^b - \omega^b \times (I^b \omega^b) \right) \qquad \text{Eq. 1.17}$$

Where M^b is the applied torque about the CoM.

2.2.3 Attitude Simulation

2.2.3.1 Quaternions

For the following equations, consider an arbitrary unit quaternion q with magnitudes $[q_0 \quad q_1 \quad q_2 \quad q_3]^T$ on a base $[w \quad i \quad j \quad k]$, where w is the real component and $[i \quad j \quad k]$ are the vector or complex components.

According to Shuster and Dillinger [33], the quaternions are considered the best option for attitude simulation because its kinematic equation is linear and which satisfies only a single constraint that is easy to enforce. The constraint is $q_0^2 + q_1^2 + q_2^2 + q_3^2 = 1$.

The kinematic equation of the quaternion can be written in a matrix representation [34]:

$$\begin{bmatrix}\dot{q}_0\\ \dot{q}_1\\ \dot{q}_2\\ \dot{q}_3\end{bmatrix} = \frac{1}{2}\begin{bmatrix}0 & -\omega_{bx} & -\omega_{by} & -\omega_{bz}\\ \omega_{bx} & 0 & \omega_{bz} & -\omega_{by}\\ \omega_{by} & -\omega_{bz} & 0 & \omega_{bx}\\ \omega_{bz} & \omega_{by} & -\omega_{bx} & 0\end{bmatrix}\begin{bmatrix}q_0\\ q_1\\ q_2\\ q_3\end{bmatrix} \qquad \text{Eq. 1.18}$$

2.2.3.2 Euler Angles

The reason for the term “6-Degrees of Freedom” to exist is that the position and orientation of one reference frame with respect to another can be represented by a set of three coordinates $[x \quad y \quad z]$ and a set of three angles $[\phi \quad \theta \quad \psi]$. These angles are called Euler or Eulerian angles and they are used in a sequence of rotations to represent the orientation of one reference frame with respect to another. Figure 2-3, Figure 2-4, Figure 2-5 and Figure 2-6 illustrate how the BRF is rotated by the angles $[\phi \quad \theta \quad \psi]$ with respect to the IRF in a Z-Y-X sequence. The principal axes of the IRF are $[X_e \quad Y_e \quad Z_e]$ while the BRF has $[x_b \quad y_b \quad z_b]$. In all figures, the axes $[x' \quad y' \quad z']$ represent positions for $[x_b \quad y_b \quad z_b]$ after each rotation. Finally, $[\dot{\phi} \quad \dot{\theta} \quad \dot{\psi}]$ are the rates of change of the Euler angles with respect to time.

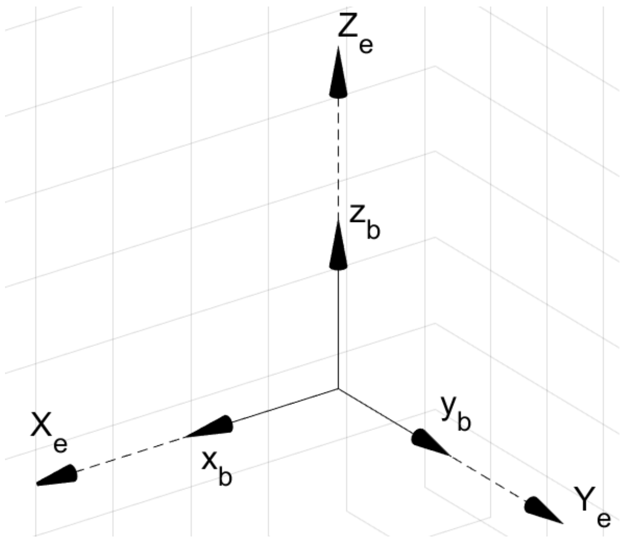


Figure 2-3 IRF (dotted) and BRF (continuous) aligned.

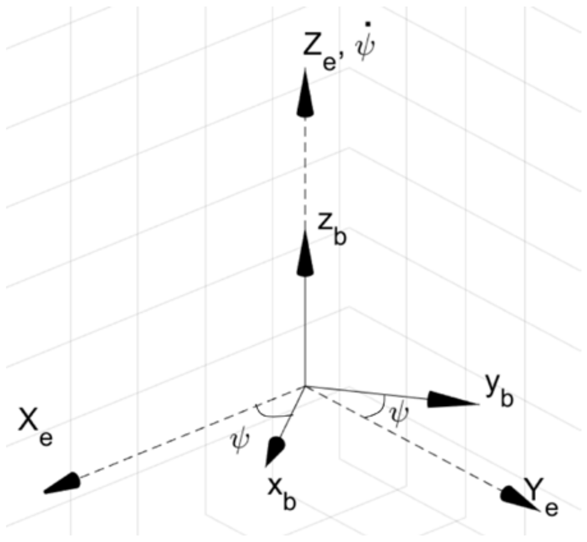


Figure 2-4 BRF (continuous lines) rotated by ψ with respect to the IRF (dotted lines).

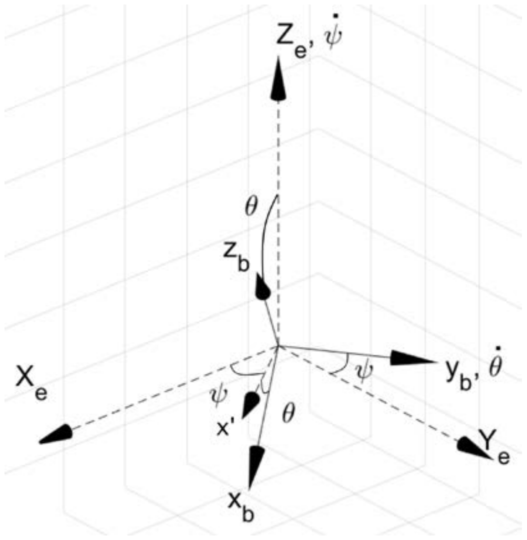


Figure 2-5 BRF (continuous lines) rotated by ψ and θ with respect to the IRF (dotted lines). x' is a previous position of the BRF.

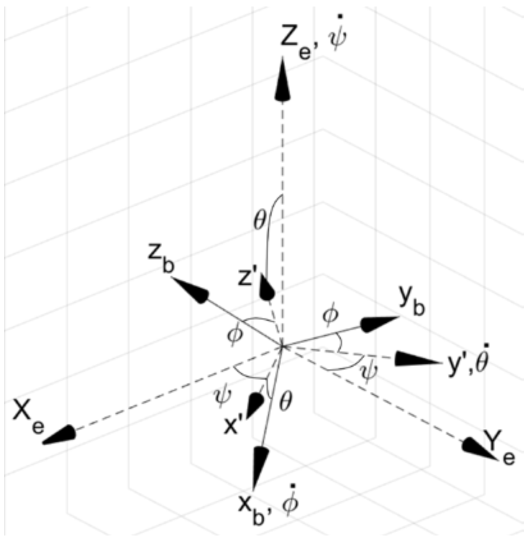


Figure 2-6 BRF (continuous lines) rotated by ψ, θ, and ϕ with respect to the IRF (dotted lines). $[x' \quad y' \quad z']$ *are previous positions of the BRF.*

The values of the angles can be determined with the help of Euler's theorem, which states that any combination of rotations of one reference frame with respect to another can be represented by one single rotation about some axis. However, before combining rotations, it is necessary to understand that combining finite rotations has different results as when combining infinitesimal rotations. While infinitesimal rotations are commutative, finite rotations are not. To demonstrate such a statement, take the following rotation matrices:

$$[R(\phi)] = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{pmatrix} \qquad [R(\theta)] = \begin{pmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{pmatrix}$$

When commutative, the following condition should be met:

$$[R(\phi)][R(\theta)] = [R(\theta)][R(\phi)] \qquad \text{Eq. 1.19}$$

Using finite values for ϕ and θ results in a inequality:

$$\begin{pmatrix} \cos\theta & 0 & -\sin\theta \\ \sin\phi\sin\theta & \cos\phi & \sin\phi\cos\theta \\ \cos\phi\sin\theta & -\sin\phi & \cos\phi\cos\theta \end{pmatrix} \neq \begin{pmatrix} \cos\theta & 0\sin\phi\sin\theta & -\cos\phi\sin\theta \\ 0 & \cos\phi & \sin\phi \\ \sin\theta & -\cos\theta\sin\phi & \cos\phi\cos\theta \end{pmatrix}$$

The condition in Eq. 1.19 is not met. Hence, finite rotations are not commutative. However, the result is different when infinitesimal angles are used. First, the limits when $\phi \to 0$ and $\theta \to 0$ are found:

$$\lim_{\phi\to 0;\,\theta\to 0}[R(\phi)][R(\theta)] = [I] \qquad\qquad \lim_{\phi\to 0;\,\theta\to 0}[R(\theta)][R(\phi)] = [I]$$

Where $[I]$ is a 3x3 identity matrix. Then in Eq. 1.19:

$$\lim_{\phi\to 0;\,\theta\to 0}[R(\phi)][R(\theta)] - [R(\theta)][R(\phi)] = [I] - [I] = 0$$

That proves that infinitesimal rotations are commutative, which means that they can be used as vectors, as well as their rates of change with respect to time, or time derivatives in other words.

As demonstrated in [31], the direction and magnitude of the angular velocity of a rigid body represent the body's axis and rate of rotation respectively. Then, Euler's theorem can be applied to represent the angular velocity of the body (single rotation) with the time-rates of change of the Euler angles (multiple rotations). It can be seen in Figure 2-6, that the components of the angular velocity in terms of the rates of change of the Euler angles are:

$$\begin{bmatrix}\omega_x\\ \omega_y\\ \omega_z\end{bmatrix}^b = \begin{bmatrix}\dot{\phi}\\ 0\\ 0\end{bmatrix} + \begin{pmatrix}1 & 0 & 0\\ 0 & \cos\phi & \sin\phi\\ 0 & -\sin\phi & \cos\phi\end{pmatrix}\begin{bmatrix}0\\ \dot{\theta}\\ 0\end{bmatrix} + \begin{pmatrix}1 & 0 & 0\\ 0 & \cos\phi & \sin\phi\\ 0 & -\sin\phi & \cos\phi\end{pmatrix}\begin{pmatrix}\cos\theta & 0 & -\sin\theta\\ 0 & 1 & 0\\ \sin\theta & 0 & \cos\theta\end{pmatrix}\begin{bmatrix}0\\ 0\\ \dot{\psi}\end{bmatrix} \qquad \text{Eq. 1.20}$$

Finally, the matrices in Eq. 1.20 are concatenated and inverted to obtain an ordinary differential equation (ODE) that will output the instantaneous values of the Euler angles when integrated. It is important to notice, however, that the Euler angles in Eq. 1.20 represent rotations of the BRF with respect to the IRF, but in Eq. 1.21, since the matrix is inverted, the Euler angles represent a rotation of the IRF with respect to the BRF.

$$\begin{bmatrix}\dot{\phi}\\ \dot{\theta}\\ \dot{\psi}\end{bmatrix} = \begin{bmatrix}1 & \sin\phi\tan\theta & \cos\phi\tan\theta\\ 0 & \cos\phi & -\sin\phi\\ 0 & \dfrac{\sin\phi}{\cos\theta} & \dfrac{\cos\phi}{\cos\theta}\end{bmatrix}\begin{bmatrix}\omega_x\\ \omega_y\\ \omega_z\end{bmatrix}^b \qquad \text{Eq. 1.21}$$

2.2.4 Coordinate Transformations

A coordinate transformation is an operation that transforms a vector from one coordinate system to another. In the general terms of linear algebra, such operations are called linear transformations, and they transform vectors from one vector space to another. In geometry, rotations, reflexions, translations and projections are examples of linear transformations [35].

Within the simulation, quaternions and Euler angles are used to create linear transformations to rotate one reference frame with respect to another, that is, to rotate the BRF with respect to the IRF, as seen in Figure 2-6. Similarly, the inverse transformation will rotate the IRF with respect to the BRF.

The next two subsections describe how the rotations are generated in both the quaternion and Euler angle approaches and how they are applied.

2.2.4.1 Transforming with quaternions

As mentioned before, and within the scope of this project, transformations also receive the name of rotations. Hence, the SMOD uses *quatrotate* from the Aerospace Toolbox™ to create a rotation matrix from an input quaternion and apply it to rotate vectors. Internally, *quatrotate* converts an input quaternion into a DCM $[R_a^b]$ by calling *quat2dcm*. Then the function uses the $[R_a^b]$ to rotate an input vector p^a into a rotated vector p^b as in Eq. 1.22.

$$p^b = [R_a^b]p^a \qquad \text{Eq. 1.22}$$

Consequently, knowing that the quaternions that result from the simulation rotate vectors of the IRF with respect to the BRF, it is deduced that using *quatrotate* on the simulation's output generates $[R_e^b]$, which according to the function's documentation has the form of:

$$[R]_e^b = \begin{pmatrix} (q_0^2 + q_1^2 - q_2^2 - q_3^2) & 2(q_1 q_2 + q_0 q_3) & 2(q_1 q_3 - q_0 q_2) \\ 2(q_1 q_2 - q_0 q_3) & (q_0^2 - q_1^2 + q_2^2 - q_3^2) & 2(q_2 q_3 + q_0 q_1) \\ 2(q_1 q_3 + q_0 q_2) & 2(q_2 q_3 - q_0 q_1) & (q_0^2 - q_1^2 - q_2^2 + q_3^2) \end{pmatrix} \qquad \text{Eq. 1.23}$$

In the opposite way, inverting the quaternion with a call to *quatinv* before the call to *quatrotate* causes vectors to instead be transformed by $[R_b^e]$. The quaternion inverting function *quatinv* is also part of the Aerospace Toolbox™.

2.2.4.2 Transforming with Euler Angles

As demonstrated in section 2.2.3.2, the combination of Euler angles is not commutative, meaning that the same angles will produce different rotations when combined in different sequences. Recalling the explanations of section 2.2.3.2, the Euler angles in Eq. 1.20 are used in a Z-Y-X sequence to represent rotations of the BRF with respect to the IRF. On the other hand, since Eq. 1.21 is the inverse of Eq. 1.20, its Euler angles are used in a X-Y-Z sequence to represent rotations of the IRF with respect to the BRF, which can be expressed as:

$$[R_e^b] = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{pmatrix} \begin{pmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{pmatrix} \begin{pmatrix} cos\psi & sin\psi & 0 \\ -sin\psi & cos\psi & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \text{Eq. 1.24}$$

And in consequence, rotations of the BRF with respect to the IRF will be done with:

$$[R_b^e] = \begin{pmatrix} -\cos\psi & -\sin\psi & 0 \\ \sin\psi & -\cos\psi & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} -\cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & -\cos\theta \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & -\cos\phi & -\sin\phi \\ 0 & \sin\phi & -\cos\phi \end{pmatrix} \quad \text{Eq. 1.25}$$

2.2.5 State Vector

At this point, the equations of motion have been derived and can be concatenated in a single state vector. The quantities of interest are the position and linear velocity of the body's CoM with respect to the IRF, the angular and linear velocities of the body with respect to the BRF, and the attitude representation. Hence, the state vector is created with a combination of Eq. 1.12, Eq. 1.17, Eq. 1.18, Eq. 1.21, Eq. 1.25 and the inverse of Eq. 1.23.

$$S = \begin{bmatrix} \dot{r}^e \\ \dot{v}^e \\ \dot{v}^b \\ \dot{\omega}^b \\ \dot{A} \end{bmatrix} = \begin{bmatrix} [R_b^e]v^b \\ [R_b^e]\dfrac{F^b}{m} \\ \dfrac{F^b}{m} - \omega^b \times v^b \\ {I^b}^{-1}\left(M^b - \omega^b \times \left(I^b \omega^b\right)\right) \\ A(t) \end{bmatrix} \quad \text{Eq. 1.26}$$

Where r^e is the position of the body with respect to the IRF, v^e is the linear velocity with respect to the IRF and $A(t)$ is the selected attitude simulation function, i.e. either Eq. 1.18 or Eq. 1.21. In addition, depending on the selected attitude simulation method, $[R_b^e]$ is generated with either Eq. 1.25 or the inverse of Eq. 1.23.

The known values in the state vector S are the input force F^b, the input torque M^b, the moment of inertia I^b, and the mass of the rigid body. Additionally, the initial values for r^e, v^e, v^b, ω^b and for whatever the attitude representation is, are also known.

2.3 Software Implementation

2.3.1 Module Requirements

The simulation module needed to meet the following requirements in order to provide a convenient rigid body motion simulation platform:

Independence from the other modules. Mass calculations must not be mandatory prior to a simulation and an animation must not be mandatory to visualise the results.

Simulation of 6 Degrees of Freedom. The module must be able to simulate a rigid body's displacement along and rotation about the *X, Y* and *Z* components of the Cartesian coordinate system.

Support for generic rigid bodies. There must be no constraints in the shape or dimensions of the rigid body being simulated.

Ease of use. Users should not have to deal with the equations of motion themselves.

Simple code. The programme's code must be written in simple and clear statements that enable any user, disregarding the amount of programming experience, to fully understand it in case they wish to modify the module's functionality.

Different solver options. The module must support more than one ordinary differential solver option so that users can choose the one of their preference.

Support for different attitude simulation methods. The module should be able to simulate the attitude of a rigid body with quaternions as well as with Euler angles.

Time-changing disturbances. The module should be able to solve problems with forces and torques that vary with time.

2.3.2 Module Workflow

The module is implemented in one class: *RbSimulation*, which corresponds to one rigid body for every instance of the class. The simulations have two mandatory steps: instantiate an object of the class and running the simulation. Figure 2-7 shows the flow with which simulations are run with the module. Steps marked with dotted lines are optional.

All the initial conditions, mass properties, disturbances and simulation time span have default values, so it is technically possible to run simulations just after instantiation, although all the results would be zero. Moreover, the attitude is simulated with quaternions by default, based on the statement in [33] that says they are the best option for the task.

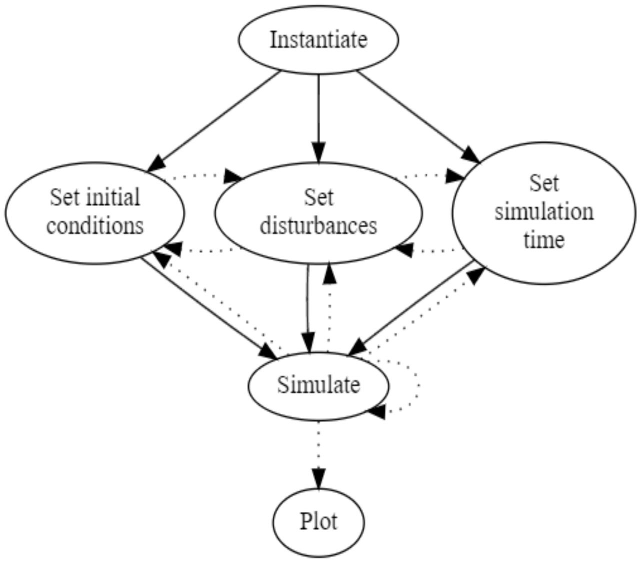


Figure 2-7 Workflow of the SMOD.

Internally, the class runs the simulation by calling the selected ODE solver to integrate the state vector S, from Eq. 1.26. The results of the simulation are stored in five motion-related vectors: the position and linear velocity with respect to the IRF (r^e and v^e), the linear and angular velocities with respect to the BRF (v^b and ω^b), and an array of either quaternions or Euler angles depending on the chosen attitude simulation method.

A guide on how to use the module, including samples of code can be found in Appendix B:.

2.3.3 RbSimulation Class

The following sections explain the purpose and functionality of the class's properties and methods, as well as how they interact with each other. Figure 2-8 shows a graphic description of the module's structure. *RbSimulation* is a handle class to provide event based behaviour.

2.3.3.1 Properties

The following paragraphs explain the purpose of the class's properties. The descriptions start with the name of the property followed by the declaration of data type within square brackets, e.g. property_name [type]. Unless otherwise noted, all the properties are of public access.

rotateV [function handle]. Depending on the chosen attitude simulation method, this handle points to either the *quatRotateV* or the *eulerRotateV* methods. This property is private to the RbSimulation class.

omegaEq [function handle]. Depending on the chosen attitude simulation method, this handle points to either the *omegaEqQuat*or or the *omegaEqEuler* methods. This property is private to the RbSimulation class.

iOmega_b [1x3 double]. Initial value in radians per second of the body's angular velocity vector as seen from the BRF. The three columns correspond to the vector's *X*, *Y* and *Z* components. Defaults to [0 0 0].

iVelocity_b [1x3 double]. Initial value in meters per second of the body's linear velocity vector as seen from the BRF. The three columns correspond to the vector's *X*, *Y* and *Z* components. Defaults to [0 0 0].

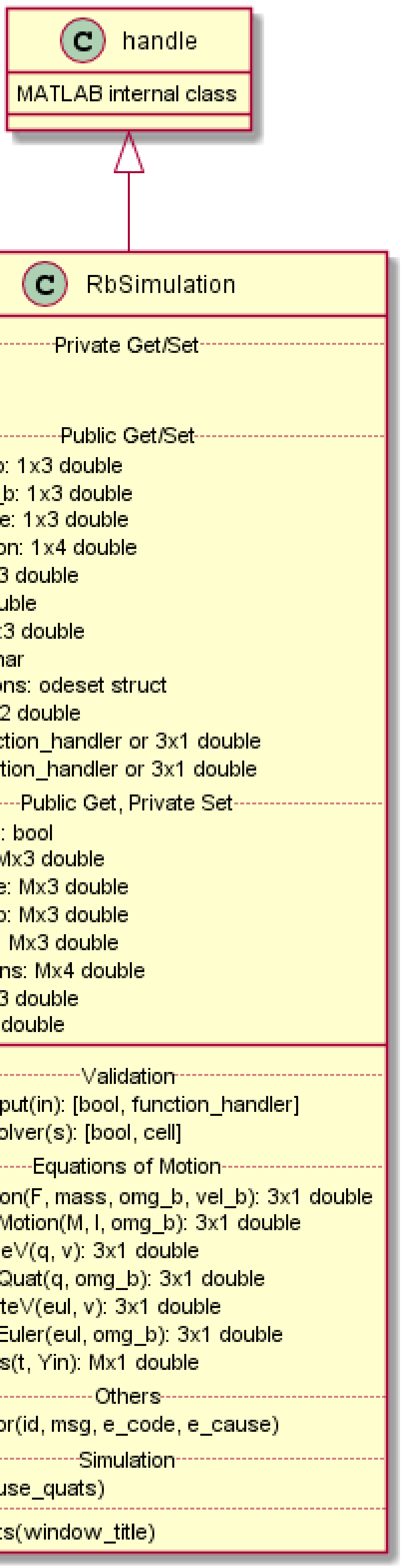


Figure 2-8 Class diagram of the RbSimulation class.

iPosition_e [1x3 double]. Initial value in meters of the body's centre of mass position with respect to the IRF. The three columns correspond to the vector's X, Y and Z components. Defaults to $[0 \quad 0 \quad 0]$.

iQuaternion [1x4 double]. Quaternion that represents the initial attitude of the body. The four columns correspond to the quaternion's $[w \quad i \quad j \quad k]$ components. Defaults to $[1 \quad 0 \quad 0 \quad 0]$. This property is ignored when the selected attitude simulation method is Euler angles.

iEuler [1x3 double]. Vector that represents the initial attitude of the body with Euler angles in radians. The three columns correspond to the $[\phi \quad \theta \quad \psi]$ angles. Defaults to $[0 \quad 0 \quad 0]$. This property is ignored when the selected attitude simulation method is quaternions.

mass [double]. Mass of the body in kg. Defaults to 1.

inertia [3x3 double]. Moment of inertia of the body in $kg\ m^2$. Defaults to a 3x3 identity matrix.

solver [char]. Name of the method chosen to that solve the ordinary differential equations. The available solution methods are those featured in ML. That is: 'ode45', 'ode23', 'ode113', 'ode15s', 'ode23s', 'ode23t', 'ode23tb' or 'ode15i'. Defaults to 'ode23'.

ode_options [struct]. Options structure for the equation solver. Its value must be generated with MATLAB®'s *odeset* function.

tspan [1x2 double]. Time span of the simulation in seconds. The columns correspond to the initial and final time. Defaults to $[0 \quad 1]$.

F_b [3x1 double or function handle]. Force vector in newtons that acts on the body's centre of mass. The property can take the shape of a 3x1 constant double or be a function handle that points to a function that takes one argument (time) and returns a 3x1 double. In both cases, the vector's rows represent the X, Y and Z components of the applied force. The property defaults to the function:

$$F_b(t) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

The possibility of using function handles allows users to use time-changing forces in the simulation.

M_b [3x1 double or function handle]. Torque vector in newtons per meter that acts about the body's centre of mass. The property can take the shape of a 3x1 constant double or be a function handle that points to a function that takes one argument (time) and returns a 3x1 double. In both cases, the vector's rows represent the *X*, *Y* and *Z* components of the applied torque. The property defaults to the function:

$$M_b(t) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

The possibility of using function handles allows users to use time-changing torques in the simulation.

2.3.3.2 Simulation results

This section also describes properties of the class, although in this case the properties are of private set access, which prevents external entities to set their values. Nevertheless, external entities can still read their values.

run_once [boolean]. Flag that indicates if the simulation has run at least once.

position [Nx3 double]. Simulated body's position in meters as seen from the IRF. It is a numeric array with 3 columns for the *X*, *Y* and *Z* components. Each row corresponds to one time step.

velocity_e [Nx3 double]. Simulated body's linear velocity in meters per second as seen from the IRF. It is a numeric array with 3 columns for the *X*, *Y* and *Z* components. Each row corresponds to one time step.

velocity_b [Nx3 double]. Simulated body's linear velocity in meters per second as seen from the BRF. It is a numeric array with 3 columns for the *X*, *Y* and *Z* components. Each row corresponds to one time step.

omega_b [Nx3 double]. Simulated body's angular velocity in radians per second as seen from the BRF. It is a numeric array with 3 columns for the *X*, *Y* and *Z* components. Each row corresponds to one time step.

quaternions [Nx4 double]. Simulated body's attitude in unit quaternions. It is a numeric array with 4 columns for the $[w \quad i \quad j \quad k]$ components. Each row corresponds to one time step. This property is set to $[1 \quad 0 \quad 0 \quad 0]$ when the selected attitude simulation method for the simulation is Euler angles.

euler [Nx3 double]. Simulated body's attitude expressed in Euler angles in radians. It is a numeric array with 3 columns for the $[\phi \quad \theta \quad \psi]$ components. Each row corresponds to one time step. This property is set to $[0 \quad 0 \quad 0]$ when the selected attitude simulation method for the simulation is quaternions.

time [Nx1 double]. Time vector of the simulation in seconds. Each row corresponds to one time step.

2.3.3.3 Methods

The following paragraphs explain the purpose of the class's methods. The descriptions start with the method's name, followed by the argument list and the data type of their return value, if any. For example methodName(arg1, arg2) [type]. Unless otherwise noted, all the methods are private to the RbSimulation class.

validateInput(in) [boolean, function handle]. Validates either an input force vector or an input torque vector, which is passed in the argument. The input vector must either be a 3x1 double or a handle that points to a function that takes one argument (time) and returns a 3x1 double. If either of those two conditions are met, the boolean output is true, and false otherwise. When the argument is a 3x1 double, it is converted to a function handle in the form of:

$$In(t) = \begin{bmatrix} c_x \\ c_y \\ c_z \end{bmatrix}$$

and returned in the second output. The component $\boldsymbol{c_i}$ is a double. When the argument is already a function handle, it is returned as is in the second output. If none of the conditions are met, the second output returns a function handle in the form of

$$f(t) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

validateSolver(s) [boolean, cell]. Validates the selected solver function. A solver is valid when the argument is a string equal to one of the following options: 'ode45', 'ode23', 'ode113', 'ode15s', 'ode23s', 'ode23t', or 'ode23tb'.

The boolean output is true if the string is indeed one of the possible options and false when it is not. The second output is a cell that contains the names of the possible solvers meant to assist the user to correct the error.

linearMotion(F, mass, omg_b, vel_b) [3x1 double]. Implementation of Eq. 1.12 in ML. The arguments are the force applied to the body's centre of mass in newtons, the mass of the body in kilograms, the body angular velocity in radians per second, and the linear velocity in meters per second as seen from the BRF. With the exception of the mass, which must be a double, all the arguments must be 3x1 doubles where the rows are the *X, Y* and *Z* components.

rotationalMotion(M, I, omg_b) [3x1 double]. Implementation of Eq. 1.17 in ML. The arguments are the torque applied about the body's centre of mass in newtons per meter, the moment of inertia of the body in $kg\,m^2$ and the body angular velocity in radians. With the exception of the moment of inertia, which must be a 3x3 symmetrical numeric matrix as in Eq. 1.13, all the arguments must be 3x1 doubles where the rows are the *X, Y* and *Z* components.

quatRotateV(q, v) [3x1 double]. Implementation of the inverse of Eq. 1.23 in ML. More information can be found in section 2.2.4.1.

The first argument must be a unit quaternion as a column vector and the second argument is the vector to be transformed, which must be a 3x1 double.

omegaEqQuat(q, omg_b) [4x1 double]. Implementation of Eq. 1.18 in ML.

The first argument must be a unit quaternion as a column vector and the second argument is the body angular velocity, which must be a 3x1 double.

eulerRotateV(eul, v) [3x1 double]. Implementation of rotations using .Eq. 1.25 in ML.

The first argument is a 3x1 vector with Euler angles where the rows, respectively, are the values for $[\phi \quad \theta \quad \psi]$. The second argument is the 3x1 vector that will be rotated. More information about how the rotation works can be found in section 2.2.4.2.

omegaEqEul(eul, omg_b) [3x1 double]. Implementation of Eq. 1.21 in ML.

The first argument must be a 3x1 vector with Euler angles where the rows, respectively, are the values for $[\phi \quad \theta \quad \psi]$. The second argument is the body angular velocity, which must be a 3x1 double.

concatEqs(t, Yin) [Mx1 double]. Implementation Eq. 1.26 in ML. Having all the equations of motion together, this method is the mathematical model of the system being simulated. The method's first argument is the moment in time at which the equations will be solved and the second argument is a vector with all the system's states.

The simulation is run by passing to the chosen solver a handle to this function, along with the time span and the initial conditions.

simulate(use_quats) [returns nothing]. Runs the simulation.

When the argument's value is true, the simulation uses quaternions as the attitude simulation method. When false, Euler angles are used. Defaults to true.

This method calls the chosen solver and passes as arguments a function handle to the *concatEqs* method, the time span vector, and an array with the initial conditions.When the solver returns, the results are separated and assigned to the simulation-result properties, and after this, the *run_once* flag is set to true.

plotResults(window_title) [returns nothing]. Plots the simulation results against time in a figure where the title is equal to the method's argument.

This method will only run if the *run_once* flag is set. Which means that the simulation must run at least once before plotting.

The results in the plot are position, linear velocity as seen from the IRF, linear velocity as seen from the BRF and the body angular velocity as seen from the BRF.

throwError(id, msg, e_code, e_cause) [returns nothing]. Throws an instance of the *RbException* class.

The exception's identifier is given by the first argument and it is appended to "Simulation:" to make simulation related errors easier to identify.

The error message is given by the fourth argument. The third and fourth arguments respectively add an error code and a cause to the exception. This method is private to the RbSimulation class.

2.4 Demonstrations

In this section the validity and accuracy of the RbSimulation class to simulate relatively simple rigid body motion is established. That is, the numerical results of the simulation of simple and well-known rigid body motions are compared against their analytical counterparts.

2.4.1 Free precession of a symmetrical top

The motion of a symmetrical top is a common problem in mechanics to demonstrate the effect of symmetry about the axis of rotation of a top. For this problem, a disc-shaped ellipsoid was chosen as the rotating top, Figure 2-9 shows its geometry and an orthogonal arrow-set to serve as an IRF. The red, green, and blue arrows are respectively collinear with the X, Y, and Z axes.

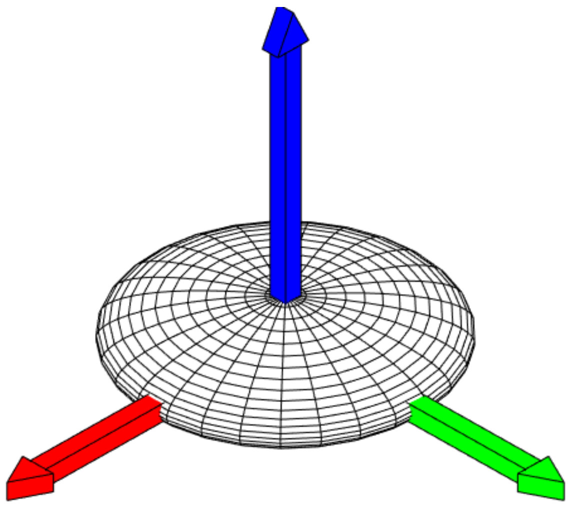


Figure 2-9 Ellipsoid for which the problem is solved.

The equations of interest are the Euler's equation of rotational motion, so in order to simplify their manual solution, appropriate values for the momens of inertia must be chosen. This assumption renders the dimensions of the disc as dependant variables. Therefore, and considering the symmetry, it is possible to choose a value of $I_x = I_y = 1kg\ m^2$ and mass of 1kg. Then, choosing a radius four times as large as its height to form a disc, the equations for the moment of inertia from Weinsstein [36] can be rearranged to solve for the height in meters

$$I_x = m * \frac{(R^2 + h^2)}{5} = m * \frac{((4h)^2 + h^2)}{5} = m * \frac{(17h^2)}{5} \rightarrow h = \sqrt{\frac{5I_x}{17m}} = 0.5423$$

In a similar way, the value of I_z in $kg\ m^2$ is:

$$I_z = m * \frac{(R^2 + R^2)}{5} = m * \frac{2 * (4h)^2}{5} = 1.8822$$

2.4.1.1 *Analytical Solution*

This analytical solution follows the steps taken by Landau & Lifshitz [37]. The idealisation of a symmetrical top's free precession results in two simplifying conditions: there are no disturbances (torques or forces) applied to the body, and the equality of the first two components of the moment of inertia $I_x = I_y$ due to the symmetry. Moreover, the description in [37] shows that the motion of the angular velocity vector ω forms two different conical shapes, one that is observed with respect to the IRF and the other one with respect to the BRF. Those two cones are called space cone and body cone respectively, and they served in this problem as a visual verification of the simulator's accuracy.

Given these assumptions, and writing Eq. 1.17 in component notation, the equations of rotational motion are reduced to:

$$\begin{aligned} I_x\dot{\omega}_x + (I_z - I_x)\omega_z\omega_y &= 0 \\ I_x\dot{\omega}_y - (I_z - I_x)\omega_z\omega_x &= 0 \\ \dot{\omega}_z &= 0 \end{aligned} \qquad \text{Eq. 1.27}$$

From Eq. 1.27 it is clearly identified that ω_z is constant with respect to time and a new constant angular velocity ω_k can be defined as:

$$\omega_k = \frac{I_z - I_x}{I_x}\omega_z \qquad \text{Eq. 1.28}$$

Where a substitution of Eq. 1.28 in Eq. 1.27 results in:

$$\begin{aligned} \dot{\omega}_x &= -\omega_k\omega_y \\ \dot{\omega}_y &= \omega_k\omega_x \end{aligned} \qquad \text{Eq. 1.29}$$

Substitution of the time derivative of the first equation in Eq. 1.29 into the second equation yields:

$$\ddot{\omega}_x = -\omega_k^2\omega_x \qquad \text{Eq. 1.30}$$

After solving Eq. 1.30, ω_x and ω_y are found as:

$$\omega_x(t) = A * cos\left(\frac{I_z - I_x}{I_x}\omega_z t\right)$$
$$\omega_y(t) = A * sin\left(\frac{I_z - I_x}{I_x}\omega_z t\right) \qquad \text{Eq. 1.31}$$

Eq. 1.31 clearly shows that the periods of both functions are equal, so a single variable can be used to represent both: $\tau = \tau_{\omega_x} = \tau_{\omega_y} = \frac{I_x}{(I_z - I_x)\omega_z}$. The next step is to assign values so that the motion can be plotted. Therefore, a value of $\tau = 5s$ is chosen so that the angular velocity vector ω_b rotates slow enough to be perceived. The 5 second period makes $\omega_k = \frac{2\pi}{5s}$. So Eq. 1.28 can be rearranged to find ω_z:

$$\omega_z = \frac{I_x \omega_k}{I_z - I_x} = 1.4243\frac{1}{s}$$

Now, choosing an amplitude of $A = 1$, also to keep the calculations simple, the solution's equations in Eq. 1.31 can be evaluated using the previous results:

$$\omega_x(t) = cos\left(\frac{2\pi}{5s}t\right)$$
$$\omega_y(t) = sin\left(\frac{2\pi}{5s}t\right) \qquad \text{Eq. 1.32}$$
$$\omega_z = 1.4243\frac{1}{s}$$

The next step is to transform the resulting angular velocity from the BRF to the IRF. To do this, Landau & Lifshitz [37] align the *Z* axis of the IRF with the angular momentum vector H by rotating it by an angle θ around the *Y* axis. In this demonstration, the aligned IRF will be called the *A* reference frame. As Figure 2-10 shows, with the original configuration where the IRF and BRF are parallel, the angle θ changes depending on the instantaneous value of the angular velocity vector ω, but when the *Z* axis of the IRF is aligned with the angular momentum vector H, the Euler angle θ becomes constant, which means that $\dot{\theta} = 0$. So the transformation's differential equation system is simplified, and the remaining two Euler angle rates become:

$$\dot{\phi} = \frac{H}{I_x} = \frac{\|\omega_b * I\|}{I_x} = 2.8615\frac{1}{sec}$$
$$\dot{\psi} = -\omega_k = -1.2568\frac{1}{sec} \qquad \text{Eq. 1.33}$$

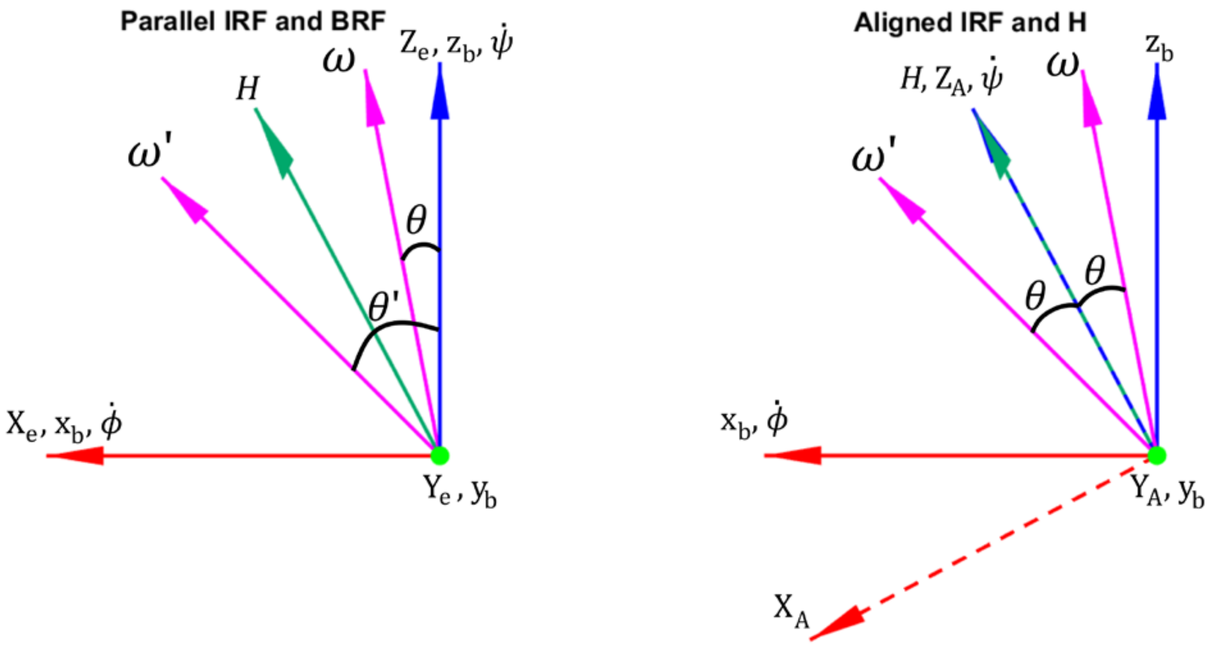


Figure 2-10 Diagrams of the parallel inertial and body reference frames (left), and the Z axis of the IRF aligned with the angular momentum vector (right).

The angles can then be obtained by an integration of Eq. 1.33

$$\phi(t) = \int \dot{\phi}\, dt = 2.8615\frac{t}{sec}$$
$$\psi(t) = \int \dot{\psi}\, dt = -1.2568\frac{t}{sec} \qquad \text{Eq. 1.34}$$

The angle θ is the angle between the angular momentum vector H and the Z axis of the BRF, so it can therefore be calculated by rearranging the terms of the dot product equation:

$$H_b \cdot z_b = \|H_b\|\|z_b\|cos\theta \rightarrow \theta = \cos^{-1}\frac{H_b \cdot z_b}{\|H_b\|\|z_b\|} = \cos^{-1}\frac{\omega_z I_z}{\|\omega_b * I\|}$$

Since the alignment of the IRF with the angular moment vector required a positive θ rotation, a rotation in the opposite direction is required to represent an IRF-to-BRF transformation. Therefore θ becomes:

$$\theta(t) = -\cos^{-1}\frac{\omega_z I_z}{\|\omega_b * I\|} = -0.3570$$

Using the angles as they are at this point would transform the body angular velocity to the *A* reference frame, which is different from the IRF used by the simulator. Then, one last rotation around the *Y* axis by θ is necessary to return to the original IRF.

Considering then a transformation matrix $[R_b^A]$ as,

$$[R_b^A] = \begin{pmatrix} -cos\psi & -sin\psi & 0 \\ sin\psi & -cos\psi & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} -\cos\theta & 0 & sin\theta \\ 0 & 1 & 0 \\ -sin\theta & 0 & -\cos\theta \end{pmatrix} \begin{pmatrix} -cos\phi & -sin\phi & 0 \\ sin\phi & -cos\phi & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

and the transformation matrix $[R_A^e]$ as

$$[R_A^e] = \begin{pmatrix} \cos\theta & 0 & -sin\theta \\ 0 & 1 & 0 \\ sin\theta & 0 & \cos\theta \end{pmatrix}$$

where the subscript *A* identifies the *A* reference frame, the angular velocity in the original IRF is given by the equation:

$$\omega_e = [R_e^A][R_A^b]\omega_b$$

Figure 2-11 shows from left to right the results of the body angular velocity with respect to the BRF, from the *A* reference frame, and finally on the right side, with respect to the IRF.

As mentioned before, the motion of the angular velocity vector ω forms a space cone when observed with respect to the IRF, and a body cone when observed with respect to the BRF. The resulting cones for this problem can be seen in Figure 2-12, which also shows the space cone formed in the reference frame *A*. The vectors were plotted with the Arrow3 Version 5 function by [38]**.**

Finally, the magnitude of the angular momentum vector H was plotted in Figure 2-13 to verify the conservation of angular momentum, which as shown in the graph, is true.

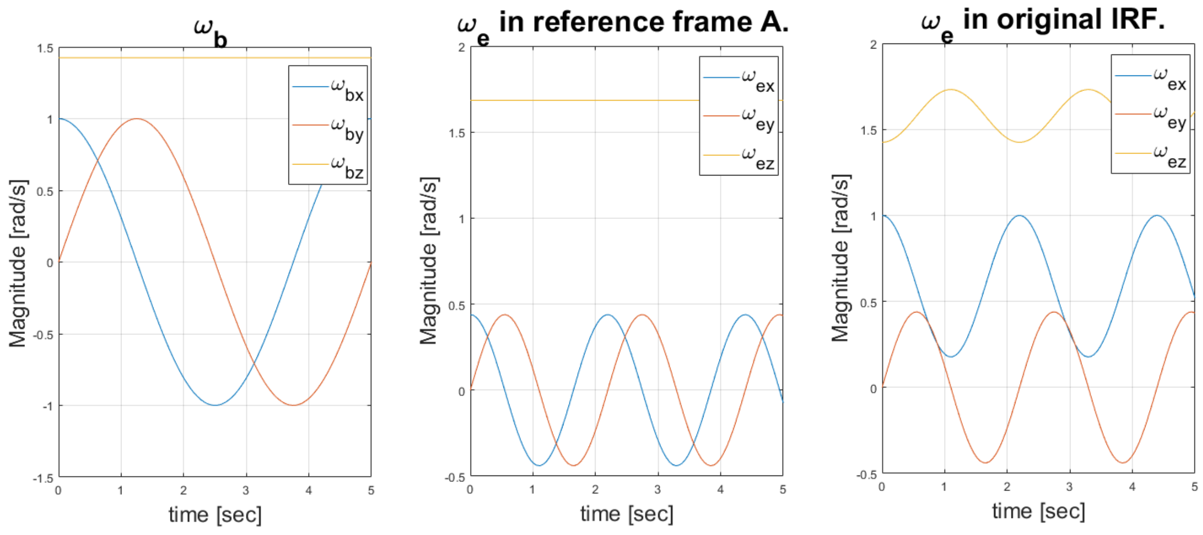


Figure 2-11 Free precession of an ellipsoidal disc. Analytical solution. Angular velocity results.

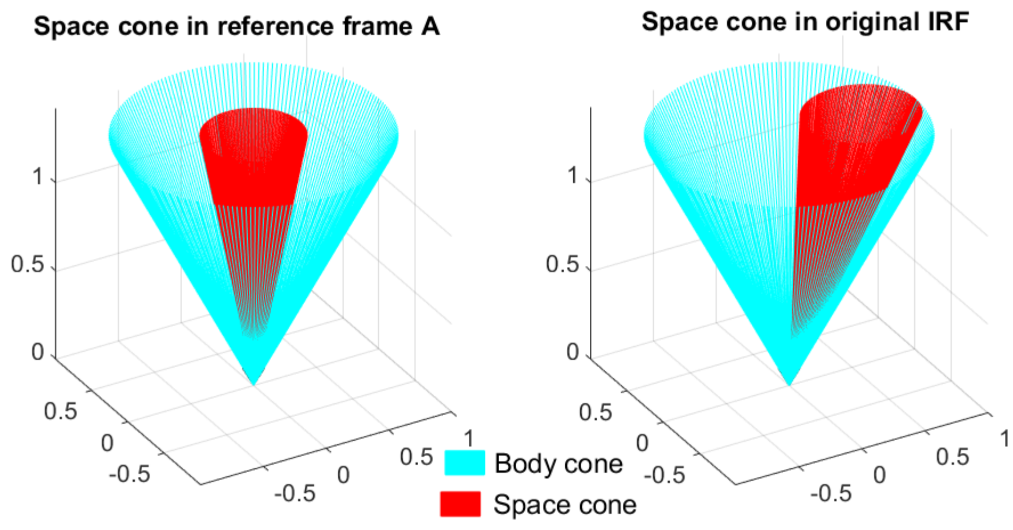


Figure 2-12 Body and space cones formed by the angular velocity vector as it rotates. The body cone (cyan) is seen from the BRF and the space cone (red) is seen from the IRF. The units of the axes are [rad/s].

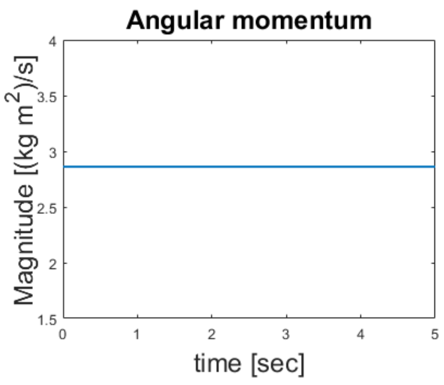


Figure 2-13 Magnitude of the angular momentum vector H with respect to time.

2.4.1.1 Simulation

Two simulations run in Script 2-1, one with quaternions and one with Euler angles. Figure 2-14 shows the results of both superimposed on the analytical solution. The superimposed simulation solution (circle markers) has the same shape as the analytical solution (continuos line), thus validating the correct operation of both attitude simulation methods. Additionally, Figure 2-15 shows the cones resulting from the simulation to visualy verify the correct operation of the simulator.

The reason for both attitude simulation methods to run is the verification of their accurate simulations. It must be noticed that the difference in attitude simulations will only be visible on the IRF, because the difference lies in the way the transformation is generated. In other words, no matter what attitude simulation method was chosen, v^b and ω^b will always be the same. Where as r^e and v^e will have different results depending on the chosen attitude simulation method.

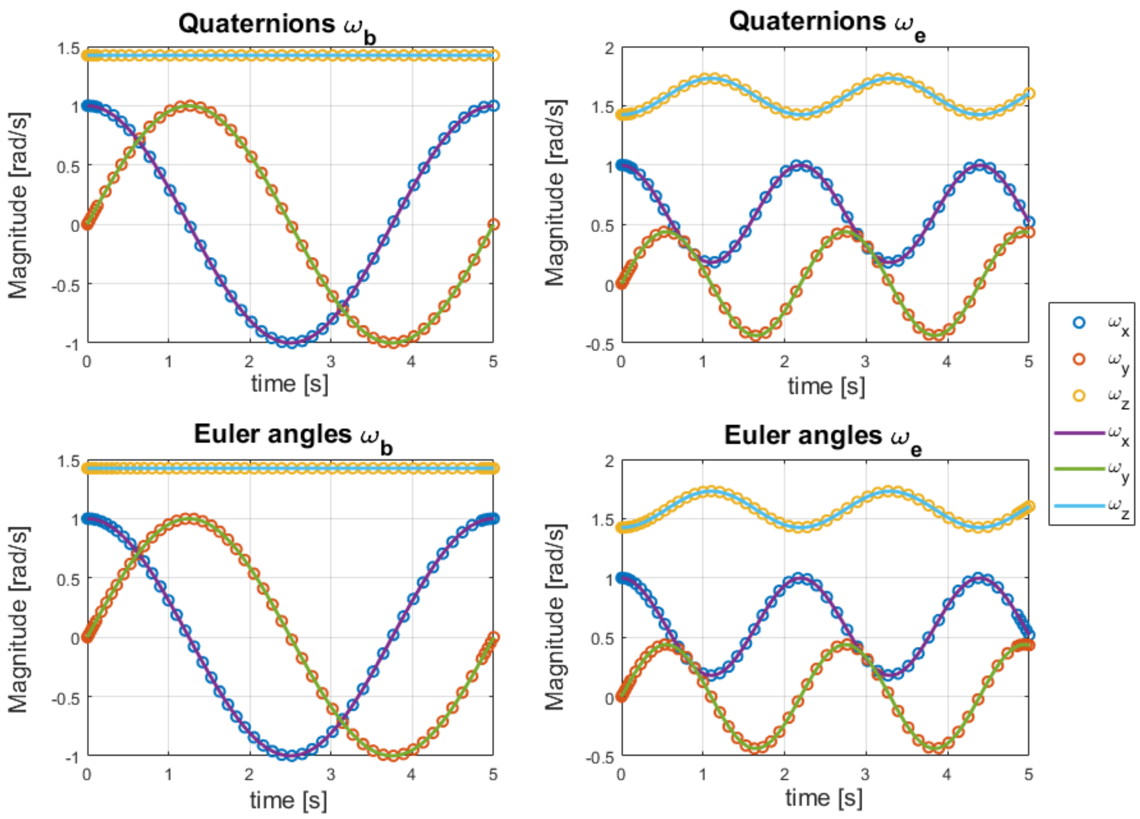


Figure 2-14 Free precession's simulation. Angular velocity results. The dotted lines are the simulation results and the continuous lines are the analytical results.

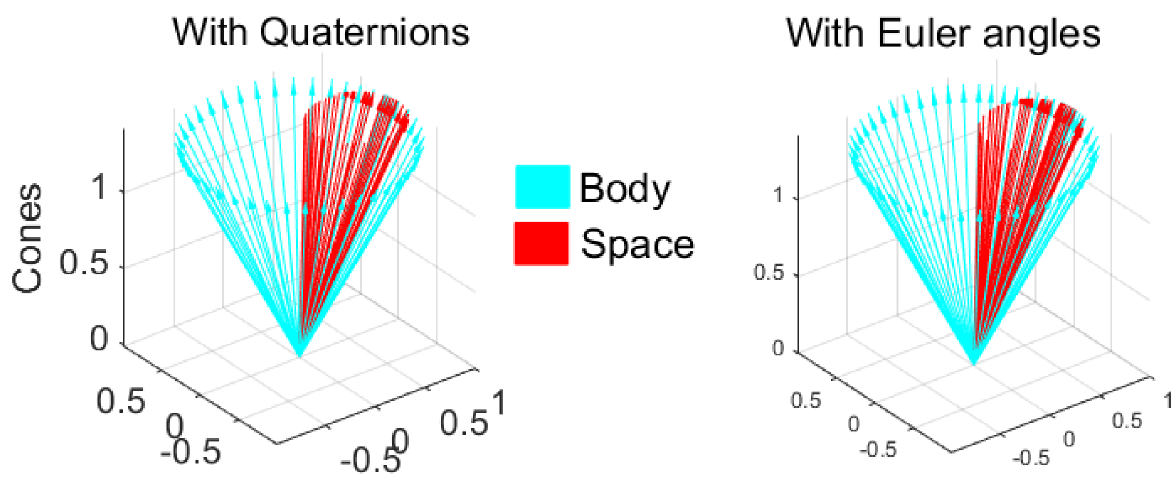


Figure 2-15 Free precession's simulation. Angular velocity cones in the refernece frame A and in the IRF.

2.4.2 Free rotation of an asymmetrical top

2.4.2.1 Theoretical basis

The free rotation of a relatively simple asymmetrical top is now considered. In his description of the problem, Tatum [39] states that the magnitude and direction of the angular velocity vector are mainly constrained by two conditions. The first one being that the angular momentum is constant in magnitude, and the second condition that the kinetic energy is conserved. Therefore, the angular momentum and the kinetic energy are respectively, at all times

$$I_1^2\omega_1^2 + I_2^2\omega_2^2 + I_3^2\omega_3^2 = H^2$$

and,

$$\frac{1}{2}I_1\omega_1^2 + \frac{1}{2}I_2\omega_2^2 + \frac{1}{2}I_3\omega_3^2 = E_k$$

Both equations can be rearranged to become ellipsoid equations:

$$\frac{\omega_1^2}{(H/I_1)^2} + \frac{\omega_2^2}{(H/I_2)^2} + \frac{\omega_3^2}{(H/I_3)^2} = 1 \qquad \text{Eq. 1.35}$$

and,

$$\frac{\omega_1^2}{(\sqrt{2E_k/I_1})^2} + \frac{\omega_2^2}{(\sqrt{2E_k/I_2})^2} + \frac{\omega_3^2}{(\sqrt{2E_k/I_3})^2} = 1 \qquad \text{Eq. 1.36}$$

Where the semi-axes are, respectively,

$$a_H = \frac{H}{I_1} \qquad b_H = \frac{H}{I_2} \qquad c_H = \frac{H}{I_3}$$

And,

$$a_E = \sqrt{2\frac{E_k}{I_1}} \qquad b_E = \sqrt{2\frac{E_k}{I_2}} \qquad c_E = \sqrt{2\frac{E_k}{I_3}}$$

The H subscript refers to the angular momentum ellipsoid and the E subscript to the kinetic energy ellipsoid.

The pair of ellipsoid equations constrain the terminus of the angular velocity to always be where the two ellipsoids intersect, and the intersection is ensured by the inequalities

$$2E_kI_1 < H^2 < 2E_kI_3$$

Up to this point, every mentioned vector is in the BRF. Landau & Lifshitz [37] prove analytically that the angular velocity vector, although periodic in the BRF, never returns to its original position when seen from the IRF. This demonstration does not show such solution due to its mathematical complexity. However, it can be found in chapter 37 of [37].

In conclusion, and with the mentioned premises, the correct operation of the simulator can be confirmed if:

1. The locus of points of the resulting ω^b lies on one of the intersection curves,
2. ω^e is not a periodic function.

2.4.2.2 Calculating the initial values

To have a demonstration closer to an actual application, a 3-unit CubeSat with deployed solar panels is used as an asymmetrical top. It is important to notice that the satellite is only an illustrative representation, its behaviour does not intend to model how a real satellite moves in orbit. The solar panels are implemented to make the moment of inertia asymmetrical. Without the panels, two of the principal moments of inertia of the satellite, being a body with square faces, would be equal and thus symmetrical. Figure 2-16 shows a model of the satellite being analysed.

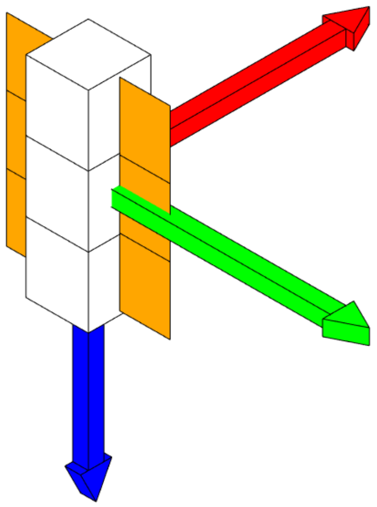


Figure 2-16 3U CubeSat.

The three cubic units are considered as one single rectangular prism with dimensions 10cm x 10cm x 34cm and together, they have an assumed mass of 3kg. These values and the orientation as shown in Figure 2-16, comply with the CubeSat standard [40].

The solar panels were chosen from the catalogue of DHV Technology™ and its dimensions and mass are 1.6mm x 82mm x 329mm and 132g respectively.

The calculation of the moment of inertia can be divided in two steps: calculating the principal moments of inertia of all the bodies individually and then using the parallel axis theorem to add them together. The formulas to calculate the principal moments of inertia of a rectangular prism are:

$$I_x = \frac{M}{12}(b^2 + c^2) \qquad \text{Eq. 1.37}$$

$$I_y = \frac{M}{12}(a^2 + c^2) \qquad \text{Eq. 1.38}$$

$$I_x = \frac{M}{12}(a^2 + b^2) \qquad \text{Eq. 1.39}$$

Where a, b and c are the dimensions of the prism along X, Y and Z respectively. The equations are used for both the cubic units and the solar cells.

Since the solar cells are located on the YZ plane and centred on the Y axis, their centres of mass, as well as the satellite's body, are located on the Y axis. So, in conclusion, the CoM of the entire system is the CoM of the satellite's body. Consequently, the parallel axis theorem only needs to be applied to the moments of inertia of the solar cells and the final moment of inertia will then be:

$$I_T = I_c + I_p + 2m_p d$$

Where I_c is the moment of inertia of the cubic units, I_p is the moment of inertia of each solar panel, m_p is the mass of one solar panel and d is the distance from the centre of mass of one solar panel to the centre of mass of the satellite's body. After converting all the units to MKS and using the values in the formulas, the total moment of inertia results in:

$$I_T = \begin{pmatrix} 0.0361 & 0 & 0 \\ 0 & 0.0338 & 0 \\ 0 & 0 & 0.0073 \end{pmatrix}$$

Next, values for the angular momentum and kinetic energy must be chosen. One of the examples from [39] uses $E_k = 20J$ and $H = 4Js$. However, only the value for the kinetic energy was taken from the example because with the satellite's moment of inertia, using the value for the angular momentum from the example would not satisfy the inequality. So the new value for the angular momentum is found by first assigning values to the inequality. It must first be noted that in this demonstration the longest dimension of the satellite is aligned to the Z axis, so the inequality must be rearranged into:

$$2E_k I_1 > H^2 > 2E_k I_3$$

Which, after replacing the values for the kinetic energy and the moments of inertia results in

$$1.44J^2s^2 > H^2 > 0.29J^2s^2$$

So, to satisfy the inequalities, a value of $H = 1Js$ was chosen.

With the intersection guaranteed, the only thing missing to run the simulation is an initial value for the angular velocity. As in the examples by Tatum [39], the initial magnitude of the X component of the angular velocity is assumed to be $\omega_{x_0} = 0$. Reducing Eq. 1.35 and Eq. 1.36 to a system of two equations with two unknowns whose solution is:

$$\omega_0 = [0 \quad 24.04 \quad 52.81]\frac{rad}{s}$$

2.4.2.3 Simulation

The values obtained in the previous section are used in a quaternion simulation for one second in Script 2-2, which plots the ellipsoids and the angular velocity that results from the simulation.

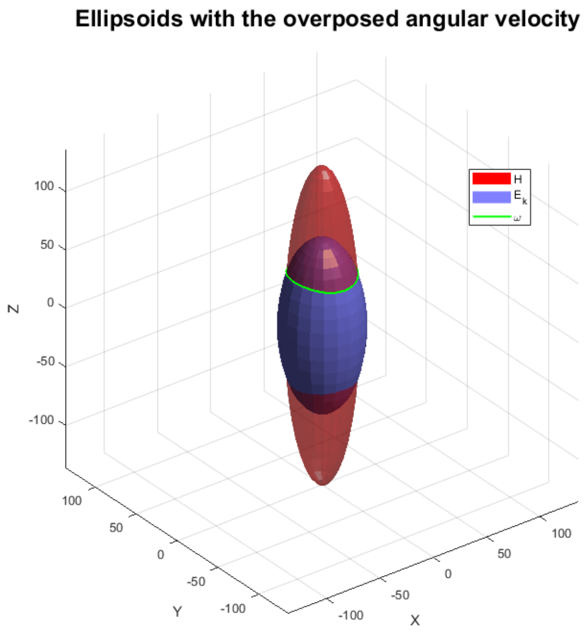


Figure 2-17 Angular momentum and kinetic energy ellipsoids with the added angular velocity.

As seen in Figure 2-17, the locus of points of the simulated angular velocity lies on one of the intersection curves, proving the first of the validation premises. Section 3.3.2.2 has a frame strip that illustrates the rotation of the asymmetrical top in a stop-motion approach. Next, to confirm that the angular velocity is not periodic in the IRF, the same simulation is run for 1, 10 and 100 seconds. The resulting angular velocities are transformed to the IRF and plotted in Figure 2-18, which shows that as time advances, the space in which the angular velocity moves becomes saturated, meaning that it does not close any cycle and proving the second validation premise. The procedures of this demonstration thus conclude that the simulator works as expected.

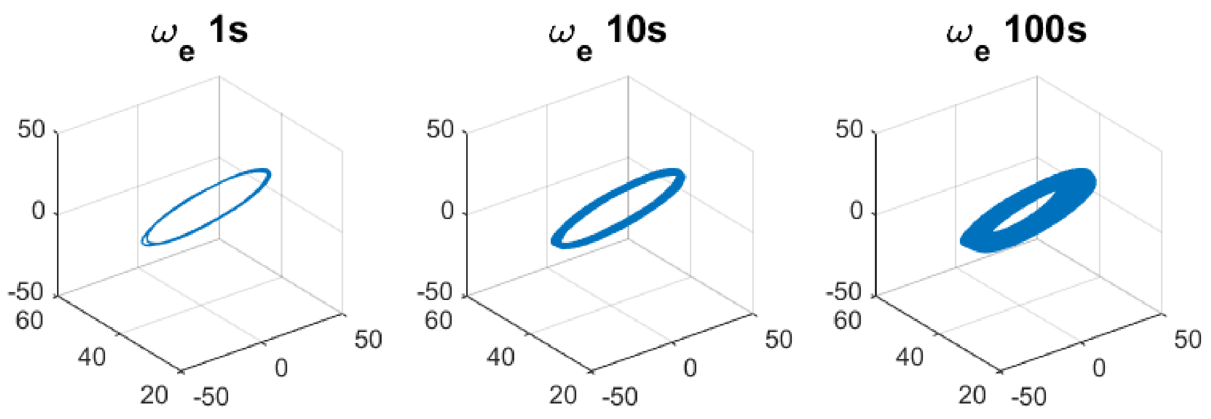


Figure 2-18 Motion of the angular velocity in [rad/s] with respect to the IRF.

2.4.3 Freefall

2.4.3.1 Analytical Solution

The free fall problem is solved with the equations of linear motion found in [32], where r stands for the position of the body or particle of interest, v for velocity, a for acceleration, and t for time:

$$2a(r_2 - r_1) = v_2^2 - v_1^2 \qquad \text{Eq. 1.40}$$

$$r_2 - r_1 = v_1 t + \frac{a}{2}t^2 \qquad \text{Eq. 1.41}$$

In a situation where a 70kg-skydiver leaves his aeroplane at 4km above the ground, the equations can be used to get the final velocity and the time the skydiver takes to reach the ground. The presented solution considers only constant forces and therefore ignores the drag force. The equation of the position as a function of time is a rearrangement of Eq. 1.41 in the form of:

$$r_2 = r_1 + v_1 t + \frac{a}{2}t^2 \rightarrow r_2 = 4000m - \frac{9.8}{2}\frac{m}{s^2}t^2 \qquad \text{Eq. 1.42}$$

Using then Eq. 1.42 in Eq. 1.40 yields the equation for the final velocity as:

$$v_2 = -9.8\frac{m}{s^2}t \qquad \text{Eq. 1.43}$$

Afterwards, by solving Eq. 1.41 for t when $r_2 = 0$, the time at which the skydiver touches the ground is:

$$0 - r_2 = 0t - \frac{a}{2}t^2 \rightarrow t = \sqrt{\frac{2r_2}{a}} = 28.55s$$

The resulting time is then used in Eq. 1.42 and Eq. 1.43 to plot the motion of the skydiver, which is shown in Figure 2-19.

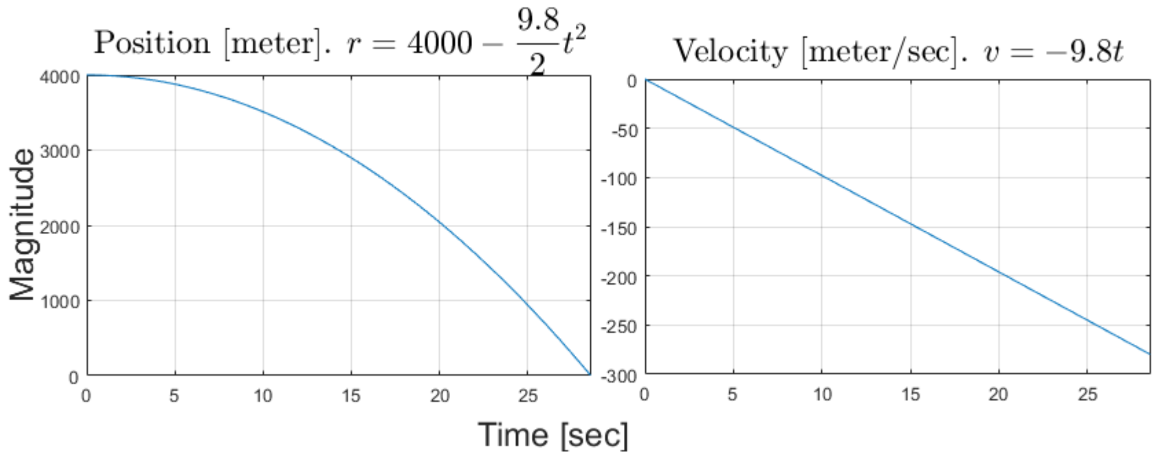


Figure 2-19 Free Fall. Analytical results.

2.4.3.2 Simulation

In Script 2-3, the values from the analytical solution were used in all three dimensions to test the independence of the vector components within the simulator. Figure 2-20 shows the simulation results, which as expected, are the same in all three axes because the forces acting on each axis are not related to one another. Figure 2-20 also shows that the velocity is the same in both reference frames. This equality is caused by the fact that there is no rotation, i.e. $\omega = 0$. Since the results for all three axes is the same, it is enough to superimpose any one of them over the analytical solution to visually validate the simulations results. The superimposition is shown in Figure 2-21, which demonstrates that the behaviour of the position and the velocity is the same in both solutions, thus proving that the simulator works as expected.

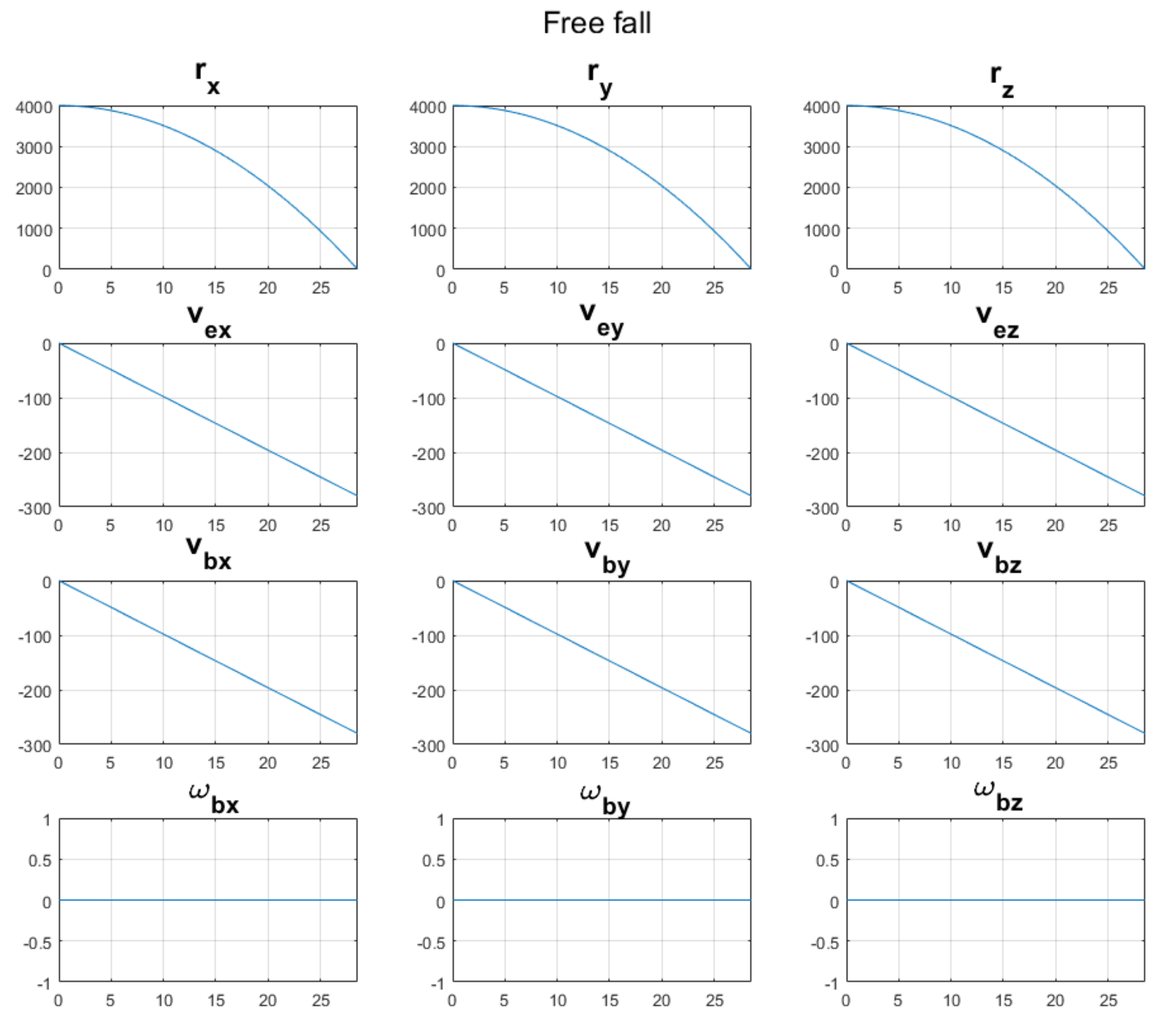


Figure 2-20 Free Fall. Simulation results.

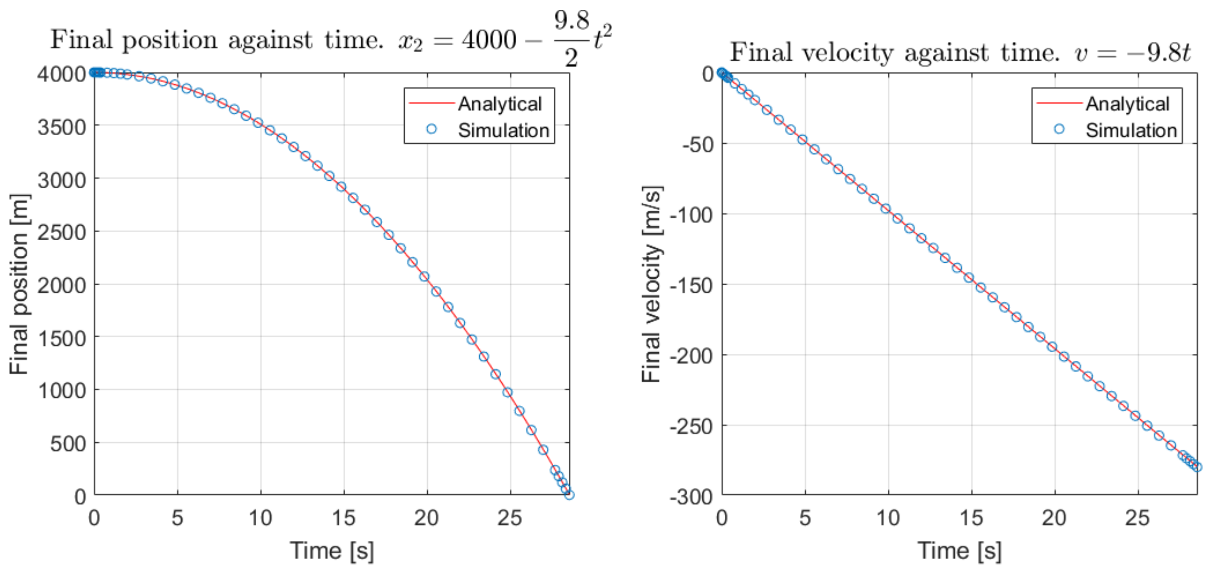


Figure 2-21 Free Fall. Simulation results (circle markers) superimposed over the analytical solution (continuous lines).

2.4.4 Parabolic Motion

2.4.4.1 Analytical Solution

This problem considers a football being kicked with an angle with respect to the ground so that its motion describes a parabola. For the purposes of this thesis, the football is considered a non-deformable body, and the drag forces acting on the ball are ignored.

In the experiment of Nunome, Takeshi, Yasuo, & Shinji [41], a football player kicks a football and applies an initial velocity of $v_1 = 28\frac{m}{s}$. With these asusmptions, Eq. 1.40 and Eq. 1.41 can tell how long and how high the ball will travel. Assuming that the football player kicks the ball at an angle of 45° and that the trajectory stays on the YZ plane, then the Y and Z components are the horizontal and vertical axes respectively and the variables of interest are:

$$r_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad r_2 = \begin{bmatrix} 0 \\ y_2 \\ 0 \end{bmatrix} \qquad v_1 = \begin{bmatrix} 0 \\ 28\cos(45°) \\ 28\sin(45°) \end{bmatrix}\frac{m}{s} \qquad a = \begin{bmatrix} 0 \\ 0 \\ -9.8 \end{bmatrix}\frac{m}{s^2}$$

Replacing the known values in Eq. 1.40 yields the equations that describe the behaviour of the ball's final position:

$$\begin{bmatrix} 0 \\ y_2 \\ z_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 28\cos(45°) \\ 28\sin(45°) \end{bmatrix}\frac{m}{s}t + \begin{bmatrix} 0 \\ 0 \\ -\frac{9.8}{2} \end{bmatrix}\frac{m}{s^2}t^2 \qquad \text{Eq. 1.44}$$

In the same manner, using the initial values in Eq. 1.41 yields a function of time for the final velocity as a system of two equations:

$$\begin{bmatrix} 0 \\ v_{y2} \\ v_{z2} \end{bmatrix} = \begin{bmatrix} 0 \\ 28\cos(45°) \\ 28\sin(45°) \end{bmatrix}\frac{m}{s} + \begin{bmatrix} 0 \\ 0 \\ -9.8 \end{bmatrix}\frac{m}{s^2}t \qquad \text{Eq. 1.45}$$

The equation for the Z component in Eq. 1.44 can be rearranged to solve for the time at which the ball is back on the ground:

$$t\left(28\sin(45°)\frac{m}{s} - \frac{9.8}{2}\frac{m}{s^2}t\right) = 0 \rightarrow t = 2 * \frac{28\sin(45°)}{9.8}\frac{m}{s} = 4.04s$$

The resulting time can then be used in Eq. 1.44 and Eq. 1.45 to plot the motion of the ball, which is shown in Figure 2-22.

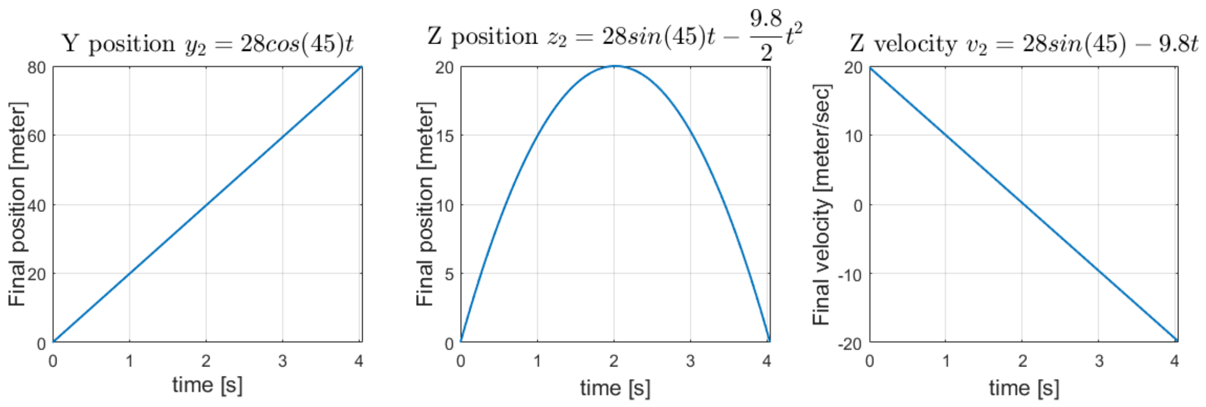


Figure 2-22 Parabolic motion. Analytical results.

2.4.4.2 Simulation

In Script 2-4, the values from the analytical solution for the parabolic motion problem were used in a simulation and superimposed over the analytical results to validate the correct operation of the simulator. The results are shown in Figure 2-23. Next, Figure 2-24 shows only the results of the simulator, which as in the free fall problem, shows that the linear velocity is the same in both the IRF and BRF because there is no rotation. It is then possible to conclude that the behaviour of the position and the velocity is the same, which confirms the correct operation of the simulator.

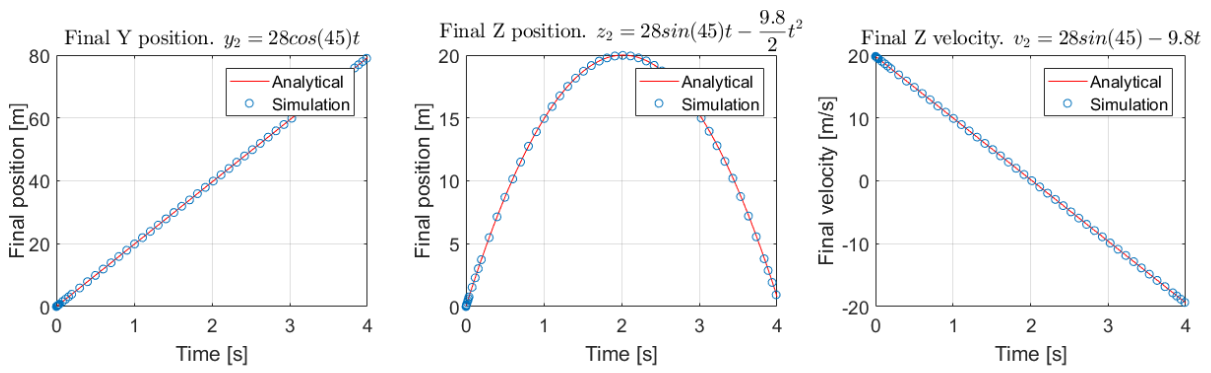


Figure 2-23 Parabolic motion. Superimposition of the simulation results (circle markers) over the analytical solution (continuous lines).

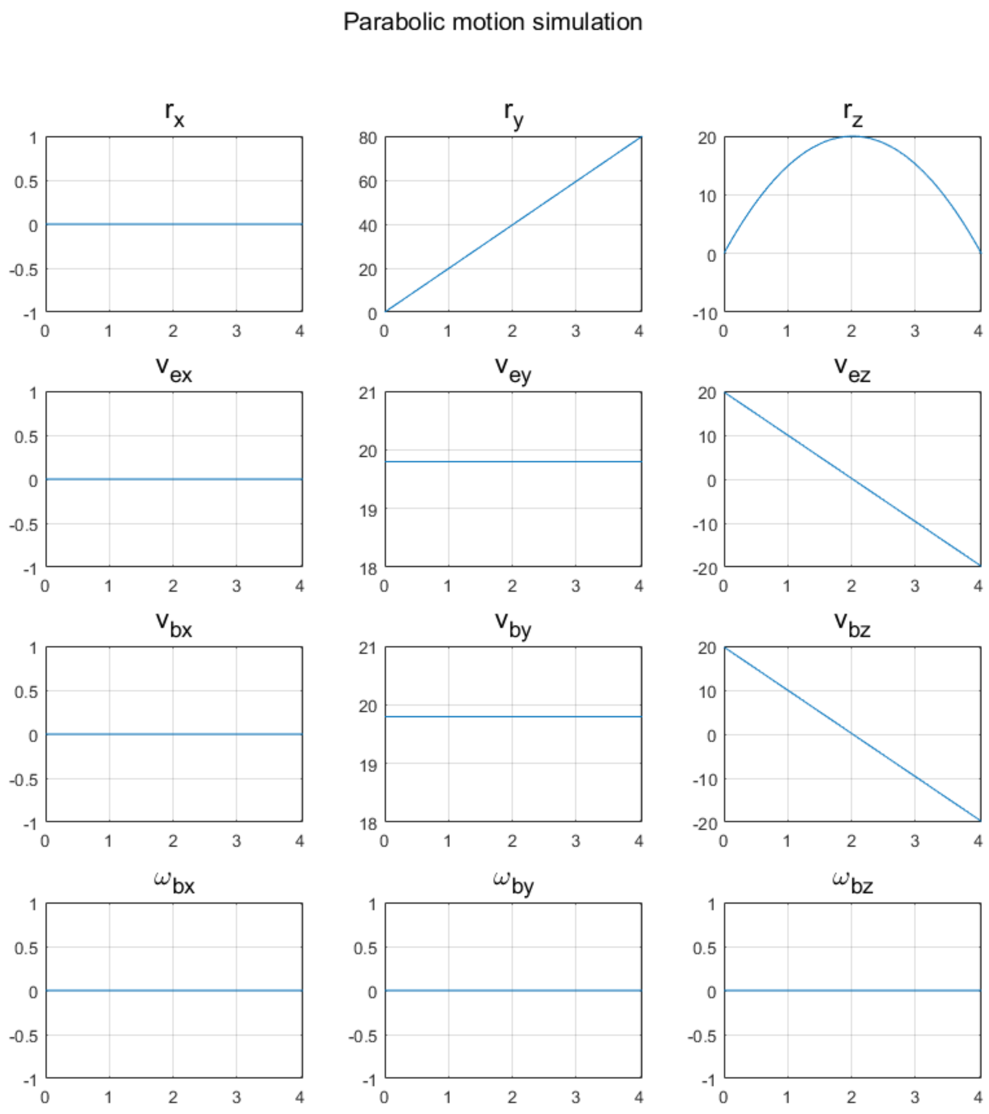


Figure 2-24 Parabolic motion. Simulation results.

2.5 Benchmarking

The accuracy of the ODE45, ODE23, ODE113, ODE15s, ODE23s, ODE23t, and ODE23tb integrators was tested to decide for a default solver for the RbSimulation class. All the integrators were run at 0.001 seconds for the Free Precession of a Symmetrical Top problem (section 2.4.1), the Free Fall problem (section 2.4.3), and the Parabolic Motion problem (section 2.4.4). The problem of the Free Rotation of an Asymmetrical Top was not included because of the lack of an analytical solution to compare the simulation against. The time step of 0.001 seconds was chosen due to the relatively long time the integrators take to solve the equations. Having a long time makes it easy to see which attitude simulation method runs faster. The results of the tests can be respectively seen in Table 1, Table 2, and Table 3, which are tables sorted from the smallest to the largest error.

Solver	Quaternions			Euler angles		
	$\boldsymbol{\omega^b}$	$\boldsymbol{\omega^e}$	**time**	$\boldsymbol{\omega^b}$	$\boldsymbol{\omega^e}$	**time**
ode45	2.60E-31	1.81E-30	5.40E+00	2.60E-31	1.09E-29	4.68E+00
ode23	3.00E-20	1.56E-20	2.58E+00	3.00E-20	1.14E-19	2.29E+00
ode113	3.65E-20	6.31E-20	1.85E+00	3.65E-20	3.20E-20	1.62E+00
ode23tb	1.79E-14	8.42E-15	1.98E+00	1.79E-14	1.84E-14	1.78E+00
ode23s	1.79E-14	1.99E-14	1.72E+01	1.79E-14	3.04E-14	1.46E+01
ode23t	2.84E-13	3.37E-13	1.16E+00	2.84E-13	3.96E-13	1.04E+00
ode15s	1.38E-12	1.66E-12	1.19E+00	1.38E-12	1.88E-12	1.10E+00

Table 1 MSE results of the integrators for the Free Precession of a Symmetrical top problem with a step of 0.001s.

Solver	Quaternions			Euler angles		
	$\boldsymbol{r^e}$	$\boldsymbol{v^e}$	**time**	$\boldsymbol{r^e}$	$\boldsymbol{v^e}$	**time**
ode45	3.3220E+00	2.7170E-02	3.14E+01	3.3220E+00	2.7170E-02	2.69E+01
ode23tb	3.3221E+00	2.7171E-02	1.12E+01	3.3221E+00	2.7171E-02	1.02E+01
ode23	3.3221E+00	2.7171E-02	1.48E+01	3.3221E+00	2.7171E-02	1.32E+01
ode113	3.3221E+00	2.7171E-02	1.04E+01	3.3221E+00	2.7171E-02	9.42E+00
ode23s	3.3221E+00	2.7171E-02	9.87E+01	3.3221E+00	2.7171E-02	9.82E+01
ode23t	3.3221E+00	2.7171E-02	6.68E+00	3.3221E+00	2.7171E-02	6.16E+00
ode15s	3.5045E+00	2.7171E-02	1.28E+01	3.5045E+00	2.7171E-02	1.16E+01

Table 2 MSE results of the integrators for the Free Fall problem with a step of 0.001s.

Solver	Quaternions			Euler angles		
	$\boldsymbol{r^e}$	$\boldsymbol{v^e}$	**time**	$\boldsymbol{r^e}$	$\boldsymbol{v^e}$	**time**
ode45	4.4403E-04	1.8136E-04	4.27E+00	4.4403E-04	1.8136E-04	3.80E+00
ode23s	4.4416E-04	1.8137E-04	1.39E+01	4.4416E-04	1.8137E-04	1.20E+01
ode23	4.4416E-04	1.8137E-04	2.13E+00	4.4416E-04	1.8137E-04	1.87E+00
ode23tb	4.4416E-04	1.8137E-04	1.58E+00	4.4416E-04	1.8137E-04	1.44E+00
ode23t	4.4424E-04	1.8137E-04	9.41E-01	4.4424E-04	1.8137E-04	8.57E-01
ode15s	4.4436E-04	1.8137E-04	9.69E-01	4.4436E-04	1.8137E-04	8.86E-01
ode113	6.5154E-04	1.3603E-04	1.49E+00	6.5154E-04	1.3603E-04	1.34E+00

Table 3 MSE results of the integrators for the Parabolic Motion problem with a step of 0.001s.

In all the experiments is clearly seen that the ODE45 solver was the most accurate of all, but also the slowest. In comparison with the ODE23 solver, ODE45 is twice as slow and with the exception of the Free Precession of a Symmetrical Top problem, where the accuracy increases by 10 orders of magnitude, ODE45 does not gain a lot of accuracy over ODE23. Therefore, to combine accuracy and speed in the default configuration of the RbSimulation class, ODE23 was chosen as default integrator.

Two more details that can be noticed in the tables are the differences in accuracy and speed between the quaternion and Euler angles atttitude simulation methods. These details are only visible in vectors that are in the IRF such as ω^e, r^e, and v^e, given that it is the quaternions or the Euler angles what transform the vectors from the BRF to the IRF. Vectors in the BRF such as ω^b do not undergo any transformation, hence, they are the same in both attitude simulation methods. In general, the quaternions are more accurate than the Euler angle implementation, although the opposite happens for the speed. Therefore, users must find an appropriate speed-accuracy trade-off when choosing their solver and attitude simulation method.

Chapter 3: Animation Module

3.1 Overview

The need for a visualisation tool is justified by the fact that understanding how physical phenomena occur is not always simple by looking at an analytical description, i.e. equations. Sometimes it is easier to understand the nature of a physical phenomenon by accompanying the mathematical model with images, diagrams, graphs and/or animations. As Giaquinto [42] states, visual thinking is widespread in mathematics. Moreover, according to Mowat [43], images can be a powerful tool for both teaching and learning. After all, one of the goals of this project is to work as an education tool. For those reasons, and because static images like graphs are not enough to fully display the motion of rigid bodies, the animation module was developed.

As seen in Chapter 1:, simulation tools usually include an animation component fitted to its system's needs in order to display their results. These animation tools were compared in order to find a base for this system's animation module and save time and effort in programming and testing. The animation classes of the Aerospace Toolbox™ were also included in the comparison and, being capable of loading geometries from different sources and having an object oriented approach, it was concluded that they were the best option for this project. Consequently, there are three main pillars upon which the animation module sits upon, all of them internal tools of MATLAB®. These pillars are, in order of importance:

1. Graphic tools
2. Aerospace Toolbox™
3. The Timeseries class

The functions and classes that belong to those three topics are briefly described in this document, detailed information about them can be found MW's online documentation or in ML's offline documentation.

3.1.1 Graphic tools

The graphic tools include classes that provide an easy way to visualise data, for example, figures, which are windows where graphic elements like plots and 3D objects can be displayed.

ML's graphic tools are also capable of doing hardware rendering using OpenGL®. Animations may require many of the system's resources to transform objects, that is, applying translations and rotations. Running animations with software may therefore result in slow refresh rates or in poor quality results. Rendering with hardware allows programmes to have a better performance when animating because the transformations are done by the system's graphics hardware, allowing the CPU to use its resources in other calculations. To render with hardware, transformation objects are created with the *hgtransform* function and take 4x4 transformation matrices which may rotate, translate and/or scale objects. When a transformation matrix is assigned, it is sent to the graphics hardware to be applied to the rendered objects.

3.1.2 MATLAB®'s Aerospace Toolbox™

The second pillar upon which the animation module sits is ML's Aerospace Toolbox™, which, among other features, includes classes that enable users to create animations from flight simulation data. The classes from the toolbox on which the animation module is based are:

- Aero.Body
- Aero.Animation

Aero.Body is a class that manipulates geometries. It is able to load, render and transform geometries. The main reason to use *Aero.Body* in this implementation is because it is able to load geometries from sources with different formats, for example, variables in the workspace, MAT files, AC3D™ files or custom geometries. This feature provides the animation module with a wide range of compatible formats while saving time in development. Another feature that saves development time is that, internally, *Aero.Body* uses a transformation object to render its loaded geometry with hardware, hence the only thing a user needs to do to apply a transformation is to pass translation and rotation vectors to the class's *move* method or to call the class's *update* method to create a time based transformation. Time based transformations are applied by passing a time value to the *update* method, which extracts from a selected data source, the translation and rotation vectors that correspond to the given time. A source for 6-DoF motion is a numeric array with seven columns, which are one column for time, three for position and three for rotation using Euler angles. The time based transformation capabilities of *Aero.Body* make animations possible. Although to display their contained geometry in an animation window, an instance of *Aero.Animation* must be created. The *Aero.Animation* class is an interface that groups instances of *Aero.Body*, displays them in a window and calls their *update* method so that they render their transformed geometries as the animation runs. It also controls the pace and direction of the animation. With the exception of the compatibility with quaternions, *Aero.Animation* meets all the requirements of the module. However, it cannot be directly implemented because the output of the SMOD is not compatible with the inputs of *Aero.Animation*. In the case of *Aero.Body*, which is the one that handles the geometries and the transformations, the translation data must be in Geographic coordinates, i.e. latitude, longitude and altitude. Therefore, the Cartesian components of the simulator's output must be converted so that they reflect the expected results. Moreover, the *move* method in *Aero.Body* internally applies a -90° rotation around the X axis to its handled geometry, deviating it from the simulators coordinate system. So, in order to correctly use the rotations from the simulator's output, a 90° rotation in the opposite direction must be applied before applying the rotations in the output.

Figure 3-1 and Figure 3-2, show three orthogonal arrows rendered by *Aero.Body* in two different windows. In both cases, the camera is set to have its up vector in the positive Z direction, which means that the Z component should point upwards. In the first case, Figure 3-1, the arrows are rendered in a figure window where, as expected, the Z vector points upwards. The second instance is an *Aero.Animation* window where the Z component points to the right while the Y component points downwards, demonstrating that the arrows went through a rotation around the X axis.

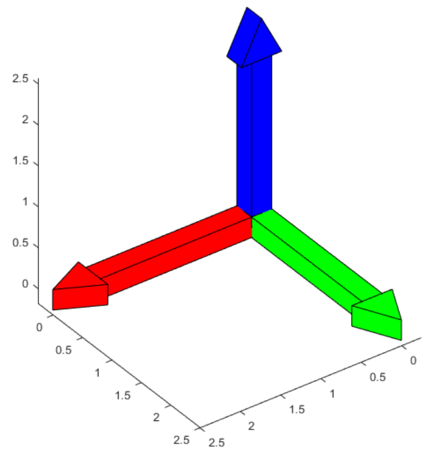


Figure 3-1 Aero.Body displayed in a figure window

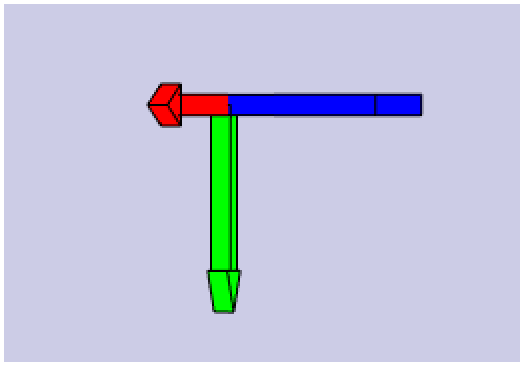


Figure 3-2 Aero.Body displayed in an Aero.Animation window

3.1.3 Timeseries Objects

Instances of the *Timeseries* class contain data vectors sampled over time. More importantly, the class provides methods to do time based manipulation of the data, i.e., interpolation, synchronization and filtering. Since the motion of rigid bodies is a set of vectors that change over time, the features of the *Timeseries* class are a convenient way to store and handle such data.

3.1.4 Module Design

The *RbAnimationBody* class was developed to wrap *Aero.Body* and use it in the desired coordinate system, which means applying translations in a Cartesian coordinate system and rotations in the desired direction.

A wrapper class takes advantage of the range of compatible formats and hardware rendering capabilities of *Aero.Body*, while customising the way the wrapped class interacts with other components of the system. An even better approach would be to derive a subclass from *Aero.Body*, but it is a sealed class, preventing any further derivation.

The *RbAnimation* class was developed to replace *Aero.Animation* due to five main reasons:

1. Since *RbAnimationBody* encapsulates the functionality of *Aero.Body* in an internal object, it is no longer compatible with *Aero.Animation*.
2. *Aero.Animation* is also a sealed class, so no subclasses can be derived from it.
3. To solve the compatibility issue, a reverse engineering process would be necessary in both the *Aero.Animation* and *Aero.Body* classes to see how they interact. However, the

code for the classes in the Aerospace Toolbox™ is obfuscated, which makes the reverse engineering process a longer and more complicated process.

4. MATLAB®'s graphic tools are easy to use and implement.
5. A custom made class will behave exactly as expected.

3.2 Software Implementation

3.2.1 Module Requirements

The animation module needed to meet the following requirements in order to provide a convenient visualisation platform:

Direct compatibility with the other modules. The inputs of the module must be designed in such a way that the outputs from the simulator and the mass distribution calculator can be directly animated, that is, without any conversion or adaptation.

Independence from the other modules. Simulations and mass calculations must not be mandatory prior to an animation.

Support for multiple bodies. The module must be capable of handling and animating multiple bodies in the scene.

Support for different rotation methods. Objects in the animation must have the possibility of rotating with quaternions and Euler angles.

Variable start and stop times. Users must be able to select at which point in the data's time the animation starts and stops, which means that the animation must not necessarily start at $t = 0$ and that animations can also be played backwards.

Real time animations. The animation's running time should match the time in the input data with an error of no more than 2%, i.e. if the input data goes from 0 to 10 seconds, the duration of the animation should be between 9.8 and 10.2 seconds.

Time scaling. Even though the established time unit for the entire system is one second, some experiments my happen in milliseconds, minutes, or some other time unit; therefore, the animation module must be able to scale the time in the input data to one second of animation.

Variable frame rate. The frame rate is an important factor in the quality of animations, so the animation module must provide an interface to control the frame rate of the output.

Exporting functionality. Users must be able to export the animations to videos to avoid having to run the code each time the animation needs to be seen, and also to allow the animations to be seen in systems without ML.

3.2.2 Module Workflow

RbAnimation is a container for all the resources related to the visualisation, it contains all the bodies that will be animated, start and final times for the animation, time scaling, etc. It is also what tells the bodies to update their location and orientation.

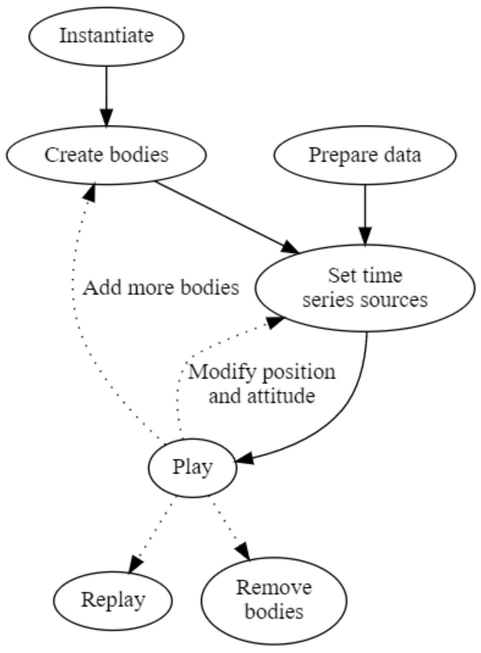


Figure 3-3 Workflow of the AMOD.

As Figure 3-3 shows, there are four mandatory steps to play animations:

1. Instantiating an object of the RbAnimation class.
2. Add some instances of the RbAnimationBody class.
3. Add time series data to the bodies.
4. Call the play function.

Bodies can be added or removed at any time, and the animation can be played backward when necessary.

3.2.3 RbAnimationBody Class

The purpose of *RbAnimationBody* is to manipulate the visualisation data of a rigid body, which means that for every body in the animation, there must be an instance of this class containing the body's geometry and transformation data. *RbAnimationBody* is a handle class to provide event based behaviour.

3.2.3.1 Properties

The following paragraphs explain the purpose of the class's properties. The descriptions start with the name of the property followed by the data type in square brackets. Example: property_name [type]. Unless otherwise noted, all the properties are of public access.

body [Aero.Body]. The encapsulated instance of the *Aero.Body* class. This property has private set access, so it cannot be reassigned. However, it has public get access to allow retrieving values from the *Aero.Body* instance.

position [1x3 double]. Location of the body in space in meters. The three columns correspond to the vector's *X*, *Y* and *Z* components.

rotation [1xN double]. Body's orientation in space. Depending on the situation, the orientation vector can represent a rotation from the BRF to the IRF or vice versa.

Vectors with Euler angles are 1x3 doubles where each column represents a rotation in radians around one axis. The default shape of the vector is $[\phi \quad \theta \quad \psi]$. Vectors with quaternions are 1x4 doubles where the components are $[w \quad i \quad j \quad k]$.

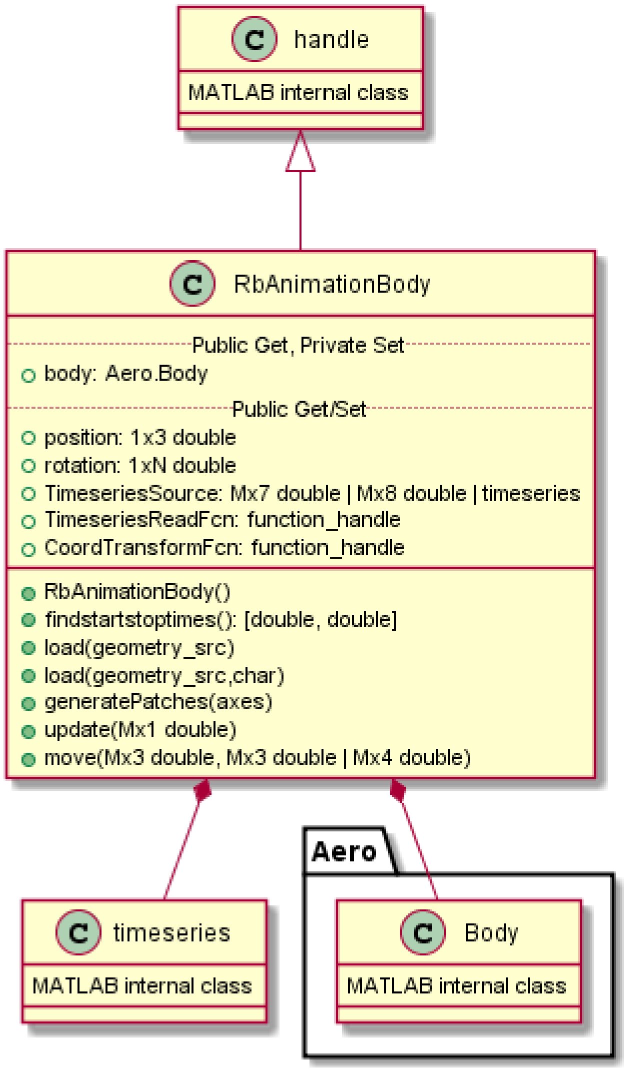


Figure 3-4 Class diagram of RbAnimationBody

TimeseriesSource [timeseries]. A *timeseries* object with the geometry's position and rotation data. The property can be set by assigning either a numeric matrix or a *timeseries* object. The numeric matrix must either be an Mx7 or an Mx8 double. The components of Mx7 matrices are interpreted as $[t \quad X \quad Y \quad Z \quad \phi \quad \theta \quad \psi]$ whereas components of Mx8 matrices are interpreted as $[t \quad X \quad Y \quad Z \quad w \quad i \quad j \quad k]$. In both cases, the first column is the time vector in seconds, the next three columns the position in meters and the remaining columns the rotation in either Euler angles in radians or unit quaternions.

Instances of the *timeseries* class used to set the property must have their data in Mx6 or Mx7 doubles. The components of Mx6 matrices are interpreted as $[X \quad Y \quad Z \quad \phi \quad \theta \quad \psi]$ whereas the components of Mx7 matrices are interpreted as $[X \quad Y \quad Z \quad w \quad i \quad j \quad k]$. In both cases, the first three columns are the position in meters and the remaining columns the rotation in either Euler angles in radians or unit quaternions. The time vector of the *timeseries* object must be in seconds.

Internally, numeric matrices are converted into *timeseries* objects, so in the end, it is always an instance of the *timeseries* class what handles the transformation data.

TimeseriesReadFcn [function handle]. A handler to the function that extracts the data out of the *TimeseriesSource*. By default, the function called is *rbInterp6DoFArrayWithTime*, which is based on the *interpTimeseries* function included in the Aerospace Toolbox™ and modified to support the extraction of quaternions. A detailed explanation of how *rbInterp6DoFArrayWithTime* works can be found in section 3.2.5.

Since the property is a function handle, users are able to point it to a custom extracting function, as long as it returns a 1x3 double with the position and a rotation vector in one of the two supported methods.

CoordTransformFcn [function handle]. A handler to a function that generates a 4x4 transformation matrix. By default, the function called is *rbBRF2IRF*, which generates transformation matrices meant to convert vectors form the BRF to the IRF. In cases where the conversion must be done in the opposite direction, the users can set the property to point to the *rbIRF2BRF* function, which is also included with the system. A detailed explanation of how *rbIRF2BRF* and *rbBRF2IRF* work can be found in section 3.2.5.

Since the property is a function handle, users are able to use customised transformation functions as long as they return a 4x4 transformation matrix.

3.2.3.2 Methods

The following paragraphs explain the purpose of the class's methods. The descriptions start with the method's name, followed by the argument list and the data type of their return value, if any. E.g. methodName(arg1, arg2): type. Unless otherwise noted, all the methods are of public access.

RbAnimationBody(). Constructor of the class. Initialises the encapsulated *Aero.Body* object.

findstartstoptimes() [double, double]. Finds the start and stop times in the *TimeseriesSource*. The first value of the returned vector is the start time and the second value the end time.

load(geometry_src, geometry_type). Wrapper that forwards its arguments to its homonymous in the encapsulated *Aero.Body*. By calling this method, the encapsulated Aero.Body loads the geometry in the *geometry_src* argument.

The second argument is optional and specifies the format of the loaded geometry. The possible values for the format are: ‘Auto’ for an automatic identification of the format, ‘Variable’ to load a *struct* from the workspace, ‘MatFile’ for stored MAT-files, ‘Ac3d’ for Ac3d files and ‘Custom’ for a custom format. When no second argument is passed, the format is considered automatic.

generatePatches(ax). Wrapper that passes its argument to its homonymous in the encapsulated *Aero.Body*. By calling this methods, the encapsulated *Aero.Body* generates a *patch* handle from the loaded geometry and displays it in the *axes* object passed in the argument.

update(t). Updates the body’s position and orientation versus time. The method calls the function pointed by *TimeseriesReadFcn* to extract the data at the time t and then pass it to the *move* method.

move(pos, rot). Translates and rotates the body. The method forwards its arguments to the function pointed by *CoordTransformFcn*, which is expected to return a 4x4 transformation matrix to be passed to the transformation object of the encapsulated *Aero.Body*.

throwError(id, msg, e_code, e_cause) [returns nothing]. Throws an instance of the *RbException* class.

The exception’s identifier is given by the first argument and it is appended to “AnimationBody:” to make simulation related errors easier to identify.

The error message is given by the fourth argument. The third and fourth arguments respectively add an error code and a cause to the exception. This method is private to the *RbAnimationBody* class.

3.2.4 *RbAnimation* Class

The purpose of the *RbAnimation* class is to group multiple instances of the *RbAnimationBody*, display them in a figure and call their *update* methods over time to create an animation. The properties and methods in *RbAnimation* are homonymous to some of the properties and methods in *Aero.Animation* to allow a transparent transition from one class to the other.

3.2.4.1 Properties

The following paragraphs explain the purpose of the class's properties. The descriptions start with the name of the property followed by the data type in square brackets. Example: property_name [type]. Unless otherwise noted, all the properties are of public access.

shown [boolean]. Flag set when the animation window is shown and cleared when the animation window is closed. The property is private to *RbAnimation*.

listener [handle]. Object that works as an event listener. When the animation window is closed, this object calls the *clearShownFlag* method. The property is private to *RbAnimation*.

Figure [figure]. Window where the animation will be. This property has private set access and cannot be reassigned. However, it has public get access to allow overlapping graphs.

CurrentAxes [axes]. Axes where the objects are rendered. This property has private set access and cannot be reassigned. However, it has public get access to allow overlapping graphs.

TStart [double]. Time in seconds at which the animation starts. Defaults to 0.

TFinal [double]. Time in seconds at which the animation stops. Defaults to 1.

FramesPerSecond [double]. Frame rate in FPS. Defaults to 20 FPS.

TimeScaling [double]. Determines how much data corresponds to one second of animation. For example: a value of 0.001 means that 0.001 seconds in a body's data will be animated in 1 second.

Bodies [cell]. Cell where the bodies being animated are stored.

Name [char]. Name of the animation object. The name appears in the title of the animation window.

FrameBuffer [struct array]. Buffer with the frames of the last animation. The buffer can be used in a later step to create movies and videos.

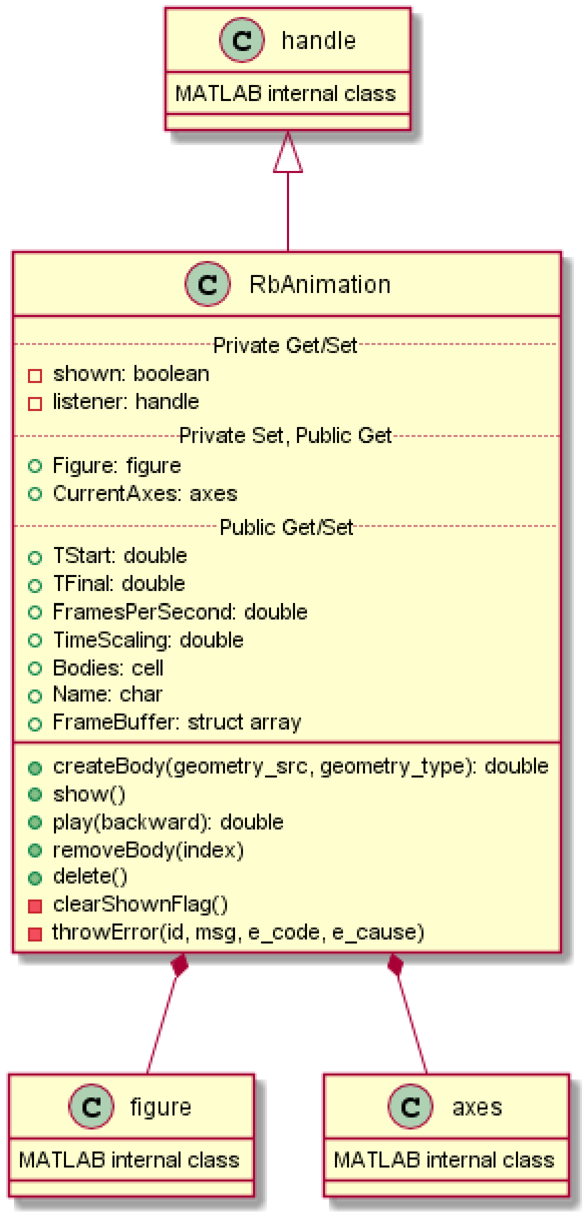


Figure 3-5 Class diagram of RbAnimation

3.2.4.2 *Methods*

The following paragraphs explain the purpose of the class's methods. The descriptions start with the method's name, followed by the argument list and the data type of their return value, if any. E.g. methodName(arg1, arg2): type. Unless otherwise noted, all the methods are of public access.

createBody(geometry_src, geometry_type) [double]. Creates a new instance of *RbAnimationBody* and adds it to the animation. The method forwards its arguments to the new body's *load* method. Then, the object's *CurrentAxes* property is passed as an argument to the new body's *generatePatches* method, so that the loaded geometry is displayed in the animation window. Finally, the new body is added to the body list and its index is the method's return value.

show(). Displays the animation window.

play(backward) [double]. Plays the animation. The method returns 0 when the animation runs successfully and 1 when it does not. The error causes are an empty body list or a wrong data type in at least one of the time related variables (*TStart*, *TFinal*, *TimeScaling* and *FramesPerSecond*).

The argument is a boolean that tells the class if the animation should be played backwards. The default value is false. Apart from the value in the argument, there are two additional ways in which the animation can be played backwards:

1. Set *TStart* greater than *TFinal*.
2. Play the animation forward and then use the *movie* function to play the saved buffer backwards.

Two backwards operation will result in a forward playback.

This method is what animates the objects, all the other properties and methods are interfaces that prepare the environment for this method to run. To create the animation, *play* goes through the following steps:

1. Opens the animation window if it not already open.
2. Compare the magnitudes of *TStart* and *TFinal* to identify if the animation should be played forward or backward. If forward, the time step between frames is positive, if backward, the time step is negative.
3. Once the sign of the time step is known, its magnitude is calculated by dividing *TimeScaling* by *FramesPerSecond*.

4. The total number of frames is equal to the nearest larger integer of the division $N_F = \left\| \frac{TFinal - TStart}{step} \right\|$.
5. The method then enters a loop where in each iteration, the current time step is passed to the *update* method of each body so that they extract their corresponding data and transform their geometries accordingly.
6. After the updates, the screen is refreshed with the *pause* function. A pause is needed because otherwise, the animation would be too fast to notice what happens. In an ideal situation, the duration of the pause would be equal to the inverse of *FramesPerSecond*, but since storing the frames in *FrameBuffer* takes time, a percentage of the inverse is used. By trial and error in the demonstration in section 3.3.1, the required percentage was determined to be 75%.

removeBody(index) [double]. Removes a body from the animation.

The argument is the index of the body that will be removed from the body list. If no argument is passed, the last body is removed. When the index is larger than the number of elements in the body list, the method throws an error.

throwError(id, msg, e_code, e_cause) [returns nothing]. Throws an instance of the *RbException* class.

The exception's identifier is given by the first argument and it is appended to "Animation:" to make simulation related errors easier to identify.

The error message is given by the fourth argument. The third and fourth arguments respectively add an error code and a cause to the exception. This method is private to the *RbAnimation* class.

3.2.5 Utility Functions for the Animation Module

rbInterp6DoFArrayWithTime(t, body): [1x3 double, 1xN double]. Extracts interpolated translation and rotation data from the *TimeseriesSource* property of an instance of *RbAnimationBody*.

The first argument is a double that indicates the time of the query data. The second argument must be an instance of *RbAnimationBody*.

The function uses ML's *interp1* function to do a 1-dimensional interpolation of columns of the data source. Interpolation is required because the *TimeseriesSource* property contains discrete samples that may not have any information at the indicated time.

The first output is a 1x3 double with the interpolated translation values in the shape $[X \quad Y \quad Z]$.

The second output is either a 1x3 or a 1x4 double with the interpolated rotation values. A 1x3 double contains Euler angles in the shape $[\phi \quad \theta \quad \psi]$ and a 1x4 double a unit quaternion in the shape $[w \quad i \quad j \quad k]$.

rbIRF2BRF(pos, rot): [4x4 double]. Creates a 4x4 matrix that transforms vectors from the IRF to the BRF.

The returned transformation matrix is a combination of a translation and a rotation using the first and second arguments respectively. The translation matrix is generated by the ML's *makehgtform* function from the first argument, which must be a 1x3 double in the shape $[X \quad Y \quad Z]$. The matrix has the following shape:

$$[P_e] = \begin{pmatrix} 1 & 0 & 0 & \mathrm{X} \\ 0 & 1 & 0 & \mathrm{Y} \\ 0 & 0 & 1 & \mathrm{Z} \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \text{Eq. 3.1}$$

The method for the rotation matrix generation depends on the shape of the second argument. A 1x3 double will be considered an array of Euler angles in the shape $[\phi \quad \theta \quad \psi]$ and the *angle2dcm* function will be used to generate a DCM as in Eq. 1.24, whereas a 1x4 double will be considered a quaternion and the *quat2dcm* function will be used as in Eq. 1.23. In either rotation case, the final result is a 3x3 matrix $[R'^{b}_{e}]$, meaning that it needs to be merged with a 4x4 in the following way:

$$[R_e^b] = \begin{pmatrix} R'_{11} & R'_{12} & R'_{13} & 0 \\ R'_{21} & R'_{22} & R'_{23} & 0 \\ R'_{31} & R'_{32} & R'_{33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \text{Eq. 3.2}$$

Matrices $[P_b]$ and $[R_e^b]$ are then combined in a single transformation matrix $[T_e^b]$:

$$[T_e^b] = [P_b][R_e^b] \qquad \text{Eq. 3.3}$$

rbBRF2IRF(pos, rot): [4x4 double]. Creates a 4x4 matrix that transforms vectors from the BRF to the IRF.

The arguments are interpreted as in *rbIRF2BRF*. Although in this case, the translation matrix is generated using the translation vector in the opposite direction, that is

$$[P_e] = \begin{pmatrix} 1 & 0 & 0 & -\mathrm{X} \\ 0 & 1 & 0 & -\mathrm{Y} \\ 0 & 0 & 1 & -\mathrm{Z} \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \text{Eq. 3.4}$$

The rotation matrix is also generated in different ways. An array of Euler angles would use Eq. 1.25 and quaternions will use the inverse of Eq. 1.23.

$$[R_e^b] = [R_b^e]^{-1} \qquad \text{Eq. 3.5}$$

Matrices $[P_e]$ and $[R_b^e]$ are then combined in a single transformation matrix $[T_b^e]$:

$$[T_b^e] = [P_e][R_b^e] \qquad \text{Eq. 3.6}$$

3.3 Demonstrations

3.3.1 Free Precession of a Symmetrical Top

The animation of the free precession runs in Script 3-1, which is based on demonstration made by Strauch [34]. The script displays a rotating ellipsoid, the space and body cones, the angular momentum vector. Additionally, the script uses the motion simulation module with quaternions as the transformation method. The importance of the solution time is close to null because the actual animation occurs after the simulation. Therefore, in this case, the accuracy of the quaternions over the Euler angles was the deciding factor.

3.3.1.1 Creating the Geometries

In Script 3-1 the geometries are loaded from MAT-files to reduce the length of this document. This section will nevertheless explain how the data in those files was generated. The animation environment includes 5 bodies: the rotating ellipsoid, the body cone, the space cone, an arrow representing the angular momentum vector in the IRF, and a system of three orthogonal arrows to represent the IRF.

While the arrows are already in a format that the *patch* function can use, the geometries for the ellipsoid and the cones have to be created as surfaces and then converted to patches with the *surf2patch* function. To create the rotating body, the *ellipsoid* function is called with the values for the position of the centroid along with the height and radius obtained in section 2.4.1. The values are $C = [0 \quad 0 \quad 0]$, $h = 0.5423m$ and $R = 4h$ respectively.

In section 2.4.1, the cones are automatically formed by plotting all the data points of the angular velocity. To create cones as meshed surfaces, their height and base radius must be computed and then passed to the *createCone* function. The height of each cone is equal to the projection of the angular velocity on the axis around which it rotates, whereas the base radius is equal to the shortest distance between the angular velocity vector and the axis of rotation. It is important to notice that the vector is a point in space and not an arrow or a line from the origin to the point. Hence the shortest distance is perpendicular to the axis of rotation. Something that simplifies the calculations is the fact that the cones are symmetrical entities, meaning that the height and radius can be calculated at any point in time with the same equations as long as the magnitude of the vector that forms the cone is known at that specific moment.

For the body cone, the height and radius are respectively the Z and X components of the angular velocity in the BRF at $t = 0$. No calculations are required because in the BRF, the Z component of the angular velocity is collinear with the axis of rotation, which means that the Z component is the projection on the axis of rotation. Next, at $t = 0$ the Y component of the angular velocity is equal to zero, meaning that at that point in time, the angular velocity vector is equal to the vector addition of the X and Z components, making the X component the shortest distance from the axis of rotation to the vector and thus the base radius of the cone.

To form the space cone, the angular velocity rotates around the angular momentum vector, which means that the first step is to compute the angle between them. To achieve this, the equation for the dot product is rearranged:

$$dot(\omega_e, H_e) = \|\omega_e\| * \|H_e\| * \cos(\alpha) \rightarrow \alpha = \cos^{-1}\frac{dot(\omega_e, H_e)}{\|\omega_e\| * \|H_e\|}$$

The angle allows the cone's height (projection) and base radius (distance) to be calculated as:

$$h = \|\omega_e\| \cos(\alpha) \qquad\qquad R = \|\omega_e\| \sin(\alpha)$$

Because the meshes generated by *createCone* are aligned to the Z axis, a rotation to align the cone's axis of rotation with the angular momentum vector is needed. The angle of rotation is the angle between the angular momentum vector and the Z axis and it can be calculated in a similar way as α:

$$\beta = \cos^{-1}\frac{dot(k, H_e)}{\|H_e\|}$$

The angular momentum vector is located in the XZ plane, hence β needs to be applied around the Y axis. For that, a corresponding DCM is generated and used to rotate all the vertices of the cone by β:

$$[R] = \begin{pmatrix} \cos\beta & 0 & -\sin\beta \\ 0 & 1 & 0 \\ \sin\beta & 0 & \cos\beta \end{pmatrix}$$

3.3.1.2 Animation

Script 3-1 uses the position and angular velocity results from the simulation in section 2.4.1 to create two data vectors, one for the rotating bodies and one for the static bodies. Figure 3-6 is a screenshot of the animation run with Script 3-1. The animation was successful in two ways:

1. The bodies moved as expected.
2. The error in the animation time was less than 1.35%.

The error is produced by the buffering in the *play* method. However, when the buffer is replayed with the *movie* function, the error is lower than 0.75%. The error was measured by averaging the running time of ten executions of each function.

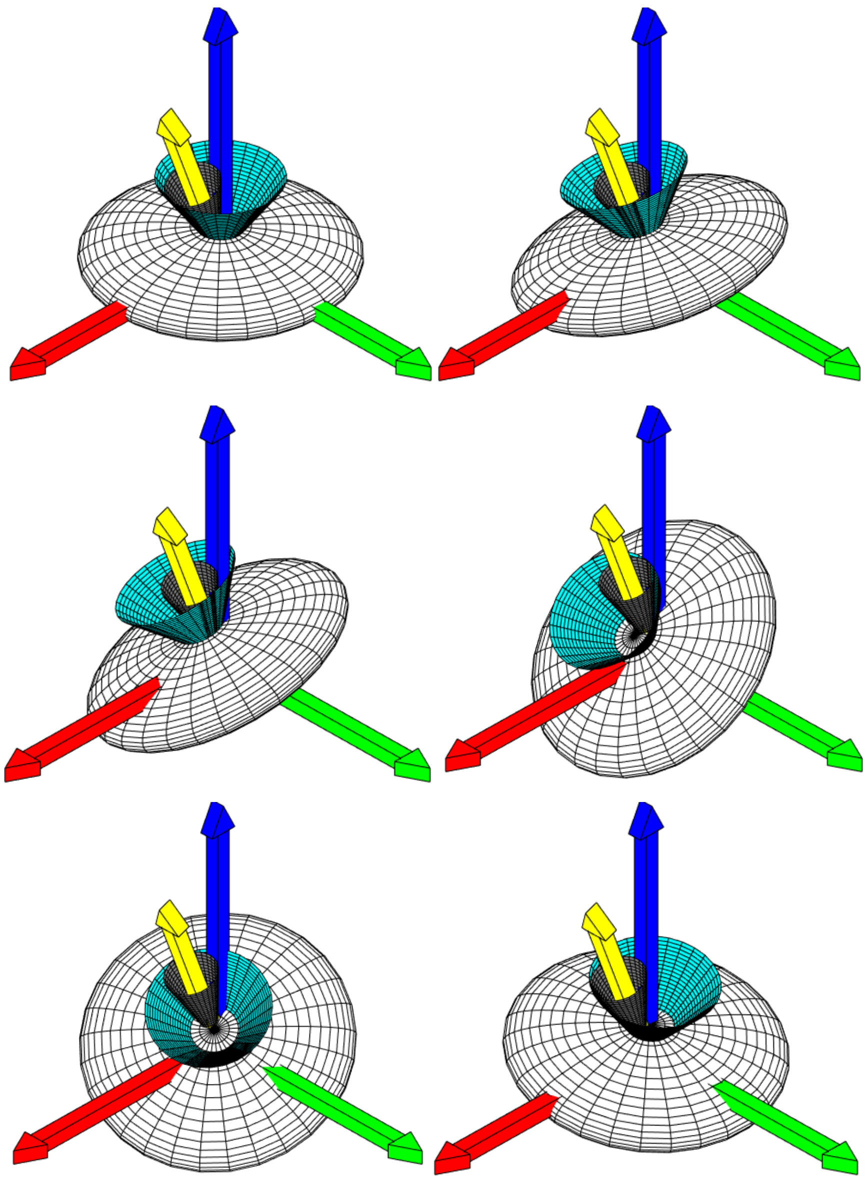


Figure 3-6 From left to right and top to bottom, the images are frames of the rotating ellipsoid at times $t = 0s$, $t = 0.5s$, $t = 1s$, $t = 1.5s$, $t = 2s$, $t = 2.5s$.

3.3.2 Free Rotation of an Asymmetrical Top

To demonstrate that the asymmetrical top never returns to its original position (see section 2.4.2), two sets of orthogonal arrows oriented in the aircraft's positive directions were added to the animation so that one is always attached to the satellite and the other one stays fixed at its original orientation.

The premises that confirm the correct operation of the animation module in this demonstration are:

1. One set of arrows must always have the same orientation as the satellite.
2. One of arrows must stay fixed at its original position.
3. With the exception of $t = 0$, the two sets of arrows must never have the same orientation.

3.3.2.1 Creating the Geometries

In Script 3-2, where the animation runs, the models for the satellite and the arrow sets are loaded from MAT files to keep the size of this document as short as possible. However, this section describes how the geometries in the files were created. The satellite model is created with the *createSatellite* function, which creates a 3U CubeSat in a 100:1 scale. The function was written to produce a scaled satellite because larger numbers are easier to handle in the code and changing the scale is done by simple scalar multiplications of the vertices. The arrow sets for the aircraft's orientation are a rotated version of the arrow set in the section 3.3.1, which are 4m long. For this reason, the satellite was scaled down to 10:1 so that in the animation it would not cover the arrow sets or be covered by them.

3.3.2.2 Animation

Script 3-2 uses the position and angular velocity results from the simulation in section 2.4.2 to generate one data vector for the satellite and its attached arrow set, and another one for the static arrow set. Both the simulation and the animation in this section have a one second timespan. Moreover, the time scale was set to 0.05 so that the animation time was extended to 20 seconds and the motion could be seen. A value of 0.05 means that the animation will last 1 second for every 0.05 seconds of simulation data. Figure 3-7 shows 6 images in a strip fashion to illustrate the tumbling motion of the satellite in the animation. The time between frames is 0.02 seconds. As expected, once the movement started, the arrow set attached to the satellite always maintained the same orientation as the satellite and it never returned to its original orientation, proving that the motion seen in the animation is correct. As in section 3.3.1, the error in time of the *play* method was measured and compared to the error of the *movie* function. Again, the functions were called ten times and their execution times were averaged, resulting in 4.2% of error for *play* and 0.26% of error for *movie*.

In conclusion, the *play* method can be used to demonstrate how bodies move in a 3D space, however, for applications that require and accurate animation time, replaying the buffer with the *movie* function is a better option.

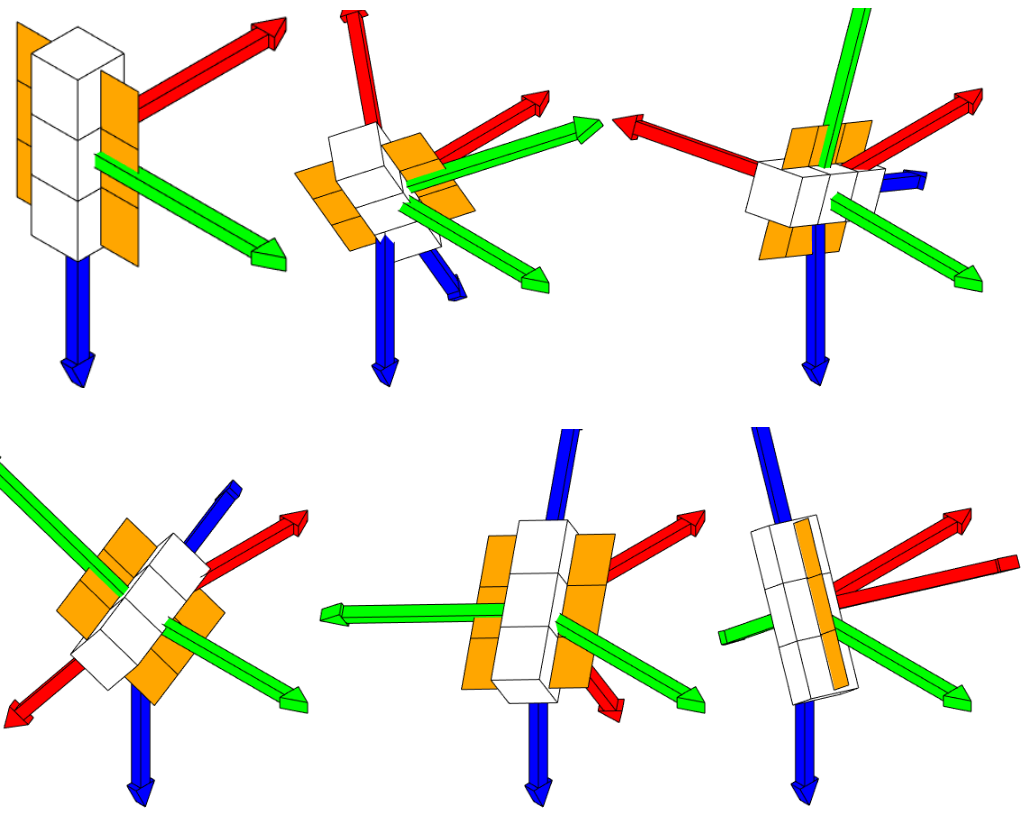


Figure 3-7 From left to right and top to bottom, the figures are frames of the tumbling satellite at times $t = 0s$, $t = 0.02s$, $t = 0.04s$, $t = 0.06s$, $t = 0.08s$, $t = 0.1s$

3.3.3 Linear Displacement

Script 3-3 creates an animation of three translating spheres. The first sphere translates back and forth along the X axis, the second sphere along the Y axis, and the third one orbits about the X axis while rotation about its own Z axis. Additionally, a set of three orthogonal arrows is added to the scene to demonstrate that the spheres travel along their corresponding axes. The translation of the first and second spheres is given by the following equation

$$r(t) = 3\sin\left(\frac{2\pi}{5}t\right) \qquad \text{Eq. 3.7}$$

Which means that the spheres will return to their original position in 5 seconds. In order for the third sphere to orbit about the X axis, the Y component of its position follows Eq. 3.7 while the Z component has the behaviour described by the following equation

$$r(t) = 3\cos\left(\frac{2\pi}{5}t\right)$$

The combination of sinusoidal and co-sinusoidal displacements along orthogonal axes causes a circular motion. Lastly, for the third sphere to rotate about its Z axis, the angle ψ must be a function of time of the form:

$$\psi(t) = \frac{\pi}{5}t$$

Which will cause the sphere to complete a cycle about its Z axis in 10 seconds. All the values used in this demonstration were chosen to simplify calculations and to produce an animation slow enough to be validated visually. This demonstration thus validates two premises: the correct translation functionality and the module's independence. As seen in Script 3-3, no simulation is run, all the values are manually generated and directly added to the scene. Figure 3-8 shows the last frame of the translating-spheres animation.

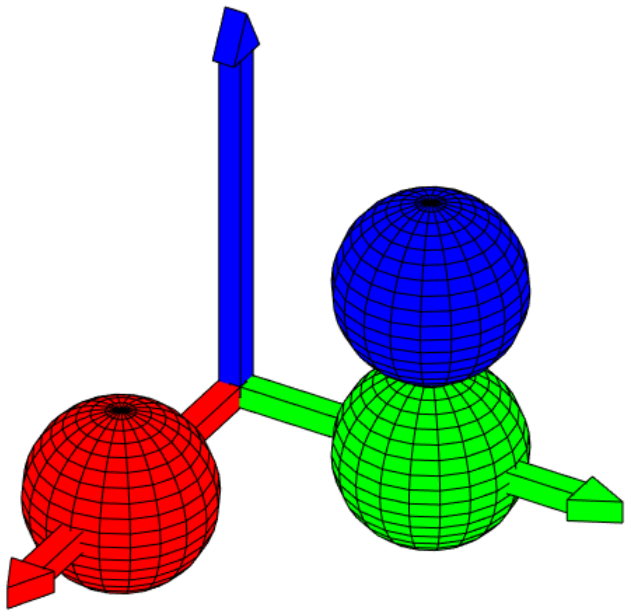


Figure 3-8 Translating spheres.

Chapter 4: Utilities

4.1 RbTimeVectorSum

4.1.1 Overview

Instances of the *RbTimeVectorSum* concatenate vectors of the shape Mx3 and computes a sum of all the values of the vectors at a specific point in time. The purpose of this class is to provide the simulator with difference sources of disturbances. For example, at an instance *t* the *RbTimeVectorSum* class could add the constant force of gravity plus a time changing thrust force and pass it to the simulator. Another use case is to store the output of the simulator to enable simulation in discrete steps, by that meaning that once each simulation step is finished, the values can be saved into an instance of *RbTimeVectorSum* and then repeat the process for a next step of simulation.

4.1.2 Design

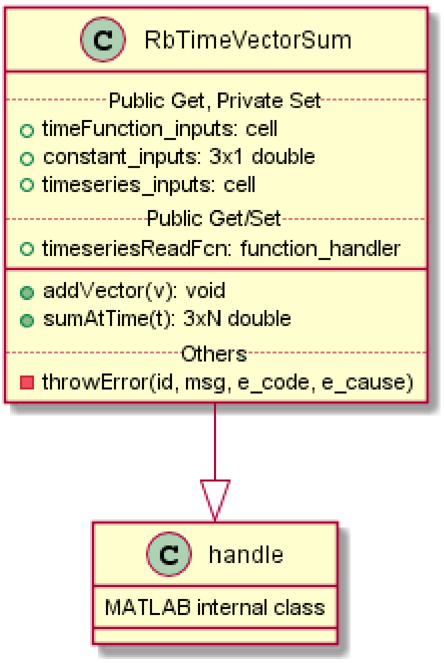


Figure 4-1 Class diagram of the RbTimeVectorSum class.

4.1.2.1 Properties

The following paragraphs explain the purpose of the class's properties. The descriptions start with the name of the property followed by the data type in square brackets. Example:

property_name [type]. Unless otherwise noted, all the properties are private to the *RbTimeVectorSum* class.

timeFunction_input [1xN cell]. Container for the time varying functions.

constant_input [3x1 double]. Container for the constants.

timeseries_input [1xN cell]. Container for instances of the class *timeseries*.

timeseriesReadFcn_input [function handle]. Points to the function that will extract the addition data at the chosen point in time. This property is of public access.

4.1.2.2 Methods

The following paragraphs explain the purpose of the class's methods. The descriptions start with the method's name, followed by the argument list and the data type of their return value, if any. E.g. methodName(arg1, arg2): type. Unless otherwise noted, all the methods are of public access.

addVector(v). Adds the vector in the argument to the corresponding container.

sumAtTime(double). Compute the sum of all the contained vectors at the time specified by the argument.

throwError(id, msg, e_code, e_cause) [returns nothing]. Throws an instance of the *RbException* class.

The exception's identifier is given by the first argument and it is appended to "Simulation:" to make simulation related errors easier to identify.

The error message is given by the fourth argument. The third and fourth arguments respectively add an error code and a cause to the exception. This method is private to the *RbTimeVectorSum* class.

4.1.3 Usage

Figure 4-2 shows the steps needed to use *RbTimeVectorSum* to concatenate vectors. The user must first instantiate an object of the class, then add the vectors of interest and then pass to the *sumAtTime* function the time at which the total value is needed.

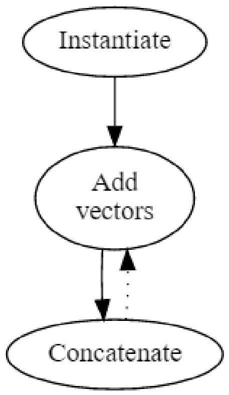


Figure 4-2 Flow of the RbTimeVectorSum class.

Conclusions

Three conclusions are made from the work developed for this project:

1. MATLAB®'s math-oriented scripting language, its included functions and its support for object oriented programming form a robust and flexible tool that will allow future modules to be added without much programming effort. Which means that the goal of making an easy-to-extend tool is met.
2. After going through the development of two modules from areas that could seem apart from each other (dynamics and animation), it is concluded on the academic side, that visualising a 3D animation of a moving rigid body is much more intuitive than looking at position or velocity plots, for which the goal of using the tool for educational purposes is met.
3. Finally, after looking at all the available open source tools, their flaws and features and how the community helps in the thrive of such tools, it is concluded that releasing the RBT to the public would draw people into further development and further extensions.

Future work

The first two modules leave space for many possible extensions. Starting with the MDC and its utilities to enable users to import geometries from 3rd party CAD software.

On the side of the SMOD, a generalised-coordinate simulation method should be added and compared with the current one to see what works better for multibody systems. Such an extension will enable users to use the simulation tool in robotics projects. Moreover, an attitude propagation module will enable users to plan future missions for satellites or find satellites that are already in space.

Finally, the SMOD can also be extended so that external entities can directly alter the state vector that goes into the simulation. A modification like that will enable controllers to take states inside a simulation step and apply control forces or torques.

On the AMOD side, future work can add dynamic control over the camera so that it tracks a moving object or to follow a user-defined behaviour.

References

[1] D. Goldsman, R. E. Nance and J. R. Wilson, “A Brief History of Simulation Revisited,” in *2010 Winter Simulation Conference*, 2010.

[2] P. Masarati, M. Morandini and P. Mantegazza, “An Efficient Formulation for General-Purpose Multibody/Multiphysics Analysis,” *Journal of Computational and Nonlinear Dynamics,* 2013.

[3] N. Docquier, A. Poncelet and P. Fisette, “ROBOTRAN: a powerful symbolic generator of multibody models,” *Mechanical Sciences,* pp. 199-219, 2013.

[4] E. Coumans and Y. Bai, “PyBullet, a Python module for physics simulation for games, robotics and machine learning,” 2016-2018. [Online]. Available: http://pybullet.org/.

[5] J. Shoer, “QuIRK: Multibody Dynamics for MATLAB 2009+,” 2010. [Online]. Available: http://spacecraftresearch.com/flux/quirk/index.html.

[6] R. Featherstone, “Spatial Vectors and Rigid-Body Dynamics Software,” February 2015. [Online]. Available: http://royfeatherstone.org/spatial/v2/index.html.

[7] R. Featherstone, Rigid Body Dynamics Algorithms, Springer US, 2008.

[8] S. A. Rawashdeh, “Passive Attitude Stabilization for Small Satellites. Master's Thesis,” Lexington, Kentucky, USA, 2010.

[9] D. Alazard, C. Cumer and K. Tantawi, “Linear dynamic modeling of spacecraft with various flexible appendages and on-board angular momentums,” in *7th International ESA Conference on Guidance, Navigation & Control Systems*, Tralee, Ireland, 2008.

[10] V. Carrara, “An Open Source Satellite Attitude and Orbit Simulator Toolbox for Matlab,” in *XVII International Symposium on Dynamic Problems of Mechanics*, Natal, Rio Grande Do Norte, Brazil, 2015.

[11] E. Stoneking, “42: A Comprehensive General-Purpose Simulation of Attitude and Trajectory Dynamics and Control of Multiple Spacecraft Composed of Multiple Rigid or Flexible Bodies,” NASA, 21 January 2018. [Online]. Available: https://software.nasa.gov/software/GSC-16720-1. [Accessed 6 February 2018].

[12] Princeton Satellite Systems, “Spacecraft Control Toolbox,” Princeton Satellite Systems, [Online]. Available: http://www.psatellite.com/products/sct/. [Accessed 6 February 2018].

[13] X. Hu and X. Li, “Research on modeling and simulation of satellite attitude control system,” in *11th IEEE International Conference on Control & Automation (ICCA)*, Taichung, Taiwan, 2014.

[14] L. Sarsfield, The cosmos on a shoestring : small spacecraft for space and earth science, Santa Monica, CA, USA: RAND, 1998.

[15] H. J. Koenigsmann and G. Gurevich, “AttSim, Attitude Simulation with Control Software in the Loop,” in *Proceedings of the AIAA/USU Conference on Small Satellites, Technical Session IIA: Management, Manufacturing, And Risk Mitigation*, Logan, UT, USA, 1991.

[16] J. M. Wing, “Computational Thinking: What and Why?,” *The Link. The magazine of Carnegie Mellon University's School of Computer Science.,* pp. 20-23, 2010.

[17] J. M. Wing, “Computational thinking and thinking about computing,” *Phil. Trans. R. Soc. A,* vol. 366, no. 1881, p. 3717–3725, 2008.

[18] J. Wing, “Computational Thinking,” *Commun. ACM,* pp. 33-35, 2006.

[19] K. Brennan and M. Resnick, “New frameworks for studying and assessing the development of computational thinking,” in *Proceedings of the 2012 annual meeting of the American Educational Research Association*, Vancouver, Canada, 2012.

[20] I. Lee, F. Martin, J. Denner, B. Coulter, W. Allan, J. Erickson, J. Malyn-Smith and L. Werner, “Computational thinking for youth in practice,” *ACM Inroads. Volume 2 Issue 1.,* pp. 32-37, March 2011.

[21] D. Moursund and D. Ricketts, “Computational Thinking,” 24 September 2016. [Online]. Available: http://iae-pedia.org/Computational_Thinking.

[22] H. M. Deitel and P. J. Deitel, C++ How to Program, Fifth Edition, Prentice Hall, 2005.

[23] C. Moler, “Technical Articles and Newsletters,” January 2006. [Online]. Available: https://de.mathworks.com/company/newsletters/articles/the-growth-of-matlab-and-the-mathworks-over-two-decades.html.

[24] S. Gace, “MATLAB simulation of fixed-mass rigid-body 6DOF,” 1 September 2016. [Online]. Available: https://de.mathworks.com/matlabcentral/fileexchange/3367-matlab-simulation-of-fixed-mass-rigid-body-6dof.

[25] S. Gace, “MATLAB Simulation of variable-mass rigid-body 6DOF,” 1 September 2016. [Online]. Available: https://la.mathworks.com/matlabcentral/fileexchange/3368-matlab-simulation-of-variable-mass-rigid-body-6dof.

[26] V. Ganapathi, “Simulation of Rigid Body Dynamics in Matlab,” 2005.

[27] B. D. Harper, Solving Dynamics Problems in MATLAB, John Wiley & Sons, Inc., 2007.

[28] D. Alazard, “Satellite Dynamics Toolbox,” 18 August 2014. [Online]. Available: https://personnel.isae-supaero.fr/daniel-alazard/matlab-packages/satellite-dynamics-toolbox.html?lang=fr.

[29] F. E. Udwadia and P. Phohomsiri, “Explicit equations of motion for constrained mechanical systems with singular mass matrices and applications to multi-body dynamics,” *Proceedings of the Royal Society A,* pp. 2097-2117, 2006.

[30] P. H. Zipfel, Modeling and Simulation of Aerospace Vehicle Dynamics, Reston, VA, USA: American Institute of Aeronautics and Astronautics, Inc., 2000.

[31] M. D. Ardema, Newton-Euler Dynamics, New York: Springer, 2005.

[32] R. Hibbeler, Dynamics, New Jersey: Pearson Prentice Hall, 2010.

[33] M. D. Shuster and W. F. Dillinger, “Spacecraft Attitude Determination and Control,” in *Fundamentals of Space Systems Second Edition*, New York, New York, Oxford University Press, Inc, 2005, pp. 236-288.

[34] F. W. Strauch, “Free Precession of a Rotating Rigid Body,” 17 October 2011. [Online]. Available: http://demonstrations.wolfram.com/FreePrecessionOfARotatingRigidBody/.

[35] S. H. Friedberg, A. J. Insel and L. E. Spence, Algebra Lineal. Primera Edicion, Publicaciones Cultural, S.A..

[36] E. W. Weisstein, “Moment of Inertia Ellipsoid,” 2007. [Online]. Available: http://scienceworld.wolfram.com/physics/MomentofInertiaEllipsoid.html.

[37] L. Landau and E. Lifshitz, Mechanics, Bristol: Pergamon Press Ltd., 1969.

[38] T. Davis, “Arrow3 Version 5,” 15 January 2013. [Online]. Available: https://de.mathworks.com/matlabcentral/fileexchange/14056-arrow3-version-5.

[39] J. B. Tatum, “Physics topics,” 16 November 2017. [Online]. Available: http://astrowww.phys.uvic.ca/~tatum/classmechs.html.

[40] The CubeSat Program, “CubeSat Design Specification Rev. 13,” 20 2 2014. [Online]. Available: http://www.cubesat.org.

[41] H. Nunome, A. Takeshi, I. Yasuo and S. Shinji, “Three-dimensional kinetic analysis of side-foot and instep soccer kicks,” *Medicine and Science in Sports and Exercise 34,* pp. 2028-36, 2006.

[42] M. Giaquinto, Visual Thinking in Mathematics, Oxford: Oxford University Press, 2007.

[43] E. Mowat, “Teaching and learning with images,” *VINE Journal of Information and Knowledge Management Systems,* pp. 5-13, 2002.

[44] H. Schaub, “Attitude Dynamics Fundamentals,” in *Encyclopedia of Aerospace Engineering, Volume 5 Dynamics and Control*, Barcelona, Spain, John Wiley & Sons, Ltd., 2010, pp. 3179-3186.

[45] A. W. Burks and A. R. Burks, “The ENIAC: First General-Purpose Electronic Computer,” *Annals of the History of Computing,* vol. 3, no. 4, pp. 310-399, 1981.

[46] A. Ross, “A Rudimentary History of Dynamics,” *Modeling, Identification and Control,* vol. 30, no. 4, p. 223–235, 2009.

Appendix A: Scripts

Chapter 2

Script 2-1

```
% Simulation of the free precession of a symmetrical top
simu = RbSimulation;
simu.iOmega_b = [1 0 1.4243];
simu.inertia = diag([1,1,1.8822]);
simu.tspan = [0 5];
simu.simulate(); % Quaternions
omega_bq = simu.omega_b;
omega_eq = quatrotate(quatinv(simu.quaternions),omega_bq);

figure;
subplot(3,2,1); plot(simu.time,omega_bq); title('Quaternions \omega_b [rad/s]');
grid on; xlim([0 5]); xlabel('Time [secs]'); ylabel('Amplitude');
subplot(3,2,2); plot(simu.time,omega_eq); title('Quaternions \omega_e [rad/s]');
grid on; xlim([0 5]); xlabel('Time [secs]'); ylabel('Amplitude');

simu.simulate(0); % Change to Euler angles
omega_be = simu.omega_b;
omega_ee = zeros(size(simu.time,1),3);
for i = 1:size(simu.time,1)
    omega_ee(i,:) = (angle2dcm(simu.euler(i,3),simu.euler(i,2),simu.euler(i,1))\omega_be(i,:)')';
end

subplot(3,2,3); plot(simu.time,simu.omega_b); title('Euler angles \omega_b [rad/s]');
grid on; xlim([0 5]); xlabel('Time [secs]'); ylabel('Amplitude');
subplot(3,2,4); plot(simu.time,omega_ee); title('Euler angles \omega_e [rad/s]');
grid on; xlim([0 5]); xlabel('Time [secs]'); ylabel('Amplitude');

% Cones
subplot(3,2,5); arrow3(zeros(size(omega_eq,1),3),omega_bq,'c');
hold on; arrow3(zeros(size(omega_eq,1),3),omega_eq,'r');
title('Quaternion''s space and body cones'); grid on;
subplot(3,2,6); arrow3(zeros(size(omega_ee,1),3),omega_be,'c');
hold on; arrow3(zeros(size(omega_ee,1),3),omega_ee,'r');
title('Euler angles space and body cones'); grid on;
```

Script 2-2

```
% Simulation of the rotation of an asymmetrical top
Ic = @(M,a,b) M*(a*a + b*b)/12; % Moment of inertia component

% Sat's moment of inertia
a_sat = 0.1; b_sat = 0.1; c_sat = 0.34; M_sat = 3;

Ix_sat = Ic(M_sat,b_sat,c_sat);
Iy_sat = Ic(M_sat,a_sat,c_sat);
Iz_sat = Ic(M_sat,a_sat,b_sat);
```

```
I_sat = diag([Ix_sat, Iy_sat, Iz_sat]);

% Panels' moment of inertia
a_pan = 0.0016; b_pan = 0.0825; c_pan = 0.329; M_pan = 0.132;

Ix_pan = Ic(M_pan,b_pan,c_pan);
Iy_pan = Ic(M_pan,a_pan,c_pan);
Iz_pan = Ic(M_pan,a_pan,b_pan);
I_pan = diag([Ix_pan, Iy_pan, Iz_pan]);

% Total moment of inertia
rx = 0; ry = 0.091; rz = 0;

Ig_pan = parallelTheorem(I_pan, M_pan, rx, ry, rz);
I = I_sat + 2*Ig_pan;

% Angular momentum and kinetic energy
H = 1; E = 20;

% Anguar momentum semi-axes
ah = H/I(1,1); bh = H/I(2,2); ch = H/I(3,3);

% Kinetic energy semi-axes
ae = sqrt(2*E/I(1,1)); be = sqrt(2*E/I(2,2)); ce = sqrt(2*E/I(3,3));

% Compute the angular velocity when for the current H, E and omg_x = 0
A = [1/bh^2, 1/ch^2; 1/be^2, 1/ce^2];
B = [1;1];
omg = sqrt(linsolve(A,B));

% Plot
fig1 = figure('Name','Ellipsoids');
ax = axes('Parent',fig1);
view(3); axis vis3d;
xlabel('X'); ylabel('Y'); zlabel('Z')

[Xh,Yh,Zh] = ellipsoid(0,0,0, ah, bh, ch);
[Xe,Ye,Ze] = ellipsoid(0,0,0, ae, be, ce);

fvc = surf2patch(Xh,Yh,Zh);
fvc.FaceColor = [1 0 0];
fvc.FaceAlpha = 0.5;
fvc.EdgeColor = 'none';
p = patch(ax,fvc);

fvc2 = surf2patch(Xe,Ye,Ze);
fvc2.FaceColor = [0.5 0.5 1];
fvc2.EdgeColor = 'none';
p2 = patch(ax,fvc2);

% Add grid and a light source for depth perception
camlight; grid on;

%% Simulation
simu = RbSimulation;
simu.iOmega_b = [0 omg(1) omg(2)];
simu.inertia = I;
simu.ode_options = odeset('InitialStep',0.01,'MaxStep',0.01);
simu.simulate();

legend('H','E_k'); hold on;
plot3(simu.omega_b(:,1),simu.omega_b(:,2),simu.omega_b(:,3),...
    'g','LineWidth',1.5,'DisplayName','\omega')
title('Ellipsoids with the overposed angular velocity');
```

Script 2-3

```
% Free fall test
simu = RbSimulation;
```

```
simu.iPosition_e = [4000 4000 4000];
simu.mass = 70;
simu.F_b = -9.81*70*[1;1;1];
simu.tspan = [0 28.55];
simu.simulate;

t = linspace(0,28.55,length(simu.time));
pos = 4000-(9.8/2)*t.^2;
v = -9.8*t;

subplot(1,2,1); plot(t, pos, 'r')
grid
title('Final position against time. $$ x_2=4000-\frac{9.8}{2}t^2 $$', 'Interpreter','latex')
ylabel('Final position [m]');
xlabel('time [s]'); xlim([0 28.6]);

hold on
plot(simu.time, simu.position(:,1),'o')

legend('Analytical','Simulation');

subplot(1,2,2); plot(t, v, 'r')
grid
title('Final velocity against time. $$ v = -9.8t $$', 'Interpreter','latex')
ylabel('Final velocity [m/s]');
xlabel('time [s]'); xlim([0 28.6]);

hold on
plot(simu.time, simu.velocity_e(:,1),'o')

legend('Analytical','Simulation');
```

Script 2-4

```
%% Parabolic motion simulation
simu = RbSimulation;
simu.iVelocity_b = [0 28*cos(deg2rad(45)) 28*sin(deg2rad(45))];
simu.F_b = [0;0;-9.81];
simu.tspan = [0 4.04];
simu.simulate;

l = length(simu.time);
t = linspace(0,4.04,l);
v_y = 28*cos(deg2rad(45))*ones(1,l);
v_z = 28*sin(deg2rad(45)) - 9.8*t;
pos_y = v_y.*t;
pos_z = 28*sin(deg2rad(45))*t - 9.8/2*(t.^2);

subplot(1,3,1); plot(t, pos_y, 'r')
grid
title('Final Y position. $$ y_2=28cos(45)t $$', 'Interpreter', 'latex')
ylabel('Final position [m]');
xlabel('time [s]'); xlim([0 4]);

hold on
plot(simu.time, simu.position(:,2),'o')

legend('Analytical','Simulation');

subplot(1,3,2); plot(t, pos_z, 'r')
grid
title('Final Z position. $$ z_2=28sin(45)t-\frac{9.8}{2}t^2 $$', 'Interpreter', 'latex')
ylabel('Final position [m]');
xlabel('time [s]'); xlim([0 4]);

hold on
plot(simu.time, simu.position(:,3),'o')

legend('Analytical','Simulation');
```

```
subplot(1,3,3); plot(t, v_z, 'r')
grid
title('Final Z velocity. $$ v_2=28sin(45)-{9.8}t $$', 'Interpreter', 'latex')
ylabel('Final velocity [m/s]');
xlabel('time [s]'); xlim([0 4]);

hold on
plot(simu.time, simu.velocity_e(:,3),'o')

legend('Analytical','Simulation');
```

Chapter 3

Script 3-1

```
%% Free Precession of a symmetrical Top.
%% Simulation
simu = RbSimulation;
simu.iOmega_b = [1 0 1.4243];
simu.inertia = diag([1,1,1.8822]);
simu.tspan = [0 5];
simu.simulate();

%% Animation
% Creates an instance of the RbAnimation class, adds the bodies, concatenates the results of the
% simulation into the required format and animates.
h = RbAnimation;
h.createBody('freePrecession_ellipsoid.mat');
h.createBody('freePrecession_body_cone.mat');
h.createBody('freePrecession_space_cone.mat');
h.createBody('ortho_arrows_irf.mat');
h.createBody('freePrecession_ang_mom.mat');

% The animation class requires a Mx8 matrix where the data is arranged as:
% [T X Y Z w i j k].
t_series1 = [simu.time, simu.position, simu.quaternions];

% The static bodies also need data to appear in the animation, so an empty
% dataset is set as their TimeseriesSource. In this case, the Euler angle
% shape is used an Mx6 double: [T X Y Z phi theta psi].
t_series2 = [simu.time, zeros(length(simu.time),6)];

% Animation as seen from the inertial reference frame. Comment out this
% section and uncomment the section below to see an animation of the motion
% as seen from the body reference frame.
h.Bodies{1}.TimeseriesSource = t_series1;
h.Bodies{1}.CoordTransformFcn = @rbBRF2IRF;
h.Bodies{2}.TimeseriesSource = t_series1;
h.Bodies{2}.CoordTransformFcn = @rbBRF2IRF;
h.Bodies{3}.TimeseriesSource = t_series2;
h.Bodies{4}.TimeseriesSource = t_series2;
h.Bodies{5}.TimeseriesSource = t_series2;

% Display the figure, set the end time, reorient the camera for a better
% view and play the animation.
h.Name = 'Free Precession Demonstration';
h.show();
h.TFinal = 5;
h.CurrentAxes.CameraTarget = [0 0 1];
h.play;
```

Script 3-2

```
% Animation of the free rotation of an asymmetrical top
%% Simulation
simu = RbSimulation;
simu.iOmega_b = [0 28.123 42.55];
simu.inertia = diag([0.0361, 0.0337, 0.00733]);
simu.ode_options = odeset('InitialStep',0.001,'MaxStep',0.001);
simu.simulate();

%% Animation
h = RbAnimation;
h.TimeScaling = 0.05; % 0.05 seconds in the simulation are 1 second of animation

h.createBody('cubesat3u.mat');
h.createBody('aircraft_brf.mat');
h.createBody('aircraft_brf.mat');

h.Bodies{1}.TimeseriesSource = [simu.time, simu.position, simu.quaternions];
h.Bodies{2}.TimeseriesSource = [simu.time, simu.position, simu. quaternions];
h.Bodies{3}.TimeseriesSource = [simu.time, zeros(length(simu.time),6)];

h.show();
h.CurrentAxes.CameraPosition = [30 30 30];
h.play;
```

Script 3-3

```
% Translation validation
len = 100;
t = 2*pi/5*linspace(0,10,len);
n = zeros(1, len); % empty vector to fill spaces

% 1 sphere along X, 1 sphere along Y, 1 sphere about X.
x = 3*sin(t);      y = x;       z = 3*cos(t);
psi = t/2; % Local rotation about Z for sphere 3

% Concatenate
data1 = [t; x;     zeros(5,len)]';
data2 = [t; n; y;  zeros(4,len)]';
data3 = [t; n; y; z; n ; n; psi]';
static = [t;       zeros(6,len)]';

% Geometries
h = RbAnimation;
[sx, sy, sz] = sphere(20);
sph = surf2patch(sx, sy, sz);

sph.cdata = [1 0 0];
h.createBody(sph, 'Variable');
h.Bodies{1}.TimeseriesSource = data1;

sph.cdata = [0 1 0];
h.createBody(sph, 'Variable');
h.Bodies{2}.TimeseriesSource = data2;

sph.cdata = [0 0 1];
h.createBody(sph, 'Variable');
h.Bodies{3}.TimeseriesSource = data3;

h.createBody('ortho_arrows_irf.mat');
h.Bodies{4}.TimeseriesSource = static;

% Animate
h.TFinal = 3;
h.play();
```

Appendix B: User’s Guide

Things to be familiar with to correctly use the system:

MATLAB®’s scripting language. All the information regarding MATLAB® functions, data types and coding syntax can be found in MathWorks™’s online documentation as well as in MATLAB®’s offline documentation.

Rigid body dynamics. Although it is not mandatory to have a deep understanding of how rigid bodies move, it is desirable to at least understand the basics in order to set up the simulation properly and reduce errors as much as possible.

The theory of rigid body dynamics of this project is mainly based on the books by [37] and [44]. Complementary information can be found in [31] and [32]. A really good source of simple explanations with images of some of the problems seen in this document and more can be found in the published classes by [39].

Adding the Toolbox to PATH

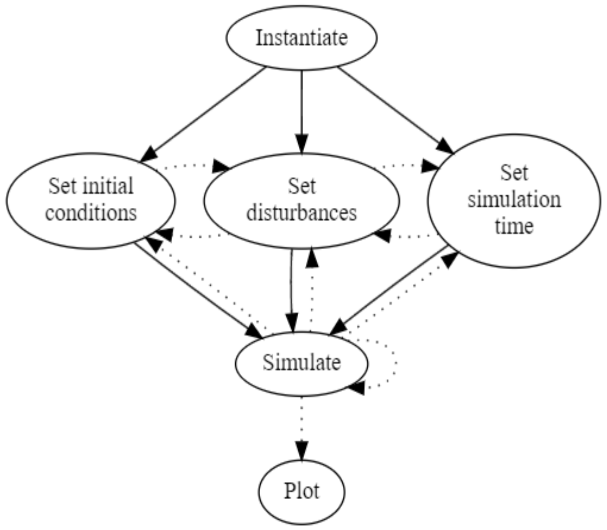
Before using the modules, the Toolbox’s directory needs to be in MATLAB®’s path so that the classes and functions can be found by MATLAB®. To achieve this, navigate within MATLAB® to the folder that contains the toolbox, right click on the toolbox’s folder, hover to “add to path” and finally click on “selected folder and subfolders”. Another option is to add the Toolbox’s directory to the path with from the command line:

```
addpath(genpath('C:\pathToToolbox\ToolboxDir\'))
```

Simulating

Express example:

```
simu = RbSimulation;
simu.iPosition_e = [0, 0, 4e3];
simu.mass = 70;
simu.F_b = [0; 0; -9.8*70];
simu.tspan = [0 40];
simu.simulate;
simu.plotResults('My First Simulation');
```



Walkthrough:

The dotted lines in the diagram represent an optional step, which means that there are many ways in which simulations can be run. Although most of the time there are only three steps involved:

1. Instantiate the class
2. Set the initial conditions, simulation time and/or disturbances
3. Simulate

These steps come after thinking about the scenario to simulate. That is, thinking about the mass properties of the body (mass and moment of inertia), initial position, orientation and velocities (linear and angular) and the time the simulation will last.

With that said, let's picture the problem from section 2.4.3: A 70kg skydiver in free fall. So first things first, the instantiation:

```
simu = RbSimulation;
```

From the problem's description, we know that the initial position is 4km above the ground, and considering the **Z** axis the one pointing to the sky:

```
simu.iPosition_e = [0, 0, 4e3];
```

The gravity is the only force acting on the skydiver, and since the simulator takes forces as inputs, the gravity acceleration must be multiplied by the skydiver's mass:

```
simu.mass = 70;
simu.F_b = [0; 0; -9.8*70];
```

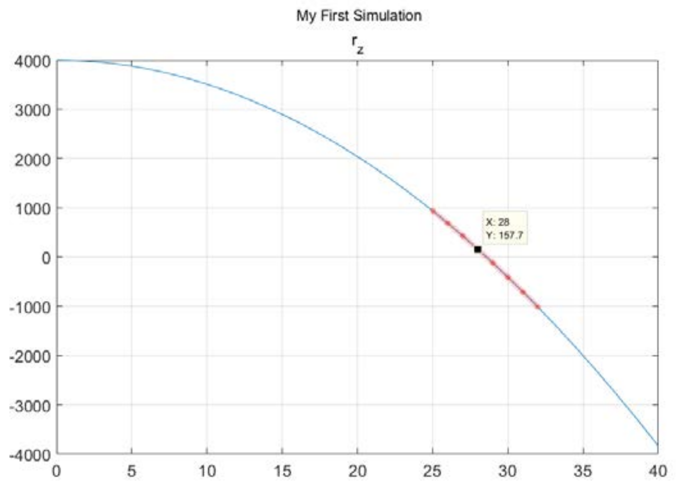
In section 2.4.3, the time in which the skydiver fall is calculated manually, but there’s always the graphical method when the analytical solution is not feasible. Hence we try with 40 seconds.

```
simu.tspan = [0 40];
```

And finally, run the simulation and plot the results. Plotting is optional.

```
simu.simulate;
simu.plotResults('My First Simulation');
```

From the graph we find that the skydiver takes approximately 28 seconds to reach the ground.



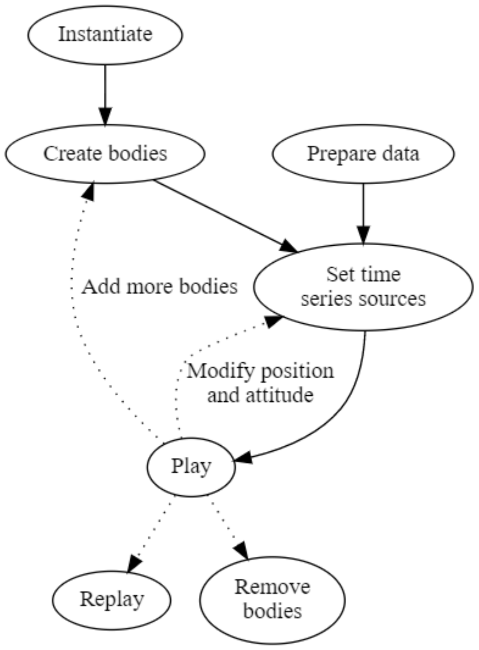
From this point onwards, the initial conditions, disturbances and simulation time can be changed without having to reinstantiate the object. Just change the values and call *simu.simulate()* and *simu.plotResults()* again.

Animating

Express example:

```
t = linspace(0, 10, 100);
x = 3*sin(2*pi/5*t);

ani = RbAnimation;
ani.createBody('cubesat3u.mat');
ani.Bodies{1}.TimeseriesSource = [t;x;zeros(5,100)]';
ani.TFinal = 5;
ani.play;
```



Walkthrough:

As seen in the diagram, three conditions must be met before playing the animation:

- An instance of RbAnimation
- At least one created body
- A time series source

So first step, the instantiation:

```
ani = RbAnimation;
```

To satisfy the second and third conditions, we load one of the included models and set its time series source with some time and position vectors.

When preparing input data, it is important to consider that the animation module takes Mx6 or Mx7 matrices (see section 3.2.3.1).To illustrate the solution to cases of motion with less than 6-DoF, this example will only have translation along the X axis, filling the rest of the columns zeros. For the position data, a sine function will cause a back and forth movement that will keep the body within the fame limits. TIP: Zeros in all motion columns means that the rigid body is static.

```
t = linspace(0, 10, 100)';
x = 3*sin(2*pi/5*t)';

ani.createBody('cubesat3u.mat');
ani.Bodies{1}.TimeseriesSource = [t;x;zeros(5,100)]';
```

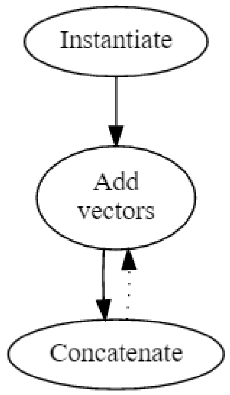
Once the conditions are satisfied, the animation can be played.

```
ani.TFinal = 5;
ani.play;
```

Adding Vectors versus Time

Although not mandatory for simulations, the *RbTimeVectorSum* class may be convenient to concatenate inputs and outputs.

On the input side, for example, the motion of a body may have disturbances of different types, e.g. constants and time varying forces. In that case, *RbTimeVectorSum* can contain both and provide the simulator, or any other object, with the total value of the disturbances at specific moments in time.



```
cat = RbTimeVectorSum;
gravity = [0, 0, -9.81*1.5];
thruster = @(t) [0; 0; 100/t];

cat.addVector(gravity);
cat.addVector(thruster);

v = cat.sumAtTime(60); % Get the total force at t=60.
```

On the output side, an experiment may require running simulations in multiple iterations. Every time an instance of *RbSimulation* runs the simulation, the previous results are overwritten, so in an iterative scenario, each new step would replace the previous one and in the end, only the results of the last step would remain. However, saving the outputs of each iteration as vectors in an instance of *RbTimeVectorSum* results in a time series of all the simulated data.

```
cat = RbTimeVectorSum;

simu = RbSimulation;
simu.ode_options = odeset('InitialStep', 0.01, 'MaxStep', 0.01);
simu.iVelocity_b = [100, 0, 0];

tfinal = 10;
t1 = 0;
for i = 0.2:0.2:tfinal     % Total of 20 seconds of simulation
   simu.tspan = [t1 i]; % simulate from the previous step to the current one
   simu.simulate;        % run the simulation

   if i < tfinal
      cat.addVector([simu.time(1:end-1,:), simu.position(1:end-1,:)]);
   else
      cat.addVector([simu.time, simu.position]);
   end

   if (simu.position(end,1) > 300)  % add a disturbance after 300 meters
      simu.F_b = [-50;0;0];
   end

   % initial conditions for the next iteration are the results of the current iteration
   simu.iPosition_e = simu.position(end, :);
   simu.iVelocity_b = simu.velocity_b(end, :);
   % initial time for next iteration
   t1 = i;
end
plot([0:0.01:tfinal],cat.sumAtTime([0:0.01:tfinal]))
```

Full System Example

In this example is based on the Free Precession demonstration from sections 2.4.1 and 3.3.1. The animation for this example will use Euler angles instead of quaternions and the motion will be seen from the body's perspective, that is, the surroundings will appear to be moving around the body.

Simulation.

As before, first instantiation and then setting the initial conditions.

```
simu = RbSimulation;
simu.iOmega_b = [1, 0, 1.4243];
simu.inertia = diag([1, 1, 1.8822]);
simu.tspan = [0 5];
```

To simulate with Euler angles, the simulation method needs a logical false value. The most common way to use represent a false logical value is a zero, but in MATLAB® you can also use *false*.

```
simu.simulate(0);  % <-- Use Euler angles!
```

Next, the resulting data needs to be packed so that bodies can use it in the animation.

```
motion_data = [simu.time, simu.position, simu.euler];
```

All the bodies need data to be in the animation, even if they're static bodies.

```
static_data = [simu.time, zeros(length(simu.time), 6];
```

That last line of code creates an array with as many rows as simulation's time steps and six columns full of zeros. The zeros will make sure that the bodies stay at the origin. Depending on the experiment, bodies can be fixed in other points.

Animation.

The geometries for this example are included with the system, so there is no need to do more math calculations to determine sizes and orientations. So directly to the instantiation and the loading of the bodies:

```
h = RbAnimation;
h.createBody('freePrecession_ellipsoid.mat');
```

```
h.createBody('freePrecession_body_cone.mat');
h.createBody('freePrecession_space_cone.mat');
h.createBody('ortho_arrows_irf.mat');
h.createBody('freePrecession_ang_mom.mat');
```

Additionally, the motion data is already prepared in the previous section, now we only need to determine which bodies we want to move. Since the goal is to observe the motion from the body's perspective, we know that the body should remain fixed, moreover, the body cone is not really a body itself, it is the concatenation of all the positions of the angular velocity vector as it rotates within the body's perspective, which means, that the cone also stays static in this example.

The are no more bodies in the BRF, so the remaining bodies must be the ones moving. The bodies are the orthogonal arrows that represent the IRF, the angular momentum vector and the space cone.

```
h.Bodies{1}.TimeseriesSource = static_data;
h.Bodies{2}.TimeseriesSource = static_data;

h.Bodies{3}.TimeseriesSource = motion_data;
h.Bodies{4}.TimeseriesSource = motion_data;
h.Bodies{5}.TimeseriesSource = motion_data;
```

By default, the animation module transforms vectors from the BRF to the IRF so that the body appears to be moving in space. Then, in order to see the space moving around the body, we need to apply an inverse transformation to the bodies that we want to move. The system includes *rbBRF2IRF* (default) and *rbIRF2BRF*, the second one is what we need. Should you need to develop your own transformation method, write a function and use the *CoordTransformFcn* property to point a function handle to it.

```
h.Bodies{3}.CoordTransformFcn = @rbIRF2BRF;
h.Bodies{4}.CoordTransformFcn = @rbIRF2BRF;
h.Bodies{5}.CoordTransformFcn = @rbIRF2BRF;
```

One last step before playing the animation is the time and camera adjustments. Since we simulated five seconds of motion, we'll also animate for five seconds. Then, to configure the camera we need to first open the animation window. More information about the camera tools can be found in https://de.mathworks.com/help/matlab/views.html

```
h.Name = 'Free Precession Demonstration'; % Window name
h.TFinal = 5;
h.show();
h.CurrentAxes.CameraTarget = [0 0 1];
```

And finally, play:

```
h.play;
```

Here is the script without interruptions:

```
% Simulation
simu = RbSimulation;
simu.iOmega_b = [1, 0, 1.4243];
simu.inertia = diag([1, 1, 1.8822]);
simu.tspan = [0 5];
simu.simulate(0);  % <-- Use Euler angles!

% Animation data preparation
motion_data = [simu.time, simu.position, simu.euler];
static_data = [simu.time, zeros(length(simu.time), 6)];

% Animation
h = RbAnimation;
h.createBody('freePrecession_ellipsoid.mat');
h.createBody('freePrecession_body_cone.mat');
h.createBody('freePrecession_space_cone.mat');
h.createBody('ortho_arrows_irf.mat');
h.createBody('freePrecession_ang_mom.mat');

h.Bodies{1}.TimeseriesSource = static_data;
h.Bodies{2}.TimeseriesSource = static_data;

h.Bodies{3}.TimeseriesSource = motion_data;
h.Bodies{4}.TimeseriesSource = motion_data;
h.Bodies{5}.TimeseriesSource = motion_data;

h.Bodies{3}.CoordTransformFcn = @rbIRF2BRF;
h.Bodies{4}.CoordTransformFcn = @rbIRF2BRF;
h.Bodies{5}.CoordTransformFcn = @rbIRF2BRF;

h.Name = 'Free Precession Demonstration'; % Window name
h.TFinal = 5;
h.show();
h.CurrentAxes.CameraTarget = [0 0 1];
h.play;
```

Modifying the System

There are some subjects you should be familiar with in order to modify the system:

Object Oriented Programming. Each module is a class or a set of classes, nothing too complicated, but still knowing what classes, encapsulation and inheritance are makes things much easier. A really good and short explanation about object oriented design can be found, as of the writing of this document, in the following video by Lucidchart: https://www.youtube.com/watch?v=UI6lqHOVHic

The video talks about design in the Unified Modelling Language (UML), but the concepts are the same in any object oriented language.

Object Oriented Programming in MATLAB®. Object oriented programming in MATLAB® has some differences from the programming languages like C++ or Java in the sense that attributes and methods have extended access capabilities, and that objects in MATLAB® are always a value class or a subclass of the handle classes. More information about OOP in MATLAB® can be found in: https://de.mathworks.com/help/matlab/object-oriented-programming.html.

The reason for which the modules are subclasses of the *handle* class and not value classes, is that value classes must be reassigned to themselves every time one of their methods modifies the class's properties. To illustrate this, the following code creates a value class called *ValueClass*:

```
classdef ValueClass {
        properties
                my_property = 0;
        end
        methods
                function obj = increase(obj)
                        obj.my_property = obj.my_property + 1;
                end
        end
end
```

The class is then instantiated in the console and tested with the following commands:

```
test = ValueClass;      % test.my_property is 0 at this point
test.increase();        % still 0
test = test.increase(); % now it is 1
```

Even though the *increase()* method was called twice, the resulting value of the property will be 1 because the first call to the method does not replace the existing object.

Having to reassign an object each time a method is called is something that users may forget and moreover, reassigning is something that does not really make sense with this particular class. Additionally, the modules were not left as value classes because *handle* objects provide event handling interfaces, which, in the future, can be used to develop a more robust physics simulation system with collision detection.

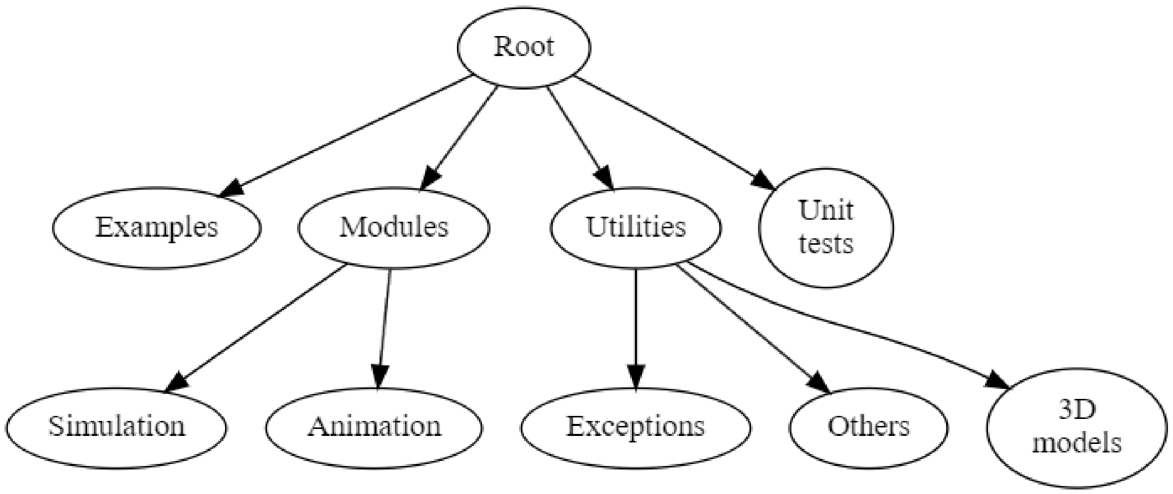
Error handling. In order to simplify the isolation and tracking of errors, it is important to be familiar with the exception classes of MATLAB®. Descriptions of the methods and classes can be found in https://de.mathworks.com/help/matlab/exception-handling.html

Unit testing. Simplifies the processes of checking whether the code works as expected or not, benchmarking the execution and debugging. Information about general unit testing can be found in http://xunitpatterns.com/index.html and information specific to MATLAB®'s Testing

Framework can be found in https://de.mathworks.com/help/matlab/matlab-unit-test-framework.html.

Naming convention. All the functions and classes developed for this project have a specific prefix to distinguish them from other toolboxes or native MATLAB® functions. Functions have the prefix *"rb"*, while classes have *"Rb"* as prefix.

The System's Project Tree. The project's root directory is divided in four branches for the modules, utilities, unit tests and examples.



Modules. Contains files for classes and functions the system cannot work without, and they are mostly what users interact with when running simulations or animations.

Utilities. Contains files for classes and functions that are not mandatory to use, but might extend the basic functionality.

Utilities/Exceptions. It is meant to contain classes related to error reporting. Having error reporting classes customised for the system makes the debugging process and the identification of problem sources faster.

Utilities/3D models. Contains geometries that are used often, so that reusing them saves lines of code and avoids possible errors.

Utilities/Others. This branch is not a directory. It represents the concept of whatever tools that may extend the system in a generic way. Like the *RbTimeVectorAddition* class, which may even be used outside of the system.

Unit tests. Contains classes that use MATLAB®'s Testing Framework to simplify tests on the code.

Examples. Contains scripts that demonstrates the functionality of modules or utilities. The code in this files should be well documented so that whoever reads it can easily understand what the script is doing.

With that said, whatever extension you develop, try to use the same structure so that the system remains consistent.