Mag.rer.nat. Bernhard Seifert, BSc

CubeSim - A Simulation Framework for small Satellites

Doctoral Thesis

to achieve the university degree of

Doktor der technischen Wissenschaften

submitted to

Graz University of Technology

Supervisor

Em.Univ.-Prof. Dipl.-Ing. Dr.techn. Otto Koudelka

Institute of Communication Networks and Satellite Communications

Prof. Dr.-Ing. Reinhold Bertrand

Institute of Flight Systems and Automatic Control, Technical University of Darmstadt

Graz, September 2022

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present doctoral thesis.

Date	Signature

Meinem Großvater

Acknowledgments

I would like to thank my first supervisor Em.Univ.-Prof. Dipl.-Ing. Dr.techn. Otto Koudelka for his consistent support and guidance during the running of this project. Many thanks also go to my second supervisor Prof. Dr.-Ing. Reinhold Bertrand for his feedback and his suggestions for improvement of this thesis.

Furthermore I would like to thank my family for all the support and patience during my research, calculations, programming and compilation of this thesis over the last years.

Abstract

In recent years small satellites have become increasingly popular. Universities and small companies got the chance to develop, test and qualify their own spacecraft. This change in paradigm from reliable, qualified and expensive electronic and mechanic components to commercially available parts not only strongly decreased development and testing costs, but also allowed to shorten the timeline of such a project. Where ESA (European Space Agency) or NASA (National Aeronautics and Space Administration) based space projects undergo several design, testing, manufacturing and qualification steps with accompanying strict reviews, CubeSat or small spacecraft projects experience more relaxed requirements.

Apart from these different approaches, large spacecraft and CubeSats share essential hardware components, such as the on-board computer, the electric power unit, the transceiver and the firmware. Though the software development and testing thereof shows similarities, hardware testing capabilities are fairly limited in low budget projects. Sophisticated custom-made test setups have to be developed to be able to test all subsystems, such as solar simulators, magnetic field generators, rotational tables, thermal vacuum facilities, GPS simulators, general ground station equipment or even radiation chambers. The verification and calibration of sensor inputs can be performed sequentially and there are hardly any dependencies, such as the testing of photo diodes can only be done in the absence of artificial magnetic fields or the transceiver can only be verified without irradiating the photo diodes.

In the Aerospace Engineering study course established at the University of Applied Sciences Wiener Neustadt, small satellites and their components are developed and tested with active student participation. Within the internationally reputable QB50 project, led by the von Karman Institute and funded by the European Commission, the University of Applied Sciences started with the first steps in this direction. The goal was to develop the 2U CubeSat PEGASUS which would use a designated scientific instrument to measure plasma properties, such as electron temperature and density, in addition to other subsystems. In order to build up expertise in the field of satellite development, the purchase of sub-systems was deliberately avoided as far as possible. All hardware was developed in-house in cooperation with the SpaceTeam of the Vienna University of Technology and the SpaceTech Group of the University of Vienna. The attitude control system including sensors and actuators was developed and tested by the author in the frame of the QB50 project.

In order to be able to test the attitude determination and control software, multiple sensor signals would have to be emulated in real time. At least a three axis rotational table, multiple solar simulators, three Helmholtz coil pairs and a GPS simulator would be required to verify the attitude determination algorithm. The interaction between the magnetorquers being driven by the attitude control algorithm and the present magnetic field is even more complicated to test. So-called *air tables* are commonly used to allow the spacecraft to move at reduced friction along two axes or to rotate around one axis. An extension to three axes and therefore a complete hardware-in-the-loop test setup for the verification of these algorithms is not possible.

In the scope of this work, another approach was taken. A purely software-based simulation framework was developed which uses the laws of mechanics to propagate celestial body and spacecraft movement. By

the use of Euler's rigid body equations, the rotation of the spacecraft is computed. This also includes the torque generated by the applied magnetic moment of the spacecraft and the magnetic field of the central celestial body. The framework is also able to deal with reaction wheels which can either be used to spin-stabilize the spacecraft or to perform attitude adjustments. In compliance with the Stefan-Boltzmann law the black body radiation of stars and the albedo reflection of other celestial bodies are simulated. Different models based on ray-tracing can be selected for that purpose.

The simulation of the aforementioned physical laws allows the computation of all spacecraft sensors at any time. These include photo diodes detecting irradiance from stars or from albedo reflection, gyroscopes measuring the spacecraft rotation in body frame, magnetometers measuring the ambient magnetic field, accelerometers detecting forces acting on the spacecraft and global navigation satellite systems such as GPS determining the relative position to Earth. In order to change the attitude of the spacecraft, reaction wheels and magnetorquers are being simulated. Orbital maneuvers such as altitude or plane changes can be accomplished by simulated thrusters.

The software framework was entirely programmed by the author in C++ which allows the direct execution of the on-board computer software, commonly developed in C or C++, on a personal computer. This enables debugging functionality or automated optimization of e.g. algorithm control parameters. The source code of CubeSim is published under the MIT software license at no charge and therefore allows the community to simulate the spacecraft in different mission scenarios including sensors, actuators and software without the necessity to spend a considerable part of the available development budget on expensive software tools or test hardware. The framework does not use any libraries provided by third parties.

Zusammenfassung

In den letzten Jahren haben Kleinsatelliten zunehmend an Popularität gewonnen. Universitäten und kleinere Firmen bekamen die Möglichkeit, ihre eigenen Satelliten zu entwickeln, zu testen und zu qualifizieren. Dieser Paradigmenwechsel von zuverlässigen, qualifizierten und teuren elektronischen und mechanischen Komponenten hin zu kommerziell erhältlichen Teilen, hat nicht nur die Entwicklungs- und Testkosten stark gesenkt, sondern auch die Zeitspanne eines solchen Projekts verkürzt. Wo von ESA (Europäischen Weltraumorganisation) oder NASA (Nationale Aeronautik- und Raumfahrtbehörde) geleitete Raumfahrtprojekte mehrere Design-, Test-, Fertigungs- und Qualifizierungsschritte mit begleitenden strengen Überprüfungen durchlaufen, erfahren Projekte zur Entwicklung eines CubeSats oder eines Kleinsatelliten einen geringeren Entwicklungs-, Dokumentations- und Testaufwand.

Abgesehen von diesen unterschiedlichen Herangehensweisen teilen sich große Satelliten und CubeSats wesentliche Hardwarekomponenten, wie den Bordcomputer (OBC), die elektrische Energieverarbeitungseinheit (EPS), den Transceiver und die Firmware. Obwohl die Software-Entwicklung und deren Tests Ähnlichkeiten aufweisen, sind die Möglichkeiten für Hardware-Tests bei Projekten mit geringen Budgets ziemlich begrenzt. Anspruchsvolle, maßgeschneiderte Testaufbauten müssen entwickelt werden, um alle Subsysteme testen zu können, wie z.B. Sonnensimulatoren, Magnetfeldgeneratoren, Drehtische, Thermalvakuumanlagen, GPS-Simulatoren, allgemeine Bodenstationsausrüstung oder sogar Strahlungskammern. Die Verifikation und Kalibrierung der Sensoren kann meist nur sequentiell erfolgen und es gibt kaum Möglichkeiten Abhängigkeiten zu überprüfen; so kann z.B. der Test von Fotodioden nicht bei gleichzeitiger Erzeugung von künstlichen Magnetfeldern erfolgen oder der Transceiver kann nur ohne Bestrahlung der Fotodioden verifiziert werden.

Im an der Fachhochschule Wiener Neustadt etablierten Studiengang Aerospace Engineering werden unter anderem Kleinsatelliten und deren Komponenten unter reger Studentenbeteiligung entwickelt und getestet. Im Rahmen des international renommierten QB50 Projekts, das vom von Karman Institut geleitet und von der Europäischen Kommission gefördert wurde, begann die Fachhochschule mit den ersten Schritten in diese Richtung. Ziel war es den 2U CubeSat PEGASUS zu entwickeln, welcher neben anderen Sub-Systemen auch ein bereitgestelltes wissenschaftliches Instrument zur Messung von Plasmaeigenschaften, wie Elektronentemperatur und -dichte, verwenden sollte. Um Expertise im Bereich Satellitenentwicklung aufzubauen, wurde bewusst auf den Zukauf von Komponenten soweit als möglich verzichtet. Sämtliche Hardware wurde in Zusammenarbeit dem SpaceTeam der Technischen Universität Wien und der Space-Tech Group der Universität Wien entwickelt. Das Lageregelungssystem inkl. Sensorik und Aktuatorik wurde im Rahmen von QB50 vom Autor entwickelt und getestet.

Um die Lagebestimmungs- und Steuerungssoftware testen zu können, müssten mehrere Sensorsignale in Echtzeit emuliert werden. Mindestens ein dreiachsiger Drehtisch, mehrere Sonnensimulatoren, drei Helmholtz-Spulenpaare und ein GPS-Simulator wären erforderlich, um den Algorithmus zur Lagebestimmung zu verifizieren. Die Wechselwirkung zwischen den vom Lageregelungsalgorithmus angesteuerten Magnetorquern und dem vorhandenen Magnetfeld ist noch schwieriger zu testen. Sogenannte Lufttische werden üblicherweise verwendet, um den Satelliten reibungsfrei entlang zweier Achsen zu bewegen oder

um eine Achse zu rotieren. Eine Erweiterung auf drei Achsen und damit ein vollständiger Hardware-in-the-Loop-Testaufbau zur Verifikation dieser Algorithmen ist nahezu unmöglich.

Im Rahmen dieser Arbeit wurde ein anderer Ansatz verfolgt. Es wurde ein rein softwarebasiertes Simulationsframework entwickelt, der die Gesetze der Mechanik nutzt, um die Bewegung von Himmelskörpern und Satelliten zu propagieren. Durch die Verwendung der Eulerschen Kreiselgleichungen starrer Körper wird die Rotation des Satelliten berechnet. Dabei wird auch das Drehmoment berücksichtigt, das durch das angelegte magnetische Moment des Satelliten und das Magnetfeld des zentralen Himmelskörpers (z.B. der Erde) erzeugt wird. Der Software-Framework ist auch in der Lage mit Reaktionsrädern (reaction wheels) umzugehen, die entweder zur Drehstabilisierung des Satelliten oder zur Durchführung von Lageanpassungen verwendet werden können. In Übereinstimmung mit dem Stefan-Boltzmann-Gesetz werden die Schwarzkörperstrahlung von Sternen und die Albedoreflexion anderer Himmelskörper simuliert. Dazu stehen verschiedene Modelle auf der Basis von Ray-Tracing zur Verfügung.

Die Simulation der oben genannten physikalischen Gesetze ermöglicht die Berechnung aller Sensoren des Satelliten zu jeder Zeit. Dazu gehören Fotodioden, die die Strahlung der Sterne oder des Albedos messen, Gyroskope, die die Rotation des Satelliten im körperstarren Bezugssystem messen, Magnetometer, die das umgebende Magnetfeld messen, Beschleunigungsmesser, die auf das Raumfahrzeug wirkende Kräfte erfassen und globale Navigationssatellitensysteme wie GPS, die die relative Position zur Erde bestimmen. Um die Lage des Satelliten zu verändern, werden Reaktionsräder und Magnetorquer simuliert. Ebenso können Bahnänderungen, wie Änderungen der Flughöhe oder der Orbitalebene, mittels Triebwerken simuliert werden.

Der Software-Framework CubeSim wurde vom Autor vollständig in C++ programmiert, was die direkte Ausführung der Bordcomputer-Software, die üblicherweise in C oder C++ entwickelt wird, auf einem Personal Computer ermöglicht. Dies erlaubt Debugging-Funktionen oder die automatische Optimierung von z.B. Algorithmus-Steuerungsparametern. Der Quellcode von CubeSim wird kostenlos unter der MIT Lizenz zur Verfügung gestellt und ermöglicht es der wissenschaftlichen Gemeinschaft, den Satelliten in verschiedenen Missionsszenarien einschließlich Sensoren, Aktuatoren und Software zu simulieren, ohne einen beträchtlichen Teil des vorhandenen Entwicklungsbudgets für teure Softwaretools oder Testhardware ausgeben zu müssen. Der Framework verwendet keine von Dritten bereitgestellten Bibliotheken.

Contents

List of Figures

List of Tables

Nomenclature

ADC	Analog-Digital Converter
ADCS	Attitude Determination and Control System
AMR	Anisotropic Magnetoresistance
AOCS	Attitude and Orbit Control System
API	Application Programming Interface
CAD	Computer Aided Design
CCD	Charge-Coupled Device
CPU	Central Processing Unit
DPL	Declared Parts List
DSP	Digital Signal Processor
DUT	Device Under Test
ECEF	Earth-centered, Earth-fixed
ECI	Earth-centered Inertial
EMI	Electromagnetic Interference
EPS	Electrical Power System
ESA	European Space Agency
FEEP	Field Emission Electric Propulsion
FEM	Finite Element Method
FR4	Flame Retardant 4
GLONASS	Globalnaja Nawigazionnaja Sputnikowaja Sistema
GMR	Giant Magnetoresistance
GNSS	Global Navigation Satellite Systems
GPIO	General Purpose Input Output
GPS	Global Positioning Systems
GTRF	Galileo Terrestrial Reference Frame
HAL	Hardware Abstraction Layer
HIL	Hardware-in-the-loop
HST	Hubble Space Telescope
HTML	Hyper Text Markup Language
I/O	Input/Output
I2C	Inter-Integrated Circuit
ICRF	International Celestial Reference System
IFM	Indium FEEP Multiemitter
IGRF	International Geomagnetic Reference Field
IIM	Interoperability and Integration Module
IPC	Inter-Process Communication
ISS	International Space Station
ITRF	International Terrestrial Reference Frame

JVM	Java Virtual Machine
LEO	Low Earth Orbit
MEMS	Micro-Electro-Mechanical Systems
NASA	National Aeronautics and Space Administration
NMEA	National Marine Electronics Association
NOAA	National Centers for Environmental Information
NORAD	North American Aerospace Defense Command
OBC	On-board Computer
OCS	Orbital Control System
OOP	Object-Oriented Programming
P-POD	Poly Picosatellite Orbital Deployer
PCB	Printed Circuit Board
PCB	Process Control Block
PHP	Hypertext Preprocessor
PID	Proportional Integral Derivative (Controller)
PIN	Positive Intrinsic Negative
POD	Plain Old Data
PPT	Pulsed Plasma Thruster
PPU	Power Processing Unit
PZ-90	Parametry Zemli 1990
RAII	Resource Acquisition Is Initialization
RAM	Random Access Memory
RGB	Red Green Blue
RMS	Root Mean Square
RPM	Rotations Per Minute
RTTI	Run-time Type Information
SI	Système international
SMD	Surface-Mounted Device
SPI	Standard Peripheral Interface
SPICE	Simulation Program with Integrated Circuit Emphasis
STK	Systems Tool Kit (formerly Satellite Tool Kit)
STL	C++ Standard Template Library
TCB	Thread Control Block
TCP/IP	Transmission Control Protocol/Internet Protocol
TLEs	Two Line Elements
TOMS	Total Ozone Mapping Spectrometer
TT&C	Telemetry, Tracking and Command
UART	Universal Asynchronous Receiver Transmitter
UI	User Interface
UV	Ultraviolet
WGS-84	World Geodetic System 1984
X3D	Extensible 3D
XML	Extensible Markup Language

Chapter 1

Introduction

This chapter shall give an overview of the research fields this thesis is dealing with, some background information and the motivation for the development of a novel and unique framework for the simulation of small satellites in different mission scenarios. The scientific problems and related questions are described and objectives are derived. These are not only proven theoretically within the frame of this thesis, but verification methods for the predefined modules and systems are presented.

1.1 Background and Motivation

The attentive reader might raise the question why there is the necessity to develop another simulation framework as there are simulation toolkits such as STK (Systems Tool Kit) from Analytical Graphics Inc. or in a broader context MATLAB with Simulink from The MathWorks [236].

A CubeSat or nano-satellite features a similar set of sensors, actuators and sub-systems like a larger spacecraft (such as a micro-satellite or beyond) despite its more stringent volume and mass constraints. These are at least the OBC (On-board Computer), the TT&C (Telemetry, Tracking and Command) and the EPS (Electrical Power System). A survey on commercially available sub-systems is provided in [209, p. 46ff].

If the satellite is required to maintain its attitude, an ADCS (Attitude Determination and Control System) is required consisting of a set of sensors to determine the attitude - such as sun sensors, Earth horizon sensors, star trackers or magnetic field sensors. In order to change or correct the attitude, actuators - such as magnetorquers, reaction wheels or lateral thrusters - are typically used. Magnetic field sensors and magnetorquers are only suitable for satellites in LEO (Low Earth Orbit) as the Earth magnetic field is required.

For changes in orbit, such as its inclination or altitude, an OCS (Orbital Control System) is used. It computes the required maneuvers and controls one or more thrusters to bring the satellite into the target orbit.

1.1.1 Experimental Test Stands

To change the orientation of a nano- or micro-satellite, due to the mostly negligibly low friction in space, only low torques are required. In LEO the Earth magnetic field is sufficiently strong that even small magnetorquers are sufficient (see section 9.6 for details). The challenge how to experimentally verify the movement or rotation of a satellite on ground is commonly met with different types of low friction tables

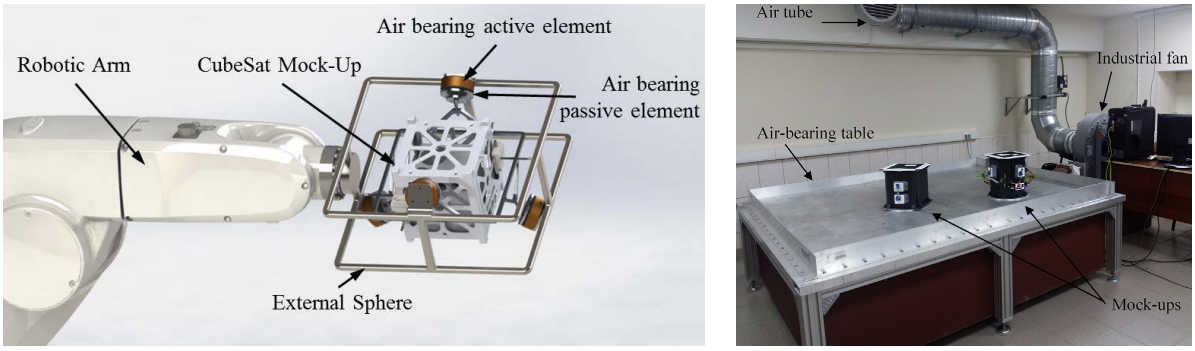

Fig. 1.1. Different low-friction experimental setups: air bearings attached to a robotic arm [80, p. 4] (left) and an air-bearing table [107, p. 266] (right).

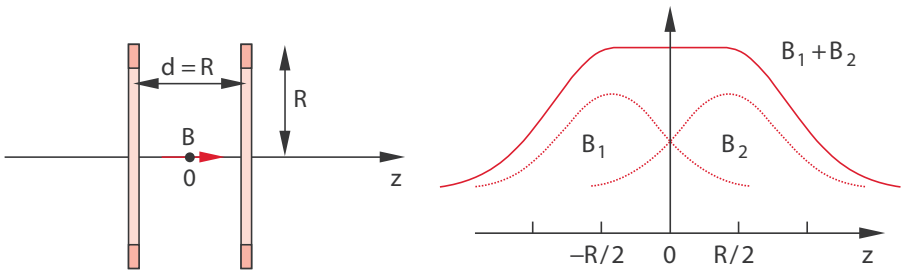

Fig. 1.2. Two annular coils with radius R at the distance d (left) and their magnetic field (right) [51, p. 92].

(see fig. 1.1). A trade-off analysis at the University of Colorado in cooperation with Lockheed Martin for the development of MockSat was done [12, p. 12ff]. For all setups the major disadvantages were the required level surface, the complexity of the setup and the limitation of movement on a plane or the rotation around one axis. Similar developments can be found in [225, p. 26ff] and [62, 18f].

In order to test magnetometers or to verify the functionality of magnetorquers, an Earth magnetic field simulator is required. This is commonly achieved by the use of three perpendicular pairs of Helmholtz coils. One pair comprises of two parallel annular coils with their distance d matching the radius R (see fig. 1.2). This configuration provides a controllable homogeneous magnetic field along the common axis at position z [51, p. 92]

$$B(z) = \frac{\mu_0 I}{(5/4)^{3/2} R} \left(1 - \frac{144\, z^4}{125\, R^4}\right) \tag{1}$$

with the vacuum permeability $\mu_0 = 4\pi \times 10^{-7}\,\mathrm{V\,s/A/m}$ and the current through the coils I. In the range $-R/3 \leqslant z \leqslant +R/3$, the inhomogeneity is only 1.2%. Such three-axis Helmholtz cages (as shown in fig. 1.3) are commonly used as described in [11], [46] and [29].

While for the verification of single- or multi-axis magnetometers, this setup is sufficient, the verification of one or more magnetorquers requires the satellite to freely move. This can be accomplished by the use of the aforementioned air tables [189, 238] or by suspending it on a string [196, p. 21]. In the latter case only one degree of freedom can be provided.

For the testing of solar panels and the calibration of sun sensors, so-called *solar* or *sun simulators* are required. These are special lamps that feature a similar spectrum and power density like the Sun. The spectral radiance of the Sun in $\mathrm{W/sr/m^3}$ is described by Planck's law [50, p. 81]

Fig. 1.3. Three-axis Helmholtz cage at the Western Michigan University [238, p. 18] (left) and at the MIT [189, p. 8] (right).

$$B_\lambda(\lambda, T) = \frac{2h\,c^2}{\lambda^5} \frac{1}{e^{h\,c/(\lambda\,k_\mathrm{B}T)} - 1} \tag{2}$$

with the Planck constant $h = 6.62607015 \times 10^{-34}\,\mathrm{J/Hz}$ [50, p. 80], the speed of light $c = 2.99792458 \times 10^9\,\mathrm{m/s}$ [51, p. 481], the Boltzmann constant $k_\mathrm{B} = 1.380649 \times 10^{-23}\,\mathrm{J/K}$ [52, p. 24], the black body temperature of the Sun $T = 5772\,\mathrm{K}$ [256, p. 9] and the wavelength λ.

With the mean radius of the Sun $R = 6.957 \times 10^8\,\mathrm{m}$ [256, p. 9], the mean distance from Sun to Earth $r = 1.49596 \times 10^{11}\,\mathrm{m}$ [155, p. 1], the spectral irradiance on earth results in

$$P_{\lambda,\,\mathrm{Earth}}(\lambda, T) = \frac{\pi R^2}{r^2} B_\lambda(\lambda, T) = 6.79444 \times 10^{-5}\, B_\lambda(\lambda, T). \tag{3}$$

and is shown in fig. 1.4. One recognizes the strong absorption of several wavelengths in the Earth atmosphere mainly caused by oxygen, water and ozone [119, p. 3]. The irradiance or power density on Earth is obtained by integrating over all wavelengths

$$P_\mathrm{Earth} = \int_0^\infty P_{\lambda,\,\mathrm{Earth}}(\lambda, T)\,\mathrm{d}\lambda = 1361.2\,\mathrm{W/m^2} \tag{4}$$

matching the solar irradiance as indicated in [155, p. 1].

The aforementioned solar simulators are shown in fig. 1.5 and described in [250, p. 8 and 10] and [146, p. 11ff]. At the University of Bologna, the solar simulator is combined with a Helmholtz cage allowing to test magnetometers, sun sensors and solar panels at the same time. The testing of magnetorquers is not feasible in that setup since the use of an air table or air bearings would make the test stand even more complex.

The Earth albedo describes the ratio between diffuse reflection of solar radiation out of the total solar radiation. The relative reflectivity of the Earth surface strongly depends on the terrain and can vary from 0.05 for a forest up to 0.9 for fresh snow [57, p. 2]. Moreover it varies over the visible light spectrum but the average values over the northern and southern hemisphere are similar [224, p. 18]. A more detailed analysis on the dependency of the reflectivity on the latitude is shown in section 4.3.

Though various modeling approaches were performed for the Earth albedo, as found in [26], [72, p. 13ff] and [117], with the goal to estimate or increase the pointing accuracy of sun sensors, test stands for experimental verification are not very common. Combining a solar simulator with an additional light source roughly matching the spectrum of the Earth albedo, is sufficient in most cases to assess the resulting pointing error.

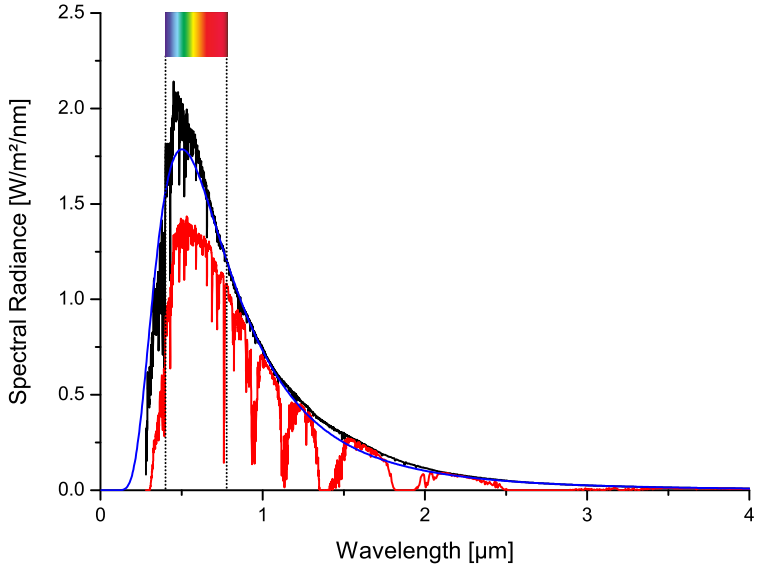

Fig. 1.4. Spectral radiance of the Sun (black), irradiance on Earth surface (red) and idealized black body radiation with $T = 5772\,\mathrm{K}$ (blue) [171].

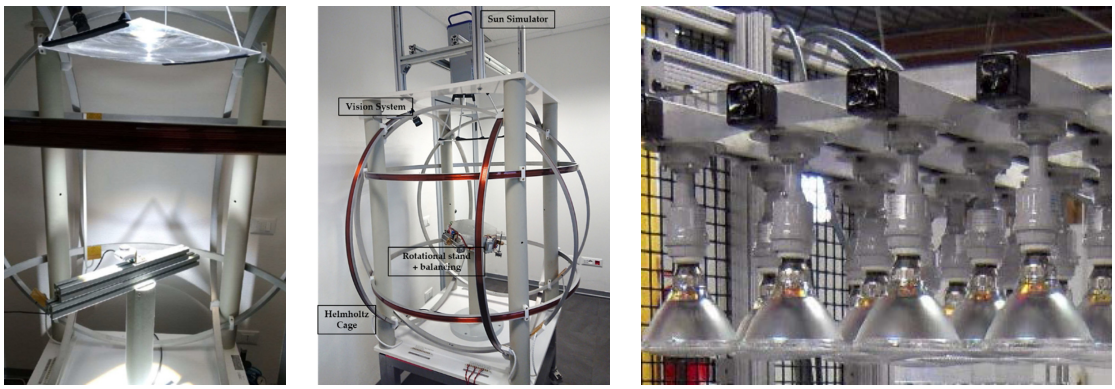

Fig. 1.5. Solar simulator at the Department of Industrial Engineering, University of Bologna [146, p. 11] (left), combined with a Helmholtz cage (center) and at the Utah State University [250, p. 10] (right).

In addition to sun sensors and magnetometers, so-called *horizon sensors* can be used to determine the orientation of the spacecraft. Such a horizon sensor is commonly comprised of a low-resolution CCD (Charge-Coupled Device) in the visible or infrared range as described in [254, p. 4f], [173] and [182]. A simulator to test or calibrate these sensors consists of a light source and an aperture. By changing the incident angle and relative position to the sensor and by variation of the light intensity, different scenarios can be simulated [250, p. 9].

To find the orbital position and velocity of the satellite, a direct determination can be performed by the use of a GNSS (Global Navigation Satellite Systems) tracker or alternatively, TLEs (Two Line Elements) from tracking services, such as NORAD (North American Aerospace Defense Command), can periodically be uplinked to allow the OBC of the satellite to propagate its position and velocity (see section 3.13 for details). For the first case not only the signal integrity and processing within the tracker needs to be verified, but also the further data processing in the AOCS (Attitude and Orbit Control System) algorithm. One option is the injection of simulated NMEA (National Marine Electronics Association) sentences via a serial communication interface as most GNSS trackers, such as the Novatel OEM615 [176], provide

a UART-compatible (Universal Asynchronous Receiver Transmitter) interface. The used NMEA 0183 protocol is described in [24] and [170]. In order to verify the receiver electronics and the data processing, so-called *GNSS simulators* can be used (see fig. 1.6). These support different constellations and allow the user to define custom trajectories.

Fig. 1.6. GNSS simulator RACELOGIC LabSat 3 [191] (left) and the Orolia GSG-5/6 simulator [179] (right) both supporting various constellations such as GPS, Galileo, GLONASS or BeiDou.

1.1.2 Hardware-in-the-loop Simulation

For testing and the verification of sensors and actuators, various test stands and experimental setups have been developed, each of which serving a specific purpose. Due to physical or budgetary restrictions, certain trade-offs in functionality and performance need to be made - a solar simulator might not be able to cover the entire satellite or the air table only allows low-friction movement in one direction. The complexity even increases when multiple test stands need to be combined, such as a solar simulator with a Helmholtz cage or a Helmholtz cage with an air-bearing table. It is obvious that only a part of the AOCS or ADCS algorithm can be tested experimentally and the code needs specifically be adapted for a certain test case. Not only does this prevent all code from being tested at once, it also introduces the risk of new error sources.

As extensive and complete testing are crucial steps regarding the verification and validation of sub-systems or the entire spacecraft, a different approach is commonly taken, the so-called HIL (Hardware-in-the-loop) simulation. Though there are different approaches what to abstract and how the interfaces shall be modeled, what this approach has in common, is a simulation computer that computes sensor signals, provides digital and analog signals and reacts on abstracted actuators commanded by the DUT (Device Under Test).

At the King Mongkut's University of Technology North Bangkok, the ADCS algorithm of a 1U CubeSat was tested by abstracting the magnetometers, magnetorquers and gyroscopes [234]. The sensors were replaced by micro-controllers connected to the DUT via an I2C (Inter-Integrated Circuit) digital communication interface and their values were computed by the simulation computer. The current through the three magnetorquers was measured and fed into the simulation computer allowing it to respond accordingly.

For the validation of the MOVE-II CubeSat, developed at the Technical University of Munich, a similar approach was taken [118]. The panels containing integrated magnetorquers and sun sensors were abstracted by embedded systems and even solar array simulators were implemented to allow virtual charging of batteries and to compute the power dissipation. All interface emulators were controlled by the simulation computer as illustrated in fig. 1.7.

At the Politecnico di Torino a similar approach was taken to validate the functional requirements of the e-st@r-I CubeSat [43]. The tasks of the simulation computer were similar but serial debug interfaces were used to connect the computer with the DUT. The commands meant for the actuators were also submitted to the simulation computer. This approach reduces the hardware development effort and provides a higher level of abstraction.

For the development of the simulation software, often MATLAB with Simulink is used as shown in [216, p. 16ff], [39, p. 4ff], [245, p. 54ff] and [78, p. 23ff]. The University of Puerto Rico set up a Simulink model for the EPS of the SPICA CubeSat as shown in fig. 1.8.

For mission planning, orbital propagation, thruster simulation and to assess atmospheric effects, the STK is commonly used [78, p. 37ff], [36, p. 12ff], [116]. While MATLAB provides an interface to a C++ library [243] and can directly access serial communication ports, such as RS-232, the module IIM (Interoperability and Integration Module) allows STK to interface with other programming languages, such as C or MATLAB, and TCP/IP (Transmission Control Protocol/Internet Protocol) sockets.

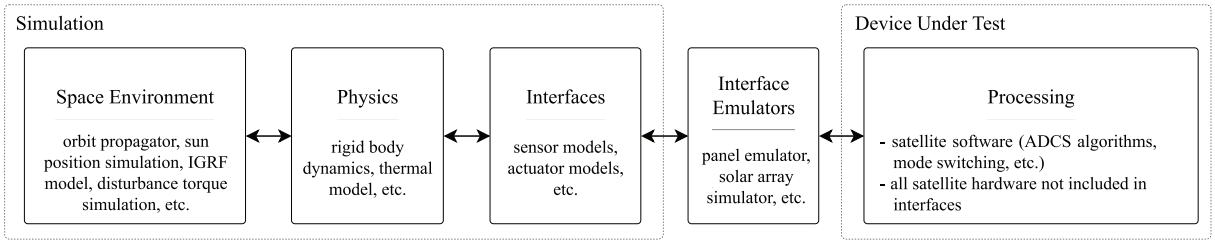

Fig. 1.7. Hardware-in-the-loop setup of MOVE-II CubeSat at the Technical University of Munich [118, p. 4].

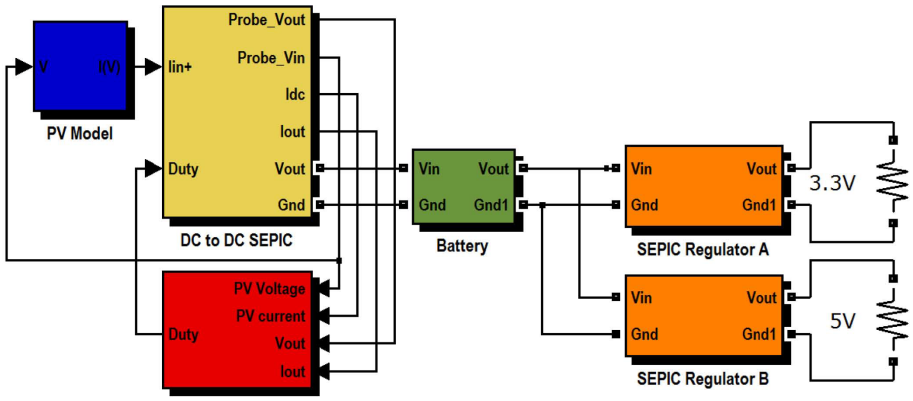

Fig. 1.8. Simulink model of the SPICA CubeSat EPS developed at University of Puerto Rico [48, p. 1793].

1.1.3 Motivation

As emphasized in the previous sections, neither experimental test stands nor simulation tools with or without HIL, have demonstrated their suitability for direct OBC code verification developed in C or C++ without the need of additional hardware or software development efforts. Test stands only allow certain tailored test cases out of all tests required to e.g. verify the functionality of an ADCS or AOCS

algorithm. Software toolkits such as STK or Simulink require an additional level of abstraction and, if combined with HIL, additional embedded systems, firmware and software need to be developed.

The CubeSim simulation framework was developed to overcome these deficits and to allow fast development and verification of CubeSats or small satellites. This not only cuts the costs and development time but also allows to perform more sophisticated verification tests that could not be done experimentally. Major design drivers and development decisions are described in detail in the next sections.

1.2 Problem Formulation and Objectives

In this section the problems being addressed with the development of framework CubeSim are discussed and the related challenges are explained. The difference to other available software and hardware tools is also shown. The related objectives of this thesis are listed subsequently.

1.2.1 Problem Formulation

When the behavior of various systems and modules is modeled, the according tasks need to be executed in simulated real-time, such as module `Motion` (see section 4.5), responsible for the movement of celestial bodies and spacecraft, needs to compute and update the velocities and positions of the objects in parallel to an ADCS task responsible for the determination of the spacecraft orientation and the control of actuators, such as reaction wheels or magnetorquers. Real real-time was not considered as suitable because this would not allow to speed up the simulation run and would set stronger requirements on the simulation computer to be used. Simulated real-time is accomplished with so-called *fibers* implemented in a cooperative multi-threading environment (see section 2.2 for details).

One or more spacecraft can entirely be simulated within the CubeSim framework. This not only includes the aforementioned ADCS or AOCS algorithm, but can also include battery charge and discharge modeling, up- and down-link simulation and even rough thermal modeling can be performed. The most common modules were implemented to simulate the movement of celestial bodies and spacecraft, to simulate the Earth magnetic field (see section 4.4), to simulate sun light (see section 4.2) and Earth albedo (see section 4.3). The CubeSim framework was designed to be modular and expandable so that new modules for e.g. atmospheric drag or more detailed gravitational field computations can easily be implemented in the future.

For HIL testing, the OBC code is executed on hardware and the sensors and actuators of the spacecraft are abstracted by providing additional embedded systems with, e.g., SPI (Standard Peripheral Interface), I2C (Inter-Integrated Circuit) or analog interface, that are fed from STK or Simulink. One of the key differences between CubeSim and other existing test methods is that no such additional hardware is required. The output of sensors is directly computed by the framework and used in the OBC code (see chapter 9). It needs to be mentioned though that in this case the proper functionality of a digital communication interface cannot be verified with CubeSim since the framework is meant to be used on a higher level where data flows, control loops, safety mechanisms and failure mitigation strategies are implemented.

If the code of the OBC or other sub-systems is executed on hardware or if embedded systems are used to simulate sensors and actuators, it is only possible to run the test in real-time. For shorter tests, such as the verification of the ADCS detumbling algorithm, this deficit might be acceptable but even for parameter tuning, where tests are performed multiple times with different settings, this approach can be too time-intensive. To verify the orbital propagation of a spacecraft revolving around the Earth, a real-time test would last more than 90 minutes for a single revolution. On the other hand sophisticated

simulations e.g. involving complex albedo computations or even ray-tracing could not even be performed in real-time on usual computers if high sensor sampling rates are demanded. Both cases are resolved by the use of the CubeSim framework and its implementation of simulated real-time.

Since broken components or sub-systems of a spacecraft cannot be repaired or replaced after launch, mission-critical components are duplicated to increase the redundancy. If a sensor is broken, does not update its output value or provides erroneous signals, the software processing the signals must be able to detect the error and consequently needs to switch to the redundant component if possible. CubeSim allows to disable or re-enable each sensor and actuator and also the sensor noise can be modified during the simulation run, if required (see e.g. section 9.5).

In order to assess the performance of an algorithm, test vectors are commonly used which consist of a set of input and expected output values [5, p. 1]. This can either be accomplished by so-called *white-box* or *black-box* tests [68, p. 25ff]. For more complex algorithms, such as a detumbling strategy as part of an ADCS, this validation concept can be extended in CubeSim. The simulation can be set up and run with different initial parameters or boundary conditions. For the presented example, the initial spin rate or the axis of rotation of the spacecraft are varied which can be of particular interest due to the different momenta of inertia around the principal axes (see section 3.8). The gathered results help to optimize the algorithm or tune control loops.

The CubeSim framework on the one hand contains predefined modules (see chapter 4) and systems (see chapter 9) that can be combined for the required application or test scenario, but on the other hand the modular development approach allows straightforward development of extensions to serve custom requirements. With predefined sensors, such as GNSS trackers, gyroscopes, accelerometers, photo detectors or magnetometers, the user can derive from these classes to implement a specific sensor with a certain range and accuracy. The same applies to predefined actuators such as thrusters, magnetorquer or reaction wheels.

The CubeSim framework also allows to implement additional sensor or actuator types, e.g., horizon sensors [154, 140], star trackers [67, 260] or solar sails [221, 130]. Modules, modeling the interaction between the elements of a simulation based on physical principles, can also be extended or new modules can be developed such as representing a more accurate gravity field [261, 168] of the Earth or computing atmospheric drag effects [34, 177].

The framework is entirely developed in C++ to allow high computational performance [210] and to allow extensions to be developed in a standardized and commonly used programming language [103]. For the development of the different modules, other tools were developed in Python, PHP (Hypertext Preprocessor) and Microsoft Excel to import e.g. ephemeris data from the NASA JPL Horizons service [166] or to post-process output data from CubeSim.

The so-called *V-model* (see fig. 1.9) describes the implementation and testing phases of a software development process [215, p. 43], wherein the verification is performed on unit, sub-system and system level and finally the software is validated against the stakeholder top-level requirements [200, p. 11]. The CubeSim framework can be used on unit, sub-system and system level to verify proper functionality of, e.g., a modeled sensor (unit), data handling and signal merging (sub-system) or the ADCS algorithm (system). It also allows to validate the top-level requirements, such as the maximum detumbling time or the pointing accuracy of the satellite.

1.2.2 Thesis Objectives

This thesis shall provide a detailed insight into the initial considerations, requirements, development, implementation, verification and use of the CubeSim framework. For project QB50, an initiative of the

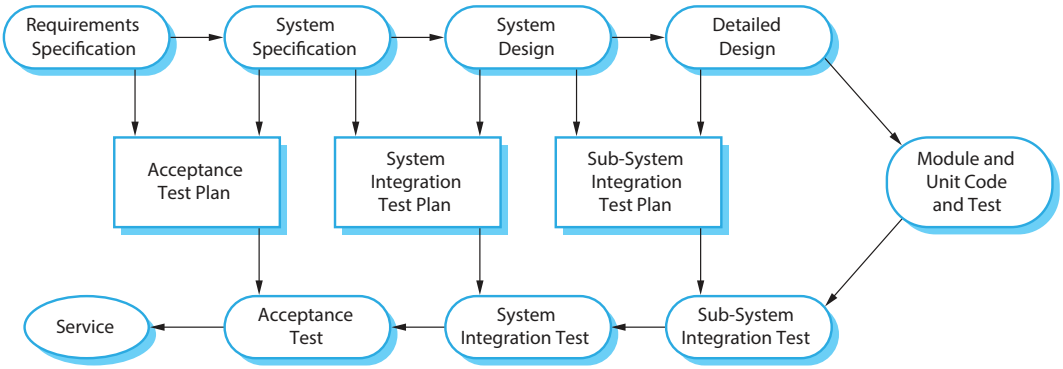

Fig. 1.9. Software development process according to the V-model [215, p. 43].

von Karman Institute and funded by the European Commission [116], the University of Applied Sciences Wiener Neustadt provided a contribution with the 2U CubeSat PEGASUS [203].

The aim of this activity was the development of most components and systems in-house including the ADCS system. It consists of various sensors and actuators and the purely experimental approach for verification of the algorithm was already excluded from the beginning. This represents the starting point for the development of CubeSim.

The requirements for the framework are to be discussed in the light of project PEGASUS and subsequent use for simulations of small satellites or (sub-)systems in general. A trade-off between complexity and functionality shall be explained allowing the user to set up a simulation quickly to obtain the desired results without the need for a deep understanding of the entire framework.

The derived class hierarchy shall be presented which was set up and optimized for easy use, high performance and maximum flexibility and expandability. The relation between the instantiated objects and their purpose in CubeSim shall be made understandable to the reader. The implementation, coding and the implicit design decisions shall be explained and justified.

The properties and methods of the developed helper classes, modules, systems, assemblies, parts and celestial bodies shall be described in detail and verification of the implemented code shall be performed by different means, such as unit tests, the comparison with literature or with output from third-party software. This particularly applies to predefined modules `Gravitation`, `Light`, `Albedo`, `Magnetics`, `Motion`, `Ephemeris` and systems `GNSS`, `Gyroscope`, `Accelerometer`, `PhotoDetector`, `Magnetometer`, `Magnetorquer`, `Thruster`, `ReactionWheel` which are crucial for the integrity of the entire simulation. Details on the verification tests can be found in chapters 4 and 9 respectively.

Chapter 2

Implementation

The CubeSim framework is developed and implemented using a modern and object-oriented programming approach, which is described in detail in this chapter. This not only includes the proper definition of encapsulated classes but also the exploitation of features, such as inheritance, polymorphism, generic programming and exception handling. The concept of simulated real-time is explained which allows the quasi-parallel execution of multiple tasks describing the behavior of systems and spacecraft. On operating system level, fibers are used for that purpose. In order to increase the simulation performance of the framework, the selected programming language for CubeSim is justified and other performance boosters, such as proper configuration of modules and systems or caching techniques, are described.

2.1 Modern Programming Techniques

In this section the design and coding techniques used for the development of the CubeSim framework are pointed out. These are in particular the object-oriented design approach as one of the pillars of C++, the special methods being implemented for various classes, inheritance, polymorphism and virtual methods being used for similar objects, such as physical parts, generic programming and templates for code reuse and the rationale behind error and exception handling and their application within the framework.

2.1.1 Object-oriented Design

Programs developed in procedural languages, such as C, Pascal or Fortran, represent a list of instructions that are executed one-by-one. With increasing program complexity, subsets of instructions were put into so-called *functions*, *subroutines*, *subprograms* or *procedures* to allow better reading of the code and code reuse. This allows to divide a procedural program into functions with a clear interface and purpose [126, p. 10].

This approach of grouping several functions serving a certain purpose was extended by the introduction of so-called *modules* or *libraries*. The division of program into functions and modules is referred to structured programming. For decades the procedural programming paradigm has been used but with steadily-increasing program sizes and complexity, two major problems solidified: first, all functions have unrestricted access to global data [126, p. 11ff] and second, this paradigm poorly models the real world [186, p. 14].

The OOP (Object-Oriented Programming) paradigm supports encapsulation and information hiding (e.g., declaration of private methods and fields), thus simplifying the interface of a class. Apart from the better

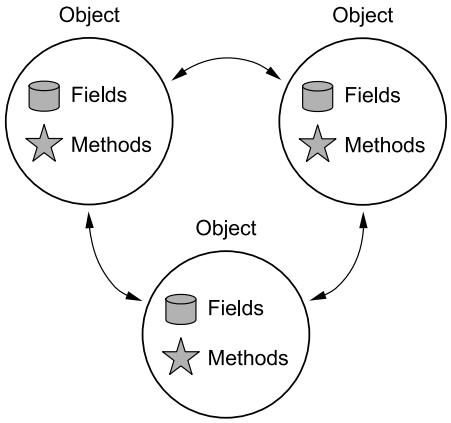

Fig. 2.1. Object-oriented paradigm [126, p. 14].

maintainability, classes and objects help to model physical counterparts. Within the CubeSim framework these are for example parts, assemblies, systems, spacecraft or celestial bodies. A well-designed object is an independent entity and the operations performed modify or read its attributes which is sometimes termed encapsulation [202, p. 19f]. Objects interact with other objects by calling their respective member functions as shown in fig. 2.1. A programming problem is no longer split into functions but divided into interacting objects [126, p. 16].

In general, objects represent different entities such as physical objects (e.g., parts or components), elements of the computer-user environment (e.g., menus, widgets or peripheral components), data-storage constructs (e.g., containers, files or databases), persons (e.g., employees or customers) or complex data types (e.g., time, complex numbers or coordinates). Classes define these objects in terms of fields and methods and objects are sometimes referred to class instances [126, p. 17]. In CubeSim, apart from POD (Plain Old Data) data types [106, p. 239], everything is an object which can either be directly accessed by the user or the objects are directly interacting with each other. The definition of classes such as `Module`, `Part`, `Assembly`, `System`, `Spacecraft` or `CelestialBody` can be found in the subsequent chapters.

2.1.2 Special Member Functions

When objects are created, copied, moved or destroyed, special member functions are called implicitly [103, p. 185ff]. When an object is generated, the so-called *constructor* is involved to initialize the object [126, p. 227ff] which may include, e.g., memory acquisition or device initialization. A constructor which does not expect any arguments is called *default constructor* which is automatically generated by the compiler if not explicitly defined [95, p. 225]. Within the CubeSim framework most classes feature explicitly defined constructors to allow the user straightforward object generation without the involvement of additional methods (see e.g. chapter 7).

When an object is copied, the so-called *copy constructor* is used which expects a constant reference [103, p. 132f] to the object being copied. Similar to the default constructor, the default copy constructor is implicitly used if not explicitly defined [126, p. 238ff]. It makes a shallow copy of all fields [132, p. 98ff]. Similar to the copy constructor is the so-called *move constructor* which expects a so-called *prvalue* reference [106, p. 83] to the object being moved [106, p. 301]. By the use of move semantics, deep copies of fields of temporary objects are avoided [132, p. 101ff].

Finally, when an object is explicitly destroyed using operator `delete` [103, p. 81f] or by leaving its declaration scope [103, p. 24ff], the so-called *destructor* is involved which deinitializes the object which may include, e.g., memory release or closing device handles. When the destructor is not explicitly defined,

a so-called *default destructor* is created which automatically calls the destructors of all fields when an object is destroyed [126, p. 232f].

2.1.3 Inheritance and Polymorphism

Closely related to classes is inheritance which allows to divide classes into more specific sub-classes, e.g., a geometric body is divided into boxes, cones, cylinders, prisms and spheres. The idea is to share common properties such as the color or material but also to allow specific definitions in the individual sub-classes - a box is defined by its length, width and height but a sphere is defined solely by its radius (see chapter 7 for details). In C++ the original class is called *base class* and the inherited sub-classes are called *derived classes* [103, p. 163]. In order to strengthen encapsulation, in C++ the visibility of fields and methods can be defined with keywords `private` meaning that the members can only be accessed from the base class, `protected` meaning that the members can be accessed from the base class and all derived classes and `public` meaning that the members can be accessed from other objects as well [103, p. 182 and 318].

Other features that is strongly used by the CubeSim framework are polymorphism and virtual methods [230, p. 585]. In C++ the keyword `virtual` can be used to declare a member function that is to be implemented differently for derived classes [132, p. 414ff]. In class `RigidBody` (see chapter 6) the surface area, volume, inertia and center of mass are to be computed for any geometric primitive in a different way. If a virtual method is called, the sub-class is determined automatically and the corresponding implementation of the virtual method is being called implicitly.

For example, when the volume of a solid is to be computed, the pure virtual method `RigidBody::_volume` is implemented [103, p. 172f] in accordance with eq. (161) for a box or with eq. (189) for a sphere. This can be particularly helpful considering an assembly (see chapter 8) which consists of multiple and different parts or sub-assemblies. To compute the total volume of such an assembly, method `RigidBody::volume` of the individual parts is called and the individual masses are accumulated according to eq. (192). No selection statements such as `if` or `switch` are required to manually determine the sub-class type and to call the corresponding method explicitly [88, p. 106f].

In CubeSim, polymorphism ensures straightforward expandability as new parts, assemblies, systems, spacecraft or modules can be derived from predefined classes or from the corresponding base class. The implementation of abstract virtual methods allows easy integration into the CubeSim object landscape.

2.1.4 Generic Programming

In contrast to polymorphism, the same method or operator of a class can be defined multiple times by changing the number of parameters or their types, which is called *overloading* [132, p. 94ff]. This feature can particularly be useful to allow reuse of common operators on newly defined data types, such as `operator ==` to compare two objects. Overloaded constructors are commonly used for most classes to allow different ways to initialize an object [126, p. 234f].

By the use of *generic classes*, the concept of overloading is extended to arbitrary data types that do not even need to be known during development. One or more data types that are used within a class - either for fields, method parameters or return values - can be parameterized [132, p. 48f]. This can particularly be useful for container classes, such as `vector`, `list` or `set` defined in the C++ STL (Standard Template Library) where elements of an arbitrary data type can be stored [144, p. 11ff]. The actual choice of the data type is made when instantiating the class.

The concept of generic programming enables code reusability and avoids function or method overloading for certain applications. Within the CubeSim framework templates are used for classes `List`, where

different elements - such as parts, assemblies or spacecraft - are stored (see section 3.1), `Polygon` which consists of vertices of arbitrary type (see section 3.14), `Matrix` representing square matrices of arbitrary element type with the derived type `Matrix3D` (see section 3.6) and `Vector` representing a vector of arbitrary length and element type with the derived types `Vector2D` (see section 3.2) and `Vector3D` (see section 3.3).

2.1.5 Error and Exception Handling

Systems are exposed to external influences beyond normal operation caused by attack, erroneous inputs, hardware or software faults, unanticipated user behavior or other environmental changes and are supposed to continue to provide certain services [232, p. 5]. One distinguishes between *fail safe*, where the system enters a safe state in case of a detected failure, *fail soft*, where the system continues to perform some of its tasks, and *hard failure*, where the system is halt immediately [98, p. 32 and 36]. Coding standards such as [4], [18] and [88] provide programming rules to help the developer to build reliable and safe software.

Procedural programming languages, such as C, do not provide sophisticated error handling mechanisms. Functions that can fail, like `FILE* fopen(char* name, char* mode)` opening a file for read or write access, return special values in case of error. If the file does not exist or cannot be accessed, the function returns `nullptr` and the error code is assigned to the global variable `errno` defined in header `<errno.h>` [115, p. 248]. It is the responsibility of the caller to check the return value of the called function [232, p. 23]. In contrast to global error indicators, there are other functions that indicate an error only by its return values, such as `void* malloc(size_t size)` returning `nullptr` if no memory could have been allocated. Functions returning error codes cannot return values for other uses and checking every return code can increase the code size by up to 40% [232, p. 27].

In order to detect any logic flaws in the program code, so-called *assertions* can be used. The macro `assert(int expression)` is defined in `<assert.h>`. If the passed parameters is evaluated to 0, an error message is printed on `stderr` and the program is terminated by calling `abort(void)` [115, p. 252ff]. When code is deployed, assertions are generally disabled by defining the `NDEBUG` macro prior to compilation [232, p. 32].

Closely related to error handling is proper release of acquired resources, such as closing open file handles or freeing allocated memory. In C so-called *goto chains* are commonly used [232, p. 39ff], [129, p. 12]. Other coding standards prescribe to avoid use of keyword `goto` in general [20, p. 14], [214, p. 15] or for purposes other than cleaning up resources in case of errors [151, p. 1].

To avoid the necessity to manually keep track of acquired resources and to release them under error conditions on the one hand and to provide a systematic, object-oriented approach to error handling in C++ on the other hand, so-called *exceptions* were introduced [103, p. 291ff]. Exceptions are thrown (keyword `throw`) in case of errors at runtime caused by a variety of exceptional circumstances, such as running out of memory or the inability to open a file [126, p. 703]. Certain restrictions on the use of exceptions are recommended, such as the avoidance of exceptions in the destructor [88, p. 137]. Exceptions are caught (keyword `catch`) by the calling functions and all objects going out of scope in the calling tree are automatically destructed. RAII (Resource Acquisition Is Initialization) is a C++ design principle where the lifetime of an acquired resource is linked to the lifetime of an object [232, p. 43]. With the destruction of the object also the bound resources are automatically released.

The CubeSim framework uses exceptions to handle error conditions and all exceptions are derived from the base class `Exception` as shown below.

```
// Class Exception
class CubeSim::Exception
{
public:

   // Class Failed
   class Failed;

   // Class Internal
   class Internal;

   // Class Parameter
   class Parameter;

private:

   // Virtual Function for RTTI
   virtual void _func(void);
};
```

The private virtual method `_func` is required to build a polymorphic exception hierarchy that for example allows dynamic type casting of caught exception objects [103, p. 70f] or for RTTI (Runtime Type Information) handling in general [132, p. 382ff].

Class `Failed` is thrown when the execution of an operation could not be performed due to unmet preconditions of the object itself or due to the combination of the object's state with the arguments passed to the method - such as the computation of an orbit from state vectors (see section 3.13) or the decomposition of a matrix (see section 3.6).

An internal exception of type `Internal` is thrown when API (Application Programming Interface) calls were not successful - such as the creation or the handling of fibers (see section 2.2.4).

Whenever a parameter passed to a method is invalid or out of range, exception `Parameter` is thrown - such as the dimensions of parts (see chapter 7), the accuracy setting of systems (see chapter 9) or the propagation time step of modules (see chapter 4).

2.2 Multi-threading Simulation

The different components of a CubeSim simulation, such as modules, spacecraft or systems, offer user-defined code to model their behavior. To implement the aforementioned simulated real-time, a multi-threading environment is implicitly set up before the CubeSim simulation can be started. The difference between processes, threads, coroutines and fibers and their implementation within the framework are explained in this section.

2.2.1 Processes

A process is a program in execution consisting of program code also known as text section and a set of data associated with the code, such as a unique identifier, the process state, execution priority, program counter, context data (stack) or memory pointers [213, p. 102ff]. This information stored in a data structure is typically called PCB (Process Control Block) [222, p. 109ff]. The process can either be in new, ready, running, waiting or terminated state as shown in 2.2. When the process was created and is schedulable, its state only switches between ready, running and waiting state. In the latter state the

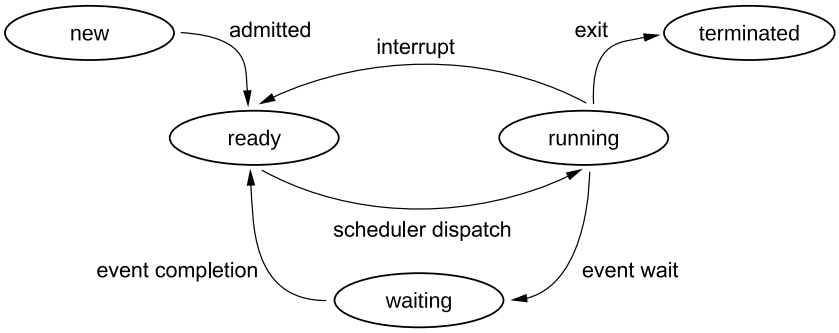

Fig. 2.2. Typical diagram of process states and transitions [213, p. 103].

process is waiting for an event or for a blocked resource, such as an I/O device or a file. The scheduler periodically interrupts the execution of the process by switching it from running to ready state. When execution shall be resumed, the process is switched back from ready to running.

While processes or tasks include a virtual address space and have the ownership of memory, I/O (Input/Output) devices and files, so-called *threads* or *lightweight processes* share these resources [222, p. 159]. One or more threads belong to a single process and no thread can exist outside a process. IPC (Inter-Process Communication) between processes is slow compared to inter-thread communication due to the separate virtual address space. Pipes or named pipes [136, p. 13], [81, p. 27ff], message queues [81, p. 32ff], semaphores [81, p. 46ff] and shared memory [81, p. 62ff] are commonly used for IPC.

2.2.2 Threads

Multithreading is the ability of an operating system to support multiple threads within a single process but the implementation is operating system-dependent. One distinguishes between user and kernel-level threads as described in [222, p. 165]. While processes can enter a variety of states, depending on their operating system-specific implementation, threads are usually either in ready, running or blocked state. The information required for the execution of a thread and for context switching, is stored within a TCB (Thread Control Block).

A scheduling strategy allows switching the execution between the individual threads. A so-called *preemptive scheduler* using the round-robin algorithm [213, p. 194ff], interrupts the thread in execution after a certain interval (so-called *time quantum* or *time slice*), puts it in ready state and continues execution of another thread in ready state. Similar to a process, when a thread is put into ready state, its context, such as registers, the stack pointer and the program counter, need to be saved and for the thread being put in running state, its context needs to be restored prior to resumed execution.

As the context switch can occur unexpectedly at any point of the program code, so-called *racing conditions* can occur, where two or more threads try to access a shared resource, such as a variable, simultaneously. This can lead to corrupt data or other undesirable side effects. Operating systems offer the concept of semaphores or mutexes to disarm these racing conditions [35] when a mutex is acquired before the code enters the critical section and is released when leaving the section again. In contrast to preemptive scheduling, in the case of so-called *non-preemptive* or *cooperative scheduling*, the threads are not interrupted but the thread can decide when to stop executing and return control to the dispatcher [213, p. 185f]. This allows to explicitly return control to the dispatcher once a critical section has been left.

2.2.3 Coroutines

Coroutines are subroutines that allow non-preemptive multi-threading by allowing execution to be suspended and resumed [22, p. 2]. In the programming language C this concept is implemented as non-local jumps defined in the header `<setjmp.h>`. The macro `int setjmp(jmp_buf env)` sets up the data structure `jmp_buf` for holding the information needed to save and restore a calling environment. The macro returns multiple times while it returns 0 when invoked directly and otherwise returns the value passed to function `longjmp`. The function `void longjmp (jmp_buf env, int val)` allows a context switch by passing the data structure of type `jmp_buf` and the return value for `setjmp` [104, p. 242ff].

2.2.4 Fibers

Single-threaded UNIX server applications serving multiple clients use their own threading architecture to simulate multi-threading behavior. The underlying libraries create multiple stacks and save and restore CPU (Central Processing Unit) registers during context switch. Microsoft added this concept of fibers to Windows to facilitate porting of UNIX server applications [197, p. 358ff]. The memory footprint of fibers is ca. 1% smaller compared to threads since the kernel context and stack does not need to be saved. The stack size of fibers can either be fixed or dynamic with incremental growth if required [174, p. 2f].

Since switching from fiber to fiber does not involve kernel transitions, the cost of fiber switches is cheaper than the one of thread switches but it needs to be noted that fiber switches still have significant costs compared to conventional functions calls. Details on fibers and more information on the difference to threads can be found in [174].

2.2.5 Implementation

To satisfy the multi-threading requirements posed by the CubeSim framework, fibers were chosen as the most suitable concept. For CubeSim the class `Fiber` was developed to wrap the Windows API functionality on fiber creation and handling. The class is defined as follows excluding certain private declarations.

```
// Class Fiber
class Fiber
{
public:

    // Class Exception
    class Exception;

    // Function
    typedef void (*Function)(void* data);

    // Suspend
    static void suspend(void);

    // Constructor
    Fiber(Function function, void* data = nullptr);

    // Destructor
    ~Fiber(void);

    // Get Data
    void* data(void) const;
```

```
    // Check if done
    bool done(void) const;

    // Get Function
    Function function(void) const;

    // Run
    void run(void) const;

private:

    // Dispatcher
    static void CALLBACK _dispatch(void* parameter);
};
```

The constructor `Fiber(Function function, void* data = nullptr)` creates a new object of type `Fiber` with the program code implemented in function `function` and the optional parameter `data` being passed to the specified function. The Windows API function `LPVOID ConvertThreadToFiber(LPVOID lpParameter)` is called by the constructor to convert the current thread into a fiber which is required before other fibers can be scheduled. This makes the current thread the main fiber to which control is passed when a running fiber is suspended (see below). Then a Windows fiber is created with the API function `LPVOID CreateFiber(DWORD dwStackSize, LPFIBER_START_ROUTINE lpStartAddress, LPVOID lpParameter)` [197, p. 359]. The function pointer is returned by method `function` and the optionally passed data pointer can be retrieved by method `data`. Both methods do not expect any arguments.

The destructor `~Fiber(void)` destroys the object and calls the Windows API function `VOID DeleteFiber(LPVOID lpFiber)` [197, p. 361].

The program code of a fiber is usually implemented as infinite loop but class `Fiber` also accepts returning functions which can be useful for complex parallelized computations. When the function returns, a flag is indicating that the execution has terminated. This flag can be read out by method `done` called without arguments.

To switch to a fiber, method `run` is called without arguments. The Windows API function `VOID SwitchToFiber(LPVOID lpFiber)` is used for the context switch [197, p. 360]. The internal method `_dispatch` is used to handle the call of function `function` when method `run` is invoked and also handles if the function returns.

When the execution of the function code of a fiber shall be interrupted to allow switching to another fiber, the static method `suspend` is called. The method does not expect any arguments. The Windows API function `SwitchToFiber` is also used for the context switch to the main fiber which is acting as the dispatcher calling the next fiber by the use of method `run`. Such a dispatcher usually stores a list of created fibers to be called sequentially. For the CubeSim framework this functionality was extended in class `Simulation` (see chapter 12) to allow fibers to block their execution for a certain time. In this manner the aforementioned simulated real-time is realized.

In fig. 2.3 an example with three tasks is shown. Each of the tasks consists of code execution (black bars) and idle times (red, blue and green bars respectively). An arrow of time at the bottom with arbitrary time units shows the progress of simulated real-time. At the beginning after one time unit, the scheduler runs task 1 which enters idle state for four time units after execution. Afterwards task 3 is run which enters idle state for two time units after execution. Now all three tasks are idle and the simulated real-time is increased by one time unit until task 2 is to be run for the first time. After execution of task 2, it is entering idle state for five time units. Now all tasks are idle again for two time units and the simulated

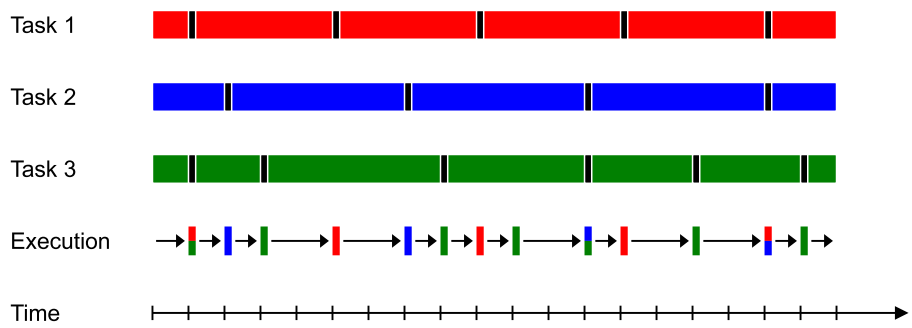

Fig. 2.3. Task execution in simulated real-time. Whenever all tasks are blocked (being in idle state), the real-time is increased (denoted with arrows).

real-time is increased by that value. Then task 1 is run again, then the real-time is increased by two time units, then task 2 is executed again and so on. It needs to be noted that the execution of a task is happening instantaneously in simulated real-time as an increase of the simulated real-time only occurs when all tasks are in idle state.

2.3 Performance Optimizations and Caching

Every simulation framework should be implemented in an efficient way and be as resource-saving as possible. This allows the user to rerun simulations without having to wait for the results unnecessarily long. Or vice versa, more parameter sweeps could be performed in a given period of time, for example, to allow fast tuning of control loops. In this section several performance optimizations are discussed, such as the selection and justification of the underlying programming language in which CubeSim is implemented. Some of the predefined modules feature options that allow the user to make a trade-off between accuracy and simulation time. The caching techniques used in the framework and implementation details thereof are also explained in detail.

2.3.1 Programming Language Selection

For the development of a component, a sub-system or an algorithm, such as for the ADCS, various unit tests and mission scenarios need to be performed. In order to be able to complete these tests in a reasonable time period, the simulation framework needs to be implemented efficiently. As already pointed out in section 2.1, modern programming techniques were used in the development of CubeSim, such as object-oriented programming, inheritance and polymorphism, generic programming and templates and exception handling. These features are offered by various modern programming languages but in the previous section the focus was already laid on C++.

This section compares the performance of different popular programming languages. Some of these languages use so-called *compilers* to parse the source code once and to translate it into a machine-readable sequence of operations and others use so-called *interpreters* which parse the source code and directly execute it [211, p. 23f]. In the latter case, the separate compilation step is omitted allowing more dynamic source code modifications, but a performance loss has to be accepted. Another approach lies in between where a compiler generates so-called *bytecode* that cannot be executed directly on a real computer but on a virtual machine, such as the JVM (Java Virtual Machine) [134], which interprets the bytecode and directly executes native machine instructions. To overcome the performance penalty caused by bytecode interpretation, a just-in-time compiler was introduced which translates the bytecode

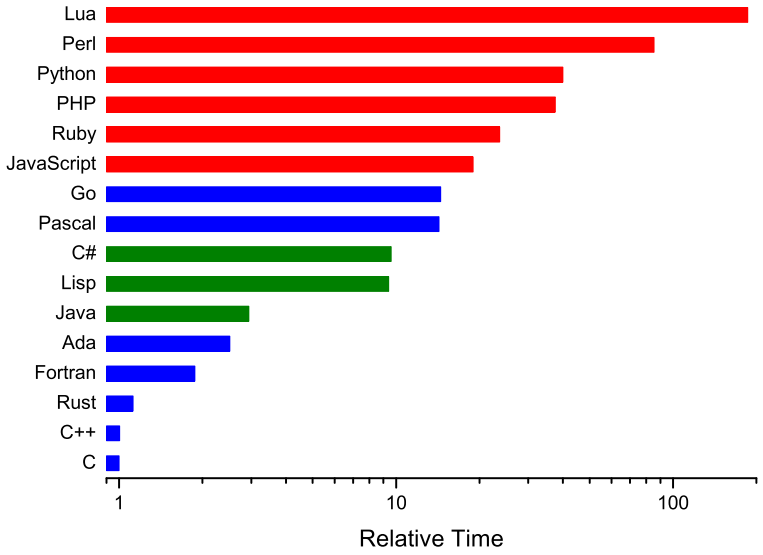

Fig. 2.4. Relative execution time of selected programming languages for binary tree computations: compiled (blue), byte code (green), interpreted (red) [183, p. 261].

into native machine language while executing the program. These translated parts of the program can be executed considerably faster the next time [63, p. 6ff].

A binary tree algorithm was implemented by Pereira et al. in a variety of different programming languages [183] and the execution time was measured and compared (see fig. 2.4). One can clearly recognize that compiler-based languages show higher performance than interpreter-based ones - as anticipated. Between C and C++ the difference is marginal.

A survey carried out in 2006 revealed that the majority of existing and future embedded projects were programmed in C (51%) followed by C++ (30%) and assembly language (8%) [152, p. 332]. In [19, p. 18ff] the use of C and C++ for embedded systems and the related advantages and disadvantages are discussed. As one major objective of CubeSim is the direct execution of OBC source code, C++ was chosen for the framework implementation which consequently enables the direct use of C and C++ code.

2.3.2 Performance Optimization

For the development of CubeSim, the modules, systems and the framework itself were provided with various options and settings to allow the user to configure the simulation appropriately. This enables him or her to perform a trade-off such as between accuracy and simulation duration or between accuracy and memory consumption. Though the mentioned modules and systems will be described in chapters 4 and 9, their tuning options are explained in this section in detail.

If the initial conditions for a simulation can be well defined, the most obvious approach to reduce computation time is to carefully select the start and end times to focus on the relevant processes within the simulation. The start time can be set with method `Simulation::time` and by the use of method `Simulation::run` either the simulated real-time duration in seconds or the end time can be passed as an argument (see chapter 12). Module `Ephemeris` can be used to initialize the position, orientation and velocity of the used celestial bodies and predefined spacecraft at the specified start time (see section 4.6). The helper class `Orbit` is used to transform the Keplerian orbit elements into state vectors in the local Cartesian frame (see section 3.13).

Due to the modular approach of CubeSim, only modules, spacecraft and celestial bodies that are relevant for the simulation, can be used. If for example the orbital propagation of a spacecraft is to be simulated,

only modules `Gravitation` and `Motion` and the nearest celestial bodies need to be included in the simulation - other modules, such as `Light` or `Albedo` would only increase the simulation duration without affecting the outcome. Also omitting outer planets in the solar system does not significantly affect the orbit of a spacecraft in LEO but due to the related n-body problem, the simulation duration would be unnecessarily prolonged (see sections 4.1 and 4.5).

Modules `Gravitation` and `Motion` continuously compute the gravitational forces between all celestial bodies and spacecraft in the simulations and update their position, velocity and orientation respectively. These computations and more precisely the update intervals can be modified with methods `Gravitation::time_step` and `Motion::time_step` which expect the new intervals $\triangle t_{\mathrm{G}}$ and $\triangle t_{\mathrm{M}}$ in seconds as an argument. Though both modules are acting independently, it is recommended to use the same time step for both modules $\triangle t = \triangle t_{\mathrm{M}} = \triangle t_{\mathrm{G}}$, otherwise module `Motion` would work with an outdated set of forces ($\triangle t_{\mathrm{M}} < \triangle t_{\mathrm{G}}$) or module `Gravitation` would compute these forces unnecessarily ($\triangle t_{\mathrm{M}} > \triangle t_{\mathrm{G}}$). Longer intervals can speed up the simulation run but verification tests of both modules have shown a strong influence of the time step on the propagation accuracy - in particular for months or years of simulated real-time - since integration errors add up over time (see section 4.5).

Module `Light` allows the user to choose between two models to compute the irradiance from a star on a celestial body or spacecraft: the star is modeled (i) as a point light source or (ii) as a disk pointing towards the observer. The user can select the model by the use of method `Light::model` by passing either `Light::MODEL_POINT` or `Light::MODEL_POINT`. Since the distance to a star is rather large in most simulation scenarios, model (i) is usually sufficient which helps to reduce simulation efforts. If model (ii) is being used, the number of points on the disk can be specified by the use of method `Light::resolution` (see section 4.2). It is relatable that with an increasing number of points, the simulation effort increases.

For module `Albedo`, which computes the diffuse reflection from an irradiated celestial body on a spacecraft, a grid is placed over the surface of the celestial body. For every point of the grid, the irradiance, incident and emergence angle and emissivity need to be considered which is very time consuming to calculate. The number of points can be specified by the use of method `Albedo::resolution` (see section 4.3).

Systems `Magnetorquer`, `ReactionWheel` and `Thruster` represent actuators which allow modification of the attitude or orbital parameters - such as the altitude, eccentricity or inclination - of a spacecraft. The computation of the acting torques or forces on a system and subsequently on the spacecraft is performed regularly at certain intervals $\triangle t_{\mathrm{S}}$. These can be modified with methods `Magnetorquer::time_step`, `ReactionWheel::time_step` and `Thruster::time_step` which expect the new interval $\triangle t$ in seconds as an argument. As module `Motion` is combining all forces and torques acting on the spacecraft to propagate its position, velocity and orientation, the considerations from above apply as well. It is recommended to use the same time step for module `Motion` and the aforementioned systems (see sections 9.6, 9.8 and 9.7).

The other predefined systems `GNSS`, `Gyroscope`, `Accelerometer`, `PhotoDetector` and `Magnetometer` compute their output values on demand and do not require time step settings. These values are not cached and simulation efforts can only be reduced by lowering the sampling frequency (see sections 9.1, 9.2, 9.3, 9.4 and 9.5).

2.3.3 Caching Techniques

Caching is mechanism that stores data that has already been requested or computed in order to speed up future requests for those calculations or data. CPUs commonly work at higher core frequencies than attached Flash or RAM (Random Access Memory). Memory access requires the insertion of idle cycles (so-called *wait states*) to retrieve the data as explained on the basis of an Arm-based 32-bit microcontroller here [229, p. 80ff]. The latency can be reduced by caching where a small and fast but expensive

memory is placed between the processor and slow and large but inexpensive memory. That way, read requests to memory locations that are cached can be satisfied quickly. Modern processors use multiple levels of caches but already the first-level cache can satisfy 90% of all requests in most cases [212, p. 115]. A similar approach is used for database servers where queries are parsed, processed and a so-called *execution plan* [76, p. 451ff] is cached to reduce subsequent physical reads [112, p. 8f].

For the CubeSim framework, caching is used to avoid the recalculation of certain properties even though the initial values have not changed. This is particularly relevant for rigid body physics where for example the mass of an assembly does not change if the masses of the containing parts and sub-assemblies remain constant. In this case the mass of the assembly is only computed once according to eq. (194) and the result is cached. If the mass of the assembly is requested one more time, the already calculated and cached value is immediately returned. Consequently, however, the question arises as to how long the value retains its validity and how it can be recognized that the value must be recalculated in the course of a new query. A hierarchical update processing approach was chosen for the implementation of the framework. In the presented example, whenever the mass of a part is updated - to, e.g., simulate fuel consumption - an event is internally triggered by calling `RigidBody::_update` and passing the property to be updated, such as `RigidBody::_UPDATE_MASS`. This not only invalidates the possibly cached mass of the part, but also calls the update method of the related assembly `RigidBody::_rigid_body->_update`, if applicable. This method in turn invalidates the mass of the assembly and would trigger automatic recalculation upon request.

This simple example shall show the basic principles of the applied caching techniques but it should be said that these are not limited to static properties. The wrench acting on a rigid body as a result of multiple forces and torques applied is calculated with eqs. (46) and (47) and cached accordingly. The computation of the momentum in dependence of the mass, center of mass or velocity or the angular momentum in dependence of the inertia, center of mass or angular rate is cached as well according to eqs. (155) and (154). Details on the dependencies and the hierarchical update processing approach are described in section 6.1.

As explained in section 2.3.2, the output values of predefined systems `GNSS`, `Gyroscope`, `Accelerometer`, `PhotoDetector` and `Magnetometer` are computed on demand. One the one hand, this can save computation time, if the output values are not calculated very often, but on the other hand, might lead to avoidable recalculations for recurring requests. One approach to improvement would be caching of the value after its calculation and keep it valid until the simulated real-time advances (see section 2.2.5). For example, if the GNSS transponder is requested by different tasks, such as ADCS, logging, or science, the position would only need to be calculated once per time step.

A large number of sensors use integrated ADCs (Analog-Digital Converters) and offer a digital communication interface such as SPI or I2C. These converters commonly offer so-called *single-sample* (or *single-shot*) and *free-running* mode as described in the datasheet of the ATmega128 micro-controller [14, p. 235]. In the first mode, a new measurement is to be triggered and after the conversion time, the result can be obtained. In order not to have to wait for the conversion result, the free-running mode can be used instead where the ADC is working continuously and the result is periodically updated when a conversion cycles is complete - as implemented in the digital temperature sensor LM95071 [235, p. 4]. This mode could be simulated in CubeSim by caching the value after calculation and keeping it valid until the simulated real-time advances by a predefined conversion or update time.

These two caching approaches for the output values of predefined systems are fundamentally different and the most appropriate approach is also dependent on the system and use case within the simulation. Therefore, caching of system values is not implemented by default but this functionality can easily be added by the user by deriving a class from the respective predefined system.

Chapter 3

Helper Classes

This chapter describes the classes that are used in many places of CubeSim. These are also accessible to the user to help her or him setting up the simulation or to process or transform the simulation results. Their purpose and application are explained in the dedicated sections.

3.1 List

Any created simulation results in a strict hierarchy of objects. The simulation might consist of several celestial bodies, physical modules or spacecraft. A Spacecraft might in turn consist of various systems and a system consists of multiple assemblies which are comprised of other assemblies and parts.

This functionality is accomplished by the use of lists. The formal definition of template class `List<T>` is shown in the listing below excluding private declarations.

```
// Class List
template <typename T> class CubeSim::List
{
public:

   // Class Item
   class Item;

   // Constructor
   List(void);

   // Copy Constructor
   List(const List& list);

   // Destructor
   ~List(void);

   // Assign
   List& operator =(const List& list);

   // Clear
   void clear(void);

   // Clone and insert Item
   T& insert(const std::string& name, const T& item);
```

```
    // Get Item
    const std::map<std::string, T*>& item(void) const;
    T* item(const std::string& name) const;

protected:

    // Remove and destroy Item
    virtual void _remove(const T& item);
};
```

A list of type `List<T>` contains multiple objects of type `T`.

The method `insert` is used to insert an object of type `T` or objects of derived classes into the list. It requires a unique name of the element to be inserted into the list.

The items are stored in an associative container and can be retrieved by method `item`. When no arguments are provided, an associative container of type `std::map<std::string, T*>` is returned. In order to find a specific item, the name needs to be passed to the method. If the specified item is found, a pointer to the item of type `T*` is returned, otherwise `nullptr` is returned by method `item`.

In order to remove all items from a list, method `clear` is to be used. The method destroys all elements by implicitly calling their destructors.

If an object is to be inserted into a list, a duplicate needs to be constructed first. Since a list may contain objects of different derived classes with a common base, the copy constructor cannot be utilized explicitly to replicate the objects. Only virtual methods are able to do so and copy constructors cannot be virtual [103, p. 185]. A common pattern to compensate for that is the use of virtual cloning methods [7, p. 211].

The assign operator `operator =(const List& list)` is defined to overwrite one list with another one. The assignee list is cleared by calling method `clear` before new elements are inserted.

The destructor `~List(void)` destroys all elements by calling method `clear`. The constructor, destructor and assignment operator obey the so-called *rule of three* as proposed in [230, p. 481].

Each object to be added to a list of type `List<T>` is required to provide method `clone` and therefore needs to be derived from base class `List<T>::Item`. A copy constructor `List(const List& list)` is provided to create a copy of an existing list which calls method `clone` to create the duplicates.

The formal definition of template class `List<T>::Item` is shown in the listing below. For reasons of clarity, private methods and member variables are omitted.

```
// Class Item
template <typename T> class CubeSim::List<T>::Item
{
public:

    // Clone
    virtual T* clone(void) const;

    // Remove and destroy
    void remove(void);
};
```

The class `List<T>` does not provide means to remove specific items. In order to remove an item `List<T>::Item` from the list `List<T>`, class `List<T>::Item` provides the method `remove` which actually calls the protected method `_remove` of class `List<T>` destroying the item and removing it from the list.

3.2 Vector2D

Two-dimensional vectors or positions are represented by objects of class `Vector2D`. These vectors are used, e.g., to specify the points of a polygon. The class is derived from the more general template class `Vector<double>` which stores an arbitrary number of elements of arbitrary type. The formal definition of class `Vector2D` is shown in the listing below.

```
// Class Vector2D
class Vector2D : private Vector<double>
{
public:

   // Unit Vectors
   static const Vector2D X;
   static const Vector2D Y;

   // Constructor
   Vector2D(void);
   Vector2D(double x, double y);
   Vector2D(const Vector<double>& v);

   // Get Element
   double& operator ()(size_t i);
   double operator ()(size_t i) const;
   double& at(size_t i);
   const double& at(size_t i) const;

   // Sign
   const Vector2D& operator +(void) const;
   const Vector2D operator -(void) const;

   // Compare
   bool operator ==(const Vector2D& v) const;
   bool operator !=(const Vector2D& v) const;

   // Multiply
   const Vector2D operator *(double a) const;
   double operator *(const Vector2D& v) const;
   Vector2D& operator *=(double a);

   // Divide
   const Vector2D operator /(double a) const;
   Vector2D& operator /=(double a);

   // Add
   const Vector2D operator +(const Vector2D& v) const;
   Vector2D& operator +=(const Vector2D& v);

   // Subtract
   const Vector2D operator -(const Vector2D& v) const;
   Vector2D& operator -=(const Vector2D& v);

   // Compute Z Coordinate of Cross Product
   double operator ^(const Vector2D& v) const;

   // Compute Angle
   double operator %(const Vector2D& v) const;
```

```
    // Compute Norm
    using Vector<double>::norm;

    // Compute Unit Vector
    const Vector2D unit(void) const;

    // X Coordinate
    double x(void) const;
    void x(double x);

    // Y Coordinate
    double y(void) const;
    void y(double y);
};
```

A null vector can be created with the default constructor `Vector2D(void)` which initializes both coordinates, x and y, to zero. By the use of constructor `Vector2D(double x, double y)`, the new vector will be initialized to the specified coordinates. The copy constructor `Vector2D(const Vector<double>& v)` is defined allowing to duplicate an existing vector.

The two elements can be either accessed by `operator ()` or by method `at`. The first element is accessed with index 1, the second element with index 2. Another way to access the elements is the use of methods `x` and `y` which select the first and the second element respectively.

The unary operator `operator -` negates all elements of the vector and the unary operator `operator +` returns a reference to the vector itself.

Two vectors can be compared by the use of operators `operator ==` and `operator !=`. In order to avoid erroneous results due to rounding errors, the following definition is being used when two values of C/C++ type `double` are compared:

$$a == b \stackrel{\text{def}}{\Longleftrightarrow} |a - b| \leqslant \varepsilon \tag{5}$$

where $\varepsilon = 100\varepsilon_0$ with the machine epsilon $\varepsilon_0 = 2.220446 \times 10^{-16}$ for C/C++ type `double` specifying an upper bound on the error caused by floating point operations. ε_0 is defined in the C/C++ standard library `<float.h>`.

A vector can be scaled with a scalar or the inner product between two vectors is computed with operator `operator *`. The operator `operator *=` can be used to multiply the vector with a scalar and to assign the result to it. The division of the vector by a scalar or divide and assign can be achieved with operators `operator /` and `operator /=` respectively.

Two vectors can added with operator `operator +` and the difference of them is computed with operator `operator -`. Moreover, operators which add and assign `operator +=` as well as subtract and assign `operator -=` are provided.

Though the cross product is not defined for two dimensions, its z component can be computed with operator `operator ^`. The angle in radians between two vectors is calculated with operator `operator %`.

The norm of the vector is given by method method `norm` and the vector can be normalized with method `unit` which returns a parallel vector of length 1.

Two static member constants representing the unit vectors `X` and `Y` are defined.

3.3 Vector3D

Three-dimensional vectors are represented by objects of class `Vector3D`. These vectors are used, e.g., to specify the position of spacecraft or celestial bodies, the magnetic field or the direction of incident light on a photo detector. Like class `Vector2D` the class is also derived from the more general template class `Vector<double>`. The formal definition of class `Vector3D` is shown in the listing below.

```
// Class Vector3D
class CubeSim::Vector3D : private Vector<double>
{
public:

   // Unit Vectors
   static const Vector3D X;
   static const Vector3D Y;
   static const Vector3D Z;

   // Constructor
   Vector3D(void);
   Vector3D(double x, double y, double z);
   Vector3D(const Vector2D& v);
   Vector3D(const Vector<double>& v);

   // Get Element
   double& operator ()(size_t i);
   double operator ()(size_t i) const;
   double& at(size_t i);
   const double& at(size_t i) const;

   // Sign
   const Vector3D& operator +(void) const;
   const Vector3D operator -(void) const;

   // Compare
   bool operator ==(const Vector3D& v) const;
   bool operator !=(const Vector3D& v) const;

   // Multiply
   const Vector3D operator *(double a) const;
   double operator *(const Vector3D& v) const;
   Vector3D& operator *=(double a);

   // Divide
   const Vector3D operator /(double a) const;
   Vector3D& operator /=(double a);

   // Add
   const Vector3D operator +(const Vector3D& v) const;
   Vector3D& operator +=(const Vector3D& v);

   // Subtract
   const Vector3D operator -(const Vector3D& v) const;
   Vector3D& operator -=(const Vector3D& v);

   // Compute Cross Product
   const Vector3D operator ^(const Vector3D& v) const;
```

```
    Vector3D& operator ^=(const Vector3D& v);

    // Compute Angle
    double operator %(const Vector3D& v) const;

    // Compute Norm
    using Vector<double>::norm;

    // Compute Unit Vector
    const Vector3D unit(void) const;

    // X Coordinate
    double x(void) const;
    void x(double x);

    // Y Coordinate
    double y(void) const;
    void y(double y);

    // Z Coordinate
    double z(void) const;
    void z(double z);
};
```

Since operators and methods being defined in class `Vector3D` are very similar to these defined in class `Vector2D`, only the differences are emphasized in this section. A null vector can be created with the default constructor `Vector3D(void)` which initializes all three coordinates, x, y and z, to zero. By the use of constructor `Vector3D(double x, double y, double z)` the new vector will be initialized to the specified coordinates. The copy constructor `Vector3D(const Vector<double>& v)` is defined allowing to duplicate an existing vector.

The z coordinate can be accessed with method `z` or operator `operator ()` with index 3.

The cross product between two vectors can be computed by the use of operator `operator ^`. In addition, operator `operator ^=` is provided which computes the cross product and assigns it to the vector.

Three static member constants representing the unit vectors `X`, `Y` and `Z` are defined.

3.4 Grid2D

A grid of points spanning the unit circle area can be implemented with class `Grid2D`. In this framework it is used for the disk model of module `Light` where, instead of a point light source, multiple emission sites are used for ray-tracing (see section 4.2). This allows the simulation of penumbra and eclipses.

The number of points (also denoted as resolution) can be set arbitrarily and the resultant list of points is optimized for a homogeneous distribution over the surface. A trade-off between required accuracy (e.g. of Earth albedo, see section 4.3) and simulation duration can be done by the user.

The definition of class `Grid2D` is shown in the listing below excluding private declarations.

```
// Class Grid2D
class CubeSim::Grid2D
{
public:
```

```
    // Constructor
    Grid2D(uint32_t resolution);

    // Resolution (Number of Points)
    uint32_t resolution(void) const;
    void resolution(uint32_t resolution);

    // Compute Points
    const std::vector<Vector2D> points(void) const;
};
```

A new grid can be created with the use of constructor `Grid2D(uint32_t resolution)` and the desired number of points (resolution) is passed as the only argument. Only grids with a resolution greater than or equal to 7 can be created.

The resolution can be retrieved by calling method `resolution` without arguments. The resolution of the grid can be modified by passing the new value to this method.

The list of points is generated by the use of method `points` which returns an STL vector [103, p. 482-488] of type `std::vector<Vector2D>`.

In fig. 3.1 two grids with different resolutions are shown as an example. While the grid in black is made up with 93 points, the red grid only features 19 points. Though the homogeneity increases with the number of points, one can already recognize the comparatively good distribution in both cases.

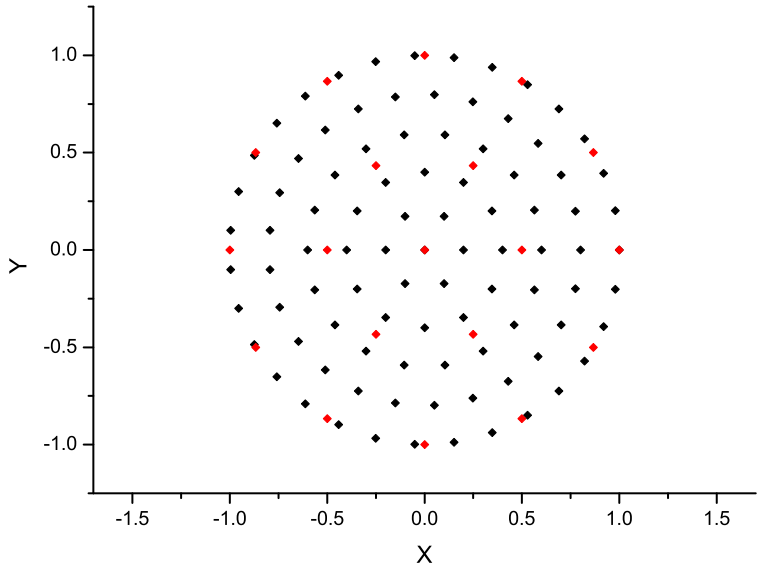

Fig. 3.1. Uniform distribution of 93 points (black) and 19 points (red) on the unit circle area.

3.5 Grid3D

Similar to a two-dimensional grid `Grid2D` spanning the unit circle area, a grid of points spanning the unit sphere area can be implemented with class `Grid3D`. In this framework it is used for module `Albedo` where the surface of a celestial body is subdivided into different regions which are then iterated to compute the resulting albedo that a nearby spacecraft is experiencing from diffuse starlight reflection (see section 4.3).

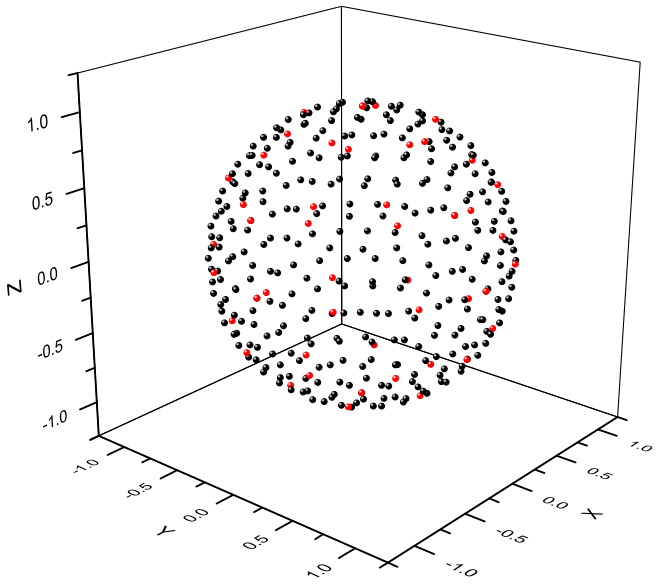

Fig. 3.2. Uniform distribution of 320 points (black) and 46 points (red) on the unit sphere area.

The number of points (also denoted as resolution) can be set arbitrarily and the resultant list of points is optimized for a homogeneous distribution over the spherical surface. Similar to `Grid2D`, a trade-off between required accuracy (e.g. of Earth albedo) and simulation duration can be done by the user.

The definition of class `Grid3D` is shown in the listing below excluding private declarations.

```
// Class Grid3D
class CubeSim::Grid3D
{
public:

    // Constructor
    Grid3D(uint32_t resolution);

    // Resolution (Number of Points)
    uint32_t resolution(void) const;
    void resolution(uint32_t resolution);

    // Compute Points
    const std::vector<Vector3D> points(void) const;
};
```

Since the methods being defined in class `Grid3D` are very similar to these defined in class `Grid2D`, only the differences are emphasized in this section. The allowed resolution is greater than or equal to 6. Method `points` is returning an STL vector of type `std::vector<Vector3D>` representing the points evenly distributed on the surface of the unit sphere.

In fig. 3.2 two grids with different resolutions are shown as an example. While the grid in black is made up with 320 points, the red grid only features 46 points. Though the homogeneity increases with the number of points, one can already recognize the comparatively good distribution in both cases.

3.6 Matrix3D

3×3 matrices are represented by objects of type `Matrix3D`. These matrices are used, e.g., to define the rotation between two reference frames or the moment of inertia of a rigid body. The class is derived from the more general template class `Matrix<double>` which allows the instantiation of $n \times m$ matrices with elements of arbitrary type. The formal definition of class `Matrix3D` is shown in the listing below.

```
// Class Matrix3D
class CubeSim::Matrix3D : private Matrix<double>
{
public:

   // Identity Matrix
   static const Matrix3D IDENTITY;

   // Constructor
   Matrix3D(void);
   Matrix3D(const Matrix<double>& A);
   Matrix3D(const Vector3D& v1, const Vector3D& v2, const Vector3D& v3);

   // Sign
   const Matrix3D& operator +(void) const;
   const Matrix3D operator -(void) const;

   // Compare
   bool operator ==(const Matrix3D& A) const;
   bool operator !=(const Matrix3D& A) const;

   // Multiply
   const Matrix3D operator *(double a) const;
   const Vector3D operator *(const Vector3D& v) const;
   const Matrix3D operator *(const Matrix3D& A) const;
   Matrix3D& operator *=(double a);
   Matrix3D& operator *=(const Matrix3D& A);

   // Divide
   const Matrix3D operator /(double a) const;
   Matrix3D& operator /=(double a);

   // Add
   const Matrix3D operator +(double a) const;
   const Matrix3D operator +(const Matrix3D& A) const;
   Matrix3D& operator +=(double a);
   Matrix3D& operator +=(const Matrix3D& A);

   // Subtract
   const Matrix3D operator -(double a) const;
   const Matrix3D operator -(const Matrix3D& A) const;
   Matrix3D& operator -=(double a);
   Matrix3D& operator -=(const Matrix3D& A);

   // Check if asymmetric
   using Matrix<double>::asymmetric;

   // Get Element
   double& operator ()(size_t row, size_t col);
```

```
    double operator ()(size_t row, size_t col) const;
    double& at(size_t row, size_t col);
    double at(size_t row, size_t col) const;

    // Compute Cofactor
    double cofactor(size_t row, size_t col) const;

    // Decompose in LU Shape
    void decompose_LU(Matrix3D& L, Matrix3D& U) const;

    // Decompose in LUP Shape
    double decompose_LUP(Matrix3D& L, Matrix3D& U, Matrix3D& P) const;

    // Compute Determinant
    using Matrix<double>::det;

    // Check if diagonal
    using Matrix<double>::diagonal;

    // Compute Inverse
    const Matrix3D inverse(void) const;

    // Check if orthogonal
    using Matrix<double>::orthogonal;

    // Check if regular
    using Matrix<double>::regular;

    // Check if singular
    using Matrix<double>::singular;

    // Solve Linear Equation Set
    const Vector3D solve(const Vector3D& v) const;

    // Check if symmetric
    using Matrix<double>::symmetric;

    // Swap Columns
    void swap_cols(size_t i, size_t j);

    // Swap Rows
    void swap_rows(size_t i, size_t j);

    // Compute Trace
    using Matrix<double>::trace;

    // Transpose
    const Matrix3D transpose(void) const;

    // Check if triangular
    using Matrix<double>::triangular;
};
```

A null matrix can be created with the default constructor `Matrix3D(void)` which initializes all nine elements to zero. By the use of constructor `Matrix3D(const Vector3D& v1, const Vector3D& v2, const Vector3D& v3)` the new matrix will be initialized to the specified three column vectors. The copy constructor `Matrix3D(const Matrix<double>& A)` is defined allowing to duplicate an existing matrix of type `Matrix3D` or to convert a general 3×3 matrix of type `Matrix<double>` to a matrix type `Matrix3D`.

The elements can be either accessed by `operator ()` or by method `at`. Similar to classes `Vector2D` and `Vector3D`, indices are 1-based and therefore range from 1 to 3. The first argument specifies the row, the second the column of the element.

The unary operator `operator -` negates all elements of the matrix and the unary operator `operator +` returns a reference to the matrix itself.

Two matrices can be compared by the use of operators `operator ==` and `operator !=`. In order to avoid erroneous results due to rounding errors, the comparison of two elements uses the function defined in 5.

A matrix being multiplied with a scalar, the product between two matrices or the multiplication of a matrix and a vector is computed with operator `operator *`. The operator `operator *=` can be used to multiply the matrix with a scalar or to multiply the matrix with another matrix and to assign the result to it. The division of the matrix by a scalar or divide and assign can be achieved with operators `operator /` and `operator /=` respectively.

Two matrices can be added with operator `operator +` and the difference of them is computed with operator `operator -`. Moreover, operators which add and assign `operator +=` as well as subtract and assign `operator -=` are provided.

In order to test if the matrix is symmetric or asymmetric, the methods `symmetric` and `asymmetric` can be used. The methods `regular` and `singular` are provided to check whether the matrix is regular or singular respectively. To test where the matrix is diagonal or triangular, methods `diagonal` or `triangular` can to be called.

The co-factor of the matrix can be computed with the aid of method `cofactor` expecting the row and column. This method is also internally used by methods `det` and `inverse` to compute the determinant and the inverse of the matrix.

With method `decompose_LUP` any regular matrix $\boldsymbol{A}$ can be decomposed into $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{U}\boldsymbol{P}$ with the lower triangular matrix $\boldsymbol{L}$, the upper triangular matrix $\boldsymbol{U}$ and the permutation matrix $\boldsymbol{P}$ [108, p. 2]. There are regular matrices $\boldsymbol{A}$ that can be decomposed into $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{U}$ which can be done with method `decompose_LU`. Decomposition of matrices is an efficient way to compute their inverse or determinant and to solve linear equation systems [187, p. 54f]. The latter is accomplished by the use of method `solve`.

Methods `swap_columns` and `swap_rows` swap two columns or two rows of the given matrix. Method `transpose` swaps columns with rows so that the transposed matrix $\boldsymbol{A}^T_{ij} = \boldsymbol{A}_{ji}$.

The identity matrix $\mathbb{1} = \delta_{ij}$ is predefined by the static member constant `IDENTITY` and the Kronecker delta δ_{ij} is defined as

$$\delta_{ij} = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases} \tag{6}$$

3.7 Rotation

Any rigid body, such as a celestial body or part of an assembly, can not only be positioned in the global of relative coordinate system,but can also be rotated relative to it. There are multiple ways to define the rotation with corresponding advantages and disadvantages. In the following sections the rotation matrix, Euler axis and angle representation, Euler angles and quaternions are presented and efficient conversion algorithms are explained.

The class `Rotation` is introduced and its formal definition is shown in the listing below excluding certain private declarations.

```
// Class Rotation
class CubeSim::Rotation
{
public:

   // Constructor
   Rotation(double yaw, double pitch, double roll);
   Rotation(const Vector3D& axis, double angle);
   Rotation(const Vector3D& b1, const Vector3D& b2, const Vector3D& b3);
   Rotation(const Matrix3D& matrix = Matrix3D::IDENTITY);

   // Get Matrix
   operator const Matrix3D&(void) const;

   // Sign
   const Rotation& operator +(void) const;
   const Rotation operator -(void) const;

   // Compare
   bool operator ==(const Rotation& rotation) const;
   bool operator !=(const Rotation& rotation) const;

   // Rotate
   const Rotation operator +(const Rotation& rotation) const;
   const Rotation operator -(const Rotation& rotation) const;
   Rotation& operator +=(const Rotation& rotation);
   Rotation& operator -=(const Rotation& rotation);

   // Get Euler Angle [rad]
   double angle(void) const;

   // Get Euler Axis
   const Vector3D& axis(void) const;

   // Get Matrix
   const Matrix3D& matrix(void) const;

   // Get Pitch Angle [rad]
   double pitch(void) const;

   // Get Roll Angle [rad]
   double roll(void) const;

   // Get Yaw Angle [rad]
   double yaw(void) const;

private:

   // Compute Pitch, Roll, Yaw Angles [rad]
   void _angles(void) const;

   // Compute Euler Angle [rad] and Axis
   void _euler(void) const;
};
```

Internally, the definition of a rotation is stored as a rotation matrix of type `Matrix3D` which allows fast combination of rotations by means of matrix multiplication. Nevertheless, the class accepts rotations specified by a rotation vector consisting of an axis and the angle (see section 3.7.2), the three Euler angles

roll, pitch and *yaw* (see section 3.7.3) or by the definition of the base vectors which actually represent the rotation matrix itself (see section 3.7.1). If the Euler angles are requested by calling method `pitch`, `roll`, `yaw` or the rotational vector `axis` or angle `angle`, the properties are computed (private methods `_angles` and `_euler`) and cached for later use. If the rotational matrix is changed, the cached values are invalidated and recomputed upon request.

An implicit cast operator is defined for retrieving the rotation matrix of type `Matrix3D`.

The unary operator `operator -` negates the angle, invalidates the Euler angles and transposes the rotational matrix. The unary operator `operator +` returns a reference to the rotation itself.

Two rotations can be compared by the use of operators `operator ==` and `operator !=` which return the result of the corresponding rotation matrix comparison.

Two rotations can be added (combined) with operator `operator +` and the difference of them is computed with operator `operator -`. Moreover, operators which add and assign `operator +=` as well as subtract and assign `operator -=` are provided.

3.7.1 Rotation Matrix

The coordinate frame $\mathcal{O}$ is considered with its basis vectors $\boldsymbol{e}_x$, $\boldsymbol{e}_y$ and $\boldsymbol{e}_z$. A rigid body with body-fixed coordinate frame $\mathcal{O}'$ and basis vectors $\boldsymbol{e}'_x$, $\boldsymbol{e}'_y$ and $\boldsymbol{e}'_z$ is rotated with respect to $\mathcal{O}$. Therefore, the basis vectors of $\mathcal{O}'$ can be represented in $\mathcal{O}$ such as $\boldsymbol{e}'_i = \sum a_{ji}\boldsymbol{e}_j$ or $\boldsymbol{e}'_i = \boldsymbol{R}\,\boldsymbol{e}_i$ with the rotation matrix being defined as

$$\boldsymbol{R} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}. \tag{7}$$

Let us assume a vector $\boldsymbol{x}$ in the global coordinate frame $\mathcal{O}$ and a rigid body whose body-fixed coordinate system $\mathcal{O}'$ rotation is represented by the rotation matrix $\boldsymbol{R}$. The vector $\boldsymbol{x}'$ referenced to $\mathcal{O}'$ can be expressed as $\boldsymbol{x}' = \boldsymbol{R}^{\mathrm{T}}\boldsymbol{x}$ or $\boldsymbol{x} = \boldsymbol{R}\,\boldsymbol{x}'$.

The rotation matrix can also be thought as the list of basis vectors defining the body-fixed coordinate system $\mathcal{O}'$. The columns represent e'_x, e'_y and e'_z in $\mathcal{O}$. $\boldsymbol{R}$ belongs to the special orthogonal group of all 3×3 denoted by $SO(3)$ and has the properties [55, p. 4]

$$\det\boldsymbol{R} = \pm 1,\ \boldsymbol{R}^{-1} = \boldsymbol{R}^{\mathrm{T}}. \tag{8}$$

Rotations can also be combined yielding in another rotation matrix by matrix multiplication which is efficient for numerical computations as no trigonometric functions are involved:

$$\boldsymbol{x} = \boldsymbol{R}_2\left(\boldsymbol{R}_1\boldsymbol{x}'\right) = \boldsymbol{R}\,\boldsymbol{x}' \text{ with } \boldsymbol{R} = \boldsymbol{R}_1\boldsymbol{R}_2. \tag{9}$$

Though rotation matrices are straightforward to use and show good computation performance - since only scalar multiplications and additions are used for matrix multiplication - the nine elements over-determine the represented rotation. In essence, four elements fully determine a rotation in $\mathbb{R}_3$ and the additional five elements might distort the length of the basis vectors or the angle between them which can result from numerical rounding errors. This can lead to the fact that, after a large number of matrix multiplications, the initial conditions (8) can no longer be fulfilled. To overcome this weakness, re-normalization techniques can periodically be applied, which correct the matrix elements so that condition (8) is maintained.

3.7.2 Rotation Vector

For building an assembly consisting of multiple parts with different positions and orientations, a more convenient way for specifying the rotation is provided. According to Euler's theorem, every rotation or compound rotation can be represented with an axis of rotation $\boldsymbol{e}$ ($|\boldsymbol{e}| = 1$) and an angle θ [180]. Rodrigues' rotation formula [198], named after O. Rodrigues, can be used to rotate any vector $\boldsymbol{x}$ in $\mathbb{R}_3$ by angle θ around the axis of rotation $\boldsymbol{e}$ according to the right hand rule:

$$\boldsymbol{x}' = \boldsymbol{x}\cos\theta + (\boldsymbol{e}\times\boldsymbol{x})\sin\theta + \boldsymbol{e}\,(\boldsymbol{e}\,\boldsymbol{x})\,(1-\cos\theta)\,. \tag{10}$$

This formula can be used to compute the rotation matrix from the axis of rotation $\boldsymbol{e}$ and the angle θ. For that, the cross-product matrix $\boldsymbol{E}$ fulfilling $\boldsymbol{e}\times\boldsymbol{x} = \boldsymbol{E}\,\boldsymbol{x}$ is required:

$$\boldsymbol{E} = \begin{pmatrix} 0 & -\boldsymbol{e}_\mathrm{x} & \boldsymbol{e}_\mathrm{y} \\ \boldsymbol{e}_\mathrm{z} & 0 & -\boldsymbol{e}_\mathrm{x} \\ -\boldsymbol{e}_\mathrm{y} & \boldsymbol{e}_\mathrm{x} & 0 \end{pmatrix}. \tag{11}$$

The rotation matrix $\boldsymbol{R}$ with $\boldsymbol{x}' = \boldsymbol{R}\,\boldsymbol{x}$ can be expressed as

$$\boldsymbol{R} = \mathbb{1} + \boldsymbol{E}\,\boldsymbol{x}\sin\theta + \boldsymbol{E}^2\,(1-\cos\theta) \tag{12}$$

with the identity matrix $\mathbb{1}$. Expanding this expression leads to the element-wise representation of the rotation matrix

$$\boldsymbol{R} = \begin{pmatrix} {\boldsymbol{e}_\mathrm{x}}^2(1-\cos\theta)+\cos\theta & \boldsymbol{e}_\mathrm{x}\boldsymbol{e}_\mathrm{y}(1-\cos\theta)-\boldsymbol{e}_\mathrm{z}\sin\theta & \boldsymbol{e}_\mathrm{x}\boldsymbol{e}_\mathrm{z}(1-\cos\theta)+\boldsymbol{e}_\mathrm{y}\sin\theta \\ \boldsymbol{e}_\mathrm{x}\boldsymbol{e}_\mathrm{y}(1-\cos\theta)+\boldsymbol{e}_\mathrm{z}\sin\theta & {\boldsymbol{e}_\mathrm{y}}^2(1-\cos\theta)+\cos\theta & \boldsymbol{e}_\mathrm{y}\boldsymbol{e}_\mathrm{z}(1-\cos\theta)-\boldsymbol{e}_\mathrm{x}\sin\theta \\ \boldsymbol{e}_\mathrm{x}\boldsymbol{e}_\mathrm{z}(1-\cos\theta)-\boldsymbol{e}_\mathrm{y}\sin\theta & \boldsymbol{e}_\mathrm{y}\boldsymbol{e}_\mathrm{z}(1-\cos\theta)+\boldsymbol{e}_\mathrm{x}\sin\theta & {\boldsymbol{e}_\mathrm{z}}^2(1-\cos\theta)+\cos\theta \end{pmatrix}. \tag{13}$$

Vice versa, the axis of rotation and the angle can be obtained from the rotation matrix. In the first step the cosine of the angle and the angle are computed:

$$\delta = \cos\theta = \frac{1}{2}\left(\boldsymbol{R}_{11}+\boldsymbol{R}_{22}+\boldsymbol{R}_{33}-1\right),\ \theta = \arccos\delta. \tag{14}$$

For $-1 < \delta < 1$ the axis results in

$$\boldsymbol{e} = \frac{1}{2\sin\theta}\begin{pmatrix} \boldsymbol{R}_{32}-\boldsymbol{R}_{23} \\ \boldsymbol{R}_{13}-\boldsymbol{R}_{31} \\ \boldsymbol{R}_{21}-\boldsymbol{R}_{12} \end{pmatrix}. \tag{15}$$

For $\delta = 1$ and consequently $\theta = 0$, the rotation matrix $\boldsymbol{R} = \mathbb{1}$ and the axis of rotation is undefined or can arbitrarily be chosen. For $\delta = -1$ and consequently $\theta = \pi$, the rotation matrix is simplified to

$$\boldsymbol{R} = \begin{pmatrix} 2{\boldsymbol{e}_\mathrm{x}}^2-1 & 2\boldsymbol{e}_\mathrm{x}\boldsymbol{e}_\mathrm{y} & 2\boldsymbol{e}_\mathrm{x}\boldsymbol{e}_\mathrm{z} \\ 2\boldsymbol{e}_\mathrm{x}\boldsymbol{e}_\mathrm{y} & 2{\boldsymbol{e}_\mathrm{y}}^2-1 & 2\boldsymbol{e}_\mathrm{y}\boldsymbol{e}_\mathrm{z} \\ 2\boldsymbol{e}_\mathrm{x}\boldsymbol{e}_\mathrm{z} & 2\boldsymbol{e}_\mathrm{y}\boldsymbol{e}_\mathrm{z} & 2{\boldsymbol{e}_\mathrm{z}}^2-1 \end{pmatrix}. \tag{16}$$

The square root of the diagonal elements would not lead to unambiguous results and the columns 1, 2, 3 cannot simply be transformed and divided by $\boldsymbol{e}_\mathrm{x}$, $\boldsymbol{e}_\mathrm{y}$, $\boldsymbol{e}_\mathrm{z}$ in order to avoid division by zero. Therefore, the largest diagonal element is determined first and corresponding column is used to compute the axis of rotation:

$$\boldsymbol{R}_{nn} = \max \boldsymbol{R}_{ii},\ \boldsymbol{e} = \frac{1}{\sqrt{2\,(\boldsymbol{R}_{nn}+1)}}\,\mathrm{col}_\mathrm{n}\,(\boldsymbol{R}+\mathbb{1})\,. \tag{17}$$

3.7.3 Euler Angles

Another way to represent an arbitrary rotation in $\mathbb{R}_3$ is the use of three angles, the so-called *Euler angles* [55, p. 7ff]. It is convenient to imagine an aircraft which is able to perform the following three independent rotations: *roll*, around the front-to-back axis; *pitch*, tilting the nose up or down around the side-to-side axis; *yaw*, rotating around the vertical axis [111, p. 2]. There are multiple conventions for Euler angles depending on the axes of rotation and the sequence - here the *z-y-x* Euler angles are used. The body frame starts with the same orientation as the global frame. The first rotation is defined by angle ϕ around the z-axis (yaw), the second rotation by angle $\theta \in [0, \pi]$ around the y-axis (pitch), and the third rotation by angle ψ around the x-axis (roll). The individual rotations are described by the following rotation matrices:

$$\boldsymbol{R}_z(\phi) = \begin{pmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{pmatrix}, \; \boldsymbol{R}_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix}, \; \boldsymbol{R}_x(\psi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\psi & -\sin\psi \\ 0 & \sin\psi & \cos\psi \end{pmatrix}. \tag{18}$$

The corresponding rotation matrix is composed of these matrices:

$$\boldsymbol{R} = \boldsymbol{R}_z(\phi)\boldsymbol{R}_y(\theta)\boldsymbol{R}_x(\psi) = \begin{pmatrix} \cos\phi\cos\theta & \cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi & \cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi \\ \sin\phi\cos\theta & \sin\phi\sin\theta\sin\psi + \cos\phi\cos\psi & \sin\phi\sin\theta\cos\psi - \cos\phi\sin\psi \\ -\sin\theta & \cos\theta\sin\psi & \cos\theta\cos\psi \end{pmatrix}. \tag{19}$$

Vice versa, from the rotation matrix, the Euler angles can be computed. In the first step, the cosine of the pitch angle is calculated as follows:

$$\delta = \cos\theta = \sqrt{\boldsymbol{R}_{11}^2 + \boldsymbol{R}_{21}^2}. \tag{20}$$

Then the pitch angle can be obtained:

$$\theta = \operatorname{arctan2}\left(-\boldsymbol{R}_{31},\ \delta\right). \tag{21}$$

The function arctan2 represents the two-argument *arcus tangens* function in all four quadrants and is defined as [205, p. 2]

$$\operatorname{arctan2}(y,\ x) = \begin{cases} \arctan\frac{y}{x} & x > 0 \\ \arctan\frac{y}{x} + \pi & x < 0,\ y \geqslant 0 \\ \arctan\frac{y}{x} - \pi & x < 0,\ y < 0 \\ +\frac{\pi}{2} & x = 0,\ y > 0 \\ -\frac{\pi}{2} & x = 0,\ y < 0 \end{cases}. \tag{22}$$

The result for $\operatorname{arctan2}(0, 0)$ is either 0 or not defined, depending on the implementation.

For the computation of yaw and roll angles, singularity checks need to be made

$$\phi = \begin{cases} \operatorname{arctan2}\left(-\boldsymbol{R}_{21},\ \boldsymbol{R}_{11}\right) & \delta \neq 0 \\ 0 & \delta = 0 \end{cases}, \quad \psi = \begin{cases} \operatorname{arctan2}\left(\boldsymbol{R}_{32},\ \boldsymbol{R}_{33}\right) & \delta \neq 0 \\ \operatorname{arctan2}\left(-\boldsymbol{R}_{23},\ \boldsymbol{R}_{22}\right) & \delta = 0 \end{cases}. \tag{23}$$

3.7.4 Quaternions

A complex number can be considered as a two-dimensional vector (x, y) or $\boldsymbol{z} = x + \boldsymbol{i}y$. The multiplication is then the rotation of the vector for which Euler's formula is used [23, p. 14]

$$e^{\boldsymbol{i}\theta} = \cos\theta + \boldsymbol{i}\sin\theta. \tag{24}$$

$$\boldsymbol{z}' = e^{\boldsymbol{i}\theta}\boldsymbol{z} = e^{\boldsymbol{i}\theta}(x + \boldsymbol{i}y) = x\cos\theta - y\sin\theta + \boldsymbol{i}(x\sin\theta + y\cos\theta) \tag{25}$$

which corresponds to to the rotation of (x, y) by angle θ

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}. \tag{26}$$

Quaternions were invented by W. R. Hamilton in the 19$^{\text{th}}$ century. He wanted to generalize complex numbers to three dimensions, i.e., numbers in the form $a + \mathbf{i}b + \mathbf{j}c$ where a, b, $c \in \mathbb{R}$ and $\mathbf{i}^2 = \mathbf{j}^2 = -1$. However, in 1966 K. O. May proved that Hamilton failed and that the set of three-dimensional numbers is not closed under multiplication [47, p. 7].

One of Hamilton's aims was the representation of rotations in three-dimensional space. As two-dimensional complex numbers, multiplication should correspond to a rotation and a scaling. He realized that four numbers are required to describe a rotation followed by scaling: one number describes the scaling, one the rotation angle and two numbers the plane in which the vector is to be rotated. Here the plane is the initial xy plane rotated by certain angles around x and y axes. Hamilton found a closed multiplication for four-dimensional numbers in the form $\mathbf{i}a + \mathbf{j}b + \mathbf{k}c$ with $\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \boldsymbol{ijk} = -1$.

A quaternion $\boldsymbol{q} \in \mathbb{H}$ may be represented as a vector,

$$\boldsymbol{q} = \begin{pmatrix} q_0 & q_1 & q_2 & q_3 \end{pmatrix}^{\mathrm{T}} = \begin{pmatrix} q_0 \\ \boldsymbol{q}_{1:3} \end{pmatrix} = q_0 + \boldsymbol{i}q_1 + \boldsymbol{j}q_2 + \boldsymbol{k}q_3. \tag{27}$$

Further, the adjoint (conjugate quaternion), norm, and inverse are defined as [55, p. 14]

$$\overline{\boldsymbol{q}} = \begin{pmatrix} q_0 \\ -\boldsymbol{q}_{1:3} \end{pmatrix}, \tag{28}$$

$$\|\boldsymbol{q}\| = \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2}, \tag{29}$$

$$\boldsymbol{q}^{-1} = \frac{\overline{\boldsymbol{q}}}{\|\boldsymbol{q}\|}. \tag{30}$$

Unit quaternions are quaternions with unit length $\|\boldsymbol{q}\| = 1$ and can be used to describe the orientation of a rigid body $\boldsymbol{q}_{\mathrm{R}}$. Vectors in three-dimensional space $\boldsymbol{q}_{\mathrm{x}}$ can be expressed with pure quaternions which have no real part $q_0 = 0$. A coordination transformation $\boldsymbol{q}_{\mathbf{x}} \rightarrow \boldsymbol{q}'_{\mathrm{x}}$ is given by the conjugate operation [23, p. 16 and p. 20]

$$\boldsymbol{q}_{\mathbf{x}}{}' = \boldsymbol{q}_{\mathbf{R}}\, \boldsymbol{q}_{\mathbf{x}}\, \overline{\boldsymbol{q}}_{\mathbf{R}}. \tag{31}$$

This corresponds to the rotation matrix

$$\boldsymbol{R} = 2\begin{pmatrix} q_0^2 + q_1^2 - \frac{1}{2} & q_1q_2 - q_0q_3 & q_0q_2 + q_1q_3 \\ q_0q_3 + q_1q_2 & q_0^2 + q_2^2 - \frac{1}{2} & q_2q_3 - q_0q_1 \\ q_1q_3 - q_0q_2 & q_0q_1 + q_2q_3 & q_0^2 + q_3^2 - \frac{1}{2} \end{pmatrix}. \tag{32}$$

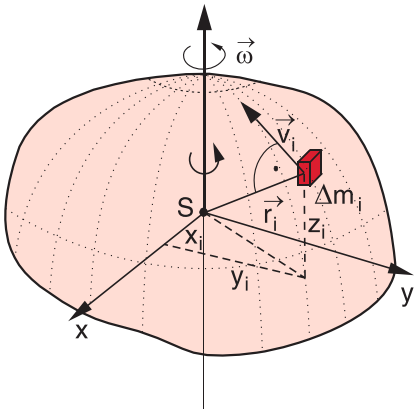

Fig. 3.3. Rotation of an arbitrary rigid body around an axis through the center of mass [52, p. 139].

In order to compute the quaternion from the rotation matrix, the following equation is used [23, p. 18]:

$$tr(\boldsymbol{R}) = \boldsymbol{R}_{11} + \boldsymbol{R}_{22} + \boldsymbol{R}_{33} = 4q_0^2 - 1 \tag{33}$$

This allows to solve for q_0

$$|q_0| = \frac{1}{2}\sqrt{tr(\boldsymbol{R}) + 1} \tag{34}$$

and the remaining components are computed as follows [70, p. 2]:

$$q_1 = \frac{\boldsymbol{R}_{32} - \boldsymbol{R}_{23}}{4q_0},\ q_2 = \frac{\boldsymbol{R}_{13} - \boldsymbol{R}_{31}}{4q_0},\ q_3 = \frac{\boldsymbol{R}_{21} - \boldsymbol{R}_{12}}{4q_0}. \tag{35}$$

3.8 Inertia

When a point mass m is moving at the constant velocity v, it has the kinetic energy $E_{\text{kin}} = \frac{1}{2}m\,v^2$. If the point mass is rotating around a fixed axis at the distance r, the tangential velocity is $v = \omega r$ and the kinetic energy can be expressed as

$$E_{\text{rot}} = \frac{1}{2}\omega^2 m\,r^2 = \frac{1}{2}\omega^2 I \tag{36}$$

with the introduced moment of inertia $I = m\,r^2$. Similar to the linear momentum of a moving point mass $p = m\,v$, the angular momentum can be expressed as [52, p. 132]

$$L = I\omega. \tag{37}$$

Now an arbitrary body is considered with the center of mass $\boldsymbol{S}$ and the angular momentum $\boldsymbol{\omega}$ (see fig. 3.3). The angular momentum of a mass element m_i with the tangential velocity $\boldsymbol{v}_i = \boldsymbol{\omega} \times \boldsymbol{r}_i$ results in [52, p. 138]

$$\boldsymbol{L}_i = m_i\,(\boldsymbol{r}_i \times \boldsymbol{v}_i) = m_i\,(\boldsymbol{r}_i \times (\boldsymbol{\omega} \times \boldsymbol{r}_i)) = m_i\left(\boldsymbol{r}_i^2\boldsymbol{\omega} - (\boldsymbol{r}_i\boldsymbol{\omega})\,\boldsymbol{r}_i\right). \tag{38}$$

The total angular momentum is the sum

$$\boldsymbol{L} = \int \left(\boldsymbol{r}^2\boldsymbol{\omega} - (\boldsymbol{r}\boldsymbol{\omega})\,\boldsymbol{r}\right) \mathrm{d}m. \tag{39}$$

This vector equation corresponds to the following three equations

$$\begin{aligned} L_x &= I_{xx}\omega_x + I_{xy}\omega_y + I_{xz}\omega_z, \\ L_y &= I_{yx}\omega_x + I_{yy}\omega_y + I_{yz}\omega_z, \\ L_z &= I_{zx}\omega_x + I_{zy}\omega_y + I_{zz}\omega_z. \end{aligned} \tag{40}$$

with I_{ij} defined as follows:

$$\begin{aligned} I_{xx} &= \int \left(y^2 + z^2\right) \mathrm{d}m, \\ I_{xy} = I_{yx} &= -\int x\, y\, \mathrm{d}m, \\ I_{yy} &= \int \left(x^2 + z^2\right) \mathrm{d}m, \\ I_{yz} = I_{zy} &= -\int y\, z\, \mathrm{d}m, \\ I_{zz} &= \int \left(x^2 + y^2\right) \mathrm{d}m, \\ I_{xz} = I_{zx} &= -\int x\, z\, \mathrm{d}m. \end{aligned} \tag{41}$$

The elements I_{ij} build the inertia matrix $\boldsymbol{I}$ and the angular momentum can be expressed as

$$\boldsymbol{L} = \boldsymbol{I}\,\boldsymbol{\omega}. \tag{42}$$

It should be noted that $\boldsymbol{L}$ and $\boldsymbol{\omega}$ are in general not parallel.

In order to compute the rotation of a rigid body, the Euler equations are numerically iterated. This requires the introduction of class `Inertia` that holds the inertial properties of the body which are the mass, the center of mass and the inertia matrix. The formal definition of class `Inertia` is shown in the listing below excluding private declarations.

```
// Class Inertia
class CubeSim::Inertia
{
public:

   // Constructor
   Inertia(void);
   Inertia(const Matrix3D& inertia, double mass, const Vector3D& center
      = Vector3D());

   // Compute Inertia around arbitrary Axis and Pivot [kg*m^2]
   double operator ()(const Vector3D& axis, const Vector3D& pivot =
      Vector3D()) const;

   // Compute Moment of Inertia Matrix
   operator const Matrix3D(void) const;

   // Add
   const Inertia operator +(const Inertia& inertia) const;
   Inertia& operator +=(const Inertia& inertia);

   // Translate [m]
   const Inertia operator +(const Vector3D& distance) const;
```

```
    const Inertia operator -(const Vector3D& distance) const;
    Inertia& operator +=(const Vector3D& distance);
    Inertia& operator -=(const Vector3D& distance);

    // Rotate
    const Inertia operator +(const Rotation& rotation) const;
    const Inertia operator -(const Rotation& rotation) const;
    Inertia& operator +=(const Rotation& rotation);
    Inertia& operator -=(const Rotation& rotation);

    // Center [m]
    const Vector3D& center(void) const;
    void center(const Vector3D& center);

    // Mass [kg]
    double mass(void) const;
    void mass(double mass);
};
```

An object is created by passing the inertia matrix of a body around its center of mass, its mass and the center of mass. The class also supports the default constructor.

Operator `operator ()` allows to compute the inertia of the body around an arbitrary axis defined by a point and the direction.

The operator `operator const Matrix3D` allows to calculate the inertia matrix around the origin $\boldsymbol{O}$ of the coordinate system which is used for combined bodies, such as assemblies. The parallel axis theorem is used to compute the inertia matrix $\hat{\boldsymbol{I}}$ around the origin $\boldsymbol{O}$ from the inertia matrix $\boldsymbol{I}$ around the center of mass $\boldsymbol{C} = (x \;\; y \;\; z)^{\mathrm{T}}$ [52, p. 132]

$$\boldsymbol{I}_{\boldsymbol{O}}(\boldsymbol{I}, m, \boldsymbol{C}) = \boldsymbol{I} + m \begin{pmatrix} y^2 + z^2 & -x\,y & -x\,z \\ -x\,y & x^2 + z^2 & -y\,z \\ -x\,z & -y\,z & x^2 + y^2 \end{pmatrix}. \tag{43}$$

The aforementioned combination of two or more bodies can be achieved by the use of operator `operator +` or `operator +=`. The operator computes the total mass and the common center of mass and uses eq. (43) to translate and combine the individual inertia matrices. Considering two bodies with masses m_1 and m_2, the centers of mass $\boldsymbol{C}_1$ and $\boldsymbol{C}_2$ and the inertia matrices $\boldsymbol{I}_1$ and $\boldsymbol{I}_2$, the common mass $m = m_1 + m_2$ and the common center of mass $\boldsymbol{C} = (m_1\,\boldsymbol{C}_1 + m_2\,\boldsymbol{C}_2)/m$ are calculated first. This allows to compute the common inertia matrix around $\boldsymbol{C}$

$$\boldsymbol{I} = \boldsymbol{I}_{\boldsymbol{O}}(\boldsymbol{I}_1, m_1, \boldsymbol{C}_1 - \boldsymbol{C}) + \boldsymbol{I}_{\boldsymbol{O}}(\boldsymbol{I}_2, m_2, \boldsymbol{C}_2 - \boldsymbol{C}). \tag{44}$$

It can be shown that the transformation of $\boldsymbol{I}$ around the origin $\boldsymbol{O}$ equals to the sum of the individual inertia matrices $\boldsymbol{I}_1$ and $\boldsymbol{I}_2$ around the origin $\boldsymbol{O}$

$$\boldsymbol{I}_{\boldsymbol{O}}(\boldsymbol{I}_1, m_1, \boldsymbol{C}_1) + \boldsymbol{I}_{\boldsymbol{O}}(\boldsymbol{I}_2, m_2, \boldsymbol{C}_2) \equiv \boldsymbol{I}_{\boldsymbol{O}}\left(\boldsymbol{I}_{\boldsymbol{O}}(\boldsymbol{I}_1, m_1, \boldsymbol{C}_1 - \boldsymbol{C}) + \boldsymbol{I}_{\boldsymbol{O}}(\boldsymbol{I}_2, m_2, \boldsymbol{C}_2 - \boldsymbol{C}), m, \boldsymbol{C}\right). \tag{45}$$

In order to move the center of mass of the body within the local coordinate system, operators `operator +(const Vector3D& distance)`, `operator -(const Vector3D& distance)`, `operator +=(const Vector3D& distance)` and `operator -=(const Vector3D& distance)` can be used.

By using operators `operator +(const Rotation& rotation)`, `operator -(const Rotation& rotation)`, `operator +=(const Rotation& rotation)` and `operator -=(const Rotation&`

rotation) the rotation of the rigid body in the local frame is affected. These methods therefore also affect the center of mass which normally does not coincide with the origin.

The center of mass can be obtained with method center when no arguments are passed and can be updated by passing the new position.

Similarly, the mass of the body is returned by calling method mass with no parameters and can be modified by passing the new mass as an argument.

3.9 Force

During the execution of the simulation various forces and torques can be applied to the defined rigid bodies. From the physical point of view, a force is defined by a vector and the point of application and, therefore, a single vector of type Vector3D cannot be utilized to completely define any acting force.

The class is Force is introduced which derives from class Vector3D and extends it by the point of application property. The formal definition of class Force is shown in the listing below excluding certain private declarations.

```
// Class Force
class CubeSim::Force : public Vector3D, public List<Force>::Item
{
public:

   // Constructor
   Force(double x, double y, double z);
   Force(double x, double y, double z, double point_x, double point_y,
      double point_z);
   Force(const Vector3D& force = Vector3D(), const Vector3D& point =
      Vector3D());

   // Copy Constructor (rigid Body Reference is reset)
   Force(const Force& force);

   // Assign (rigid Body Reference is maintained)
   Force& operator =(const Force& force);

   // Sign
   const Force& operator +(void) const;
   const Force operator -(void) const;

   // Scale
   const Force operator *(double factor) const;
   const Force operator /(double factor) const;
   Force& operator *=(double factor);
   Force& operator /=(double factor);

   // Add (Point of Application of passed Force is discarded)
   const Force operator +(const Force& force) const;
   Force& operator +=(const Force& force);

   // Subtract (Point of Application of passed Force is discarded)
   const Force operator -(const Force& force) const;
   Force& operator -=(const Force& force);

   // Translate [m]
```

```
    const Force operator +(const Vector3D& distance) const;
    const Force operator -(const Vector3D& distance) const;
    Force& operator +=(const Vector3D& distance);
    Force& operator -=(const Vector3D& distance);

    // Rotate
    const Force operator +(const Rotation& rotation) const;
    const Force operator -(const Rotation& rotation) const;
    Force& operator +=(const Rotation& rotation);
    Force& operator -=(const Rotation& rotation);

    // Point of Application [m]
    const Vector3D& point(void) const;
    void point(const Vector3D& point);

    // X Coordinate [m]
    void x(double x);
    using Vector3D::x;

    // Y Coordinate [m]
    void y(double y);
    using Vector3D::y;

    // Z Coordinate [m]
    void z(double z);
    using Vector3D::z;

private:

    // Update Properties
    static const uint8_t _UPDATE_MAGNITUDE = 1;
    static const uint8_t _UPDATE_POINT = 2;

    // Update Property
    void _update(uint8_t update);

    // Variables
    RigidBody* _rigid_body;
};
```

The attentive reader will already have noticed the introduction of units. Throughout the entire framework solely SI units are being used. In class `Force` the force is therefore specified in Newtons and the coordinates of the point of application are given in meters.

Since class `Force` is publicly derived from `Vector3D`, most operators and methods can be used as explained in section 3.3. The point of application can be accessed via method `point` and can be moved by operators `operator +(const Vector3D& distance)`, `operator -(const Vector3D& distance)`, `operator +=(const Vector3D& distance)` and `operator -=(const Vector3D& distance)`.

In order to add or subtract two forces, the operators `operator +(const Force& force)` and `operator +=(const Force& force)` or `operator -(const Force& force)` and `operator -=(const Force& force)` can be utilized. Hereby the point of application of the second force is discarded.

If the force is rotated, both, the direction and the point of application are being rotated. For that purpose, operators `operator +(const Rotation& rotation)`, `operator -(const Rotation& rotation)`, `operator +=(const Rotation& rotation)` and `operator -=(const Rotation& rotation)` are provided.

3.10 Torque

One or more torques can be applied to rigid bodies. Like forces, torques can me defined for fixed rigid bodies like parts or sub-assemblies. The resultant torque on the free rigid body, like the entire spacecraft, is consequently computed automatically. Though a torque is fully specified by a vector, the separate class `Torque` is introduced for the purpose of distinctness. It is derived from class `Vector3D` and its formal definition is shown in the listing below excluding certain private declarations.

```
// Class Torque
class CubeSim::Torque : public Vector3D, public List<Torque>::Item
{
public:

   // Constructor
   Torque(double x, double y, double z);
   Torque(const Vector3D& torque = Vector3D());

   // Copy Constructor (rigid Body Reference is reset)
   Torque(const Torque& torque);

   // Assign (rigid Body Reference is maintained)
   Torque& operator =(const Torque& torque);
   Torque& operator =(const Vector3D& torque);

   // Sign
   const Torque& operator +(void) const;
   const Torque operator -(void) const;

   // Scale
   const Torque operator *(double factor) const;
   Torque operator /(double factor) const;
   Torque& operator *=(double factor);
   Torque& operator /=(double factor);

   // Add
   const Torque operator +(const Torque& torque) const;
   Torque& operator +=(const Torque& torque);
   using Vector3D::operator +;
   using Vector3D::operator +=;

   // Rotate
   const Torque operator +(const Rotation& rotation) const;
   const Torque operator -(const Rotation& rotation) const;
   Torque& operator +=(const Rotation& rotation);
   Torque& operator -=(const Rotation& rotation);
   using Vector3D::operator -;
   using Vector3D::operator -=;

   // X Coordinate [N*m]
   void x(double x);
   using Vector3D::x;

   // Y Coordinate [N*m]
   void y(double y);
   using Vector3D::y;
```

```
    // Z Coordinate [N*m]
    void z(double z);
    using Vector3D::z;

private:

    // Update Properties
    static const uint8_t _UPDATE_MAGNITUDE = 1;

    // Update Property
    void _update(uint8_t update);

    // Variables
    RigidBody* _rigid_body;
};
```

The components `x`, `y` and `z` of class `Torque` are defined in Newton-meters. Since the class is publicly derived from `Vector3D`, most operators and methods can be used as explained in section 3.3.

Torques can be scaled with operators `operator *`, `operator *=`, `operator /` and `operator /=` which expect a scalar of type `double`. Two torques can be combined by the use of operators `operator +(const Torque& torque)` and `operator +=(const Torque& torque)`.

If the torque is to be rotated, the according operators `operator +`, `operator -`, `operator +=` and `operator -=` can be used which expect the rotation of type `Rotation` as an argument.

During the simulation run, the magnitude of the torque or force or the point of application of the force can be modified, if required. This might be necessary when, e.g., the direction of gravitational force, the thrust of a propulsion system or the torque generated by the magnetorquer or reaction wheels is to be updated. All forces and torques being applied to a rigid body are merged which results in a single torque and a single force acting on the center of mass of the rigid body. The merging process and the concept of wrenches is explained in 3.11.

As explained in 2.3, caching is required in order to increase the performance of the code and to reduce the simulation duration. Therefore, the merging process is only recomputed when one or more forces or torques have been modified. If the magnitude of the torque is changed, method `_update(uint8_t update)` with argument `_UPDATE_MAGNITUDE` is called. If the torque is bound to a rigid body (see chapter 6), its update method `RigidBody::_update(uint8_t update)` is called with the argument `RigidBody::_UPDATE_TORQUE` notifying the rigid body that one of the acting torques has changed in magnitude. This, in turn, invalidates the wrench computation which is recomputed automatically on-demand. The required reference to the rigid body is stored in the internal variable `_rigid_body`.

3.11 Wrench

Only top-level rigid bodies, like spacecraft or celestial bodies, are free to move whose movement can be described by a translation of and a rotation around the center of mass [52, p. 130]. As already described in the previous two sections, forces and torques can be applied to any parts of a spacecraft, system or assembly. The resultant force acts on the center of mass affecting the translation and the computed torque changes the rotation around the center.

The class `Wrench` takes care of this merging process and is defined as follows excluding private declarations.

```
// Class Wrench
class CubeSim::Wrench
{
public:

   // Constructor
   Wrench(void);
   Wrench(const Force& force, const Torque& torque, double mass);
   Wrench(const std::map<std::string, Force*>& force, const
      std::map<std::string, Torque*>& torque, double mass = 0.0, const
      Vector3D& center = Vector3D());

   // Sign
   const Wrench& operator +(void) const;
   const Wrench operator -(void) const;

   // Scale
   const Wrench operator *(double factor) const;
   Wrench& operator *=(double factor);
   Wrench operator /(double factor) const;
   Wrench& operator /=(double factor);

   // Add
   const Wrench operator +(const Wrench& wrench) const;
   Wrench& operator +=(const Wrench& wrench);

   // Translate [m]
   const Wrench operator +(const Vector3D& distance) const;
   const Wrench operator -(const Vector3D& distance) const;
   Wrench& operator +=(const Vector3D& distance);
   Wrench& operator -=(const Vector3D& distance);

   // Rotate
   const Wrench operator +(const Rotation& rotation) const;
   const Wrench operator -(const Rotation& rotation) const;
   Wrench& operator +=(const Rotation& rotation);
   Wrench& operator -=(const Rotation& rotation);

   // Force [N]
   const Force& force(void) const;
   void force(double x, double y, double z);
   void force(const Force& force);

   // Mass [kg]
   double mass(void) const;
   void mass(double mass);

   // Torque [N*m]
   const Torque& torque(void) const;
   void torque(double x, double y, double z);
   void torque(const Torque& torque);
};
```

An object of type `Wrench` can either be constructed from a single force and single torque `Wrench(const Force& force, const Torque& torque, double mass)` or from two lists `Wrench(const std::map<std::string, Force*>& force, const std::map<std::string, Torque*>& torque,`

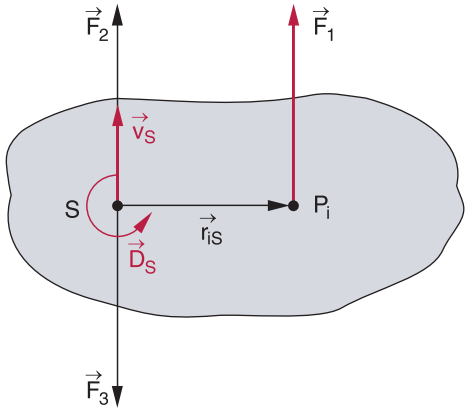

Fig. 3.4. Multiple forces acting on a rigid body and the resultant torque and force at the center of mass are shown [52, p. 130].

`double mass = 0.0, const Vector3D& center = Vector3D())`. In the latter case, optionally the mass of the rigid body and its center of mass can be specified. For the merging process only the center is important but for the combination of two wrenches, the mass is also needed, as being explained later on.

Wrenches can be scaled with operators `operator *`, `operator *=`, `operator /` and `operator /=` which expect a scalar of type `double`. Operators `operator +(const Vector3D& distance)` and `operator +=(const Vector3D& distance)` or `operator -(const Vector3D& distance)` and `operator -=(const Vector3D& distance)` are used to translate a wrench object, which essentially moves the point of application of the resultant force. This can particularly be useful for coordinate transformations. Wrenches can be rotated by the use of operators `operator +(const Rotation& rotation)`, `operator -(const Rotation& rotation)`, `operator +=(const Rotation& rotation)` and `operator -=(const Rotation& rotation)`. This affects the force, its point of application and the torque.

With the use of method `force` the resultant force can either be read out or modified. The same applies to the torque with method `torque` accordingly. The mass of the rigid body, which does not need to be a single physical rigid body, but may also be a compound of multiple bodies, can be read out or modified with method `mass`.

As the wrench object only represents an intermediate result and the underlying objects `Force` and `Torque` already use their caching methods `_update`, this class spares on separate caching strategies.

Let the force $\boldsymbol{F}_1$ acts at point $\boldsymbol{r}_\mathrm{i}$ on the rigid body with its center $\boldsymbol{S}$ (see fig. 3.4). A force $\boldsymbol{F}_2 \| \boldsymbol{F}_1$ acting at $\boldsymbol{r}_\mathrm{S}$ and the anti-parallel force $\boldsymbol{F}_3 = -\boldsymbol{F}_2$ are added which do not change the resultant force or torque since they cancel each other. Forces $\boldsymbol{F}_1$ and $\boldsymbol{F}_3$ represent a pair of forces which produces a torque $\boldsymbol{\tau} = (\boldsymbol{r}_\mathrm{i} - \boldsymbol{r}_\mathrm{S}) \times \boldsymbol{F}_1$. Force $\boldsymbol{F}_2$ acts on the center of mass and results in the acceleration $\ddot{\boldsymbol{x}} = \boldsymbol{F_1}/m$. This can be generalized for multiple forces $\boldsymbol{F}_\mathrm{i}$ acting on the rigid body of mass M and resulting in the force

$$\boldsymbol{F} = \sum \boldsymbol{F}_\mathrm{i} \tag{46}$$

and the torque

$$\boldsymbol{\tau} = \sum (\boldsymbol{r}_\mathrm{i} - \boldsymbol{r}_\mathrm{S}) \times \boldsymbol{F}_\mathrm{i} + \sum \boldsymbol{\tau}_\mathrm{i} \tag{47}$$

with the generalized computation of the center of mass

$$\boldsymbol{r}_\mathrm{S} = \frac{1}{M} \iiint \rho(\boldsymbol{r}) \boldsymbol{r} \mathrm{d}V. \tag{48}$$

The simulation needs to be able to combine wrenches of rigid bodies on the same hierarchical layer, e.g., multiple parts of an assembly or multiple assemblies of a system. For that purpose operators `operator`

`+(const Wrench& wrench)` and `operator +=(const Wrench& wrench)` are defined. Let the first wrench have force $\boldsymbol{F}_1$, point of application $\boldsymbol{r}_1 = \boldsymbol{C}_1$ acting on the center of mass, torque $\boldsymbol{\tau}_1$ being defined for the first rigid body with mass m_1. The same applies for the second wrench object with index 2.

With the common mass $m = m_1 + m_2$, the common center of mass is computed

$$\boldsymbol{C} = \frac{1}{m} \left(\boldsymbol{C}_1 m_1 + \boldsymbol{C}_2 m_2 \right). \tag{49}$$

The total torque results in

$$\boldsymbol{\tau} = \boldsymbol{\tau}_1 + \boldsymbol{\tau}_2 + (\boldsymbol{r}_1 - \boldsymbol{C}) \times \boldsymbol{F}_1 + (\boldsymbol{r}_2 - \boldsymbol{C}) \times \boldsymbol{F}_2 \tag{50}$$

and the common force is

$$\boldsymbol{F} = \boldsymbol{F}_1 + \boldsymbol{F}_2 \tag{51}$$

with the point of application identical to the common center of mass $\boldsymbol{C}$. This process is repeated for all rigid bodies in the same hierarchical layer. The sequence of combinations does not affect the final result.

3.12 Location

The simulation framework uses the right-handed ecliptic coordinate system which is a heliocentric celestial coordinate system. The center of the Sun represents its origin, the plane of reference is the ecliptic plane and the x-axis is pointing to the vernal equinox [2, p. 24]. Though this approach is beneficial for numerical simulations of the entire solar system by avoiding certain coordinate transformations, it is unpractical to represent a location on a celestial body. In particular, on Earth the location of an object is commonly represented by the latitude, the longitude and the altitude which is also used by global navigation satellite systems (see section 9.1). The class `Location` is defined as follows excluding certain private declarations.

```
// Class Location
class CubeSim::Location
{
public:

    // Constructor
    Location(const CelestialBody& celestial_body, const Vector3D& point
        = Vector3D(), const Time& time = Time());
    Location(const CelestialBody& celestial_body, double latitude,
        double longitude, double altitude, const Time& time = Time());

    // Convert to Vector3D
    operator const Vector3D(void) const;

    // Altitude [m]
    double altitude(void) const;
    void altitude(double altitude);

    // Celestial Body
    const CelestialBody& celestial_body(void) const;
    void celectial_body(const CelestialBody& celestial_body);

    // Latitude [rad]
    double latitude(void) const;
    void latitude(double latitude);
```

```
    // Longitude [rad]
    double longitude(void) const;
    void longitude(double longitude);

    // Point
    const Vector3D point(void) const;
    void point(const Vector3D& point);

    // Radius [m]
    double radius(void) const;
    void radius(double radius);

    // Time
    const Time& time(void) const;
    void time(const Time& time);

private:

    // Compute Normal (Distance from the Surface to the Z-Axis along the
       Ellipsoid Normal)
    double _normal(void) const;
};
```

An object of type `Location` can be constructed from the celestial body, the position vector and the time (optionally) or from the celestial body, the latitude, the longitude, the altitude and the time (optionally). Though the location on a celestial body is time-invariant (as for example for the Earth-centered Earth-fixed coordinate system), the time information can be used for GNSS fixes.

An implicit cast operator `operator const Vector3D(void)` is defined for retrieving the position vector of type `Vector3D`. Alternatively, method `point` can be used to retrieve this vector or to modify it. Internally, latitude, longitude and altitude are used to store the location information. If a new location is being specified by a vector of type `Vector3D`, the flattening and the radius of the celestial body (see section 11) are considered and by the use of the numeric Newton-Raphson method, these three properties are computed. This also involves the private method `_normal`.

The method `celestial_body` returns the reference celestial body and also allows to modify it. Due to the different flattening and the radius of the celestial bodies, the return value of `point` changes when the celestial body is modified.

Methods `latitude`, `longitude` and `altitude` are used to read out location information if called without arguments. The properties are updated with the respective method if called with the new value.

The method `time` allows to get the timestamp of the GNSS fix if no arguments are passed or to modify it by passing the new timestamp.

3.13 Orbit

The class `Orbit` allows transformation of Keplerian orbit elements into position and velocity state vectors represented in Cartesian coordinates. This can particularly be useful to initialize the position and velocity of celestial bodies and spacecraft at the begin of the simulation. The six Keplerian orbit elements semi-major axis a, the eccentricity e, the argument of the periapsis ω, the longitude of the ascending node Ω, the inclination i, the mean anomaly M and their geometric representation can be found in fig. 3.5. The

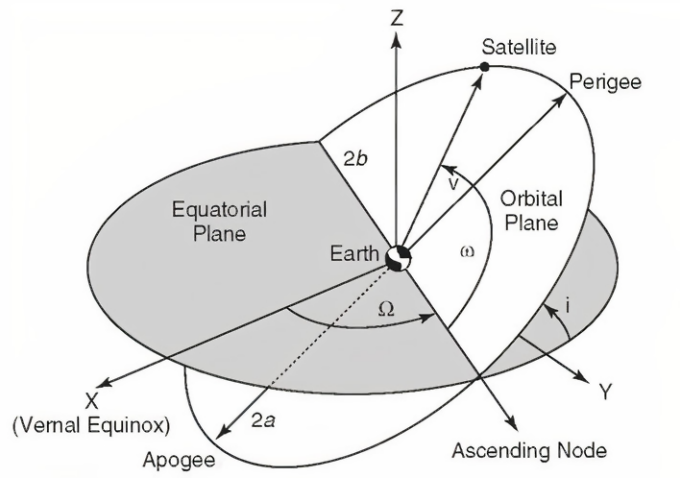

Fig. 3.5. Illustration of the Keplerian orbit elements [32, p. 8]

relation between the mean anomaly M, the eccentric anomaly E and the true anomaly ν can be found in eqs. (**??**) and (62).

The module `Ephemeris` uses Keplerian orbit elements to compute the position and velocity of the planets in the solar system at any point in time (see section 4.6 for details). Moreover, class `Orbit` also allows the computation of Keplerian orbit elements from the position and velocity state vectors of a celestial body or spacecraft around the central body. This allows the observation of the orbit elements of a spacecraft over time which can be particularly useful if a propulsion system is simulated or if the influence of atmospheric drag on the spacecraft is to be observed. The class `Orbit` is defined as follows excluding private declarations.

```
// Class Orbit
class CubeSim::Orbit
{
public:

    // Reference Frames
    static const Rotation REFERENCE_ECLIPTIC;
    static const Rotation REFERENCE_ECI;

    // Constructor
    Orbit(double semimajor_axis, double eccentricity, double
        argument_periapsis, double longitude_ascending, double
        inclination, double mean_anomaly, double period, const Time&
        time, const Rotation& reference = REFERENCE_ECLIPTIC);
    Orbit(const CelestialBody& central, double semimajor_axis, double
        eccentricity, double argument_periapsis, double
        longitude_ascending, double inclination, double mean_anomaly,
        const Time& time, const Rotation& reference = REFERENCE_ECLIPTIC);
    Orbit(const std::set<const CelestialBody*>& central, double
        semimajor_axis, double eccentricity, double argument_periapsis,
        double longitude_ascending, double inclination, double
        mean_anomaly, const Time& time, const Rotation& reference =
        REFERENCE_ECLIPTIC);
    Orbit(const CelestialBody& central, const RigidBody& rigid_body,
        const Rotation& reference = REFERENCE_ECLIPTIC);
```

```
    Orbit(const std::set<const CelestialBody*>& central, const
        RigidBody& rigid_body, const Rotation& reference =
        REFERENCE_ECLIPTIC);

    // Get Apoapsis [m]
    double apoapsis(void) const;

    // Get Argument of Periapsis [rad]
    double argument_periapsis(void) const;

    // Get eccentric Anomaly [rad]
    double eccentric_anomaly(void) const;

    // Get Eccentricity
    double eccentricity(void) const;

    // Get Epoch
    const Time& epoch(void) const;

    // Get standard gravitational Parameter [m^3/s^2]
    double gravitational_parameter(void) const;

    // Get Inclination [rad]
    double inclination(void) const;

    // Get Longitude of the ascending Node [rad]
    double longitude_ascending(void) const;

    // Get mean Anomaly [rad]
    double mean_anomaly(void) const;

    // Get Periapsis [m]
    double periapsis(void) const;

    // Compute Perimeter (Travel Distance for one Revolution) [m]
    double perimeter(void) const;

    // Get Period [s]
    double period(void) const;

    // Get Position relative to Barycenter [m]
    const Vector3D position(void) const;

    // Reference Frame
    const Rotation& reference(void) const;
    void reference(const Rotation& reference);

    // Get semi-major Axis [m]
    double semimajor_axis(void) const;

    // Time
    const Time& time(void) const;
    void time(const Time& time);

    // Get true Anomaly [rad]
    double true_anomaly(void) const;

    // Get Velocity relative to Barycenter [m/s]
    const Vector3D velocity(void) const;
};
```

`Orbit(double semimajor_axis, double eccentricity, double argument_periapsis, double longitude_ascending, double inclination, double mean_anomaly, double period, const Time& time, const Rotation& reference = REFERENCE_ECLIPTIC)` allows the initialization of the object with the Keplerian orbit elements semi-major axis a, the eccentricity e, the argument of the periapsis ω, the longitude of the ascending node Ω, the inclination i, the mean anomaly M and the orbital period T. As the mean anomaly M changes over time, the simulation time needs to be specified as well. It is also possible to specify the reference system. Two reference systems are predefined: the ecliptic and the ECI (Earth-centered Inertial) coordinate system. In contrast to the ecliptic coordinate system, the ECI coordinate system considers the tilt of the Earth rotational axis of $-23.4392811°$ around the x-axis (see section 4.6). The ecliptic coordinate system is commonly used to describe the position and motion of the planets around the Sun and the ECI coordinate system is commonly used to describe the orbit of a spacecraft around the Earth.

Kepler's third law [52, p. 63] describes the relation between the orbital period T, the length of the semi-major axis a and the mass of the central body m

$$T = 2\pi\sqrt{\frac{a^3}{\mu}} \tag{52}$$

with the standard gravitational parameter $\mu = G\,m$ [223, p. 4] and the gravitational constant $G = 6.67384\times10^{-11}\,\mathrm{m/s^2}$ [52, p. 471]. The constructor method computes the standard gravitational parameter from the specified semi-major axis and orbital period

$$\mu = \frac{4\pi^2 a^3}{T^2}. \tag{53}$$

If the orbital period is not known but the single central celestial body is, the constructor `Orbit(const CelestialBody& central, double semimajor_axis, double eccentricity, double argument_periapsis, double longitude_ascending, double inclination, double mean_anomaly, const Time& time, const Rotation& reference = REFERENCE_ECLIPTIC)` can be used instead and the other arguments are the same. In this case, the constructor method computes the standard gravitational parameter $\mu = G\,m$ directly from the mass m of the specified celestial body.

In the rare case that a planet or spacecraft is orbiting two or more central bodies, such as a binary star system, constructor `Orbit(const std::set<const CelestialBody*>& central, double semimajor_axis, double eccentricity, double argument_periapsis, double longitude_ascending, double inclination, double mean_anomaly, const Time& time, const Rotation& reference = REFERENCE_ECLIPTIC)` can be used instead. Details on the transformation from Keplerian orbit elements to Cartesian state vectors $\boldsymbol{x}$ and $\boldsymbol{v}$ representing the relative position and velocity of the orbiting body can be found in section 3.13.1. In contrast to the previous definition, a constant reference to a set of celestial bodies `const std::set<const CelestialBody*>& central` is specified. Here, the constructor method computes the total mass of the virtual central body by summing up the individual masses $m = \sum m_i$.

If the Keplerian orbit elements of the orbit to be defined are not known, the orbit can also be initialized from the specified central and the orbiting body `Orbit(const CelestialBody& central, const RigidBody& rigid_body, const Rotation& reference = REFERENCE_ECLIPTIC)`. Again, the reference frame can optionally be defined. The constructor method takes the current relative position and velocity of the orbiting body and the mass of the central body to compute the orbit. Details on the transformation can be found in section 3.13.2. It is also possible to pass a constant reference to a set of celestial bodies `const std::set<const CelestialBody*>& central` instead of a constant reference to a single celestial body `const CelestialBody& central` as the central body.

The reference coordinate system can be modified or returned with the use of method `reference`.

For eccentric orbits with $0 < e$, the distance between the orbiting and the central body is not constant over time. The farthest point is called apoapsis and the nearest point periapsis which can be retrieved by methods `apoapsis` and `periapsis` respectively. The argument of the periapsis (also called argument of perifocus or argument of pericenter) ω describes the angle between the ascending node and the periapsis. It is returned by method `argument_periapsis`.

The eccentric anomaly describes the position of a body on its elliptic Kepler orbit. Kepler's equation gives the relation between the eccentric anomaly E and the mean anomaly M [33, p. 17]

$$M = E - e \sin E. \tag{54}$$

The following equation describes the relation between the true anomaly ν and the eccentric anomaly [206, p. 1]

$$E = 2 \arctan \frac{\tan \frac{\nu}{2}}{\sqrt{\frac{1+e}{1-e}}}. \tag{55}$$

The methods `eccentric_anomaly` and `eccentricity` return the eccentric anomaly E and the eccentricity e respectively.

The specified time at which the constellation between the orbiting and central body or the Keplerian orbit elements are defined is passed to any of the constructor methods. Internally, the point in time is computed where $M = 0$ which is referred as epoch. Method `epoch` can be used to retrieve it. The aforementioned gravitational parameter μ is returned by calling method `gravitational_parameter`. The inclination of the orbit with respect to the reference coordinate system is denoted as inclination i and is returned by calling method `inclination`. The ascending node is the point where the orbiting celestial body intersects with the plane of reference. The angle between the reference direction and this point is called longitude of the ascending node Ω and is returned by method `longitude_ascending`. The mean anomaly M is the angle between the periapsis and the current position of the orbiting body which increases linearly in time in contrast to the true anomaly ν. The following equation describes the relation between true anomaly ν and eccentric anomaly E

$$\cos \nu = \frac{\cos E - e}{1 - e \cos E}. \tag{56}$$

The so-called *equation of center* allows the expression of the true anomaly as a series of terms of the eccentricity e and mean anomaly M [199, p. 90]

$$\nu = M + \left(2e - \frac{1}{4}e^3\right) \sin M + \frac{5}{4}e^2 \sin 2M + \frac{13}{12}e^3 \sin 3M + \mathcal{O}\left(e^4\right). \tag{57}$$

Methods `mean_anomaly` and `true_anomaly` return the mean anomaly M and the true anomaly ν. The method `perimeter` returns the travel distance of the orbiting celestial body l for one revolution. The Gauss-Kummer relation is used to compute it from the semi-major axis a and the eccentricity e [37, p. 127]

$$l = \pi (a + b) \sum_{n=0}^{\infty} \binom{1/2}{n}^2 h^n = \pi (a + b) \left(1 + \frac{1}{4}h + \frac{1}{64}h^2 + \frac{1}{256}h^3 + \dots\right). \tag{58}$$

The orbital period T is the duration that the orbiting celestial body needs to complete one revolution and can be retrieved by calling method `period`.

By changing the virtual simulation time t with method `time`, the values of the eccentric anomaly E, the mean anomaly M and the true anomaly ν as well as the state vectors $\boldsymbol{x}$ and $\boldsymbol{v}$ representing the

relative position and velocity are automatically updated. The specified time can be returned by calling the method without arguments.

The methods `position` and `velocity` are used to retrieve the actual values of the aforementioned position and velocity vectors $\boldsymbol{x}$ and $\boldsymbol{v}$. In order to obtain the semi-major axis a, method `semimajor_axis` is used.

3.13.1 Transformation of Keplerian Orbit Elements

In this section the transformation of Keplerian orbit elements into Cartesian state vectors is described in detail. It is assumed that the gravitational parameter μ is well defined. The orbital period is computed [205, p. 1]

$$T = 2\pi\sqrt{\frac{a^3}{\mu}} \tag{59}$$

with the semi-major axis a. The mean anomaly is computed from the virtual simulation time t and the mean anomaly at epoch M_0

$$M = M_0 + 2\pi\frac{\triangle t}{T} = \triangle t\sqrt{\frac{\mu}{a^3}} \tag{60}$$

with the elapsed time $\triangle t = t - t_0$ since epoch t_0. In the next step eq. (54) needs to be solved numerically with, e.g., the use of the Newton-Raphson method [147, pp. 22-27]

$$E_0 = M,\ E_{i+1} = E_i - \frac{E_i - e\sin E_i - M}{1 - e\cos E_i}. \tag{61}$$

The true anomaly can be computed from the eccentric anomaly E and eccentricity e

$$\nu = 2\arctan2\left(\sqrt{1+\mathrm{e}}\sin\frac{\mathrm{E}}{2},\ \sqrt{1-\mathrm{e}}\cos\frac{\mathrm{E}}{2}\right) \tag{62}$$

with the arctan2 function defined in eq. (22).

The distance to the central body r_c is computed with the semi-major axis a

$$r_c = a\left(1 - e\cos E\right). \tag{63}$$

The following intermediate state vectors represent the position and velocity in the orbital frame with the z-axis perpendicular to the orbital plane and the x-axis pointing to periapsis

$$\hat{\boldsymbol{r}} = r_c\begin{pmatrix}\cos\nu\\ \sin\nu\\ 0\end{pmatrix},\ \hat{\boldsymbol{v}} = \frac{\sqrt{\mu a}}{r_c}\begin{pmatrix}-\sin E\\ \sqrt{1-e^2}\cos E\\ 0\end{pmatrix}. \tag{64}$$

In the last step the three rotations need to be applied to take into account the rotation around the z-axis (argument of periapsis ω), the rotation around the x-axis (inclination i) and again the rotation around the z-axis (longitude of the ascending node Ω)

$$\boldsymbol{r} = \boldsymbol{R}_z\left(-\Omega\right)\boldsymbol{R}_x\left(-i\right)\boldsymbol{R}_z\left(-\omega\right)\hat{\boldsymbol{r}},\ \boldsymbol{v} = \boldsymbol{R}_z\left(-\Omega\right)\boldsymbol{R}_x\left(-i\right)\boldsymbol{R}_z\left(-\omega\right)\hat{\boldsymbol{v}}. \tag{65}$$

Details on rotation formalism and on rotation matrices in general can be found in section 3.7.

3.13.2 Transformation of Cartesian Coordinates

The transformation from state vectors defined in Cartesian coordinates into Keplerian orbit elements does not involve the necessity of solving equations numerically and is a straightforward process. From the relative position and velocity vectors $\boldsymbol{r}$ and $\boldsymbol{v}$ with respect to the central celestial body, the orbital momentum [206, p. 1]

$$\boldsymbol{h} = \boldsymbol{r} \times \boldsymbol{v} \tag{66}$$

and the eccentricity vector

$$\boldsymbol{e} = \frac{\boldsymbol{v} \times \boldsymbol{h}}{\mu} - \frac{\boldsymbol{r}}{\|\boldsymbol{r}\|} \tag{67}$$

are computed with the use of the gravitational parameter μ. The eccentricity is the absolute value of the eccentricity vector $e = \|\boldsymbol{e}\|$.

Then a vector pointing to the ascending node is calculated

$$\boldsymbol{n} = (0, 0, 1)^{\mathrm{T}} \times \boldsymbol{h} = (-h_y, h_x, 0)^{\mathrm{T}}. \tag{68}$$

In the next step, the true anomaly ν is computed

$$\nu = \begin{cases} \arccos \frac{\boldsymbol{e} \cdot \boldsymbol{r}}{\|\boldsymbol{e}\| \|\boldsymbol{r}\|} & 0 \leqslant \boldsymbol{r} \cdot \boldsymbol{v} \\ 2\pi - \arccos \frac{\boldsymbol{e} \cdot \boldsymbol{r}}{\|\boldsymbol{e}\| \|\boldsymbol{r}\|} & \text{otherwise.} \end{cases} \tag{69}$$

The inclination i follows from the orbital momentum vector

$$i = \arccos \frac{h_z}{\|h\|}. \tag{70}$$

The eccentric anomaly E can further be computed from the true anomaly ν and the eccentricity e

$$E = 2 \arctan \frac{\tan \frac{\nu}{2}}{\sqrt{\frac{1+e}{1-e}}}. \tag{71}$$

The longitude of the ascending node Ω results from the vector $\boldsymbol{n}$ pointing towards the ascending node

$$\Omega = \begin{cases} \arccos \frac{n_x}{\|\boldsymbol{n}\|} & 0 \leqslant n_y \\ 2\pi - \arccos \frac{n_x}{\|\boldsymbol{n}\|} & \text{otherwise.} \end{cases} \tag{72}$$

The argument of periapsis ω is computed as follows

$$\omega = \begin{cases} \arccos \frac{\boldsymbol{n}\,\boldsymbol{e}}{\|\boldsymbol{n}\| \|\boldsymbol{e}\|} & 0 \leqslant e_z \\ 2\pi - \arccos \frac{\boldsymbol{n}\,\boldsymbol{e}}{\|\boldsymbol{n}\| \|\boldsymbol{e}\|} & \text{otherwise.} \end{cases} \tag{73}$$

Eq. (54) is used to calculate the mean anomaly M from the eccentric anomaly E and finally the semi-major axis a is obtained

$$a = \frac{1}{\frac{2}{\|\boldsymbol{r}\|} - \frac{\|\boldsymbol{v}\|^2}{\mu}}. \tag{74}$$

3.14 Polygon

A polygon is a figure that is described by connected straight line segments defined by so-called *vertices*. Though a polygon is usually referred to a plane figure, the context is also extended to three dimensions. Classes `Polygon2D` and `Polygon3D` are defined for two and three dimensions respectively. As both classes share considerable similarities, the template class `_Polygon<T>` is created for code reuse.

3.14.1 Template class _Polygon

The template class `_Polygon<T>` defines a class which allows to store multiple vertices (either of type `Vector2D` or `Vector3D`). It is defined as follows.

```
// Class _Polygon
template <typename T> class CubeSim::_Polygon
{
public:

    // Constructor
    _Polygon(void);
    _Polygon(const std::vector<T>& vertex);

    // Insert Vertex
    void operator +=(const T& vertex);
    void operator +=(const std::vector<T>& vertex);
    void insert(const T& vertex);
    void insert(const std::vector<T>& vertex);

    // Get Vertex
    const std::vector<T>& vertex(void) const;
    T& vertex(size_t index) const;

    // Compute Perimeter
    double perimeter(void) const;

    // Remove Vertex
    void remove(size_t index);

protected:

    // Variables
    std::vector<T> _vertex;
};
```

The class type `T` denotes the type of a vertex. In this framework the instantiated classes `Polygon2D` with vertices of type `Vector2D` and `Polygon3D` with vertices of type `Vector3D` are defined. It is important to note that the class type `T` provides method `norm` which calculates the Euclidean norm of a vertex.

The constructor `_Polygon(void)` allows the creation of an object that does not contain any vertices. By the use of constructor `_Polygon(const std::vector<T>& vertex)` an STL vector containing the vertices can be passed.

With methods `operator +=` or `insert` one or more vertices can be appended to the list of existing vertices.

Method `vertex` allows retrieval of a certain vertex if the index is specified or of the entire list of vertices if being called without arguments. In the latter case an r-value of type `std::vector<T>` is returned.

As mentioned before, the class computes the perimeter of the polygon by calling method `perimeter`. The method sums up the distances between two adjacent vertices including the distance between the last and the first vertex in the list.

Method `remove` allows the removal of a certain vertex specified by its index.

3.14.2 Polygon2D

The class `Polygon2D` derives from the template class `_Polygon<T>` and uses two-dimensional vertices of type `Vector2D`. For the defined methods `area`, `center`, `inside` and `order`, the polygon ρ must be Star-shaped: there exists a point $\boldsymbol{z}$ so that for each point $\boldsymbol{p} \in \rho$ the segment $\boldsymbol{zp}$ lies entirely in ρ. For class `Polygon2D` additionally the condition $z \equiv \mathbf{0}$ must be met. The class is defined as follows.

```
// Class Polygon2D
class CubeSim::Polygon2D : public _Polygon<Vector2D>
{
public:

   // Order of Vertices
   static const uint8_t ORDER_CW = 1;
   static const uint8_t ORDER_CCW = 2;

   // Constructor
   Polygon2D(void);
   Polygon2D(const std::vector<Vector2D>& vertex);

   // Compute Area
   double area(void) const;

   // Compute Center of Mass
   const Vector2D center(void) const;

   // Insert Vertex
   void insert(double x, double y);
   using _Polygon<Vector2D>::insert;

   // Check if Point is inside
   bool inside(const Vector2D& point) const;

   // Compute Order of Vertices
   uint8_t order(void) const;

   // Check if Star-shaped
   bool star(void) const;
};
```

The constructor `Polygon2D(void)` allows the creation of an object that does not contain any vertices. By the use of constructor `Polygon2D(const std::vector<Vector2D>& vertex)` an STL vector containing the vertices can be passed.

Method `area` computes the area of the polygon. For that purpose, the triangles spanned by two adjacent vertices and the origin are summed up.

The center of mass of the polygon can be computed by the use of method `center`. The method calculates the center of mass and the area of the individual triangles and computes the resulting center of mass.

In order to check if a point is inside the polygon, method `inside` can be used. It parses the individual triangles the polygon is comprised of and determines if the point lies in one triangle.

A polygon is a positively oriented curve (counter clockwise) such that when moving along it, one has the interior of the polygon to the left and the exterior on the right. Negatively oriented curves (clockwise) have the interior on the right and the exterior on the left when moving along them. Method `order` can be used to determine the order of the polygon vertices. It either returns `ORDER_CW` (clockwise) or `ORDER_CCW` (counter clockwise).

By the use of method `star`, one can find out if the polygon is Star-shaped, such as defined at the begin of this section.

Within the framework, two-dimensional polygons are used to define the base of prisms (see section 7.4).

3.14.3 Polygon3D

The class `Polygon3D` derives from the template class `_Polygon<T>` and uses three-dimensional vertices of type `Vector3D`. It is defined as follows.

```
// Class Polygon3D
class CubeSim::Polygon3D : public _Polygon<Vector3D>
{
public:

   // Constructor
   Polygon3D(void);
   Polygon3D(const std::vector<Vector3D>& vertex);

   // Insert Vertex
   void insert(double x, double y, double z);
   using _Polygon<Vector3D>::insert;
};
```

The constructor `Polygon3D(void)` allows the creation of an object that does not contain any vertices. By the use of constructor `Polygon2D(const std::vector<Vector3D>& vertex)` an STL vector containing the vertices can be passed.

Apart from an additional overloaded method `insert`, the class does not define other methods than already defined in the template class `_Polygon`.

Within the framework, three-dimensional polygons are not used but could be beneficial in the future, such as they could be used to define the path of the winding wire of magnetic components like magnetorquers.

3.15 Materials

Materials can be assigned to parts (see chapter 7) to define their physical properties in terms of mass and color. It is required to properly set the material of a part to rely on the computation of mass, inertia, center of mass but also on dynamic properties such as momentum or angular momentum. The default color of the assigned material is useful when exporting the corresponding part or the assembly, the system or the spacecraft containing the part. Details can be found in section 3.16.

The class `Material` is defined as follows excluding private declarations.

```
// Class Material
class CubeSim::Material
{
public:

   // Default Materials
   static const Material ALUMINUM;
   static const Material COPPER;
   static const Material EPOXY;
   static const Material FR4;
   static const Material PEEK;
   static const Material POLYAMIDE;
   static const Material PTFE;
   static const Material STEEL;

   // Constructor
   Material(const std::string& name = "", double density = 0.0, const
      Color& color = Color::BLACK);

   // Copy Constructor (Part is reset)
   Material(const Material& material);

   // Assign (Part is maintained)
   Material& operator =(const Material& material);

   // Color
   const Color& color(void) const;
   void color(const Color& color);

   // Density [kg/m^3]
   double density(void) const;
   void density(double density);

   // Name
   const std::string& name(void) const;
   void name(const std::string& name);
};
```

The materials aluminum, copper, epoxy resin, FR4, PEEK, PTFE, polyamide and steel are predefined and accessible via the static member constants `ALUMINUM`, `COPPER`, `EPOXY`, `FR4`, `PEEK`, `POLYAMIDE` and `STEEL`. Their properties name, density and typical color are predefined.

By the use of constructor `Material(const std::string& name = "", double density = 0.0, const Color& color = Color::BLACK)` new materials can be created and used later on. The arguments name, density and color are optional.

The color of the material is returned by method `color` as an r-value of type `Color` when no arguments are supplied. In order to modify the color, the new color is passed to this method. See section 3.17 for details.

Method `density` is called without arguments to get the physical density of the material. Passing another density to the method, changes its value.

The property name can be useful for automatic generation of a DPL (Declared Parts List), which is usually required during the development of components or systems for space flight. The name can be

Table 3.1. Material properties used in CubeSim (units in kg/m^3).

Material	Alloy / Blend	Density	Color	Reference
Aluminum	EN AW-6061	2700	#ADB2BD	[94, p. 2]
Copper	Cu-ETP	8900	#C06935	[15, p. 1]
Epoxy Resin	Araldite AV 138M-1, HV 998-1	1700	#505050	[96, p. 1]
FR4	TG150, unfilled	2000	#285000	[45, p. 1]
PEEK	VICTREX® PEEK 450G	1300	#8C7846	[244, p. 13]
Polyamide	PA66 / Nylon	1150	#F2F2F2	[56, p. 1]
PTFE	unfilled	2160	#FFFFFF	[258, p. 1]
Steel	1.4301 / A2	7900	#E0DFDB	[93, p. 2]

retrieved by calling method `name` without arguments. In order to change the name, the new name is passed to the method.

Changing the density of an object of type `Material` informs the part, the material is assigned to, via method `_update` since with an updated density, the mass of the part needs to be recomputed in accordance to eq. (156). See section 6.1.9 for details.

The density and color used for the predefined materials is found in tbl. 3.1. The color value is specified in hexadecimal notation (see section 3.17) and references are listed to justify the density value.

3.16 CAD

It can be useful to export objects of the CubeSim simulation, such as celestial bodies, spacecraft, systems, assemblies or parts, to allow further processing with compatible third party applications such as CAD (Computer Aided Design) tools. This can particularly be useful to check, e.g., if an assembly built up in CubeSim is correct. For that purpose, the open standard X3D (Extensible 3D), based on XML (Extensible Markup Language), is used to describe the components and scenery [105].

The dimensions, position, orientation and color of all parts are considered when the X3D code is generated by the exporter (see also section 3.17 and chapter 7).

The class `CAD` is defined as follows excluding private declarations.

```
// Class CAD
class CubeSim::CAD
{
public:

   // Export Code
   const std::string code(void) const;

   // Insert Item (only Reference is stored)
   void operator +=(const Assembly& assembly);
   void operator +=(const CelestialBody& celestial_body);
   void operator +=(const Part& part);
   void operator +=(const Spacecraft& spacecraft);
   void operator +=(const System& system);
   void insert(const Assembly& assembly);
   void insert(const CelestialBody& celestial_body);
   void insert(const Part& part);
   void insert(const Spacecraft& spacecraft);
```

```
    void insert(const System& system);
};
```

The class solely uses the default constructor to create an object of type `CAD`. Objects of type `Assembly`, `CelestialBody`, `Part`, `Spacecraft` and `System` can be inserted into the by either using the addition assignment operator `operator +=` or method `insert`. In both cases a constant reference of the desired object is passed as an argument.

Internally, only the addresses (pointers) of all passed objects are stored in STL vectors. This allows to modify an object during the simulation run and to generate periodic exports without the need to setup the `CAD` export every time. This can particularly be useful, e.g., to animate the orientation of a spacecraft during the detumbling or fine-pointing phase (see section 9.8).

Method `code` is used to generate and return the X3D export code. The method does not expect any arguments.

For the following example, a simplified 1U CubeSat according to the CalPoly CubeSat specifications [31, p. 23] was designed and exported with the aid of class `CAD`. The spacecraft consists of four natural color aluminum rails, four green FR4 (Flame Retardant 4) side panels with two black Spectrolab solar cells [218, p. 1] each, two beige FR4 boards acting as top and bottom panels and a blue cylinder representing part of the allowable volume within the P-POD (Poly Picosatellite Orbital Deployer) deployment system [31, p. 11].

To build up the satellite, assemblies were used to represent the side panels, prisms to define the top and bottom panel and a cylinder representing the additional usable volume on the top. Features such as material definition, manual coloring and rotation and positioning of components were used. Once the satellite is properly defined, the generation of the X3D export only requires a few lines of code.

Tests with different X3D viewers have revealed strong differences in quality and accuracy. The Google Chrome extension X3D Viewer version 1.1.7 [219] does not interpret extrusions used for prisms correctly and FreeWRL version 4.0 [137] could not interpret assemblies correctly. Finally, the instantreality Instant Player version 2.8 [100] was used to render the X3D export correctly as shown in fig. 3.6.

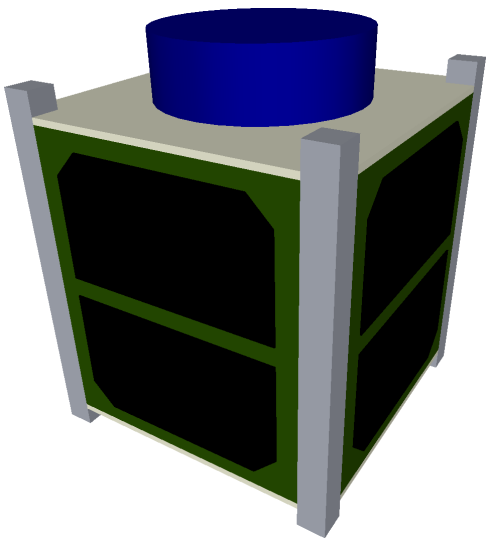

Fig. 3.6. instantreality Instant Player rendering of a 1U CubeSat.

3.17 Color

Colors are particularly required for the CAD exporter (see section 3.16). This allows the user to import the exported X3D code into a third-party software and to verify if all materials have properly been assigned to all parts. It also allows to highlight some components by assigning different colors.

The additive RGB (Red Green Blue) color model is used to represent any color by specifying the ratio between the primary colors red, green and blue.

The class `Color` is defined as follows excluding some declarations.

```
// Class Color
class Color
{
public:

    // Default Colors
    static const Color ALICEBLUE;
    static const Color ANTIQUEWHITE;
    ...
    static const Color YELLOW;
    static const Color YELLOWGREEN;

    // Constructor
    Color(void);
    Color(uint8_t red, uint8_t green, uint8_t blue);

    // Convert to String
    operator const std::string(void) const;

    // Blue
    uint8_t blue(void) const;
    void blue(uint8_t blue);

    // Green
    uint8_t green(void) const;
    void green(uint8_t green);

    // Red
    uint8_t red(void) const;
    void red(uint8_t red);
};
```

A variety of predefined colors from `ALICEBLUE` to `YELLOWGREEN` allows the user to directly use the desired colors without specifying the values for red, green and blue.

The default color (black) can be created with the default constructor `Color(void)` which initializes values for red, green and blue to zero. By the use of constructor `Color(uint8_t red, uint8_t green, uint8_t blue)`, the new color will be initialized to the red, green and blue values as specified.

The cast operator `const std::string(void)` is used to convert the values for red, green and blue into the hexadecimal color code used for HTML (Hyper Text Markup Language) such as #FFFF00 representing the color yellow or #808080 representing gray [255, p. 87], [1].

Methods `red`, `green` and `blue` return the values for red, green and blue if no arguments are passed. By specifying the new value when calling these methods, the corresponding color component is modified.

3.18 Constant

The mathematical and natural constants being used within the CubeSim framework are also accessible to the user.

The static class `Constant` provides theses constants and is defined as follows.

```
// Class Constant
class CubeSim::Constant
{
public:

   // Gravitational Constant [m^3/kg/s^2]
   static const double G;

   // Pi
   static const double PI;

   // Stefan-Boltzmann Constant [W/m^2/K^4]
   static const double SIGMA;

   // Epsilon (for Vector and Matrix Comparison)
   static const double EPSILON;
};
```

The gravitational constant `G` defined with $G = 6.67430(15) \times 10^{-11}\,\mathrm{N\,m^2\,kg^{-2}}$ [175, p. 2] is used by module `Gravitation` to compute the attractive force between two rigid bodies (see section 4.1).

The first 20 digits of the versatile constant `PI` are defined with $\pi = 3.14159265358979323846$ [17, p. 8]. The constant is used by different modules, systems and parts.

The Stefan-Boltzmann constant `SIGMA` defined with $\sigma = 5.670374419 \times 10^{-8}\,\mathrm{W\,m^{-2}\,K^{-4}}$ [175, p. 1] is used by module `Light` to compute the emitted light of stars due to thermal radiation (see section 4.2).

For the comparison between two elements of a vector or a matrix constant `EPSILON` is being used with $\varepsilon = 100\varepsilon_0$ and the machine epsilon $\varepsilon_0 = 2.220446 \times 10^{-16}$. See eq. (5) and section 3.2 for details.

Chapter 4

Modules

Modules build a group of classes that model certain behavior relevant to the simulation scenario. This can involve Newtonian motion to simulate the movement and rotation of celestial bodies and spacecraft, the computation of irradiance from stars and albedo radiation of celestial bodies. Other modules model the magnetic and gravity field of the Earth and their influence on spacecraft, such as the torques generated by magnetorquers or the output signal of simulated magnetometers.

When setting up a simulation, the user decides which modules to use since omitting irrelevant modules reduces complexity and speeds up the simulation. Some of the modules can also be tweaked to make a trade-off between accuracy and simulation duration, such as the number of regions to be used for albedo radiation computation.

Considering the hierarchy of the simulation framework, the class `Module` is derived from classes `RigidBody` and `List<Module>::Item`. The class `Module` is defined as follows excluding certain private declarations.

```
// Module
class CubeSim::Module : public Behavior, public List<Module>::Item
{
public:

   // Class Albedo
   class Albedo;

   // Class Ephemeris
   class Ephemeris;

   // Class Gravitation
   class Gravitation;

   // Class Light
   class Light;

   // Class Magnetics
   class Magnetics;

   // Class Motion;
   class Motion;

   // Constructor
   Module(void);

   // Copy Constructor (Simulation Reference is reset)
```

```
    Module(const Module& module);

    // Assign (Simulation Reference is maintained)
    Module& operator =(const Module& module);

    // Clone
    virtual Module* clone(void) const = 0;

    // Get Simulation
    Simulation* simulation(void) const;

private:

    // Variables
    Simulation* _simulation;
};
```

The class `Behavior` adds the virtual methods `_init` and `_behavior` which are to be defined by the derived class. As defined in section 2.2, the inherited virtual method `_behavior` represents the functionality of the module and its implementation might be very different depending on the purpose of the module. The virtual method `_init` can be used for module initialization.

`Module` is derived from `List<Module>::Item` to allow configured objects to be inserted into the simulation (see chapter 12 for details). This requires the definition of method `clone`.

The class defines a copy constructor `Module(const Module& module)` to create a duplicate from an existing module. The new object is not part of any simulations.

While the behavior of systems focuses on the input and output properties, the behavior of modules focuses on a certain physical effect, as explained above. For that, the method `_behavior` needs to access all objects inserted into the simulation, such as all spacecraft or celestial bodies, which is accomplished by implicitly setting the internal pointer `_simulation` once the module is inserted into a simulation. By calling `simulation` without any arguments, the pointer to the simulation, type `Simulation*`, is returned. If the module was not inserted into a simulation, `nullptr` is returned.

A number of predefined modules is provided to cover most of the simulation scenarios for small spacecraft which are described in detail in this section. In addition, the user is free to define new modules by deriving a class from `Module`. If required, method `_behavior` can be defined to allow the module continuous execution in the framework. This can be relevant for periodically updating the motion of the celestial bodies (see section 4.5) while other modules provide interfaces which are requested on demand only (e.g. see section 4.3).

The predefined modules `Gravitation`, `Light`, `Albedo`, `Magnetics`, `Motion` and `Ephemeris` are discussed in the following sections of this chapter.

4.1 Gravitation

In order to simulate the movement of the planets in our Solar system around its barycenter or to propagate the trajectory of a spacecraft around the Earth, gravitation is the prevailing force in space. It acts between any pieces of matter in general, but module `Gravitation` only computes gravitational forces between celestial bodies and between celestial bodies and spacecraft. The force between spacecraft is negligible in most cases and is therefore not considered. For any two masses m_1 and m_2 at the distance $\boldsymbol{r}$, the acting gravitational force is

$$\boldsymbol{F}_1 = -\boldsymbol{F}_2 = G\,\frac{m_1 m_2}{\|\boldsymbol{r}\|^3}\,\boldsymbol{r} \tag{75}$$

with the gravitational constant $G = 6.67430(15) \times 10^{-11}\,\mathrm{N\,m^2\,kg^{-2}}$ [175, p. 2]. Newton was the first who derived this relation from empirical observations and recognized that the force is reduced with the square of the distance between the masses. Further he discovered the principle *actio est reactio*, which describes that if a body m_1 exerts a force $\boldsymbol{F}_1$ on body m_2, body m_1 experiences the same force in opposite direction $\boldsymbol{F}_2 = -\boldsymbol{F}_1$ as seen in eq. (75). It has to be mentioned that this equation applies to point masses only but it is possible to generalize it to interacting bodies with spatial extent. For the following, one point mass m and a body with spatial extent with total mass M are considered. The distance between the point mass and the center of mass of the other body is $\boldsymbol{r}$. A mass element of the body with spatial extent is denoted with $\mathrm{d}m$ which relative position to the center of mass is $\boldsymbol{x}_m$. This leads to the gravitational force

$$\boldsymbol{F} = \int G\,m\,\mathrm{d}m\,\frac{\boldsymbol{r} - \boldsymbol{x}_m}{\|\boldsymbol{r} - \boldsymbol{x}_m\|^3}. \tag{76}$$

In [52, p. 67ff] it is assumed that the body with spatial extent is a sphere which allows analytical computation of eq. (76). It is shown that the sphere attracts the point mass in the same way as if the entire mass M is concentrated at the center. Another approximation regarding the size of the celestial bodies and their distance can be made. If their mean radius is significantly smaller than their distance, eq. (76) can be approximated as

$$\boldsymbol{F} \xrightarrow[\boldsymbol{r} \gg \boldsymbol{x}_m]{} \int G\,m\,\mathrm{d}m\,\frac{\boldsymbol{r}}{\|\boldsymbol{r}\|^3} = G\,\frac{M\,m\,\boldsymbol{r}}{\|\boldsymbol{r}\|^3} \tag{77}$$

which equals to the gravitational force between two point masses as shown in eq. (75). Module `Gravitation` computes the gravitational forces between celestial bodies and between celestial bodies and spacecraft with the assumption of point masses. This also means that the spatial extent of a celestial bodies or their flattening at the poles is completely discarded. As shown for module `Motion` (see section 4.5), this approach is sufficiently accurate to predict the trajectories of the planets in the Solar system. If a spacecraft orbits the Earth in LEO (Low Earth Orbit), the flattening of Earth needs to be considered to guarantee orbit stability. Vice versa, the flattening of the Earth can also be derived from the observation of spacecraft orbits [120].

4.1.1 Implementation

The class `Gravitation` is used to simulate the gravitational force between celestial bodies and between celestial bodies and spacecraft. For that purpose Newton's law of gravitation for point masses is used. It assumes a constant density throughout the celestial body or spacecraft and does not consider flattening of the celestial body. The class is derived from `Module` and defined as follows excluding certain private declarations.

```
// Class Gravitation
class CubeSim::Module::Gravitation : public Module
{
public:

    // Constructor
    Gravitation(double time_step = _TIME_STEP);

    // Clone
```

```
    virtual Module* clone(void) const;

    // Compute gravitational Field [m/s^2]
    const Vector3D field(const Vector3D& point) const;

    // Time Step [s]
    double time_step(void) const;
    void time_step(double time_step);

private:

    // Default Time Step [s]
    static const double _TIME_STEP;

    // Force Name
    static const std::string _FORCE;

    // Behavior
    virtual void _behavior(void);

    // Compute gravitational Field [m/s^2]
    const Vector3D _field(const CelestialBody& celestial_body, const
        Vector3D& point) const;
};
```

The class defines the constructor `Gravitation(double time_step = _TIME_STEP)` which allows the initialization of the time step. Smaller time steps allow more accurate simulation results but require more computational time.

During the simulation, the time step can be retrieved by calling method `time_step` without arguments or modified by passing the new value to the method.

The computation of the gravitational field strength is done by method `field` which expects the point of interest in the global coordinate system as parameter. The method is iterating over all celestial bodies in the simulation, computing their contribution to the gravitational field (see method `_field`) and summing up the resulting vectors. Finally the sum, corresponding to the gravitational field at the requested location, is returned.

At the begin of the simulation, a named force `"Gravitation"` of class `Force` is inserted into each celestial body and spacecraft in the simulation which is later controlled in dependence of the computed gravitational force.

The class defines method `_behavior` which consists of an infinite loop that is executed periodically when a time step is elapsed. The updated positions of the celestial bodies and spacecraft are used to compute and update the gravitation force acting on each celestial body and spacecraft in the simulation.

Since class `Module` is derived from `List<Module>::Item`, class `Gravitation` needs to define method `clone`.

4.1.2 Verification

The mean Earth radius is $r_\mathrm{E} = 6.37101 \times 10^6\,\mathrm{m}$ and the Earth mass is $m_\mathrm{E} = 5.97219 \times 10^{24}\,\mathrm{kg}$ [166]. Using eq. (75), the gravitation acceleration on Earth is computed

$$g = G\,\frac{m_\mathrm{E}}{r_\mathrm{E}^2} = 9.82026\,\mathrm{m/s^2} \tag{78}$$

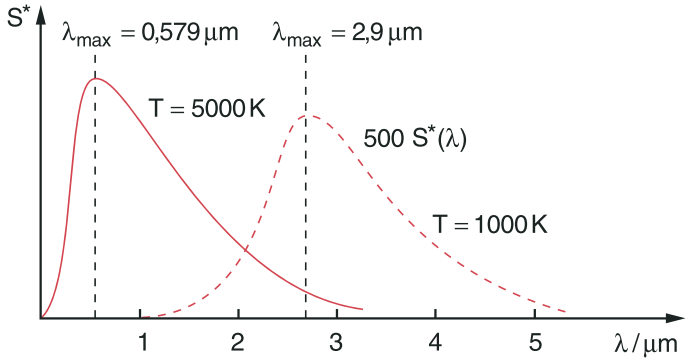

Fig. 4.1. Electromagnetic spectra for 5,000 K (solid) and 1,000 K (dashed) surface temperature [50, p. 83].

In a set up CubeSim simulation, the gravitational field at the point $\boldsymbol{x} = (r_\mathrm{E} \;\; 0 \;\; 0)^\mathrm{T}$ in the Earth body frame was computed and resulted in $9.82025\,\mathrm{m/s^2}$ which matches the analytical calculation very well.

In section 4.5.4, module `Motion` is verified for which the accurate computation of gravitational forces is essential.

4.2 Light

Similar to planets, stars are celestial bodies that have the shape of an ellipsoid. In contrast to planets, stars mainly consist of gaseous compounds, which undergo a fusion reaction to heavier elements and release energy. Also because of their lower density, gravitation is the determining force for their shape. The Sun rotates once every 27 days [92, p. 1] and therefore the centripetal force at the equator is almost five orders of magnitude smaller than the gravitational force. This results in a rather low flattening (also called oblateness) of 5×10^{-5} [257]. Though there are stars which have significantly higher angular rates and therefore stronger flattening [259, p. 4], the focus of module `Light` is laid on the Sun and similar stars where the assumption of a spherical shape is a reasonable approximation.

Depending on the surface temperature of the star, light of a characteristic electromagnetic spectrum is emitted, following the Stefan-Boltzmann law [50, p. 83f]. It allows computation of the radiated power

$$P = \sigma\, \varepsilon(T)\, A\, T^4 \tag{79}$$

with the area of the celestial body A, the surface temperature T, the temperature dependent emissivity $\varepsilon(T)$ and the factor $\sigma = 5.67 \times 10^{-8}\,\mathrm{W\,m^{-2}K^{-4}}$. Most stars are considered as black bodies with $\varepsilon \approx 1$.

The peak of the spectrum at the wavelength λ_max is related to T by Wien's displacement law [50, p. 82f]

$$\lambda_\mathrm{max} T = 2.897 \times 10^{-3}\,\mathrm{m}. \tag{80}$$

In fig. 4.1, the electromagnetic spectra of different radiating bodies are shown. The shape of the intensity distribution of the thermal radiation is in alignment with eq. (80).

In order to compute the irradiance of a test object, such as a photo detector, different approaches are possible. The star can be assumed as a point light source and the irradiance at the distance $\boldsymbol{r}$ results in

$$E_\mathrm{point} = \sigma\, 4\pi\, r_\mathrm{s}^2\, T^4 \frac{1}{4\pi \left\|\boldsymbol{r}\right\|^2} = \frac{\sigma\, r_\mathrm{s}^2\, T^4}{\left\|\boldsymbol{r}\right\|^2} \tag{81}$$

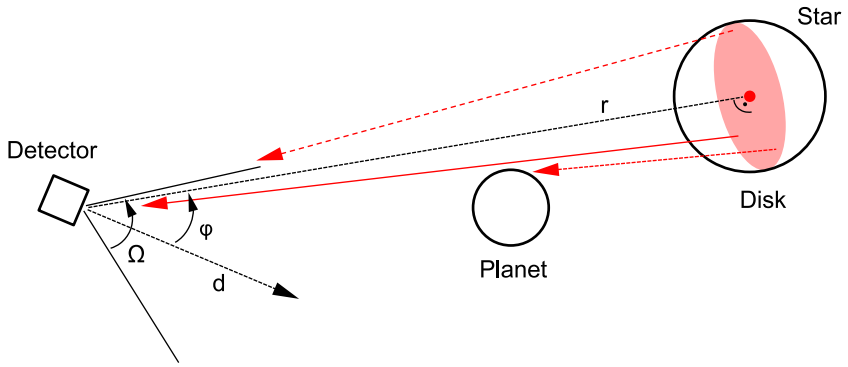

Fig. 4.2. Detector facing direction $\boldsymbol{d}$ with an opening angle Ω irradiated by a star at relative distance $\boldsymbol{r}$.

where r_s denotes the mean radius of the star. Ray-tracing is used to look for any obstacles (other celestial bodies) between the center of the star and the location of the observer. This approximation might be sufficient for large distances $\boldsymbol{r}$ and small radii r_s but effects such as penumbra (half-shadow) are not covered by this model.

In a more complex model, the surface of the star is represented as a disk as projection of a sphere. The disk is subdivided into different areas and the point light source model is applied to each area (see fig. 3.1). This leads to the irradiance by a single point out of n on the disk

$$E_\mathrm{disk} = \frac{1}{n} E_\mathrm{point}. \tag{82}$$

Also in this model ray-tracing is used but if only a fraction of the light rays is blocked by another celestial body, still a reduced amount of light is detected by the observer. This can be useful for simulating the irradiance when a spacecraft is disappearing behind the horizon or during a solar eclipse.

Module `Light` assumes a detector of area A which is collecting the light from all stars in the simulation which is not shaded by other celestial bodies (see fig. 4.2). The detector has a certain opening angle Ω and is facing a certain direction $\boldsymbol{d}$. The relative distance to the center of the star is $\boldsymbol{r}$ and the enclosing angle with $\boldsymbol{d}$ is

$$\varphi = \pi - \arccos \frac{\boldsymbol{r}\,\boldsymbol{d}}{\|\boldsymbol{r}\|\,\|\boldsymbol{d}\|}. \tag{83}$$

The relative distance to the center of the star $\boldsymbol{r}$ is used for the single ray of light if the star is considered as point light source (red dot). For the disk light source model, the disk (red shaded) is used as a projection of the spherical shape of the star and is perpendicular to $\boldsymbol{r}$. Light rays that impact on the detector enclosing an angle greater than $\Omega/2$ as well as light rays that are blocked by other celestial bodies are discarded (dashed red arrows). Considering the cosine losses because of angle φ, the effective detector area A is reduced leading to the radiation flux

$$\Phi = \cos\varphi\, A\, E \tag{84}$$

with E denoting the irradiance at the location of the senor, either E_point or E_disk depending on the choice of the light model.

4.2.1 Implementation

The class `Light` allows to compute the irradiance at a certain location where the detector is facing an arbitrary direction and features a defined opening angle. For that purpose it simulates the emitted light

by a single star or by all stars in the simulation under the consideration of the the surface temperature. `Light` is derived from class `Module` and defined as follows excluding private declarations.

```
// Class Light
class CubeSim::Module::Light : public Module
{
public:

    // Default Resolution (Number of Points)
    static const uint32_t DEFAULT_RESOLUTION = 100;

    // Computation Models
    static const uint8_t MODEL_POINT = 1;
    static const uint8_t MODEL_DISK = 2;

    // Constructor
    Light(uint8_t model = MODEL_POINT, uint32_t resolution =
        DEFAULT_RESOLUTION);
    Light(CelestialBody& celestial_body, uint8_t model = MODEL_POINT,
        uint32_t resolution = DEFAULT_RESOLUTION);

    // Specific Celestial Body
    const CelestialBody* celestial_body(void) const;
    void celestial_body(const CelestialBody* celestial_body);

    // Clone
    virtual Module* clone(void) const;

    // Compute Irradiance [W/m^2]
    double irradiance(const Vector3D& point, const Vector3D& direction,
        double angle = Constant::PI) const;

    // Computation Model
    uint8_t model(void) const;
    void model(uint8_t model);

    // Resolution (Number of Grid Points)
    uint32_t resolution(void) const;
    void resolution(uint32_t resolution);
};
```

The class features different constructors. Using `Light(uint8_t model = MODEL_POINT, uint32_t resolution = DEFAULT_RESOLUTION)` allows to define the used model, either the point light source by passing `MODEL_POINT` to parameter `model` or the disk light source by passing `MODEL_DISK`. For the latter model, the resolution can be passed which defines the number of points on the disk (see fig. 3.1). If parameter `resolution` is omitted, the default resolution `DEFAULT_RESOLUTION` (100 points) is used. Celestial bodies with a surface temperature lower than $2,000\,\mathrm{K}$ are discarded to increase simulation speed (see the Hertzsprung-Russel diagram [59] for details on the spectral classes of stars). If only a certain star shall be used to compute the irradiance with this instance of `Light`, constructor `Light(CelestialBody& celestial_body, uint8_t model = MODEL_POINT, uint32_t resolution = DEFAULT_RESOLUTION)` which offers the third parameter `celestial_body`.

The celestial body can also be retrieved by the use of method `celestial_body` when no arguments are passed. It is modified by passing an object of type `CelestialBody` to the method.

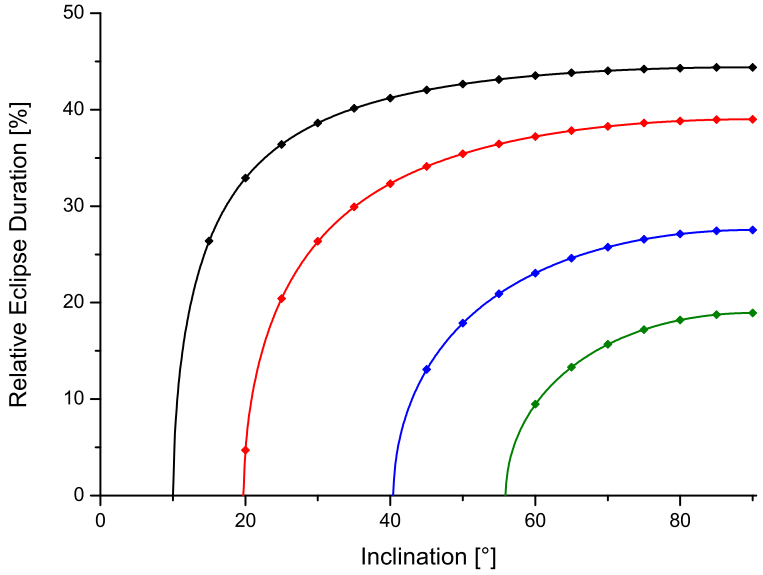

Fig. 4.3. Relative eclipse duration as a result of CubeSim simulation (points) compared with analytical derivation (lines) for different altitudes: 100 km (black), 400 km (red), 2,000 km (blue), 5,000 km (green).

The simulation mode can also be changed even during a running simulation by passing the new model to method `model` (either `MODEL_POINT` or `MODEL_DISK`). The currently used model can be retrieved by calling the method without arguments.

If model `MODEL_DISK` is used, method `resolution` is relevant since it allows to modify the number of points on the disk by passing the new number. If the method is called without arguments, it returns the current resolution.

The irradiance is computed on demand by using method `irrandiance` and passing the position of the detector `point`, the direction it is facing to `direction` and the opening angle `angle`, where the latter argument can be omitted in which case the default opening angle of 180° is used.

Since class `Module` is derived from `List<Module>::Item`, class `Light` needs to define method `clone`.

4.2.2 Verification

The relative duration of the eclipse is simulated for circular orbits around the Earth at altitudes h of 100 km, 400 km, 2,000 km and 5,000 km in dependence of different inclination angles γ where $\gamma = 0°$ corresponds to a sun-synchronous orbit. The mean anomaly M corresponding to the true anomaly for circular orbits is defined as $M = 0°$ when the spacecraft is passing the ecliptic. The following condition is met when the spacecraft is entering or leaving eclipse

$$\left(r_\mathrm{e} + h\right)^2 \left(\cos^2 M \, \cos^2 \gamma + \sin^2 M\right) = r_\mathrm{e}^2 \tag{85}$$

with the mean Earth radius r_e. The equation is solved numerically and the results are compared with CubeSim simulations (see fig. 4.3). The model of a point light source is used for module `Light`.

For the next verification, the two different models for module `Light` are compared. With an altitude of $a = 5,000\,\mathrm{km}$ and an inclination of $\gamma = 60°$, the point in time when the spacecraft is leaving eclipse is considered in detail. In fig. 4.4, the irradiance using the point light source model is shown in black and one recognizes the immediate change from 0 to $1365.86\,\mathrm{W/m^2}$ [77, p. 5]. In contrast, the disk light source

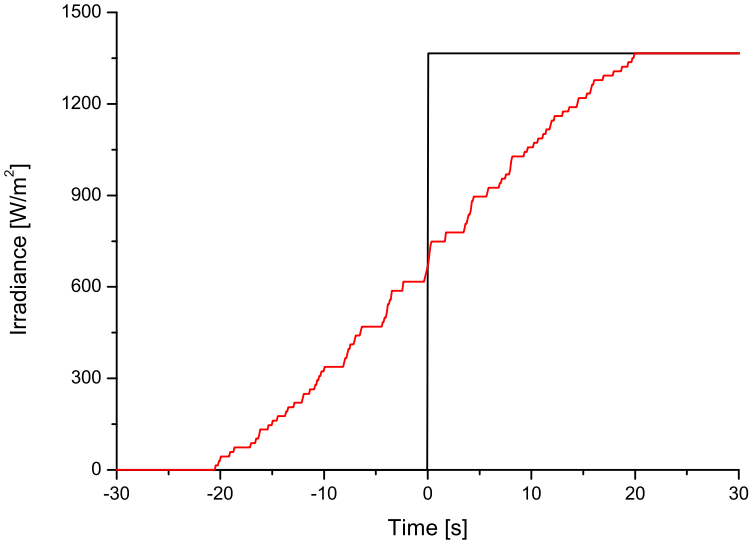

Fig. 4.4. Comparison between point light source (black) and disk light source model (red) when the spacecraft is leaving eclipse.

model results are shown in red and the steady increase of irradiance over the interval of ca. 40 seconds can be observed. The steps result from the defined resolution which was left at the predefined value of 100 points.

4.3 Albedo

The Part of the incident light from a star being reflected by a celestial body is referred to as the planetary albedo [224, p. 1]. Though the reflectivity is dependent on the wavelength of the light as shown in [139, p. 23], the implementation of module `Albedo` only assumes visible light and a frequency independent reflectivity to increase computational performance (see also section 4.3.3) as the expected error for Earth is less than 18% if UV (ultraviolet) light is excluded.

Similar to module `Light` and its disk model, a certain number of points (resolution) is evenly distributed on the spherical surface of the celestial body (see fig. 3.2). In contrast to a star with a spherical shape, the reflecting surface element in this model is considered flat. The light emitted from a hemispherical light source with area A_{sphere} is distributed uniformly over a hemispherical surface. The radiant flux Φ representing the integration of the source irradiance $\hat{E}_{\text{sphere}}$ over the light source area A_{sphere} must be equal to the integral of the irradiance $E_{\text{sphere}}(r)$ over a hemisphere with radius r. In spherical coordinates the radiant flux is computed as

$$\Phi = \int_{A_{\text{sphere}}} \mathrm{d}A\, \hat{E}_{\text{sphere}} = \int_0^{\pi/2} \mathrm{d}\phi \int_0^{2\pi} \mathrm{d}\theta\, E_{\text{sphere}}(r)\, r^2 \cos\phi \tag{86}$$

with the irradiance at distance r following the inverse square law [247, p. 23]

$$E_{\text{sphere}}(r) = \frac{\Phi}{2\pi r^2} \tag{87}$$

and the longitude θ, the latitude ϕ and the Jacobi determinant $r^2 \cos\phi$ [13, p. 8]. In contrast, if a flat surface with area A_{flat} and source irradiance $\hat{E}_{\text{flat}}$ is considered, the irradiance $E_{\text{flat}}(r, \phi)$ on the hemisphere with radius r is dependent on the latitude ϕ. The radiant flux equals to

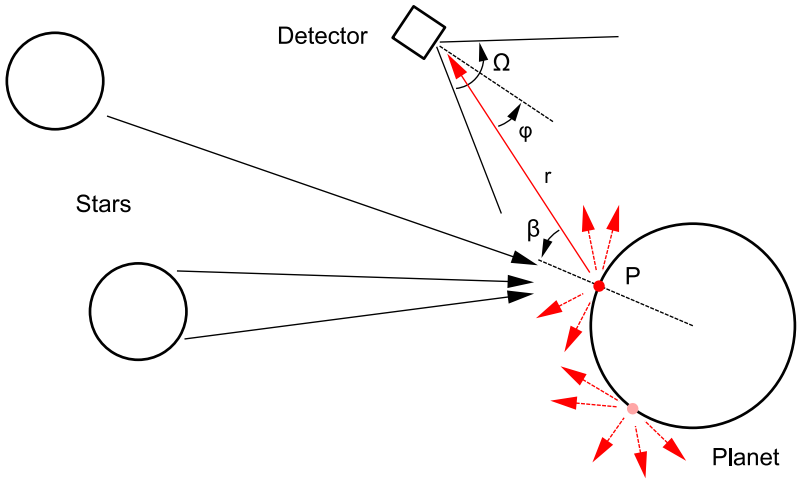

Fig. 4.5. Light from stars accumulated at red dot and diffusely reflected in all directions. Detector with an opening angle θ is irradiated.

$$\Phi = \int_{A_{\text{flat}}} \mathrm{d}A\, \hat{E}_{\text{flat}} = \int_0^{\pi/2} \mathrm{d}\phi \int_0^{2\pi} \mathrm{d}\theta\, E_{\text{flat}}(r,\phi)\, r^2 \cos\phi \tag{88}$$

with the irradiance

$$E_{\text{flat}}(r,\phi) = \frac{\Phi}{\pi r^2} \sin\phi. \tag{89}$$

One recognizes that due to the normalization of the irradiance to match the radiant flux, the factor $\frac{1}{2}$ is not required in $E_{\text{flat}}(r,\phi)$ compared to $E_{\text{sphere}}(r)$ which concludes that at $\phi = \frac{\pi}{2}$ the irradiance from a flat surface is two times higher than from a hemisphere with equal radiant flux.

The albedo irradiance is computed by the use of a virtual photo detector at an arbitrary location, facing an arbitrary direction and featuring a certain opening angle (see fig. 4.5). To compute the albedo irradiance contribution from a certain point $\boldsymbol{P}$ (red dot), the incoming irradiance is accumulated by parsing all modules of type `Light` in the simulation. The diffusely reflected light spreads in all directions in accordance with eq. (89). Under consideration of the cosine losses due to angles β and φ, the irradiance at the photo detector results in

$$E_{\text{albedo}}(r,\beta,\varphi,R) = \begin{cases} \frac{E_{\text{source}}}{\pi r^2} R(\boldsymbol{P}) \cos\beta \cos\varphi & \beta < \frac{\pi}{2},\ \varphi < \frac{\Omega}{2} \\ 0 & \text{otherwise} \end{cases} \tag{90}$$

with the incoming irradiance E_{source} at point $\boldsymbol{P}$ and its local relative reflectivity $R(\boldsymbol{P})$. Eq. (90) is computed for each point on the surface of the celestial bodies. There are points on the surface that are illuminated by stars (light red dot in fig. 4.5) but if the angle $\beta > \frac{\pi}{2}$, no reflected light shines on the photo detector.

4.3.1 The Earth Albedo

Though module `Albedo` can be used for different celestial bodies, the focus is laid on the Earth in this section. Different albedo values are found in literature but in average 29 [224, p. 1] to 43.4% [139, p. 23] of the incoming solar energy on Earth is scattered back into space. The reflectivity not only depends on the type of terrain but also on the weather conditions. While snow has a reflectivity between 45 and

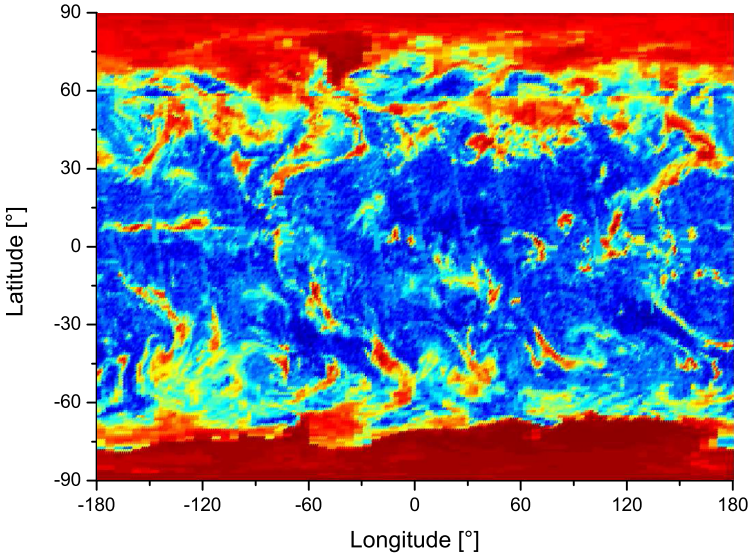

Fig. 4.6. Typical reflectivity data of the Earth surface [237, p. 3].

90% and grass between 18 and 32%, asphalt has a reflectivity as low as 15%. The value of water strongly depends on the angle of incident of sun light and can vary between 5 and 22% [142, p. 60].

The NASA TOMS (Total Ozone Mapping Spectrometer) was a spectrometer for measuring the properties of the Earth ozone layer [143]. For that purpose a daily mapping of the reflectivity of the Earth surface was computed as shown in fig. 4.6. It can be recognized that the reflectivity mainly depends on the latitude and is almost independent from the longitude.

In [237, p. 4] reflectivity data of fig. 4.6 was used to compute the average reflectivity for a certain latitude φ (see fig. 4.7, black points). The following fitting function was created

$$R(\varphi) = \begin{cases} 0.183722 + 0.031123\,\varphi + 0.332548\,\varphi^2 - 0.035054\,\varphi^3 & -1.497 < \varphi \\ 1 & \text{otherwise} \end{cases} \tag{91}$$

which is also shown in the figure as black curve where φ denotes the latitude in rad. The virtual method `reflectivity` of class `CelestialBody::Earth` is using eq. (91) to compute the reflectivity (see also chapter 11).

4.3.2 Implementation

The class `Albedo` allows to compute the albedo irradiance at a certain location where the detector is facing an arbitrary direction and features a defined opening angle. For that purpose, the class uses all modules of type `Light` in the simulation and computes the reflection from a single celestial body or all bodies in the simulation. `Albedo` is derived from class `Module` and defined as follows excluding certain private declarations.

```
// Class Albedo
class CubeSim::Module::Albedo : public Module
{
public:

    // Constructor
    Albedo(uint32_t resolution = _RESOLUTION);
```

```
    Albedo(CelestialBody& celestial_body, uint32_t resolution =
        _RESOLUTION);

    // Specific Celestial Body
    const CelestialBody* celestial_body(void) const;
    void celestial_body(const CelestialBody* celestial_body);

    // Clone
    virtual Module* clone(void) const;

    // Compute Irradiance [W/m^2]
    double irradiance(const Vector3D& point, const Vector3D& direction,
        double angle = Constant::PI) const;

    // Resolution (Number of Grid Points)
    uint32_t resolution(void) const;
    void resolution(uint32_t resolution);

private:

    // Default Resolution (Number of Grid Points)
    static const uint32_t _RESOLUTION = 400;
};
```

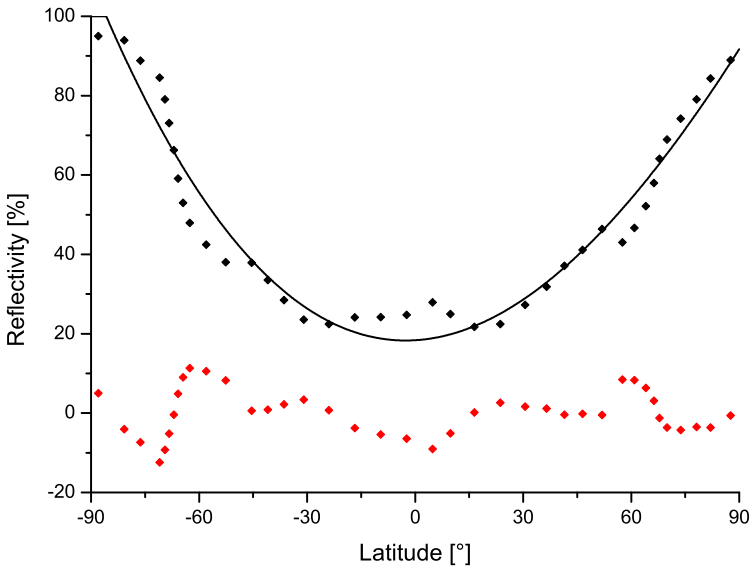

Fig. 4.7. Reflectivity in dependence of the latitude (black points) [237, p. 4], fitting function (black curve) and deviation (red symbols).

The class features different constructors. Using `Albedo(uint32_t resolution = _RESOLUTION)` allows to define the resolution representing the number of points on the sphere covering the celestial body. If parameter `resolution` is omitted, the default resolution (400 points) is used. If only a certain celestial body shall be used to compute the albedo irradiance with this instance of `Albedo`, constructor `Albedo(CelestialBody& celestial_body, uint32_t resolution = _RESOLUTION)` can be used which offers the additional parameter `celestial_body`.

Method `resolution` allows to modify the number of points on the sphere by passing the new number. If the method is called without arguments, it returns the current resolution.

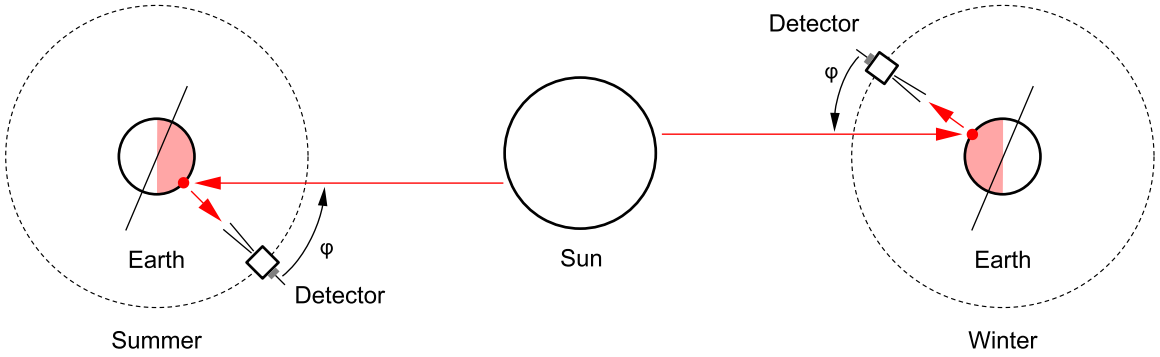

Fig. 4.8. Measurement of the Earth albedo on a polar orbit with a narrow-angle photo detector during summer solstice (left) and winter solstice (right).

The celestial body can also be retrieved by the use of method `celestial_body` when no arguments are passed. It is modified by passing an object of type `CelestialBody` to the method.

The albedo irradiance is computed on demand by using method `irrandiance` and passing the position of the detector `point`, the direction `direction` the detector is facing to and its opening angle `angle`, where the latter argument can be omitted in which case the default opening angle of 180° is used. For the computation, all modules of type `Light` and the reflectivity value obtained from method `reflectivity` of class `CelestialBody` are considered.

Since class `Module` is derived from `List<Module>::Item`, class `Albedo` needs to define method `clone`.

4.3.3 Verification

A satellite is put on a stable LEO orbit at 400 km altitude where the orbital plane is perpendicular to the x-axis of the celestial coordinate system (see section 5.3). The spin rate is set to $\boldsymbol{\omega}_0 = (0 \;\; 2\pi/T \;\; 0)^{\mathrm{T}}$ rad/s with the orbital period $T = 5,543.8\,\mathrm{s}$. This defines the so-called *Y-Thomson spin* where the z-axis of the spacecraft is along direction of flight and the x-axis is pointing to Nadir (Earth center) [114, p. 5]. The spin rate around the y-axis is chosen so that one full rotation matches the orbit period [239], [227, p. 3].

As shown in fig. 4.8, the spacecraft is equipped with a narrow-angle photo detector of 10° opening angle which is always facing the Earth and a 180° wide-angle photo detector pointing away from Earth (small gray box on top of the spacecraft). At the low altitude, the narrow-angle sensor allows a detailed mapping of the Earth surface (disk with ca. 70 km diameter).

This configuration allows the measurement of the incident solar radiation and the Earth albedo. From the ratio, the local reflectivity of the Earth surface can be computed as shown in fig. 4.9. The measured data points are in good accordance with the underlying Earth reflectivity model as defined in eq. (91). In order to allow coverage of the entire latitude range, the simulation was performed two times, on 6 June 2000 03:48 (summer solstice) and on 21 December 2000 14:37 (winter solstice). It can be clearly seen from the results that a single scan only allows a latitude range of ca. 150° due to the Earth axis tilt.

Due to the small field of view of the photo detector facing the Earth, the resolution of module `Albedo` was set to 10^7 resulting in comparatively long simulation duration of several minutes for one orbital period of the spacecraft. This way, the sensor is accumulating albedo irradiance from 75 points in average resulting in low-noise measurement results. Due to the orbital eccentricity of the Earth-Moon barycenter around the solar system barycenter of ca. $e = 0.01671$ (see table 4.3), the distance between Sun and Earth varies between the perihelion $r_{\mathrm{per}} = (1-e)\,a = 1.471 \times 10^{11}\,\mathrm{m}$ and the aphelion $r_{\mathrm{aph}} = (1+e)\,a = 1.521 \times 10^{11}\,\mathrm{m}$ with the semi-major axis $a = 1.496 \times 10^{11}\,\mathrm{m}$. Since the resulting solar flux following the inverse square

law, it consequently varies by ±3.31% over the year. The second wide-angle photo detector is used to precisely acquire the actual solar flux and allows more accurate measurements.

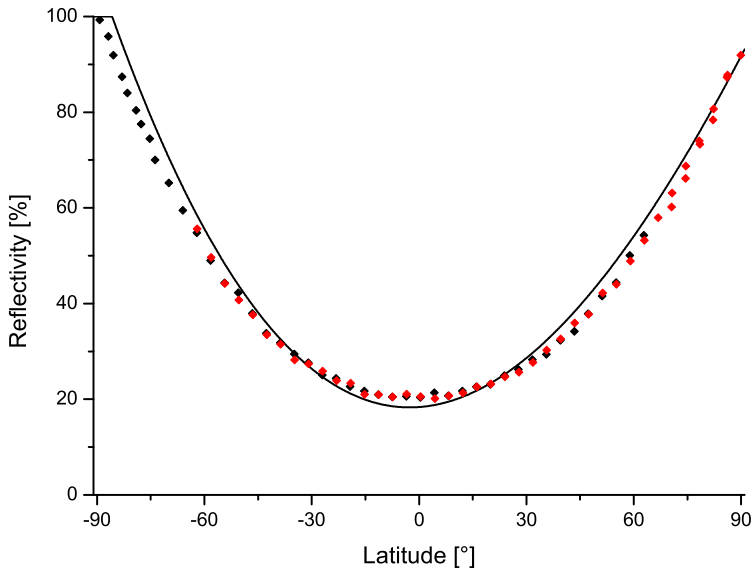

Fig. 4.9. Computed local reflectivity of the Earth surface from measurements during winter solstice (black) and summer solstice (red).

4.4 Magnetics

Module `Magnetics` is used to compute the magnetic field, more precisely the magnetic flux density, at any location at any point in time. For satellites, the magnetic field allows on the one hand to determine the attitude of the spacecraft (see section 9.5) and on the other hand allows the generation of torques by the use of magnetorquers (see section 9.6). In the solar system, apart from the Earth and Jupiter, the planets do not have a significant magnetic field [226, p. 563]. Therefore, magnetometers and magnetorquers can only be used in low altitude orbits around the Earth and Jupiter. As the focus of CubeSim is laid on the mission simulation of small satellites, orbits around Jupiter are currently not considered and only the magnetic field of the Earth is used in module `Magnetics`. Nevertheless, the module is designed to consider the magnetic flux density from all celestial bodies being used in the simulation and to compute the vector sum at a specific location.

4.4.1 International Geomagnetic Reference Field

The IGRF (International Geomagnetic Reference Field) is a mathematical model describing the Earth's magnetic field between epochs 1900 A.D. and the present [61]. The actual version of the IGRF model is IGRF-12 being valid from year 1900 to 2020, was released in 2015. In addition to the geomagnetic field, the model also describes the annual rate of change (secular variation). The magnetic flux density $\boldsymbol{B}$ is defined as the gradient of the magnetic scalar potential V

$$\boldsymbol{B}\left(r, \phi, \bar{\theta}, t\right) = -\nabla V\left(r, \phi, \bar{\theta}, t\right) \tag{92}$$

with the radial distance from the Earth center r, the east longitude ϕ, the co-latitude $\bar{\theta}$ and t representing the elapsed time in years after the epoch T_0 which is 1 January 2015 for IGRF-12.

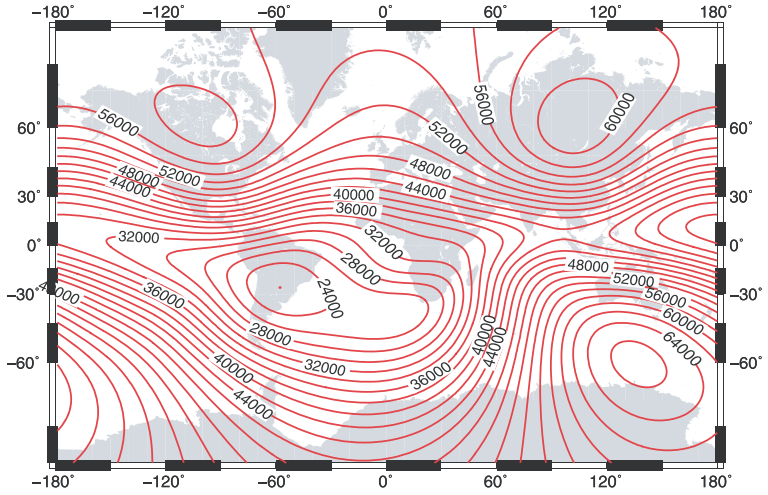

Fig. 4.10. Magnetic field intensity in nT at the Earth mean radius in 2015 [61, p. 13].

The magnetic scalar potential model consists of the spherical harmonic coefficients and their rates of change. This allows to express V as follows

$$V\left(\mathbf{r},\phi,\bar{\theta},t\right)=a\sum_{n=1}^{N}\sum_{m=0}^{n}\left(\frac{a}{r}\right)^{n+1}\left(g_n^m\left(t\right)\cos\left(m\phi\right)+h_n^m\left(t\right)\sin\left(m\phi\right)P_n^m\left(\cos\bar{\theta}\right)\right) \tag{93}$$

with the Earth mean radius $a = 6371.2\,\mathrm{km}$, the aforementioned Gauss coefficients g_n^m and h_n^m and the Schmidt quasi-normalized associated Legendre functions of degree n and order m P_n^m. The variation of the Gauss coefficients in time is considered as follows

$$g_n^m\left(t\right)=g_n^m\left(T_0\right)+\dot{g}_n^m\left(T_0\right)\left(t-T_0\right),\quad h_n^m\left(t\right)=h_n^m\left(T_0\right)+\dot{h}_n^m\left(T_0\right)\left(t-T_0\right) \tag{94}$$

with $\dot{g}_n^m$ and $\dot{h}_n^m$ specified in nT/yr.

At the begin of the simulation, the start time is used to compute the valid values of g_n^m and h_n^m. As the rate of change is low, the coefficients are not recalculated for each magnetic field computation. The coefficients are automatically recomputed after 1 day of simulated real-time which is beneficial for the computational performance.

For fast computation of the magnetic flux density at a certain point $\boldsymbol{r} = \left(x\ \ y\ \ z\right)^{\mathrm{T}}$, the longitude ϕ and - in contrast to the IGRF model - the latitude θ are computed first

$$r=\sqrt{x^2+y^2+z^2},\quad \theta=\arccos\frac{z}{r},\quad \phi=\operatorname{arctan2}\left(\mathrm{y},\mathrm{x}\right). \tag{95}$$

Then eq. (93) is used to compute the magnetic potential $V\left(\boldsymbol{r},t\right)$ at location $\boldsymbol{r}$. The magnetic potential $V\left(\boldsymbol{r}+\triangle\boldsymbol{r},t\right)$ is also computed in the near vicinity of the location $\boldsymbol{r}+\triangle\boldsymbol{r}$ with the difference $\triangle\boldsymbol{r}$ representing a displacement of 100 m in all components. This allows computing the gradient via the difference

$$\boldsymbol{B}=\frac{V\left(\boldsymbol{r},t\right)-V\left(\boldsymbol{r}+\triangle\boldsymbol{r},t\right)}{\triangle\boldsymbol{r}}. \tag{96}$$

Fig. 4.10 shows the magnetic field intensity in 2015 at the Earth mean radius. One can clearly recognize the South Atlantic Anomaly where the magnetic flux density drops by more than 40% [86, p. 245].

4.4.2 Implementation

The class `Magnetics` uses the individual magnetic fields of the celestial bodies and computes the resulting magnetic field (flux density) at an arbitrary location. It is derived from class `Module` and defined as follows excluding certain private declarations.

```
// Class Magnetics
class CubeSim::Module::Magnetics : public Module
{
public:

   // Constructor
   Magnetics(void);
   Magnetics(CelestialBody& celestial_body);

   // Specific Celestial Body
   const CelestialBody* celestial_body(void) const;
   void celestial_body(const CelestialBody* celestial_body);

   // Clone
   virtual Module* clone(void) const;

   // Compute magnetic Field [T]
   const Vector3D field(const Vector3D& point) const;

private:

   // Compute magnetic Field [T]
   const Vector3D _field(const CelestialBody& celestial_body, const
      Vector3D& point) const;
};
```

The class defines the default constructor `Magnetics(void)` and the additional constructor `Magnetics(CelestialBody& celestial_body)`. With the latter one, the computation of the magnetic field is restricted to a specific celestial body. In most simulation scenarios, only the Earth is used. This allows to increase the performance of the simulation since it removes the necessity of iterating through all celestial bodies in the simulation though these do not contribute to the resultant magnetic field at the location of interest.

The celestial body can be retrieved by the use of method `celestial_body` when no arguments are passed. It is modified by passing an object of type `CelestialBody` to the method.

The computation of the magnetic field is done by method `field` which expects the point of interest in the global coordinate system as parameter. If no specific celestial body was specified, the method is iterating over all celestial bodies in the simulation, computing their contribution to the magnetic flux density (see method `_field`) and summing up the resulting vectors. Finally, the vector sum, corresponding to the magnetic field at the requested location, is returned. For each celestial body, the point is transformed from global coordinates into the body frame of the celestial body which allows a more efficient modeling of the magnetic field, since the models - such as the IGRF model for the Earth - are commonly defined in the body frame of the planet. The point in the body frame is passed to method `magnetic_field` defined in class `CelestialBody` and the result is re-transformed to consider the orientation of the celestial body with respect to the global coordinate system (see chapter 11 for details).

Since class `Module` is derived from `List<Module>::Item`, class `Magnetics` needs to define method `clone`.

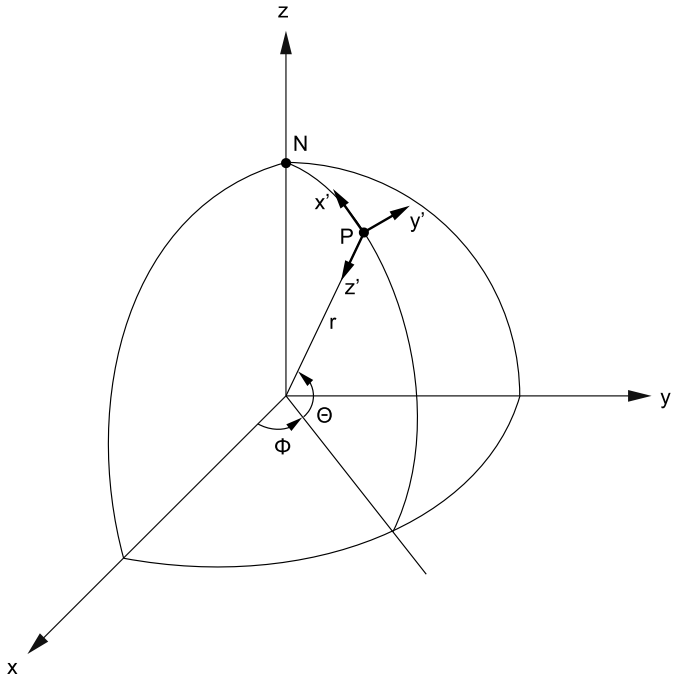

Fig. 4.11. Orthogonal spherical coordinate system used in the IGRF magnetic field model.

4.4.3 Verification

In fig. 4.10 the intensity of the Earth magnetic field at the mean radius is shown for 1 January 2015. The National Centers for Environmental Information NOAA provide the Geomag software which uses the IGRF coefficients to compute the components of the Earth magnetic field and the total intensity [167].

Instead of B_x, B_y and B_z components of the Earth magnetic field $\boldsymbol{B}$ in the body frame, vectors $\boldsymbol{x}'$, $\boldsymbol{y}'$ and $\boldsymbol{z}'$ are spanning an orthogonal coordinate system at the point $\boldsymbol{P}$ aligned to the surface of the Earth. The flattening of the Earth is not considered and a mean radius of $r = 6,371.009\,\mathrm{km}$ is assumed instead [128, p. 5]. $\boldsymbol{x}'$ and $\boldsymbol{y}'$ are spanning a tangential plane where $\boldsymbol{x}'$ is pointing to the North, $\boldsymbol{y}'$ to the east and $\boldsymbol{z}'$ towards the Earth center (see fig. 4.11).

For verification of the CubeSim magnetic field computation and the comparison with the output of the Geomag software, the transformation from the orthogonal coordinate system $(\boldsymbol{x}', \boldsymbol{y}', \boldsymbol{z}')$ to the body frame needs to be performed. The base vectors can be expressed as follows

$$\boldsymbol{x}' = \begin{pmatrix} -\sin\theta\cos\phi \\ -\sin\theta\sin\phi \\ -\cos\theta \end{pmatrix}, \quad \boldsymbol{y}' = \begin{pmatrix} -\sin\phi \\ \cos\phi \\ 0 \end{pmatrix}, \quad \boldsymbol{z}' = \begin{pmatrix} -\cos\theta\cos\phi \\ -\cos\theta\sin\phi \\ -\sin\theta \end{pmatrix}. \tag{97}$$

The rotation matrix being used for the transformation of the vector components is computed in a similar manner as done in [201, p. 35]

$$\boldsymbol{R} = \begin{pmatrix} \boldsymbol{x}\,\boldsymbol{x}' & \boldsymbol{x}\,\boldsymbol{y}' & \boldsymbol{x}\,\boldsymbol{z}' \\ \boldsymbol{y}\,\boldsymbol{x}' & \boldsymbol{y}\,\boldsymbol{y}' & \boldsymbol{y}\,\boldsymbol{z}' \\ \boldsymbol{z}\,\boldsymbol{x}' & \boldsymbol{z}\,\boldsymbol{y}' & \boldsymbol{z}\,\boldsymbol{z}' \end{pmatrix} = \begin{pmatrix} -\sin\theta\cos\phi & -\sin\phi & -\cos\theta\cos\phi \\ -\sin\theta\sin\phi & \cos\phi & -\cos\theta\sin\phi \\ -\cos\theta & 0 & -\sin\theta \end{pmatrix} \tag{98}$$

with the unit vectors $\boldsymbol{x}$, $\boldsymbol{y}$ and $\boldsymbol{z}$. The latitude θ and longitude ϕ depend on the observer point $\boldsymbol{P}$ as defined in eq. (95). The components of the magnetic field are then computed $\boldsymbol{B} = \boldsymbol{R}\,\boldsymbol{B}'$.

Table 4.1. Coordinates in geodetic and ECEF representation (units in years, m and deg).

Date	$\boldsymbol{\theta}$	$\boldsymbol{\phi}$	$\boldsymbol{a}$	$\boldsymbol{x}$	$\boldsymbol{y}$	$\boldsymbol{z}$
2019.5	70.3	30.8	1.000×10^5	1.881257×10^6	1.121454×10^6	6.076552×10^6
2018.7	−70.3	−30.8	9.144×10^3	1.854949×10^6	-1.105772×10^6	-5.991014×10^6
2016.5	70.3	30.8	1.042×10^3	1.852603×10^6	1.104373×10^6	5.983386×10^6
2019.3	48.2	16.4	3.000×10^5	4.277701×10^6	1.258996×10^6	4.955371×10^6

Table 4.2. Comparison of computed magnetic flux densities and the resulting deviation (units in µT).

$\boldsymbol{B_x}$	$\boldsymbol{B_y}$	$\boldsymbol{B_z}$	$\hat{\boldsymbol{B}}_{\boldsymbol{x}}$	$\hat{\boldsymbol{B}}_{\boldsymbol{y}}$	$\hat{\boldsymbol{B}}_{\boldsymbol{z}}$	$\triangle \boldsymbol{B}$
−23.864	−11.218	−44.973	−23.945	−11.266	−44.919	0.109
24.715	−15.065	−24.802	24.761	−15.092	−24.740	0.082
−24.904	−11.650	−46.788	−24.990	−11.701	−46.731	0.115
−38.017	−9.908	−16.136	−38.067	−9.922	−16.011	0.135

For the verification of module `Magnetics` four locations were chosen as listed in tbl. 4.1. Also the date was varied between 2015 and 2020 to verify the secular variation. From the latitude θ, longitude ϕ and the altitude a the coordinates x, y and z in the ECEF (Earth-centered, Earth-fixed) coordinate system were calculated considering the WGS84 ellipsoid. Details on the computation can be found in section 9.1.

For the given locations, the magnetic field was computed with module `Magnetics` (see tbl. 4.2, columns B_x, B_y and B_z) and with the Geomag software where in the latter case, the transformation of the results was done with the rotation matrix $\boldsymbol{R}$ defined in eq. (98) - see tbl. 4.2, columns $\hat{B}_x$, $\hat{B}_y$ and $\hat{B}_z$. The deviation $\triangle B = \left\| \boldsymbol{B} - \hat{\boldsymbol{B}} \right\|$ between both magnetic flux density vectors is also computed and one can recognize the good accordance as the maximum deviation is as low as 135 nT.

4.5 Motion

Module `Motion` is updating the position and orientation of free rigid bodies (either celestial bodies or spacecraft) at certain predefined intervals (so-called *time steps*). The motion of a rigid body is composed of the translation of its center of mass and the rotation of the body around its center of mass [52, p. 130]. Therefore, the translation and the rotation can be treated independently in the following two subsections.

4.5.1 Translation

Newton was the first one who figured out that *every body continuous in its state of rest, or of uniform motion in a straight line, unless compelled to change that state by forces acting upon it* (first law) [85, p. 7]. When an external force is applied *the time rate of change of linear momentum of a body is proportional to the force acting upon it and occurs in the direction in which the force acts* (second law). Considering a system of particles m_i, the center of mass $\bar{\boldsymbol{r}}$ is defined with the equation

$$\sum m_i \left(\boldsymbol{r}_i - \bar{\boldsymbol{r}} \right) = \boldsymbol{0}. \tag{99}$$

This leads to

$$\sum m_i \boldsymbol{r}_i = \sum m_i \bar{\boldsymbol{r}} = M \bar{\boldsymbol{r}} \tag{100}$$

with the total mass $M = \sum m_i$. Differentiating after time and under the assumption of constant particle masses m_i results in

$$\sum m_i \boldsymbol{v}_i = M\bar{\boldsymbol{v}} = \bar{\boldsymbol{p}} \tag{101}$$

with the total linear momentum $\bar{\boldsymbol{p}}$. The second law can be formalized as

$$\boldsymbol{F} = \frac{\mathrm{d}\bar{\boldsymbol{p}}}{\mathrm{d}t} \tag{102}$$

with an external force $\boldsymbol{F}$ acting on the system of particles. With the absence of an external force, $\bar{\boldsymbol{p}}$ is constant and consequently also $\bar{\boldsymbol{v}}$ is constant. In other words, the center of mass is either at rest or performs a uniform motion as stated in Newton's first law for a single particle.

The simulation framework uses discrete time steps to propagate the state vectors of all rigid bodies. In the case of a spacecraft, if a system or module modifies the position of a part or assembly, the center of mass of the spacecraft is changed. This would contradict Newton's first law as no external force is considered for the moment. Assuming that the position of a part with mass m_i is changed from $\boldsymbol{r}_i$ to $\boldsymbol{r}_i + \triangle\boldsymbol{r}_i$ within the last time step, the change of the center of mass is

$$\triangle\bar{\boldsymbol{r}} = \frac{1}{M}\, m_i \triangle\boldsymbol{r}_i. \tag{103}$$

In order to correct this discrepancy, Module `Motion` would need to correct the position of the spacecraft by $-\triangle\bar{\boldsymbol{r}}$. Under the assumption of typical small spacecraft dimensions of less than 1 m and that the moving part weighs less than 10% of the spacecraft, the center of mass displacement would be in the order of $\sim 10\,\mathrm{cm}$. This can be considered as negligible in most simulation scenarios and therefore, the position of the spacecraft is not corrected in every time step to account for internal shifts of assemblies or parts to save computational power.

Similarly, assuming that the velocity of an internal part with mass m_i is changed from $\boldsymbol{v}_i$ to $\boldsymbol{v}_i + \triangle\boldsymbol{v}$ within the last time step, the change of the linear momentum is

$$\triangle\bar{\boldsymbol{v}} = \frac{1}{M}\, m_i \triangle\boldsymbol{v}_i. \tag{104}$$

This would contradict Newton's second law as still no external force is considered. In order to correct this discrepancy, Module `Motion` would need to correct the velocity of the spacecraft by $-\triangle\bar{\boldsymbol{v}}$. Under the previously taken assumption and further assuming a relative velocity of typically no more than 1 m/s, which would be rather fast, e.g., for a linear stage, the change in spacecraft velocity would be in the order of $\sim 10\,\mathrm{cm/s}$. Comparing this to typical orbital velocities of $\sim 7\,\mathrm{km/s}$ in LEO and the fact that the assumed movement cannot last longer than 1 s (otherwise the part would leave the spacecraft's envelope), one can consider this (even transient) discrepancy as negligible in most simulation scenarios and therefore, the velocity of the spacecraft is not corrected in every time step to account for internal movements of assemblies or parts to save computational power.

In the previous considerations, no external forces acting on the celestial body or spacecraft were considered. In order to simulate forces, e.g., caused by gravitation, eq. (102) is used

$$\boldsymbol{F} = \frac{\mathrm{d}\boldsymbol{p}}{\mathrm{d}t} = M\frac{\mathrm{d}\boldsymbol{v}}{\mathrm{d}t} = M\boldsymbol{a} \tag{105}$$

with the acceleration $\boldsymbol{a}$. Since it was shown that the simulation does not need to distinct between the center of mass of the spacecraft and the spacecraft state vectors, the bar notation can be omitted. Module `Motion` takes into account all acting forces and torques on the rigid body and numerically propagates the Cartesian state vectors $\boldsymbol{x}$ and $\boldsymbol{v}$ from the previous time $t = -\triangle t$ to the current time $t = 0$

$$\boldsymbol{x}\left(-\triangle t\right) \rightarrow \boldsymbol{x}\left(0\right),\ \boldsymbol{v}\left(-\triangle t\right) \rightarrow \boldsymbol{v}\left(0\right). \tag{106}$$

In the upcoming equations the following formalism will be used

$$\boldsymbol{x}_i = \boldsymbol{x}\left(-i\triangle t\right),\ \boldsymbol{v}_i = \boldsymbol{v}\left(-i\triangle t\right),\ \boldsymbol{a}_i = \boldsymbol{a}\left(-i\triangle t\right) \tag{107}$$

with the acceleration $\boldsymbol{a}\left(t\right)$. At $t = -\triangle t$ the vectors $\boldsymbol{x}_1$, $\boldsymbol{v}_1$ and $\boldsymbol{a}_1$ are known and for the computation of $\boldsymbol{x}_0$ and $\boldsymbol{v}_0$ the acceleration $\boldsymbol{a}\left(t\right)$ needs to be extrapolated in the time frame $-\triangle t \leqslant t \leqslant 0$. This allows the straight forward computation of the velocity vector by integration

$$\boldsymbol{v}\left(t\right) = \boldsymbol{v}_1 + \int_{-\triangle t}^{t} \boldsymbol{a}\left(t'\right) \mathrm{d}t' \tag{108}$$

and consequently the position vector with

$$\boldsymbol{x}\left(t\right) = \boldsymbol{x}_1 + \int_{-\triangle t}^{t} \boldsymbol{v}\left(t'\right) \mathrm{d}t'. \tag{109}$$

There are different approaches how $\boldsymbol{a}\left(t\right)$ can be extrapolated. The most trivial approach is to assume that the acceleration is constant within the last time step $\boldsymbol{a}\left(t\right) = \boldsymbol{a}_0 = \boldsymbol{a}_1$. Applying eqs. (108) and (109), the updated state vectors are obtained

$$\boldsymbol{v}_0 = \boldsymbol{v}\left(0\right) = \boldsymbol{v}_1 + \int_{-\triangle t}^{0} \boldsymbol{a}\left(t'\right) \mathrm{d}t' = \boldsymbol{v}_1 + \boldsymbol{a}_1\triangle t \tag{110}$$

and

$$\boldsymbol{x}_0 = \boldsymbol{x}\left(0\right) = \boldsymbol{x}_1 + \int_{-\triangle t}^{0} \boldsymbol{v}\left(t'\right) \mathrm{d}t' = \boldsymbol{x}_1 + \boldsymbol{v}_1\triangle t + \frac{1}{2}\boldsymbol{a}_1\triangle t^2. \tag{111}$$

It is obvious that the assumption of having constant acceleration over the entire time step requires very small time steps to lead to acceptable results. In order to speed up the entire simulation framework, it is desirable though to increase the time steps.

To increase the accuracy, it is possible to store the acceleration two time steps ago $\boldsymbol{a}_2$. Following the slope from $\boldsymbol{a}_2$ to $\boldsymbol{a}_1$ allows better prediction of $\boldsymbol{a}_0$ and the values in between $\boldsymbol{a}\left(t\right)$ with $-\triangle t \leqslant t \leqslant 0$. Generalized, a first order polynomial function is defined

$$\boldsymbol{a}\left(t\right) = \sum_{i=0}^{1} \boldsymbol{k}_i\, t^i = \boldsymbol{k}_0 + \boldsymbol{k}_1\, t \tag{112}$$

which fulfills the following constraints

$$\boldsymbol{a}\left(-2\triangle t\right) = \boldsymbol{a}_2,\ \boldsymbol{a}\left(-\triangle t\right) = \boldsymbol{a}_1. \tag{113}$$

Solving the set of equations results in the coefficients $\boldsymbol{k}_0$ and $\boldsymbol{k}_1$ and the acceleration

$$\boldsymbol{a}\left(t\right) = 2\boldsymbol{a}_1 - \boldsymbol{a}_2 + \frac{1}{\triangle t}\left(\boldsymbol{a}_2 - \boldsymbol{a}_1\right)t. \tag{114}$$

Applying eqs. (108) and (109), the updated state vectors are obtained

$$\boldsymbol{v}_0 = \boldsymbol{v}_1 + \int_{-\triangle t}^{0} \boldsymbol{a}\left(t'\right) \mathrm{d}t' = \boldsymbol{v}_1 + \frac{1}{2} \triangle t \left(3\boldsymbol{a}_1 - \boldsymbol{a}_2\right) \tag{115}$$

and

$$\boldsymbol{x}_0 = \boldsymbol{x}_1 + \int_{-\triangle t}^{0} \boldsymbol{v}\left(t'\right) \mathrm{d}t' = \boldsymbol{x}_1 + \boldsymbol{v}_1 \triangle t + \frac{1}{6} \left(4\boldsymbol{a}_1 - \boldsymbol{a}_2\right) \triangle t^2. \tag{116}$$

Eq. (112) can be extended to a second order polynomial

$$\boldsymbol{a}\left(t\right) = \sum_{i=0}^{2} \boldsymbol{k}_i \, t^i = \boldsymbol{k}_0 + \boldsymbol{k}_1 \, t + \boldsymbol{k}_2 \, t^2 \tag{117}$$

with the constraints

$$\boldsymbol{a}\left(-3\triangle t\right) = \boldsymbol{a}_3, \; \boldsymbol{a}\left(-2\triangle t\right) = \boldsymbol{a}_2, \; \boldsymbol{a}\left(-\triangle t\right) = \boldsymbol{a}_1. \tag{118}$$

Solving the set of equations results in the coefficients $\boldsymbol{k}_0$, $\boldsymbol{k}_1$ and $\boldsymbol{k}_2$and the corresponding acceleration

$$\boldsymbol{a}\left(t\right) = 3\boldsymbol{a}_1 - 3\boldsymbol{a}_2 + \boldsymbol{a}_3 + \frac{1}{2\triangle t} \left(5\boldsymbol{a}_1 - 8\boldsymbol{a}_2 + 3\boldsymbol{a}_3\right) t + \frac{1}{2\triangle t^2} \left(\boldsymbol{a}_1 - 2\boldsymbol{a}_2 + \boldsymbol{a}_3\right) t^2. \tag{119}$$

Applying eqs. (108) and (109), the updated state vectors are obtained

$$\boldsymbol{v}_0 = \boldsymbol{v}_1 + \int_{-\triangle t}^{0} \boldsymbol{a}\left(t'\right) \mathrm{d}t' = \boldsymbol{v}_1 + \frac{1}{12} \left(23\boldsymbol{a}_1 - 16\boldsymbol{a}_2 + 5\boldsymbol{a}_3\right) \triangle t \tag{120}$$

and

$$\boldsymbol{x}_0 = \boldsymbol{x}_1 + \int_{-\triangle t}^{0} \boldsymbol{v}\left(t'\right) \mathrm{d}t' = \boldsymbol{x}_1 + \boldsymbol{v}_1 \triangle t + \frac{1}{24} \left(19\boldsymbol{a}_1 - 10\boldsymbol{a}_2 + 3\boldsymbol{a}_3\right) \triangle t^2. \tag{121}$$

In the last case, eq. (112) is further extended to a third order polynomial

$$\boldsymbol{a}\left(t\right) = \sum_{i=0}^{3} \boldsymbol{k}_i \, t^i = \boldsymbol{k}_0 + \boldsymbol{k}_1 \, t + \boldsymbol{k}_2 \, t^2 + \boldsymbol{k}_3 \, t^3 \tag{122}$$

with the constraints

$$\boldsymbol{a}\left(-4\triangle t\right) = \boldsymbol{a}_4, \; \boldsymbol{a}\left(-3\triangle t\right) = \boldsymbol{a}_3, \; \boldsymbol{a}\left(-2\triangle t\right) = \boldsymbol{a}_2, \; \boldsymbol{a}\left(-\triangle t\right) = \boldsymbol{a}_1. \tag{123}$$

Solving the set of equations results in the coefficients $\boldsymbol{k}_0$, $\boldsymbol{k}_1$, $\boldsymbol{k}_2$ and $\boldsymbol{k}_3$ and the corresponding acceleration

$$\boldsymbol{a}\left(t\right) = 4\boldsymbol{a}_1 - 6\boldsymbol{a}_2 + 4\boldsymbol{a}_3 - \boldsymbol{a}_4 + \frac{1}{6\triangle t} \left(26\boldsymbol{a}_1 - 57\boldsymbol{a}_2 + 42\boldsymbol{a}_3 - 11\boldsymbol{a}_4\right) t + \frac{1}{2\triangle t^2} \left(3\boldsymbol{a}_1 - 8\boldsymbol{a}_2 + 7\boldsymbol{a}_3 - 2\boldsymbol{a}_4\right) t^2. \tag{124}$$

Applying eqs. (108) and (109), the updated state vectors are obtained

$$\boldsymbol{v}_0 = \boldsymbol{v}_1 + \int_{-\triangle t}^{0} \boldsymbol{a}\left(t'\right) \mathrm{d}t' = \boldsymbol{v}_1 + \frac{1}{24} \left(55\boldsymbol{a}_1 - 59\boldsymbol{a}_2 + 37\boldsymbol{a}_3 - 9\boldsymbol{a}_4\right) \triangle t \tag{125}$$

and

$$\boldsymbol{x}_0 = \boldsymbol{x}_1 + \int_{-\triangle t}^{0} \boldsymbol{v}\left(t'\right) \mathrm{d}t' = \boldsymbol{x}_1 + \boldsymbol{v}_1 \triangle t + \frac{1}{360}\left(323\boldsymbol{a}_1 - 264\boldsymbol{a}_2 + 159\boldsymbol{a}_3 - 38\boldsymbol{a}_4\right) \triangle t^2. \tag{126}$$

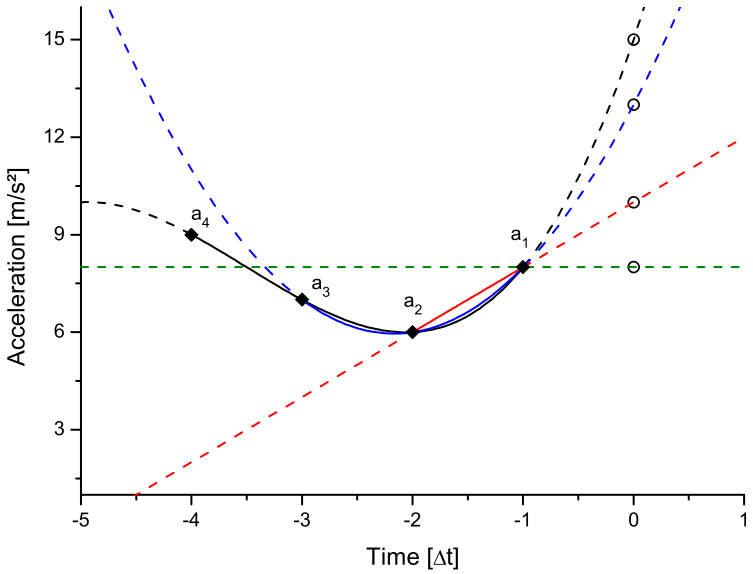

Fig. 4.12. Acceleration prediction with constant value (green), linear extrapolation (red), second order polynomial (blue) and third order polynomial (black).

In fig. 4.12 the comparison between the presented prediction models is shown. On the x-axis the discrete time steps and on the y-axis the acceleration values for different points $\boldsymbol{a}_i$ are shown. The circles show the predicted acceleration $\boldsymbol{a}_0$ at $t = 0$. For the trivial approach, a constant acceleration $\boldsymbol{a}_1$ is assumed in the interval $-\triangle t \leqslant t \leqslant 0$ which leads to the dashed green curve. For the linear prediction, a slope through $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ (solid red curve) is extrapolated (dashed red curve) which results in the circle on the red curve at $t = 0$. For the non-linear prediction model, a second order polynomial is defined by $\boldsymbol{a}_1$,$\boldsymbol{a}_2$ and $\boldsymbol{a}_3$ (solid blue curve). It is extrapolated for $-\triangle t \leqslant t$ and gives the prediction of the acceleration $\boldsymbol{a}_0$ marked with the circle on the blue curve. If the order of the polynomial is further increased, also the fourth historic acceleration value $\boldsymbol{a}_4$ at $t = -4\triangle t$ is considered. A third order polynomial is computed (solid black curve) and again it is extrapolated (dashed black curve) to predict $\boldsymbol{a}_0$ which is marked with a circle on the black curve. One recognizes that the predicted value can significantly depend on the used prediction model. Though in fig. 4.12 extremely fast changing values are used, even with more stable values, small errors sum up at every time step which can lead to significant implications on the simulation results - such as orbit stability or AOCS performance.

4.5.2 Rotation

In eq. (101) the linear momentum $\bar{\boldsymbol{p}}$ of a system of particles with masses m_i and velocities $\boldsymbol{v}_i$ was defined. Similarly, the angular momentum of a particle is defined as

$$\boldsymbol{L}_i = m_i \boldsymbol{r}_i \times \boldsymbol{v}_i = \boldsymbol{r}_i \times \boldsymbol{p}_i. \tag{127}$$

For the entire system, the total angular momentum is the sum of the individual angular momenta

$$\bar{\boldsymbol{L}} = \sum \boldsymbol{L}_i = \sum m_i \boldsymbol{r}_i \times \boldsymbol{v}_i = \sum \boldsymbol{r}_i \times \boldsymbol{p}_i. \tag{128}$$

In section 3.8 it was shown that the so-called *inertia matrix* $\boldsymbol{I}$ can be defined so that the the angular momentum of the system can be expressed as

$$\boldsymbol{L} = \boldsymbol{I}\,\boldsymbol{\omega} \tag{129}$$

with the angular velocity $\boldsymbol{\omega}$.

Deriving $\boldsymbol{L}_i$ according to time, one obtains the torque

$$\frac{\mathrm{d}\boldsymbol{L}_i}{\mathrm{d}t} = \frac{\mathrm{d}\boldsymbol{r}_i}{\mathrm{d}t} \times \boldsymbol{p}_i + \boldsymbol{r}_i \times \frac{\mathrm{d}\boldsymbol{p}_i}{\mathrm{d}t} = \boldsymbol{v}_i \times \boldsymbol{p}_i + \boldsymbol{r}_i \times \boldsymbol{F}_i = \boldsymbol{r}_i \times \boldsymbol{F}_i = \boldsymbol{\tau}_i \tag{130}$$

since $\boldsymbol{v}_i \parallel \boldsymbol{p}_i$. The torque applied to a system of particles changes its angular momentum and can be written as the sum

$$\bar{\boldsymbol{\tau}} = \sum \boldsymbol{r}_i \times \boldsymbol{F}_i = \sum \boldsymbol{\tau}_i \tag{131}$$

and

$$\bar{\boldsymbol{\tau}} = \frac{\mathrm{d}\bar{\boldsymbol{L}}}{\mathrm{d}t}. \tag{132}$$

Similar to eq. (102), it is obvious that the angular momentum of a closed system is constant over time when the applied torque vanishes.

In contrast to the propagation of the position of a rigid body, for the propagation of the orientation and angular velocity, also inner angular momenta $\boldsymbol{L}_i$ need to be considered. These can for example originate from motors or reaction wheels. Eq. (129) can therefore be extended to

$$\bar{\boldsymbol{L}} = \boldsymbol{I}\,\boldsymbol{\omega} + \sum \boldsymbol{L}_i. \tag{133}$$

As explained in section 4.5.1, the simulation framework uses discrete time steps to propagate the state vectors of all rigid bodies. In the case of a spacecraft, if a system or module modifies the position or orientation of a part or assembly, the inertia matrix of the spacecraft is changed. Also if the angular momentum of a part or assembly is changed, the angular momentum of the spacecraft would be different. This would contradict eq. (132) as no external torques are considered for the moment. With the angular momentum $\bar{\boldsymbol{L}}_1 = \bar{\boldsymbol{L}}(-\triangle t)$ for the last time step and $\bar{\boldsymbol{L}}_0 = \bar{\boldsymbol{L}}(0)$ now, the following adjustment is used to conserve the angular momentum

$$\boldsymbol{\omega}_0 = \boldsymbol{\omega}_1 + \boldsymbol{I}^{-1}(\bar{\boldsymbol{L}}_1 - \bar{\boldsymbol{L}}_0). \tag{134}$$

The Euler equation describes the rotation of a rigid body under application of an external torque [233, p. 451]

$$\bar{\boldsymbol{\tau}}' = \boldsymbol{\omega}' \times \bar{\boldsymbol{L}}' + \boldsymbol{I}' \frac{\mathrm{d}\boldsymbol{\omega}'}{\mathrm{d}t} \tag{135}$$

with the apostrophe indicating the representation in the body frame. The angular velocity can be expressed in the inertial frame as

$$\boldsymbol{\omega} = \boldsymbol{R}\,\boldsymbol{\omega}' \tag{136}$$

with the rotation matrix $\boldsymbol{R}$ (see section 3.7.1). The angular acceleration is the derivative of $\boldsymbol{\omega}$

$$\frac{\mathrm{d}\boldsymbol{\omega}}{\mathrm{d}t} = \frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{R}\,\boldsymbol{\omega}' = \frac{\mathrm{d}\boldsymbol{R}}{\mathrm{d}t}\,\boldsymbol{\omega}' + \boldsymbol{R}\,\frac{\mathrm{d}\boldsymbol{\omega}'}{\mathrm{d}t} = \frac{\mathrm{d}\boldsymbol{R}}{\mathrm{d}t}\,\boldsymbol{R}^{\mathrm{T}}\boldsymbol{R}\,\boldsymbol{\omega}' + \boldsymbol{R}\,\frac{\mathrm{d}\boldsymbol{\omega}'}{\mathrm{d}t} = \\ = \boldsymbol{S}(\boldsymbol{\omega})\,\boldsymbol{R}\,\boldsymbol{R}^{\mathrm{T}}\,\boldsymbol{\omega} + \boldsymbol{R}\,\frac{\mathrm{d}\boldsymbol{\omega}'}{\mathrm{d}t} = \boldsymbol{\omega}\times\boldsymbol{\omega} + \boldsymbol{R}\,\frac{\mathrm{d}\boldsymbol{\omega}'}{\mathrm{d}t} = \boldsymbol{R}\,\frac{\mathrm{d}\boldsymbol{\omega}'}{\mathrm{d}t} \tag{137}$$

by using the identity

$$\frac{\mathrm{d}\boldsymbol{R}}{\mathrm{d}t} = \boldsymbol{S}(\boldsymbol{\omega})\boldsymbol{R} \tag{138}$$

with the skew-symmetric matrix $\boldsymbol{S}(\boldsymbol{\omega})$ representing the cross product operator $\boldsymbol{\omega}\times$ as shown in [83, p. 2]. This allows to use eq. (135) in the inertial frame

$$\bar{\boldsymbol{\tau}} = \boldsymbol{R}\,\bar{\boldsymbol{\tau}}' = \boldsymbol{R}\,(\boldsymbol{\omega}'\times\bar{\boldsymbol{L}}') + \boldsymbol{R}\,\boldsymbol{I}'\,\frac{\mathrm{d}\boldsymbol{\omega}'}{\mathrm{d}t} = (\boldsymbol{R}\,\boldsymbol{\omega}')\times(\boldsymbol{R}\,\bar{\boldsymbol{L}}') + \boldsymbol{R}\,\boldsymbol{I}'\,\boldsymbol{R}^{\mathrm{T}}\boldsymbol{R}\,\frac{\mathrm{d}\boldsymbol{\omega}'}{\mathrm{d}t} = \boldsymbol{\omega}\times\bar{\boldsymbol{L}} + \boldsymbol{I}\,\frac{\mathrm{d}\boldsymbol{\omega}}{\mathrm{d}t} \tag{139}$$

since the cross product is invariant under rotation. In order to propagate the rotation of the rigid body for the next time step, the angular acceleration $\boldsymbol{\alpha}$ needs to be computed first. From eq. (139) follows

$$\boldsymbol{\alpha} = \frac{\mathrm{d}\boldsymbol{\omega}}{\mathrm{d}t} = \boldsymbol{I}^{-1}(\bar{\boldsymbol{\tau}} - \boldsymbol{\omega}\times\bar{\boldsymbol{L}}). \tag{140}$$

One recognizes that even in the absence of external torques, $\boldsymbol{\omega}$ is not constant as illustrated in fig. 4.15. For the numeric integration, the considerations of the translation can be reapplied. A linear regression leads to

$$\boldsymbol{\alpha}\,(t) = 2\boldsymbol{\alpha}_1 - \boldsymbol{\alpha}_2 + \frac{1}{\triangle t}\,(\boldsymbol{\alpha}_2 - \boldsymbol{\alpha}_1)\,t. \tag{141}$$

with the corresponding definition

$$\boldsymbol{\alpha}\,(-2\triangle t) = \boldsymbol{\alpha}_2,\ \boldsymbol{\alpha}\,(-\triangle t) = \boldsymbol{\alpha}_1. \tag{142}$$

This allows updating of the angular velocity

$$\boldsymbol{\omega}_0 = \boldsymbol{\omega}_1 + \int_{-\triangle t}^{0} \boldsymbol{\alpha}\left(t'\right)\mathrm{d}t' = \boldsymbol{\omega}_1 + \frac{1}{2}\,\triangle t\,(3\boldsymbol{\alpha}_1 - \boldsymbol{\alpha}_2) \tag{143}$$

and the consequent change in orientation

$$\triangle\varphi = \int_{-\triangle t}^{0} \boldsymbol{\omega}\left(t'\right)\mathrm{d}t' = \boldsymbol{\omega}_1\triangle t + \frac{1}{6}\,(4\boldsymbol{\alpha}_1 - \boldsymbol{\alpha}_2)\,\triangle t^2. \tag{144}$$

The rotation vector formalism (see section 3.7.2) is being used to propagate the orientation of the rigid body with the axis of rotation $\boldsymbol{\omega}$ and the angle $\triangle\varphi$.

4.5.3 Implementation

Module `Motion` uses Newton's laws of motion to numerically propagate the position and velocity of all rigid bodies in the simulation. The class `Motion` is derived from class `Module` and defined as follows excluding certain private declarations.

```
// Class Motion
class CubeSim::Module::Motion : public Module
{
public:

   // Constructor
   Motion(double time_step = _TIME_STEP);

   // Clone
   virtual Module* clone(void) const;

   // Time Step [s]
   double time_step(void) const;
   void time_step(double time_step);

private:

   // Default Time Step [s]
   static const double _TIME_STEP;

   // Behavior
   virtual void _behavior(void);
};
```

The class defines a constructor `Motion(double time_step = _TIME_STEP)` which allows the initialization of the time step. Smaller time steps allow more accurate simulation results but require more computational time. If the time step is omitted, the default time step of 1 s is used. The module uses advanced prediction techniques to allow a reasonable trade-off between simulation accuracy and computational time (see section 4.5.1).

The time step can be retrieved by calling method `time_step` without arguments or modified by passing the new value to the method.

The class defines method `_behavior` which consists of an infinite loop that is executed periodically when a time step is elapsed.

Since class `Module` is derived from `List<Module>::Item`, class `Motion` needs to define method `clone`.

4.5.4 Verification

For the direct verification of the presented propagation strategy, the position of Mars was computed over ten revolutions with a simulation time step of 1 h. Mars was chosen since the Earth and Moon orbit their common barycenter and their absolute position is a result of two superimposed orbits. This would make the comparison with reference vectors more difficult. The reference data was retrieved from the NASA JPL Horizons service [166]. In fig. 4.13 the results are presented where the relative position error is the ratio between the absolute position error and the orbit perimeter. One can clearly see that the results of all predictive models are very similar but the non-predictive model shows strong deviations.

The simulation was then repeated with a larger time step of 24 h to obtain a larger differences between the different prediction approaches and the result can be seen in fig. 4.14. One can clearly recognize that the non-predictive model still shows the largest deviations but also the predictive models show one magnitude larger deviations from the real planet position. In this case, even the linear prediction shows slightly better performance than the higher order predictive models. A time step of 24 h is comparatively long but also shows that the propagation model is able to achieve deviations of ca. 0.1% for 18 years

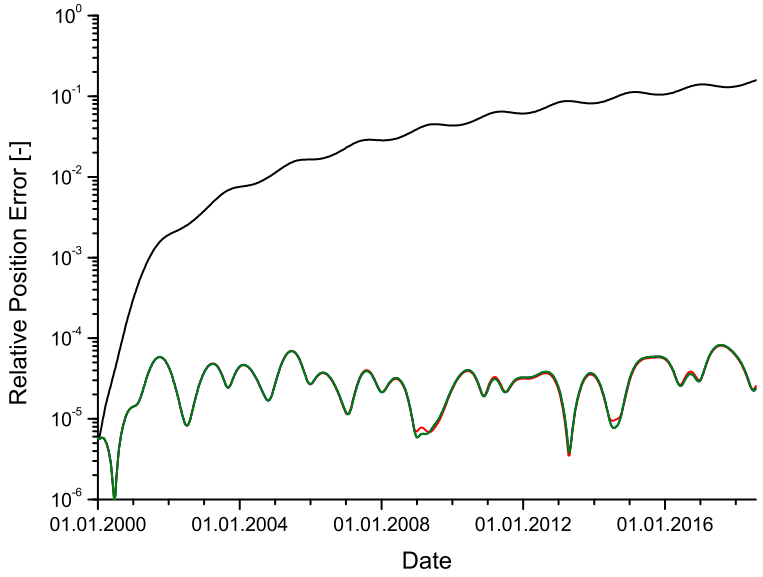

Fig. 4.13. Mars position propagation using different models with 1 h time step: no prediction (black), linear (red), second order polynomial (blue), third order polynomial (green).

simulated real-time. In practice, the time step is usually chosen significantly smaller to be able to simulate intra-spacecraft distances more exactly which is particularly relevant for formation flights.

As seen in eq. (129), in general $\boldsymbol{\omega}$ and $\boldsymbol{L}$ are not parallel. In fig. 4.15 the so-called *nutation cone* for an arbitrary rotation is illustrated. This shows that both, $\boldsymbol{\omega}$ and c rotate around the constant angular momentum $\boldsymbol{L}$.

For the following simulation, a 2.041 kg 2U CubeSat with different sub-systems was used and the inertia matrix in body frame was computed by CubeSim

$$\boldsymbol{I} = \begin{pmatrix} 8.93 & -0.10 & -0.18 \\ -0.10 & 8.99 & -0.18 \\ -0.18 & -0.18 & 3.70 \end{pmatrix} \times 10^{-3}\,\mathrm{kg\,m^2}. \tag{145}$$

No additional torques were applied to the spacecraft and the initial spin rate was set to $\boldsymbol{\omega}_0 = (-0.1 \;\; 0.1 \;\; 0.2)^{\mathrm{T}}$ rad/s. This leads to the constant angular momentum $\boldsymbol{L} = (9.39 \;\; 8.72 \;\; 7.41)^{\mathrm{T}} \times 10^{-4}\,\mathrm{kg\,m^2/s}$. The propagation over 200 seconds is shown in fig. 4.16 and one can clearly recognize the initial conditions $\boldsymbol{\omega} = \boldsymbol{\omega}_0$ at $t = 0$. The vector of the angular rate $\boldsymbol{\omega}$ rotates around the angular momentum $\boldsymbol{L}$ but the angle they enclose is constant at 0.427 rad.

For the following example, the 2U CubeSat was simplified and defined as a box with $10 \times 10 \times 20$ cm. With a uniform density of $\rho = 1\,\mathrm{g/cm^3}$, the mass results in exactly 2 kg. The inertia matrix in the body frame was computed again by CubeSim

$$\boldsymbol{I} = \begin{pmatrix} 8.33 & 0 & 0 \\ 0 & 8.33 & 0 \\ 0 & 0 & 3.33 \end{pmatrix} \times 10^{-3}\,\mathrm{kg\,m^2}. \tag{146}$$

See also section 7.1 for the analytical computation. Now, the x and y components of the initial angular rate were omitted $\boldsymbol{\omega}_0 = (0 \;\; 0 \;\; 0.2)^{\mathrm{T}}$ rad/s leading to a rotation around the z-axis only. This leads to the constant angular momentum $\boldsymbol{L} = (0 \;\; 0 \;\; 6.67)^{\mathrm{T}} \times 10^{-4}\,\mathrm{kg\,m^2/s}$ which is parallel to $\boldsymbol{\omega}_0$. Therefore, the

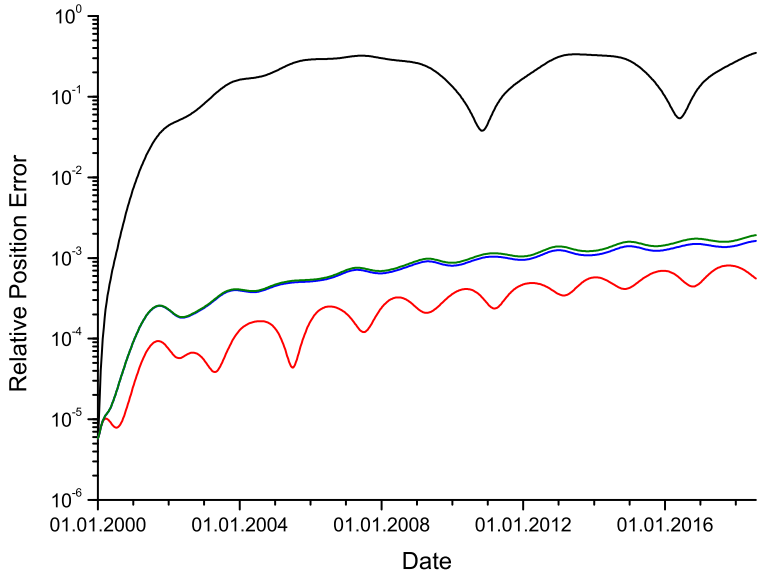

Fig. 4.14. Mars position propagation using different models with 24 h time step: no prediction (black), linear (red), second order polynomial (blue), third order polynomial (green).

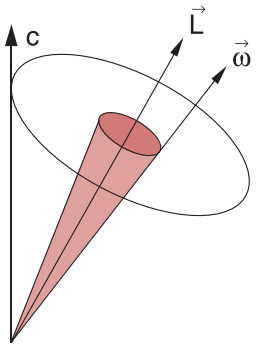

Fig. 4.15. Nutation cone with figure axis c, angular momentum $\boldsymbol{L}$ and angular rate $\boldsymbol{\omega}$ [52, p. 144].

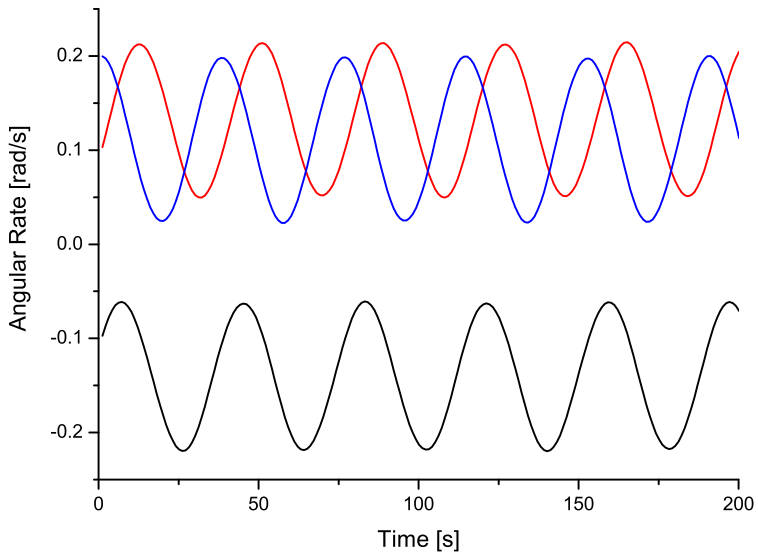

Fig. 4.16. Angular rate in global frame shown for 5 full revolutions: x (black), y (red), z (blue).

rotation is stable and both, the angular rate $\boldsymbol{\omega}$ and the figure axis c, do not vary in time. This expected behavior was confirmed by the CubeSim simulation.

4.6 Ephemeris

As already explained in the previous sections, module `Gravitation` (see section 4.1) and module `Motion` (see section 4.5) enable the simulation of the movement and rotation of celestial bodies and spacecraft in the solar system and beyond. By computing the gravitational forces between the planets and the sun and between spacecraft and all celestial bodies and successively updating their position, velocity and rotation require solving the n-body problem [87] numerically.

Since the simulation start time can be chosen arbitrarily, the initial position, velocity and rotation of the celestial bodies and spacecraft, being part of the simulation, need to be computed first. This is done by the module `Ephemeris`. When the simulation is started, the module computes the initial conditions at the given start date from the Keplerian orbit elements stored for each predefined celestial body or spacecraft. These are the six classical elements, semi-major axis a, the eccentricity e, the argument of the periapsis ω, the longitude of the ascending node Ω, the inclination i and the mean anomaly M, and in addition the period T for one revolution around the respective barycenter.

The ephemeris data of the planets in our solar system and the Earth-Moon barycenter was obtained from [223, p. 28] where mean values of the orbital elements and their rates of change from years 1800 to 2050 are used (see tbl. 4.3). The elements are with respect to the mean ecliptic and equinox of J2000.0 (1 January 2000 at 12:00 terrestrial time). In contrast to the listed six Keplerian elements, the mean longitude L and the longitude of perihelion $\bar{\omega}$ are used. The following relations are used to transform the orbital elements

$$\omega = \bar{\omega} - \Omega,\ M = L - \bar{\omega} + b\,T^2 + c\cos(fT) + s\sin(fT) \tag{147}$$

where the last three terms added to the mean anomaly are only required for planets Jupiter, Saturn, Uranus, Neptune and Pluto in the extended interval from 3000 BC to 3000 AD. In module `Ephemeris` they are not considered.

The inclination of the Earth-Moon orbital plane is slightly negative (-2.672×10^{-7} rad) where the inclination is commonly limited to $0 \leqslant i < \pi$. Therefore, a transformation of the orbital elements is required $i \to -i$ and $\Omega \to \Omega + \pi$ which also affect the argument of the periapsis $\omega \to \omega + \pi$.

In [205, p. 2] it is comprehensible, that this modifications lead to the same transformation from the Keplerian orbit elements to Cartesian coordinates. Similarly, the mean anomaly is limited to $0 \leqslant M < 2\pi$. If the value is out of range, it needs to be updated as well $M \to M \pm 2\pi$.

With eq. (52) the orbital period is computed from the semi-major axis and the Sun mass 1.98847×10^{30} kg [256, p. 9]

$$T = 2\pi\sqrt{\frac{a^3}{G\,M_{\text{sun}}}}. \tag{148}$$

The rates of change for the orbital elements between 1800 and 2050 AD are shown in tbl. 4.4. The corresponding rate of change for the orbital period is computed as follows

$$\dot{T} = \frac{\mathrm{d}T}{\mathrm{d}a}\dot{a} = 3\pi\sqrt{\frac{a}{G\,M_{\text{sun}}}}\dot{a}. \tag{149}$$

Table 4.3. Orbital elements of the planets around the Sun (units in m, s and rad).

Body	a	e	i	L	$\bar{\omega}$	Ω
Mercury	5.791×10^{10}	2.056×10^{-1}	0.122	4.403	0.508	0.844
Venus	1.082×10^{11}	6.777×10^{-3}	0.059	3.176	0.959	1.338
EM	1.496×10^{11}	1.671×10^{-2}	0.000	1.753	1.797	3.142
Mars	2.279×10^{11}	9.339×10^{-2}	0.032	−0.079	5.000	0.865
Jupiter	7.783×10^{11}	4.839×10^{-2}	0.023	0.600	4.787	1.754
Saturn	1.427×10^{12}	5.386×10^{-2}	0.043	0.872	5.916	1.984
Uranus	2.871×10^{12}	4.726×10^{-2}	0.013	5.467	1.692	1.292
Neptune	4.498×10^{12}	8.590×10^{-3}	0.031	−0.962	4.768	2.300
Pluto	5.906×10^{12}	2.488×10^{-1}	0.299	4.170	1.986	1.925

Table 4.4. Rates of change for the orbital elements of the planets in our Solar system (units in m/s and rad/s).

Body	$\dot{a}$	$\dot{e}$	$\dot{i}$	$\dot{L}$	$\dot{\bar{\omega}}$	$\dot{\Omega}$
Mercury	5.54×10^{2}	1.91×10^{-7}	-1.04×10^{-6}	2.61×10^{1}	2.80×10^{-5}	-2.19×10^{-5}
Venus	5.83×10^{3}	-4.11×10^{-7}	-1.38×10^{-7}	1.02×10^{1}	4.68×10^{-7}	-4.85×10^{-5}
EM	8.41×10^{3}	-4.39×10^{-7}	-2.26×10^{-6}	6.28×10^{0}	5.64×10^{-5}	0.00×10^{0}
Mars	2.76×10^{4}	7.88×10^{-7}	-1.42×10^{-6}	3.34×10^{0}	7.76×10^{-5}	-5.11×10^{-5}
Jupiter	-1.74×10^{5}	-1.33×10^{-6}	-3.21×10^{-7}	5.30×10^{-1}	3.71×10^{-5}	3.57×10^{-5}
Saturn	-1.87×10^{6}	-5.10×10^{-6}	3.38×10^{-7}	2.13×10^{-1}	-7.31×10^{-5}	-5.04×10^{-5}
Uranus	-2.94×10^{6}	-4.40×10^{-7}	-4.24×10^{-7}	7.48×10^{-2}	7.12×10^{-5}	7.40×10^{-6}
Neptune	3.93×10^{5}	5.11×10^{-7}	6.17×10^{-8}	3.81×10^{-2}	-5.63×10^{-5}	-8.88×10^{-7}
Pluto	-4.73×10^{5}	5.17×10^{-7}	8.41×10^{-9}	2.53×10^{-2}	-7.09×10^{-6}	-2.07×10^{-6}

The orbital data of the Earth and the Moon around their barycenter (marked as EM in tbls. 4.3 and 4.4) are retrieved from NASA JPL Horizons service [166] for the same time range. The mean values and their rates were computed with linear regression. The Keplerian orbit elements of the planets and the Earth-Moon barycenter are listed in 4.3. It is obvious that values for the eccentricity, the inclination and the longitude of the ascending node are identical for both bodies and that the difference between the mean longitudes and the longitudes of perihelion are always π. The mean orbital period is $T = 2.348 \times 10^{6}\,\mathrm{s}$ which corresponds to 27.18 days.

In [223, p. 28] the rates of change for all Keplerian orbit elements are given which result from a linear regression between the years 1800 and 2050 AD. Approximating the Earth and the Moon as point masses, they form a classic two-body system and feature stable orbits. Despite that, for sake of completeness, the rates are given in tbl. 4.4. It is obvious that apart from the semi-major axis, the rates of change are identical for both bodies. The orbital period is changing with $\dot{T} = -6.24 \times 10^{-2}$. A more detailed analysis on the generalized two-body problem with extended rigid bodies can be found in [204].

In addition, the ephemeris data of the international space station ISS and the Hubble space telescope orbiting the Earth is listed in tbl. 4.7. It needs to be mentioned that due to their low altitude, atmospheric

Table 4.5. Keplerian orbit elements and the orbital period of the Earth and the Moon around their barycenter (units in m and rad).

Body	a	e	i	L	$\bar{\omega}$	Ω
Earth	4.658×10^{6}	5.554×10^{-2}	0.090	1.129	4.600	2.181
Moon	3.783×10^{8}	5.554×10^{-2}	0.090	4.270	1.459	2.181

Table 4.6. Rates of change for the Keplerian orbit elements and the orbital period of the Earth and the Moon around their barycenter (units in m/s and rad/s).

Body	$\dot{a}$	$\dot{e}$	$\dot{i}$	$\dot{L}$	$\dot{\tilde{\omega}}$	$\dot{\Omega}$
Earth	-8.23×10^{-2}	-4.10×10^{-8}	-8.63×10^{-8}	8.40×10^{1}	7.10×10^{-1}	-3.38×10^{-1}
Moon	-6.69×10^{0}	-4.10×10^{-8}	-8.63×10^{-8}	8.40×10^{1}	7.10×10^{-1}	-3.38×10^{-1}

Table 4.7. Keplerian orbit elements of predefined spacecraft around the Earth (units in m, s and rad).

Body	a	e	ω	Ω	i	M	T
ISS	6.760×10^{6}	1.124×10^{-3}	1.496	4.487	0.928	4.564	5.527×10^{3}
Hubble	6.984×10^{6}	2.660×10^{-3}	2.845	3.628	0.829	5.980	5.809×10^{3}

effects strongly affect their orbits and the orbital values are only approximated. The values in the table were taken from [166] with reference to J2000.0 epoch.

In order to compute the position and the velocity of the celestial bodies and spacecraft in the simulation, class `Orbit` (see section 3.13) is used to transform the six Keplerian orbit elements into Cartesian coordinates in the global coordinate system (see section 5.3). The computation of the position and the velocity for the Earth and the Moon requires an intermediate step since both are orbiting around their barycenter which in turn orbits around the barycenter of the solar system. Therefore, the position and velocity of the EM barycenter are calculated first (see tbl. 4.3) and then the relative position and velocity of the Earth and the Moon are calculated (see tbl. 4.5). Finally, the position and velocity vectors are added respectively to obtain absolute coordinates in the global frame.

The Earth's rotation axis is tilted and precessing with the mean value of $\varepsilon = 23.4392811°$ [223, p. 21] which corresponds to a rotation around the x-axis in the global coordinate system. The angular rate of the Earth is $\omega = 7.292115 \times 10^{-5}$ rad/s [99, p. 4] which takes into account the difference between a solar day and a sidereal day. It also considers leap years since the number of days for one revolution of the Earth around the Sun is not integral [153, p. 2]. When the positions of the celestial bodies are computed at the begin of the simulation, the ephemeris module is also properly setting up the rotation and the angular rate of the Earth.

4.6.1 Implementation

The class `Ephemeris` uses orbital data of the planets in the solar system and of the spacecraft ISS and Hubble to allow proper initialization of their position, velocity, rotation and angular rate at the begin of a simulation where the start time can be chosen arbitrarily (see section 12). It is derived from class `Module` and defined as follows excluding certain private declarations.

```
// Class Ephemeris
class CubeSim::Module::Ephemeris : public Module
{
public:

    // Clone
    virtual Module* clone(void) const;

private:
```

```
    // Initialization
    virtual void _init(void);
};
```

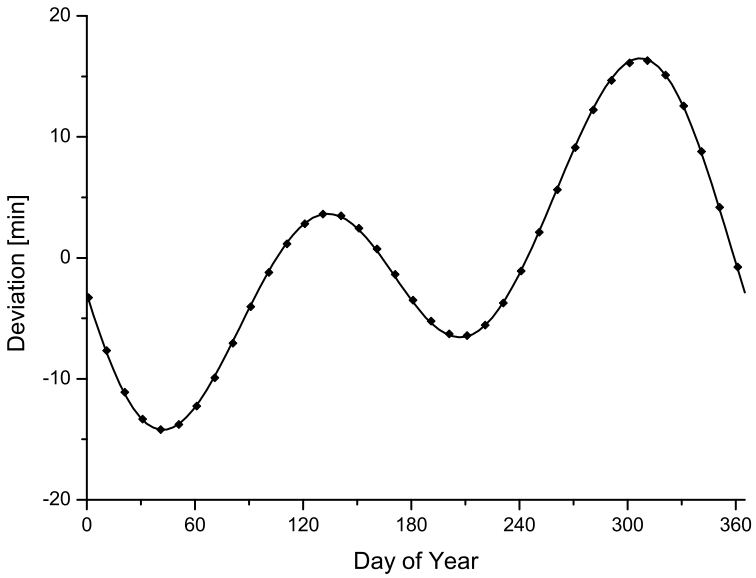

Fig. 4.17. Comparison of the equation of time as a result of the CubeSim simulation (symbols) and direct computation (curve).

The class does not provide any public operators or methods. Method `_init` initializes the position, rotation and velocity of all the celestial bodies and spacecraft in the simulation. It is called implicitly when the simulation is started.

Since class `Module` is derived from `List<Module>::Item`, class `Ephemeris` needs to define method `clone`.

4.6.2 Verification

Solar noon is the moment when the Sun reaches the highest elevation observed from Earth. Due to the eccentricity of the Earth orbit and the obliquity of the ecliptic (Earth axis tilt), noon does not occur exactly at 12:00. Fig. 4.17 shows the relative deviation of up to 30 minutes occurring over one year as computed by the CubeSim simulation for the year 2000. In October the maximum positive difference is ca. +16 minutes and in February the maximum negative difference is ca. −14 minutes which match the numbers found in literature [149, p. 13].

In [185] a detailed analysis and derivation of the so-called *equation of time* is performed. The second order approximation describing the time difference is obtained

$$E_t(t) = 7.654 \sin M - 9.86 \sin 2L - 0.659 \cos 2L \sin M + 0.212 \sin 4L + 0.08 \sin 2M \tag{150}$$

with the mean anomaly M and the mean longitude L

$$M = \frac{2\pi(t - 3.507)}{365.25}, \quad L = \frac{2\pi(t - 81.2)}{365.25} \tag{151}$$

where t represents the days elapsed since 1 January 2000. The result of eq. (150) is shown in fig. 4.17 (black curve) which shows good accordance with the simulation results (symbols).

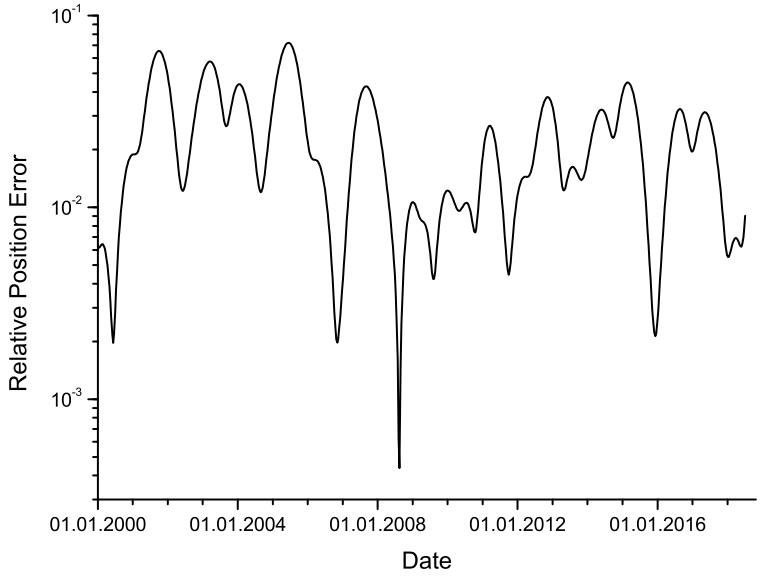

Fig. 4.18. Relative error of computed Mars position compared with reference data.

For the next verification, the position of Mars was computed by module `Ephemeris` and compared with reference data was retrieved from the NASA JPL Horizons service [166]. The relative distance error is computed by dividing the absolute deviation by the travel distance around the Sun. The results are shown in fig. 4.18. In contrast to the Earth and the Moon orbiting their common barycenter, the orbits of the planets cannot be precisely described by solving the equation of motion for a two-body problem. Other planets, in particular Jupiter, the Earth and Venus influence the motion of Mars which results in deviations from the ideal elliptical orbit. More details on the mutual influence of the planets can be found in [91].

It is obvious that the major limitation of module `Ephemeris` is the exclusion of the aforementioned mutual influences between the planets (n-body problem). Therefore, if more accurate ephemeris data is required for certain simulation scenarios or if the desired simulation time is far in the future, direct data from [166] should be used. For most small spacecraft simulations scenarios in Earth orbits, the deviations as shown in fig. 4.18 can be accepted though.

Chapter 5

Coordinate Systems

In the simulation framework different coordinate systems are being used, such as the body, the local or the global coordinate system. Depending on the application, the most suitable frame is chosen. Conversion between the coordinate systems is done implicitly and methods are provided to the user for that purpose.

5.1 Body Frame

The body coordinate system or body frame is used to describe properties of the body itself which do not change with the movement or the rotation of the body, such as the definition of an applied force or torque by using respective methods `force` and `torque` defined in class `RigidBody`. This can be useful when the user needs to model a thruster since in that case the force vector is fixed in the body frame such as the axis of the nozzle. If the force was given in another coordinate system, the user would have to realign the vector in each iteration. In the second example, the torque generated by a magnetic magnetorquer is modeled. Similar to the thrust vector, the torque vector is fixed with respect to the planar coil or solenoid and there is no need for the user to realign it in each iteration.

A couple of internal methods exist in class `RigidBody` which need to be defined by a class derived from `RigidBody`. The particularly applies to classes `Part`, `Assembly`, `System` and `Spacecraft`. Method `_center` needs to return the center of mass in the body frame, method `_contains` checks if a point passed in the body frame is within the volume of the rigid body and method `_inertia` needs to return the moment of inertia in the body frame. If the rigid body contains moving parts, the internal angular and linear momenta are expressed by methods `_angular_momentum` and `_momentum` also in the body coordinate system. More details on these methods can be found in chapter 6.

It needs to be noted that the center of mass does not necessarily have to coincide with the origin of the body frame. The position of the center of mass for various parts is given in chapter 7.

5.2 Local Frame

The local coordinate system is relevant for groups of bodies, such as if multiple parts build an assembly, if multiple assemblies build a system or if multiple systems build a spacecraft. In that case the body coordinate system of the parent is called local frame for their child objects. A child object can be positioned in the local frame in terms of location and rotation. Please refer to the methods `position`, `move`, `rotation` and `rotate` of class `RigidBody` in section 6 for details.

Method `angular_momentum` of class `RigidBody` returns the angular momentum of the rigid body in the local frame also taking into account the inner angular momentum expressed by method `_angular_momentum` in the body frame, see eq. (154). The relevant angular rate as set or returned by method `angular_rate` is also defined in the local frame.

Similarly, method `momentum` returns the momentum of the rigid body in the local frame and is also considering the inner momentum as returned by method `_momentum` in the body frame, see eq. (155). The relevant velocity as set or returned by method `velocity` is also defined in the local frame.

Method `center` of class `RigidBody` returns the center of mass in the local frame, method `contains` checks if a point passed in the local frame is within the volume of the rigid body and method `inertia` computes the moment of inertia in the local frame. Method `wrench` computes the wrench in local frame from all applied forces and torques as defined by methods `force` and `torque` in the body frame.

5.3 Global Frame

The global or celestial coordinate system is used to represent the location of the celestial bodies and spacecraft in the solar system and beyond. Its origin coincides with the barycenter of the solar system and the xy-plane is defined by the Earth's orbit at the reference epoch J2000.0. The x-axis is along the ascending node of the Earth's orbital and the mean equatorial plane at the reference epoch (vernal equinox). The z-axis is perpendicular to the xy-plane at the reference epoch [166, p. 26]. This coordinate system is also called ecliptic coordinate system and denoted as ICRF/J2000.0 (International Celestial Reference Frame) [166, p. 25].

Module `Ephemeris` uses the Keplerian orbit elements around the solar system barycenter to compute the initial position and velocity of predefined celestial bodies and spacecraft. Depending on the simulation start time, it computes the coordinates in the global frame (see section 4.6).

For some applications, it is required to convert body or local frame coordinates to global coordinates, such as if the absolute position and orientation of a sun sensor needs to be determined to compute the irradiance. Method `locate` of class `RigidBody` can be used for that purpose (see chapter 6).

5.4 Earth-centered, Earth-fixed Frame

The orgin of the ECEF coordinate system coincides with the center of mass of the Earth and the xy-plane is defined by the Earth's mean equator on 1 January 2000 at 12:00 terrestrial time (J2000.0 reference epoch). The x-axis is along the prime meridian (longitude of zero), the z-axis is pointing to the north pole and the y-axis is defined to make the system right-handed [40, p. 2].

As indicated by the name, this reference frame is fixed to the Earth and moves and rotates with the Earth if observed from the global frame. ECEF coordinates are commonly used to represent the position of satellites relative to the Earth. It gives precise values without having the necessity to choose a reference ellipsoid, such as required for the World Geodetic System 1984 [101, p. 22]. Details on the conversion between the Cartesian ECEF coordinates and the spherical coordinates longitude, latitude and altitude with respect to the flattening of the Earth reference ellipsoid can be found in section 9.1.

5.5 Earth-centered Inertial Frame

Like the ECEF coordinate system, also the ECI coordinate system uses the Earth center of mass as its origin. The xy-plane is defined by the Earth's mean equator on 1 January 2000 at 12:00 terrestrial

time (J2000.0 reference epoch). The x-axis is the intersection between the Earth's orbital and the mean equatorial plane and is pointing to the Sun at the vernal equinox. The z-axis is pointing to the north pole and the y-axis is defined to make the system right-handed [251, p. 7], [82, p. 331].

In contrast to the ECEF coordinate system, the ECI coordinate system is not rotating. In particular for non-geostationary objects in space, such as satellites in LEO, it is advantageous to represent their position in ECI coordinates.

For the transformation between ECEF and ECI coordinates, the position of the Sun is considered in both frames. At the vernal equinox at solar noon on the prime meridian the ECEF and ECI coordinate systems coincide. This epoch is denoted as t_0 and required for the later transformation between both coordinate systems. For t_0 the vernal equinox on 20 March 2009 at 11:53 [165] is used since it has a negligible time difference to solar noon at 12:07 under the consideration of eq. (150).

The conversion from a point in ECEF coordinates $\hat{\boldsymbol{x}}$ to a point in ECI coordinates $\boldsymbol{x}$ involves the rotation along the z-axis. The longitude in the ECEF frame is computed and modified according to the difference in time [60, p. 26]

$$\lambda = \operatorname{arctan2}\left(\hat{y},\, \hat{x}\right) + \omega \triangle t \tag{152}$$

with the Earth's angular rate $\omega = 7.292115 \times 10^{-5}$ rad/s [99, p. 4] and time difference $\triangle t$ denoting the seconds elapsed since 20 March 2009 12:07. From the longitude λ the ECI coordinates can be computed

$$x = \hat{r}\cos\lambda, \quad y = \hat{r}\sin\lambda, \quad z = \hat{z} \tag{153}$$

with the distance from the z-axis $\hat{r} = \sqrt{\hat{x}^2 + \hat{y}^2}$.

Chapter 6

Rigid Bodies

The class `RigidBody` implements the concept of rigid bodies to which multiple forces and torques can be applied. Parts are derived from rigid bodies as well as celestial bodies. The position, velocity, rotation and angular rate of a rigid body can be defined. Certain methods are provided which compute physical properties, such as the angular momentum, the area, the center of mass, the momentum or the mass. The class is defined as follows excluding certain private declarations.

```
// Class RigidBody
class CubeSim::RigidBody : private List<Force>, private List<Torque>
{
public:

   // Constructor
   RigidBody(void);
   RigidBody(const Vector3D& position, const Rotation& rotation);
   RigidBody(const Vector3D& position, const Rotation& rotation, const
      Vector3D& velocity,
      const Vector3D& angular_rate);

   // Copy Constructor (Rigid Body Reference is reset)
   RigidBody(const RigidBody& rigid_body);

   // Assign (Rigid Body Reference is maintained)
   RigidBody& operator =(const RigidBody& rigid_body);

   // Compute angular Momentum (local Frame) [kg*m^2/s]
   const Vector3D angular_momentum(void) const;

   // Angular Rate (local Frame) [rad/s]
   const Vector3D& angular_rate(void) const;
   void angular_rate(double x, double y, double z);
   void angular_rate(const Vector3D& angular_rate);

   // Compute Surface Area [m^2]
   double area(void) const;

   // Compute Center of Mass (local Frame) [m]
   const Vector3D center(void) const;

   // Check if contains Point (local Frame)
   bool contains(double x, double y, double z) const;
```

```
    bool contains(const Vector3D& point) const;

    // Get Force (Body Frame) [N]
    const std::map<std::string, Force*>& force(void) const;
    Force* force(const std::string& name) const;

    // Compute Moment of Inertia (local Frame, around Center for free
       rigid Bodies) [kg*m^2]
    const Inertia inertia(void) const;

    // Insert Force (Body Frame) and Torque (Body Frame)
    Force& insert(const std::string& name, const Force& force);
    Torque& insert(const std::string& name, const Torque& torque);

    // Locate in global Frame
    const std::pair<Vector3D, Rotation> locate(void) const;

    // Compute Mass [kg]
    double mass(void) const;

    // Compute Momentum (local Frame) [kg*m/s]
    const Vector3D momentum(void) const;

    // Move (local Frame) [m]
    void move(double x, double y, double z);
    void move(const Vector3D& distance);

    // Position (local Frame) [m]
    const Vector3D& position(void) const;
    void position(double x, double y, double z);
    void position(const Vector3D& position);

    // Rotate (local Frame)
    void rotate(const Rotation& rotation);
    void rotate(const Vector3D& axis, double angle);

    // Rotation (local Frame)
    const Rotation& rotation(void) const;
    void rotation(const Rotation& rotation);
    void rotation(const Vector3D& axis, double angle);

    // Get Torque (Body Frame) [N*m]
    const std::map<std::string, Torque*>& torque(void) const;
    Torque* torque(const std::string& name) const;

    // Velocity (local Frame) [m/s]
    const Vector3D& velocity(void) const;
    void velocity(double x, double y, double z);
    void velocity(const Vector3D& velocity);

    // Compute Volume [m^3]
    double volume(void) const;

    // Compute Wrench (local Frame)
    const Wrench wrench(void) const;

protected:

    // Cache
```

```
    static const uint8_t _CACHE_ANGULAR_MOMENTUM = 0x01;
    static const uint8_t _CACHE_AREA = 0x02;
    static const uint8_t _CACHE_CENTER = 0x04;
    static const uint8_t _CACHE_INERTIA = 0x08;
    static const uint8_t _CACHE_MASS = 0x10;
    static const uint8_t _CACHE_MOMENTUM = 0x20;
    static const uint8_t _CACHE_VOLUME = 0x40;
    static const uint8_t _CACHE_WRENCH = 0x80;

    // Update Properties
    static const uint8_t _UPDATE_ANGULAR_MOMENTUM = 1;
    static const uint8_t _UPDATE_ANGULAR_RATE = 2;
    static const uint8_t _UPDATE_AREA = 3;
    static const uint8_t _UPDATE_CENTER = 4;
    static const uint8_t _UPDATE_FORCE = 5;
    static const uint8_t _UPDATE_INERTIA = 6;
    static const uint8_t _UPDATE_MASS = 7;
    static const uint8_t _UPDATE_MOMENTUM = 8;
    static const uint8_t _UPDATE_POSITION = 9;
    static const uint8_t _UPDATE_ROTATION = 10;
    static const uint8_t _UPDATE_TORQUE = 11;
    static const uint8_t _UPDATE_VELOCITY = 12;
    static const uint8_t _UPDATE_VOLUME = 13;
    static const uint8_t _UPDATE_WRENCH = 14;

    // Update Property
    void _update(uint8_t update);

private:

    // Compute angular Momentum (Body Frame) [kg*m^2/s]
    virtual const Vector3D _angular_momentum(void) const;

    // Compute Surface Area [m^2]
    virtual double _area(void) const = 0;

    // Compute Center of Mass (Body Frame) [m]
    virtual const Vector3D _center(void) const = 0;

    // Check if contains Point (Body Frame)
    virtual bool _contains(const Vector3D& point) const = 0;

    // Compute Moment of Inertia (Body Frame) [kg*m^2]
    virtual const Inertia _inertia(void) const = 0;

    // Compute Mass [kg]
    virtual double _mass(void) const = 0;

    // Compute Momentum (Body Frame) [kg*m/s]
    virtual const Vector3D _momentum(void) const;

    // Remove and destroy Item
    virtual void _remove(const Force& force);
    virtual void _remove(const Torque& torque);

    // Compute Volume [m^3]
    virtual double _volume(void) const = 0;
```

```
    // Compute Wrench (Body Frame)
    virtual const Wrench _wrench(void) const;
};
```

When a rigid body is constructed, its initial position and rotation in the local frame can be defined. Additionally, dynamic properties like the velocity and angular rate can be set. If all arguments are omitted, the position of rigid body is the origin of the coordinate system and no rotation is applied. In that case the rigid body rests as no velocity or angular rate is applied.

The method `angular_momentum` can be used to compute the angular momentum of the rigid body in the local frame. If the rigid body consists of other rigid bodies (see assemblies, chapter 8) which have a non-zero angular rate, their angular momenta are considered as well. This is particularly required to model electro-mechanical systems, such as reaction wheels.

The angular momentum $\boldsymbol{L}$ of the rigid body is computed as

$$\boldsymbol{L} = \boldsymbol{I}\,\boldsymbol{\omega} + \boldsymbol{R}\sum \boldsymbol{L}_i + m\,\boldsymbol{C} \times \boldsymbol{v} \tag{154}$$

with the moment of inertia $\boldsymbol{I}$, the angular rate $\boldsymbol{\omega}$, the total internal angular momentum $\boldsymbol{L}_i$ defined in the body frame, the rotation matrix $\boldsymbol{R}$ to transform from the body frame into the local frame, the mass m, the center of mass in the local frame $\boldsymbol{C}$ and the linear motion in the local frame $\boldsymbol{v}$. The third term vanishes for free rigid bodies, otherwise it contributes to the angular momentum when the center of mass is not pointing in the same direction like the velocity. The resultant internal angular momentum is cached to reduce the computation time where the virtual private method `_angular_momentum` is used. Details can be found in chapter 8.

The method `angular_rate` is used to retrieve or set the angular rate of a rigid body in the local frame. The module `Motion` uses the conversion of angular momentum to iteratively modify the orientation of a free rigid body (such as a spacecraft) if the angular rate is changing (see section 4.5).

The area of rigid body is computed by the use of method `area`. It calls the virtual private method `_area` and caches the result for performance reasons.

Method `center` allows the computation of the center of mass in the local frame. It calls the virtual private method `_center` which computes the center of mass in the body coordinate system and caches the result for performance reasons.

The class also features the method `contains` which allows to test whether a certain point in the local frame is within the volume of the rigid body. This applies to compound rigid bodies as well. It calls the virtual private method `_contains` which evaluates the request in the body coordinate system. Method `contains` can be useful for debugging (e.g. numerical computation of the center of mass) or to numerically compute the effective area which is relevant for air drag simulation.

Geometric primitives, such as boxes, cones, prisms or spheres, are used for parts which represent mechanical components of the spacecraft like structural elements, sensors or actuators. In this section the focus is laid on the concept of rigid bodies and common properties. Additional properties, such as the material, are described in section 3.15 and chapter 7.

As explained in sections 3.9 and 3.10, multiple forces and torques acting on the rigid body can be simulated. Method `force` can be used to retrieve an applied force by specifying the name of force or to return a map of all forces of type `std::map<std::string, Force*>` if no name is passed. In order to insert a new force into that list, method `insert` can be used which expects the name of the force and the force of type `Force` as arguments. It returns a reference to the inserted copy of the passed force. All forces are defined in the body frame coordinate system.

Similarly, method `torque` is used to get an applied torque by specifying the name of torque or to return a map of all torques of type `std::map<std::string, Torque*>` if no name is passed. To insert a new torque into that list, method `insert` can be used which expects the name of the torque and the torque of type `Torque` as arguments. It returns a reference to the inserted copy of the passed torque. All torques are defined in the body frame coordinate system.

The resulting wrench is computed by method `wrench` which considers all forces and torques applied to the rigid body. It calls the virtual private method `_wrench` and caches the result to reduce computation time. Details on the computation can be found in section 3.11.

The method `inertia` computes the moment of inertia of the rigid body or the compound rigid body in the local frame. It calls the virtual private method `_inertia` which computes the moment of inertia in the body coordinate system and caches the result for performance reasons.

The rigid body can be part of a hierarchy and be displaced and rotated multiple times. It may be useful to locate the rigid body in the global frame which is required, e.g., to compute the torque of a coil in the Earth magnetic field or to determine the orientation of a sun sensor with respect to the sun. This can be accomplished with method `locate` which returns the position and the rotation in the global coordinate system.

Method `mass` is used to compute the mass of a rigid body or of the compound rigid body. It calls the virtual private method `_mass` and caches the result to reduce computation time.

Method `momentum` can be used to compute the momentum of the rigid body in the local frame. If the rigid body consists of other rigid bodies (see assemblies, chapter 8) which have a non-zero velocity, their momenta are considered as well. This is particularly required to model electro-mechanical systems, such as deployment mechanisms or propellant reservoirs. The momentum $\boldsymbol{p}$ of the rigid body is computed as

$$\boldsymbol{p} = m\,\boldsymbol{v} + \boldsymbol{R}\sum \boldsymbol{p}_i + m\,\boldsymbol{\omega} \times (\boldsymbol{C} - \boldsymbol{r}) \tag{155}$$

with the total internal momentum $\boldsymbol{p}_i$ defined in the body frame, the rotation matrix $\boldsymbol{R}$ to transform from the body into the local frame and the position of the rigid body in the local frame $\boldsymbol{r}$. The third term vanishes for free rigid bodies, otherwise it generates a momentum when the axis of rotation does not coincide with the center of mass.

By the use of methods `move` and `position`, the location of the rigid body in the local frame is modified where `move` changes it incrementally and `position` expects the new absolute position of type `Vector3D`. When `position` is called without arguments, the current position is returned.

Method `rotation` returns the rotation of the rigid body in the local frame if no arguments are passed. It also allows updating it, when the new rotation is passed to the method (either an object `Rotation` or the axis and the angle.

The current velocity of the rigid body can be retrieved by calling method `velocity` without arguments. It can also be modified by passing the new velocity of type `Vector3D` or individual coordinates.

Method `volume` allows the computation of the volume of the rigid body. It calls the virtual private method `_volume` and caches the result to reduce computation time.

To speed up the simulation, multiple values are cached as already explained. A sophisticated event handling system is implemented which reacts on changed properties. If for example the velocity of the rigid body is changed, the momentum is affected or if an acting force changes its magnitude, the resulting wrench needs to be recalculated. Therefore, the update method `_update` is introduced which is called by methods of class `RigidBody` or derived classes, such as `Part` or `Assembly`.

6.1 Caching Implementation

As shown in the previous listing, the cache flag register `_cache` is introduced, and the flags `_CACHE_ANGULAR_MOMENTUM`, `_CACHE_AREA`, `_CACHE_CENTER`, `_CACHE_INERTIA`, `_CACHE_MASS`, `_CACHE_MOMENTUM`, `_CACHE_VOLUME` and `_CACHE_WRENCH` are defined. These flags correspond to the methods `angular_momentum`, `area`, `center`, `inertia`, `mass`, `momentum`, `volume` and `wrench`. Whenever a value is computed for the first time, the corresponding flag is set. If the method is called a second time, the cached value is returned without the necessity of recalculation. Whenever a property of the rigid body is changed, e.g., its dimension, dependent properties, such as volume, need to be updated by using method `_update`. Such an update process might also propagate, e.g., the changed volume requires to update the mass as well. Depending on the passed argument, method `_update` selectively clears one or more flags and requires recalculation once the invalidated property is requested again.

To reuse the example given above: if the velocity of the rigid body is modified, method `velocity` calls `_update(_UPDATE_VELOCITY)` which in turn calls method `_update(_UPDATE_MOMENTUM)` which then clears the bit `_CACHE_MOMENTUM` of the flag register `_cache`. If the rigid body is contained by a parent rigid body, a change of the momentum in the local frame affects the momentum of the parent rigid body. Therefore, method `_rigid_body->_update(_UPDATE_MOMENTUM)` of the parent rigid body is called.

In the following, the alterable properties and their impact on the derived properties of the rigid body and, if applicable, on the parent rigid body are explained in greater detail.

6.1.1 Angular Momentum

The angular momentum is computed according to eq. (154). It changes with the angular rate, the velocity, the inertia, the mass and the center of mass. When method `_update(_UPDATE_ANGULAR_MOMENTUM)` is called, flag `_CACHE_ANGULAR_MOMENTUM` of the flag register `_cache` is cleared.

If the rigid body is contained by another rigid body, the updated angular momentum also affects the angular momentum of the parent rigid body as shown in eq. (154, second term). Therefore, `_rigid_body->_update(_UPDATE_ANGULAR_MOMENTUM)` is called.

6.1.2 Angular Rate

The angular rate of a rigid body can be modified by the user or for example by module `Motion` during the simulation. It directly affects the angular momentum and momentum of the rigid body as to eqs. (154) and (155). When the angular rate is updated, method `_update(_UPDATE_ANGULAR_RATE)` is being called. The method then calls recursively methods `_update(_UPDATE_ANGULAR_MOMENTUM)` and `_update(_UPDATE_MOMENTUM)`.

6.1.3 Area

The area of a rigid body depends on its shape and its dimensions, e.g., for a box as defined in the derived class `Part::Box` in section 7.1, the area is computed with eq. (157). When method `_update(_UPDATE_AREA)` is called, flag `_CACHE_AREA` of the flag register `_cache` is cleared.

If the rigid body is contained by another rigid body, the updated area also affects the area of the parent rigid body according to eq. (191). Therefore, `_rigid_body->_update(_UPDATE_AREA)` is called.

6.1.4 Center

The center of mass of a rigid body depends on its shape and its dimensions, e.g., for a box as defined in the derived class `Part::Box` in section 7.1, the center of mass is computed with eq. (158). When method `_update(_UPDATE_CENTER)` is called, flag `_CACHE_CENTER` of the flag register `_cache` is cleared.

As the center of a rigid body also affects the angular momentum and momentum as defined in eqs. (154) and (155), methods `_update(_UPDATE_ANGULAR_MOMENTUM)` and `_update(_UPDATE_MOMENTUM)` are called recursively.

If the rigid body is contained by another rigid body, the updated center of mass also affects the center of mass of the parent rigid body according to eq. (193). Therefore, `_rigid_body->_update(_UPDATE_CENTER)` is called.

6.1.5 Dimension

Each object of class `Part`, such as a sphere, cylinder or box, is fully defined by one ore more dimensions like the radius, length or width (see chapter 7). Though these properties are usually not modified during an active simulation run, the framework supports changes during run-time. This can particularly be of interest when draining of the propellant reservoir of a propulsion system shall be simulated since this changes the center of mass of the spacecraft and in turn might affect the stability of the AOCS.

The dimensions of a part affect the area, volume, inertia and center of mass as for example shown for the derived class `Part::Box` in section 7.1. If a dimension is modified, method `_update(_UPDATE_DIMENSION)` is called which recursively calls the methods `_update(_UPDATE_AREA)`, `_update(_UPDATE_CENTER)`, `_update(_UPDATE_INTERIA)` and `_update(_UPDATE_VOLUME)`.

6.1.6 Force

One or more forces applied to a rigid body are merged with applied torques to compute the wrench (see section 3.11) consisting of the total force acting on the center of mass and the torque as computed with eqs. (46) and (47). When the magnitude of a force is changed, method `_update(_UPDATE_MAGNITUDE)` is called and if the point of application is modified, method `_update(_UPDATE_POINT)` is called.

If the force is bound to a rigid body, its update method `_rigid_body->_update(_UPDATE_FORCE)` is called in both cases notifying the rigid body that one of the acting forces has changed in magnitude or point of application. This in turn triggers calling of the method `_rigid_body->_update(_UPDATE_WRENCH)`.

6.1.7 Inertia

The inertia is a cacheable property and when `_update(_UPDATE_INERTIA)` is being called, the corresponding flag `_CACHE_INERTIA` of the flag register `_cache` is cleared.

Since the angular momentum changes with the inertia as according to eq. (154), method `_update(_UPDATE_ANGULAR_MOMENTUM)` is called.

If the rigid body is contained by another rigid body, the updated inertia also affects the inertia of the parent rigid as shown in section 3.8 and chapter 8. Therefore, `_rigid_body->_update(_UPDATE_INERTIA)` is called.

6.1.8 Mass

The mass is a cacheable property and when `_update(_UPDATE_MASS)` is being called, the corresponding flag `_CACHE_MASS` of the flag register `_cache` is cleared.

Since the inertia changes with the mass as for exmaple according to eq. (160) for class `Part::Box`, method `_update(_UPDATE_INERTIA)` is called. Moreover, the angular momentum and the momentum are affected by the mass as shown in eqs. (154) and (155). This requires calling methods `_update(_UPDATE_ANGULAR_MOMENTUM)` and `_update(_UPDATE_MOMENTUM)` recursively.

If the rigid body is contained by another rigid body, the updated mass also affects the mass of the parent rigid body according to eq. (194). Therefore, `_rigid_body->_update(_UPDATE_MASS)` is called. With eq. (193) the center of a multi-body system is computed. Since it is subject to change with the mass, method `_rigid_body->_update(_UPDATE_CENTER)` is called recursively.

6.1.9 Material

Each object of class `Part` not only features dimensions to describe the shape of the primitive but also its material can be defined (see chapters 3.15 and 7). The material of a part affects the mass due to its defined density in accordance to eq. (156). Therefore, method `_update(_UPDATE_MASS)` is called when the material is replaced by another material or if the density of the material assigned to the part is modified. In the latter case `_part->_update(_UPDATE_MATERIAL)` is called.

6.1.10 Momentum

The momentum is computed according to eq. (155). It changes with the angular rate, the velocity, the mass, the center of mass and the position in the local frame. When method `_update(_UPDATE_MOMENTUM)` is called, flag `_CACHE_MOMENTUM` of the flag register `_cache` is cleared.

If the rigid body is contained by another rigid body, the updated momentum also affects the momentum of the parent rigid body as shown in eq. (155, second term). Therefore, `_rigid_body->_update(_UPDATE_MOMENTUM)` is called.

6.1.11 Position

The position of a rigid body affects the center of mass and momentum as defined in eqs. (193) and (155). This requires calling methods `_update(_UPDATE_CENTER)` and `_update(_UPDATE_MOMENTUM)` recursively.

If the rigid body is contained by another rigid body, the updated position also affects the wrench and the inertia of the parent rigid body according to eqs. (50) and (41). Therefore, `_rigid_body->_update(_UPDATE_WRENCH)` and `_rigid_body->_update(_UPDATE_INERTIA)` are called.

6.1.12 Rotation

Since the rotation of a rigid body affects the center of mass and momentum as defined in eqs. (193) and (155), methods `_update(_UPDATE_CENTER)` and `_update(_UPDATE_MOMENTUM)` are called recursively.

If the rigid body is contained by another rigid body, the updated rotation also affects the wrench and the inertia of the parent rigid body according to eqs. (50) and (41). Therefore, `_rigid_body->_update(_UPDATE_WRENCH)` and `_rigid_body->_update(_UPDATE_INERTIA)` are called.

6.1.13 Torque

One or more torques applied to a rigid body are merged with applied forces to compute the wrench (see section 3.11) consisting of the total force acting on the center of mass and the torque as computed with eqs. (46) and (47). When the magnitude of a torque is changed, method `_update(_UPDATE_MAGNITUDE)` is called.

If the torque is bound to a rigid body, its update method `_rigid_body->_update(_UPDATE_TORQUE)` is called notifying the rigid body that one of the acting torques has changed in magnitude. Since the wrench of the rigid body depends on the applied torques, the recursive method `_rigid_body->_update(_UPDATE_WRENCH)` is triggered.

6.1.14 Velocity

The velocity of a rigid body can be modified by the user or for example by module `Motion` during the simulation. It directly affects the angular momentum and momentum of the rigid body as to eqs. (154) and (155). When the velocity is updated, method `_update(_UPDATE_VELOCITY)` is being called. The method then calls recursively methods `_update(_UPDATE_ANGULAR_MOMENTUM)` and `_update(_UPDATE_MOMENTUM)`.

6.1.15 Volume

Since the volume of a rigid body is a cacheable property, the corresponding flag `_CACHE_VOLUME` of the flag register `_cache` is cleared if method `_update(_UPDATE_VOLUME)` is called. Since the volume of a part is used to compute its mass as shown in eq. (156), method `_update(_UPDATE_MASS)` is called recursively.

If the rigid body is contained by another rigid body, the modified volume affects the volume of the parent rigid body as shown in eq. (192). Therefore, `_rigid_body->_update(_UPDATE_VOLUME)` is called.

6.1.16 Wrench

As the wrench is a cacheable property, the corresponding flag `_CACHE_WRENCH` of the flag register `_cache` is cleared if method `_update(_UPDATE_WRENCH)` is called.

If the rigid body is contained by another rigid body, method `_rigid_body->_update(_UPDATE_WRENCH)` of the parent rigid body is called recursively since the wrench of the parent rigid body needs to be updated according to eqs. (51) and (50).

Chapter 7

Parts

Parts represent basic physical components that can be used to build an assembly (see chapter 8). These are derived from rigid bodies and allow the specification of the material (see section 3.15). The class contains the predefined solids box, cone, cylinder, prism and sphere but also allows the implementation of additional and even more complex shapes.

The class `Part` is defined as follows excluding certain private declarations.

```
// Class Part
class CubeSim::Part : public RigidBody, public List<Part>::Item
{
public:

   // Class Box
   class Box;

   // Class Cone
   class Cone;

   // Class Cylinder
   class Cylinder;

   // Class Prism
   class Prism;

   // Class Sphere
   class Sphere;

   // Clone
   virtual Part* clone(void) const = 0;

   // Material
   const Material& material(void) const;
   void material(const Material& material);

private:

   // Update Properties
   static const uint8_t _UPDATE_DIMENSION = 1;

   // Compute Mass
   virtual double _mass(void) const;
```

```
    // Update Property
    void _update(uint8_t update);
};
```

With the use of method `material` the currently defined material can be retrieved or a new material can be assigned.

The virtual method `_mass` defined in class `RigidBody` is implemented which uses the volume of the part V and the density ρ of the assigned material to compute the mass

$$m = \rho V. \tag{156}$$

7.1 Box

The class `Box` represents a cuboid defined by its length, width and height. Its origin in the body frame is one corner of the cuboid where the length spans along the x-axis, the width along the y-axis and the height along the z-axis. The class `Box` is defined as follows excluding certain private declarations.

```
// Class Box
class CubeSim::Part::Box : public Part
{
public:

    // Constructor
    Box(void);
    Box(double length, double width, double height);

    // Clone
    virtual Part* clone(void) const;

    // Height [m]
    double height(void) const;
    void height(double height);

    // Length [m]
    double length(void) const;
    void length(double length);

    // Width [m]
    double width(void) const;
    void width(double width);

private:

    // Compute Surface Area [m^2]
    virtual double _area(void) const;

    // Compute Center of Mass (Body Frame) [m]
    virtual const Vector3D _center(void) const;

    // Check if Point is inside (Body Frame)
    virtual bool _contains(const Vector3D& point) const;

    // Compute Moment of Inertia Tensor (Body Frame) [kg*m^2]
    virtual const Inertia _inertia(void) const;
```

```
    // Compute Volume [m^3]
    virtual double _volume(void) const;
};
```

An empty box can be created with the default constructor `Box(void)` where the length, width and height are all set to zero. With the use of constructor `Box(double length, double width, double height)` a box is created and initialized to the specified dimensions. As for all rigid bodies, the position and the orientation can be modified once the object is created. These methods are defined in the class `RigidBody` (see chapter 6).

The virtual methods `_area`, `_center`, `_contains`, `_inertia` and `_volume` are implemented. The surface area is computed as follows

$$A = 2\,(l\,w + w\,h + h\,l) \tag{157}$$

with the length l, the width w and the height h. The center of the box is defined by

$$\boldsymbol{C} = \frac{1}{2}\begin{pmatrix} l & w & h \end{pmatrix}^{\mathrm{T}}. \tag{158}$$

A point $\boldsymbol{x} = (x \;\; y \;\; z)^{\mathrm{T}}$ given in the body frame is within the box when the following condition is fulfilled:

$$\eta(\boldsymbol{x}) = (0 \leqslant x \leqslant l) \wedge (0 \leqslant y \leqslant w) \wedge (0 \leqslant z \leqslant h)\,. \tag{159}$$

The inertia of the box in the body frame around the center of mass is computed as

$$\boldsymbol{I} = \frac{m}{12}\begin{pmatrix} w^2 + h^2 & 0 & 0 \\ 0 & l^2 + h^2 & 0 \\ 0 & 0 & l^2 + w^2 \end{pmatrix} \tag{160}$$

with the mass m which is calculated from the volume of the part and the density of the specified material. The volume of the box is

$$V = l\,w\,h. \tag{161}$$

7.2 Cone

The class `Cone` represents a solid with a circular base that tapers smoothly to a point above the center. A cone is defined by its radius and its height. In the body frame, the circular base coincidences with the xy-plane with the center laying in the source of the coordinates where the height extends along the z-axis. The class `Cone` is defined as follows excluding certain private declarations.

```
// Class Cone
class CubeSim::Part::Cone : public Part
{
public:

    // Constructor
    Cone(void);
    Cone(double radius, double height);
```

```
    // Clone
    virtual Part* clone(void) const;

    // Height [m]
    double height(void) const;
    void height(double height);

    // Radius [m]
    double radius(void) const;
    void radius(double radius);

private:

    // Compute Surface Area [m^2]
    virtual double _area(void) const;

    // Compute Center of Mass (Body Frame) [m]
    virtual const Vector3D _center(void) const;

    // Check if Point is inside (Body Frame)
    virtual bool _contains(const Vector3D& point) const;

    // Compute Moment of Inertia Tensor (Body Frame) [kg*m^2]
    virtual const Inertia _inertia(void) const;

    // Compute Volume [m^3]
    virtual double _volume(void) const;
};
```

An empty cone can be created with the default constructor `Cone(void)` where the radius and the height are set to zero. With the use of constructor `Cone(double radius, double height)` a cone is created and initialized to the specified dimensions. As for all rigid bodies, the position and the orientation can be modified once the object is created. These methods are defined in the class `RigidBody` (see chapter 6).

The virtual methods `_area`, `_center`, `_contains`, `_inertia` and `_volume` are implemented. The surface area is computed as follows

$$A = \pi\, r \left(r + \sqrt{r^2 + h^2} \right) \tag{162}$$

with the radius r and the height h. The center of the cone is defined by

$$\boldsymbol{C} = \begin{pmatrix} 0 & 0 & \frac{h}{4} \end{pmatrix}^{\mathrm{T}}. \tag{163}$$

A point $\boldsymbol{x} = (x \;\; y \;\; z)^{\mathrm{T}}$ given in the body frame is within the cone when the following condition is fulfilled:

$$\eta(\boldsymbol{x}) = (0 \leqslant z \leqslant h) \wedge \left(\sqrt{x^2 + y^2} \leqslant \frac{r}{h} \left(h - z \right) \right). \tag{164}$$

The inertia of the cone in the body frame around the center of mass is computed as

$$\boldsymbol{I} = 3m \begin{pmatrix} \frac{h^2}{80} + \frac{3r^2}{20} & 0 & 0 \\ 0 & \frac{h^2}{80} + \frac{3r^2}{20} & 0 \\ 0 & 0 & \frac{r^2}{10} \end{pmatrix} \tag{165}$$

with the mass m which is calculated from the volume of the part and the density of the specified material. The volume of the cone is

$$V = \frac{\pi\, r^2\, h}{3}. \tag{166}$$

7.3 Cylinder

The class `Cylinder` represents a solid with a circular base and straight parallel sides. Similar to the cone, the cylinder is also defined by its radius and its height. In the body frame, the circular base coincidences with the xy-plane with the center laying in the source of the coordinates where the height extends along the z-axis. The class `Cylinder` is defined as follows excluding certain private declarations.

```
// Class Cylinder
class CubeSim::Part::Cylinder : public Part
{
public:

   // Constructor
   Cylinder(void);
   Cylinder(double radius, double height);

   // Clone
   virtual Part* clone(void) const;

   // Height [m]
   double height(void) const;
   void height(double height);

   // Radius [m]
   double radius(void) const;
   void radius(double radius);

private:

   // Compute Surface Area [m^2]
   virtual double _area(void) const;

   // Compute Center of Mass (Body Frame) [m]
   virtual const Vector3D _center(void) const;

   // Check if Point is inside (Body Frame)
   virtual bool _contains(const Vector3D& point) const;

   // Compute Moment of Inertia Tensor (Body Frame) [kg*m^2]
   virtual const Inertia _inertia(void) const;

   // Compute Volume [m^3]
   virtual double _volume(void) const;
};
```

An empty cylinder can be created with the default constructor `Cylinder(void)` where the radius and the height are set to zero. With the use of constructor `Cylinder(double radius, double height)` a cylinder is created and initialized to the specified dimensions. As for all rigid bodies, the position and the

orientation can be modified once the object is created. These methods are defined in the class `RigidBody` (see chapter 6).

The virtual methods `_area`, `_center`, `_contains`, `_inertia` and `_volume` are implemented. The surface area is computed as follows

$$A = 2\pi\, r\, (r + h) \tag{167}$$

with the radius r and the height h. The center of the cylinder is defined by

$$\boldsymbol{C} = \begin{pmatrix} 0 & 0 & \frac{h}{2} \end{pmatrix}^{\mathrm{T}}. \tag{168}$$

A point $\boldsymbol{x} = (x \quad y \quad z)^{\mathrm{T}}$ given in the body frame is within the cylinder when the following condition is fulfilled:

$$\eta(\boldsymbol{x}) = (0 \leqslant z \leqslant h) \wedge \left(\sqrt{x^2 + y^2} \leqslant r\right). \tag{169}$$

The inertia of the cylinder in the body frame around the center of mass is computed as

$$\boldsymbol{I} = \frac{m}{2} \begin{pmatrix} \frac{3r^2+h^2}{6} & 0 & 0 \\ 0 & \frac{3r^2+h^2}{6} & 0 \\ 0 & 0 & r^2 \end{pmatrix} \tag{170}$$

with the mass m which is calculated from the volume of the part and the density of the specified material. The volume of the cylinder is

$$V = \pi\, r^2\, h. \tag{171}$$

7.4 Prism

The class `Prism` represents a straight prism which is defined by its base (a polygon of type `Polygon2D`) and its height. In the body frame the base lays in the xy-plane and the height extends along the z-axis. The class `Prism` is defined as follows excluding certain private declarations.

```
// Class Prism
class CubeSim::Part::Prism : public Part
{
public:

   // Constructor
   Prism(const Polygon2D& base = Polygon2D(), double height = 0.0);

   // Base [m]
   const Polygon2D& base(void) const;
   void base(const Polygon2D& base);

   // Clone
   virtual Part* clone(void) const;

   // Height [m]
   double height(void) const;
   void height(double height);
```

```
private:

    // Compute Surface Area [m^2]
    virtual double _area(void) const;

    // Compute Center of Mass (Body Frame) [m]
    virtual const Vector3D _center(void) const;

    // Check if Point is inside (Body Frame)
    virtual bool _contains(const Vector3D& point) const;

    // Compute Moment of Inertia Tensor (Body Frame) [kg*m^2]
    virtual const Inertia _inertia(void) const;

    // Compute Volume [m^3]
    virtual double _volume(void) const;
};
```

An empty prism can be created with the default constructor `Prism(void)` where the base and the height are set to zero. With the use of constructor `Prism(const Polygon2D& base, double height)` a prism is created and initialized. As for all rigid bodies, the position and the orientation can be modified once the object is created. These methods are defined in the class `RigidBody` (see chapter 6).

The virtual methods `_area`, `_center`, `_contains`, `_inertia` and `_volume` are implemented. The surface area is computed as follows

$$A = 2A_b + p\,h \tag{172}$$

with the base area A_b and the perimeter p of the polygon defining the base. The surface area of a star-shaped polygon is the sum of the triangles

$$A_b = \frac{1}{2}\sum_{i=1}^{n} \|\boldsymbol{x}_i \times \boldsymbol{x}_{i+1}\| \tag{173}$$

with $\boldsymbol{x}_{n+1} = \boldsymbol{x}_1$. The perimeter of the polygon is the sum of the distances between the vertices $\boldsymbol{x}_i$

$$p = \sum_{i=1}^{n} \|\boldsymbol{x}_i - \boldsymbol{x}_{i+1}\|\,. \tag{174}$$

The center of the prism is defined by

$$\boldsymbol{C} = \begin{pmatrix} C_{b,x} & C_{b,y} & \frac{h}{2} \end{pmatrix}^{\mathrm{T}} \tag{175}$$

with the center of the base

$$\boldsymbol{C}_b = \frac{1}{6A_b}\sum_{i=1}^{n} (\boldsymbol{x}_i + \boldsymbol{x}_{i+1})\,\|\boldsymbol{x}_i \times \boldsymbol{x}_{i+1}\|\,. \tag{176}$$

A point $\boldsymbol{x} = (x \quad y \quad z)^{\mathrm{T}}$ given in the body frame is within the prism when the following condition is fulfilled

$$\eta(\boldsymbol{x}) = (0 \leqslant z \leqslant h) \wedge \bigvee_{i=1}^{n} \boldsymbol{x} \in \triangle(\boldsymbol{x}_i, \boldsymbol{x}_{i+1}). \tag{177}$$

A point $\boldsymbol{x}$ lies within a triangle $\triangle$ spanned by points $\mathbf{0}$, $\boldsymbol{x}_i$ and $\boldsymbol{x}_{i+1}$ if the triangle is counter clockwise oriented and the point is left from all edges or if the triangle is clockwise oriented and the point is on the right side. This can be tested with the individual cross products which must have the same sign

$$\boldsymbol{x} \times \boldsymbol{x}_i, \quad \boldsymbol{x}_{i+1} \times \boldsymbol{x}, \quad \boldsymbol{x} \times (\boldsymbol{x}_{i+1} - \boldsymbol{x}_i) - \boldsymbol{x}_i \times \boldsymbol{x}_{i+1}. \tag{178}$$

The inertia of the prism in the body frame around the center of mass is composed from the individual triangular prisms. The inertia of such as wedge with a triangular base $\{\mathbf{0}, \boldsymbol{x}_i, \boldsymbol{x}_{i+1}\}$ and a height h can be computed with the following intermediate values

$$xx = \frac{1}{6} \sum_{i=1}^{n} m_i \left(x_{i,x}^2 + x_{i,x}\, x_{i+1,x} + x_{i+1,x}^2 \right) - m\, C_x^2 \tag{179}$$

$$xy = \frac{1}{12} \sum_{i=1}^{n} m_i \left(x_{i,x}(2x_{i,y} + x_{i+1,y}) + x_{i+1,x}(x_{i,y} + 2x_{i+1,y}) \right) - m\, C_x C_y \tag{180}$$

$$yy = \frac{1}{6} \sum_{i=1}^{n} m_i \left(x_{i,y}^2 + x_{i,y}\, x_{i+1,y} + x_{i+1,y}^2 \right) - m\, C_y^2 \tag{181}$$

$$zz = m \frac{h^2}{12} \tag{182}$$

with the mass of the individual wedges m_i and the total mass m which is calculated from the volume of the part and the specified material. The volume of the prism is

$$V = A_b\, h. \tag{183}$$

The negative terms in eqs. (179), (180) and (181) on the right result from the center of mass and the parallel axis theorem [52, p. 132]. Finally, the inertia evaluates to

$$\boldsymbol{I} = \begin{pmatrix} yy + zz & -xy & 0 \\ -xy & xx + zz & 0 \\ 0 & 0 & xx + yy \end{pmatrix}. \tag{184}$$

7.5 Sphere

The class `Sphere` represents a solid sphere which is defined by its radius only. In the body frame the center lays in the source of the coordinates. The class is defined as follows excluding certain private declarations.

```
// Class Sphere
class CubeSim::Part::Sphere : public Part
{
public:

    // Constructor
    Sphere(double radius = 0.0);

    // Clone
    virtual Part* clone(void) const;

    // Radius [m]
    double radius(void) const;
    void radius(double radius);
```

```
private:

    // Compute Surface Area [m^2]
    virtual double _area(void) const;

    // Compute Center of Mass (Body Frame) [m]
    virtual const Vector3D _center(void) const;

    // Check if Point is inside (Body Frame)
    virtual bool _contains(const Vector3D& point) const;

    // Compute Moment of Inertia Tensor (Body Frame) [kg*m^2]
    virtual const Inertia _inertia(void) const;

    // Compute Volume [m^3]
    virtual double _volume(void) const;
};
```

An empty sphere can be created with the constructor `Sphere(double radius = 0.0)` when no radius is provided. In that case the radius is initialized to zero. When an argument is passed, a sphere is created and initialized to the specified radius. As for all rigid bodies, the position and the orientation can be modified once the object is created. These methods are defined in the class `RigidBody` (see chapter 6).

The virtual methods `_area`, `_center`, `_contains`, `_inertia` and `_volume` are implemented. The surface area is computed as follows

$$A = 4\pi\, r^2 \tag{185}$$

with the radius r. The center of the sphere is defined by

$$\boldsymbol{C} = \boldsymbol{0}. \tag{186}$$

A point $\boldsymbol{x} = (x \quad y \quad z)^{\mathrm{T}}$ given in the body frame is within the sphere when the following condition is fulfilled:

$$\eta(\boldsymbol{x}) = \sqrt{x^2 + y^2 + z^2} \leqslant r. \tag{187}$$

The inertia of the sphere in the body frame around the center of mass is computed as

$$\boldsymbol{I} = \frac{2m\, r^2}{5} \mathbb{1} \tag{188}$$

with the mass m which is calculated from the volume of the part and the density of the specified material. The volume of the sphere is

$$V = \frac{4\pi\, r^3}{3}. \tag{189}$$

Chapter 8

Assemblies

An assembly comprises of parts and sub-assemblies which allows to group related parts and enables reuse of a defined assembly, such as a Sun sensor assembly, once defined, can be put on each side of a spacecraft. This facilitates the modeling process when setting up the simulation. A similar hierarchical concept in used in most CAD systems and allows the user to follow the same approach.

Considering the hierarchy of the simulation framework, the class `Assembly` is derived from classes `RigidBody`, `List<Part>`, `List<Assembly>` and `List<Assembly>::Item`.

The class `Assembly` is defined as follows.

```
// Class Assembly
class CubeSim::Assembly : public RigidBody, private List<Assembly>,
    private List<Part>, public List<Assembly>::Item
{
public:

    // Constructor
    Assembly(void);

    // Copy Constructor
    Assembly(const Assembly& assembly);

    // Assign
    Assembly& operator =(const Assembly& assembly);

    // Get Assembly
    const std::map<std::string, Assembly*>& assembly(void) const;
    Assembly* assembly(const std::string& name) const;

    // Insert Assembly, Part, Force (Body Frame) and Torque (Body Frame)
    Assembly& insert(const std::string& name, const Assembly& assembly);
    Part& insert(const std::string& name, const Part& part);
    using RigidBody::insert;

    // Get Part
    const std::map<std::string, Part*>& part(void) const;
    Part* part(const std::string& name) const;

private:

    // Compute angular Momentum (Body Frame) [kg*m^2/s]
```

```
    virtual const Vector3D _angular_momentum(void) const;

    // Compute Surface Area [m^2]
    virtual double _area(void) const;

    // Compute Center of Mass (Body Frame) [m]
    virtual const Vector3D _center(void) const;

    // Check if Point is inside (Body Frame)
    virtual bool _contains(const Vector3D& point) const;

    // Compute Moment of Inertia (Body Frame) [kg*m^2]
    virtual const Inertia _inertia(void) const;

    // Compute Mass [kg]
    virtual double _mass(void) const;

    // Compute Momentum (Body Frame) [kg*m/s]
    virtual const Vector3D _momentum(void) const;

    // Compute Volume [m^3]
    virtual double _volume(void) const;

    // Compute Wrench (Body Frame)
    virtual const Wrench _wrench(void) const;
};
```

The default constructor `Assembly(void)` creates an empty assembly object where the copy constructor `Assembly(const Assembly& assembly)` can be used to duplicate an existing assembly.

An existing assembly can also be overwritten by the use of the assignment operator `operator =`.

By the use of method `insert`, parts, sub-assemblies, forces and torques can be inserted. In order to distinguish or modify them later on, a unique name has to be specified. The method returns a reference to the inserted copy of the passed object.

Method `assembly` is used to find a sub-assembly by specifying the name or to retrieve an associative map of type `std::map<std::string, Assembly*>` consisting of all sub-assemblies.

In order to find a specific part or to retrieve an associative map of type `std::map<std::string, Part*>` consisting of all parts in the assembly, method `part` can be used.

The virtual methods `_angular_momentum`, `_area`, `_center`, `_contains`, `_inertia`, `_mass`, `_momentum`, `_volume` and `_wrench` are implemented as follows.

The internal angular momentum is given by

$$\boldsymbol{L} = \sum \boldsymbol{L}_{a,\mathrm{i}} + \sum \boldsymbol{L}_{p,\mathrm{i}}. \tag{190}$$

with the angular momenta of sub-assemblies $\boldsymbol{L}_{a,\mathrm{i}}$ and parts $\boldsymbol{L}_{p,\mathrm{i}}$.

The surface area of the assembly is also the sum of the surface areas of the containing sub-assemblies and parts

$$A = \sum A_{a,\mathrm{i}} + \sum A_{p,\mathrm{i}} \tag{191}$$

with the surface areas of sub-assemblies $A_{a,\mathrm{i}}$ and parts $A_{p,\mathrm{i}}$.

Similar to the surface area of the assembly, the volume is the sum of the volumina of the sub-assemblies $V_{a,\mathrm{i}}$ and parts $V_{p,\mathrm{i}}$

$$V = \sum V_{a,\mathrm{i}} + \sum V_{p,\mathrm{i}}. \tag{192}$$

The center of mass of the assembly is computed from the individual masses and their center locations

$$\boldsymbol{C} = \frac{1}{m}\left(\sum m_{a,i}\,\boldsymbol{C}_{a,\mathrm{i}} + \sum m_{p,i}\,\boldsymbol{C}_{p,\mathrm{i}}\right) \tag{193}$$

with the masses of sub-assemblies $m_{a,\mathrm{i}}$ and parts $m_{p,\mathrm{i}}$ and the total mass

$$m = \sum m_{a,i} + \sum m_{p,i}. \tag{194}$$

To check whether a point lies within the assembly, the following condition must be met

$$\eta(\boldsymbol{x}) = \bigvee \boldsymbol{x} \in V_{a,\mathrm{i}} \vee \bigvee \boldsymbol{x} \in V_{p,\mathrm{i}}. \tag{195}$$

Method `contains` of the sub-assemblies and parts is used for that purpose.

In order to compute the inertia of the assembly, the inertia of the sub-assemblies and parts are combined. This can be done pairwise for all inertia matrices of the sub-assemblies $\boldsymbol{I}_{a,\mathrm{i}}$ and the parts $\boldsymbol{I}_{p,\mathrm{i}}$ by using eq. (44). The sequence of the pairwise combinations does not affect the final result. Details can be found in section 3.8.

The linear momentum of the assembly in body frame is the sum of the momenta of the sub-assemblies $\boldsymbol{p}_{a,\mathrm{i}}$ and the parts $\boldsymbol{p}_{p,\mathrm{i}}$

$$\boldsymbol{p} = \sum \boldsymbol{p}_{a,\mathrm{i}} + \sum \boldsymbol{p}_{p,\mathrm{i}}. \tag{196}$$

To compute the total wrench acting on the assembly, the wrenches of sub-assemblies $\boldsymbol{\xi}_{a,\mathrm{i}}$ and parts $\boldsymbol{\xi}_{p,\mathrm{i}}$ are combined. This can be done pairwise by using eqs. (50) and (51). The sequence of the pairwise combinations does not affect the final result. Details on the computation can be found in section 3.11.

Chapter 9

Systems

A system is an entity that is either modeling physical behavior or is representing the behavior of an intelligent component. Physical modeling can be rather versatile, such as the simulation of a battery with certain time-varying charge and discharge currents. Depending on the underlying model, the energy capacity, the temperature or the change of the voltage level can be computed in simulated real-time. An intelligent component, on the other hand, can be a system featuring digital components such as micro-controllers or DSPs (Digital Signal Processors). If the firmware is being developed in C/C++, it can be executed within the simulation framework with minor modifications. Not only, e.g., the pre-processing of sensor readings can be verified that way, but the entire on-board computer with its firmware can be represented by a single system object.

Considering the hierarchy of the simulation framework, the class `System` may consist of sub-systems and assemblies and is derived from classes `RigidBody`, `Behavior` and `List<System>::Item`.

The class `System` is defined as follows excluding certain private declarations.

```
// Class System
class CubeSim::System : public Behavior, public RigidBody, private
    List<Assembly>, private List<System>, public List<System>::Item
{
public:

    // Accelerometer
    class Accelerometer;

    // Class GNSS
    class GNSS;

    // Class Gyroscope
    class Gyroscope;

    // Class Magnetometer
    class Magnetometer;

    // Class Magnetorquer
    class Magnetorquer;

    // Class Photodetector
    class Photodetector;

    // Class ReactionWheel
    class ReactionWheel;
```

```
    // Constructor
    System(void);

    // Copy Constructor (Spacecraft and System References are reset)
    System(const System& system);

    // Assign (Spacecraft and System References are maintained)
    System& operator =(const System& system);

    // Get Assembly
    const std::map<std::string, Assembly*>& assembly(void) const;
    Assembly* assembly(const std::string& name) const;

    // Clone
    virtual System* clone(void) const = 0;

    // Disable
    void disable(void);

    // Enable
    void enable(void);

    // Insert Assembly, System, Force (Body Frame) and Torque (Body
       Frame)
    Assembly& insert(const std::string& name, const Assembly& assembly);
    System& insert(const std::string& name, const System& system);
    using RigidBody::insert;

    // Check if enabled
    bool is_enabled(void) const;

    // Get Simulation
    Simulation* simulation(void) const;

    // Get Spacecraft
    Spacecraft* spacecraft(void) const;

    // Get System
    const std::map<std::string, System*>& system(void) const;
    System* system(const std::string& name) const;

private:

    // Compute angular Momentum (Body Frame) [kg*m^2/s]
    virtual const Vector3D _angular_momentum(void) const;

    // Compute Surface Area [m^2]
    virtual double _area(void) const;

    // Compute Center of Mass (Body Frame) [m]
    virtual const Vector3D _center(void) const;

    // Check if contains Point (Body Frame)
    virtual bool _contains(const Vector3D& point) const;

    // Compute Moment of Inertia (Body Frame) [kg*m^2]
    virtual const Inertia _inertia(void) const;

    // Compute Mass [kg]
```

```
    virtual double _mass(void) const;

    // Compute Momentum (Body Frame) [kg*m/s]
    virtual const Vector3D _momentum(void) const;

    // Compute Volume [m^3]
    virtual double _volume(void) const;

    // Compute Wrench (Body Frame)
    virtual const Wrench _wrench(void) const;
};
```

The class `Behavior` adds the virtual method `_behavior` which is to be defined by the class deriving from `System`. As defined in section 2.2, the inherited virtual method `_behavior` represents the functionality or properties of the system and its implementation can be very different depending on the purpose of the system. It is important to emphasize that the execution of the code is executed in simulated real time, ensuring that the behavior is strongly deterministic in time, regardless of how the entire simulation is structured or how powerful the simulation computer is. The discrepancy between real-time and simulated real-time certainly depends on the hardware computing power, the complexity of the entire simulation or the choice of the minimum simulation time steps.

In the following sub-sections multiple predefined systems are described and their behavior is explained in greater detail. One clearly recognizes that the set of systems is not limited to the predefined ones. Depending on future simulation requirements, new systems can conveniently be derived from the base class `System` or existing systems like `System::GNSS` or `System::Magnetometer` can be tweaked to better meet the demands. Since the method `_behavior` is virtual, it can be overridden by the derived system if changes are required.

The default constructor `System(void)` is automatically called when an object of a derived class is created. In addition, the copy constructor `System(const System& system)` is provided when such an object is copied. In this case any references to the spacecraft and parent system are reset.

An existing system can also be overwritten by the use of the assignment operator `operator =`. Here the references to the spacecraft and parent system are maintained.

Each system provides methods `enable` and `disable` to allow activation or deactivation of specific systems during the simulation. This feature can be used to simulate broken communication with a sensor or sub-system or any malfunction thereof. It depends on the definition of the system how it reacts on any state transition. When predefined systems representing a sensor like `System::Gyroscope` or `System::Accelerometer` are disabled, they return an invalid number `NaN` or a vector of `NaN` values if applicable. System `System::Magnetorquer`, e.g., disables its magnetic moment if the system is disabled or system `System::Thruster` reduces its generated force to 0 in that case. The state of the system can be acquired by the use of method `is_enabled`.

With the aid of method `insert`, assemblies, systems, forces and torques can be inserted into the system. In order to distinguish or modify them later on, a unique name has to be given. The method returns a reference to the inserted copy of the passed object.

To obtain the spacecraft, the system was inserted into, method `spacecraft` can be called which is returning a pointer of type `Spacecraft*`. The method does not expect any arguments. If the system was not inserted into a spacecraft, `nullptr` is returned.

The method `simulation` called without any arguments, returns a pointer to the simulation, type `Simulation*`. Internally method `_spacecraft->simulation()` is called if the system was inserted into

a spacecraft. If the system was not inserted into a spacecraft or the spacecraft was not inserted into a simulation, `nullptr` is returned.

Method `system` is used to find a sub-system by specifying the name or to retrieve an associative map of type `std::map<std::string, System*>` consisting of all sub-systems.

As class `System` is derived from class `RigidBody`, the virtual methods `_angular_momentum`, `_area`, `_center`, `_contains`, `_inertia`, `_mass`, `_momentum`, `_volume` and `_wrench` need to be implemented. Similar to class `Assembly`, these methods iterate through all inserted assemblies and sub-systems to compute the resultant values.

9.1 Global Navigation Satellite Systems

Means to detect the absolute orbital position of a spacecraft are essential for most missions. For satellites orbiting the Earth, GNSS (Global Navigation Satellite Systems) can be used for this purpose. The number of spacecraft, their orbit and the transmission frequency range are different between GPS (Global Positioning Systems), GLONASS (Globalnaja Nawigazionnaja Sputnikowaja Sistema) and Galileo [49, p. 9]. The spacecraft continuously synchronize their very accurate local clocks under consideration of signal delays and relativistic effects. Since the inception of GPS in 1978, these techniques have matured rapidly [58, p. 3]. The spacecraft transmit their position and the actual time stamp and the receiver combines the received data frames in order to compute its own position and velocity. The navigation systems use different geodetic reference systems, i.e., GPS uses WGS-84 (World Geodetic System 1984), GLONASS uses PZ-90 (Parametry Zemli 1990) and Galileo uses the GTRF (Galileo Terrestrial Reference Frame). The definition of the Earth-fixed reference PZ-90 and WGS-84 differs by under 15 m [192, p. 190]. The GTRF was defined to be in accordance with the ITRF (International Terrestrial Reference Frame) within an accuracy of 3 cm [9, p. 76]. What they have in common is that they all model the Earth as an ellipsoid considering the ablation at the poles. The system `System::GNSS` uses the WGS-84 reference system, but also other implementations or adaptations are possible. The radius of the WGS-84 reference ellipsoid is $r = 6.378137 \times 10^6$ m with a flattening $f = 3.352811 \times 10^{-3}$ [101, p. 22].

To compute the position `x` of a spacecraft `spacecraft` of type `Spacecraft` relative to Earth `earth` of type `CelestialBody` in the ECEF coordinate system, the following code can be used:

```
Vector3D x = spacecraft.position() - earth.position() -
    earth.rotation();
```

The first operation computes the relative position of the spacecraft from Earth in the global coordinate frame and the second operation transforms the vector into the ECEF frame. Since the system `System::GNSS` is simulating a GNSS transponder, the Cartesian coordinates still need to be transformed into an NMEA (National Marine Electronics Association) compatible representation consisting of longitude, latitude and altitude [133, p. 146]. Latitude and longitude are usually specified in degrees and the altitude in meters [3, p. 1-3]. Let $\boldsymbol{x}$ be a point outside the ellipsoid in Cartesian coordinates which may be expressed as follows:

$$\boldsymbol{x} = \begin{pmatrix} (N+h)\cos\varphi\ \cos\lambda \\ (N+h)\cos\varphi\ \sin\lambda \\ \left(\left(1-e^2\right)N+h\right)\sin\varphi \end{pmatrix} \tag{197}$$

with the latitude φ, longitude λ, altitude h, the radius of curvature in the prime vertical N being defined as

$$N = \frac{a}{\sqrt{1 - e^2 \sin^2 \varphi}} \tag{198}$$

and the eccentricity

$$e = \sqrt{1 - \left(\frac{b}{a}\right)^2} = \sqrt{(2 - f)\, f} \tag{199}$$

with the semi-major axis a of the ellipsoid, the semi-minor axis b and the flattening f. The usual notation of the flattening is

$$f = 1 - \frac{b}{a}. \tag{200}$$

One immediately recognizes that the transformation from geodetic coordinates into Cartesian can be done analytically with the given formulae. In order to convert Cartesian coordinates into geodetic ones, only the computation of the longitude is straightforward:

$$\lambda = \arctan2\,(y,\ x) \tag{201}$$

where the function arctan2 is used again, see eq. (22). Bowring proposed his geodetic-latitude equation

$$\kappa - 1 - \frac{e^2 a \kappa}{\sqrt{x^2 + y^2 + (1 - e^2)\, z^2 \kappa^2}} = 0. \tag{202}$$

After the equation is being solved numerically, the latitude

$$\varphi = \arctan2\left(\kappa z,\ \sqrt{x^2 + y^2}\right) \tag{203}$$

and the altitude

$$h = \frac{e^2 + 1/k - 1}{e^2} \sqrt{x^2 + y^2 + z^2 k^2} \tag{204}$$

can be computed [28]. The class `Location` performs these coordinate transformations in both directions and uses the proposed Newton-Raphson iterative method to solve eq. (202). Argument κ is initialized with

$$\kappa_0 = \frac{1}{1 - e^2} \tag{205}$$

and the following expressions are evaluated iteratively until κ converges:

$$c = \frac{\left(x^2 + y^2 + \left(1 - e^2\right) z^2 k^2\right)^{3/2}}{a e^2}, \quad \kappa \to \frac{c + \left(1 - e^2\right) z^2 \kappa^3}{c - x^2 - y^2}. \tag{206}$$

The actual location of the spacecraft can be acquired by using method `location` which does not expect any arguments. It returns an object of type `Location` which stores the longitude, the latitude, the altitude, a reference to the celestial body and a time stamp. Methods `altitude`, `latitude`, `longitude`, `celestial_body` and `time` can be used to get or set these values.

The radial distance to the point can be computed with method `radius` when no arguments are passed or modified by passing the new radius to the method. In the latter case the altitude is updated accordingly.

Implicit conversion from Cartesian to geodetic coordinates can be performed with the use of method `point` to set the Cartesian coordinates of the specified position. Consequently the latitude, the longitude, the

altitude and the radius can be read-out. Transforming geodetic coordinates into Cartesian ones works similarly - if the latitude, the longitude and the altitude were modified previously, method `point` is to be called without arguments to return the Cartesian representation. A more detailed description of class `Location` can be found in section 3.12.

9.1.1 Implementation

The class `GNSS` is used to simulate a GNSS transponder to retrieve the position of a spacecraft around the Earth in the ECEF frame. This allows the verification of the AOCS algorithm which typically propagates the spacecraft position with respect to the Earth. It is derived from `System` and defined as follows excluding certain private declarations.

```
// Class GNSS
class CubeSim::System::GNSS : public System
{
public:

    // Clone
    virtual System* clone(void) const;

    // Get Location
    const Location location(void) const;

protected:

    // Constructor
    GNSS(double spatial_accuracy = _SPATIAL_ACCURACY, double
        temporal_accuracy = _TEMPORAL_ACCURACY);

    // Copy Constructor
    GNSS(const GNSS& gnss);

    // Spatial Accuracy [m]
    double _spatial_accuracy(void) const;
    void _spatial_accuracy(double spatial_accuracy);

    // Temporal Accuracy [s]
    double _temporal_accuracy(void) const;
    void _temporal_accuracy(double temporal_accuracy);

private:

    // Default spatial Accuracy [m]
    static const double _SPATIAL_ACCURACY;

    // Default temporal Accuracy [s]
    static const double _TEMPORAL_ACCURACY;
};
```

A specific GNSS transponder class can be derived from class `GNSS`. Therefore, the constructor `GNSS(double spatial_accuracy = _SPATIAL_ACCURACY, double temporal_accuracy = _TEMPORAL_ACCURACY)` is `protected`. It allows the initialization of the base class with spatial accuracy (default 0 m) and temporal inaccuracy (default 0 s). This is of outstanding importance since all transponders have certain uncertainties which the AOCS algorithm has to deal with. The spatial and temporal inaccuracies can be retrieved by using methods `_spatial_accuracy` and `_temporal_accuracy`

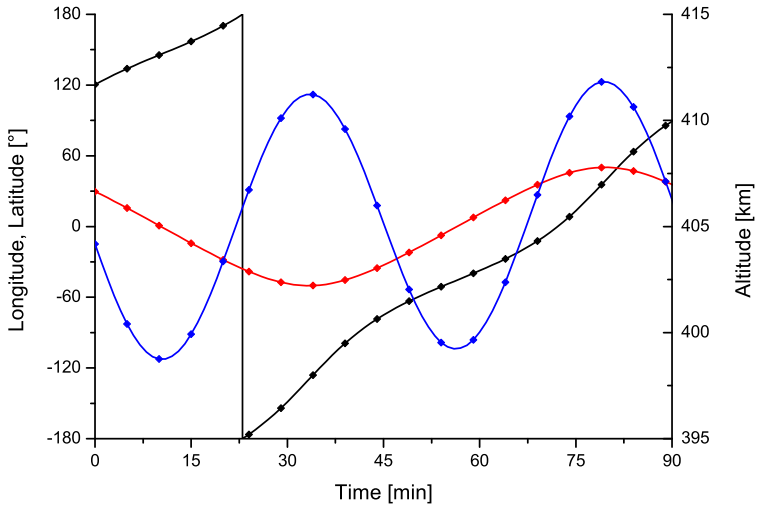

Fig. 9.1. Longitude (black curve), latitude (red curve) and altitude (blue curve) of the spacecraft and the GNSS transponder outputs (symbols).

respectively when no argument is passed. If the value needs to be updated, the new value is passed to these methods. The random number generator and normal distribution library of the C++ STL are used to generate representative values [230, p. 129 and 1185ff]. Typical commercially available GNSS transponders have position inaccuracies in the range from 1 to 10 m and velocity inaccuracies of 25 cm/s [172, 176].

Since class `GNSS` is derived from `System`, it also uses the `System` methods `enable` and `disable`. This way, it is possible to operate the GNSS transponder only for certain periods in time. This is commonly done to save power since the transponders have a comparatively high power draw (such as the NewSpace NGPS-01-422 with 1.5 W [172] or the NovAtel OEM615 with 1.2 W [176]).

As class `System` is derived from `List<System>::Item`, class `GNSS` needs to define method `clone`.

9.1.2 Verification

For the verification of `GNSS`, a spacecraft was put on a stable LEO orbit at 400 km altitude with the eccentricity $e = 0$, the argument of the periapsis $\omega = 120°$, the longitude of the ascending node $\Omega = 70°$, the inclination $i = 50°$ and the mean anomaly $M = 20°$.

The dual-frequency, multi-constellation GNSS receiver NovAtel OEM615 was used to provide location information with 1.5 m RMS (Root Mean Square) horizontal accuracy in the ECEF frame during the simulation. The time to the first fix is not considered in the simulation and could be implemented by deriving a new transponder from class `GNSS`.

The results are shown in fig. 9.1 for one orbit and one can recognize the good accordance of the measurements with the simulated location. In [148, p. 131], the equatorial and polar Earth radius are defined with 6.378137×10^6 m and 6.356752×10^6 mresulting in variation of the altitude of 21.385 km for a spacecraft orbiting the Earth on a 90° inclination orbit. With $i = 50°$, the chosen inclination is lower resulting in a lower variation of the altitude (blue curve) of 11.98 km.

For a more detailed view, the absolute spatial error was computed and shown in fig. 9.2 (black solid curve). The mean spatial error is also computed (black dotted line) matching the value in the specified data sheet of 1.5 m very well.

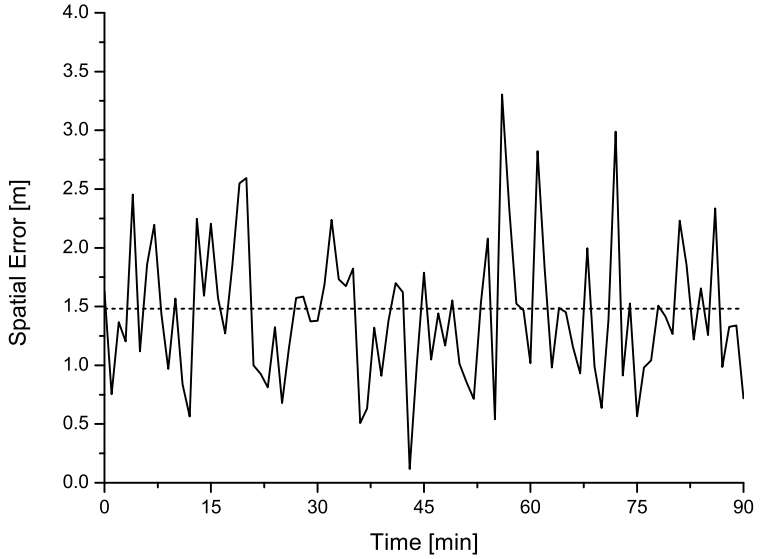

Fig. 9.2. Spatial error of the GNSS transponder for one orbit (black curve) and mean accuracy (dotted line).

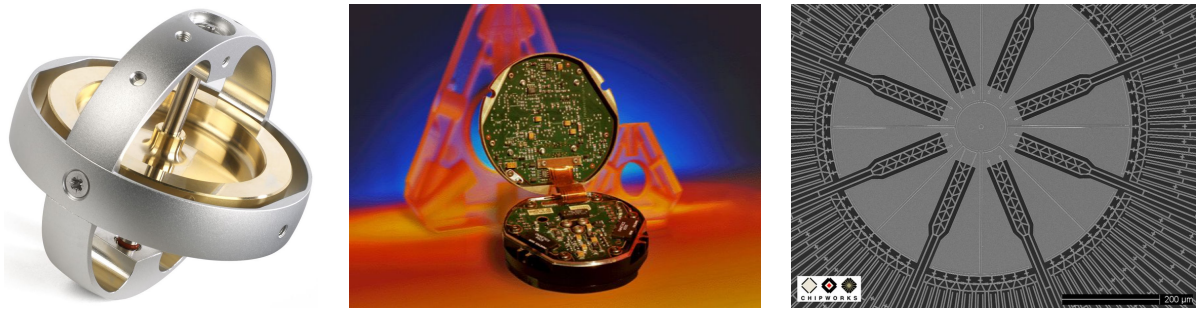

Fig. 9.3. Mechanical gyroscope from Brightfusion Ltd., digital ring laser gyroscope GG1320AN from Honeywell, detail view of a MEMS gyroscope as used in the Apple iPhone 4 photographed at Chipworks (from left to right).

9.2 Gyroscope

A gyroscope is a versatile device or sensor which is most commonly used to measure the angular velocity of an object, such as a spacecraft, around a certain body axis. If the axis of rotation is not fixed or if the angular velocity vector shall be acquired, three gyroscopes need to be used. Different measurement techniques can be used such as solid-state ring lasers, fiber optic-based interferometers or MEMS-based (Micro-Electro-Mechanical Systems) inertia sensors (see fig. 9.3). Due to their small size, the absence of moving parts, the low power consumption and their accuracy and performance increase in recent years [241, p. 4ff], MEMS-based sensors are most commonly used for CubeSats.

9.2.1 Implementation

The class `Gyroscope` represents a three-axis gyroscope to measure the rotation of the spacecraft. It is derived from `System` and defined as follows excluding certain private declarations.

```
// Class Gyroscope
class CubeSim::System::Gyroscope : public System
{
public:

   // Clone
   virtual System* clone(void) const;

   // Measure Spin Rate [rad/s]
   const Vector3D spin_rate(void) const;

protected:

   // Constructor
   Gyroscope(double accuracy = _ACCURACY, double range = _RANGE);

   // Copy Constructor (reset Part)
   Gyroscope(const Gyroscope& gyroscope);

   // Accuracy [rad/s]
   double _accuracy(void) const;
   void _accuracy(double accuracy);

   // Part
   Part* _part(void) const;
   void _part(Part& part);

   // Range [rad/s]
   double _range(void) const;
   void _range(double range);

private:

   // Default Accuracy [rad/s]
   static const double _ACCURACY;

   // Default Range [rad/s]
   static const double _RANGE;
};
```

A specific gyroscope class can be derived from class `Gyroscope`. Therefore, the constructor `Gyroscope(double accuracy = _ACCURACY, double range = _RANGE)` is `protected`. It allows the initialization of the base class with accuracy (default 0 rad/s) and measurement range (default ∞ rad/s). As a real gyroscope, the simulated gyroscope is defined with a certain accuracy and measurement range which allows better modeling of the sensor to be used.

The spin rate that the sensor experiences can be retrieved by the use of method `spin_rate` which does not expect any arguments. The range limitation and accuracy settings are implicitly considered (see below).

With the use of `_accuracy` the accuracy in all three axes can be altered also during the simulation by passing the new accuracy as an argument. It can be important to take into account exterior influences such as the changing ambient temperature or the EMI (Electromagnetic Interference) environment. When no argument is passed to method `_accuracy`, the actual accuracy is returned. The random number generator and normal distribution library of the C++ STL are used to generate representative values.

The measurement range R can be modified by the use of method `_range` called with the new measurement range. The actual range is obtained by calling the method without arguments. If the angular rate exceeds

the defined range $\omega_i \notin [-R; +R]$, the value is clipped component-wise. For the selection of a gyroscope, the trade-off between measurement range and accuracy needs to be done. If against the assumptions, the spin rate of the spacecraft, e.g., after deployment exceeds the measurement range, the ADCS algorithm needs to deal with these saturated values. In this case the spacecraft needs to be detumbled first so that the spin rate can accurately be measured by the gyroscope.

The method `_part` allows the definition of the physical body of the gyroscope. The part can be, e.g., a cylinder or a box with the corresponding device dimensions. The correlation not only allows to better simulate the total mass and mass distribution of the spacecraft, but is also important for the orientation of the sensor with respect to the spacecraft. At the begin of the simulation the relative orientation between the part and the spacecraft $\triangle\boldsymbol{R} = \boldsymbol{R}_{\text{sc}}^{\text{T}}\,\boldsymbol{R}_{\text{gyro}}$ in the spacecraft body frame is computed. For the later computation of the spin rate $\boldsymbol{\omega}$, the angular rate of the spacecraft $\boldsymbol{\omega}_{\text{sc}}$ needs to be transformed into the spacecraft body frame first considering the current spacecraft orientation $\boldsymbol{R}_{\text{sc}}$ in the global frame. Then the relative orientation is considered resulting in

$$\boldsymbol{\omega} = \triangle\boldsymbol{R}^{\text{T}}\,\boldsymbol{R}_{\text{sc}}^{\text{T}}\,\boldsymbol{\omega}_{\text{sc}} = \boldsymbol{R}_{\text{gyro}}^{\text{T}}\,\boldsymbol{\omega}_{\text{sc}}. \tag{207}$$

The intermediate step via $\triangle\boldsymbol{R}$ is done due to performance reasons since the computation of the orientation of the physical part representing the gyroscope would involve the repeated iteration through the systems hierarchy. This would result in negative implications on the overall performance. The part can be set by calling method `_part` and passing a reference to an object of type `Part`. The currently set part can be retrieved by calling the method without arguments. In that case a pointer is returned which can also be `nullptr` if no part has been set yet.

Since class `Gyroscope` is derived from `System`, it also uses methods `enable` and `disable`. This way, it is possible to simulate outages of the sensor either on purpose by modeling the power saving plan of the spacecraft or due to random malfunction, e.g., due to latch-up effects or EMI.

As class `System` is derived from `List<System>::Item`, class `Gyroscope` needs to define method `clone`.

9.2.2 Verification

The MEMS-based gyroscope L3G4200D by STMicroelectronics (now obsolete) was modeled in CubeSim according to the data sheet [228]. The range is limited to $\pm 4.36\,\text{rad/s}$ corresponding to $\pm 250\,^\circ/\text{s}$. The accuracy was set to $\pm 5 \times 10^{-4}\,\text{rad/s}$ which corresponds to the zero-rate level change with temperature $\pm 0.03\,^\circ/\text{s/K}$.

The CubeSim simulation from section 7.1 was used and the deviation between the angular rate of the spacecraft (in the body frame) and the reading of the gyroscope was observed over time as shown in fig. 7.1. One can see that the RMS value of the deviation is at the defined level of $\pm 5 \times 10^{-4}\,\text{rad/s}$.

9.3 Accelerometer

Accelerometers are sensors which allow the measurement of the linear acceleration or vibration in one up to three axes depending on their configuration. Mechanical accelerometers are commonly made of a test mass suspended by one or more springs. A movement of the mass can either be detected by displacement sensors (e.g. fiber-optic [184]) or by electromagnetic induction [51, p. 124]. The latter case is exploited in so-called *geophones* (see fig. 9.5, left) which are commonly used to detect seismic activities, such as earthquakes or tectonic plate shifts [41, p. 7ff].

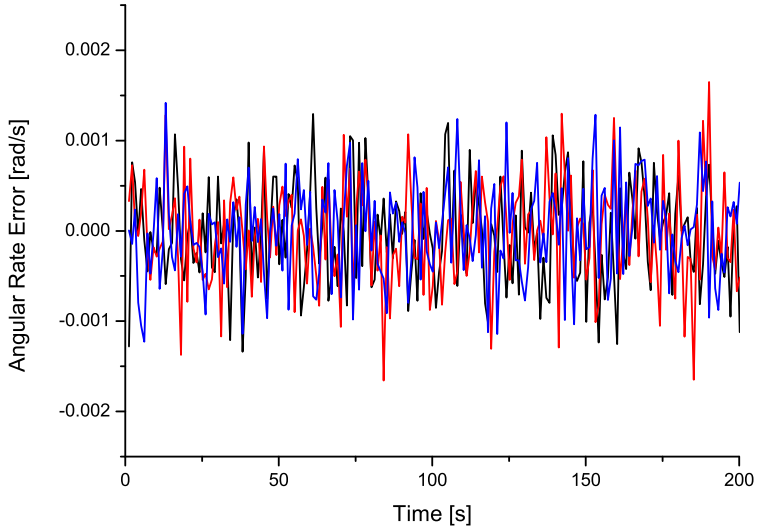

Fig. 9.4. Angular rate error as read out by the simulated gyroscope L3G4200D for all three axes: x (black), y (red), z (blue).

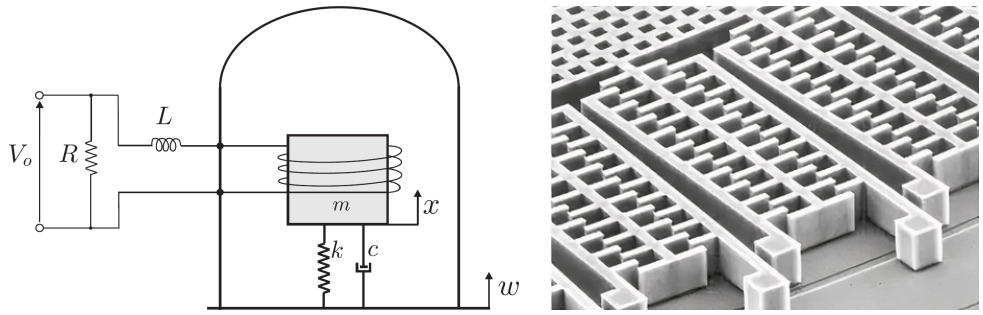

Fig. 9.5. Geophone using electromagnetic induction to detect movement of the test mass [41, p. 7] (left), MEMS-based accelerometer measuring the capacity between comb structures to detect displacement [79] (right).

In combination with gyroscopes, accelerometers are the most important sensors for inertial navigation for guided missile systems [25]. In contrast to short- and mid-range missiles, components in a spacecraft on a stable orbit around the Earth do not experience a net acceleration due to the equality of gravity and centripetal force. Therefore, accelerometers only show non-zero acceleration values $\boldsymbol{a}$ in case of (i) linear acceleration of the spacecraft due to the use of a propulsion system and (ii) spinning of the spacecraft. For the first case, if the thrust vector of the propulsion system passes through the center of mass of the spacecraft, the spacecraft experiences a linear acceleration

$$\boldsymbol{a} = \frac{1}{m}\boldsymbol{F} \tag{208}$$

with the thrust $\boldsymbol{F}$ and the spacecraft mass m. Sometimes the acceleration is not specified in the SI (Système international) units $\mathrm{m/s^2}$ but as multiples of the gravitation acceleration on Earth $g = 9.81\,\mathrm{m/s^2}$ [52, p. 26]. This results in the conversion $\boldsymbol{a}' = \boldsymbol{a}/g$.

In the second case, the spacecraft is not accelerated in linear direction, but is spinning around the center of mass. If the distance of the accelerometer from it is denoted with $\boldsymbol{r}$, the centripetal acceleration [52, p. 43] acting on it is

$$\boldsymbol{a}_c = -\boldsymbol{\omega}^2\boldsymbol{r}. \tag{209}$$

If the spacecraft is not on a stable orbit, the gravitational force originating from the Earth or from another celestial body is not compensated by inertia and the accelerometer detects the residue net force. The above considerations can also be generalized. It is assumed that the spacecraft follows a certain trajectory $\boldsymbol{x}_{\mathrm{sc}}(t)$ and spins around its center of mass with the angular rate $\boldsymbol{\omega}_{\mathrm{sc}}(t)$. Depending on the location of the accelerometer with respect to the center of mass of the spacecraft, the trajectory of the accelerometer can be described as a certain trajectory $\boldsymbol{x}(t)$ and a rotation around its own center of mass $\boldsymbol{\omega}(t)$. The latter is not detected by the accelerometer and therefore does not contribute to the measured signal. The acceleration acting on the sensor test mass is exclusively

$$\boldsymbol{a}(t) = \ddot{\boldsymbol{x}}(t) + \boldsymbol{a}_g(t) \tag{210}$$

with the acceleration $\boldsymbol{a}_g(t)$ caused by the sum of gravitational forces. To be able to compute the second term, module `Gravitation` is used (see section 4.1). For the first term, system `Accelerometer` computes the absolute position $\boldsymbol{x}_i = \boldsymbol{x}(-i\triangle t)$ of the sensor for certain time steps $t = -i\triangle t$ considering the position and rotation of the spacecraft. For the algorithm four points are used $0 \leqslant i \leqslant 3$. Then a third order polynomial is fitted between these points which results in the function

$$\boldsymbol{x}(t) = \boldsymbol{x}_0 + \frac{11\boldsymbol{x}_0 - 18\boldsymbol{x}_1 + 9\boldsymbol{x}_2 - 2\boldsymbol{x}_3}{6\triangle t}\, t + \frac{2\boldsymbol{x}_0 - 5\boldsymbol{x}_1 + 4\boldsymbol{x}_2 - \boldsymbol{x}_3}{2\triangle t^2}\, t^2 + \frac{\boldsymbol{x}_0 - 3\boldsymbol{x}_1 + 3\boldsymbol{x}_2 - \boldsymbol{x}_3}{6\triangle t^3}\, t^3. \tag{211}$$

Deriving $\boldsymbol{x}(t)$ two times after the time leads to

$$\ddot{\boldsymbol{x}}(t) = \frac{2\boldsymbol{x}_0 - 5\boldsymbol{x}_1 + 4\boldsymbol{x}_2 - \boldsymbol{x}_3}{\triangle t^2} + \frac{\boldsymbol{x}_0 - 3\boldsymbol{x}_1 + 3\boldsymbol{x}_2 - \boldsymbol{x}_3}{\triangle t^3}\, t \tag{212}$$

where the second term vanishes at $t = 0$ leading to

$$\ddot{\boldsymbol{x}}(0) = \frac{2\boldsymbol{x}_0 - 5\boldsymbol{x}_1 + 4\boldsymbol{x}_2 - \boldsymbol{x}_3}{\triangle t^2}. \tag{213}$$

9.3.1 Implementation

Similar to module `Motion` (see section 4.5), the acceleration can also be determined with little computational effort. The value is updated at each time step $\triangle t$ as implemented in method `_behavior` (see begin of this chapter for details).

The class `Accelerometer` represents a three-axis accelerometer to measure the linear acceleration of the spacecraft. It is derived from `System` and defined as follows excluding certain private declarations.

```
// Class Accelerometer
class CubeSim::System::Accelerometer : public System
{
public:

   // Measure Acceleration [m/s^2]
   const Vector3D acceleration(void) const;

   // Clone
   virtual System* clone(void) const;

   // Time Step [s]
   double time_step(void) const;
   void time_step(double time_step);

protected:
```

```
    // Constructor
    Accelerometer(double accuracy = _ACCURACY, double range = _RANGE,
        double time_step = _TIME_STEP);

    // Copy Constructor (reset Part)
    Accelerometer(const Accelerometer& accelerometer);

    // Accuracy [m/s^2]
    double _accuracy(void) const;
    void _accuracy(double accuracy);

    // Part
    Part* _part(void) const;
    void _part(Part& part);

    // Range [m/s^2]
    double _range(void) const;
    void _range(double range);

private:

    // Default Accuracy [m/s^2]
    static const double _ACCURACY;

    // Default Range [m/s^2]
    static const double _RANGE;

    // Default Time Step [s]
    static const double _TIME_STEP;

    // Behavior
    virtual void _behavior(void);
};
```

A specific accelerometer class can be derived from class `Accelerometer`. Therefore, the constructor `Accelerometer(double accuracy = _ACCURACY, double range = _RANGE, double time_step = _TIME_STEP)` is `protected`. Similar to class `Gyroscope`, it allows the initialization of the base class with the specified accuracy (default $0\,\mathrm{m/s^2}$), measurement range (default $\infty\,\mathrm{m/s^2}$) and time step (default $1\,\mathrm{s}$). As a real accelerometer, the simulated accelerometer is defined with a certain accuracy and measurement range values to allow better modeling of the sensor to be used.

The acceleration of the sensor is computed and returned by method `acceleration` which does not expect any arguments. The range limitation and accuracy settings are considered implicitly (see below).

With the use of method `_accuracy`, the accuracy in all three axes can be altered also during the simulation by passing the new accuracy as an argument. It can be important to take into account exterior influences such as the changing ambient temperature or the EMI environment. When no argument is passed to method `_accuracy`, the actual accuracy is returned. The random number generator and normal distribution library of the C++ STL are used to generate representative values.

The measurement range R can be modified by the use of method `_range` called with the new measurement range. The actual range is obtained by calling the method without arguments. If the linear acceleration exceeds the defined range $a_i \notin [-R; +R]$, the value is clipped component-wise. For the selection of a suitable accelerometer, the trade-off between measurement range and accuracy needs to be done. If against the assumptions, the acceleration exceeds the measurement range due to stronger acceleration or

higher spin rates of the spacecraft than expected, the ADCS algorithm can be checked if it can handle these circumstances.

Method `_part` allows the definition the physical body of the accelerometer. The part can be, e.g., a cylinder or a box with the corresponding device dimensions. The correlation not only allows to better simulate the total mass and mass distribution of the spacecraft, but it is also important for the orientation of the sensor with respect to the spacecraft. At the begin of the simulation the relative orientation between the part and the spacecraft $\triangle \boldsymbol{R} = \boldsymbol{R}_{\mathrm{sc}}^{\mathrm{T}} \boldsymbol{R}_{\mathrm{acc}}$ and the relative position of the sensor in the spacecraft $\triangle \boldsymbol{x} = \boldsymbol{x}_{\mathrm{sc}} - \boldsymbol{x}_{\mathrm{acc}}$ are computed in the spacecraft body frame. For the later computation of $\boldsymbol{x}_i = \boldsymbol{x}\left(-i\triangle t\right)$ (see begin of this section), the position and orientation of the spacecraft are considered to compute the absolute position of the sensor

$$\boldsymbol{x}_{\mathrm{acc}} = \boldsymbol{R}_{\mathrm{sc}} \triangle \boldsymbol{x} + \boldsymbol{x}_{\mathrm{sc}}, \ \boldsymbol{R}_{\mathrm{acc}} = \boldsymbol{R}_{\mathrm{sc}} \triangle \boldsymbol{R}. \tag{214}$$

The sensor value is computed in global frame and is then transformed into the sensor body frame.

The part can be set by calling method `_part` and passing a reference to an object of type `Part`. The currently set part can be retrieved by calling the method without arguments. In that case a pointer is returned which can also be `nullptr` if no part has been set yet.

The time step can also be controlled by calling method `time_step` and passing the new value. This can particularly be useful when fast movements or high spin rates of the spacecraft shall be simulated. The actual value can be obtained by calling the method without arguments.

Since class `Accelerometer` is derived from `System`, it also uses the `System` methods `enable` and `disable`. This way, it is possible to simulate outages of the sensor either on purpose by modeling the power saving plan of the spacecraft or due to random malfunction, e.g., due to latch-up effects or EMI.

As class `System` is derived from `List<System>::Item`, class `Accelerometer` needs to define method `clone`.

9.3.2 Verification

In the following simulation scenario a 2U CubeSat of 2 kg with the dimensions $10 \times 10 \times 20$ cm (x, y, z) and with one corner at the origin of the local frame is used. The corresponding center of mass is therefore at $x = 5$ cm, $y = 5$ cm and $z = 10$ cm. As explained in section 4.5, each free rigid body rotates around its center of mass which makes the absolute positioning in the local frame irrelevant for the simulation.

The accelerometer type ADXL330 manufactured by Analog Devices was modeled [10]. Currently, system `Accelerometer` only supports an absolute inaccuracy but, since in the data sheet the non-linearity is specified with 0.3% of full scale, an estimated accuracy value of $0.03\,\mathrm{m/s^2}$ is assumed for the $\pm 1\,\mathrm{g}$ measurement range. The accelerometer is placed at $x = 5$ cm, $y = 10$ cm and $z = 10$ cm with the same orientation as the satellite. The CubeSat is rotating around the z-axis at the rate of $\omega = 2\,\mathrm{rad/s}$ which corresponds to a comparatively high spin rate of ca. 19 revolutions per minute.

As shown in eq. (209), the centripetal acceleration depends on the angular rate ω and the distance from the axis of rotation r. In this scenario the axis of rotation is parallel to the z-axis and passes the center of mass of the spacecraft. This results in $r = 5$ cm and leads to the centripetal acceleration $a = -\omega^2 r = -0.2\,\mathrm{m/s^2}$.

In fig. 9.6 the output of the simulated accelerometer is shown over the course of one minute. One can clearly recognize that the off-axis components x and z are almost 0 and the signal of the y-axis shows the expected acceleration of $-0.2\,\mathrm{m/s^2}$ with the specified noise of $0.03\,\mathrm{m/s^2}$.

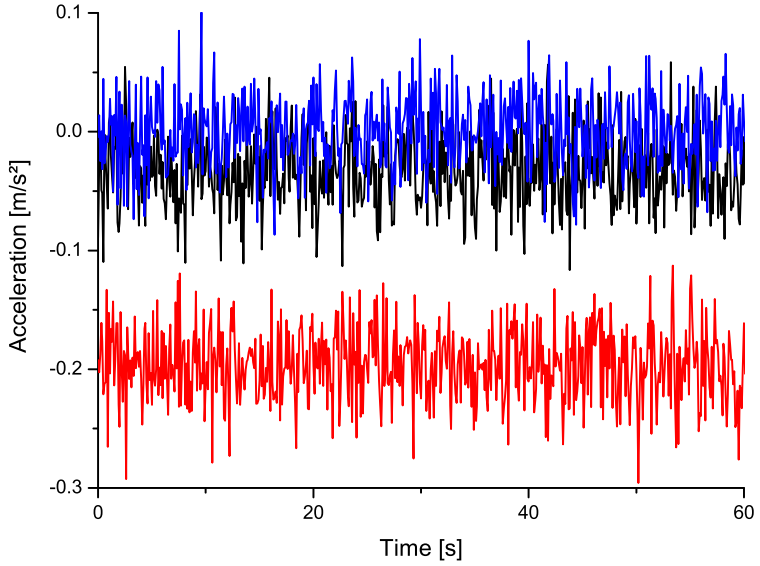

Fig. 9.6. Acceleration as read out by the simulated accelerometer ADXL330 when the spacecraft is spinning around the z-axis with 2 rad/s: x (black), y (red), z (blue).

For the next test, the same spacecraft model is used but this time the center of mass coincides with the origin of the local coordinate system. The same accelerometer is used and positioned also in the origin. A steady force of 200 mN along the z-axis through the center of mass of the spacecraft is applied. This comparatively strong force is rather optimistic for currently existing propulsion systems that can fit into a 2U envelope [131], [242] but is chosen intentionally for a better signal to noise ratio of the results. With the spacecraft mass of $m = 2\,\text{kg}$ and the applied force $F = 200\,\text{mN}$ the constant acceleration results in $a = F/m = 0.1\,\text{m/s}^2$ according to eq. (102).

In this simulation scenario the gravitational force from the Earth, the Sun or other celestial bodies is omitted. The spacecraft does not have any initial velocity. In fig. 9.7 the output of the simulated accelerometer is shown over the course of one minute. One can clearly recognize that the off-axis components x and y are almost 0 and the signal of the z-axis shows the expected acceleration of $0.1\,\text{m/s}^2$ with the specified noise of $0.03\,\text{m/s}^2$.

In the last test, the same spacecraft is used with the same accelerometer configuration, but this time it is put on a stable LEO orbit at 400 km altitude, eccentricity $e = 0.01$, argument of the periapsis $\omega = 120°$, longitude of the ascending node $\Omega = 70°$, inclination $i = 50°$ and mean anomaly $M = 20°$. For that purpose in additional to module `Motion`, the modules `Gravitation` and `Ephemeris` are used. The Sun and the Earth are added to the simulation. As the acting centripetal and gravitational force are of the same magnitude but in opposite direction, the spacecraft and its components do not experience any net force. This scenario is also called *free fall* in physics.

The output of the simulated accelerometer is computed over the course of one minute. Though the gravitational acceleration is approx. $8.7\,\text{m/s}^2$ at 400 km altitude, the resultant net force vanishes and all components show 0 with the specified noise of $0.03\,\text{m/s}^2$ as shown in fig. 9.8.

9.4 Photo Detector

A photo detector is a solid-state device that allows to convert light (photons) to electrical energy. The so-called *photoelectric effect* discovered by A. Einstein describes the minimum photon energy (work function) required to excite and free an electron. The number of electrons is proportional to the light intensity. A

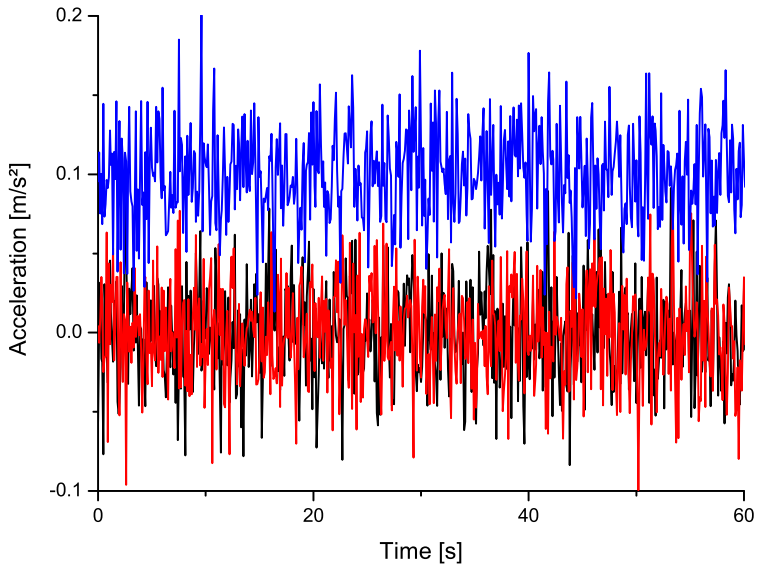

Fig. 9.7. Acceleration as read out by the simulated accelerometer ADXL330 when the spacecraft is accelerated uniformly in z direction with 200 mN: x (black), y (red), z (blue).

semiconductor photo detector can be made from different semiconductor materials with a characteristic band gap energy that determines the light-absorbing capabilities, such as the sensitive bandwidth or the quantum efficiency. More details on semiconductor photo detectors can be found in [21, p. 662] and other types of light sensors, such as thermal detectors, are described in [194, p. 545ff]. Typical SMD (Surface-Mounted Device) and through-hole photo detectors with different sensitive bandwidths are shown in fig. 9.9.

The sensor features a certain sensitive area A and collects incident light within a certain opening angle Ω (cone around the perpendicular axis). With the angle between the axis and the direction to the light source or ray of light φ, the radiant flux is computed

$$\Phi = A \cos\varphi \left(\sum I_{\mathrm{l},i} + \sum I_{\mathrm{a},i} \right) \tag{215}$$

with the accumulated irradiance from light sources $I_{\mathrm{l},i}$ and the albedo sources $I_{\mathrm{a},i}$. See sections 4.2 and 4.3 for details on the computation of the resultant incident light at a certain detection point. There, the limited opening angle Ω is also considered which allows accurate simulation of the limitation of incoming light for specific sensor types. It has to be noted that the radiant flux has the unit of power. Any additional conversion to the photo current or modeling of a linear behavior need to be performed a custom class derived from class `Photodetector`.

9.4.1 Implementation

The class `Photodetector` represents a photo sensor which provides a linear output in dependence of the illumination level. Its output value relies on one or more modules of type `Light` (see section 4.2) and `Albedo` (see section 4.3). Though the Sun and Earth albedo are the predominant sources of light for most simulation scenarios, class `Photodetector` considers also the light from other stars and the albedos from other celestial bodies, if applicable. It is derived from `System` and defined as follows excluding certain private declarations.

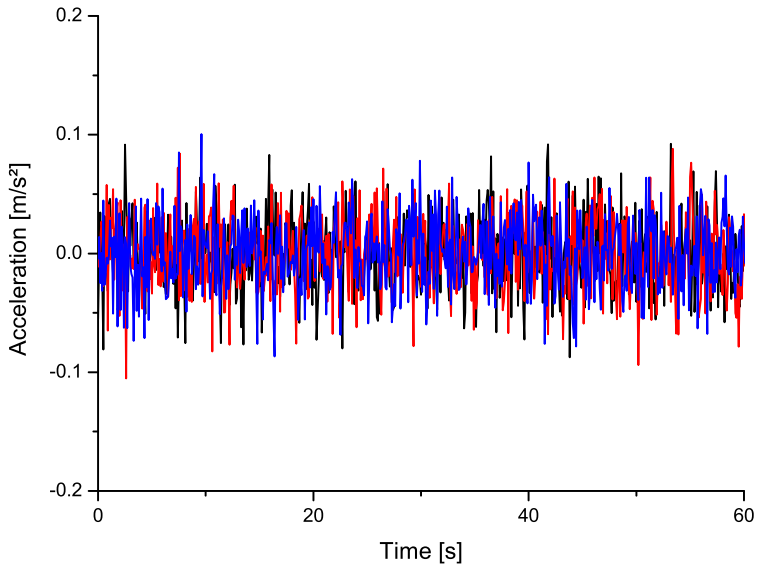

Fig. 9.8. Acceleration as read out by the simulated accelerometer ADXL330 when the spacecraft is on a stable LEO orbit: x (black), y (red), z (blue).

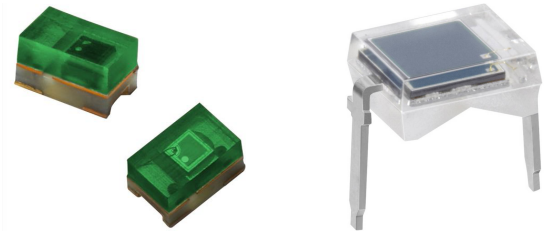

Fig. 9.9. SMD ambient light sensor by Vishay Semiconductors (left) and wide-bandwidth through-hole photo diode by OSRAM (right).

```
// Class Photodetector
class CubeSim::System::Photodetector : public System
{
public:

   // Clone
   virtual System* clone(void) const;

   // Measure Radiant Flux [W]
   double radiant_flux(void) const;

protected:

   // Constructor
   Photodetector(double area, double angle = _ANGLE, double accuracy =
      _ACCURACY, double range = _RANGE);

   // Copy Constructor
   Photodetector(const Photodetector& photodetector);

   // Accuracy [W/m^2]
   double _accuracy(void) const;
   void _accuracy(double accuracy);
```

```
    // Opening Angle [rad]
    double _angle(void) const;
    void _angle(double angle);

    // Effective Sensor Area [m^2]
    double _area(void) const;
    void _area(double area);

    // Part
    Part* _part(void) const;
    void _part(Part& part);

    // Range [W/m^2]
    double _range(void) const;
    void _range(double range);

private:

    // Default Accuracy [W/m^2]
    static const double _ACCURACY;

    // Default opening Angle [rad]
    static const double _ANGLE;

    // Default Range [W/m^2]
    static const double _RANGE;
};
```

A specific photo detector class can be derived from class `Photodetector`. For that purpose, the constructor `Photodetector(double area, double angle = _ANGLE, double accuracy = _ACCURACY, double range = _RANGE)` is defined `protected`. It allows the initialization of the base class with the active sensor area, the opening angle (also known as *field of view*, default π rad), the accuracy (default $0\,\mathrm{m/W^2}$) and the measurement range (default $\infty\,\mathrm{m/W^2}$). As a real photo detector, the simulated photo detector is defined with a certain accuracy and measurement range which allows better modeling of the sensor to be used.

The radiant flux resulting from modules `Light` and `Albedo` is computed with method `radiant_flux` which does not expect any arguments. The range limitation and accuracy settings are implicitly considered (see below).

With the use of `_accuracy` the accuracy can be altered also during the simulation by passing the new accuracy as an argument. It can be important to take into account exterior influences such as the changing ambient temperature or the EMI environment. When no argument is passed to method `_accuracy`, the actual accuracy is returned. The random number generator and normal distribution library of the C++ STL are used to generate representative values.

The measurement range R can be modified by the use of method `_range` called with the new measurement range. The actual range is obtained by calling the method without arguments. If the radiant flux exceeds the defined range $\Phi \notin [0; +R]$, the value is clipped. For the selection of a photo detector, the trade-off between measurement range and accuracy needs to be done. If against the assumptions, the radiant flux exceeds the measurement range if, e.g., the Earth albedo was not considered, the ADCS algorithm has to deal with saturated values.

The active sensor are can be retrieved by calling method `_area` without arguments. The area can be updated by passing the new value to the method. The field of view of the photo detector can be changed

by passing the new angle $0 < \gamma \leqslant \pi$ to method `_angle`. The actual angle is obtained by calling the method without arguments.

Method `_part` allows the definition the physical body of the photo detector. The part can, e.g., be a cylinder or a box with the corresponding device dimensions. The correlation not only allows to better simulate the total mass and mass distribution of the spacecraft, but is also important for the orientation of the sensor with respect to the spacecraft. At the begin of the simulation the relative orientation between the part and the spacecraft $\triangle\boldsymbol{R} = \boldsymbol{R}_{\mathrm{sc}}^{\mathrm{T}}\,\boldsymbol{R}_{\mathrm{pd}}$ in the spacecraft body frame is computed. For the later computation of Φ, the orientation of the photo detector (z-axis) needs to be computed in the global frame

$$\boldsymbol{R}_{\mathrm{pd}} = \boldsymbol{R}_{\mathrm{sc}}\,\triangle\boldsymbol{R}, \quad \boldsymbol{e}_{\mathrm{z}} = \boldsymbol{R}_{\mathrm{pd}}\,\boldsymbol{e}_{\mathrm{z,pd}} \tag{216}$$

with the z-axis base vector $\boldsymbol{e}_{\mathrm{z}}$. The part can be set by calling method `_part` and passing a reference to an object of type `Part`. The currently set part can be retrieved by calling the method without arguments. In that case a pointer is returned which can also be `nullptr` if no part has been set yet.

Since class `Photodetector` is derived from `System`, it also uses the `System` methods `enable` and `disable`. This way, it is possible to simulate outages of the sensor either on purpose by modeling the power saving plan of the spacecraft or due to random malfunction, e.g., due to latch-up effects or EMI.

As class `System` is derived from `List<System>::Item`, class `Photodetector` needs to define method `clone`.

9.4.2 Verification

In the following simulation scenario a 2U CubeSat of 2 kg with the dimensions $10 \times 10 \times 20$ cm (x, y, z) is used. The photo detector type TEMD6200FX01 manufactured by Vishay Semiconductors was modeled [246]. It is a highly sensitive PIN (Positive Intrinsic Negative) photo diode in an SMD package and its spectral sensitivity corresponds to that of the human eye. The spectral bandwidth ranges from 430 to 610 nm and the sensitivity is rather uniform - the angle of half sensitivity is 60° following the cosine law. In the data sheet, the illuminance is specified up to 10^4 lx which corresponds to 83.35 W/m^2 for sun light [145]. Experiments with solar simulators have shown that the photo diode features a linear increase of the photo current with the irradiance (4.80×10^{-8} A m^2/W) up to the required irradiance of 1400 W/m^2.

Currently, system `Photodetector` only supports an absolute inaccuracy. For the photo diode and the required measurement circuitry, an absolute error in current of 1 μA is assumed which corresponds to 20 W/m^2 irradiance. Six of the aforementioned photo detectors are being used - one at each side - to allow determination of the sun direction.

The CubeSat is put on a stable circular LEO orbit at 400 km altitude with an angular rate of $\boldsymbol{\omega}_0 = (1 \;\; 2 \;\; -1)^{\mathrm{T}}$ mrad/s. An inclination of $\gamma = 19°$ from the sun-synchronous orbital plane is chosen so that the satellite never enters eclipse (see section 4.2.2 for details on the orbit configuration).

For the verification of the photo detectors, the measured irradiance values are used to compute the direction to the sun in spherical coordinates (spacecraft body frame). This is compared with the simulated results as shown in fig. 9.10. The longitude or azimuthal angle (black curve) varies between −180 and +180° with three wrap-arounds and the latitude, elevation or copolar angle (red curve) varies between −38.6 and +18.3° [38, p. 4]. The simulation was run two times and the sun direction was computed from the individual irradiance values provided by the photo detectors, one with (blue and green symbols) and one without (black and red symbols) module `Albedo`. This allows estimation of the introduced systematic pointing error when the ADCS system would not consider the Earth albedo.

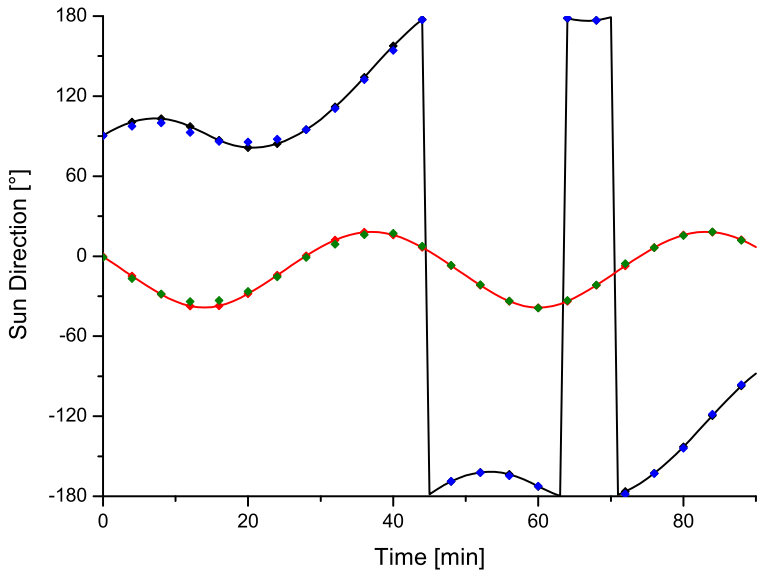

Fig. 9.10. Sun direction in spherical coordinates: longitude (black curve), latitude (red curve); measured direction without (black and red symbols) and with Earth albedo (blue and green symbols).

The central angle γ between two points on a sphere, also known as the great-circle distance, is computed as follows

$$\cos\gamma = \sin\phi_1 \, \sin\phi_2 + \cos\phi_1 \, \cos\phi_2 \, \cos(\theta_1 - \theta_2) \tag{217}$$

with the longitudes of both points θ_1 and θ_2 and the latitudes ϕ_1 and ϕ_2. This equation results from conversion into Cartesian coordinates and application of the law of cosines. Details on spherical trigonometry can be found in [252, p. 205ff].

Without the influence of Earth albedo, the measurements only show a small deviation (max. 2.1°) and with simulated albedo, the deviation increases to 4.9°. The results are shown in fig. 9.11 and it is obvious that only when the spacecraft is passing the sun-facing side of the Earth, the albedo compromises the measurement.

9.5 Magnetometer

Magnetometers are sensors that allow the measurement of the local magnetic flux density. While some types only allow to determine the magnitude, others also allow the determination of the direction. There are various techniques to detect the direction of the magnetic field or, in particular, the magnetic field lines. A compass is a magnetic dipole that aligns anti-parallel to the local magnetic field lines like those from the Earth's magnetic field. A very common type of magnetometers are Hall sensors which exploit the *Hall effect* to determine the magnetic flux density [51, p. 99]. The so-called *Lorentz force* $\boldsymbol{F}$ is acting on a charge moving through a magnetic field

$$\boldsymbol{F} = q\,\boldsymbol{v} \times \boldsymbol{B} \tag{218}$$

with the charge q, the velocity $\boldsymbol{v}$ and the magnetic flux density $\boldsymbol{B}$ [51, p. 95]. Similarly, electrons moving through a conductor are deflected sidewards when the conductor is exposed to a magnetic field. This leads to a change of the electron distribution across the cross section area of the conductor and thus to a measurable voltage, the so-called *Hall voltage* (see fig. 9.12).

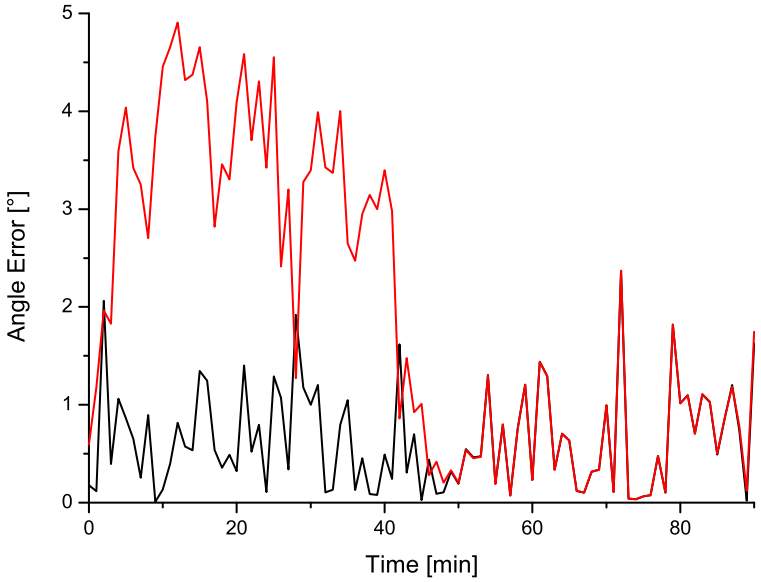

Fig. 9.11. Absolute pointing error to the sun with (red) and without simulated Earth albedo (black).

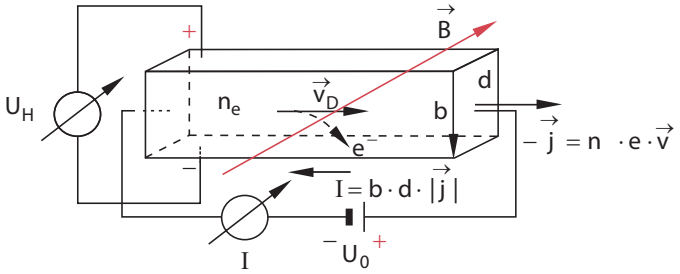

Fig. 9.12. Emerging electric field when a current-carrying conductor is exposed to a magnetic field [51, p. 99].

MEMS-based sensors commonly use the Lorentz force to provoke a movement of a cantilever or deflection of flat spring shaped structures. Capacitive sensing methods are used to determine the deflection and thus the applied magnetic field [127, p. 3].

It is also possible to exploit other physical effects, such as the GMR (Giant Magneto-Resistance) or the AMR (Anisotropic Magneto-Resistance) effect, to detect the strength or the direction of the magnetic field. The electrical resistance significantly changes with the magnetic flux density due to the magnetization of the material and the spin-orbit interaction [64, p. 2ff], [122, p. 1f].

9.5.1 Implementation

The class `Magnetometer` represents a three-axis magnetometer to measure the magnetic field at the sensor location. It relies on one or more modules of type `Magnetics` (see section 4.4). Though the Earth is the predominant source of the magnetic field for most simulation scenarios, class `Magnetometer` considers also the magnetic fields of other celestial bodies, if applicable. It is derived from `System` and defined as follows excluding certain private declarations.

```
// Class Magnetometer
class CubeSim::System::Magnetometer : public System
{
public:

   // Clone
   virtual System* clone(void) const;

   // Measure magnetic Field [T]
   const Vector3D magnetic_field(void) const;

protected:

   // Constructor
   Magnetometer(double accuracy = _ACCURACY, double range = _RANGE);

   // Copy Constructor (reset Part)
   Magnetometer(const Magnetometer& magnetometer);

   // Accuracy [T]
   double _accuracy(void) const;
   void _accuracy(double accuracy);

   // Part
   Part* _part(void) const;
   void _part(Part& part);

   // Range [T]
   double _range(void) const;
   void _range(double range);

private:

   // Default Accuracy [T]
   static const double _ACCURACY;

   // Default Range [T]
   static const double _RANGE;
};
```

A specific magnetometer class can be derived from class `Magnetometer`. Therefore, the constructor `Magnetometer(double accuracy = _ACCURACY, double range = _RANGE)` is `protected`. Similar to class `Gyroscope`, it allows the initialization of the base class with accuracy (default 0 T) and measurement range (default ∞ T). As a real magnetometer, the simulated magnetometer is defined with a certain accuracy and measurement range which allows better modeling of the sensor to be used.

The magnetic field at the sensor position resulting from the modules of type `Magnetics` is computed with method `magnetic_field` which does not expect any arguments. The range limitation and accuracy settings are implicitly considered (see below).

With the use of `_accuracy` the accuracy in all three axes can be altered also during the simulation by passing the new accuracy as an argument. It can be important to take into account exterior influences such as the changing ambient temperature or other sources of magnetic fields inside the spacecraft such as power distribution lines. When no argument is passed to method `_accuracy`, the actual accuracy is returned. The random number generator and normal distribution library of the C++ STL are used to generate representative values.

The measurement range R can be modified by the use of method `_range` called with the new measurement range. The actual range is obtained by calling the method without arguments. If the magnetic field exceeds the defined range $B_i \notin [-R; +R]$, the value is clipped component-wise. For the selection of a suitable magnetometer, the trade-off between measurement range and accuracy needs to be done.

Method `_part` allows the definition the physical body of the magnetometer. The part can, e.g., be a cylinder or a box with the corresponding device dimensions. The correlation not only allows to better simulate the total mass and mass distribution of the spacecraft, but is also important for the orientation of the sensor with respect to the spacecraft. At the begin of the simulation the relative orientation between the part and the spacecraft $\triangle\boldsymbol{R} = \boldsymbol{R}_{\mathrm{sc}}^{\mathrm{T}}\,\boldsymbol{R}_{\mathrm{mag}}$ is calculated. For the later computation of the magnetic field, the orientation of the spacecraft is considered to determine the orientation of the sensor

$$\boldsymbol{R}_{\mathrm{mag}} = \boldsymbol{R}_{\mathrm{sc}}\,\triangle\boldsymbol{R}. \tag{219}$$

For the later computation of $\boldsymbol{B}_{\mathrm{mag}}$ in the magnetometer body frame, the magnetic field at the spacecraft position in the global frame needs to be transformed into the spacecraft body frame first with the current spacecraft orientation $\boldsymbol{R}_{\mathrm{sc}}$ and then into the body frame of the magnetometer

$$\boldsymbol{B}_{\mathrm{mag}} = \triangle\boldsymbol{R}^{\mathrm{T}}\,\boldsymbol{R}_{\mathrm{sc}}^{\mathrm{T}}\,\boldsymbol{B} = \boldsymbol{R}_{\mathrm{mag}}^{\mathrm{T}}\,\boldsymbol{B}. \tag{220}$$

The part can be set by calling method `_part` and passing a reference to an object of type `Part`. The currently set part can be retrieved by calling the method without arguments. In that case a pointer is returned which can also be `nullptr` if no part has been set yet.

Since class `Magnetometer` is derived from `System`, it also uses the `System` methods `enable` and `disable`. This way, it is possible to simulate outages of the sensor either on purpose by modeling the power saving plan of the spacecraft or due to random malfunction, e.g., due to latch-up effects or EMI.

As class `System` is derived from `List<System>::Item`, class `Magnetometer` needs to define method `clone`.

9.5.2 Verification

In the following simulation scenario a 2U CubeSat of 2 kg with the dimensions $10 \times 10 \times 20$ cm (x, y, z) is used. The magnetometer type HMC5983 manufactured by Honeywell was modeled [90]. It is a 3-axis magnetometer based on the aforementioned AMR effect with a field range from -8×10^{-4} to $+8 \times 10^{-4}$ T. Currently, system `Magnetometer` only supports an absolute inaccuracy but, since in the data sheet the noise floor is specified with 2×10^{-7} T, this value is being used. The accelerometer is modeled with the same orientation as the satellite.

The CubeSat is put on a stable LEO orbit at 400 km altitude, eccentricity $e = 0.01$, argument of the periapsis $\omega = 120°$, longitude of the ascending node $\Omega = 70°$, inclination $i = 50°$ and mean anomaly $M = 20°$. For the verification of the magnetometer, the local magnetic field at the satellite position is compared with the HMC5983 reading.

In fig. 9.13 the output of the simulated magnetometer is shown over the course of an entire orbit (ca. 92 minutes). One can recognize the good correlation between the computed magnetic field at the spacecraft position using module `Magnetics` and the sensor output.

In addition, the deviation between the magnitude of the simulated magnetic field and the reading of the magnetometer was computed which is shown in fig. 9.14. One can see that the RMS value of the deviation is at the defined level of $\pm 2 \times 10^{-7}$ T.

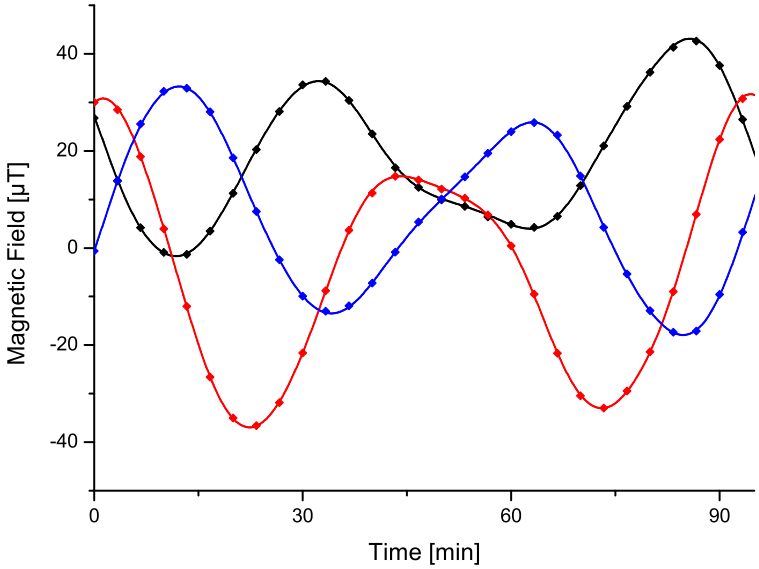

Fig. 9.13. Simulated magnetic field (lines) and HMC5983 magnetometer output (symbols) when the spacecraft is on a stable LEO orbit: x (black), y (red), z (blue).

9.6 Magnetorquer

A magnetic torquer or magnetorquer comprises of one or more electromagnetic coils with adjustable current. It is an actuator and is used to detumble, stabilize or control the attitude of a satellite. Typically, solenoids with long ferrite cores are used (see fig. 9.15) but it also possible to directly integrate flat air coils into the PCB (Printed Circuit Board) of, e.g., a sub-system or a side panel as it was done for satellite PEGASUS [203]. Information on proper dimensioning and performance estimations can be found here [8].

A flat air coil with an effective area per turn A, number of turns N and applied current I generates the magnetic moment

$$\boldsymbol{p}_\mathrm{m} = N\,I\,\boldsymbol{A} \tag{221}$$

where the vector $\boldsymbol{A}$ is perpendicular to the area A with $\|\boldsymbol{A}\| = A$. In the presence of a magnetic field $\boldsymbol{B}$, the torque

$$\boldsymbol{\tau} = \boldsymbol{p}_\mathrm{m} \times \boldsymbol{B} = N\,I\,\boldsymbol{A} \times \boldsymbol{B} \tag{222}$$

is generated [51, p. 107]. Now a solenoid with the length l, the effective area per turn A, number of turns N and the applied current I is considered. In contrast to the flat air coil, a core with relative magnetic permeability μ_r is considered. Though the magnetic moment $\boldsymbol{p}_\mathrm{m}$ of an air core solenoid is identical to the flat air coil, the contribution of the magnetized core needs to be considered. The magnetic field strength of an idealized solenoid is

$$\boldsymbol{H} = \frac{N\,I}{l}\frac{\boldsymbol{A}}{A}. \tag{223}$$

The magnetic flux density in the core is therefore [51, p. 88 and 108]

$$\boldsymbol{B} = \mu_0\,(\boldsymbol{H} + \boldsymbol{M}) = \mu_0\,\mu_\mathrm{r}\,\boldsymbol{H} \tag{224}$$

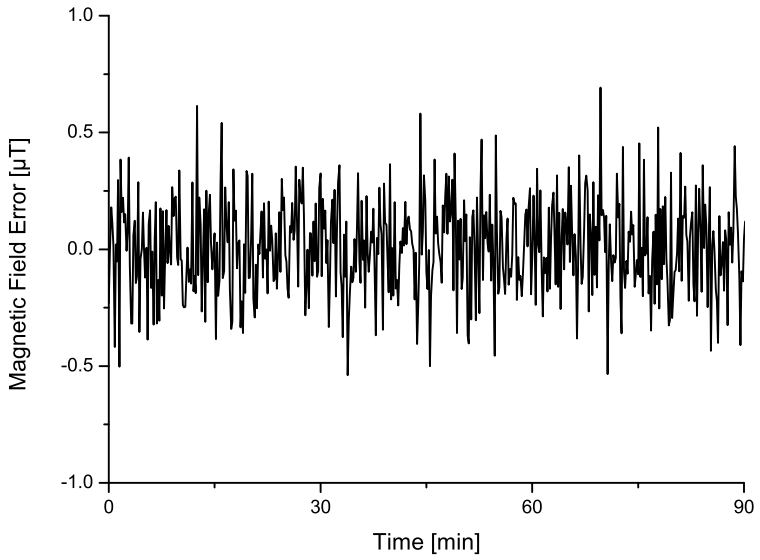

Fig. 9.14. Magnetic field error (magnitude) as read out by the simulated magnetometer HMC5983.

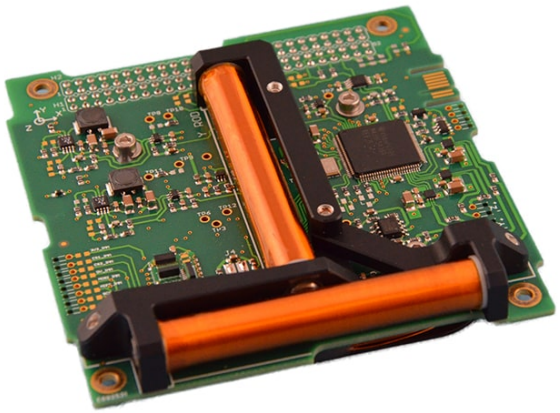

Fig. 9.15. Three-axis magnetorquer board from ISISPACE with two solenoids and a flat air coil on the rear side (not visible).

with the magnetization $\boldsymbol{M} = \chi \boldsymbol{H}$ and the magnetic susceptibility $\chi = \mu_\mathrm{r} - 1$. The magnetization representing the volumetric magnetic moment density is therefore

$$\boldsymbol{M} = \chi \boldsymbol{H} = (\mu_\mathrm{r} - 1)\,\frac{N\,I}{l}\,\frac{\boldsymbol{A}}{A} \tag{225}$$

and the total magnetic moment is consequently

$$\boldsymbol{p}_\mathrm{m} = N\,I\,\boldsymbol{A} + \boldsymbol{M}\,A\,l = \mu_\mathrm{r}\,N\,I\,\boldsymbol{A}. \tag{226}$$

9.6.1 Implementation

The class `Magnetorquer` represents a single-axis magnetorquer to generate a torque in the presence of an external magnetic field. It relies on one or more modules of type `Magnetics` (see section 4.4). Though the Earth is the predominant source of the magnetic field for most simulation scenarios, class `Magnetorquer`

considers also the magnetic fields of other celestial bodies, if applicable. It is derived from `System` and defined as follows excluding certain private declarations.

```
// Class Magnetorquer
class CubeSim::System::Magnetorquer : public System
{
public:

    // Clone
    virtual System* clone(void) const;

    // Current [A]
    double current(void) const;
    void current(double current);

    // Time Step [s]
    double time_step(void) const;
    void time_step(double time_step);

protected:

    // Constructor
    Magnetorquer(double area, double range = _RANGE, double accuracy =
        _ACCURACY, double time_step = _TIME_STEP);

    // Copy Constructor (reset Part)
    Magnetorquer(const Magnetorquer& magnetorquer);

    // Accuracy [A]
    double _accuracy(void) const;
    void _accuracy(double accuracy);

    // Effective Coil Area [m^2]
    double _area(void) const;
    void _area(double area);

    // Part
    Part* _part(void) const;
    void _part(Part& part);

    // Relative magnetic Permeability
    double _permeability(void) const;
    void _permeability(double permeability);

    // Range [A]
    double _range(void) const;
    void _range(double range);

private:

    // Default Accuracy [A]
    static const double _ACCURACY;

    // Default Range [A]
    static const double _RANGE;

    // Default Time Step [s]
    static const double _TIME_STEP;
```

```
    // Behavior
    virtual void _behavior(void);
};
```

A specific magnetorquer class can be derived from class `Magnetorquer`. For that purpose, the constructor `Magnetorquer(double area, double range = _RANGE, double accuracy = _ACCURACY, double time_step = _TIME_STEP)` is defined `protected`. Similar to aforementioned sensor classes, it allows the initialization of the base class with the range (default ∞ A), the accuracy (default 0 A) and the time step (default 1 s). As a real magnetorquer, the simulated magnetorquer is defined with a certain accuracy considering the number of turns, the enclosed area, the number of turns and the current. The current range allows better modeling of the actuator to be used.

For operation, the current through the coil can be set by calling method `current` and passing the new current value. Method `current` returns the actually set current if being called without arguments.

With the use of method `_accuracy` the accuracy can be altered also during the simulation by passing the new accuracy as an argument. When no argument is passed to the method, the actual accuracy is returned. The random number generator and normal distribution library of the C++ STL are used to generate representative values.

The current range R can be modified by the use of method `_range` called with the new range. The actual range is obtained by calling the method without arguments. If the commanded current exceeds the defined range $I \notin [-R; +R]$, the value is clipped. For the selection of a suitable magnetorquer, the trade-off between torque range and control resolution needs to be done. If the ADCS algorithm attempts to command a current that exceeds the allowable range, the physical behavior of the simulated magnetorquer is still accurate.

The effective coil area can be controlled by calling method `area` and passing the new value. As this value typically does not change during operation, this method is usually only called in the constructor of the class derived from class `Magnetorquer`. The actual value can be obtained by calling the method without arguments.

Depending on the construction of the coil (either as flat air coil or as solenoid with a ferromagnetic core), the relative magnetic permeability needs to be set accordingly which can be done by calling method `permeability` and passing the new unit-less value. The actual value is returned by the method if being without arguments.

Method `_part` allows the definition the physical body of the magnetorquer. The part can, e.g., be a cylinder or a box with the corresponding device dimensions. The correlation not only allows to better simulate the total mass and mass distribution of the spacecraft, but is also important for the orientation of the actuator with respect to the spacecraft. The currently set object can be retrieved by calling the method without arguments. In that case a pointer is returned which can also be `nullptr` if no part has been set yet.

At the begin of the simulation the relative orientation between the part and the spacecraft $\triangle \boldsymbol{R} = \boldsymbol{R}_{\mathrm{sc}}^{\mathrm{T}} \boldsymbol{R}_{\mathrm{mtq}}$ is calculated. For the later computation of the magnetic field in the magnetorquer body frame, the orientation of the spacecraft is considered

$$\boldsymbol{R}_{\mathrm{mtq}} = \boldsymbol{R}_{\mathrm{sc}} \triangle \boldsymbol{R}. \tag{227}$$

Furthermore, a named torque `"Magnetorquer"` of class `Torque` is inserted into the specified part which is later controlled in dependence of the ambient magnetic field and the set current passing through the coil. During the simulation, similar to class `Magnetometer`, the magnetic field $\boldsymbol{B}_{\mathrm{mtq}}$ in the magnetorquer body frame is computed

$$\boldsymbol{B}_{\mathrm{mtq}} = \triangle\boldsymbol{R}^{\mathrm{T}}\,\boldsymbol{R}_{\mathrm{sc}}^{\mathrm{T}}\,\boldsymbol{B} = \boldsymbol{R}_{\mathrm{mtq}}^{\mathrm{T}}\,\boldsymbol{B}. \tag{228}$$

It is assumed that $\boldsymbol{e}_z$ is perpendicular to the coil area. The torque in the body frame results therefore in

$$\boldsymbol{\tau} = \mu_{\mathrm{r}}\,A\,I\,\boldsymbol{e}_z \times \boldsymbol{B}_{\mathrm{mtq}} \tag{229}$$

which is periodically updated at each time step implicitly considering the defined accuracy settings.

The time step can be controlled by calling method `time_step` and passing the new value. This can particularly be useful when fast movements or high spin rates of the spacecraft shall be simulated. The actual value can be obtained by calling the method without arguments.

Since class `Magnetorquer` is derived from `System`, it also uses the `System` methods `enable` and `disable`. This way, it is possible to simulate outages of the actuator either on purpose by modeling the power saving plan of the spacecraft or due to random malfunction, e.g., due to latch-up effects or EMI.

As class `System` is derived from `List<System>::Item`, class `Magnetorquer` needs to define method `clone`.

9.6.2 Verification

For the verification of the magnetorquer, a simulation was set up including a 1U CubeSat of 1 kg with the dimensions $10 \times 10 \times 10$ cm (x, y, z). It is equipped with three orthogonal magnetorquers with an effective coil area of 0.38 m^2 each and a controllable current between -1 and $+1$ A. Again, the three-axis magnetometer type HMC5983 is used to measure the Earth magnetic field.

The CubeSat is put on a stable LEO orbit at 400 km altitude, eccentricity $e = 0.01$, argument of the periapsis $\omega = 120°$, longitude of the ascending node $\Omega = 70°$, inclination $i = 50°$ and mean anomaly $M = 20°$. The initial spin rate is set to $\boldsymbol{\omega}_0 = (0\ \ 1\ \ 2)^{\mathrm{T}}$ rad/s.

The so-called *B-dot detumbling controller* [54] is applied to detumble the spacecraft before fine-pointing ADCS algorithms could take over. A magnetic moment is generated which allows to spin down the spacecraft [53, p. 39], [73, p. 3]

$$\boldsymbol{p}_{\mathrm{m}} = -\frac{k}{\|\boldsymbol{B}\|}\dot{\boldsymbol{B}} \tag{230}$$

with the magnitude of the magnetic moment $\|\boldsymbol{p}_{\mathrm{m}}\| = \mu_{\mathrm{r}}\,N\,I\,A$, the local magnetic flux density $\boldsymbol{B}$, its derivative $\dot{\boldsymbol{B}}$ and a proportional factor which can either be kept constant or be tuned in accordance with orbital parameters [16]. The optimal torque to efficiently spin down the spacecraft can be approximated as $\boldsymbol{\tau} \sim -\boldsymbol{\omega}$ but as described in eq. (229), by varying the magnetic moment $\boldsymbol{p}_{\mathrm{m}}$, only the torque $\boldsymbol{\tau} = \boldsymbol{p}_{\mathrm{m}} \times \boldsymbol{B}$ can be generated. It would be optimal when $\boldsymbol{\omega} \perp \boldsymbol{B}$, in which case the resulting magnetic moment would also be perpendicular to both, $\boldsymbol{\omega}$ and $\boldsymbol{B}$. Even in the general case when $\boldsymbol{\omega} \not\perp \boldsymbol{B}$, the best achievable magnetic moment would be $\boldsymbol{p}_{\mathrm{m}} \sim \boldsymbol{\omega} \times \boldsymbol{B}$. In the rotating body coordinate system of the spacecraft, the derivative of the magnetic field can be expressed as $\dot{\boldsymbol{B}} = \boldsymbol{\omega} \times \boldsymbol{B}$ which results in eq. (230) with the normalization of the magnetic field and the arbitrary proportional factor k. Details on the stability of the B-dot controller can be found in [138, p. 40ff].

In the simulation, the required current through all three magnetorquers was computed every 0.1 s similar to eq. (230)

$$\boldsymbol{I} = -\frac{g}{\|\boldsymbol{B}\|}\dot{\boldsymbol{B}} \tag{231}$$

with the gain factor g and the current through the three magnetorquer coils $\boldsymbol{I} = (I_{\mathrm{x}}\ \ I_{\mathrm{y}}\ \ I_{\mathrm{z}})^{\mathrm{T}}$.

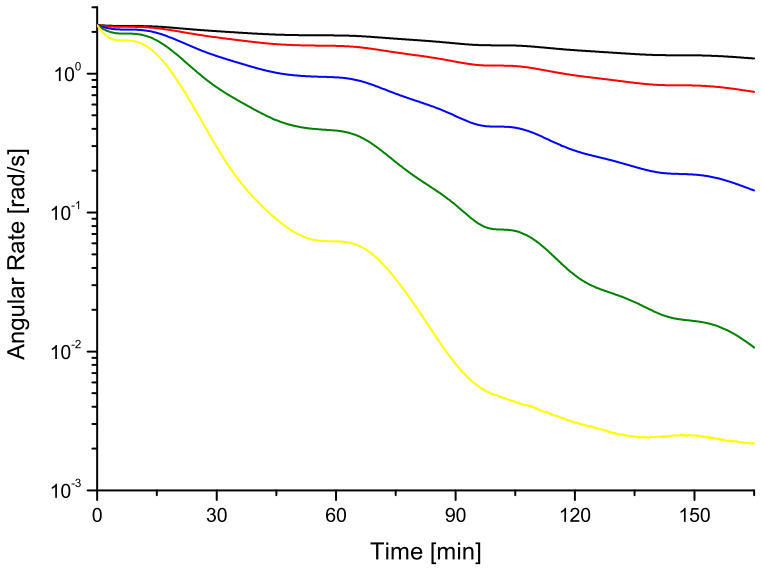

Fig. 9.16. Angular rate of detumbling spacecraft for different gain factors: $g = 0.01$ (black), $g = 0.02$ (red), $g = 0.05$ (blue), $g = 0.1$ (green), $g = 0.2$ (yellow).

The simulation was run with different values for the gain factor g and the change of the angular rate is shown in fig. 9.16 over the course of ca. two orbits. One can clearly recognize that with values of $g = 0.2$ and larger, the detumbling of the CubeSat can be achieved quite fast within three hours.

In fig. 9.17, the magnetic flux density in the global frame is shown for the entire simulation duration. Since one orbit lasts ca. 90 minutes, the magnetic field vector as observed in the spacecraft body frame rotates at a rate of ca. 1.16 mrad/s or multiples thereof which poses the lower detumble limit of the B-dot controller since the algorithm relies on a measurable magnetic field change that predominantly results from the spacecraft rotation.

9.7 Thruster

Though there is variety of different propulsion system technologies, not all thrusters lend themselves to suitable miniaturization. In particular the scaling of chemical propulsion systems is difficult [181, p. 2]. Apart from thrusters of lower complexity based on hot and cold gas [217, p. 838f], electric propulsion systems are becoming more and more dominant for CubeSats and small satellites [242, p. 2], [208, p. 2].

These can again be subdivided into the exploitation of underlying electrostatic, electrothermal and electromagnetic physical concepts [178, p. 2]. A coaxial PPT (Pulsed Plasma Thruster) with thrust-steering capability was co-developed by the author in 2013 [125] (fig. 9.18, left). The author also made major contributions to the development of the IFM Nano Thruster at FOTEC which was successfully in-orbit tested in 2018 [207] and later commercialized by the FOTEC spin-out ENPULSION [65] (fig. 9.18, right).

Apart from peculiarities of the different technologies, such as ignition behavior, thrust stability, thrust noise, thrust vector misalignment or total impulse, all thrusters have a certain controllable thrust magnitude which is modeled by the system `Thruster`. The good controllability particularly applies to electric propulsion systems where certain technologies, such as the liquid metal propellant based IFM Nano Thruster using the FEEP (Field Emission Electric Propulsion) technology. It even allows control of the specific impulse at constant thrust levels [110], [109].

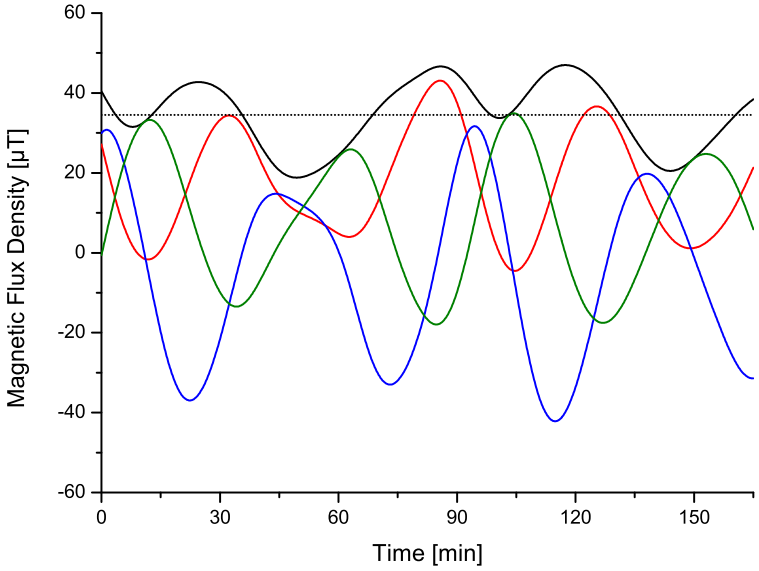

Fig. 9.17. Magnetic flux density in the global frame: magnitude (black), mean magnitude (black dotted), components x (red), y (blue) and z (green).

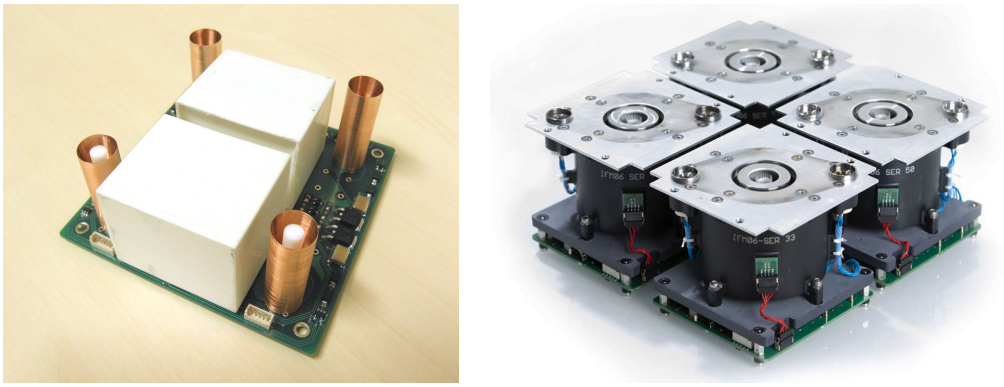

Fig. 9.18. PPT with thrust-steering capability [125] (left), four clustered IFM Nano Thrusters [65] (right).

Furthermore, what all thruster technologies have in common, it the underlying physical principle to generate thrust from a mass flow. According to eq. (102), a force $\boldsymbol{F}$ causes the change of the linear momentum $\boldsymbol{p}$ and vice versa

$$\boldsymbol{F} = \frac{\mathrm{d}\boldsymbol{p}}{\mathrm{d}t} = \dot{m}\,\boldsymbol{v}_\mathrm{p} + m\,\dot{\boldsymbol{v}}_\mathrm{p} \tag{232}$$

where $\dot{m}$ denotes the mass flow rate and $\boldsymbol{v}_\mathrm{p}$ the propellant exhaust velocity. For a continuous mass flow at constant exhaust velocity, only the first term is relevant. A measure for the propellant utilization efficiency is the so-called *specific impulse* which is proportional to the propellant exhaust velocity

$$I_\mathrm{sp} = \frac{1}{g_0}\,\|\boldsymbol{v}_\mathrm{p}\| \tag{233}$$

with the gravitation acceleration on Earth $g_0 = 9.80665\,\mathrm{m/s^2}$ [169, p. 46].

9.7.1 Implementation

The class `Thruster` represents a propulsion system to generate a controllable thrust in a fixed direction. It is derived from `System` and defined as follows excluding certain private declarations.

```
// Class Thruster
class CubeSim::System::Thruster : public System
{
public:

   // Clone
   virtual System* clone(void) const;

   // Thrust [N]
   double thrust(void) const;
   void thrust(double thrust);

   // Time Step [s]
   double time_step(void) const;
   void time_step(double time_step);

protected:

   // Constructor
   Thruster(double range = _RANGE, double accuracy = _ACCURACY, double
      time_step = _TIME_STEP);

   // Copy Constructor (reset Part)
   Thruster(const Thruster& thruster);

   // Accuracy [N]
   double _accuracy(void) const;
   void _accuracy(double accuracy);

   // Part
   Part* _part(void) const;
   void _part(Part& part);

   // Range [N]
   double _range(void) const;
   void _range(double range);

private:

   // Default Accuracy [N]
   static const double _ACCURACY;

   // Default Range [N]
   static const double _RANGE;

   // Default Time Step [s]
   static const double _TIME_STEP;

   // Behavior
   virtual void _behavior(void);
};
```

A specific thruster class can be derived from class `Thruster`. Therefore, the constructor `Thruster(double range = _RANGE, double accuracy = _ACCURACY, double time_step = _TIME_STEP)` is defined `protected`. Similar to other actuator classes, it allows the initialization of the base class with the range (default ∞ N), the accuracy (default 0 N) and the time step (default 1 s). As a real thruster, the simulated thruster is defined with a certain accuracy considering the achievable thrust stability. The thrust range allows better modeling of the propulsion system to be used. With the use of method `_accuracy` the accuracy can be altered also during the simulation by passing the new accuracy as an argument. When no argument is passed to the method, the actual accuracy is returned. The random number generator and normal distribution library of the C++ STL are used to generate representative values.

The thrust range R can be modified by the use of method `_range` called with the new thrust range. The actual range is obtained by calling the method without arguments. If the commanded thrust exceeds the defined range $T \notin [0; R]$, the value is clipped. For the selection of a suitable thruster, the trade-off between thrust range, stability and control resolution needs to be done. If the ADCS algorithm attempts to command a thrust that exceeds the allowable range, the physical behavior of the simulated thruster is still accurate.

Method `_part` allows the definition the physical body of the thruster. The part can, e.g., be a cylinder or a box with the corresponding device dimensions. The correlation not only allows to better simulate the total mass and mass distribution of the spacecraft, but is also important for the orientation of the actuator with respect to the spacecraft. The currently set part can be retrieved by calling the method without arguments. In that case a pointer is returned which can also be `nullptr` if no part has been set yet.

A named force `"Thruster"` of class `Force` is inserted into the specified part which is later controlled in dependence of the commanded thrust and accuracy where the force vector is in the direction of the z-axis.

The time step can be controlled by calling method `time_step` and passing the new value. This defines the intervals for updating the force in accordance with the defined accuracy. The actual value can be obtained by calling the method without arguments.

With the use of method `thrust` the nominal thrust can be commanded by passing the value as the only argument. To obtain the currently set thrust level, the method can be called without arguments.

Since class `Thruster` is derived from `System`, it also uses the `System` methods `enable` and `disable`. This way, it is possible to simulate outages of the actuator either on purpose by modeling the power saving plan of the spacecraft or due to random malfunction, e.g., due to latch-up effects or EMI.

As class `System` is derived from `List<System>::Item`, class `Thruster` needs to define method `clone`.

9.7.2 Verification

Eq. (52) allows the computation of the orbital period T from the semi-major axis a and the standard gravitational parameter $\mu = G\,M$. On a circular orbit (eccentricity $e = 0$), the velocity is constant and can be computed via the orbit travel distance

$$v = \frac{2\pi\,a}{T} = \sqrt{\frac{\mu}{a}}. \tag{234}$$

For an orbital transfer from $a \rightarrow a + h$, the difference in velocity results in

$$\triangle v = \sqrt{\mu}\left|\frac{1}{\sqrt{a}} - \frac{1}{\sqrt{a+h}}\right|. \tag{235}$$

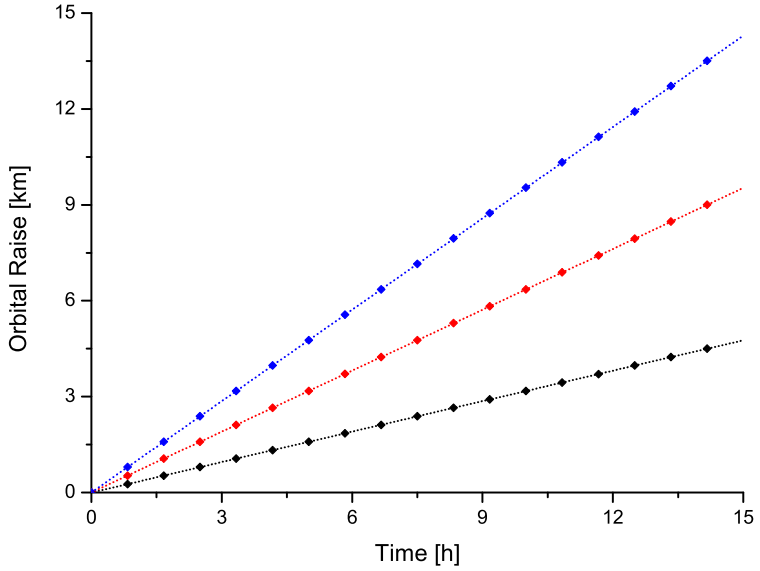

Fig. 9.19. Orbit raise for different thrust levels: $T = 100\,\mu$N (black), $T = 200\,\mu$N (red), $T = 300\,\mu$N (blue). Predicted results described by eq. (238) (dotted lines).

If the thruster of the spacecraft is providing force F for a certain maneuver time t_m and the thrust vector is parallel to the spacecraft trajectory, the velocity change $\triangle v$ can be expressed as

$$\triangle v = \frac{F\,t_\mathrm{m}}{m} \tag{236}$$

with the spacecraft mass m. This allows the computation of the orbit raise

$$h = a - \frac{1}{\left(\frac{1}{\sqrt{a}} - \frac{\triangle v}{\sqrt{\mu}}\right)^2}. \tag{237}$$

For small changes $h \ll a$ due to low thrust, short maneuver time or high spacecraft mass, the orbit raise can be approximated as

$$h \approx \frac{2\triangle v\,a^{3/2}}{\sqrt{\mu}} = \frac{2F\,t_\mathrm{m}\,a^{3/2}}{m\sqrt{\mu}}. \tag{238}$$

For the verification of the thruster, a simulation was set up including a 2U CubeSat of 2 kg with the dimensions $10 \times 10 \times 20$ cm (x, y, z). It is equipped with an IFM Nano Thruster capable of delivering up to 350 µN continuous thrust [124]. The orientation of the thruster is chosen to accelerate the spacecraft in z-axis direction.

The CubeSat is put on a LEO orbit at 400 km altitude in Nadir-pointing mode, eccentricity $e = 0$, argument of the periapsis $\omega = 120°$, longitude of the ascending node $\Omega = 70°$, inclination $i = 50°$ and mean anomaly $M = 20°$. Three different thrust levels 100 μN, 200 μN and 300 μN are applied for 15 hours in each simulation run and the relative change of the semi-major axis was observed over time. In fig. 238 the simulation results are compared with theory. It can be recognized that that simulation results are matching eq. (238) very well.

For the second verification, an orbital plane change maneuver is simulated. In this scenario, the same satellite as described above is put on the same orbit with an initial inclination of $i = 50°$. This time the orientation of the CubeSat is different to allow thrust application perpendicular to the spacecraft orbit trajectory. While the x-axis is still pointing to Nadir, the y-axis of the spacecraft is along direction of flight. The spin rate $\boldsymbol{\omega}_0 = \begin{pmatrix}0 & 0 & -2\pi/T\end{pmatrix}^\mathrm{T}$ rad/s is chosen to so that one full rotation matches the orbit

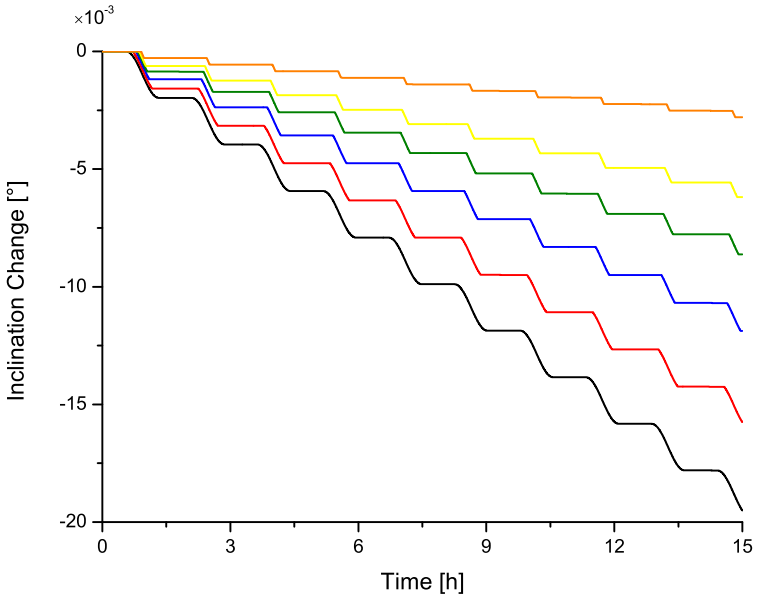

Fig. 9.20. Inclination change for different thruster operation intervals: ±90° (black), ±53° (red), ±37° (blue), ±26° (green), ±18° (yellow), ±8° (orange) around the ascending node.

period (see also section 4.3.3). For a simple plane change, where only the direction but not the magnitude of the orbital velocity changes, it is most efficient to pulse the thruster for a short moment when the spacecraft passes the ascending $\nu + \omega = 0$ or descending node $\nu + \omega = \pi$ [69, p. 2]. The required velocity change for an inclination change of θ can be expressed as

$$\triangle v = 2v \sin \frac{\theta}{2} \tag{239}$$

with v denoting the orbital velocity [71, p. 4.1.5-201f]. For low-thrust propulsion systems, such as electric propulsion, the short-term operation at the ascending or descending node would result in a very low inclination change rate. In the simulation, around the ascending node different angle segments were tested, where the thruster was enabled at 300 μN. On the one hand, this allows to increase the inclination change rate, but on the other hand it reduces the $\triangle v$ efficiency. In 9.20 the inclination change for different thruster operation intervals per orbit over a simulation duration of 15 hours is shown. When enabling the thruster for half of the time (±90°, black curve), the inclination changes by 0.0195° and for 4.5% operation time (±8°, orange curve), the inclination change achieved is as low as 0.0028°.

The $\triangle v$ efficiency in dependence of the thruster operation interval around the ascending node is shown in fig. 9.21. For short-term operation the efficiency reaches its maximum corresponding to eq. (239) which describes the ideal $\triangle v$ budget for the required plane change. The figure also shows the achievable inclination change rate which varies significantly with the thruster operation interval.

For the mission planning, a trade-off analysis needs to be performed to balance propellant use efficiency and maneuver duration. It was shown that the CubeSim framework can be used for that purpose.

9.8 Reaction Wheels

A reaction wheel is a type of flywheel that is used to allow controlling the attitude of a spacecraft. Due to the conversion of angular momentum, variations of the angular velocity cause the spacecraft to rotate around the axis of the reaction wheel. A typical reaction wheel consists of a flywheel mass which is optimized for a large moment of inertia to mass ratio to reduce the mass of the sub-system [141]. A

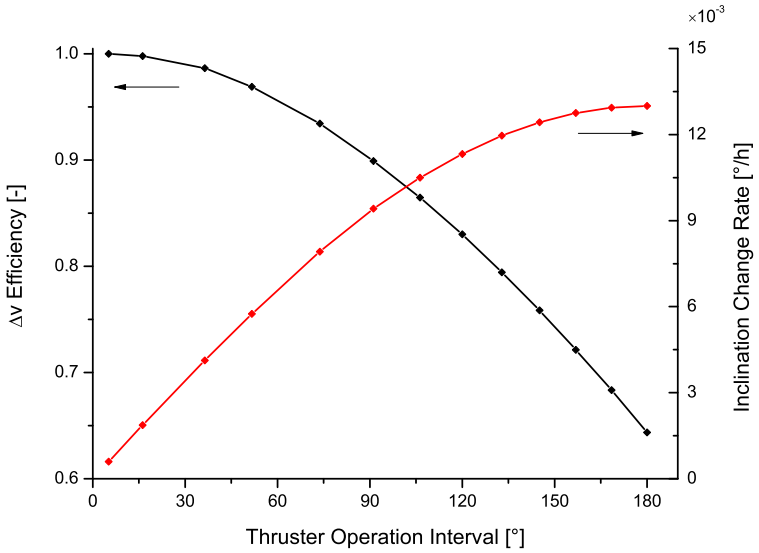

Fig. 9.21. $\triangle v$ efficiency (black) and inclination change rate (red) vs. thruster operation interval around the ascending node.

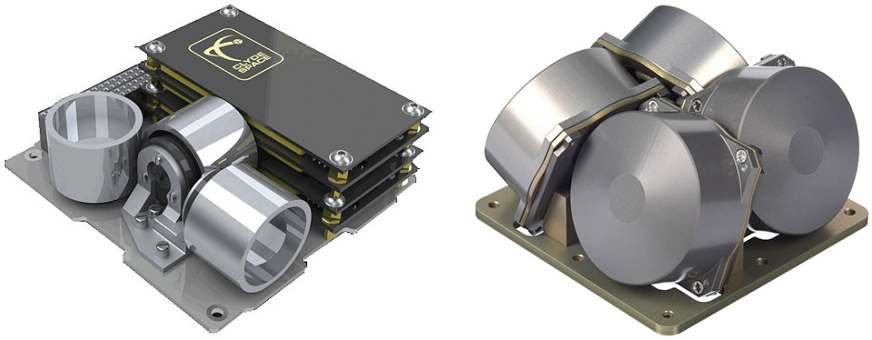

Fig. 9.22. Three-axis reaction wheel unit by Clyde Space (left) and four-axis unit by NanoAvionics (right).

servo, DC or stepper motor is used to accelerate or decelerate the flywheel where great emphasis is placed on the accuracy and stability of the rotational speed.

Nominally, reaction wheels have zero speed and may be rotated in either direction as commanded by the ADCS. It should be noted that at low speeds, static friction can occur resulting in non-linear behavior and irregular motion. One way to mitigate this issue is to set the nominal speed slightly above zero to a few RPM (Rotations Per Minute) [74, p. 305f].

In order to allow attitude control around all three axes, three reaction wheels are being used (see fig. 9.22, left) where the axes of the wheels are typically perpendicular [113, p. 52]. For critical missions, where reliability is a major concern, a fourth reaction wheel is added, where the orientation is chosen in a way that the axes of three out of four reaction wheels span a three-dimensional space (see fig. 9.22, right). This still allows three-axis attitude control even if one wheel is malfunctioning.

Further information on reaction wheels and also momentum wheels and typical performance figures can be found in [253, p. 201ff].

The underlying physical principle of reaction wheels is the conservation of angular momentum. When no external torque is acting on the spacecraft and the spin rate of a reaction wheel is changed, the spacecraft changes its angular rate in the counter direction to maintain the total angular momentum.

The angular momentum is the sum of the angular momentum of the reaction wheel and the spacecraft

$$L = I_{\mathrm{rw}}\,\omega_{\mathrm{rw}} + I_{\mathrm{sc}}\,\omega_{\mathrm{sc}} \tag{240}$$

where the I_{rw} denotes the moment of inertia of the reaction wheel's flywheel around the main axis and I_{sc} is the moment of inertia of the spacecraft around the same axis including the contribution of the flywheel itself. This results in a change of the angular rate of the spacecraft when the spin rate of the reaction wheel is changed

$$\omega_{\mathrm{sc}} = \frac{L - I_{\mathrm{rw}}\,\omega_{\mathrm{rw}}}{I_{\mathrm{sc}}} = \frac{L}{I_{\mathrm{sc}}} - \frac{I_{\mathrm{rw}}}{I_{\mathrm{sc}}}\,\omega_{\mathrm{rw}}. \tag{241}$$

This concept can be extended to three or more reaction wheels in which case the moment of inertia matrices are used instead of scalars in similar equations as above.

9.8.1 Implementation

The class `ReactionWheel` represents a single-axis reaction wheel that can spin in either direction. It is derived from `System` and defined as follows excluding certain private declarations.

```
// Class ReactionWheel
class CubeSim::System::ReactionWheel : public System
{
public:

    // Clone
    virtual System* clone(void) const;

    // Spin Rate [rad/s]
    double spin_rate(void) const;
    void spin_rate(double spin_rate);

    // Time Step [s]
    double time_step(void) const;
    void time_step(double time_step);

protected:

    // Constructor
    ReactionWheel(double range = _RANGE, double accuracy = _ACCURACY,
        double acceleration = _ACCELERATION, double time_step =
        _TIME_STEP);

    // Copy Constructor (reset Part)
    ReactionWheel(const ReactionWheel& reaction_wheel);

    // Spin Rate Acceleration [rad/s^2]
    double _acceleration(void) const;
    void _acceleration(double acceleration);

    // Accuracy [rad/s]
    double _accuracy(void) const;
    void _accuracy(double accuracy);

    // Part
    Part* _part(void) const;
```

```
    void _part(Part& part);

    // Range [rad/s]
    double _range(void) const;
    void _range(double range);

private:

    // Default Spin Rate Acceleration [rad/s^2]
    static const double _ACCELERATION;

    // Default Accuracy [rad/s]
    static const double _ACCURACY;

    // Default Range [rad/s]
    static const double _RANGE;

    // Default Time Step [s]
    static const double _TIME_STEP;

    // Behavior
    virtual void _behavior(void);
};
```

A specific reaction wheel class can be derived from class `ReactionWheel`. Therefore, the constructor `ReactionWheel(double range = _RANGE, double accuracy = _ACCURACY, double acceleration = _ACCELERATION, double time_step = _TIME_STEP)` is `protected`. Similar to other actuator classes, it allows the initialization of the base class with the spin rate range (default ∞ rad/s), the accuracy (default 0 rad/s), the acceleration (default ∞ rad/s^2) and the time step (default 1 s). As a real reaction wheel, the simulated reaction wheel is defined with a certain accuracy representing the stability and characteristics of the motor that is driving the flywheel. The spin rate range allows better modeling of the actuator to be used. With the use of method `_accuracy` the accuracy can be altered also during the simulation by passing the new accuracy as an argument. When no argument is passed to the method, the actual accuracy is returned. The random number generator and normal distribution library of the C++ STL are used to generate representative values.

The spin rate range R can be modified by the use of method `_range` called with the new range. The actual range is obtained by calling the method without arguments. If the commanded spin rate exceeds the defined range $\omega \notin [-R; +R]$, the value is clipped. For the selection of a suitable reaction wheel, the trade-off between moment of inertia, maximum spin rate, system mass and stability needs to be done. If the ADCS algorithm attempts to command a spin rate that exceeds the allowable range, the physical behavior of the simulated reaction wheel is still accurate.

A reaction wheel cannot immediately change its spin rate due to the limitation of the motor's torque. This limit can also be modeled by using method `_acceleration` which allows limiting the spin rate change by calling the method with the desired value. In order to read out the current setting, the method can be called without arguments.

Method `_part` allows the definition the physical body of the flywheel. It is recommended to use a rotational symmetric part with respect to the z-axis such as a cylinder. The dimensions and the choice of the material not only define the mass of the flywheel but the moment of inertia which in turn affects the resultant angular momentum at a certain spin rate (see section 3.8). The currently set part can be retrieved by calling the method without arguments. In that case a pointer is returned which can also be `nullptr` if no part has been set yet.

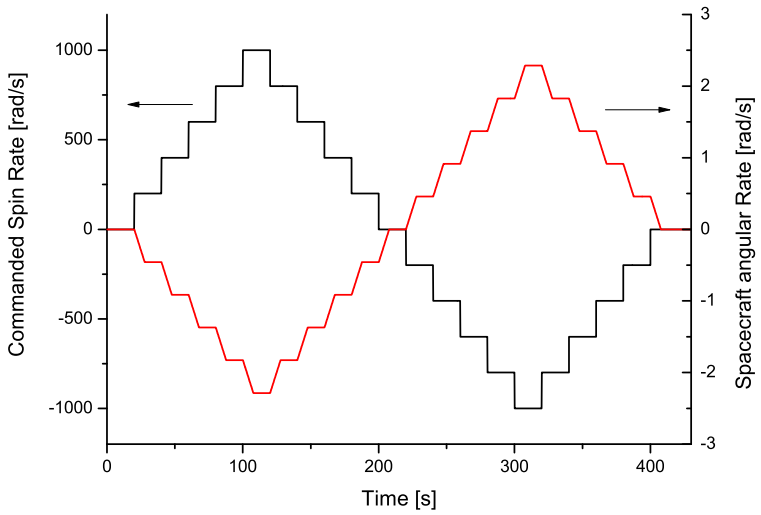

Fig. 9.23. Commanded spin rate of the reaction wheel (black) and resulting spacecraft angular rate around the z-axis (red).

The time step can also be controlled by calling method `time_step` and passing the new value. This defines the intervals for updating the spin rate in accordance with the defined acceleration and implicitly considers the specified accuracy. The actual value can be obtained by calling the method without arguments.

For operation, the desired spin rate can be set by calling method `spin_rate` and passing the new value. The actual spin rate is following a ramp defined by the acceleration limit until reaching the set point. The method returns the currently set spin rate if being called without arguments.

Since class `ReactionWheel` is derived from `System`, it also uses the `System` methods `enable` and `disable`. This way, it is possible to simulate outages of the actuator either on purpose by modeling the power saving plan of the spacecraft or due to random malfunction, e.g., due to latch-up effects or EMI.

As class `System` is derived from `List<System>::Item`, class `ReactionWheel` needs to define method `clone`.

9.8.2 Verification

For the verification of the reaction wheel, a simulation was set up including a 1U CubeSat of 1 kg with the dimensions $10 \times 10 \times 10$ cm (x, y, z). It is equipped with single reaction wheel type RW210 manufactured by Hyperion Technologies [97]. The variant with momentum storage capability of 6 mN m s at $\omega_{\mathrm{rw,max}} = 1.5 \times 10^4$ RPM was chosen to be modeled. The control uncertainty is ± 0.5 RPM according to the data sheet. Under the assumption of the flywheel diameter of 26 mm, the height was set to 10.6 mm. If it is made of stainless steel, the mass would be 45.19 g and with the resulting inertia $I_{\mathrm{rw}} = 3.81872 \times 10^{-6}\,\mathrm{kg\,m^2}$ the angular momentum matches the data sheet value at $\omega_{\mathrm{rw,max}}$. The flywheel rotational axis is set in z-axis direction.

For the simulation, first the commanded spin rate of the reaction wheel was increased in several steps to $1,000$ rad/s then decreased down to $-1,000$ rad/s and finally reset to zero. The maximum torque of the RW210 reaction wheel is 10^{-4} Nm which corresponds to a spin rate change of $26.18\,\mathrm{rad/s^2}$. Thus, for a step size of 200 rad/s, it takes the reaction wheel 7.64 s to reach the commanded spin rate. The angular rate of the spacecraft was observed, as shown in fig. 9.23.

Due to the controlled angular acceleration of the reaction wheel, the angular rate of the spacecraft also shows a corresponding slope. The moment of inertia of the spacecraft around the z-axis consists of the

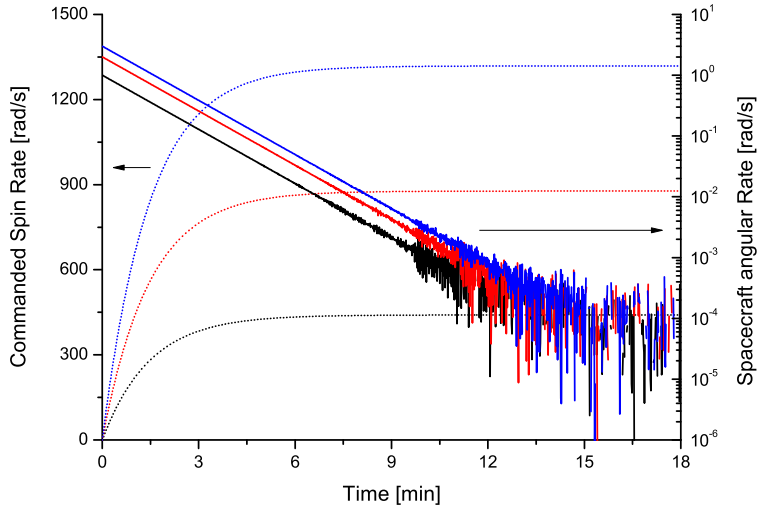

Fig. 9.24. Commanded spin rate of the reaction wheel (dotted) and angular rate of the spacecraft (solid): components x (black), y (red), z (blue).

moment of inertia of the $10 \times 10 \times 10\,\mathrm{cm}$ box (see section 7.1) with $1\,\mathrm{kg}$ and the flywheel (see section 7.3) in the spacecraft body frame

$$I_{\mathrm{sc}} = \frac{1}{12}\left(0.1^2 + 0.1^2\right) + I_{\mathrm{rw}} = 1.67049 \times 10^{-3}\,\mathrm{kg\,m^2}. \tag{242}$$

At the begin of the simulation, the spin rate of the spacecraft was zero and due to the conservation of angular momentum, the following equation applies

$$L = I_{\mathrm{rw}}\,\omega_{\mathrm{rw}} + I_{\mathrm{sc}}\,\omega_{\mathrm{sc}} = 0 \tag{243}$$

and the angular rate of the spacecraft in dependence of the spin rate of the reaction wheel is

$$\omega_{\mathrm{sc}} = -\frac{I_{\mathrm{rw}}}{I_{\mathrm{sc}}}\,\omega_{\mathrm{rw}}. \tag{244}$$

At $\omega_{\mathrm{rw}} = 1000\,\mathrm{rad/s}$ the angular rate of the spacecraft results in $\omega_{\mathrm{sc}} = -2.28599\,\mathrm{rad/s}$ matching the computed angular rate in the simulation of $-2.28598\,\mathrm{rad/s}$ very well (see fig. 9.23).

For the second verification, the detumbling process of the satellite with the use of three reaction wheels is simulated. In this scenario, the dimensions and the mass of the spacecraft remain identical. The initial spin rate $\hat{\boldsymbol{\omega}}_0 = (1\ \ 2\ \ 3)^{\mathrm{T}}\,\mathrm{rad/s}$ in the spacecraft body frame is chosen and three PID (Proportional Integral Derivative) controllers [102, sec. 3.3] for the three axes are implemented to detumble the spacecraft. These have the transfer function

$$\boldsymbol{\omega}_{\mathrm{rw}} = k_{\mathrm{p}}\,\hat{\boldsymbol{\omega}}_{\mathrm{sc}} + k_{\mathrm{i}} \int \hat{\boldsymbol{\omega}}_{\mathrm{sc}}\,\mathrm{d}t + k_{\mathrm{d}}\frac{\mathrm{d}\hat{\boldsymbol{\omega}}_{\mathrm{sc}}}{\mathrm{d}t} \tag{245}$$

where for sake of simplicity only the integral part was used which results in the gain factors $k_{\mathrm{i}} = 5$ and $k_{\mathrm{p}} = k_{\mathrm{d}} = 0$. $\hat{\boldsymbol{\omega}}_{\mathrm{sc}}$ is represented in the spacecraft body frame. In fig. 9.24 the angular rate of the spacecraft for all three axes (solid lines) and the corresponding spin rate of the reaction wheels (dotted lines) are shown. One notices the increasing noise as the angular rate of the spacecraft reaches ca. $5 \times 10^{-3}\,\mathrm{rad/s}$ which results from the modeled uncertainty of the reaction wheel RW210. Therefore, this simulation can not only be used to optimize the parameters of the PID controllers but also shows the expected accuracy for certain types of reaction wheels.

For the third verification, a detumbled spacecraft with same dimensions and mass as before is simulated. The three reaction wheels are used to put the spacecraft in Nadir-pointing mode. For that purpose the three PID controllers for computing the spin rate of the reaction wheels are reused. This time, instead of $\hat{\boldsymbol{\omega}}_{\mathrm{sc}}$ as the error function, the Euler angles (see section 3.7.3) computed from the difference in rotation between the current spacecraft orientation and the desired Y-Thomson rotation are used. These are expressed in the spacecraft body frame: roll (ψ, rotation around the x-axis), pitch (θ, around the y-axis) and yaw (ϕ, around the z-axis). This leads to the vector equation

$$\boldsymbol{\omega}_{\mathrm{rw}} = k_{\mathrm{p}}\,\boldsymbol{\Omega} + k_{\mathrm{i}} \int \boldsymbol{\Omega}\,\mathrm{d}t + k_{\mathrm{d}} \frac{\mathrm{d}\boldsymbol{\Omega}}{\mathrm{d}t} \tag{246}$$

with the introduced vector consisting of the three Euler angles $\boldsymbol{\Omega} = (\psi \;\; \theta \;\; \phi)^{\mathrm{T}}$. This time the gain factor $k_{\mathrm{p}} = 10$ for proportional control is chosen due to the different transfer function. If the system was only controlled by a P-type controller, a certain misalignment would persist, because of the non-zero final angular rate of the spacecraft in Y-Thomson spin. Therefore, a small integration gain is introduced $k_{\mathrm{i}} = 0.1$. Still the differential gain is not used $k_{\mathrm{d}} = 0$ in this scenario.

Fig. 9.25 shows the simulation results with the spacecraft angular rate in body frame (solid lines) and the corresponding spin rates of all three reaction wheels (dotted lines). One recognizes, due to the proportional gain, the spin rates rise quickly at the begin of the simulation. As the difference in orientation to Nadir-pointing mode decreased after two minutes (see fig. 9.26), the spin rates also drop accordingly. As opposed to the detumbling simulation performed previously, the non-zero angular rate of the spacecraft around its y-axis persists to perform one full rotation per orbit. This can be seen in fig. 9.25 when the angular rate curves start to converge. The red dashed line shows the nominal angular rate of 1.133×10^{-3} rad/s around the y-axis as described in section 9.7.2 which shows good accordance with the simulation results. The noise in fig. 9.25 results again from the simulated uncertainty of ± 0.052 rad/s of the used RW210 reaction wheels.

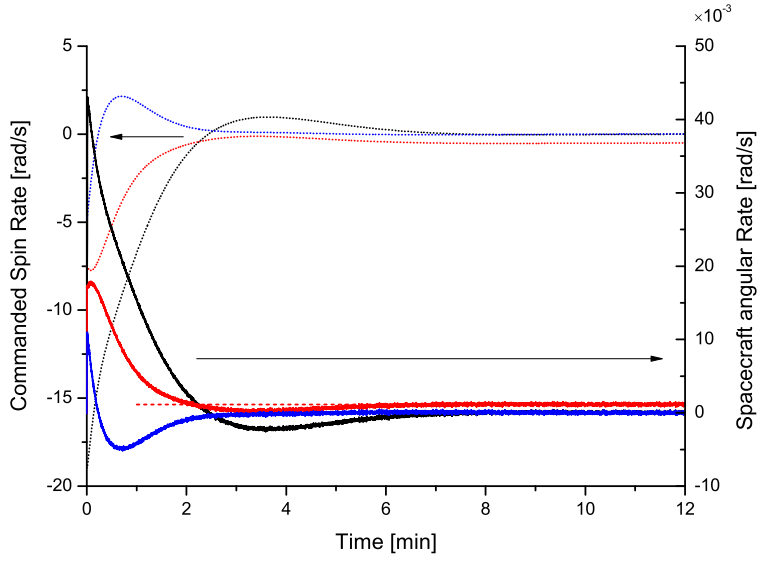

Fig. 9.25. Commanded spin rate of the reaction wheel (dotted) and angular rate of the spacecraft (solid): components x (black), y (red, final value dashed), z (blue).

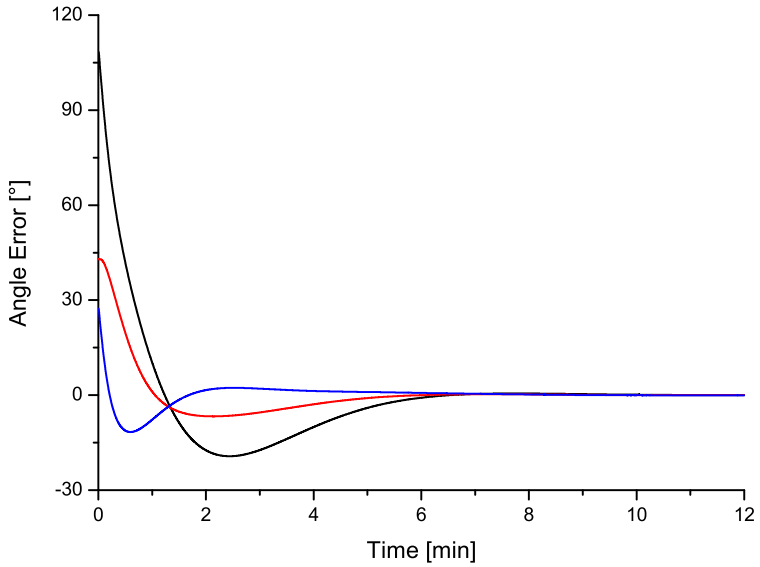

Fig. 9.26. Euler angle deviations from Nadir-pointing mode orientation: ψ (black), θ (red), ϕ (blue).

Chapter 10

Spacecraft

A spacecraft comprises of multiple systems such as the on-board computer, an AOCS unit, the PPU (Power Processing Units) or thrusters. Systems are derived from class `Behavior` which allows to individually model their behavior. As spacecraft are also derived from class `Behavior`, it is possible to assign them routines that, e.g., monitor or log the performance of all systems. Since spacecraft are derived from class `RigidBody`, forces and torques can be inserted which is particularly relevant for modules such as `Gravitation` or `Motion`. Apart from celestial bodies, spacecraft are free rigid bodies meaning that they only rotate around their center of mass.

Considering the hierarchy of the simulation framework, class `Spacecraft` consists of systems and is derived from classes `RigidBody`, `Behavior`, `List<System>` and `List<Spacecraft>::Item`.

The class `Spacecraft` is defined as follows excluding certain private declarations.

```
// Class Spacecraft
class CubeSim::Spacecraft : public Behavior, public RigidBody, private
    List<System>, public List<Spacecraft>::Item
{
public:

    // Class Hubble
    class Hubble;

    // Class ISS
    class ISS;

    // Constructor
    Spacecraft(void);

    // Copy Constructor (Simulation Reference is reset)
    Spacecraft(const Spacecraft& spacecraft);

    // Assign (Simulation Reference is maintained)
    Spacecraft& operator =(const Spacecraft& spacecraft);

    // Insert System, Force and Torque
    System& insert(const std::string& name, const System& system);
    using RigidBody::insert;

    // Compute Orbit
    const Orbit orbit(const CelestialBody& central, const Rotation&
        reference = Orbit::REFERENCE_ECLIPTIC) const;
```

```
    // Get Simulation
    Simulation* simulation(void) const;

    // Get System
    const std::map<std::string, System*>& system(void) const;
    System* system(const std::string& name) const;

private:

    // Constructor (Vector Constants might not yet be initialized)
    Spacecraft(const Vector3D& position, const Vector3D& velocity, const
        Vector3D& angular_rate = Vector3D(), const Rotation& rotation =
        Rotation(Vector3D(0.0, 0.0, 1.0), 0.0));

    // Compute angular Momentum (Body Frame) [kg*m^2/s]
    virtual const Vector3D _angular_momentum(void) const;

    // Compute Surface Area [m^2]
    virtual double _area(void) const;

    // Compute Center of Mass (Body Frame) [m]
    virtual const Vector3D _center(void) const;

    // Check if Point is inside (Body Frame)
    virtual bool _contains(const Vector3D& point) const;

    // Compute Moment of Inertia (Body Frame) [kg*m^2]
    virtual const Inertia _inertia(void) const;

    // Compute Mass [kg]
    virtual double _mass(void) const;

    // Compute Momentum (Body Frame) [kg*m/s]
    virtual const Vector3D _momentum(void) const;

    // Compute Volume [m^3]
    virtual double _volume(void) const;

    // Compute Wrench (Body Frame)
    virtual const Wrench _wrench(void) const;
};
```

The class `Behavior` adds the virtual method `_behavior` which is to be defined by the class deriving from `Spacecraft`. As defined in section 2.2, the inherited virtual method `_behavior` can be used to represent the functionality on spacecraft level such as the coordination between its systems.

One recognizes the predefined spacecraft `Hubble` and `ISS` which are representing the HST (Hubble Space Telescope) and the ISS (International Space Station) as seen in fig. 10.1. They can be used as source of typical LEO orbits for custom spacecraft. As both spacecraft use propulsion systems and their altitude is rather low, the positioning accuracy is limited. See module `Ephemeris` for details (see section 4.6).

The default constructor `Spacecraft(void)` creates an empty spacecraft object and the copy constructor `Spacecraft(const Spacecraft& spacecraft)` can be used to duplicate an existing spacecraft. In this case any potential reference to the simulation is reset. A spacecraft can also be overwritten by the use of the assignment operator `operator =`.

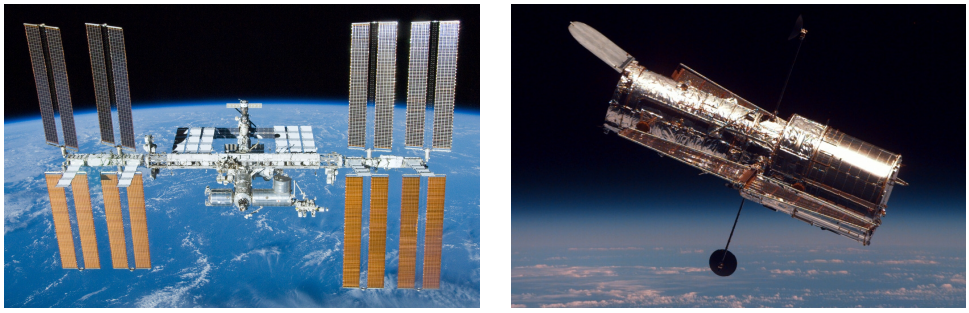

Fig. 10.1. The International Space Station (left) and the Hubble Space Telescope (right).

With the help of method `insert`, systems, forces and torques can be inserted. In order to distinguish or modify them later on, a unique name has to be given. The method returns a reference to the inserted copy of the passed object.

Method `orbit` allows to compute the current orbit of the spacecraft and expects the central celestial body and optionally the reference system as arguments. See section 3.13 for details.

If the spacecraft is inserted into a simulation, a pointer to the simulation `Simulation*` can be retrieved by calling method `simulation` without arguments. If the spacecraft is not part of a simulation, `nullptr` is returned.

Method `system` is used to find a system by specifying the name or to retrieve an associative map of type `std::map<std::string, System*>` consisting of all systems.

As class `Spacecraft` is derived from class `RigidBody`, the virtual methods `_angular_momentum`, `_area`, `_center`, `_contains`, `_inertia`, `_mass`, `_momentum`, `_volume` and `_wrench` need to be implemented. Similar to classes `Assembly` and `System`, these methods iterate through all inserted systems to compute the resultant value. See chapters 8 and 9 for details.

Chapter 11

Celestial Bodies

Celestial bodies are simulated to investigate their influence on the spacecraft because of their gravitational or magnetic field. The simulation framework also allows numerical computation of the celestial body trajectories by solving the n-body problem iteratively.

The class `CelestialBody` is derived from class `RigidBody` and class `List<CelestialBody>::Item` and is defined as follows excluding certain private declarations.

```
// Class CelestialBody
class CubeSim::CelestialBody : public RigidBody, public
    List<CelestialBody>::Item
{
public:

    // Class Earth
    class Earth;

    // Class Jupiter
    class Jupiter;

    // Class Mars
    class Mars;

    // Class Mercury
    class Mercury;

    // Class Moon
    class Moon;

    // Class Neptune
    class Neptune;

    // Class Pluto
    class Pluto;

    // Class Saturn
    class Saturn;

    // Class Sun
    class Sun;

    // Class Uranus
    class Uranus;
```

```
// Class Venus
class Venus;

// Constructor
CelestialBody(void);

// Copy Constructor (Simulation Reference is reset)
CelestialBody(const CelestialBody& celestial_body);

// Assign (Simulation Reference is maintained)
CelestialBody& operator =(const CelestialBody& celestial_body);

// Compute Circumference [m]
double circumference(void) const;

// Clone
virtual CelestialBody* clone(void) const = 0;

// Density [kg/m^3]
double density(void) const;
void density(double density);

// Relative Flattening
double flattening(void) const;
void flattening(double flattening);

// Compute gravitational Field (Body Frame) [m/s^2]
const Vector3D gravitational_field(double x, double y, double z)
    const;
const Vector3D gravitational_field(const Location& location) const;
virtual const Vector3D gravitational_field(const Vector3D& point)
    const;

// Locate Point
const Location locate(double x, double y, double z) const;
const Location locate(const Vector3D& point) const;

// Compute magnetic Field (Body Frame) [T]
const Vector3D magnetic_field(double x, double y, double z) const;
const Vector3D magnetic_field(const Location& location) const;
virtual const Vector3D magnetic_field(const Vector3D& point) const;

// Compute Orbit
const Orbit orbit(const CelestialBody& central, const Rotation&
    reference = Orbit::REFERENCE_ECLIPTIC) const;

// Compute relative Reflectivity
virtual double reflectivity(double longitude, double latitude) const;

// Mean Radius [m]
double radius(void) const;
void radius(double radius);

// Get Simulation
Simulation* simulation(void) const;

// Surface Temperature [K]
double temperature(void) const;
```

```
    void temperature(double temperature);

protected:

    // Constructor
    CelestialBody(double radius, double flattening, double density,
        double temperature, const Vector3D& position, const Vector3D&
        velocity, const Vector3D& angular_rate = Vector3D(), const
        Rotation& rotation = Rotation(Vector3D(0.0, 0.0, 1.0), 0.0));

private:

    // Compute Surface Area [m^2]
    virtual double _area(void) const;

    // Compute Center of Mass (Body Frame) [m]
    virtual const Vector3D _center(void) const;

    // Check if Point is inside (Body Frame)
    virtual bool _contains(const Vector3D& point) const;

    // Compute Moment of Inertia (Body Frame, Origin) [kg*m^2]
    virtual const Inertia _inertia(void) const;

    // Compute Mass [kg]
    virtual double _mass(void) const;

    // Compute Volume [m^3]
    virtual double _volume(void) const;
};
```

One immediately recognizes the celestial bodies of our solar system consisting of the sun `Sun`, the planets `Earth`, `Jupiter`, `Mars`, `Mercury`, `Neptune`, `Pluto`, `Saturn`, `Uranus`, `Venus` and the Earth moon `Moon`. Their properties, such as density, flattening, radius or mean temperature are predefined and allow immediate use in the simulation.

With the use of the default constructor `CelestialBody(void)` an object of class `CelestialBody` is created. The copy constructor `CelestialBody(const CelestialBody& celestial_body)` is provided to duplicate an existing object. In this case any potential reference to the simulation is reset. A celestial body can also be overwritten by the use of the assignment operator `operator =`. Here the reference to the simulation is maintained.

The method `circumference` returns the equatorial circumference c based on the defined mean radius r_e of the celestial body

$$c = 2\pi\, r_\mathrm{e}. \tag{247}$$

Internally method `radius` is called without arguments returning the aforementioned radius r_e. If the radius is to be modified, the new value can be passed to the method.

Method `density` returns the mean density ρ of the celestial body when called without any arguments. In order to modify the density, the new values can be passed to the method.

Due to the spin rate of a celestial body, it is generally not perfectly spherical but experiences flattening f at the poles. This reduces the polar radius r_p accordingly

$$r_\mathrm{p} = (1 - f)\, r_\mathrm{e}. \tag{248}$$

A class derived from `CelestialBody` can implement the virtual method `gravitational_field`. It receives a reference to a point of type `Vector3D` in the body frame and needs to return the local gravitational field strength. If the method is not overridden by the derived class, Newton's law of gravitation eq. (75) is used instead (see section 4.1 for details).

To transform a point in the global coordinate system into the body frame of a celestial body, method `locate` is to be used. The point of type `Vector3D` is passed to the method which returns the location of type `Location` (see section 3.12 for the class definition and further information).

Similar to the gravitational field, the magnetic field of a class derived from `CelestialBody` can be modeled by overriding the virtual method `magnetic_field`. It receives a reference to a point of type `Vector3D` in the body frame and needs to return the local magnetic field strength. If the method is not overridden by the derived class, the method always returns 0.

The orbit of a celestial body around a central celestial body is computed by the use of method `orbit`. It expects two arguments, a reference to the central celestial body and optionally the reference frame. The ecliptic frame is used by default (see also section 3.13).

For computing the albedo of a celestial body, the reflectivity map over its surface needs to be defined. The virtual method `reflectivity` can be overridden by a class derived from `CelestialBody`. It receives the longitude and latitude specifying the point on the surface and needs to return the reflectivity (between 0 and 1). In the current implementation, no frequency-dependent reflectivity is used (see section 4.3 for details).

If the celestial body is inserted into a simulation, a pointer to the simulation can be retrieved by calling method `simulation` without arguments. If the celestial body is not part of a simulation, `nullptr` is returned.

In particular for stars, the black-body temperature is relevant as being used in eq. (79) in module `Light` (see section 4.2). For planets with atmosphere, the black-body temperature can significantly differ from the surface temperature due to the natural greenhouse effect and the planet's energy budget [240, p. 4]. For Mars the difference is negligible (0.2 K), the Earth shows a difference of 34 K and Venus has a significant difference between black-body and surface temperature of 510.4 K [135, p. 3].

As class `CelestialBody` is derived from class `RigidBody`, the virtual methods `_area`, `_center`, `_contains`, `_inertia`, `_mass` and `_volume` need to be implemented. The volume of an ellipsoid is computed by method `_volume` with

$$V = \frac{4\pi}{3} r_\mathrm{e}^3 \, (1 - f) \tag{249}$$

and assuming uniform density, the mass

$$m = \rho \, V \tag{250}$$

is returned by method `_mass`.

Method `_center` returns the actual position of the celestial body.

To check whether a point $\boldsymbol{p} = (x \;\; y \;\; z)^\mathrm{T}$ lies within the celestial body, method `_contains` uses the condition

$$z^2 \leqslant (1 - f)^2 \, (r_\mathrm{e}^2 - x^2 - y^2). \tag{251}$$

The inertia of an ellipsoid, as computed by method `_inertia`, is defined as

Table 11.1. Predefined celestial bodies in our solar system (units in kg/m^3, m, kg, K).

Body	Density	Radius	Mass	Flattening	Temp.	Reference
Sun	1409.892	6.95700×10^8	1.98847×10^{30}	0.00005	5772.0	[256, p. 9]
Earth	5531.956	6.37101×10^6	5.97219×10^{24}	0.00335	254.0	[155, p. 1]
Jupiter	1418.214	6.99110×10^7	1.89819×10^{27}	0.06487	109.9	[156, p. 1]
Mars	3955.919	3.38992×10^7	6.41710×10^{23}	0.00589	209.8	[157, p. 1]
Mercury	5426.978	2.44000×10^6	3.30200×10^{23}	0.00009	439.6	[163, p. 1]
Neptune	1637.483	2.47660×10^7	1.02400×10^{26}	0.01710	46.6	[159, p. 1]
Pluto	1859.554	1.18830×10^6	1.30700×10^{22}	0.00000	37.5	[160, p. 1]
Saturn	761.743	5.82320×10^7	5.68340×10^{26}	0.09796	81.0	[161, p. 1]
Uranus	1300.229	2.53620×10^7	8.68130×10^{25}	0.02293	58.1	[162, p. 1]
Venus	5243.777	6.05184×10^7	4.86850×10^{24}	0.00000	226.6	[164, p. 1]
Moon	3344.589	1.73753×10^6	7.34900×10^{22}	0.00120	270.4	[158, p. 1]

$$\boldsymbol{I} = \frac{m}{5} \begin{pmatrix} r_\mathrm{e}^2 + r_\mathrm{p}^2 & 0 & 0 \\ 0 & r_\mathrm{e}^2 + r_\mathrm{p}^2 & 0 \\ 0 & 0 & 2r_\mathrm{e}^2 \end{pmatrix}. \tag{252}$$

The density, equatorial mean radius, mass, flattening and black-body temperature of the sun, the planets of our solar system and the Earth moon are listed in table 11.1. Apart from the NASA JPL Horizons service [166], other references were used as listed in the table.

When the specified mass did not match according to eqs. (249) and (250), the density was slightly adapted to obtain the correct mass for the given equatorial mean radius and flattening.

Chapter 12

Simulations

A simulation object represents the top-level object in CubeSim. Multiple spacecraft, modules and celestial bodies can be inserted into a single simulation object. This allows the user to simulate only the relevant properties by omitting other spacecraft, modules and celestial bodies that would not affect the simulation results. This approach reduces the simulation complexity and therefore allows to increase the simulation performance. After its configuration, the simulation can be started and stopped at any point in virtual real-time.

Considering the hierarchy of the simulation framework, the class `Simulation` is derived from `List<CelestialBody>`, `List<Module>` and `List<Spacecraft>` to allow storing the corresponding objects.

The class `Simulation` is defined as follows excluding certain private declarations.

12.1 Implementation

```
// Class Simulation
class CubeSim::Simulation : private List<CelestialBody>, private
    List<Module>, private List<Spacecraft>
{
public:

    // Constructor
    Simulation(const Time& time = _TIME);

    // Get Celestial Body
    const std::map<std::string, CelestialBody*>& celestial_body(void)
        const;
    CelestialBody* celestial_body(const std::string& name) const;

    // Delay [s]
    void delay(double time);
    void delay(const Time& time);

    // Insert celestial Body, Module and Spacecraft
    CelestialBody& insert(const std::string& name, const CelestialBody&
        celestial_body);
    Module& insert(const std::string& name, const Module& module);
```

```
    Spacecraft& insert(const std::string& name, const Spacecraft&
        spacecraft);

    // Get Module
    const std::map<std::string, Module*>& module(void) const;
    Module* module(const std::string& name) const;

    // Run
    void run(double time);
    void run(const Time& time);

    // Get Spacecraft
    const std::map<std::string, Spacecraft*>& spacecraft(void) const;
    Spacecraft* spacecraft(const std::string& name) const;

    // Stop
    void stop(void);

    // Time
    const Time time(void) const;
    void time(const Time& time);

private:

    // Default Time
    static const Time _TIME;

    // Behavior
    static void _behavior(void* parameter);

    // Parse Systems
    static void _parse(std::vector<Fiber*>& fiber, const
        std::map<std::string, System*>& system);
};
```

The constructor `Simulation(const Time& time = _TIME)` creates an empty simulation object with an optional argument defining the start time of the simulation. If it is omitted, the default time 2015-01-01 00:00:00 is used.

When the simulation is running, different threads are executed quasi-simultaneously using cooperative multi-threading (see section 2.2 for details). When a thread shall be delayed, e.g., a control loop is periodically executed with intervals of inactivity, method `delay` can be used. Either the delay interval or the wake-up time is passed to the method. Internally the corresponding fiber is suspended and the passed delay is used to determine when the fiber execution shall be continued (see section 2.2.4 for details).

By the use of method `insert`, celestial bodies, modules and spacecraft can be inserted. In order to distinguish or modify them later on, a unique name has to be given. The method returns a reference to the inserted copy of the passed object.

Methods `celestial_body`, `module` and `spacecraft` are used to find a celestial body, module or spacecraft respectively by specifying the name or to retrieve an associative map of the corresponding types `std::map<std::string, CelestialBody*>`, `std::map<std::string, Module*>` and `std::map<std::string, Spacecraft*>` consisting of all celestial bodies, modules or spacecraft that were inserted into the simulation.

After the simulation was properly set up by inserting and configuring all required objects, it can be run by calling method `run`. Similar to method `delay`, the method expects either the simulation duration or the end time.

Whenever the simulation shall be interrupted or stopped, method `stop` can be called without arguments. After the simulation was stopped, it can be resumed by using method `run` again.

In order to set the start time of a simulation run (this can be relevant for modules like `Ephemeris`, see section 4.6), method `time` is called passing the start time as the only argument. During a simulation run, method `time` returns the actual virtual real-time. The method is called without arguments in that case.

The internal method `_behavior` is used to dispatch the callback used for the multi-threading environment. Method `_parse` is used to recursively find all systems and sub-systems of all spacecraft inserted into the simulation.

Chapter 13

Conclusions and Outlook

This chapter summarizes the essential parts of the thesis and compares the development results and findings with the established objectives. The CubeSim framework is also critically reviewed and existing limitations and possible extension options are further discussed. An outlook on the future use and further development of the framework in the scientific community and academia is given.

13.1 Summary

In chapter 1 the idea behind the developed CubeSim framework is explained. With experimental test stands only certain aspects of a spacecraft or sub-systems can be verified, such as the functionality of actuators or the verification or calibration of sensors. A combined test where on the one hand multiple sensors need to be stimulated and on the other hand, the test system needs to react on actuator settings, is hardly possible, in particular if sophisticated algorithms are involved, such as for ADCS. Different strategies, such as HIL-based simulations are discussed and pure software-based simulation frameworks are presented (see section 1.1). The motivation for the new CubeSim framework, particularly developed for the simulation of small spacecraft, is explained and justified. The problems that shall be solved with this framework are addressed in this chapter and based on these, the objectives of this thesis are listed (see section 1.2).

In chapter 2 the implementation details of CubeSim are addressed. The modern programming techniques, that are being used, are shown in detail, such as the object-oriented design approach, the use of special member functions, the strong use of inheritance and polymorphism, generic programming and exception handling (see section 2.1). Different approaches for multi-threading and the operation system-specific implementation of processes, threads, coroutines and fibers are discussed. The concept of simulated real-time is introduced allowing the quasi-parallel execution of multiple tasks, such as required for the modeling of sub-system behavior (see section 2.2). The performance of CubeSim is discussed and various improvements, such as tweaking of module settings or data caching techniques, are explained to reduce the CubeSim simulation duration (see section 2.3).

Chapter 3 lists a variety of helper classes that were exclusively developed by the author for the support of different systems and modules of CubeSim. The generic container `List<T>` was implemented to allow storing and accessing similar components, such as parts or systems, in a unified way (see section 3.1). Two-dimensional and three-dimensional vectors `Vector2D` and `Vector3D` as well as matrices `Matrix3D` are often used for mathematical computations performed in systems and modules (sections 3.2, 3.3 and 3.6). Grids classes `Grid2D` and `Grid3D` were implemented to support modules such as `Light` or `Albedo` by creating a mesh over a circular or spherical surface (sections 3.4 and 3.5). The rotation formalism

being used in CubeSim is based on rotation matrices and the functionality is provided by class `Rotation`. The class also allows to combine rotations and to convert into Euler notation (see section 3.7). The Euler equations are used for the numeric propagation of rotating objects and momenta of inertia are represented by class `Inertia` which also allows the combination of multiple rigid bodies such as being done in an assembly (see section 3.8). Forces of type `Force` and torques of type `Torque` are generated by sub-systems, such as thrusters, magnetorquers or reaction wheels. These are acting on a rigid body and are combined to a single object of type `Wrench` (sections 3.9, 3.10 and 3.11). To deal with GNSS coordinates, class `Location` allows transformations between spherical and Cartesian coordinate systems where the ablation of celestial bodies is considered (see section 3.12). For the computation of the Kepler orbital elements describing the trajectory of a spacecraft orbiting a celestial body, class `Orbit` was developed. It also allows to compute the velocity and position of an object which are necessary for the initialization of its state vectors (see section 3.13). Materials of type `Material` can be assigned to parts. Though custom materials can be created, commonly used materials for space applications are already predefined (see section 3.15). Classes `Color` and `CAD` allow the export of sub-systems or spacecraft to the open X3D format which can be imported into CAD software for verification and further processing (sections 3.17 and 3.16). Commonly used physical and mathematical constants are collected in class `Constant` which are accessible to the user (see section 3.18).

In chapter 4 the concept of modules to represent physical behavior is introduced. When setting up a simulation, the user can decide which modules to use and which to discard. This allows to reduce the simulation complexity and consequently the simulation duration. Module `Gravitation` continuously computes the attractive forces between celestial bodies and spacecraft and in combination with module `Motion`, these modules propagate the movement and rotation of all rigid bodies (sections 4.1 and 4.5). Module `Light` allows to compute the thermal radiation power from stars where the shading by celestial bodies is considered. Combined with module `Albedo`, also the light reflected from a planet can be taken into account. This is particularly important for the performance assessment of solar sensor-based ADCS algorithms (sections 4.2 and 4.3). The magnetic field at any point can be computed by the use of module `Magnetics` which relies on models such as the IGRF of the Earth. This allows to properly model magnetic field sensors and magnetorquers (see section 4.4). For a CubeSim simulation, the start time can arbitrarily be defined. To initialize the position and velocity of the simulated celestial bodies, such as the planets in our solar system, module `Ephemeris` can be used (see section 4.6).

The framework uses different coordinate systems to define the position and rotation of rigid bodies. These are defined in chapter 5. The body coordinate system describe properties of the body itself which do not change with the movement or the rotation of the body (see section 5.1). The local coordinate system is relevant for groups of bodies, such as for assemblies. In that case the body coordinate system of the parent is called local frame for their child objects (see section 5.2). The global or celestial coordinate system is used to represent the location of the celestial bodies and spacecraft in the solar system (see section 5.3). To facilitate the the specification of the location of objects with respect to the Earth surface, the body-fixed ECEF coordinate system is used (see section 5.4). Though the ECI coordinate system also has its origin at the center of mass of the Earth, it is fixed with respect to the stars and is commonly used to describe orbital motion of spacecraft around the Earth (see section 5.5).

In chapter 6 the class `RigidBody` is defined which implements the mechanical concept of rigid bodies to which multiple forces and torques can be applied. Celestial bodies, spacecraft or parts are derived from rigid bodies. The class defines various physical properties, such as mass, volume, surface area or moment of inertia, which are implemented in different ways. Sophisticated caching techniques were developed by the author to increase the simulation performance by omitting the necessity to recompute physical properties that have not changed (see section 6.1).

In chapter 7 parts are introduced which represent basic physical components. Different geometric primitives are predefined such as boxes of type `Box`, cones of type `Cone`, cylinders of type `Cylinder`, prisms of type `Prism` and spheres of type `Sphere`. Since parts are derived from rigid bodies, they are required to define physical properties, such as their surface area, volume, the center of mass and the moment of inertia. Also the method `_contains` is implemented to check whether a point lies within the shape.

Parts can be used to build more complex arrangements, so-called *assemblies* which are explained in chapter 8. An assembly can also consist of one or more sub-assemblies allowing a similar hierarchical concept as being used in most CAD systems. This allows the user to combine and reuse created assemblies.

In chapter 9 the concept of systems is introduced. A system is an entity that is either modeling physical behavior or which represents the behavior of an intelligent component. Systems consist of one or more assemblies and are derived from class `Behavior` which allows to individually model their behavior. Multiple systems are predefined in CubeSim. A GNSS transponder is defined in class `GNSS` which represents a device to determine the location of the spacecraft in the ECEF frame (see section 9.1). A three-axis gyroscope is used to measure the rotation around three axes and is defined in class `Gyroscope` (see section 9.2). In combination with three-axis accelerometers of type `Accelerometer`, ADCS algorithms allow to propagate the position and orientation of a spacecraft or missile (see section 9.3). Photo detectors, such as sun sensors, are used to determine the orientation of the spacecraft. They are defined in class `Photodetector` and use the aforementioned modules `Light` and `Albedo` (see section 9.4). Class `Magnetometer` defines three-axis magnetic field sensors that help to determine the orientation of a spacecraft in LEO orbit by comparison with the local Earth magnetic field (see section 9.5). Class `Magnetorquer` also relies on module `Magnetics` and allows the user to model a magnetorquer coil which creates a magnetic dipole that interacts with the ambient magnetic field and generates a torque (see section 9.6). In order to model propulsion systems, class `Thruster` was developed. Such a thruster exerts a linear controllable force (see section 9.7). Similar to magnetorquers, reaction wheels are commonly used to rotate or align a spacecraft in the absence of an ambient magnetic field. Reaction wheels are defined in class `ReactionWheel` (see section 9.8).

A spacecraft, as explained in chapter 10, comprises of multiple systems. Similar to systems, spacecraft are also derived from class `Behavior` which makes it possible to assign them routines that, e.g., monitor or log the performance of all systems. Forces and torques acting on certain parts of the spacecraft result in a movement or rotation of the spacecraft by the use of module `Motion`. The ISS as well as the Hubble space telescope are predefined in the CubeSim framework.

Celestial bodies are introduced in chapter 11 and are used to investigate their influence on the spacecraft because of their gravitational or magnetic field. Modules `Gravitation` and `Motion` allow numerical computation of the celestial body trajectories by solving the n-body problem iteratively. The Sun of type `Sun` and all planets in the solar system `Mercury`, `Venus`, `Earth`, `Mars`, `Jupipter`, `Saturn`, `Uranus`, `Neptune` and `Pluto` as well as the Earth moon `Moon` are predefined. Their position and velocity are initialized by module `Ephemeris` at the begin of a simulation run.

The simulation object is defined in chapter 12 and represents the top-level object of type `Simulation` in CubeSim. Multiple spacecraft, modules and celestial bodies can be inserted into a single simulation object. The user can decide which objects to be used for the simulation run to increase the performance and reduce computation time. The simulation can be started and stopped at any point in virtual real-time.

13.2 Review of the Objectives

For the QB50 project, the University of Applied Sciences Wiener Neustadt provided a contribution with the 2U CubeSat PEGASUS. Within the scope of the aerospace engineering study program, this project should also be used to build up competencies in the field of small satellite development. Therefore, most components and systems were developed and test in-house including the ADCS system consisting of variety of sensors and actuators. Due to its high complexity and internal dependencies, a purely experimental approach for the verification of the algorithm was excluded from the beginning. A simulation framework needed to be used instead.

One objective of the thesis is the functionality to model the sensors and actuators of the spacecraft. The multi-protocol GNSS tracker Novatel OEM615 was used to determine the location of the spacecraft PEGASUS at any time as required for the operation of the specified payload [89]. The system `GNSS` was developed to allow modeling of the specific GNSS transponder. In order to verify its functionality, a test was carried out and the results can be found in fig. 9.1 and 9.2 in section 9.1.2. The simulation also allowed to evaluate the propagation accuracy of the algorithm being used since the tracker needed to be periodically disabled to preserve energy.

To measure the spin rate of the spacecraft, the digital MEMS-based three-axis gyroscope L3G4200D by STMicroelectronics was used. This information is of particular importance with regard to the implementation of the detumbling algorithm. The system `Gyroscope` was developed to allow modeling of the specific gyroscope. A test was conducted to compare the angular rate of the spacecraft with the sensor reading and the results are shown in fig. 9.4 in section 9.2.2.

The three-axis accelerometer ADXL330 manufactured by Analog Devices was used for PEGASUS to support the gyroscope readings. For this purpose the system `Accelerometer` was developed to allow modeling of the specific accelerometer and several tests were carried out to verify the functionality of the system. The accelerometer was placed outside the center of mass and the spacecraft was put in rotation around its longitudinal axis. The centripetal force was measured as shown in fig. 9.6 in section 9.3.2. Then the simulated spacecraft was equipped with a thruster and a uniform force of 200 mN was applied. The linear acceleration was detected by the sensor and the results can be seen in fig. 9.7. In the last test, the spacecraft was put on a stable LEO orbit where the centripetal and gravitational forces are balanced out. The measurement results in this microgravity environment are shown in fig. 9.8.

The ambient magnetic field was acquired by the digital three-axis magnetometer type HMC5983 manufactured by Honeywell. This helped the ADCS algorithm to determine the orientation of the spacecraft with respect to the Earth. The system `Magnetometer` was developed to allow modeling of the specific magnetometer. In order to verify the functionality of the modeled sensor, the spacecraft was again put on a stable LEO orbit and the local magnetic flux density was compared with the sensor readings in consideration of the spacecraft orientation. The results can be found in fig. 9.13 and 9.14 in section 9.5.2.

Photo detectors type TEMD6200FX01 manufactured by Vishay Semiconductors are sensitive in the visible light range and were used on each of the six side panels to determine the direction to the Sun. In order to model these sensors, the system `Photodetector` was developed. Similar to the PEGASUS configuration, the simulated spacecraft was equipped with six photo detectors and the spacecraft was put in rotation around all three axes. From the six signals, the Sun direction was computed in the body frame of the spacecraft and compared with the actual direction. The results can be found in fig. 9.10 and 9.11 in section 9.4.2

Spacecraft PEGASUS solely used magnetorquers for detumbling and to modify its orientation. In contrast to commercially-available solenoid or air coil-based systems, flat coils (also known as *pancake coils*) were directly integrated into multi-layer side panel PCBs. It was possible to equip five out of six side panels

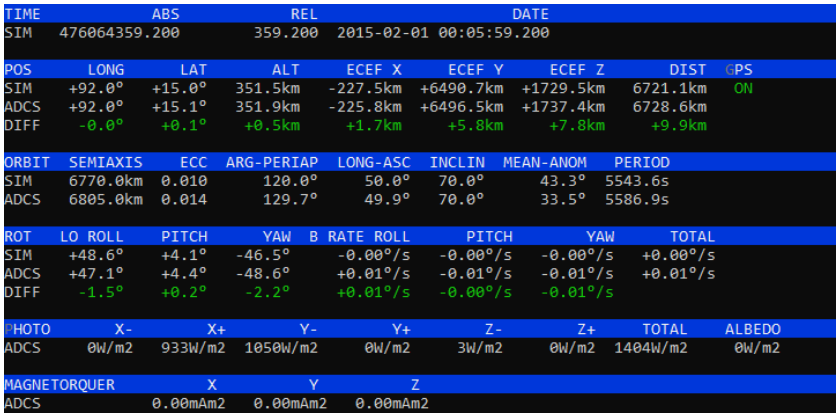

Fig. 13.1. CubeSim user interface for the verification of the ADCS algorithm for spacecraft PEGASUS.

with magnetorquer coils. This novel approach not only saved mass and volume to a considerable extent but also facilitated integration. The system `Magnetorquer` was developed to allow modeling of flat air coils with different number of turns and physical dimensions. For verification, a 1U CubeSat with three orthogonal magnetorquers was simulated. The spacecraft was put into rotation around all three axis and a B-dot controller was implemented to detumble it. The results are shown in fig. 9.16 in section 9.6.2.

For demonstration purposes, the spacecraft was also equipped with a µPPU-based propulsion system consisting of four individually controllable thruster heads which allowed thrust steering and rotation control around two axes. It should be mentioned that the thrusters were not intended for attitude control. In order to model thrusters or propulsion systems in general, the system `Thruster` was developed. The class allows to model different technologies, such as electric or chemical thrusters. For the verification of the system, a 2U CubeSat with an IFM Nano Thruster was simulated. Two scenarios were simulated, namely an orbit raise in which a constant thrust was applied in the direction of flight, and secondly an inclination change maneuver where the thrust was applied perpendicular to the direction of flight. The results can be found in fig. 9.19 and 9.20 in section 9.7.2.

With the help of the CubeSim framework all sensors an actuators of PEGASUS could successfully be modeled. The developed ADCS algorithm used the sensor inputs to determine the position and orientation of the spacecraft and allowed to control the magnetorquers to modify the attitude of the spacecraft. The algorithm was implemented in C++ and used a HAL (Hardware Abstraction Layer) to interface with the hardware. In order to run CubeSim simulations, only the HAL needed to be modified but the algorithm itself did not require any adaptation. This not only simplified the tests that were carried out but also reduced the risk of introduced errors.

Fig. 13.1 shows the UI (User Interface) developed to verify the ADCS algorithm of spacecraft PEGASUS. The lines starting with `SIM` denote the output from CubeSim and the lines starting with `ADCS` result from the ADCS algorithm. Line `DIFF` shows the differences between these sets of values. The first section shows general information on the running simulation such as the virtual real-time and the seconds elapsed since startup. The second section shows the location of the spacecraft in ellipsoidal coordinate system and in the Cartesian ECEF frame. In the next section the orbit of the spacecraft is computed from the state vectors and the Keplerian orbit elements are shown. Section four shows the deviation from the nominal Nadir-pointing mode and the angular rates of the spacecraft. The section below shows the measurement values of the photo detectors attached to the six side panels and the last section shows the commanded momenta of the magnetorquers.

Spacecraft PEGASUS was launched with a PSLV-XL C-38 rocket from Satish Dhawan Space Center on 23 June 2017. With an apogee of 516 km and a perigee of 501 km, the satellite entered a high-

inclination orbit of 97.4° with an orbital period of 94.6 min [150]. For acceptance testing, the photo detectors were tested and calibrated by the use of a solar simulator resulting in a max. output signal of ca. 1360 W/m^2. After launch during the commissioning process of all subsystems, raw data of all sensors, such as magnetometers, gyroscopes or photo detectors, was obtained, transmitted to the ground station and analyzed. The total irradiance as a combination of all photo detectors showed no more than ca. 600 W/m^2. The ADCS algorithm compares this measured irradiance with the threshold of 680.4 W/m^2 and therefore erroneously concludes that the satellite is in eclipse. Without knowing the direction to the sun, the algorithm is not able to compute the orientation of spacecraft PEGASUS and therefore does not activate any actuators. Unfortunately, since it was not possible to correct the threshold value or to update the ADCS code, the algorithm could not be verified in space. It should be mentioned that the calibration and verification of the photo detectors was not within the author's scope of work

In summary, it can be concluded that all thesis objectives as being defined in section 1.2.2 were fully met.

13.3 Limitations and Extensions

The major objective of this thesis was the development, presentation, justification and verification of the CubeSim framework. The essential components and their interaction were not only explained theoretically, but also demonstrated by means of practical case studies. The current set of modules, systems, parts and materials allowed the simulation of all sensors and actuators of spacecraft PEGASUS and of the developed ADCS algorithm [203].

Though CubeSim can be used in the current state of development to model most CubeSats and small satellites, even if they are equipped with reaction wheels and propulsion systems, it is well understood that a simulation software can never cover all possible use cases. Since the framework was developed and implemented in modern C++, it allows the user to develop new modules or systems to further extend the capabilities or to customize the framework for specific missions (see chapter 2).

In this section, the current state of the CubeSim framework is presented. On the one hand its limitations are highlighted, but on the other hand, possible ways to overcome these are pointed out.

13.3.1 Rigid Bodies

As shown in chapters 6 to 10, a spacecraft consists of one or multiple systems which in turn consist of one or more assemblies which again consist of physical parts. Parts, assemblies and systems represent rigid bodies and can be arbitrarily arranged in terms of position and rotation. These modifications can also be performed while a simulation is running. This allows the simulation of rotatable solar arrays or gimbals for thrust vector control or thrust steering [66, p. 3ff]. Rigid bodies do not deform regardless of external forces or torques acting on it. This limitation plays only a minor role, since the forces to be expected generally do not lead to any mechanical deformation.

The satellite itself consists of discrete individual parts and no continuous media such as gases or liquid can be modeled. This prevents direct simulation of thermal control systems which use coolants or chemical propulsion systems where propellant storage, feeding, combustion and finally ejection processes take place. Thruster or entire propulsion systems can nevertheless be simulated as shown in section 9.7 where the thruster force acts on a mechanical part of the thruster assembly, such as the nozzle. The propellant consumption of electric, chemical or cold-gas propulsion systems can be modeled by continuous reduction of the remaining propellant amount in dependence of the generated thrust level and mass efficiency [195, p. 5].

Though liquids and flows cannot be simulated with CubeSim, thermal control systems can be simulated by modeling the main components using the thermodynamic potential, coolant flows, thermal resistance, conductance and capacitance similar to an electrical system [121, p. 3]. This set of linear and non-linear equation can be solved by an iterative method, such as Newton-Raphson, at each time step as it is implemented in SPICE-based (Simulation Program with Integrated Circuit Emphasis) simulation applications [249].

13.3.2 Missing Components

As already pointed out in previous chapters, most of the systems and modules were developed and implemented to allow the simulation of spacecraft PEGASUS. It is fully understood that there is a variety of additional CubeSat or small satellite components and sub-systems that cannot directly be modeled in the framework at the moment. These include for example solar cells, batteries, power processing systems, transceivers, horizon sensors, star trackers, cameras, solar sails or other scientific or technical payloads. Nevertheless, CubeSim was designed in a way to allow the straightforward development and implementation of missing systems when required.

The commonly used geometric primitives were implemented to allow the user to set up more complex assemblies in a hierarchical manner. More complex shapes, such as truncated cones, hemispheres, non-star-shaped prisms or other Platonic solids, are not supported at the moment. Similar to the extensibility of systems, the user can easily define new shapes.

The same applies to predefined materials as listed in tbl. 3.1 in section 3.15. New materials can easily be introduced or derived from existing ones and later be used for different parts of an assembly.

All planets in our solar system are predefined but for specific mission scenarios, the moons of other celestial bodies or asteroids might need to be defined. In order to model star trackers, a comprehensive star catalog would need to be added to the simulation. Commercial systems use a few hundred up to more than $20,000$ stars to determine the orientation [44, 27].

13.3.3 Physical Models

The modules already developed are suitable for a variety of CubeSat simulation scenarios, but it is also obvious that certain physical effects cannot be taken into account at the moment. These are for example atmospheric drag, solar pressure, season- or even weather-dependent albedo or the thermal modeling of the spacecraft itself as a balance of Sun irradiation, power dissipation and thermal radiation. The accurate modeling of atmospheric drag is sophisticated and strongly dependent on the location, orientation and shape of the spacecraft and even on the solar activity [190, p. 163], [188, p. 3]. Atmospheric effects can cause instabilities of the ADCS algorithm [193, 231] and the density strongly increases at altitudes below 300 km [30, p. 3].

It is not possible to compute the ground station coverage or the link budget of a communication path with CubeSim. This would require a module which can model the properties of the transmission and reception antennas, the propagation, noise and interference caused by atmospheric distortions and other characteristics of the satellite such as the shape or properties of the outer hull [123]. There are specialized tools available, such as *HispaSim* or *Satellite Link Budget Analyzer* that are optimized for solving this problem [84, 6].

Also existing modules could be improved and refined, such as module `Gravitation` computes the gravitational forces based on point masses (see section 4.1). This approach is valid for spacecraft but for celestial bodies certain effects, such as the gravitational bulge of the Earth leading to orbital perturbations, are not considered at the moment [75].

The physical electromagnetic radiation model on which module `Light` is based, considers stars as ideal black body radiators (see section 4.2). The difference between the spectral distribution according to Planck's law as stated in eq. (1.1.1) and the actual spectrum is shown in fig. 1.4 in section 1.1.1. For other stars the difference is even more significant as compiled by R. Walker [248, p. 7]. Due to the low atmospheric density of $< 5 \times 10^{-11}\,\mathrm{kg/m^3}$ at an altitude of 300 km [188, p. 3], spectral changes due to the Earth's atmosphere can be neglected in most cases.

Though magnetometers and magnetorquers can be simulated by the use of module `Magnetics`, for both systems only the ambient magnetic flux density is used and no local distortions arising from ferromagnetic materials or active magnetorquers are considered (see section 4.4). The magnetorquers integrated into the side panels of spacecraft PEGASUS generate local magnetic flux densities of a few µT which could compromise the reading of the nearby magnetometers. Module `Magnetics` would need to be extended to include local current paths of coils, conductors and solar cells to better compute the more realistic magnetic flux density. However, the inclusion of ferromagnetic materials and the related non-linear distortion effects would be difficult to simulate. Dedicated FEM (Finite Element Method) solvers appear to be better suited to address these problems [42].

When systems such as sensors and actuators are modeled, their measurement value can be read out or their output value can be commanded immediately without any lag. This is a simplification that does not apply unconditionally to all systems. Sensors with integrated A/D converters feature a certain bandwidth and update rate and also the data transfer to the OBC via a digital communication interface and the data processing by the OBC take a finite time. For sensitive control loops, such as detumbling controllers, the missing lag time and thus the missing phase shift of the feedback signal can significantly affect the control loop performance. These lags can partly be considered by using method `delay` of class `Simulation` in method `_behavior` of the respective system.

13.4 Outlook

As pointed out in the previous section, sensor signals and actuator commands are abstracted and the underlying layers for data acquisition, signal processing and data transmission are not considered. The CubeSim framework could be extended to provide digital communication interfaces such as SPI, I2C or UART or simple GPIO (General Purpose Input Output) lines. This would allow to test and verify low level drivers that need to process, convert and calibrate sensor signals. Also the aforementioned communication lag would be considered that way. For the satellite PEGASUS, these hardware drivers were tested separately which significantly increased the testing efforts.

Since small spacecraft are limited in power and battery capacity, a proper energy household is crucial for a successful mission. The power draw of a system could be computed in dependence of its actual operation state and the battery charge could be updated in simulated real-time. This would allow to test different mission scenarios, such as scientific operation inside and outside of eclipse. Also the use of high electric loads such as heaters or electric thrusters could be simulated to check in order to check if the capacity of the used batteries is sufficient.

The source code of the CubeSim framework is published under the MIT software license which allows free use for any purposes [220, p. 14f]. This shall allow the CubeSat and small satellite community to use and extend the functionality of the framework. Though the framework was carefully developed and extensively tested as shown in detail in this thesis, any remaining issues and bugs could be discovered more easily if the software is used by a broader community for a variety of applications.

`https://github.com/bernhard-seifert/CubeSim`

Bibliography

[1] *HTML True Color Chart.*

[2] *Explanatory Supplement to the Astronomical Ephemeris and the American Ephemeris and Nautical Almanac.* Nautical Almanac Office, U.S. Naval Observatory, H.M. Nautical Almanac Office, Royal Greenwich Observatory, 1961.

[3] *NMEA Reference Manual.* SiRF Technology, Inc., 2005.

[4] *SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems.* Software Engineering Institute, Carnegie Mellon University, 2016.

[5] Actel Corporation. *Test Vector Guidelines.* Application Note AC189, 2003.

[6] Adil Hussein M. Al-Dalowi, , Abdulqadir Ismail Khoshnaw, and Ghassan A. QasMarrogy. Satellite link budget calculator by using matlab/GUI. *Cihan University-Erbil Scientific Journal*, 2017(Special-1):160–171, 2017.

[7] A. Alexandrescu. *Modern C++ Design, Generic Programming and Design Patterns Applied.* Addison Wesley, 2001.

[8] H. Ali, M. R. Mughal, Q. Islam, J. Praks, and L. M. Reyneri. *Analysis and Design of Integrated Magnetorquer Coils for Attitude Control of Nanosatellites.* 11th European CubeSat Symposium, 2020.

[9] Z. Altamimi. *Global Terrestrial Reference Systems and Frames.* 2012.

[10] Analog Devices. *ADXL330 Datahseet, Small low Power 3-Axis ±3g iMEMS Accelerometer.* 2007.

[11] C. Ananda and N. Bartel. *CaNOP 3U CubeSat Attitude Determination and Control Testing System: Helmholtz Cage Design.* Carthage College, Kenosha WI, 2016.

[12] D. Anderson, D. Greer, B. Hutchinson, K. Lee, A. Levandoski, A. Mezich, S. O'Donnell, Z. Reynolds, K. Sample, C. Sheahan, P. Sieira, and Z. Toelkes. *Lockheed Martin's sateLLight Adcs fault MAnagement System (LLAMAS) - Conceptual Design Document.* Senior Project ASEN 4018, Department of Aerospace Engineering Sciences, University of Colorado, 2017.

[13] S. Andrilli and D. Hecker. *Elementary Linear Algebra, Fourth Edition - Prerequisite: Section 3.1, Introduction to Determinants.* Elsevier, 2010.

[14] Atmel Corporation. *Datasheet: ATmega128A, 8-bit AVR Microcontroller with 128KBytes In-System Programmable Flash.* 2011.

[15] Aurubis AG. *Material Datasheet: Cu-ETP.*

[16] G. Avanzini and F. Giulietti. *Magnetic Detumbling of a Rigid Spacecraft.* Guidance, Control, and Dynamics, Vol. 35, No. 4, pp. 1326-1334, 2012.

[17] David H. Bailey. *Pi and normality.* 2017.

[18] A. Ballman. *SEI CERT C++ Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems in C++.* Software Engineering Institute, Carnegie Mellon University, 2016.

[19] M. Barr. *Programming Embedded Systems in C and C++.* O'Reilly, 1999.

[20] M. Barr. *BARR-C:2018: Embedded C Coding Standard.* Barr Group, 2018.

[21] M. Bass. *Handbook of Optics, Volume 1 - Fundamentals, Techniques, and Design, Second Edition.* McGraw-Hill, Inc., 1995.

[22] M. J. Bedy, S. Carr, X. Huang, and C.-K. Shene. *The Design and Construction of a User-Level Kernel for Teaching Multithreaded Programming.* 29th ASEE/IEEE Frontiers in Education Conference, 1999.

[23] M. Ben-Ari. *A Tutorial on Euler Angles and Quaternions.* 2017.

[24] K. Betke. *The NMEA 0183 Protocol.* 2001.

[25] S. M. Bezick, A. J. Pue, and C. M. Patzelt. *Inertial Navigation for Guided Missile Systems.* Johns Hopkins APL Technical Digest, Vol. 28, No. 4, 2010.

[26] D. D. V. Bhanderi and T. Bak. *Modeling Earth Albedo for Satellites in Earth Orbit.* Proceedings of AIAAConference on Guidance, Navigation and Control AIAA, 2005.

[27] Blue Canyon Technologies. *Datasheet: Star Trackers.* 2021.

[28] B. R. Bowring. *Transformation from spatial to geographical Coordinates.* 1976.

[29] M. R. Brewer. *CubeSat Attitude Determination and Helmholtz Cage Design.* Theses and Dissertations. 1030, Air Force Institute of Technology, 2012.

[30] T P Brito, C C Celestino, and R V Moraes. Study of the decay time of a CubeSat type satellite considering perturbations due to the earth's oblateness and atmospheric drag. *Journal of Physics: Conference Series*, 641:012026, oct 2015.

[31] California Polytechnic State University. *CubeSat Design Specification Rev. 13.* 2014.

[32] A. Calletti. *Orbital Determination.* 2011.

[33] Michel Capderou. *Satellites, Orbits and Missions.* Springer, 2005.

[34] S. F. R. Carná and R. Bevilacqua. *High fidelity model for the atmospheric re-entry of CubeSats equipped with the Drag De-Orbit Device.* Acta Astronautica 156 (2019) 134-156, 2018.

[35] S. Carr, J. Mayo, and C.-K. Shene. *Race Conditions: A Case Study.* Department of Computer Science, Michigan Technological University, 2001.

[36] B. R. Castello. *CubeSat Mission Planning Toolbox.* Master thesis, Faculty of California Polytechnic State University, San Luis Obispo, 2012.

[37] T. R. Chandruoatla and T.J. Osler. *Perimeter of an Ellipse.* 2010.

[38] D. Chistikov and M. Paterson. *Globe-hopping.* 2018.

[39] C. Clark and S. Chesi. *Effective Hardware-in-the-loop CubeSat Attitude Control Verification and Test.* CubeSat Developers Workshop, 2015.

[40] James R. Clynch. *Earth Coordinates.* 2006.

[41] C. Collette, P. Carmona-Fernandez, S. Janssens, K. Artoos, M. Guinchard, and C. Hauviller. *Review of Sensors for low Frequency Seismic Vibration Measurement.* CERN ATS/Note/2011/001, 2011.

[42] COMSOL. *AC/DC Module User's Guide, Version 5.4.* 2018.

[43] Sabrina Corpino and Fabrizio Stesina. Verification of a cubesat via hardware-in-the-loop simulation. *IEEE Transactions on Aerospace and Electronic Systems*, 50(4):2807–2818, oct 2014.

[44] CubeSpace. *Interface Control Document: CubeStar, Version 1.6.* 2020.

[45] Cylex Engineering Plastics Ltd. *Datasheet: FR4, Typical Properties.*

[46] Rodrigo Cardoso da Silva, Igor Seiiti Kinoshita Ishioka, Chantal Cappelletti, Simone Battistini, and Renato Alves Borges. *Helmholtz cage design and validation for nanosatellites HWIL testing.* IEEE Transactions onAerospace and Electronic Systems, 55 (6), p. 1, 2019.

[47] E. B. Dam, M. Koch, and M. Lillholm. *Quaternions, Interpolation and Animation.* Department of Computer Science University of Copenhagen, 1998.

[48] Rachid Darbali-Zamora, Daniel A. Merced Cirino, Cesar S. Gonzalez-Ortiz, and Eduardo I. Ortiz-Rivera. An electric power supply design for the space plasma ionic charge analyzer (SPICA) CubeSat. In *2014 IEEE 40th Photovoltaic Specialist Conference (PVSC).* IEEE, jun 2014.

[49] S. Dawoud. *GNSS Principles and Comparison.*

[50] W. Demtröder. *Experimentalphysik 3, 4th Edition.* Springer Spektrum, 2009.

[51] W. Demtröder. *Experimentalphysik, 6th Edition.* Springer Spektrum, 2012.

[52] W. Demtröder. *Experimentalphysik 1, 7th Edition.* Springer Spektrum, 2015.

[53] M. Desouky. *Algorithms and optimal Control for Spacecraft Magnetic Attitude Maneuvers.* PhD Thesis, 2019.

[54] M. A. A. Desouky and O. Abdelkhalik. *Efficient B-dot Law for Spacecraft Magnetic Detumbling.* AAS 19-665.

[55] J. Diebel. *Representing Attitude: Euler Angles, Unit Quaternions, and Rotation Vectors.* 2006.

[56] Direct Plastics Ltd. *Material Datasheet: Nylon 66 Natural.*

[57] E. Dobos. *Albedo.* Encyclopedia of Soil Science, 120014334, 2003.

[58] C. E. Dunn, S. M. Lichten, D. C. Jefferson, and J. S. Border. *Subnanosecond GPS-based Clock Synchronization and Precision Deep-Space Tracking.* 1992.

[59] J. A. Dutton. *The Hertzsprung-Russell Diagram.* PennState, College of Earth and Mineral Sciences.

[60] Matthew C. Dykstra. *Single Station Doppler Tracking for Satellite Orbit Prediction and Propagation.* 2015.

[61] E. Thebault et al. *International Geomagnetic Reference Field: the 12th generation.* Earth, Planets and Space, 2015.

[62] C. Eberl, M. Fromm, T. Kiley, H. Li, T. Ly, A. McBride, N. Puldon, K. Sotebeer, and M. Tilong. *Cubesat Active Systematic CApture DEvice (CASCADE) - Conceptual Design Document.* Department of Aerospace Engineering Sciences, University of Colorado, 2016.

[63] D. J. Eck. *Introduction to Programming Using Java.* Hobart and William Smith Colleges, 2014.

[64] I. Ennen, D. Kappe, T. Rempel, C. Glenske, and A. Hütten. *Giant Magnetoresistance: Basic Concepts, Microstructure, Magnetic Interactions and Applications.* MDPI Sensors, 2016.

[65] ENPULSION. *Datasheet: NANO, IFM Nano Thruster.* 2018.

[66] C. B. F. Ensworth. *Thrust Vector Control for Nuclear Thermal Rockets, AIAA-2013-4075.* 49th Joint Propulsion Conference and Exhibit, 2013.

[67] A. O. Erlank. *Development of CubeStar, A CubeSat-Compatible Star Tracker.* Department of Electrical and Electronic Engineering, Stellenbosch University, 2013.

[68] ESA Board for Software Standardisation and Control (BSSC). *Guide to software verification and validation.* ESA PSS-05-10 Issue 1 Revision 1, 1995.

[69] R. Falck and L. Gefert. *A Method of Efficient Inclination Changes for Low-Thrust Spacecraft.* NASA AIAA-2002-4895, 2002.

[70] J. A. Farrell. *Computation of the Quaternion from a Rotation Matrix.* 2015.

[71] Federal Aviation Administration. *Advanced Aerospace Medicine On-line.* 2017.

[72] Edemar Morsch Filho, Laio Oriel Seman, Cezar Antonio Rigo, Vicente de Paulo Nicolau, Raul Garcia Ovejero, and Valderi Reis Quietinho Leithardt. *Irradiation Flux Modelling for Thermal-Electrical Simulation of CubeSats: Orbit, Attitude and Radiation Integration.* MDPI Energies, 2020.

[73] R. Fonod and E. Gill. *Magnetic Detumbling of Fast-tumbling Picosatellites.* IAC-18-C1.3.11, 2018.

[74] P. Fortescue, J. Stark, and G. Swinerd. *Spacecraft Systems Engineering.* John Wiley & Sons Ltd., 2003.

[75] R. H. Frick and T. B. Garber. *Perturbations of a Synchronous Satellite, R-399-NASA.* NASA, 1962.

[76] G. Fritchey. *SQL Server 2017 QueryPerformance Tuning, 4th Edition.* Apress Media LLC, 2018.

[77] C. Fröhlich. *Total Solar Irradiance: What Have We Learned from the Last Three Cycles and the Recent Minimum?* Space Science Review, 2011.

[78] R. Galliath, O. Hasson, A. Montero, C. Renfro, and D. Resmini. *Design and Analysis of a CubeSat.* Bachelor thesis, Faculty of the Worcester Polytechnic Institute, 2020.

[79] Jack Ganssle. *A Designer's Guide to MEMS Sensors.* 2012.

[80] Irina Gavrilovich, Sébastien Krut, Marc Gouttefarde, François Pierrot, and Laurent Dusseau. *Innovative Approach to Use Air Bearings in Cubesat Ground Tests.* CubeSat Workshop, ESA, CNES, 2016.

[81] S. Goldt, S. v. d. Meer, S. Burkett, and M. Welsh. *The Linux Programmer's Guide.* 1995.

[82] Mohinder S. Grewal, Lawrence R. Weill, and Angus P. Andrews. *Global Positioning Systems, Inertial Navigation, and Integration.* John Wiley and Sons, Inc., 2001.

[83] F. Hamano. *Derivative of Rotation Matrix - Direct MatrixDerivation of Well-Known Formula.* Proceedings of Green Energy and Systems Conference, 2013.

[84] Luis Escolar Haro, Gregorio Juliana Quiros, Alvaro Rodriguez Villalba, Ramon Martinez Rodriguez-Osorio, Pablo Honold, Antonio Arana, Alejandro Martinez, and Ines Sanz. HispaSim: A web application for satellite link budget optimization and management. In *2017 11th European Conference on Antennas and Propagation (EUCAP)*. IEEE, mar 2017.

[85] E. Harokopos. *The Axiom and Laws of Motion*. 2003.

[86] G. A. Hartmann and I. G. Pacca. *Time evolution of the South Atlantic Magnetic Anomaly*. Annals of the Brazilian Academy of Sciences, 2009.

[87] Douglas C. Heggie. *The Classical Gravitational N-Body Problem*. 2005.

[88] M. Henricson and E. Nyquist. *Industrial Strength C++*. Prentice Hall PTR, 1997.

[89] H. Hoang, K. Røed, T. A. Bekkeng, J. I. Moen, L. B. N. Clausen, E. Trondsen, B. Lybekk, H. Strøm, D. M. Bang-Hauge, A. Pedersen, C. D. A. Nokes, C. Cupido, I. R. Mann, M. Ariel, D. Portnoy, and E. Sagi. The multi-needle langmuir probe instrument for QB50 mission: Case studies of ex-alta 1 and hoopoe satellites. *Space Science Reviews*, 215(2), feb 2019.

[90] Honeywell. *Datasheet: HMC5983, 3-Axis Digital Compass IC*. 2011.

[91] J. Horner, J. B. Gilmore, and D. Waltham. *The influence of Jupiter, Mars and Venus on Earth's orbital evolution*. 2015.

[92] R. Howard. *The Rotation of the Sun*. Scientific American, 1975.

[93] HSM Stahl- und Metallhandel GmbH. *Werkstoffdatenblatt 1.4301 / X5CrNi18.10, nichtrostender Stahl, austenitisch*.

[94] HSM Stahl- und Metallhandel GmbH. *Werkstoffdatenblatt EN AW-6061, AlMg1SiCu / 3.3211*.

[95] J. R. Hubbard. *Schaum's Outline of Theory and Problems of PROGRAMMING WITH C++*. McGraw-Hill, 1996.

[96] Huntsman Advanced Materials (Switzerland) GmbH. *Technical Datasheet: Araldite AV 138M-1, Hardener HV 998-1*. 2017.

[97] Hyperion Technologies. *Datasheet: RW210, Reaction Wheel*. 2019.

[98] IEEE. *610.12-1990 - IEEE Standard Glossary of Software Engineering Terminology*. The Institute of Electrical and Electronics Engineers, 1990.

[99] IERS. *Technical Note No. 32, General Definitions and Numerical Standards*.

[100] instantreality. *Instant Player version 2.8, https://www.instantreality.org/downloads/*. 2016.

[101] International Civil Aviation Organization. *World Geodetic System - 1984 (WGS-84) Manual, Second Edition*. 2002.

[102] K. Iqbal. *Introduction to Control Systems*. University of Arkansas at Little Rock, 2021.

[103] ISO/IEC. *14882:1998(E), International Standard, Programming Languages - C++*. American National Standards Institute, 1998.

[104] ISO/IEC. *9899:1999 (E), International Standard, Programming Languages - C*. InterNational Committee for Information Technology Standards, 1999.

[105] ISO/IEC. *19775-1:2013(E), Information technology - Computergraphics, image processing andenvironmental data representation, Extensible 3D (X3D), Part 1: Architecture and base components.* 2013.

[106] ISO/IEC. *14882:2017(E), International Standard, Programming Languages - C++.* American National Standards Institute, 2017.

[107] D. Ivanov, M. Koptev, Y. Mashtakov, M. Ovchinnikov, N. Proshunin, S. Tkachev, A. Fedoseev, and M. Shachkov. Determination of disturbances acting on small satellite mock-up on air bearing table. *Acta Astronautica*, 142:265–276, jan 2018.

[108] D. P. Jacobs and V. Trevisan. *Linear-Time LUP Decomposition of Forest-like Matrices.* 1998.

[109] D. Jelem, A. Reissner, C. Scharlemann, B. Seifert, N. Buldrini, F. Plesescu, and T. Hörbe. *IFM Nano Thruster.* 68th International Astronautical Congress, 2017.

[110] D. Jelem, B. Seifert, N. Buldrini, T. Hörbe, A. Reissner, and T. Vogel. *Performance Mapping and Qualification of the IFM Nano Thruster FM for in Orbit Demonstration.* 35th International Electric Propulsion Conference, 2017.

[111] Y. Jia. *Rotation in the Space.* 2017.

[112] D. Jones. *The Definitive Guide To SQL Server Performance Optimization.* realtimepublishers.com, 2002.

[113] G. Kazokaitis, V. Jurenas, and D. Eidukynas. *Research and Analysis of spherical Magnetic Drive for Attitude Control on Nano Satellites.* JVE International Ltd. Vibroengineering Procedia, Vol. 15, 2017.

[114] Shaun Kenyon, Christopher Bridges, Doug Liddle, Bob Dyer, James Parsons, David Feltham, Rupert Taylor, Dale Mellor, Andrew Schofield, Rosie Linehan, Richard Long, Juan Fernandez, Haval Kadhem, Phil Davies, Jonathan Gebbie, Nick Holt, Peter Shaw, Lourens Visagie, Theodoros Theodorou, Vaios Lappas, and Craig Underwood. *STRAND-1: Use of a $500 Smartphone as the central Avionics of a Nanosatellite.* 62nd International Astronautical Congress, 2011.

[115] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, 2nd Edition.* Prentice Hall Software Series, 1988.

[116] G. Kerschen. *Mission analysis of QB50, a nanosatellite intended to study the lower thermosphere.* University of Liège, 2011.

[117] N. S. Khalifa and T. E. Sharaf-Eldin. *Earth Albedo perturbations on Low Earth Orbit Cubesats.* International Journal of Aeronautical and Space Sciences 14(2), 193-199, 2013.

[118] Jonis Kiesbye, David Messmann, Maximilian Preisinger, Gonzalo Reina, Daniel Nagy, Florian Schummer, Martin Mostad, Tejas Kale, and Martin Langer. Hardware-in-the-loop and software-in-the-loop testing of the move-ii cubesat. *Aerospace*, 6(12):130, dec 2019.

[119] Sedong Kim, Hyomin Jeong, Jin Young Park, Seung Yeop Baek, Ajeong Lee, and Soon-Ho Choi. *Innovative flat-plate solar collector (FPC) with coloured water flowing through a transparent tube.* RSC Adv., 2019, 9, 24192, Royal Society of Chemistry, 2019.

[120] D. G. King-Hele and R. H. Merson. *A New Value for the Earth's Flattening, derived from Measurements of Satellite Orbits.* Nature 183, 881-882, 1959.

[121] K&K Associates. *Thermal Network Modeling Handbook, Version 97.003.* K&K Associates, Developers of Thermal Analysis Kit (TAK), 1999-2000.

[122] S. Kokado and M. Tsunoda. *Anisotropic Magnetoresistance Effect: General Expression of AMR Ratio and Intuitive Explanation for Sign of AMR Ratio.*

[123] O. Koudelka. *Link Budget Calculations.* Institute of Communication Networks and Satellite Communications, TU Graz, 2015.

[124] D. Krejci, V. Hugonnaud, T. Schönherr, B. Little, A. Reissner, B. Seifert, Q. Koch, E. B. Borràs, and J. González del Amo. *Full Performance Mapping of the IFM Nano Thruster including Direct Thrust Measurements.* JoSS Journal of Small Satellites Vol. 8, No. 2, pp. 881-893, 2019.

[125] D. Krejci, B. Seifert, and C. Scharlemann. *Endurance testing of a pulsed plasma thruster for nanosatellites.* Acta Astronautica Volume 91, October-November 2013, Pages 187-193, 2013.

[126] R. Laforne. *Object-Oriented Programming in C++, 4th Edition.* Sams Publishing, 2002.

[127] Giacomo Laghi, Antonio F. Longoni, Paolo Minotti, Alessandro Tocchio, and Giacomo Langfelder. *100 μA, 320 nT/$\sqrt{}$Hz, 3-Axis Lorentz Force MEMS Magnetometer.*

[128] K. M. Laundal and A. D. Richmond. *Magnetic Coordinate Systems.* Space Science Reviews, 2016.

[129] LDRA Ltd. *CERT-C Standards Model Summary for C.* 2021.

[130] M. Leipold, D. Kassing, M. Eiden, and L. Herbeck. *Solar Sails for Space Exploration - The Development and Demonstration of Critical Technologies in Partnership.* ESA bulletin 98, 1999.

[131] Dan R. Lev, R. Zimmerman, B. Shoor, L. Appel, M. Ben-Ephraim, J. Herscovitz, and O. Epstein. *Electric Propulsion Activities at Rafael in 2019.* 36th International Electric Propulsion Conference, 2019.

[132] J. Liberty. *C++ Unleashed.* Sams, 1999.

[133] M. Ligas and P. Banasik. *Conversion between Cartesian and geodetic coordinates on a rotational ellipsoid by solving a system of nonlinear equations.* 2011.

[134] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison Wesley Longman, Inc., 1997.

[135] W. S. Liu. *Comparison of the greenhouse effect between Earth and Venus using multiple atmospheric layer models.* 11th International Conference on Environmental Science and Development, 2020.

[136] R. Love. *LINUX System Programming.* O'Reilly Media Inc., 2007.

[137] Tuomas J. Lukka and John Stewart. *FreeWRL version 4.0, http://freewrl.sourceforge.net/.* 2017.

[138] J. A. Macés Hérnandez. *Design and Analysis of the Attitude Control System for the S2TEP Mission.* Master Thesis, 2017.

[139] A. Mallama, B. Krobusek, and H. Pavlov. *Comprehensive wide-band magnitudes and albedos for the planets, with applications to exo-planets and Planet Nine.* Icarus 282, Elsevier, 2017.

[140] Maryland Aerospace Inc. *Datasheet: MAI-SES, Static Earth Sensor.* 2016.

[141] G. Maymon, M. Oslin, and G. Cruz-Ortiz. *Reaction Wheel for CubeSat Attitude Control.* Planetary CubeSats/Smallsats Symposium, 2017.

[142] A. McEvoy, T. Markvart, and L. Castaner. *Practical Handbook of Photovoltaics: Fundamentals and Applications, 1st edition.* Elsevier Science, 2003.

[143] R. D. McPeters, P. K. Bhartia, A. J. Krueger, J. R. Herman, C. G. Wellemeyer, C. J. Seftor, W. Byerly, and E. A. Celarier. *Total Ozone Mapping Spectrometer (TOMS) Level-3 Data Products User's Guide.* NASA/TP-2000-209896, 2000.

[144] S. Meyers. *Effective STL - 50 Specific Ways to Improve Your Use of the Standard Template Library.* Pearson Education, Inc., 2001.

[145] P. Michael. *A Conversion Guide: Solar Irradiance and Lux Illuminance.* 2019.

[146] Dario Modenini, Anton Bahu, Giacomo Curzi, and Andrea Togni. *A Dynamic Testbed for Nanosatellites Attitude Verification.* MDPI Aerospace, 2020.

[147] O. Montenbruck and E. Gill. *Satellite Orbits: Models, Methods, Applications.* Springer, 2005.

[148] H. Moritz. *Geodetic Reference System 1980.* Journal of Geodesy, 2000.

[149] M. Mueller. *Equation of Time - Probelm in Astronomy.* Acta Physica Polonica A 88 Supplement, 1995.

[150] N2YO.com. *PEGASUS Satellite Details 2017-036V NORAD 42784.* 2017.

[151] Meiyappan Nagappan, Romain Robbes, Yasutaka Kamei, Éric Tanter, Shane McIntosh, Audris Mockus, and Ahmed E. Hassan. An empirical study of goto in c code. feb 2015.

[152] Mouaaz Nahas and Adi Maait. Choosing appropriate programming language to implement software for real-time resource- constrained embedded systems. In *Embedded Systems - Theory and Design Methodology.* InTech, mar 2012.

[153] NASA. *Launching From Florida: Life in the Fast Lane!*

[154] NASA. *Spacecraft Earth Horizon Sensors.* NASA SP-8033, 1969.

[155] NASA. *Earth Fact Sheet, https://nssdc.gsfc.nasa.gov/planetary/factsheet/earthfact.html.* 2020.

[156] NASA. *Jupiter Fact Sheet, https://nssdc.gsfc.nasa.gov/planetary/factsheet/jupiterfact.html.* 2020.

[157] NASA. *Mars Fact Sheet, https://nssdc.gsfc.nasa.gov/planetary/factsheet/marsfact.html.* 2020.

[158] NASA. *Moon Fact Sheet, https://nssdc.gsfc.nasa.gov/planetary/factsheet/moonfact.html.* 2020.

[159] NASA. *Neptune Fact Sheet, https://nssdc.gsfc.nasa.gov/planetary/factsheet/neptunefact.html.* 2020.

[160] NASA. *Pluto Fact Sheet, https://nssdc.gsfc.nasa.gov/planetary/factsheet/plutofact.html.* 2020.

[161] NASA. *Saturn Fact Sheet, https://nssdc.gsfc.nasa.gov/planetary/factsheet/saturnfact.html.* 2020.

[162] NASA. *Uranus Fact Sheet, https://nssdc.gsfc.nasa.gov/planetary/factsheet/uranusfact.html.* 2020.

[163] NASA. *Mercury Fact Sheet, https://nssdc.gsfc.nasa.gov/planetary/factsheet/mercuryfact.html.* 2021.

[164] NASA. *Venus Fact Sheet, https://nssdc.gsfc.nasa.gov/planetary/factsheet/venusfact.html.* 2021.

[165] NASA Goddard Institute for Space Studies. *ModelE AR5 Simulations: Past Climate Change and Future Climate Predictions, Time and Date of Vernal Equinox.* https://data.giss.nasa.gov/modelE/ar5plots/srvernal.html, 2020.

[166] NASA/JPL. *Horizons, Version 3.75.* 2013.

[167] National Centers forEnvironmental Information. *Geomag 7.0 software, https://www.ngdc.noaa.gov/IAGA/vmod/igrf.html.* 2019.

[168] National Geospatial-Intelligence Agency. *EGM2008 Spherical Harmonics, Data package.* 2008.

[169] National Institute of Standards and Technology. *The International System of Units (SI).* NIST Special Publication 330, 2019.

[170] National Marine Electronics Association. *NMEA 0183 - Standard For Interfacing Marine Electronic Devices, Version 3.01.* 2002.

[171] National Renewable Energy Laboratory. *ASTM G173-03 Reference Spectra Derived from SMARTS v. 2.9.2.* https://www.nrel.gov/grid/solar-resource/spectra-am1.5.html.

[172] NewSpace. *Data heet on GPS Receiver.* 2020.

[173] T. Nguyen, K. Cahoy, and A. Marinan. *Attitude Determination for Small Satellites with Infrared Earth Horizon Sensors.* Journal of Spacecraft and Rockets, Vol. 55, No. 6, 2018.

[174] G. Nishanov. *Fibers under the magnifying glass.* Open Standards, P1364 R0, 2018.

[175] NIST. *2018 CODATA Recommended Values of the fundamental Constants of Physics and Chemistry.* NIST SP 959, 2019.

[176] NovAtel. *Data heet on OEM615 GNSS Receiver.* 2014.

[177] D. L. Oltrogge and K. Leveque. *An Evolution of CubeSat Orbital Decay.* 25th AIAA/USU Conference on Small Satellites, 2011.

[178] Dillon O'Reilly, Georg Herdrich, and Darren F. Kavanagh. *Electric Propulsion Methods for Small Satellites: A Review.* MDPI Aerospace, 2021.

[179] OROLIA SAS. *Datasheet: GSG-5/6 Series: Advanced GNSS Simulators.* 2018.

[180] B. Palais and R. Palais. *Euler's fixed point theorem: The axis of a rotation.* Birkhäuser Verlag, 2007.

[181] Jose C. Pascoa, Odelma Teixeira, and Gustavo Filipe. *A Review of Propulsion Systems for CubeSats.* Proceedings of the ASME 2018 International Mechanical Engineering Congress and Exposition IMECE, 2018.

[182] Anatoly Pelemeshko, Alena Kolesnikova, Alexander Melkov, Vitaliy Prokopyev, and Alexander Zadorozhny. *High-precision CubeSat sun sensor coupled with infrared Earth horizon detector.* Materials Science and Engineering 734, 2019.

[183] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: how do energy, time, and memory relate? ACM, oct 2017.

[184] Philtec. *User Manual: Fiberoptic Displacement Sensor with Analog Output.*

[185] A. Pianese. *The Equation of Time: Computation Formulas for Engineering Applications.* 2019.

[186] A. Pillay. *Object Oriented Programmingusing Java.* School of Computer Science, University of KwaZulu-Natal, 2007.

[187] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C The Art of Scientific Computing Third Edition.* Cambridge University Press, 2007.

[188] W. Priester. *On the Variations of the Thermospheric Structure, TN D-3167.* Goddard Space Flight Center, NASA, 1966.

[189] Meghan K. Prinkey, David W. Miller, Paul Bauer, Kerri Cahoy, Evan D. Wise, Christopher M. Pong, RyanW. Kingsbury, Anne D. Marinan, Hang Woon Lee, and Erin L. Main. *CubeSat Attitude Control Testbed Design: Merritt 4-Coil per axis Helmholtz Cage and Spherical Air Bearing.* AIAA Guidance, Navigation, and Control (GNC) Conference, 2013.

[190] Liying Qian and Stanley C. Solomon. Thermospheric density: An overview of temporal and spatial variations. *Space Science Reviews*, 168(1-4):147–173, aug 2011.

[191] RACELOGIC Ltd. *Datasheet: LabSat 3 GNSS Simulator, v3.0.* 2019.

[192] G. S. Rao. *Global Navigation Satellite Systems.* Tata McGraw-Hill, 2010.

[193] S. A. Rawashdeh and J. E. Lumpp. *Aerodynamic Stability for CubeSats at ISS Orbit, JoSS, Vol. 2, No. 1.* A. Deepak Publishing, 2013.

[194] M. Razeghi. *Fundamentals of Solid State Engineering.* Kluwer Academic Publishers, 2002.

[195] A. Reissner, N. Buldrini, B. Seifert, T. Hörbe, F. Plesescu, J. Gonzalez del Amo, and L. Massotti. *Detailed Performance Characterization of the mN-FEEP Thruster.* American Institute of Aeronautics and Astronautics, 2014.

[196] Nicolas Le Renard, Alex Nichols, Jordan Skaro, Louie Thiros, and Madeline Tran. *PolySat Helmholtz Cage.* College of Engineering, California Polytechnic State University, San Luis Obispo, 2017.

[197] J. Richter and C. Nasarre. *Windows via C/C++, Fifth Edition.* Microsoft Press, A Division of Microsoft Corporation, 2008.

[198] O. Rodrigues. *Des lois géométriques qui régissent les déplacements d'un système solide dans l'espace, et de la variation des coordonnées provenant de ces déplacements considérés indépendamment des causes qui peuvent les produire.* 1840.

[199] A. E. Roy. *Orbital Motion, 4th Edition.* Routledge, 2004.

[200] Michael J. Ryan and Louis S. Wheatcraft. On the use of the terms verification and validation. *INCOSE International Symposium*, 27(1):1277–1290, jul 2017.

[201] Matthew N. O. Sadiku. *Elements of Electromagnetics, 7th edition.* Oxford University Press, 2018.

[202] S. R. Schach. *Object-Oriented Software Engineering.* McGraw-Hill, 2008.

[203] C. Scharlemann, David Birschitzky, Harald Fuchs, Lionel Gury, Stefan Hauth, David Jelem, Franz Kerschbaum, Dominik Kohl, Christof Obertscheider, Roland Ottensamer, Thomas Riel, Bernhard Seifert, Richard Sypniewski, Michael Taraba, Robert Trausmuth, and Thomas Turetschek. *PEGASUS - An Austrian Nanosatellite for QB50.* 2015.

[204] D. J. Scheeres. *Stability in the full Two-Body Problem.* Celestial Mechanics and Dynamical Astronomy, 2002.

[205] R. Schwarz. *Keplerian Orbit Elements to Cartesian State Vectors.* 2017.

[206] Rene Schwarz. *Cartesian State Vectors to Keplerian Orbit Elements.* 2017.

[207] B. Seifert, N. Buldrini, T. Hörbe, and F. Plesescu. *In-Orbit Demonstration of the Indium-FEEP IFM Nano Thruster.* 6th Space Propulsion Conference, 2018.

[208] B. Seifert, A. Reissner, N. Buldrini, D. Krejci, F. Plesescu, T. Hörbe, and C. Scharlemann. *Integrated Electric Propulsion Systems for small Satellites.* Space Propulsion Conference, 2014.

[209] B. Seifert, C. Scharlemann, and A. Reissner. *Integrated Mission Platform for Austrian CubeSat Technology (IMPACT) - Final Technical Report.* FOTEC, AE2015-021-01, 2015.

[210] P. Sestoft. *Numeric performance in C, C# and Java.* IT University of Copenhagen, 2010.

[211] P. Sestoft. *Programming Language Concepts for Software Developers.* IT University of Copenhagen, Denmark, 2010.

[212] J. P. Shen and M. H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors.* Waveland Press, Inc., 2005.

[213] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts, 8th Edition.* John Wiley & Sons, Inc., 2009.

[214] G. Skinner, S. Shah, and B. Shannon. *C Style and Coding Standards.*

[215] I. Sommerville. *Software engineering, 9th Edition.* Addison-Wesley, 2011.

[216] Andrew Sorensen. *Hardware-in-the-loop Simulation for Verification of CubeSat Attitude Determination and Control Subsystems.* Master thesis, Faculty of California Polytechnic State University, San Luis Obispo, 2018.

[217] Matt Sorgenfrei, Terry Stevenson, and Glenn Lightsey. *Performance Characterization of a Cold Gas Propulsion System for a Deep Space CubeSat.* AAS Guidance, Navigation and Control Conference, 2017.

[218] Spectrolab, Inc. *29.5% NeXt Triple Junction (XTJ) Solar Cells.* 2012.

[219] ssddevs. *x3d-viewer version 1.1.7, https://chrome.google.com/webstore/detail/x3d-viewer/nneaojlgmfmngeckfemdbfpgedgfpgdg.* 2021.

[220] A. M. St. Laurent. *Understanding Open Source and Free Software Licensing.* O'Reilly Media, Inc., 2004.

[221] R. L. Staehle. *Solar Sails for CubeSats.* Keck Institute for Space Studies (KISS) Workshop, California Institute of Technology, 2011.

[222] W. Stallings. *Operating Systems - Internals and Design Principles, 7th Edition.* Prentice Hall, 2012.

[223] E. Myles Standish and James G. Williams. *Orbital Ephemerides of the Sun, Moon, and Planets.*

[224] G. L. Stephens, D. O'Brien, P. J. Webster, P. Pilewski, S. Kato, and J. Li. *The albedo of Earth.* Reviwes of Geophysics, American Geophysical Union, 2015.

[225] J. Stevens. *CubeSAT ADCS Validation and Testing Apparatus.* Honors Theses 2777, Western Michigan University, 2016.

[226] D. J. Stevenson. *Planetary magnetic fields.* Division of Geological and Planetary Science, California Institute of Technology, Pasadena, 1983.

[227] W. H. Steyn and Y. Hashida. *An Attitude Control System for a Low-Cost Earth Observation Satellite with Orbit Maintenance Capability.* 13th AIAA/USU Conference on Small Satellites, 1999.

[228] STMicroelectronics. *Datasheet: L3G4200D, MEMS motion sensor: ultra-stable three-axis digital output gyroscope.* 2010.

[229] STMicroelectronics. *RM0090 Reference manual: STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced Arm®-based 32-bit MCUs, Rev. 18.* 2019.

[230] B. Stroustrup. *The C++ Programming Language, 4th Edition.* Addison-Wesley, 2013.

[231] Runhan Sun, Camilo Riano-Rios, Riccardo Bevilacqua, Norman G. Fitz-Coy, and Warren E. Dixon. CubeSat adaptive attitude control with uncertain drag coefficient and atmospheric density. *Journal of Guidance, Control, and Dynamics*, 44(2):379–388, feb 2021.

[232] D. Svoboda. *Beyond errno - Error Handling in C.* Software Engineering Institute, Carnegie Mellon University, 2016.

[233] K. R. Symon. *Mechanics, 2nd edition.* Addison-Wesley Publishing Company, Inc., 1960.

[234] Wittawat Tapsawat, Teerawat Sangpet, and Suwat Kuntanapreeda. Development of a hardware-in-loop attitude control simulator for a cubesat satellite. *IOP Conference Series: Materials Science and Engineering*, 297:012010, jan 2018.

[235] Texas Instruments. *Datasheet: LM95071, SPI/MICROWIRE 13-Bit Plus Sign Temperature Sensor.* 2019.

[236] The MathWorks. *Simulink - Simulation and Model-Based Design, Using Simulink Version 6.* 2004.

[237] S. Theil, P. Appel, and A. Schleicher. *Low Cost, Good Accuracy - Attitude Determination Using Magnetometer and Simple Sun Sensor.* 17th AIAA/USU Conference on Small Satellites, 2003.

[238] N. Theoret. *Attitude Determination Control Testing System (Helmholtz Cageand Air Bearing).* Honors Theses. 2783, Western Michigan University, 2016.

[239] W. T. Thomson. *Spin Stabilisation of Attitude against Gravity Torque.* Journal of Astronautical Science, No. 9, p. 31-33, 1962.

[240] K. E. Trenberth, J. T. Fasullo, and J. Kiehl. *Earth's Global Energy Budget.* American Meteorological Society, 2009.

[241] A. A. Trusov. *Overview of MEMS Gyroscopes: History, Principles of Operations, Types of Measurements.* 2011.

[242] A. R. Tummala and A. Dutta. *An Overview of Cube-Satellite Propulsion Technologies and Trends.* MDPI Aerospace, 2017.

[243] J. Urban. *Interfacing C++ libraries to Matlab.* Masarykova Univerzita, 2012.

[244] VICTREX®. *PEEK - High-Performance Polymers Designed for the Most Demanding Automotive Applications.* 2015.

[245] E. M. Vidal. *Development of models for Attitude Determination and Control System components for CubeSat applications.* Master thesis, LuleåUniversity of Technology, 2017.

[246] Vishay Semiconductors. *Datasheet: TEMD6200FX01, Ambient Light Sensor.* 2014.

[247] N. Voudoukis and S. Oikonomidis. *Inverse Square Law for Light and Radiation: A Unifying Educational Approach.* EJERS, European Journal of Engineering Research and Science, Vol. 2, No. 11, 2017.

[248] R. Walker. *Analysis and Interpretation of Astronomical Spectra - Theoretical Background and Practical Applications for Amateur Astronomers.* 2012.

[249] C. Warwick. *In a Nutshell: How SPICE Works.* EMC Society, 2009.

[250] Steve Wassom, Quinn Young, Lynn Chidester, Rees Fullmer, Mitch Whiteley, Robert Burt, Mike Watson, Bryan Bingham, and Keegan Ryan. *Integrated CubeSat Test Facility for Precision Pointing and Power Generation.* Space Dynamics Laboratory, Utah State University Research Foundation, 7th Annual CubeSat Developers' Workshop, 2010.

[251] Kevin Wedeward and Aly El-Osery. *Location and Navigation: Navigation Mathematics, Coordinate Frames.* 2016.

[252] G. Wentworth and D. E. Smith. *Plane and spherical Trigonometry.* Ginn and Company, 1914.

[253] J. R. Wertz. *Spacecraft Attitude Determination and Control.* Kluwer Academic Publishers, 1978.

[254] J. H. Wessels. *Infrared Horizon Sensor for CubeSat Implementation.* Master Thesis, Faculty of Engineering, Stellenbosch University, 2018.

[255] WHATWG and W3C. *HTML - Living Standard.* 2021.

[256] K. Wilhelm and B. N. Dwivedi. *On the radial acceleration of disk galaxies (pre-print).* 2020.

[257] D. R. Williams. *Sun Fact Sheet.* NASA Goddard Space Flight Center, 2018.

[258] WS Hampshire Inc. *TEFLON® - Typical Properties of PTFE.*

[259] J.-P. Zahn, C. Ranc, and P. Morel. *On the shape of rapidly rotating stars.* Astronomy and Astrophysics, EDP Sciences, 2010.

[260] H. Zhao. *Development of a low-cost multi-camera star tracker for small satellites.* Graduate College, University of Illinois, 2020.

[261] P. Zingerle, R. Pail, T. Gruber, and X. Oikonomidou. The combined global gravity field model XGM2019e. 94(7), jul 2020.