HELSINKI UNIVERSITY OF TECHNOLOGY
Faculty of Electronics, Communications and Automation
Department of Automation and Systems Technology

Paavo Heiskanen

Development of a dynamic simulator of a mobile robot for astronaut assistance

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Technology.

Helsinki, 08/11/2008

Supervisors:

Professor Aarne Halme
Helsinki University of Technology

Professor Kalevi Hyyppä
Luleå University of Technology

Instructor:

Seppo Heikkilä M.Sc.(Tech.)
Helsinki University of Technology

Preface

The research for this Master's Thesis was carried out at the Helsinki University of Technology Automation Technology Laboratory during the spring of 2008. It will also be presented in ASTRA 2008, the 10th ESA Workshop on Advanced Space Technologies for Robotics and Automation.

First and foremost I would like to express my gratitude to the European Space Agency (ESA) for supplying me with a grant without which my work would have been impossible. I also thank the personnel of Helsinki University of Technology, Faculty of Electronics, Communications and Automation and especially the Automation Technology laboratory for their continuous endeavor to educate and help me.

I would not have completed this thesis without help from people contributing to open source mailing lists and newsgroups, or the helpful souls at several IRC-channels, who voluntarily gave their time to help me, with no benefit to themselves. Your contribution was invaluable. Another great source of assistance has been my collagues, especially Mr. Jan Hakenberg, I thank you. Also my gratitude goes to Michael Bailey for proofreading this thesis.

Finally, I would like to thank my friends and family, who have always encouraged and urged me to do my best. I hope to stand up to your expectations. And finally, exceptionally warm thanks go to my fiancée Outi Lahtela who had to suffer living with me when I was writing this thesis.

Talk is cheap. Show me the code.

Linus Torvalds

Helsinki, August 11, 2008

Paavo Heiskanen

Teknillinen Korkeakoulu Diplomityön tiivistelmä

Tekijä:	Paavo Heiskanen
Työn aihe:	Dynaamisen kenttärobottisimulaattorin kehittäminen astronauttien avustamiseksi
Päivämäärä:	11. elokuuta 2008 **Sivumäärä:** 97
Tiedekunta:	Elektroniikan, tietoliikenteen ja automaation tiedekunta
Laitos:	Automaatio- ja systeemitekniikan laitos (AS)
Ohjelma:	Master's Degree Programme in Space Science and Technology
Professuuri:	Automaatiotekniikka (Aut-84)
Työn valvojat:	Professori Aarne Halme (TKK) Professori Kalevi Hyyppä (LTU)
Työn ohjaaja:	Seppo Heikkilä (TKK)

Robottien käytöstä on havaittu saatavan suurta hyötyä avaruustutkimuksessa. Avaruus on erittäin vaarallinen toimintaympäristö ihmisille, muttei niinkään roboteille. Näin ollen robotit voivat olla suureksi avuksi yksinkertaisissa tehtävissä kuten tiedustelussa, tavaroiden siirtämisessä ja tieteellisissä mittauksissa. Tämän työn tutkimuskohteena on kentaurirobotti WorkPartner, jonka avulla on mahdollista suorittaa useita näistä tehtävistä. WorkPartner on kenttärobotti joka on suunniteltu interaktiiviseen yhteistyöhön ihmisten kanssa ja jota käytetään Euroopan avaruusjärjestön Network Partnering -ohjelmassa astronautti-robotti yhteistyön tutkimiseen.

Lopputyö esittelee “SimPartner”-ohjelmiston, dynaamisen robottisimulaattorin WorkPartner-robotille käyttäen ODE (Open Dynamics Engine) -fysiikkakirjastoa. Ohjelmisto sisältää tarkan mallin robotista, käsittäen osien mitat, painot, nivelet, sensorit ja toimilaitteet. Kaikki sensorit ja toimilaitteet tarjotaan käytettäväksi asiakas-palvelin-rajapintoina.

Realistinen dynaaminen robottimalli on monella tapaa hyödyllinen astronautti-robotti -yhteistyön kehittämisessä. Robottimallia voidaan käyttää kontrollikoodin kehittämiseen ja robotin käytöksen ennustamiseen esimerkiksi teleoperointitehtävissä. Mallia voidaan myös käyttää voimien ja vääntömomenttien ennustamiseen tilanteissa, joissa mittausten tekeminen varsinaisesta robotista olisi vaikeaa tai mahdotonta. Lisäksi mallia voidaan käyttää turvallisesti odottamattomissa tilanteissa ja tehtävien opetuksessa silloin, kun varsinainen robotti on muissa tehtävissä.

Työssä osoitettaan että varmennettavissa olevan tosiaikaisen dynaamisen robottisimulaattorin luominen on mahdollista vertaamalla simulaatiosta saatuja tuloksia mittauksiin jotka on tehty suoraan robotista. Vertailun mahdollistavat TKK:lla aiemmin tehdyt tutkimukset, joissa robottialustaa on testattu esimerkiksi pyöräkävelyn aikana.

Mallin suorituskyky varmistetaan vertaamalla saatuja tuloksia matemaattisiin malleihin. Lisäksi mallin puutteet ja epäideaalisuudet arvioidaan luotettavan lopputuloksen varmistamiseksi. Suorituskyvyn analysoinnin lisäksi simulaattori esitellään muiden jo olemassa olevien avaruusrobottisimulaattorien kontekstissa.

Avainsanat: robottisimulaattori, simulointi, WorkPartner, modulaarinen simulaattori, mallinnus

Helsinki University of Technology

Abstract of the Master's Thesis

Author:	Paavo Heiskanen
Title of the thesis: assistance	Development of a dynamic simulator of a mobile robot for astronaut
Date:	August 11, 2008 **Number of pages:** 97
Faculty:	Faculty of Electronics, Communications and Automation
Department:	Automation and System Technology
Program:	Master's Degree Programme in Space Science and Technology
Professorship:	Automation Technology (Aut-84)
Supervisors:	Professor Aarne Halme (TKK) Professor Kalevi Hyyppä (LTU)
Instructor:	Seppo Heikkilä (TKK)

The need for robotic assistance has been identified to be essential in space exploration missions. The hazardous space exploration environment is extremely difficult for humans but manageable for robots. Thus robots can be a valuable aid even in simple tasks such as scouting, moving objects, and performing measurements. This work is targeted to a centauroid robot, called WorkPartner, which can perform many of these required tasks. The WorkPartner is a mobile service robot, which is designed to work interactively with humans and is currently used within the ESA Network Partnering programme to research astronaut-robot cooperation.

This thesis describes "SimPartner", a dynamic robot simulator of the WorkPartner robot created using ODE (Open Dynamics Engine) software. The software incorporates an accurate model of the robot, including part lengths, masses, joints, actuators and sensors. All the model's sensors and actuators are provided by using a client/server architecture.

There are several reasons why a realistic dynamic robot model is useful for robotic astronaut assistance development. The robot model can be used to develop the robot's control code and to predict its behavior e.g. in tele-operated tasks. The model can also be used to estimate forces and torques that would be difficult to measure from the actual robot. In addition, the model could be safely used to define and test tasks for handling unexpected events and to enable off-line robot task teaching.

The scientific contribution of this thesis is to demonstrate that it is possible to create a verifiable real-time dynamic mobile robot simulator for a centaur-like mobile service robot. This is achieved by comparing the simulation with the measurements from the actual WorkPartner robot. This can be done for example by comparing the joint torques during wheel walking, which has already been studied at TKK.

In addition to the analysis of simulation performance, the simulator is presented and discussed in the context of previous space robot simulators. Furthermore, the scientific validity of the approach is demonstrated by verifying the mathematical concepts behind the model, and also calculating and verifying the performance levels and limitations of the model.

Keywords: robot simulator, simulation, WorkPartner, modular simulator, modeling

Contents

Index of Figures

Index of Illustrations

Index of Code Examples

Index of Tables

Symbols, Abbreviations and Glossary

AABB	Axis-Aligned Bounding Box
ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
com	center of mass
DARTS	Dynamics Algorithms for Real-Time Simulation
DSHELL	DARTS Shell
DTD	Document Type Definition (defines XML document structure)
EADS	European Aeronautic Defence and Space Company
FPS	Frames Per Second
GIM	Generic Intelligent Machines – a research group at TKK[1]
Hardware-in-the-loop	Actual hardware is wired to the simulator to study the effects of simulated sensor input.
HMI	Human-Machine Interface
LCP	Linear Complementarity Problem
NTP	Network Time Protocol
ODE	Open Dynamics Engine
Operator-in-the-loop	Simulation in real time permitting human interface with the simulation.
OSI	Open Systems Interconnection
P/S/G	Player/Stage/Gazebo (an open source simulator project)
RCET	Rover Chassis Evaluation Tools
RFC	Request For Comments - “standards” that define the internet
RMPET	Rover Mobility Performance Evaluation Tool
ROAMS	Rover Analysis, Modeling and Simulation
RPET	Rover Performance Evaluation Tool
SDL	Simple DirectMedia Layer
SLAM	Simultaneous Localization And Mapping
SOA	Spatial Operator Algebra
Tcl	Tool command language – a programming language
TCP/IP	Transmission Control Protocol / Internet Protocol
UML	Unified Modeling Language
XML	eXtensible Markup Language

1 See http://gim.tkk.fi/ for more information.

1 Introduction

The purpose of this thesis was to create and validate a dynamic robot simulator to be used in Seppo Heikkilä's PhD study on *Astronaut and robot cooperation for natural and seamless task* execution. The WorkPartner robot is a novel construction and development time using the robot is severely limited. Furthermore, since part of the study is being conducted in the ESA premises in Noordwijk, the Netherlands, it is difficult to perform testing with the actual robot.

1.1 Thesis objectives

With the above mentioned constraints in mind it was decided to design and program a robot simulator that could be used to develop software for the actual robot. The major design objectives were:

- Real-time dynamics modeling software with the SpacePartner body and torso capable of interacting with other virtual objects (forces, locations and velocities of different parts extractable).
- To develop a robust XML-based language to describe the robot and its environment (including masses, dimensions, and joints).
- 3D visualization of simulation for analysis, debugging and control purposes.
- MySQL-based modular software architecture, i.e. separation of different functionalities (physics, visualization, communication, control).
- Comparison of simulation data with results acquired from the actual robot.
- The developed software should run on Linux (Ubuntu/Debian).

The minor/optional design objectives were:

- GIM interface to the model.
- Editor for the XML-based world and robot model (use of an existing editor could be the best option).
- The developed software should run on Mac and Windows.

A crude division of work was derived from these objectives:

- State-of-the-art study, literature review – 2 weeks
- Framework functionality – 5 weeks
- SpacePartner modeling – 5 weeks
- Additional features – 5 weeks
- Testing and evaluation – 5 weeks
- Thesis finalization – 3 weeks

1.2 History

The history of computer science is closely related to the history of dynamic simulation, as described by (Kovo 1999, p.5-6). The first computers were used to solve partial differential equations related to the development of the atomic bomb. Military, aeronautic and space industries have been using computer simulation from as early as 1950s. During the fifties and sixties analog computers were used extensively. These machines were able to solve differential equations very fast and the first hardware-in-the-loop solutions were in fact reached using them. The digital revolution of the 1970s replaced the analog computers with digital ones. During this shift hybrid computers containing both analog and digital components were also used. Nowadays simulation is used everywhere in our society, from weather forecasts to ensuring that traffic lights are timed efficiently. Wind tunnel testing is very expensive since the pieces to be tested must be fabricated and tested. Computer simulation of the effect of drag forces on cars and aeroplanes offers great savings for vehicle manufacturers. With the development of 3D-graphics, simulators can also be used as training simulators for airline pilots and other personnel. The whole video game industry can also be seen as a branch of computer simulation.

The computer technology used in computer simulation has traditionally been of the highest standard and even now a major portion of supercomputer time is devoted to calculating weather forecasts. However, the rapid development of PC technology has made it possible to simulate dynamic systems on desktop computers. Dynamic robot simulators are now available for commercial, off-the-shelf hardware.

The fundamental trade-off in dynamic real-time simulation is between real-time performance and simulation accuracy. At a very low level, accuracy is defined by the number of bits the program uses to represent floating-point numbers. This is a feature of digital computers that sets an ultimate limit on how accurate the system can be. In practice, the limit is much poorer. Processing power and memory define the length of the possible time step that the computer is able to calculate within the given real-time constraint.

Distributed computing and parallel processing, combined with the low cost of memory and storage media offer some relief to these problems but it has to be understood that there is a limit to the accuracy a real-time computer based simulation software can achieve.

1.3 Core concepts

This section is an overview of different simulators used in modeling and testing of terrestrial mobile robots and planetary exploration rovers. The goal of this section is to identify the best features of different simulators and to validate design choices for creating a good quality mobile robot simulator.

1.3.1 Mobile robots

A mobile robot is an automatic machine that is capable of moving in its environment and is usually able to interact with it. Mobile robots are often characterized by their means of locomotion, namely legged, wheeled or tracked. Robots are often used in tasks that are too repetitive, monotonous and/or dangerous for human beings. An example of a mobile robot is shown in illustration 1[1].

Illustration 1: WorkPartner - a mobile service robot (artist's impression).

1.3.2 Planetary rovers

A planetary rover is a mobile robot located in an extraterrestrial environment, exploring its surroundings. Rovers are very practical in planetary exploration and have been used since the Russian Lunokhod 1 landed on the moon in 1970. The extraordinary success of NASAs MER-A and MER-B (more commonly known as Spirit and Opportunity) solidified the role of autonomous rovers in planetary exploration.

According to (Bauer, Leung, & Barfoot 2005), the downside of the use of autonomously moving rovers is the increased need to test the stability of mechanical solutions, as well as the sensor and actuator hardware and software, and perhaps

1 Picture source http://automation.tkk.fi/WorkPartner

most importantly the onboard computer system. This increases the need for quality simulation software. Simulations also make multiple iterations cheaper in the early design phases as prototypes do not have to be built. Simulation also makes it possible to study the effects of parametric changes on design details. Finally, it is very difficult to reproduce extraterrestrial conditions, such as martian gravity, without using simulations.

1.3.3 Physics engines

The core of the simulator software is the physics engine. Physics engines can be categorized using many different metrics, the most relevant here being accuracy and required computing power. These two often form a trade-off, increased accuracy deteriorates real-time performance and vice versa. As both are crucial for a mobile robot simulator, it is necessary to determine when the accuracy is *good enough* for the simulation task at hand. When the accuracy is determined it is possible to check whether the frame rate is sufficient for real-time operator or hardware-in-the-loop performance.

(Erleben 2004, p.10) wrote that the functionality of a physics simulator can be crudely divided in two main parts, physics simulation and collision detection. The physics simulation component calculates the motion of the objects in the systems based on their current state (position, velocity, acceleration, forces, torques and impulses). This requires integration, which also creates inherent error in the simulator. Collision detection is a geometrical problem of intersecting objects and it is computationally intensive. When the positions of all the objects have been calculated by the simulation component, the collision detection component determines collisions, or technically, points of intersection between objects. After this the collisions have to be resolved and the resulting forces calculated. This is a very complex problem, and general solutions do not exist. Collision detection problems cause unidealities and instabilities in dynamic simulations. Often a multitude of simplifications must be made to make the collision detection system work.

Physics engines can be grouped by their simulator paradigms. Some of the well known include:

- Constraint-based methods (Erleben 2004, p.20)
  Very complex paradigms that do not allow penetration and are typically very good at handling complex configurations with static contacts.
- Penalty methods (Erleben 2004, p.20)
  Simpler than constraint-based methods, can easily be extended to handle soft bodies. Allow penetration of objects.
- Impulse-based methods (Erleben 2004, p.20)
  Interaction between objects is simulated as collision impulses. Do not allow penetration. Static contacts modeled as a series of micro-collisions.
- Collision synchronization (Optimization-based) (Erleben 2004, p.23)
  Makes large time steps possible by synchronizing collisions at the end of each frame.
- Port-based modeling (Poulakis & Joudrier 2006)
  The system is modeled using bond graphs in which different components and subsystems are connected via bonds that exchange energy. The model has causality and using Kirchoff's laws the total energy transfer can be calculated.

The existence of these different paradigms itself reflects the complexity and computational costliness of physics simulation. The field has been researched from the 1960s onwards but only now is it becoming possible to build accurate real-time physics engines.

Appendix 1 lists four commonly used physics engines. It can be seen that there is no all-in-one solution but rather developers must select their engine carefully to suit their project. For example, SimMechanics offers seamless Matlab interaction and ease of use as systems can be built with Simulink-style function blocks. The downside is that it completely lacks collision detection. In comparison, ODE only offers a C++ -interface but is more versatile and has collision detection.

1.4 Thesis outline

The outline of this thesis follows the process of the associated software framework development. The second section describes the state-of-the-art research where existing simulators were studied to establish development targets for the software. The third section describes the software framework itself and the fourth the associated testing and validation. The final conclusions are presented in the fifth chapter.

2 Previous Work

This section starts with an overview on mobile robot simulator usage reasons and principles. After this some state-of-the-art simulators are presented. The details of some of the simulators can be found in appendix 1. After this planetary rover simulators are presented.

2.1 Mobile robot simulators

The necessity for mobile robot simulators has been recognized by several different robotics research groups. Simulation is used in some phase of almost every mobile robot research project (P. Aarnio, Koskinen, & Ylönen 2001, p.1). There are several reasons why robot simulators are useful, including:

1) Reduction of development time of the robot control code.
2) Increased quality of the robot control code.
3) Enabling the testing of complex control algorithms in real time using powerful computers to perform tasks that are normally done by simple controllers.
4) Cost savings by avoiding unnecessary damage to actual robot equipment when testing new control strategies or stability solutions.
5) Simulation of complex systems without having to build them.
6) Studying robot behavior in an unattainable environment.

Previously, several research groups have also built whole simulator packages themselves, leading to robotics specialists concentrating on things that are not essential in constructing a robust robot control code, such as ground contact modelling and impact forces as described in (Buehler et al. 1999).

The majority of simulators developed by researchers are created using open source source rationale to promote platform independence (Vaughan, Gerkey, & Howard 2003), distributed software development to loosen the coupling between different modules (Collett, MacDonald, & Gerkey 2005) and making use of other available open source libraries. However, proprietary simulators such as the ADAMS package (Fraczek & Morecki 1999) and Envision (P. Aarnio, Koskinen, & Salmi 2000) are also used.

Use of simulators is nowadays vital when developing mobile robots. There are several reasons for this. One advantage is that when the architecture of the robot is selected, simulators can be used to emulate the sensor information, making it

possible to program the control code when the actual robot does not yet exist. A second advantage is that the robustness of the control system can be tested with several different environments. Furthermore, since testing time with the actual robot is limited, simulators can be used to enhance the parallel development of systems. Using simulators can greatly reduce the cost and effort in building mobile robots. There are several papers that describe this, for example MIT's DARPA Urban Challenge team used two different simulators when creating their competition vehicle, as described in (Leonard et al. 2007).

2.1.1 SimMechanics

SimMechanics is a commercial tool for simulating mechanical systems. It is an extension of Matlabs Simulink software. Its key features are ease of use and integrability to existing Simulink block diagrams. Mechanical systems (linear and nonlinear) can be modeled with SimMechanics blocks that can be connected to Simulink blocks. CAD models can also be directly translated to function blocks by using separate software. SimMechanics also offers Matlabs powerful mathematical tools, such as integration and optimization. Furthermore other Matlab extensions, such as the Real-Time Toolkit can be integrated into the development environment. The drawback of this software for mobile robotics is that there is no built-in collision detection, but the user must take care of this. The system also offers automatic C-code generation.

2.1.2 Vortex

The Vortex simulation toolkit, developed by CMLabs Software, is a proprietary development platform that offers physically accurate modeling of ground vehicles, soil, terrain, and other real-world objects. It has a C++ API and integrated 3D-graphics utility. It also supports geometric collision detection. The software has been used for example in developing training simulators for tower cranes, deep sea remotely operated vehicles and explosive ordnance device robots by EADS. It has also received an award by the Military Training Technology magazine. Vortex is targeted to the market segment in which real-time performance is more important than high-fidelity physics modeling. Robot systems are developed using fundamental building blocks, such as cuboids and spheres. Sensors and Actuators are then incorporated into the system, making it possible to interact with the environment and receive data from it.

2.1.3 The P/S/G simulator project

The Player/Stage simulator has been cited as the *de-facto* standard in the open source robotics community. Its design goals are platform independence, enhanced scalability, development process simplification, real-time performance, integration with existing infrastructure, promotion of software reuse, programming language independence and transport independence (Collett, MacDonald, & Gerkey 2005).

Player

The original Player simulator is a network-oriented architecture that abstracted physical robot properties by using:

- Character device model

  Popularized by *nix[1] systems, abstracts all I/O devices as data files. Data can be collected from a device by reading and sent to the device by writing a file. In Player, robot sensors can be accessed by reading them, and actuators manipulated by writing to them.
- The interface/driver model

  As the character device model defines only the semantics of the devices but not the data formats, an additional model is needed. This determines the content of the streams and provides device independence, yielding portable code.
- The client/server model

  This abstraction provides a way to implement a robot interface. Player user's control program, the client, is separated by a standardized medium (TCP socket) from the server, which executes low-level device control. This yields language-neutrality as the client can be written using any programming language that can support the communication medium (Vaughan, Gerkey, & Howard 2003).

Stage

The Stage (2D) multiple robot simulator ”simulates a population of mobile robots, sensors and environmental objects”. The Stage was built to experiment with swarms of several robots that would be expensive to purchase and maintain. The multi-robot system simulation is designed to be achieved by:

1 Unix, Linux, Solaris, etc.

- "Good enough" fidelity
  Computationally cheap models of devices rather than ideal emulation.
- Linear scaling with population
  Sensor models use algorithms that are independent of population size.
- Configurable, composable device models
- Various sensor and actuator models are provided and they are sufficiently flexible.
- Player interface
  All sensor and actuator models are available through Player's interfaces.

Although there is no guarantee that robot behavior in Stage is comparable with that of real robots, users have found that software developed with Stage works with "little or no modification with the real robots and vice versa". (Gerkey, Vaughan, & Howard 2003)

Gazebo

The Gazebo software brings 3D-capability to the simulator package. Whereas 2D is generally sufficient in simulations that include indoor robots, a three dimensional simulator is needed when outdoor mobile robots are concerned. Gazebo is designed to accurately reproduce the dynamic environments which a robot may encounter. Simulated objects have a mass, velocity and numerous other attributes that contribute to realistic interaction with the environment. Therefore these objects can be pushed, pulled, carried, etc. The architecture of Gazebo is based on the idea that it should be easy to create new robots, actuators, sensors, and other objects. The general structure of Gazebo components is shown in illustration 2. It incorporates two external libraries, namely ODE for rigid body dynamics and collision modeling, and OpenGL/GLUT for visualization. Simulator development is simplified by the use of these external libraries and internal abstraction also makes it possible to replace them if better alternatives become available. Gazebo interfaces with Player in the same way that Stage does. The Player device server treats Gazebo in the same way as all devices capable of sending and receiving data (Koenig & Howard 2004)

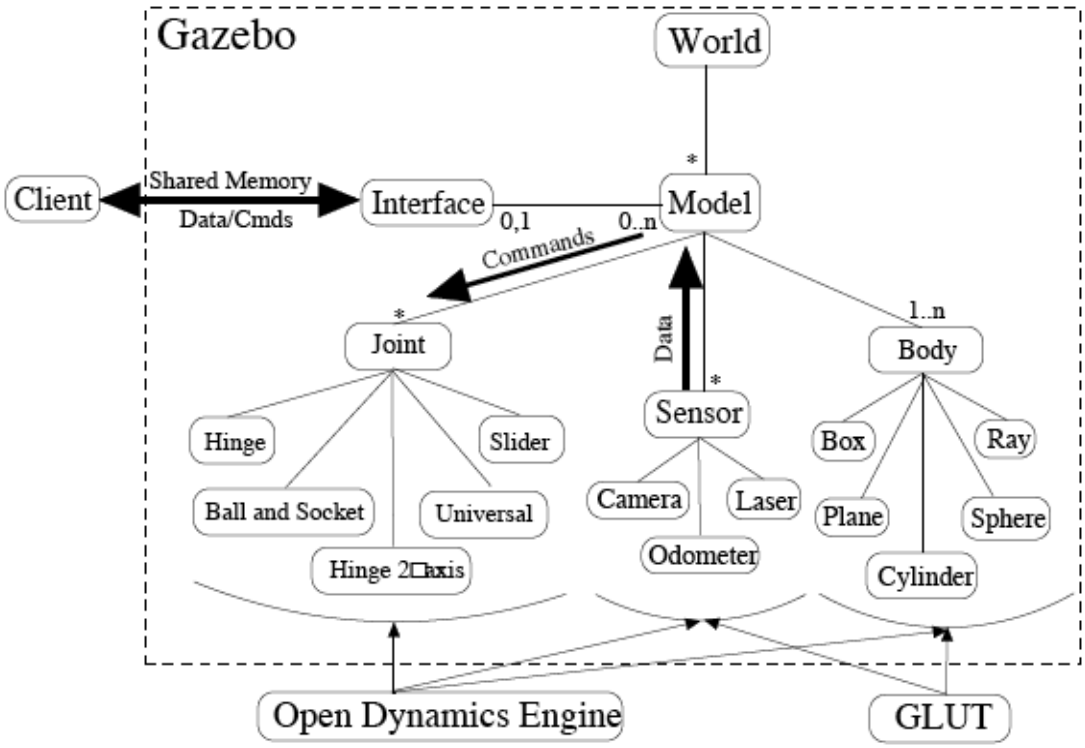

Illustration 2: Gazebo components.

Gazebo also has a number of significant limitations that make it unsuitable for some tasks. The most important limitations are the lack of soil/surface modeling, non-deformable objects and missing fluid/thermal dynamics. As a technical aspect, distributed computing is not available in Gazebo due to its monolithic physics engine. (Koenig & Howard 2004)

It should also be noted that the software is not yet in version 1.0 maturity. The current version number is 0.10, released 1 July 2008.

2.1.4 WebOts

WebOts is a commercial mobile robot simulation software designed to be used at robotics research and teaching institutes. WebOts uses ODE as a physics engine and its main features include:

- Sensor and actuator libraries
- Ready-made models for several robots
- C, C++ and Java interfaces, plus a TCP/IP interface
- Simulation of multi-agent systems with local and global communication systems. (Michel 2004).

WebOts is primarily meant for developing cross-platform, easily implementable robot source code for ready-made and custom robots.

2.1.5 Digital Spaces

Digital Spaces is an open source immersive multimedia presentation and simulation engine programmed in C++. It uses ODE as a physics engine and OGRE (Object-Oriented Graphics Rendering Engine) for visualization. It is designed to be used in the Microsoft Windows operating system and is currently at version 0.10 maturity.

2.2 Planetary rover simulators

This section presents an overview of different planetary rover simulators used today. It starts with a description of the rationale of usage of these simulators and then presents the simulators in detail.

The necessity of advanced simulation software in space applications is obvious. Currently simulators are used in all fields of engineering, and space engineering is no exception.

In the conceptual study and preliminary analysis phase, simulators can be used extensively to test different designs without the inherent cost of building prototypes. By building a spacecraft model and testing it in the simulator, all different design possibilities can easily be tested. Spacecraft response to commands can also be tested by using operator-in-the-loop simulations.

In the design, development and testing phase simulators serve in different roles. When hardware and software choices are made the simulated components can be left out of the model and the actual component response can be tested using hardware-in-the-loop simulations. When the spacecraft concept is ready, the onboard software functionality can be tested with just the sensors connected to the simulator. The response of the spacecraft to artificial stimuli can thus easily be tested.

In the operations phase the commands that are about to be transmitted to the satellite can be transmitted to a simulator first, confirming the correctness of the commands and verifying that the desired reaction is produced in the spacecraft. This reduces the chance of a mission loss due to, for example, a typing error in the spacecraft control commands.

If, however, something goes wrong and the mission is lost, simulators provide a good way to analyze the reasons even when very little data is available from the actual mission. A good example of this is the analysis of the failure of the Mars Global

Surveyor satellite. The last communication with the satellite occurred in November 2006, after which the mission was deemed lost. The decisive event was two high-gain antenna direction commands “commanded with slightly different (operator input) precision”. This led to a catastrophic chain of events that eventually caused the mission to fail. (NASA 2007)

By simulating the commands the engineers were able to determine the events very accurately, yielding several findings in operational procedures and processes, spacecraft design weaknesses and lifetime management considerations that can now be taken into account when designing future missions.

2.2.1 ROAMS

ROAMS is a real-time physics-based simulator for planetary surface exploration rovers. It is designed to provide a virtual testing ground for rover navigation, mechanical, electrical, sensor, power and control subsystems. It can be used for both operator-in-line and off-line subsystems since it is based on the DARTS/DSHELL framework. Rover subsystems and the base model have been developed using the Rocky-7 Mars rover prototype. For example, a novel kinematics solution has been developed using constrained optimization to be used in driving on Mars-like terrains. Some pre-existing model libraries, such as the solar panels and batteries, could be reused from the DARTS/DSHELL framework. The simulator has also been used to simulate a planning system for a rover swarm, three rovers working together to perform complex tasks. (Yen, A. Jain, & Balaram 1999)

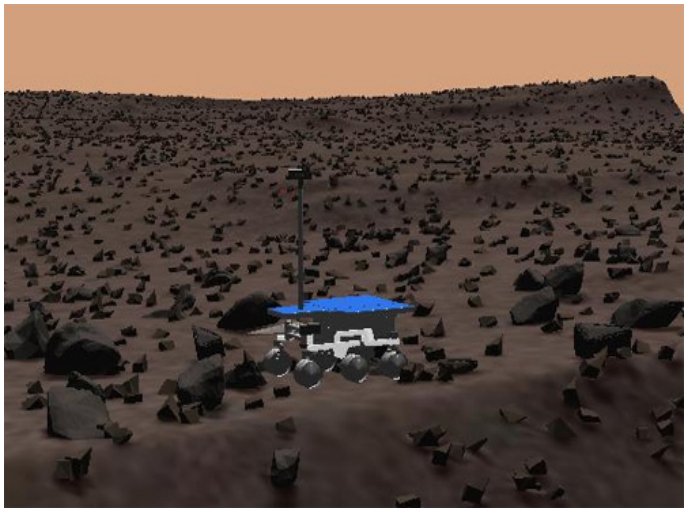

Illustration 3: ROAMS screen shot.

To test the rover's on-board software it is set to run in a Unix real-time system. ROAMS creates the simulated sensor input to the software, which in turn computes a

sequence of way-points using its navigation and collision avoidance systems. The resulting wheel-motor and steering motor commands are then passed to ROAMS which approximates the next position of the rover and solves the inverse kinematics for the configuration of the rover. Then the next set of sensor inputs is passed to the rover software. With this method the stability and robustness of the software can be qualitatively measured. (Yen, A. Jain, & Balaram 1999)

ROAMS is generally used to model 6-wheeled rovers using rocker-bogey suspension with variable numbers of steerable wheels. Whereas the DARTS framework provides the kinematic solution for general multi-body topologies, several specific models are needed by ROAMS. These include wheel sliding, slipping and sinkage as well as terrain feature (smooth, piecewise-smooth, nonlinear) approximations. Once the contact force has been determined, DARTS/DSHELL can be used to determine the rover state. (A. Jain et al. 2003)

Currently, ROAMS is being developed under the following design goals (A. Jain et al. 2004):

- Validated physics based models
  High fidelity to support closed-loop testing. This requires the development of good quality mechanical, actuator, sensor and environmental models.
- Model configurability
  Rover configuration can evolve considerably during the design phase. The simulator has to be versatile enough to allow users to configure model data files easily.
- Closed-loop simulations
  The simulator has to be embeddable to an environment consisting of a mixture of on-board software, real and simulated hardware. This also raises performance considerations as the simulation must be able to run in real-time.
- Layered toolkit approach
  The simulator should provide a good level of instrumentation and features in order to be useful. A layered design in which several modules are provided as plug-ins is adopted to avoid the code size and external dependency explosion. This also promotes code reuse.
- Spacecraft simulation framework
  The simulator is built upon an existing DARTS/DSHELL framework. This enables the developers to focus on rover-specific extensions and make their

contributions usable by all projects sharing the same infrastructure. A case example of this is the DSENDS entry simulator which is based on the above-mentioned framework and also uses ROAMS dynamic simulation and terrain modeling libraries.

- Open source tools

  The simulator development emphasizes the use of open source software whenever possible. This has led to inclusion of several libraries and tools into the simulator.

- Usable

  Simulator is developed with the user in mind, providing several user interfaces and reducing the learning curve.

ROAMS can be run in several different modes, including stand-alone mode that provides a Tcl command line interface for user-simulation interaction, and a C++ interface to allow rover software to interact with ROAMS. The whole simulator can also be run as a Matlab S-function block so that it can be integrated into larger simulations. The ground contact computations are made by using the SWIFT++ library. To reduce computation time, only a small patch of ground underneath the rover is used for calculating the possible contact points. As the rover moves, these patches are created and destroyed. ROAMS can also be used to estimate kinematic parameter dependencies using the Monte Carlo method. This capability is inherited from the DARTS/DSHELL framework. (A. Jain et al. 2003)

2.2.2 RCAST

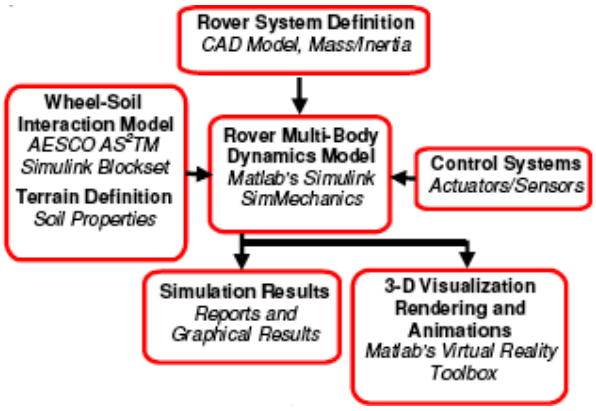

Illustration 4: RCAST architecture.

RCAST is a rover chassis and analysis computer simulation which couples a rigid multi-body dynamics engine (Matlab's SimMechanics) together with AESCO's Soft Soil Tire Model (AS^{2}TM) terramechanics module to study locomotion performance.

It has been developed for phase A studies of the ExoMars rover. The rover model is defined using CAD models from SolidWorks or ProEng. This simulator has been developed to support rover chassis design and optimization, and the model has been verified using various scenarios including slope climbing. Since wheel walking is a novel concept in planetary rovers, it has to be simulated in an early phase of chassis development. Simulation results showed that "wheel walking can enable the rover to climb slopes which are significantly steeper than that achieved by actuating all wheel motors and attempting to drive straight up a slope". (Bauer, Leung, & Barfoot 2005)

It was also necessary to validate the simulation results using wheel-soil interaction experiments. A testbed was constructed and the results were compared with a single wheel simulator. The test setup was used to measure interaction forces and torques as a function of the slip ratio. These measurements confirmed the validity of the simulation and also yielded soil parameters that could be used to tune the simulation of the rover chassis. To fully validate the simulation, a testbed for the full rover chassis will be required. (Bauer, Leung, & Barfoot 2005)

2.2.3 RCET

RCET is a set of tools to support design, selection and optimization of exploration rovers in Europe. The goal is that RCET will enable accurate predictions and characterizations of rover performance as related to the locomotion subsystem. RCET is designed to be a database-driven application to simulate rovers, augmented with two hardware testbeds. RCET will also incorporate a set of parametric tools to allow design and simulation of rovers in a short time. Parametric tools consist mainly of a 2D-simulator to help deciding in the first-order trade-offs. After this the 3D-simulator will be used for validating the rover concept. The testbeds are similar to those used in RCAST, one for single wheel testing and one for verifying simulations using a complete chassis prototype. (Michaud et al. 2004) As a matter of fact, RCET would have been used for ExoMars simulations had it been ready when the conceptual study began. (Bauer, Leung, & Barfoot 2005)

The central piece of the simulator architecture is a database. It makes report generation easy and allows easy data comparison between the real measurements and simulation. RCET, like RCAST, is a simulator to study motion control of wheeled rovers. It is developed precisely for this purpose to allow fast prototyping at the early design phases and also to make it possible to make quantitative analysis suitable for

concept validation. (Michaud et al. 2004)

Using RCET, researchers were able to compare two chassis models on a key metrics, including drawbar pull as a function of wheel slippage and friction coefficient while climbing over rocks. The metrics were first simulated, then validated using the testbeds. This led to a conclusion that “there is less difference in terms of performances between two different rover chassis than between the same architecture with different internal dimension”. Simulations can yield such valuable information that the design team can then use to justify their trade-offs. (Michaud et al. n.d.)

2.2.4 RPET

This software consists of two main modules, Rover Mobility Performance Evaluation Tool (RMPET) and Mobility Synthesis (MobSyn). It is a simple, user friendly and accurate tool to perform preliminary analysis for the configuration of planetary rovers. (Patel et al. 2004)

RMPET

This tool is used within the RPET to compute the mobility performance parameters and the Mean Free Path (MFP) depending on the type of the locomotion system (wheeled, tracked or legged) and soil (martial, lunar, terrestrial or user defined). The mobility performance parameters include (Patel, A. Ellery, Allouis, Sweeting, & L. Richter 2004):

- Soil Shear Strength, which determines the maximum shear stress the soil can resist.
- Soil Thrust is the maximum tractive effort the soil can provide.
- Soil Slip is the difference between the vehicle's translational velocity and the rotational velocity of the wheel/track.
- Motion Resistances are forces acting in opposition to the soil thrust, caused by soil compaction due to sinkage, bulldozing and gravitational resistance.
- Drawbar Pull is the difference between soil thrust and motion resistance. It is the “most important value in the development of a vehicle as it defines the ability of a vehicle to traverse over a specified terrain. In order for a vehicle to negotiate terrain it must have a positive Drawbar Pull”(ibid).

The MFP defines the expected distance the vehicle can move in a straight line before it encounters an obstacle it cannot negotiate. This can be expressed in units of vehicle

scale, for example turning circle diameter. Then a large MFP can be interpreted as the vehicle's ability to traverse in the terrain. A small MFP means that the terrain is impassable.

MobSyn (Mobillity Synthesis)

MobSyn is used in the RPET simulation software to compute the configuration equations for the chosen locomotion type and to yield the ideal wheel/track width and wheel diameter for the desired performance. These calculations are made on the basis of motion resistances, power/torque availability, terrain, etc. that are inputs to the system.

2.3 Related Frameworks

This section provides a short overview of the different frameworks used in the mobile robot and planetary rover simulators. These frameworks form the core of these simulators, and so their performance characteristics ultimately define the usability of the tools developed on top of them.

2.3.1 DARTS

DARTS is a high fidelity, flexible multi-body dynamics simulator that is used for real-time hardware-in-the-loop design, testing and integration of spacecraft software. DARTS is written in ANSI C and developed by Jet Propulsion Laboratory of NASA. It uses SOA mathematical framework for solving multi-body dynamics. Its flexibility is demonstrated by its extensive use in NASA applications and also by totally unrelated projects, such as the solving the dynamics of large-scale molecular systems in the NEIMO software project (A. Jain n.d.).

DARTS won the NASA software of the Year award in 1997 as a technology enabler, and for saving over 10 million dollars on NASA missions. It has been used on the Cassini, Galileo, Mars Pathfinder, Stardust, New Millennium, and Neptune Orbiter projects (Curto n.d.).

The system takes a text input file that is read at runtime and specifies the bodies that make up the spacecraft and the hinges that connect them. The bodies are connected as a tree topology, with the root of the tree as base and different parts as nodes. Because the model data is not hard-coded it is possible to construct models easily for different missions. Models can also be changed without recompiling the source code (Biesiadecki, A. Jain, & James 1997).

2.3.2 DSHELL (DARTS Shell)

DSHELL is a C++ model library for DARTS. It is portable from desktops to real-time hardware-in-the-loop simulation environments. It includes libraries of several hardware models, for example sensors, motors and encoders. The library includes extensive instrumentation so that the user has high visibility into the simulation, yielding high effectiveness as a design, development and testing tool. Actuator models interface directly with the DARTS simulator, for example by applying forces to model nodes (thrusters), or attaining information from the node (sensors). Motors can be attached to the hinges to move the bodies that are connected by that hinge. DSHELL was also been used as a basis for the Cassini High Speed Simulator (HSS) which will be used to test command sequences prior to uplink (Biesiadecki et al. 1997).

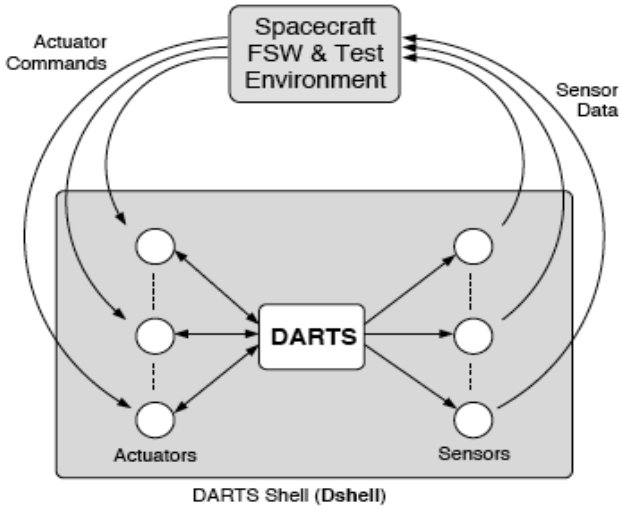

Illustration 5: Data flow in a DSHELL simulation.

2.3.3 ODE

Open Dynamics Engine is a free rigid body dynamics library with collision detection. It is currently used in the majority of commercial and open source robot simulators and games, the developers reporting over 1000 applications. It is quoted to be fast, robust and stable when simulating articulated rigid body dynamics with hard contacts. The ODE collision engine can also be replaced with other options (such as the Bullet) if the user deems this necessary. ODE uses first order integration for speed and stability. This, however, means that ODE is not accurate enough for quantitative engineering. (Smith n.d.)

2.4 Conclusions

Mobile robot or planetary rover simulator structure depends highly on the planned usage of the simulator. Normal mobile robot simulators tend to be more general whereas planetary rover simulators can be programmed for very specific tasks, such as determining optimal wheel diameter or axle length. As with other software, best practices still remain rather uniform regardless of software type. The general structure of a simulator is presented in illustration 6.

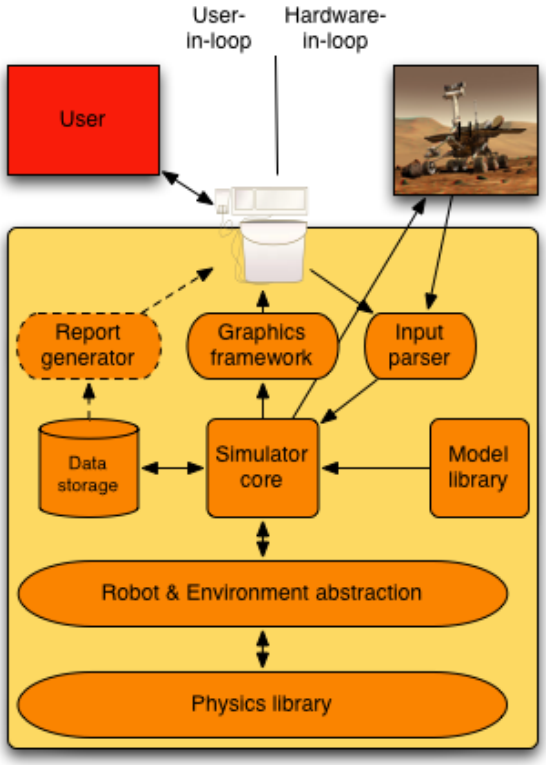

Illustration 6: General mobile robot simulator structure.

Object-orientedness, modularity, scalability, platform independence and open source paradigms tend to create software that is both extensible and flexible.

An issue that is in focus in planetary rover simulator development is the wheel-soil interaction. This is understandable as alien planets are very hostile environments for robots and even a small error can cause a mission to fail. On the other hand, this is a feature that is completely lacking from terrestrial mobile robot simulators. This is probably because earlier mobile robots tended to move inside buildings, and autonomous outdoor robots are still relatively rare. Including the wheel-soil interaction in a terrestrial robot simulator would yield a novel software. A feature that can be found in most of the terrestrial simulators is the possibility to genuinely

interact with the environment. This is not included in the planetary rover simulators as it has not yet been part of mission parameters to move rocks etc.

It is also a feature in the simulators to have a distinct physics/collision library. The benefit of this is that the library can be independently developed and even replaced if a better alternative appears. Furthermore, storing the information in a database makes report generation and simulation validation easier. Models should be input in a standardized, human-readable format without a need for software recompilation. It is also imperative that the software is user friendly and provides information at a level that is detailed enough to support the user's research. This means that individual forces/torques/voltages/etc. should be easily observable.

Thus, it is possible to form a list of features that identifies a good quality mobile robot simulator:

- **On target**
  The software should be developed with the end user in mind, taking into account his needs and wishes.

- **Open Source**
  Open source software brings many great advantages. First of all, the code is verifiable by members of the scientific community and thus gives more credibility to the tool. Secondly, it is possible to use some of the vast amount of open source libraries available.

- **Modular**
  Modular code makes it possible to use the best libraries available and change them if necessary; it also promotes code reuse.

- **Flexible**
  The software modules should be as flexible as possible, as this encourages other developers to contribute to the code and also improves the general quality of the code.

- **Parametrized**

  Good quality software enables the user to make changes in the way the software operates without the need for recompiling the program. This can be achieved by placing as many program parameters to human readable text files as possible.

- **Platform independent**

  A good quality code should be programmed so that it can be run on different platforms.

- **Real-time ready**

  It should be possible to run the simulator in real time with operator-in-loop.

- **Connected to actual hardware**

  It should be possible to connect the simulator to real robot equipment to make hardware-in-the-loop simulations possible.

- **Verifiable**

  The accuracy of the simulator must be stated.

3 SimPartner Framework

This chapter presents a description of the SimPartner framework which is the central element of this thesis. A framework means a re-usable software system that consists of libraries, definition files, and so on, used to solve a complex problem. Its development was driven by the research presented in the previous chapters. The detailed structure of the software framework is illustrated in a UML sketch that can be found in appendix 2.

The first section is an overview of the SimPartner itself, followed by a description of the physics library. The third section introduces the database, followed by the environment definition method. After this, the robot, sensors and actuators are described. The eight section introduces the window manager subsystem, followed by robot modelling and the clients used to control the robot.

3.1 Overview

SimPartner is an object-oriented dynamic robot simulator which combines several existing open source libraries and technologies to create a versatile simulator framework. The open source projects included are:

- Open Dynamics Engine.
- Boost, peer-reviewed, portable C++-libraries that are becoming a part of the future C++ standard. The Boost libraries used include:
  - UBLAS – Basic Linear Algebra.
  - PO – Program Options, enhance command line and text file parameter usage.
  - Graph – Node and vertex graph implementation.
  - Lexical Cast – Lexical casting of characters to numbers and vice versa.
  - Pointer Vector – A convenient way to store objects.
  - Asio – Asynchronous input/output, contains an implementation of tcp/ip communication protocol.
- libxml/libxml++, XML parser.
- MySQL++, C++ wrapper for MySQL 's C API.

- SDL, a multimedia library that provides low level access to audio, keyboard, mouse, joystick and 3D hardware using

  - OpenGL, high performance graphics library.
  - GLU, OpenGL Utility Library.
  - GLUT, OpenGL Utility Toolkit.

- MySQL Database, the world's most popular open source database.

A detailed table of library versions can be found in appendix 3. SimPartner features a modularized design that allows the user to change parts of the code without having to reprogram the whole framework. This modularized structure is general practice in software engineering and can also be seen in the cases shown earlier. The modules in SimPartner are:

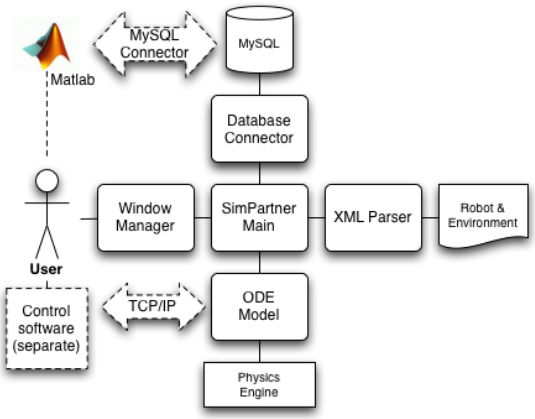

Illustration 7: Modularized structure of SimPartner.

- **SimPartner Main**

  SimPartner main program creates the executable application. Reads in simulation parameters from the properties file. Handles communication between the different modules of the framework.

- **XML Parser**

  Receives the files containing the environment and robot specifications from the main program. Validates the files against the DTD provided. Parses the simulation world specifications (gravity, surface plane) and the environment and robot bodies, joints, etc.

- **ODE Model**

  Encapsulates the ODE in classes. Handles communication between the SimPartner framework and the Open Dynamics Engine. Keeps the internal storage of the bodies pose, velocities, etc. ODE models of sensors and actuators also include an implementation of the TCP/IP communication stack that allows the user to control the robot and monitor it's state through an external software.

- **Window Manager**

  Shows the simulation results to the user using OpenGL graphics library. Receives input from the user through the keyboard and passes the user commands on to the main program.

- **Control software**

  Modeled robots can be controlled with separate control software using a TCP/IP-based client/server architecture, explained in detail in section 3.7.

- **Data analysis**

  The simulation data that accumulates in the database must be analysed somehow. Due to the wide user base of the MySQL database software there exist several ways this can be done. For details see section 3.4.2.

There is a lot of simulation data that needs to be updated for every simulation step. This is done partially by using standard C++ data types such as arrays and vectors. A homogeneous transformation entity is used for storing the position and orientation information of the bodies in the simulation. A homogeneous transformation is a 4x4 matrix that stores the position and orientation of the body.

$$T = \begin{bmatrix} R_{00} & R_{01} & R_{02} & p_x \\ R_{10} & R_{11} & R_{12} & p_y \\ R_{20} & R_{21} & R_{22} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1)$$

Orientation is stored in a 3x3 rotation matrix. This is a real special orthogonal matrix of which the transpose is equal to its inverse and has a determinant of 1. Rotation matrix has many useful properties and is extensively used in control theory.

3.1.1 Parametrization

As shown in the literature review, a high quality simulator software should be parametrized. This means that it should be possible to change the behavior and usage of the simulator without the need for recompiling software. In SimPartner this parametrization is extended to many levels. The core parameters are stored in the properties file which can be defined in the command line when starting the software. Thus the user can have several property files ready and use the one which is most convenient. The default file is *parameters.txt* residing in the application root

```
dbname=simpartnerdb
dbserver=localhost
log=FALSE
logEveryStep=FALSE
environment=template.xml
stepsize=0.001
contactMaxCorrectingVel=0.2
contactSurfaceLayer=0
autoDisableFlag=TRUE
globalERP=0.2     #Common values are 0.1 ... 0.8
```

Code example A: Part of a properties file.

directory. Parameters read from the properties file can also be overridden with command line arguments.[1]

3.2 Open Dynamics Engine details

ODE was already briefly mentioned in section 2.3.3 but for understanding its behavior and limitations a more detailed overview is necessary. The purpose of this section is to provide understanding of the ODE core dynamics modeling principles. The mathematical concepts used in the modeling are presented and the resulting capabilities and constraints are elaborated to provide understanding of ODE's potential to model different applications. This section is based on (Smith n.d.), comments in the engine source code, and the project wiki-page[2].

3.2.1 Bodies and geoms

ODE is a rigid body dynamics simulator. It uses two different concepts for simulating the dynamics and collisions. A body is an object that has certain immutable properties, such as mass and an inertia matrix. The body also stores information on properties that change over time, such as pose and linear and angular velocities. A body is dimensionless and for collision detection purposes we need another concept,

1 See ODE manual for the details of the parameters used in the file.
2 http://opende.sourceforge.net/wiki/index.php/Main_Page

a geometry object, geom. Geoms store spatial properties for various different shapes, such as spheres, cylinders, planes, etc. If a body is connected to a geom it moves dynamically, following newtonian mechanics. Otherwise it is immovable, such as a ground plane in the simulator.

3.2.2 Joints

Joint is an object that attaches two bodies to each other with certain degrees of freedom. The number of joints in ODE is constantly growing; when this thesis was written there were at least fixed, prismatic (slider), hinge, ball-socket, and motor joints. There also exists a special type of joint used for collisions which is presented in a separate paragraph. A joint transfers force and torque between the two bodies, making it possible to create more complex structures. A joint can also have constraint parameters, that further limit the amount the bodies can move with respect to each other.

Motor joints can be linear or angular, they have a special velocity and maximum force parameters that can be used to make controllable parts in the simulation. It is also possible to create motors that move to a specified position because the motors store their pose wrt. the original orientation and position.

3.2.3 The simulation loop

ODE is used with fixed time steps. For every simulation step, the same actions are performed for all the bodies and geoms. The simple overall action sequence is collide, step, destroy. First of all, possible collisions are culled by checking whether Axis-Aligned Bounding Boxes (AABBs) attached to geoms intersect with each other. After this, geoms that actually collide are connected in the contact points that create separate contact joints.

When the collision checking is done, the ODE simulation space is integrated for one step. Bodies in motion move and bodies that are connected with joints transfer momentum to each other. The final step destroys the contact joints created by collisions. After this the new information about bodies (position, velocity, forces, etc.) is passed on and the process starts over.

3.2.4 Collisions

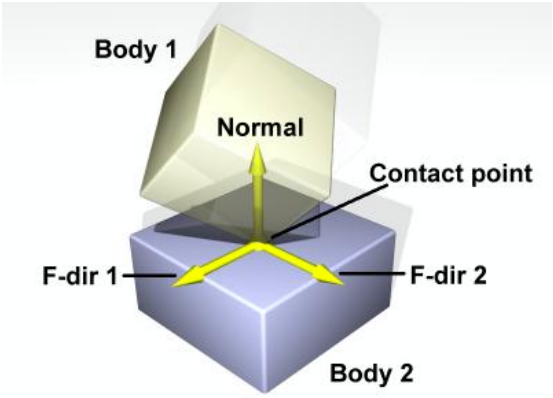

Illustration 8: A collision, picture source (Smith n.d.).

As mentioned above, geoms that intersect generate contact points. When contact joints are generated from these points several parameters can be introduced. For example, maximum penetration, coulomb friction coefficients in two perpendicular directions, bounciness and other parameters that affect the collision can be defined. A collision is shown in illustration 8.

The amount of force which the contact joint delivers to the colliding bodies depends on the amount of interpenetration between the geoms. This can cause simulation instabilities if the parameters are not precisely fixed and the time step set to be in the correct region (approximately 0.02 – 0.001 seconds per step).

3.3 Physics engine wrapper

The ODE engine is currently in version 0.10 maturity. There are several things that are either not implemented or not guaranteed to work stably, such as Cylinder-Cylinder collisions and variable time-step iteration. Some of these deficiencies can be remedied by simple workarounds or design choices. For example, a way of combining the fixed time-step simulation with a variable-time step visualization is shown in illustration 9. When the simulation is done this way two things are achieved. First, ODE behaves deterministically in that the same simulations always yield the same results if no outside input is introduced to the system. Second, lag caused by components not directly related to the simulation (for example slow database connections) will not affect the simulation but rather the user experience.

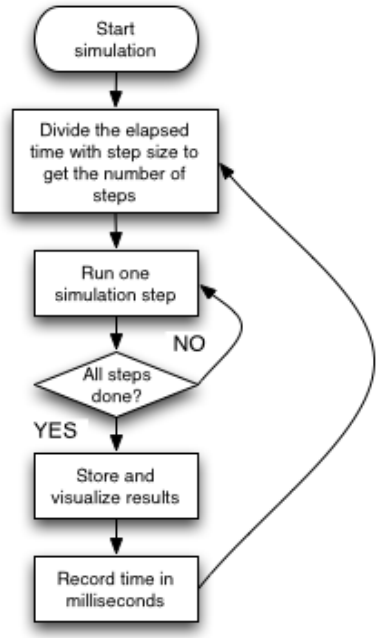

Illustration 9: Fixed time step simulation with variable time visualization.

3.4 Database

All the necessary data obtained from the physics simulation is stored to a MySQL database if the user so desires. The data storage is controlled with a simulator property file entry. Storage can be done either every simulation step or every screen refresh step. There are several reasons which justify using a database instead of a text or a binary file. These reasons include:

1. ACID(Atomicity, Consistency, Isolation, Durability)-properties. These are database properties that guarantee information validity at all times.

   Atomicity means that if all of the tasks of a transaction are not performed, none of them are.

   Consistency means that the database will be in a correct state when transactions begin and end. This will uphold database integrity.

   Isolation means that all transactions are separate from each other. This makes database processes serializable.

   Durability means that all transactions will persist. All information will survive a system failure or a program crash if they are committed to the database.

2. SQL-queries

   SQL provides an easy yet powerful interface to simulation data.

3. Integrability

   MySQL databases can easily be integrated to several existing applications. Simulation data can be accessed with Matlab applications locally (with separate software), over a web interface globally or a direct network connection can be directly made to the database, whichever way is most convenient.

Data can be stored in the database either every simulation step or every visualization step. Storing every simulation step slows the simulation considerably and is only used to validate the physics model in simple cases, such as falling bodies. The structure of the SimPartner database is presented in appendix 4.

3.4.1 Selected Tables

This section describes the database structure and the way the data is stored to the database. The database is the main interface for the user to access the simulation results and so it is necessary to explain clearly the way the data is stored.

Body

The body table stores information on the physical properties of the body in question, such as the physical dimensions and the type of the the body. Furthermore, the location of the center of mass (in body coordinates) and the inertia tensor are stored. The inertia tensor is a 3x3 matrix that stores the moments of inertia of the body with respect to different global frame axes. The inertia tensor is automatically calculated for standard object types by ODE but it can also be redefined if necessary.

Pose

Pose refers to the position and orientation of a body in a given coordinate system. In the SimPartner framework, the pose of all bodies is stored in the database. While homogenous transformations are used internally in the software, the rotation data is stored in axis-angle format to the database. Homogenous matrices are powerful and easy to use when calculating rotations and translations, but they have many redundant parameters and it is impossible to infer the pose of the object by looking at the transformation. The axis-angle representation, however, is a virtual opposite in

the way that it is very easy to visualize but hard to combine mathematically. Furthermore, in this representation the information is presented with only four variables, whereas a rotation matrix needs nine.

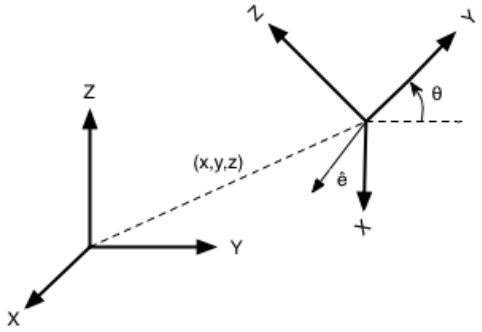

Illustration 10: Position, axis and angle representation.

The pose of a body is stored in

$$P = \langle position, axis, angle \rangle = (\bar{p}, \hat{e}, \theta), \text{ where}$$
$$\bar{p} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}, \hat{e} = \begin{bmatrix} e_x \\ e_y \\ e_z \end{bmatrix}. \tag{2}$$

The rotation axis and angle describe the orientation of the frame with respect to the global frame. The axis parameters are the coefficient of a (normalized) vector around which the object is oriented. The angle parameter corresponds to the angle the object is rotated around this vector. There exists a set of formulae to form a conversion between a rotation matrix and the axis-angle parameters. Given a rotation matrix

$$R = \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{bmatrix} \tag{3}$$

the rotation angle can be calculated with

$$\theta = \cos^{-1}\left(\frac{R_{11} + R_{22} + R_{33} - 1}{2}\right) \tag{4}$$

and the rotation axis with

$$\begin{bmatrix} e_x \\ e_y \\ e_z \end{bmatrix} = \frac{1}{\sqrt{(R_{21} - R_{12})^2 + (R_{02} - R_{20})^2 + (R_{10} - R_{01})^2}} \begin{bmatrix} R_{32} - R_{23} \\ R_{31} - R_{31} \\ R_{21} - R_{12} \end{bmatrix} \tag{5}$$

This transformation is singular when $\theta = 0 \vee \pi(rad)$. This singularity means that the object frame is aligned to the global frame. If theta equals zero the rotation matrix is an identity matrix. In this case the angle is zero and the axis can be freely chosen. In SimPartner the a axis vector is $\hat{e} = [1\,0\,0]^T$. If theta equals pi it means that the object frame is aligned with the global frame but some of the vectors are reversed with respect to the global frame. In this case the vector is determined by looking at the signs of the elements of the rotation matrix.

For example, if a frame is set to rotate at a constant angular velocity of $\omega = [1\,1\,1]^T$ the rotation axis is $\hat{e} = [\pm\frac{1}{\sqrt{3}}, \pm\frac{1}{\sqrt{3}}, \pm\frac{1}{\sqrt{3}}]^T$. The rotation angle grows continuously until the value reaches pi. Then the frame flips around over the singularity and decreases back to zero.

Force and torque

The force and torque table stores information received from ODE that affects the forces and torques applied to a body in a given simulation step through the collisions detected. This data can be used to validate the correct operation of the physics engine and also to monitor whether the forces exceed the material strength of the body in question.

3.4.2 Data analysis

For efficient control code programming, the user should be able to analyse the accumulated simulation data easily. Fortunately, there are several ways to achieve this due to the wide acceptance of the MySQL database software. The simplest way to access the data is to use a SQL client where the queries can be typed and the resulting rows can be copy-pasted to a spreadsheet program. This is the way the data in this thesis was generally analysed.

```
conn = database(dbname,username,
                password,driver,url);
data = fetch(conn,query);
time = cell2mat(data(:,1));
position = cell2mat(data(:,2));
plot(time,position);
```

Code example B: Accessing the database from Matlab.

However, there are more sophisticated ways to do the analysis. Matlab offers a wide

assortment of tools suitable for simulation data analysis. Matlab also has a Database Toolbox that can be used to convert database rows into Matlab workspace variables. With MySQL, the toolbox can be used by installing the MySQL Connector/J driver. MySQL Connector/J is a native Java driver that enables communication between the database and a client software, in this case Matlab. After installing this driver the user is able to access the data easily, as shown in code example B. This effectively integrates all the analysis power of Matlab to the SimPartner framework. It also demonstrates the importance of using widely accepted methods of storing data, as integration with other software is very straightforward.

3.5 Environment definition

The simulation environment (World) is defined using an XML file. The location and name of this file is defined in the program properties. The XML file is also validated against a DTD to guide the user to write conforming files that can be interpreted by the XML parser. The structure of an environment file is:

- **Surface plane**

  Surface plane of the simulation world is defined with four parameters *a,b,c,d* that are the coefficients of the equation

$$a*x+b*y+c*z=d \tag{6}$$

- **Gravity vector**

  Three components of the gravity vector of the environment $\vec{g}_x, \vec{g}_y, \vec{g}_z$.

- **Collision space**

  Type of the ODE collision space to use. The options are (Smith n.d.):

  - *Simple*

    No collision culling, checks intersection of all pairs, $O(n^2)$ complexity.

  - *Multi-resolution hash table space (Hash)*

    Uses internal data structure to record how objects are positioned in a three-dimensional grid of cells. Intersection testing has $O(n)$ complexity.

- *Quadtree space*

  Uses a pre-allocated hierarchical grid-based tree to quickly cull collision checks. “Exceptionally quick” for large numbers of objects, no complexity given.

- **Objects**

  Defines the external objects of the environment. These objects form the landscape of the simulator and can be interacted with, but they offer no sensor data and cannot be equipped with actuators. Objects are defined by their physical dimensions, rotational and translational velocities, and homogenous transformation matrix. An example of a body definition can be seen in code example C.

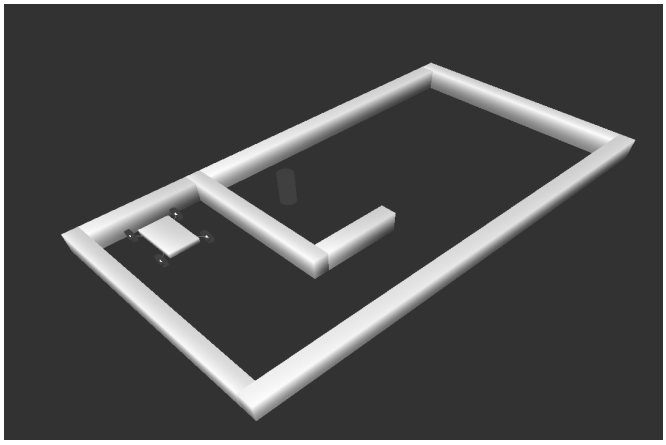

Illustration 11: A maze.

The XML file can store an arbitrary number of objects, thus forming the environment in which the robot operates. Thus far, no sensors or actuators can be placed to the

```
<object type="Body" name="TestParticle">
      <mass>1</mass>
      <transformation>
<T00>1</T00><T01>0</T01><T02>0</T02><T03>0</T03>
<T10>0</T10><T11>1</T11><T12>0</T12><T13>10</T13>
<T20>0</T20><T21>0</T21><T22>1</T22><T23>0</T23>
      </transformation>
      <translationalVelocity>
            <vx>0</vx>
            <vy>0</vy>
            <vz>0</vz>
      </translationalVelocity>
      <rotationalVelocity>
            <omegax>0</omegax>
            <omegay>0</omegay>
            <omegaz>0</omegaz>
      </rotationalVelocity>
</object>
```

Code example C: Definition of a test particle at rest at 10 m.

objects, only passive environments can be formed. Illustration 11 shows a simple maze that consists of seven box objects, one cylinder object and a simple wheeled robot. It is also possible to define joints between objects in the environment. The joints are passive and are mainly included for testing purposes and building simple joined structures.

3.5.1 Terrain modeling with heightfield

In addition to the surface plane defined earlier, SimPartner incorporates a possibility to model terrain features with a heightfield. If the user so desires, the heightfield can be set to be used in the properties file. The core of the heightfield is a height value matrix. This matrix stores the height values over a given set of index samples. Height y in matrix position (m,n) is given by

$$H=\begin{bmatrix} y_{11} & \cdots & y_{1m} \\ \vdots & \ddots & \vdots \\ y_{m1} & \cdots & y_{mn} \end{bmatrix}, where \quad m,n \geq 2. \tag{7}$$

The heightfield uses depth and width values to store information about the size of the field in the simulation environment. In other words a field with width w and depth d covers an area

$$\begin{aligned} &[\frac{-w}{2},\frac{w}{2}] \quad in\, x-direction\, and \\ &[\frac{-d}{2},\frac{d}{2}] \quad in\, z-direction. \end{aligned} \tag{8}$$

Combining 7 and 8 it is possible to calculate the height at points defined by the field

$$y(i\frac{w}{m-1}-\frac{w}{2}, j\frac{d}{n-1}-\frac{d}{2})=H(m,n)\, for \quad i\in 1\ldots m,\ j\in 1\ldots n. \tag{9}$$

Illustration 12: WorkPartner on a heightfield.

Points that reside inside the area but not at the points defined by the field are linearly interpolated by the physics engine. Furthermore, the field offers parameters for scaling the altitude values and setting an offset. The heightfield is thus a simple but powerful way for the user to create a terrain with a practically unconstrained accuracy.

3.6 Robot definition

Robots are stored as graphs where body objects are represented as nodes and joints as the vertices. The robot definitions are loaded from an XML file which is validated against a DTD in the same manner as the environment definitions. This design approach forces the user to

a) define robots in a syntactically correct manner where no essential information is left out and

b) build robots where all parts are connected with joints and the design is *complete* in the graph sense. This means that all the bodies are connected to each other by some path and all joints are connected to two bodies.

An example of a simple robot graph is presented in illustration 13. The XML file structure for different body types is similar to the one used to describe environments. Added features are sensors and actuators that enable the robot to sense and interact with its environment.

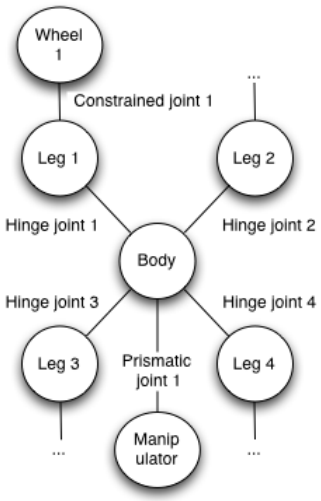

Illustration 13: Robot graph.

3.7 Sensors and actuators

Sensors and actuators are the only way the user can directly control the robot when the simulation is on-line. Sensors form a hierarchy where the base class uses a TCP

communication protocol implementation to relay data with the client software. Further functionality is defined in the derived class.

3.7.1 TCP/IP communication

Layer	Technology
4. Transport	TCP
3. Network	IP
2. Data link	MAC
1. Physical	Binary

Illustration 14: first layers of the OSI model.

SimPartner uses the four first layers of the OSI model in its implementation of sensor/actuator communication with the client software. The software itself is the fifth, application layer. The port of the sensors and actuators is defined in the XML file with two conditions:

1. The port number must be greater than 1024.

2. The port number must be unique.

The framework then reserves this port for the robot device model in question. The SimPartner implementation is a server model, the communication has to be initiated by the client software. The communication implementation uses sockets, a method of point-to-point communication defined in RFC 147. This is a standard way of communicating with two remote machines, but can also be used in one computer, which is often the case with the SimPartner framework. This approach also makes it possible for the user to program the client software with the programming language of his choice as this approach is well documented and implementations exist for most of the major programming languages.

3.7.2 Sensors

Sensors are always attached to a body in the robot. The principle of the communication protocol of the sensor is very simple. When communication is established, the first thing the sensor does is to send its information to the client software. The client can also send data to the sensor, this is used to define the sensor's operational parameters. The sensor then parses the data sent by the client

software and adjusts its behavior accordingly. The advantage of this approach is that the client can acquire fresh sensor data easily but also use the same communication to control the sensor. Sensors that are initially included in the SimPartner framework are described below. Currently velocity and force sensors are not implemented but due to the modular structure they can easily be added later.

Echo sensor

This sensor simply echoes the last input command it has received to the client. It is usable for testing the correct operation of the server/client interface.

Position sensor

This sensor takes the coordinates of the body it is associated with from the ODE engine and passes them to the client. It can be used for creating simple control interfaces in the client. More advanced sensors, i.e. with noise or increasing error can be extended from this sensor type.

Scanner sensor

A scanner sensor models a laser scanner. It has a range of 10 meters with no added noise or bias. Eleven sensor rays are emitted from the sensor to cover an area of ± 0.1 radians of the z-axis in the z – x -plane of the body to which the sensor is attached. The angle between two emitted rays is thus 0.02 radians. The rays are primitives in the Open Dynamics Engine and are included in the collision detection algorithm. When a collision between a ray and a body is detected, no forces are applied but rather the distance between the starting point and the collision is calculated. This makes it possible to retrieve the distance of the object from the sensor. This information is stored in the sensor and sent to the client when queried.

3.7.3 Actuators

Actuators can be built on top of constrained and prismatic joints. This constrained joint uses a ball joint and a motor that both connect the body parts together and make it possible to record its angular position with respect to the original state. It is the possible to control this joint to move so that a desired position is reached. The joint can also be set to revolve at a constant velocity or to deliver a desired force to the bodies it is connected to, creating an angular motor. When the actuator is built on top a of a prismatic joint it models a linear motor. The joint also has stops, angle values with respect to all three axes that can be set to model the physical limitations of the

joint. These stops must be set within reasonable limits in joints that are controlled by angle values, otherwise the joint may become unstable, causing the robot to oscillate wildly.

The actuator is then modeled on top of this joint by using a structure similar to the one used in the sensor, described above. Initially all actuators are set to a state where the actuator tries to keep its original position and orientation with a force of 1000N on each axis. When a command is sent to an actuator, it responds with its previous state and records the new state if the command is recognized to be within a predefined communications format.

Joint controller

Joint controller is a motor that controls the angular velocity of the bodies to which it is attached. It accepts three maximum force values (in newtons) and three velocities (in radians per second) as an input[1] and sets its internal parameters accordingly. The maximum force values reflect the greatest force per joint axis which the controller can use to achieve the desired velocity.

Angle controller

Angle controller can be used to set a motor to a desired angle. It incorporates a PID-controller with anti wind-up protection. The input for this controller is three maximum force values, three angles and the values for P, I and D parameters[2].

3.8 WindowManager, visualization and control

SimPartner framework uses OpenGL to visualize the simulation to the user. OpenGL offers platform-independent graphics functions that are designed to be versatile but still easy to use. The terrain and the robot can be plotted in several different ways, either as a wireframe or with solid faces. Object frames and forces affecting the bodies can also be plotted. The camera can be controlled by freely moving it through the space or it can be set to follow the robot.

The SDL library is used to pass commands from the user to the framework. SDL offers a set of events generated by the keyboard, mouse or operating system. User interaction can be achieved by programming functionality based on these events.

1 Message example: <1000,1000,100,0,0,0.2>
2 Message example: <1000,1000,100,0,0,0.1,0.5,0.1,0.1>

3.9 WorkPartner model

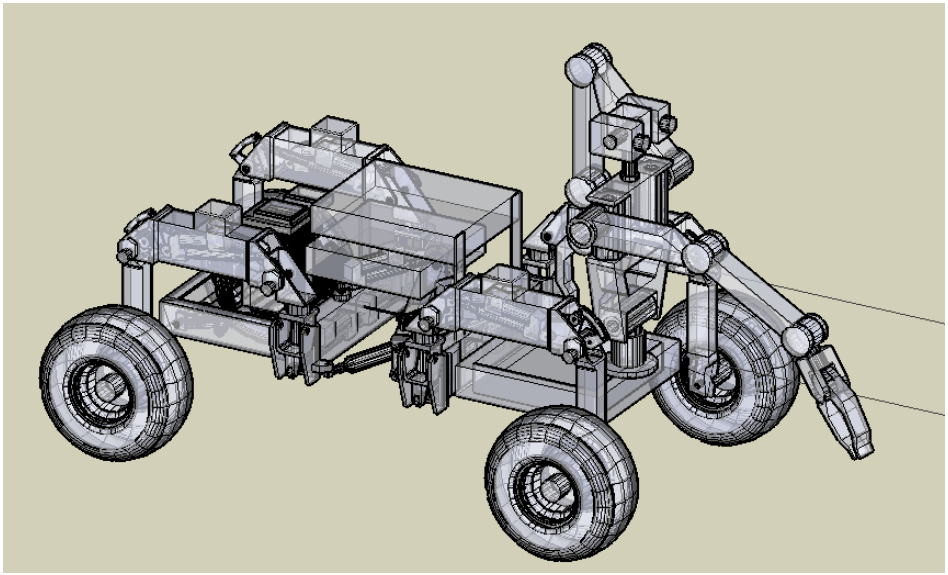

Illustration 15: WorkPartner CAD model.

The robot is modeled by using the latest available CAD model of the robot drawn in 1999. The configuration information and measurements were read from the model and translated into a robot XML file described in section 3.6. Due to the complexity of the model, the robot was modeled in different generations, which are described further. The idea behind this design approach was to validate the correct functionality of each design iteration before new parts were added.

3.9.1 Generation 1

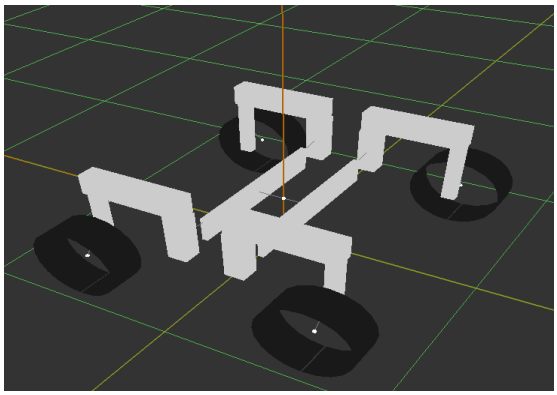

Illustration 16: 1st generation model.

The first generation model of the robot contains the wheels and supporting structures for the undercarriage. The wheels and other joints have simple motors that can be used to drive the robot. The fact that the wheels of the WorkPartner robot are not steerable causes added complexity to the steering system. The robot chassis consists of two independent halves and has an intricate lever system that is used to split-steer

the vehicle. This means that the two halves of the robot can be adjusted to be at an angle with respect to each other, causing it to turn when the wheels are spinning. In the first generation model, this system is modeled with one hinge joint that can be twisted to achieve split steering.

The model was primarily used to confirm the correct operation of the remote actuator control over the TCP/IP system using a gamepad controller. The wheels were controlled with a joypad where forward/backward commands increased the rotational velocity of the wheels and left/right commands caused the central joint to turn for a prespecified amount. Other leg joints can also be controlled to change the configuration of the robot. This client can also be used with models of the later generations.

3.9.2 Generation 2

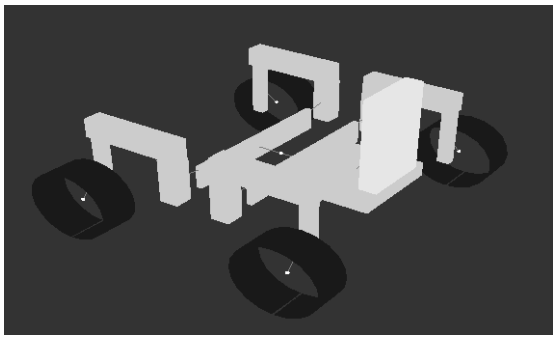

Illustration 17: 2nd generation model.

The second generation model incorporates a front support and a torso that has a laser scanner model and a position sensor. The torso can be rotated using a simple angle controller. This robot can sense its environment and could be used for creating crude SLAM navigation clients. All other joints and motors are the same as in the generation 1 model.

3.9.3 Generation 3

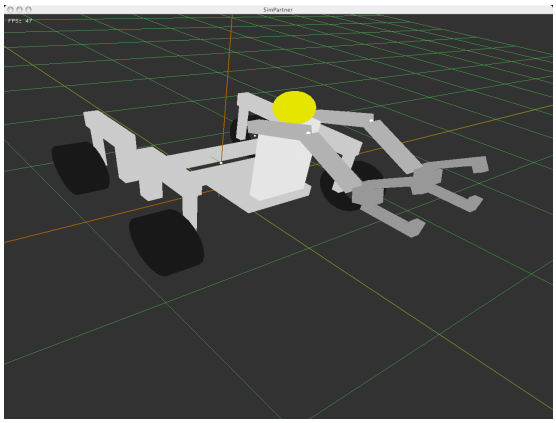

Illustration 18: 3rd generation model.

For the third generation, manipulators are attached to the torso. The manipulators consist of upper and lower "arms" plus a pair of claws. The part that is connected to the torso and the lower part are connected using simple angle controllers so that the positions of the manipulators can be controlled accurately. The wrist that connects the claws to the arms can be turned. Both sides of the claws can also be controlled independently of each other with three degrees of freedom.

3.9.4 Generation 4

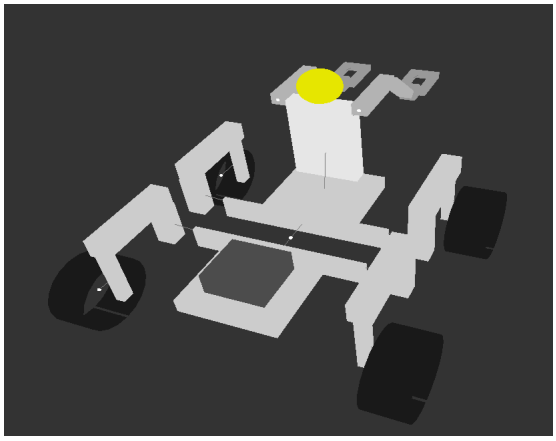

Illustration 19: 4th generation model.

The actual WorkPartner robot is tilted to the left and has a power pack on the back of the robot. So for the validation against real WorkPartner test results a more realistically balanced model was created. It is no longer symmetrical, the torso is shifted five centimeters to the left and the masses of the parts have been altered to make it compatible with the actual robot. Furthermore, a rear support and a weight have been sited to make the weight distribution resemble that of the actual robot.

3.10 SimPartner clients

Due to the versatile TCP/IP control structure and the standardized communication protocol for sensors and actuators, it is easy to create different programs to control the robot. The control methods created within the scope of this thesis are the interactive client and the sequencer client.

3.10.1 The interactive client

The first client developed for the SimPartner framework was the interactive gamepad client. The test setup for this client is shown in illustration 20. The client runs on a Linux PC and the commands are input using a game controller that is connected to the computer's USB port. The client software transforms the commands into values that are encapsulated into the communication protocol and transferred to the simulator over a local network. The advantage of this approach is that it allows an easy way to test new object/joint configurations and other types of fast prototyping. It was also useful in the development phase of the simulator software, as changes can be made easily and the effects observed visually.

Illustration 20: Test setup for the gamepad client.

3.10.2 The sequencer client

The sequencer client takes an XML-file as input and parses it. After this it sends the commands to the robot. The client is thus not interactive but rather meant to be used to create complex command sequences that would be impossible to perform accurately using the interactive client. Furthermore the sequencer client makes it possible to run the exact same series of commands several times, for example with

```
<command>
      <commandString>1000,1000,100,0,0,0</commandString>
      <port>1028</port>
</command>
<command>
      <commandString>1000,1000,0,0,0,0</commandString>
      <port>1030</port>
</command>
<command>
      <commandString>0,0,-0.2,0.5,0.1,0.1</commandString>
      <port>1044</port>
</command>
<command>
      <commandString>2</commandString>
      <port>0</port>
</command>
<command>
      <commandString>1000,1000,100,0,0,0</commandString>
      <port>1030</port>
</command>
<command>
      <commandString>1000,1000,0,0,0,0</commandString>
      <port>1027</port>
</command>
<command>
      <commandString>0,0,-0.2,0.5,0.1,0.1</commandString>
      <port>1041</port>
</command>
```

Code example D: Control sequence.

different environmental parameters. The user needs only to type in a new version number to the XML file and program the relevant parsing code to the file. See code example D for details on sequencer client code. Version 1, which is included with the SimPartner framework, is a simple sequencer in which commands can be sent to the client and which has built-in pausing and repeating commands. The commands sent will be valid until the next command arrives. For example, if a certain wheel motor velocity is set the motor will rotate with this velocity until the new velocity is given.

4 SimPartner Performance

This chapter describes the performance of the SimPartner framework by introducing a series of tests done with it. The results are compared against mathematical models and real test data to validate the accuracy and realism of the simulator.

4.1 ODE accuracy

4.1.1 Integrator

The accuracy of the Open Dynamics Engine can be verified by a series of tests that can be compared against theoretical result values. The simplest of these tests is to check the integrator accuracy using a test particle in free fall with earth gravity g. A test particle was positioned to an altitude of 500 meters and the simulation was started. The results were then compared with values computed with the formula

$$h(t) = h_0 - \frac{1}{2} g t^2 \tag{10}$$

that describes the ideal free fall motion. The results of this test are shown in Figure 1. It can be seen that the error increases linearly as a function of time. This is an expected result as ODE uses 1st order Euler integration. The step size of the

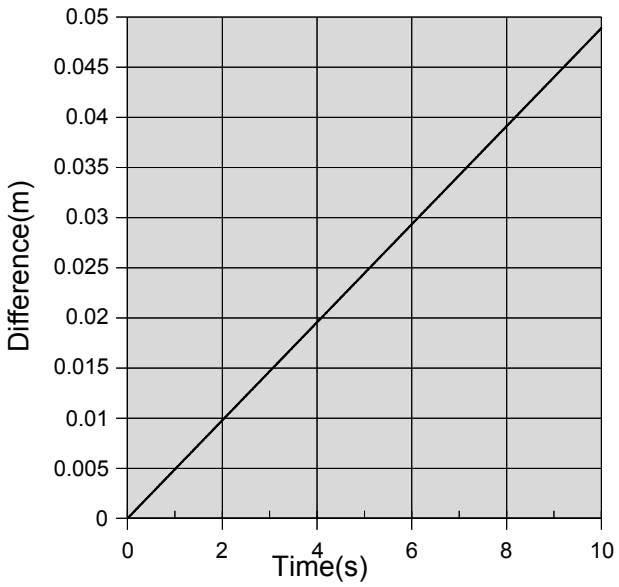

Figure 1: Incrementing error of a body in free fall.

simulation was 0.001 seconds and the resulting error can be described with a linear integration error coefficient

$$e(t) \propto e_t * t, \text{ where } e_t = 0.00489\,\text{m/s}. \tag{11}$$

When the simulation is repeated with different time step lengths as illustrated in Figure 2, a more general error estimate can be formed. There is a linear correlation

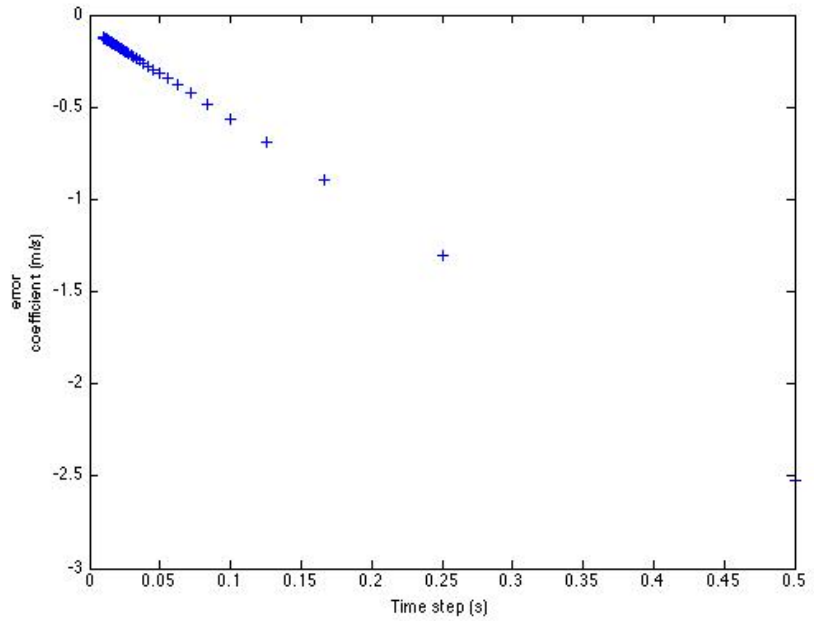

Figure 2: Effect of time step on integration error coefficient.

between the integration error coefficient and the time step length. This is again an expected result as ODE uses first order integration to solve the positions of bodies. Integration error coefficient can be approximated with

$$e_t(\Delta t) \propto e_h * \Delta t, \text{ where } e_h = 4.9050\,\text{m/s}^2 \tag{12}$$

As can be seen, this coefficient is almost exactly half of our gravitational constant. The magnitude of the force applied to the body also affects the error. To test this, a free fall simulation is run with the gravitational constant value of 50. The simulation yielded an integration error coefficient of 0.025 with time step of 0.001. This is an expected result as it is approximately five times larger than our previous result, as is our force. The compounding integration error can thus be approximated to be

$$e(t) \propto \frac{F}{2} * \Delta t. \tag{13}$$

This yields the result that to keep the simulation accurate it is necessary to keep the time steps and affecting forces as small as possible, as was expected. Also to be noted is the fact that motion with constant velocity is simulated with near perfect accuracy. To test this, a body was simulated with a velocity of $\bar{v}=(10,10,0)$. The vertical position error grows linearly as before but the horizontal position error remains zero. This is shown in figure 3.

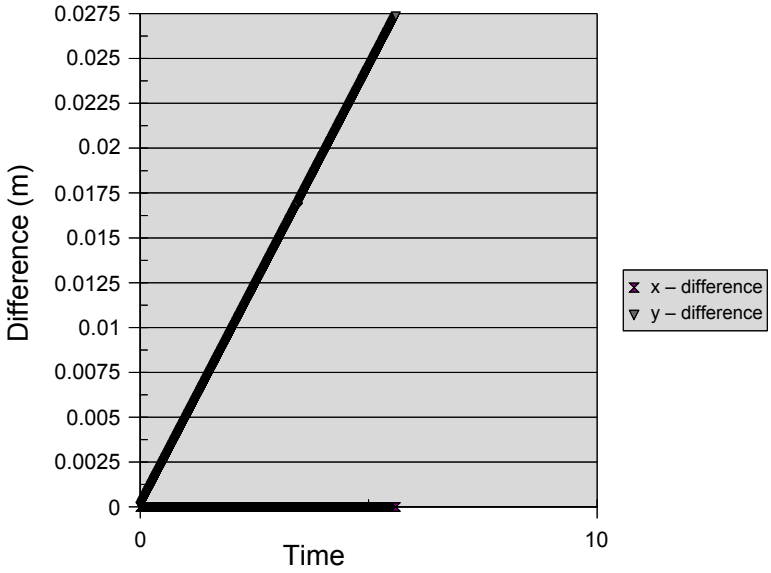

Figure 3: Falling body with initial velocity.

4.1.2 Friction

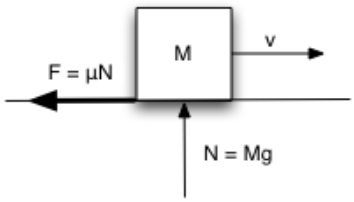

Illustration 21: Coulomb friction.

ODE uses the Coulomb friction model. In this model two friction forces can be defined. Tangential friction is directly proportional to the normal force and opposite to the velocity vector. Normal friction is perpendicular to the velocity vector and can be used to make vehicles skid in turns. It is thus assumed that the mass and contact surface area of the object do not affect the velocity change but rather that the velocity should decrease linearly depending only on the gravity and the friction coefficient, following the equation

$$v(t)=v_0-\mu g t. \tag{14}$$

To assess how well ODE simulates friction, a simulation setup was constructed in which a (1x1x1m) cube weighing 1kg is set up with an initial velocity of 10m/s.

However, an interesting behavior is observed in the simulation. The velocity error does not grow linearly as before, but in a stepwise manner. This behavior cannot be explained by any physical processes but rather it has to do with the implementation of ODE. This behavior is shown in Figure 4. Most likely it is due to the fact that the object is not sliding with constant contact points but rather bouncing slightly as the contact forces are not evenly distributed over the contact surface. This would cause impulses and torques acting on the object.

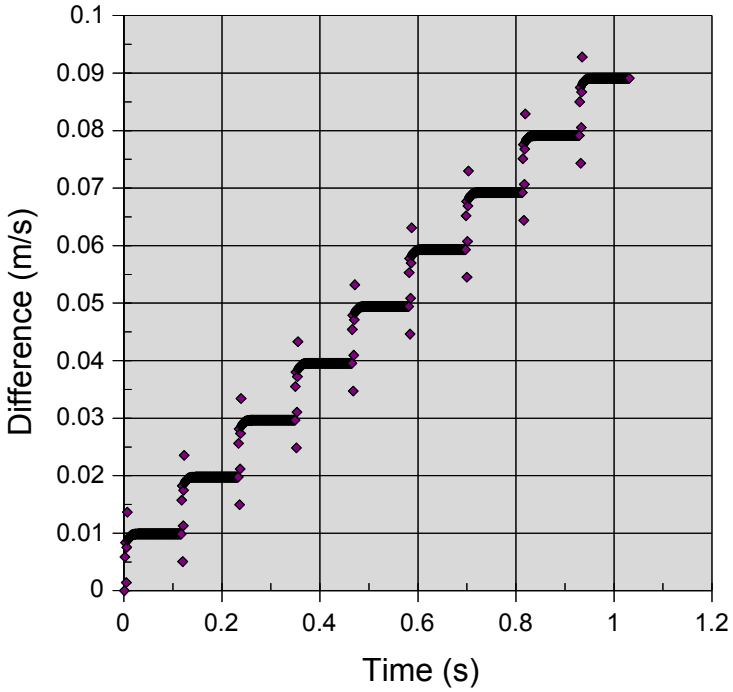

Figure 4: Velocity difference of a sliding cube with friction.

4.1.3 Collision

The accuracy of the collision engine was determined by colliding two spheres with identical mass and equal but opposite velocity vectors and starting positions. The goal of the simulation was to create a perfectly symmetrical and elastic collision between the two spheres. All error correction and bounciness parameters were set to zero. Figure 5 depicts the trajectories of the spheres. It can be seen that the collision takes place in the predicted altitude and position, with the distance of the centers of mass being 1.0 which is equal to two times the radius of both spheres.

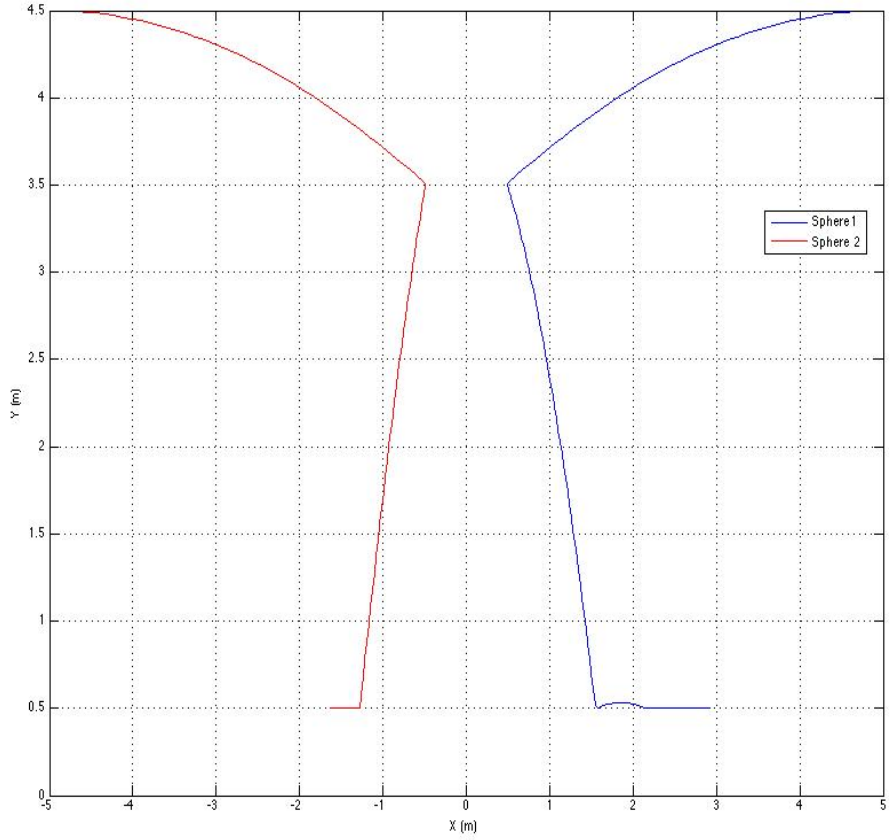

Figure 5: Collision of two spheres.

The collision, however, is not ideal. The velocity of sphere 1 (right) is slightly higher than that of sphere 2. This causes the sphere to bounce back with a greater velocity and even bounce from the ground plane. This in turn causes asymmetry in the positions of the spheres. This behavior is illustrated in figure 6, where initially the positions of the two spheres are perfectly symmetrical. After the spheres come in contact with the ground plane the asymmetry starts to grow *ad infinitum* as the rolling friction is not used and the velocity of the spheres is different.

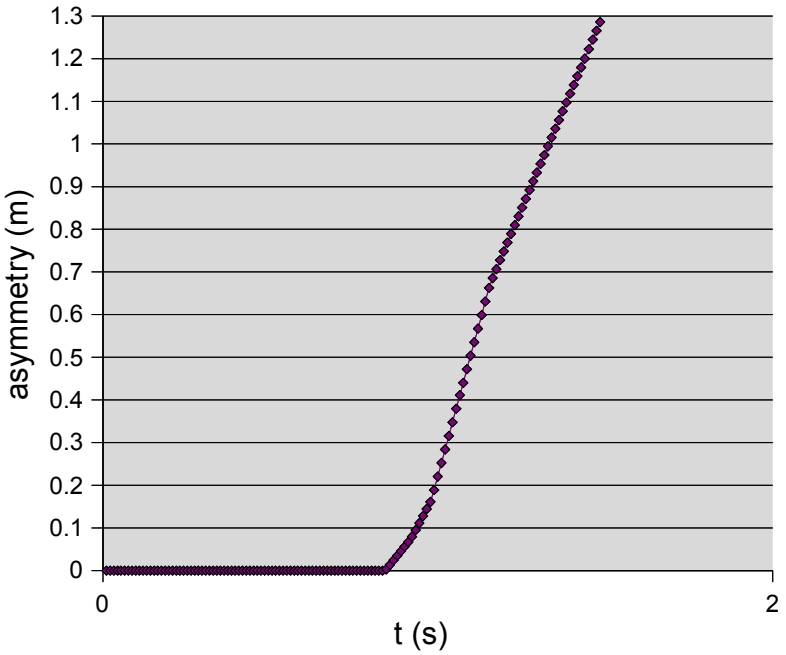

Figure 6: Asymmetry of the collision of two spheres.

4.2 Robot behavior

As described in section 3.9, the robot was modeled in several different generations. These models were also validated in parallel with the development. This section describes the validation tests made with the models. The validation was done mostly against mathematical models due to the fact that there exists only a limited amount of test data on the actual robot. However, the final validation was made against the test data (Leppänen 2007).

4.2.1 Generation 1 – driving a circular path

The simulation was started and the robot was turned so that smallest possible turning circle radius was achieved. The robot was then set to drive on a circular track using the gamepad client and the body part positions stored in the database. Robot leg position data was then extracted from the database (see appendix 5) and the robot leg coordinates averaged to obtain the trajectory of the center of the robot. The results were that the robot travelled a circular path with turning circle radius of 4.30 meters shown in figure 7. The result was obtained using the friction values of 0.5 and 1.0 for tangential and normal friction, respectively.

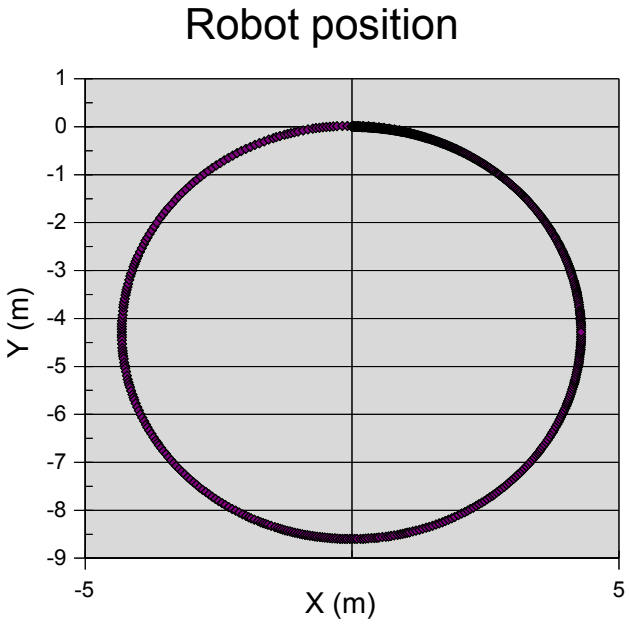

Figure 7: Trajectory of the robot when driving a circular path.

The geometrical center point of the robot is the turning center. Thus both wheel axes point to the center of the circular trajectory (see illustration 22). From this symmetry it is possible to deduce the turning radius using the sine rule and properties of triangles. This simplified trajectory assumes that there is no skidding of the wheels and that the steering angle stays fixed for the whole duration of the maneuvre. Calculating the theoretical turning radius of the vehicle makes it possible to assess and verify the simulation.

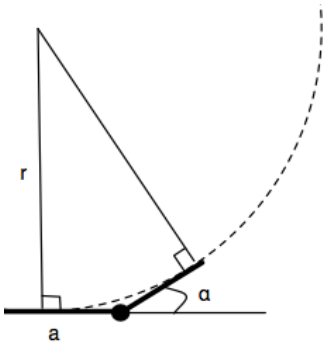

Illustration 22: Turning radius of a vehicle with split steering.

The turning radius of the robot can thus be calculated with

$$r = \frac{a \sin(\pi - \alpha) \sin(\frac{\pi - \alpha}{2})}{\sin(\frac{\alpha}{2}) \sin(\alpha)}. \quad (15)$$

Which can be further simplified to

$$r = \frac{a \cos(\frac{1}{2}\alpha)}{\sin(\frac{1}{2}\alpha)} = a * \cot(\frac{1}{2}\alpha). \quad (16)$$

When the turning radius is calculated using (16), using the predefined steering angle $\alpha = 0.2972\,\text{rad}$ and the axle length $a = 0.6\,\text{m}$ the result is 4.01 meters. The error in turning radius is 0.29 meters or 6.7% which shows that there are significant unidealities associated with the simulation when using the parameters described above. The main reason for the error is the fact that the speed of the wheels cannot be accurately controlled using the gamepad client. All the wheels are turning with the same velocity, whereas the inner wheels should turn slower than the outer ones. This explains the fact that the simulated trajectory has a larger radius than the calculated.

To further test the turning behavior the velocities of the wheels must be considered. The inner wheels travel a shorter distance and since the robot does not have a differential the velocities must be calculated manually. The distances the wheels travel can be determined from

$$\begin{aligned} d_i &= 2\pi r \\ d_o &= 2\pi (r + a), \end{aligned} \quad (17)$$

where r is the radius of the circle and a is the axle width. From this it is possible to derive

$$\begin{aligned} v &= \frac{d}{t} \\ \Rightarrow v_i &= \frac{2\pi r}{t}, v_o = \frac{2\pi (r + a)}{t} \\ \frac{v_i}{v_o} &= \frac{\frac{2\pi r}{t}}{\frac{2\pi (r + a)}{t}} = \frac{r}{r + a} \\ v_o &= v_i (\frac{a}{r} + 1), \textit{when } r > a. \end{aligned} \quad (18)$$

To test this the turning angle is set to $\alpha = 0.2\,\text{rad}$. This corresponds to a turning circle radius of $r = 5.98\,\text{m}$. The inner wheel pair velocities are set to be

$v_i = 1.00 \frac{m}{s}$, thus giving us the outer wheel velocity $v_o = 1.10 \frac{\text{m}}{\text{s}}$.

This configuration was tested using the sequencer client and the simple angle controller. The central joint was turned to the desired angle and the wheel motors started. The accuracy of the simulation improved dramatically.

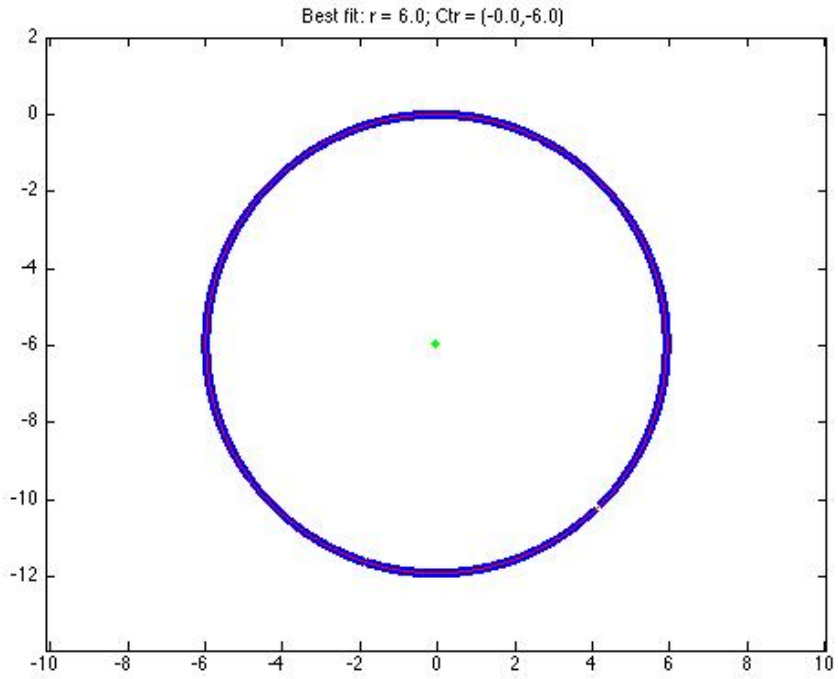

Figure 8: Circle fitted to the data points of the trajectory.

The simulation lasted for nearly two minutes, generating over 346 000 rows in the database. This means that there is a large amount of data that can be analyzed to verify the simulation. First, the path of the center of mass of the robot was calculated as above. The trajectory was very close to the estimated one, generating a circle with a radius of 5.9516 meters and a radius error of 2.8 cm or 0.5%. The center point of the fitted circle is (-0.0292,-5.9512) and so the error is also about the same in this respect.

Figure 9 shows the error compared to the fitted circle as a function of time. The error is large at the beginning as the vehicle is only turning its central joint and is not yet travelling in the desired trajectory. The error then decreases for some time until it starts to increase again. A slight oscillation can also be detected. It can be noted that the accuracy is very good and the track remains circular up to within one millimeter throughout the whole simulation.

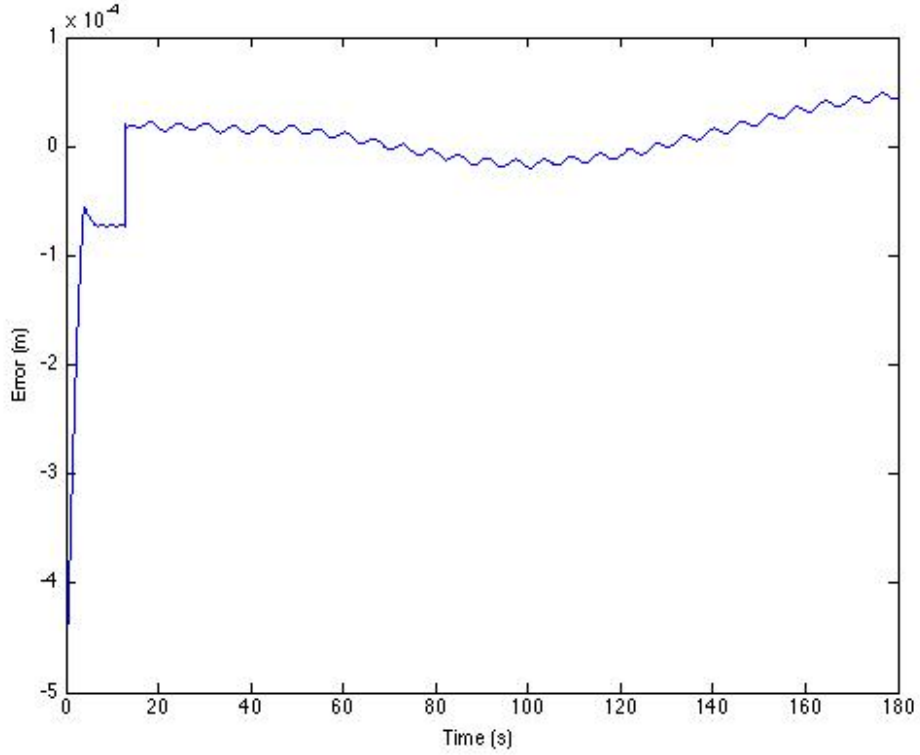

Figure 9: Error as a function of time in a circular trajectory

4.2.2 Generation 2 – a moving laser scanner

The second generation robot model includes a torso with a simple laser scanner and a position sensor. The torso of the robot is turned with a constant angular velocity of $\omega = 0.01 \frac{\text{rad}}{\text{s}}$ The simulation environment has a wall set at a distance of six meters from the robot center, which gives $L = 5.335\,\text{m}$ from the scanner.

Figures 29 and 30 in appendix 6 show the sensor readings while the robot is turning. Sensor rays that are pointing away from the turning direction show first a decrease in the measured distance, then the distance starts to increase. The rays that point

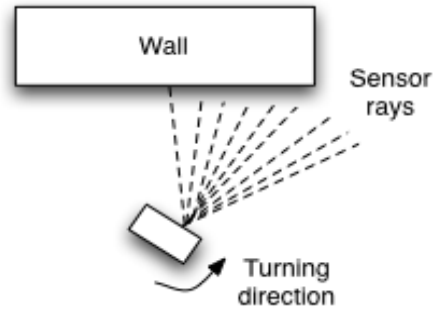

Illustration 23: A moving laser scanner.

towards the turning direction show a steady increase in the distance, which is expected. In the situation described above the distance measurement can be calculated with

$$d = \frac{L}{\cos\alpha} = \frac{L}{\cos(\alpha_0 - \omega t)}. \tag{19}$$

The sensor error behavior is shown in figure 10. After an initial jitter, the error grows linearly. This is consistent with the integrator error of the ODE and offers an insight to the best possible accuracy that any ODE sensor can achieve. The error levels are

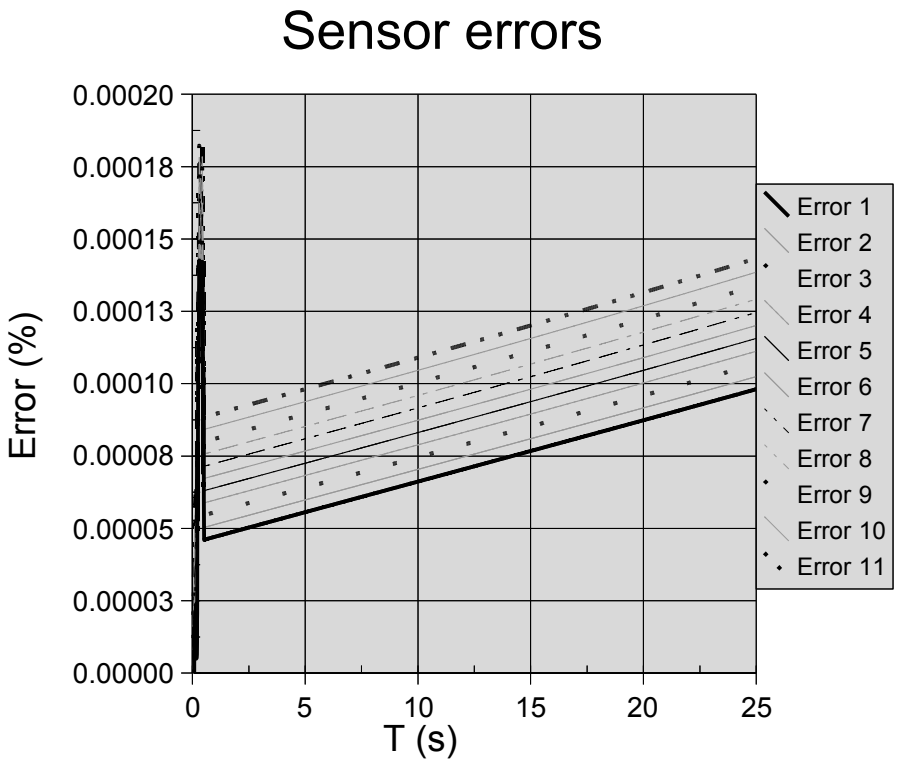

Figure 10: Distance scanner sensor error.

also proportional to the distance to be measured, meaning that the longer the distance, the greater will be the relative error.

4.2.3 Generation 3 – Manipulator

In this generation, the WorkPartner model includes all the essential parts of the actual robot. The first validation done to this model was a test in which the robot advanced towards a pole held on the top of two cubes. The robot then picked up the pole, advanced towards another pair of cubes and laid down the pole smoothly so that it remained on top of the other pair of cubes. This validated the ability of the simulated robot to perform a complex task.

The sequencer client was used for this simulation. The command sequence was completely passive, no sensor data was used to guide the robot. In practice this means that the command sequence was done in steps, with motor on/off commands and pauses. The command sequence was

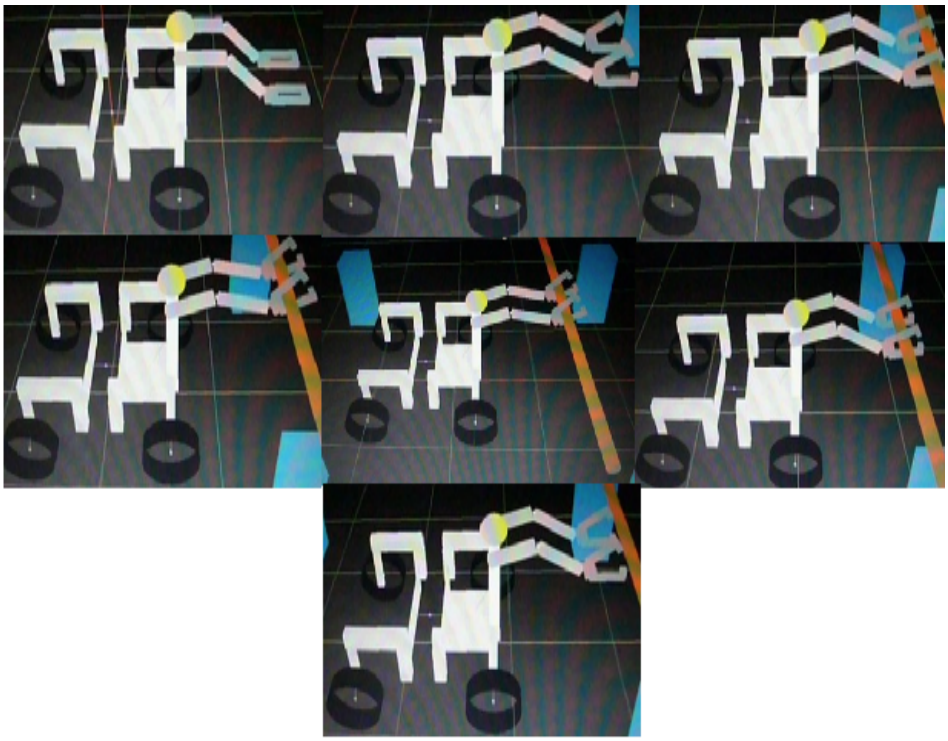

Illustration 24: Action sequence.

- Simulation start.
- Open claws, raise and twist the manipulators so that the claws are aligned correctly. Turn wheel motors on.
- Drive for a specified amount of time, then turn wheel motors off.
- Close claws.
- Raise manipulators.
- Wheel motors on.
- Drive for a specified amount of time, then turn wheel motors off.
- Lower the manipulators
- Open the claws
- Wait for a specified amount time to allow the pole to stop moving.
- Wheel motors on backwards

The purpose of this validation was to prove that SimPartner is able to perform complex task sequences and that the robot performs well. A more advanced version of this task could be done using a client that would have sensors and the commands

would be calculated from the actual position/environment data. This would reduce the development time of the control code and increase the accuracy of the robot.

Force measurements were also recorded during the simulation. Figure 11 shows the sum of the forces in the y-direction or opposite the gravity vector (see appendix 5). During the time interval of 10 – 25 seconds the robot is moving and picking up the pole weighing 48.9 newtons (mass 5 kg). This can be seen as noise and oscillations in the graph. At 25 seconds the pole is held by the robot, which causes a level increase in the total load. At approximately 45 seconds in the simulation, the robot lowers the pole back on the supporting pylons, after which the robot backs off.

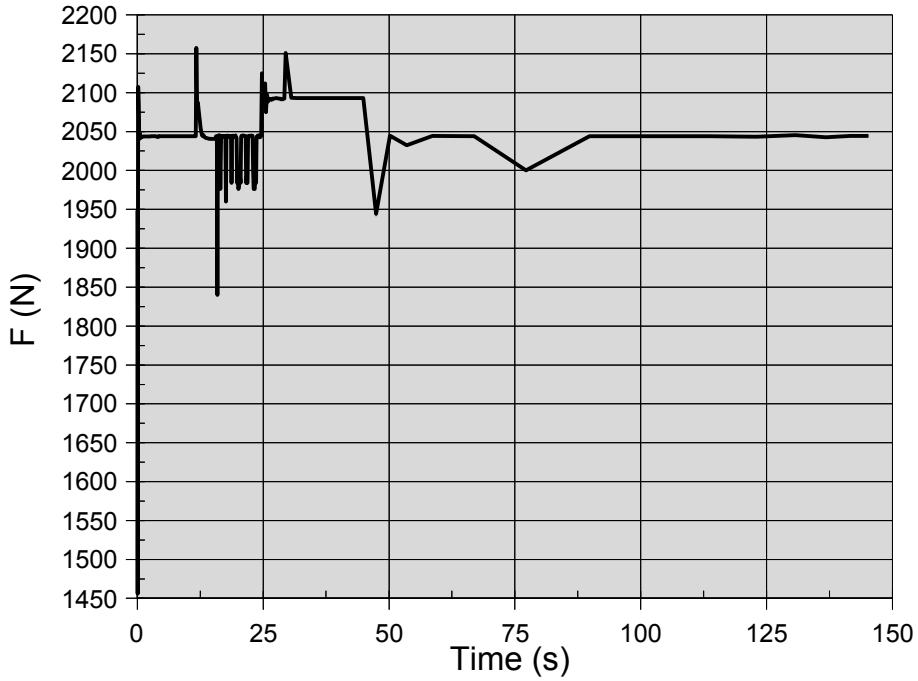

Figure 11: Sum of the wheel forces.

An interesting detail in the wheel forces can be seen when individual wheel forces are plotted instead of the sum. Wheels one and two (on the positive side of the x-axis, or in front of the robot) carry a substantially larger amount of the weight of the robot than wheels three and four. This can be explained by the geometry of the robot; the torso and its support structures make the front end significantly heavier. The difference is about 2 to 1, or 700 N and 350 N per wheel in each pair. This agrees with the fact that the front support and torso structure mass was set to be 51 kilograms when the whole weight of the robot was 202 kg. The behavior is illustrated in Figure 12.

Actually, this behavior has been observed in the WorkPartner robot. The robot is powered by a gasoline engine and batteries which are placed on the rear part of the robot to improve stability by acting as a counterweight for the long manipulators.

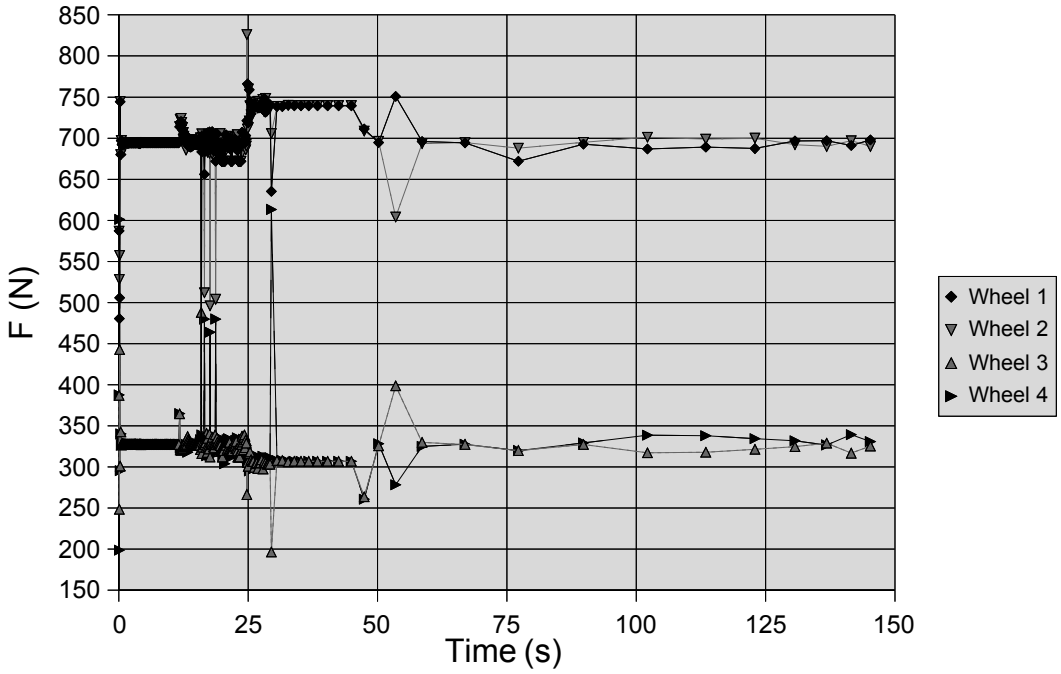

Figure 12: Forces affecting each wheel in the third generation simulation.

The weight of the pole also appears to be carried mainly by wheels one and two. Furthermore the forces on wheels three and four decreases when the pole is grabbed this is explained by the fact that the pole actually causes the whole robot to tilt, further decreasing the load on the rear wheel pair.

4.3 Use case – control code development

The next validation is to verify that the control code can be developed using the SimPartner framework. The scenario is that the robot is placed on a very slippery surface (μ = 0.025) and the goal is to make the robot move as fast as possible. The assumption is that it is possible to use wheel walking to make the robot move with a greater velocity.

Due to the great forces that take place in the rolling walking movement, a 1st generation robot model was used. It was observed that trying to achieve rolling walking directly in the way it is done in the WorkPartner (see section 4.3.3) causes simulation instabilities. Therefore, the code development was performed in stages as shown in the following paragraphs. The mass of the robot in this simulation is 18 kg, which corresponds to a weight of 176 N.

4.3.1 Movement by rotation of skidding wheels

The initial setup is simply to set the rotational velocity of the wheels to be one radian per second, which will cause the robot to move slowly as the wheels are mainly skidding and only partly moving the robot forward. The motion data is retrieved from the database and the average velocity of the center of mass is calculated to be 0.0038 meters per second (see appendix 5). The velocity without skidding would be

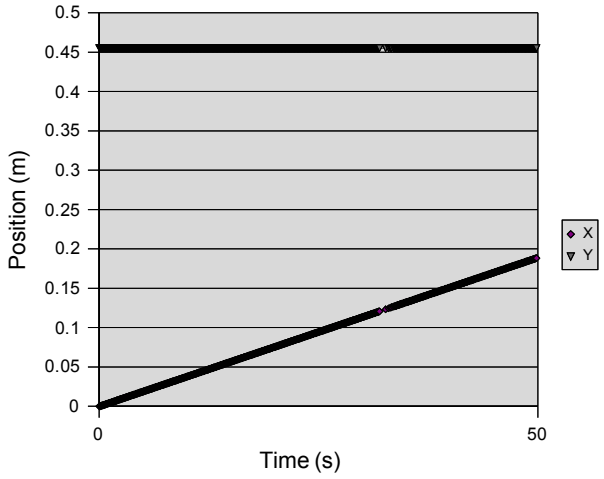

Figure 13: Robot CoM position with skidding wheels.

0.235 meters per second, so it can be seen that the speed is severely reduced by the skidding of the wheels. The y-position stays very steady, as would be expected. This can also be seen in the wheel forces, shown in figure 14 (for wheel number reference to geometry, see figure 19). Despite the small high frequency oscillation, caused by the numerical inaccuracies of the simulation, the forces are in the correct range since the ideal force would be 44 newtons per wheel.

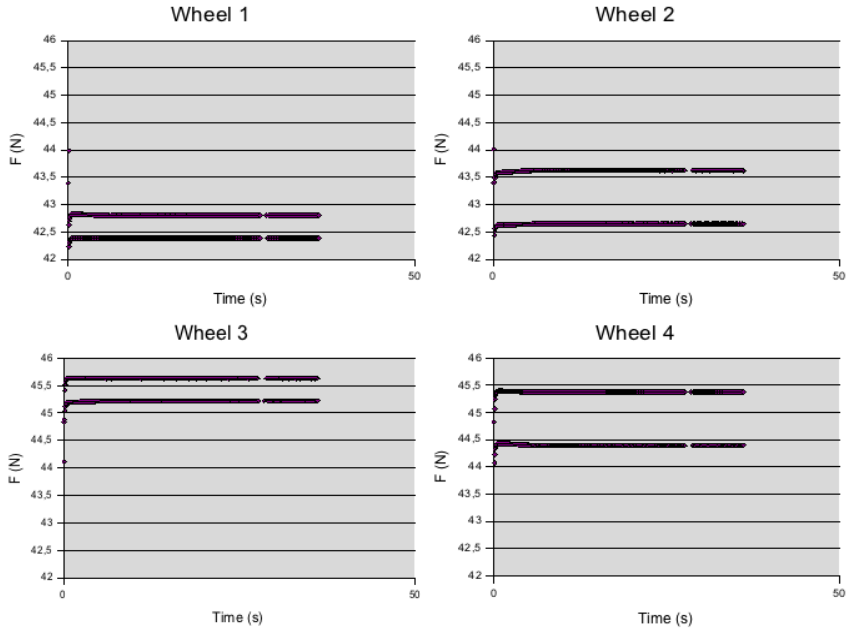

Figure 14: Wheel forces in skidding wheel movement.

4.3.2 Caterpillar movement

The next improvement was to use the robot "knee" joints. The simplest way to achieve this is to use caterpillar-like movement. The motion sequence is as follows:

1. Rear wheels are locked and front wheels set in free rotation. Rear legs are then pushed back and front wheels forward.

2. Rear wheels are unlocked and front wheels locked

3. Rear and front wheels are brought back under the robot.

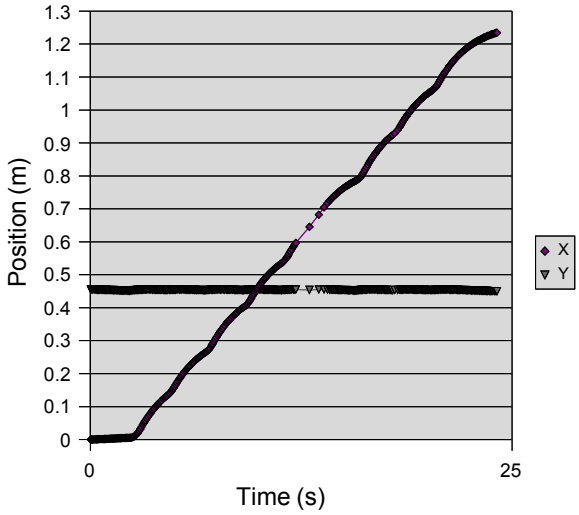

Figure 15: Robot CoM position with caterpillar locomotion.

This approach gave great increase to the velocity of the robot. The movement is fairly smooth and natural looking. There is a slight oscillation in the turner joint y-position (amplitude 1 cm). The average velocity during the course of the simulation increased dramatically, to 0.057 meters per second. The behavior of the wheel forces is entirely different, shown in figure 16.

Wheels one and two are in the front. The large oscillation, with the other side carrying most of the weight of the vehicle, is caused by the contact approximation of the simulator that causes the entire chassis to rock slightly in this locomotion mode. There is still a significant difference in the load distribution between wheels one and four on the right side of the robot and wheels two and three on the left. The caterpillar locomotion causes load variation in the order of 15-20 newtons per wheel in the course of the simulation.

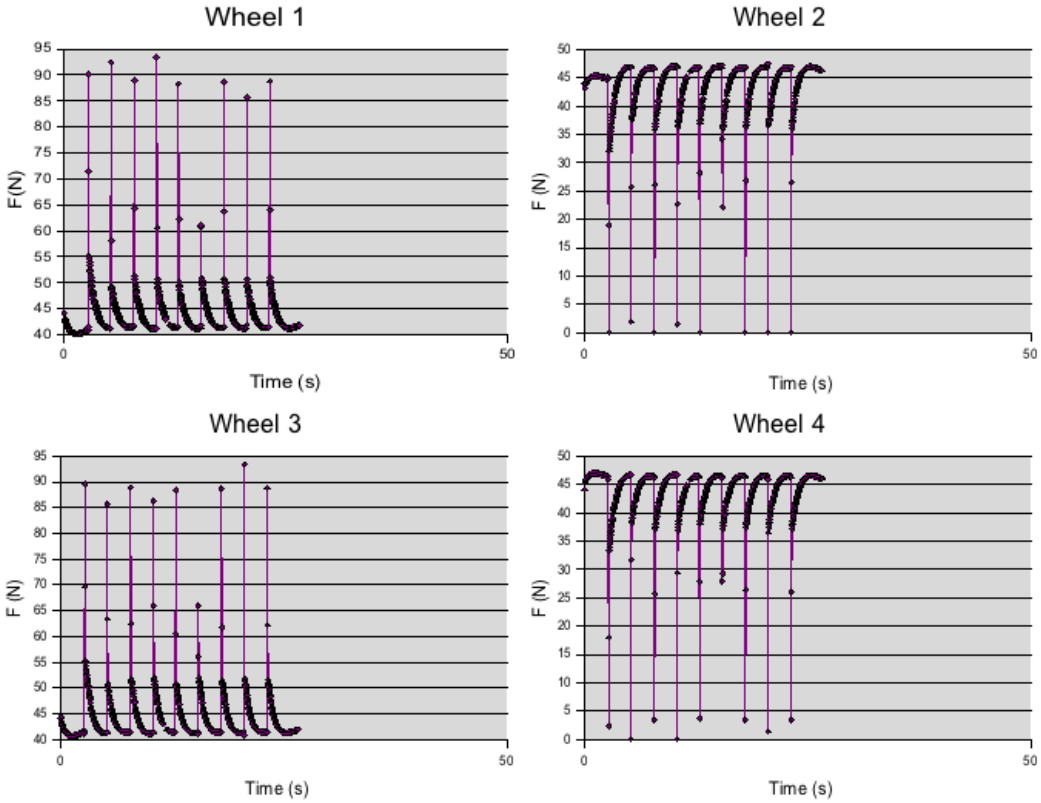

Figure 16: Wheel forces in caterpillar locomotion.

4.3.3 Rolling Walking

Next, a rolling walking (rolking) simulation was developed. The development is based on a video[1] which describes this locomotion type. The rolling walking sequence is based on the idea that the joint motors are used to move the robot with the wheel motors assisting in the movement. During the movement the wheel of the moving leg is assisting the movement by rotating and the other wheels are kept unlocked.

The motion sequence is (for a visual representation, see appendix 7):

1. Initial state: Left legs move backward, right legs forward.

2. Left rear leg moves forward and right rear leg is allowed to move passively to a backward position.

3. Left front leg moves forward and right front leg is allowed to move passively to a backward position. Now the robot is in a configuration that is a mirror image of the initial state.

1 http://automation.tkk.fi/files/workpartner/newrolking.mpg

4. Right rear leg moves forward and left rear leg is allowed to move passively to a backward position.

5. Right front leg moves forward and left front leg is allowed to move passively to a backward position. The robot is now in the initial state.

Figure 17 shows the motion of the robot with this locomotion scheme. The simulator is currently slightly unideal for this locomotion as the rolling friction is incompletely

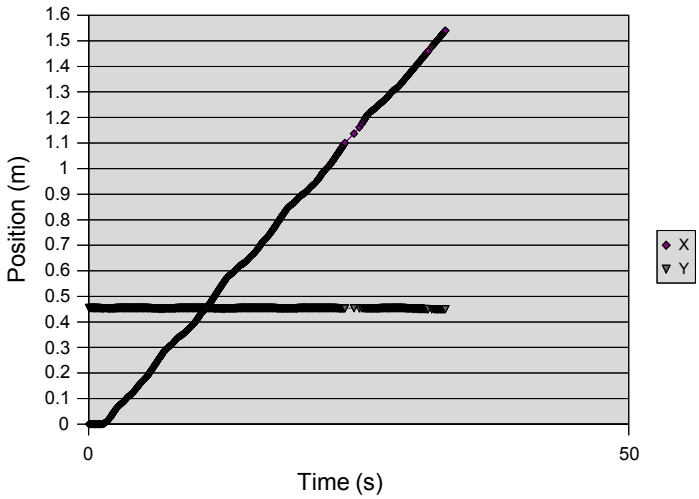

Figure 17: Robot CoM position with rolling walking.

modeled (see section 4.5.5). The velocity of the robot is 0.048 meters per second, which is slightly slower than in the previous simulation. The wheel forces are much more dynamic in this simulation, as would be expected. This is shown in Figure 18.

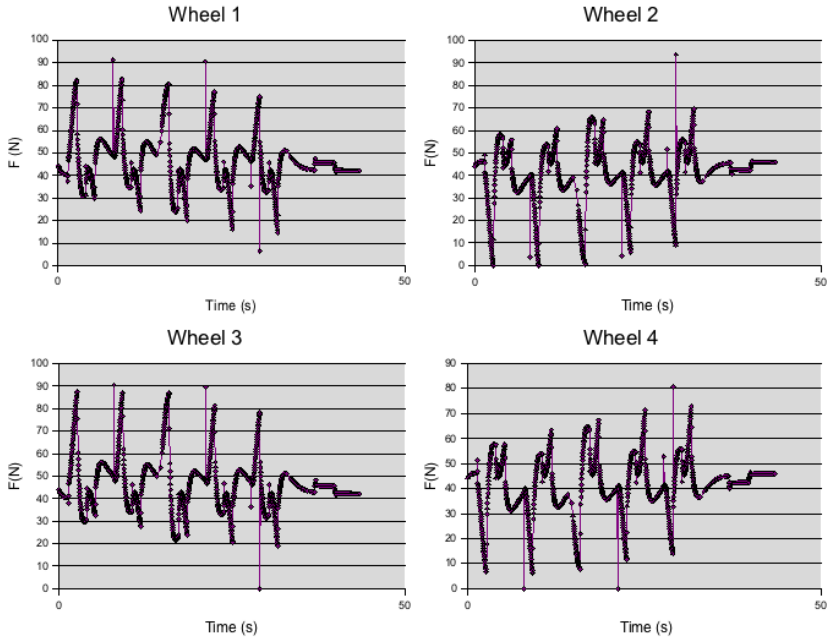

Figure 18: Wheel forces in rolling walking.

To analyze the forces in detail, another simulation was run with one motion sequence. The results are displayed in Figure 19.

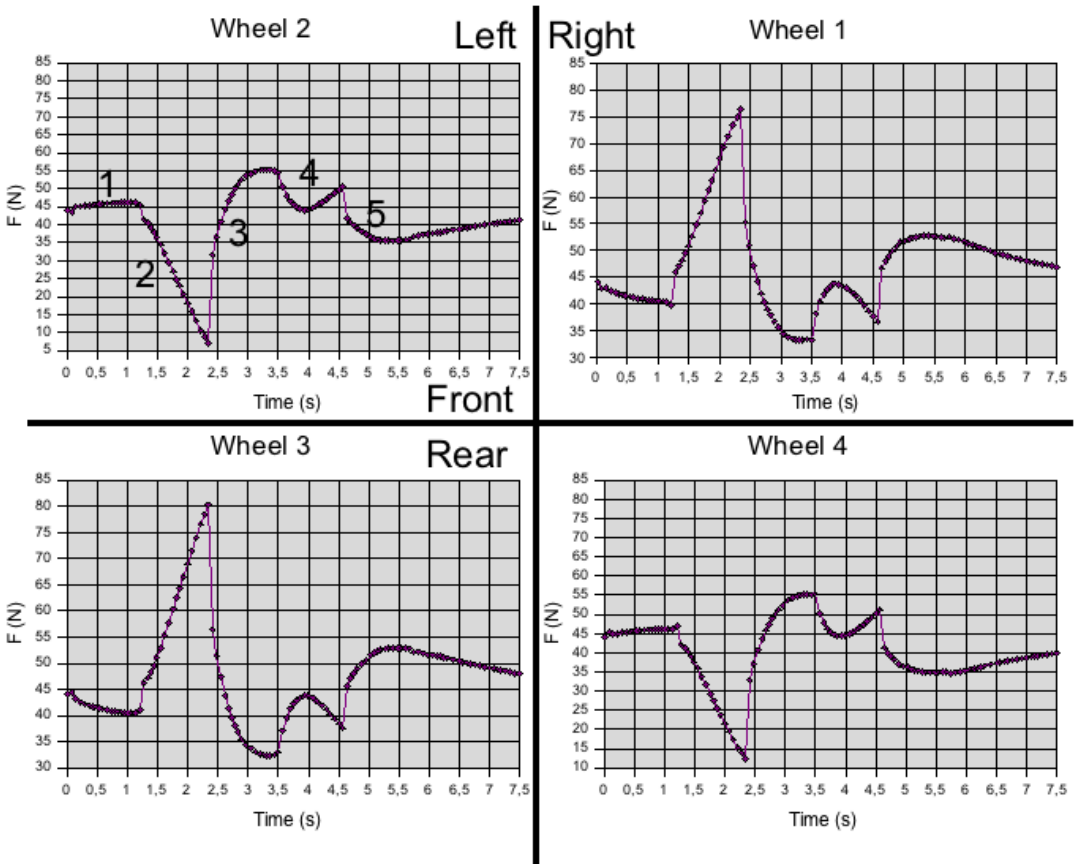

Figure 19: Wheel forces in one motion sequence.

The figure is organized so that the wheel numbers correspond to the correct locations on the robot when viewed from above. The numbers depict the different phases of the motion sequence. In (1) the robot is moving from the initial pose so that the wheels on the left side move backwards and the wheels on the right forwards so that the knee angle is 0.2 radians. After approximately 1.3 seconds the first rolling walking action begins (2), with the rear left wheel moving forward and rear right backwards. In this position, the robot is in a singular configuration, left legs are pointing towards each other under the robot and right legs point outwards. This causes the forward pointing legs one and three to bear most of the weight of the robot. In (3) the front left wheel moves forward while the front right moves backward. This causes the load to shift to wheels two and four.

In (4) the rear right leg moves forward, causing a small load variation. The final movement (5) puts the robot to a configuration in which the motion sequence could be started again. The wheel loads start to converge to be the same that in (1).

Rolling walking has been simulated before. Figure 20 from (P. Aarnio 2002, p.106) shows the wheel forces in a similar locomotion simulation to that presented here. The scale is 200 newtons and the weight of the robot is different. The simulation is similar to that described here. The movement that takes place in the

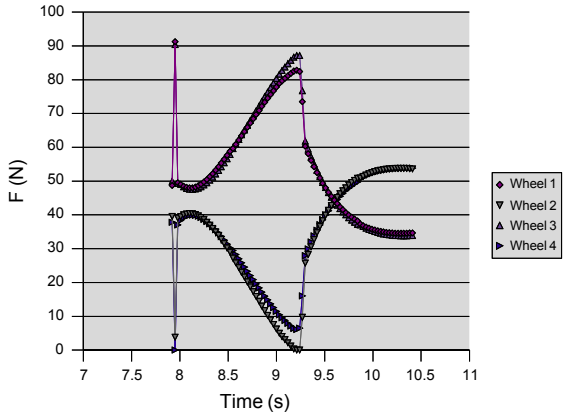

Figure 21: Wheel forces in SimPartner.

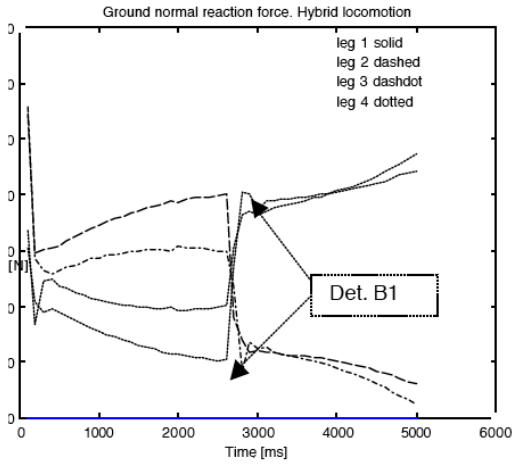

Figure 20: Wheel forces from an earlier simulation.

figure is when leg 4 is moving forward with the wheel rolling freely.

Figure 21 is an excerpt taken from the wheel force graphs that were presented earlier. The figures show similarities between the two simulations. Before the transient the forces increase in two of the wheels, while decreasing in the other two. After the transient, the SimPartner force graph converges, and the original graph diverges. Furthermore, the duration of the phenomenon is different in the simulations, approximately 3.5 seconds in the original simulation compared to two seconds in SimPartner. The leg numbers correspond to each other in the graphs.

The conclusion of the simulation is that the results are rather similar to the simulation performed earlier, but it cannot be definitely said that the results are in agreement. Therefore, it is necessary to compare the results to those obtained with the actual robot.

4.4 SimPartner validation

The validation was made by comparing SimPartner data to measurements performed using the actual robot. This was achieved by running a simulation similar to that described in Ilkka Leppänen's dissertation (Leppänen 2007). As stated before, no detailed technical drawing of the robot is available in its dissertation configuration. Therefore it is necessary to deduce the mass distribution of the robot from the test data. As a basis (Leppänen 2007, p.72) states that the whole robot has a mass of 270

kilograms and that each of the legs weighs 21 kilograms.

The test data used in the validation covers four test runs over variable terrain. The test runs are approximately 55 meters long and the durations were between 700 and 1400 seconds. Measurements were generally taken ten times per second.

4.4.1 Model weight distribution

After tidying the force measurements from invalid values (mostly zero, sensor not turned on) and calculating the averages we obtained the data presented in Table 1.

	Average Force					
Test Run	*Leg 1*	*Leg 2*	*Leg 3*	*Leg 4*	*Total*	*No. of measurements*
1	-591.55	-469.4	-508.82	-393.61	-1945.04	8617
2	-549.38	-432.48	-461.66	-359.36	-1784.12	12570
3	-598.81	-506.06	-476.76	-387.28	-1951.06	7412
4	-575.83	-484.52	-472.87	-392.34	-1903.9	6683
Average	**-578.9**	**-473.12**	**-480.03**	**-383.15**	**-1896.03**	
Std. Dev.	21.89	30.99	20.23	16.09	77.49	

Table 1: Force measurements from tests with the WorkPartner robot.

We can observe that there is a discrepancy between the reported mass of the robot and the force measurements. The total average force is approximately 1900 newtons, corresponding to a mass of 194 kilograms, the difference being 76 kilograms. The author of the study confirmed that this is due to the positioning of the force sensors. The wheel and parts of the leg come after the the force sensor in the kinematic chain and are thus not visible in the measurements. The mass of the parts not shown by the sensor is about 20 kilograms per leg, which causes the difference observed.

Also worth noting is the considerable weight difference between the various parts. Legs 2 and 3 have nearly equal average forces, whereas leg 1 carries a significantly larger amount of the weight than leg 4. The author of the study also confirmed this notion, explaining that when measured by scales, the robot is indeed tilted to the left and front. This means that leg 1 is located at the front left and leg 4 at the front right. It can also be deduced that leg 2 is located at the rear left and leg 3 at the rear right. The next step is thus to modify the SimPartner model so that the weights correspond to the measured averages plus the additional 196 newtons not shown by the measurements. The target values for SimPartner (using previously defined wheel numbers) are:

- Wheel 1 = 383.15 newtons + 196 newtons = 579.15 newtons.

- Wheel 2 = 578.9 newtons + 196 newtons = 774.9 newtons.
- Wheel 3 = 473.12 newtons + 196 newtons = 669.12 newtons.
- Wheel 4 = 480.03 newtons + 196 newtons = 676.03 newtons.

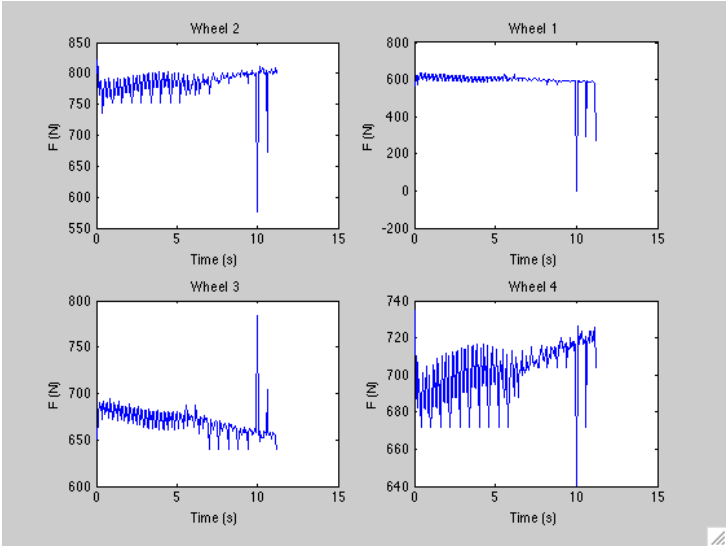

Illustration 25: Wheel forces when creating the validation model.

A model representing these values was created (see section 3.9.4). A ten second test run with the model yielded the following average wheel forces:

- Wheel 1 – 595.6401 newtons.
- Wheel 2 – 785.1331 newtons.
- Wheel 3 – 670.1860 newtons.
- Wheel 4 – 702.6742 newtons.

The measurements are noisy as the robot is no longer symmetrical but is rocking slightly from side to side in the simulation. The average error compared to the measurements done with the actual robots are in the range of 1 – 27 newtons, the maximum error being around 4 percent. This is acceptable for the validation.

4.4.2 Test terrain

(Leppänen 2007, p.78) presented the height profile of the terrain used in the test runs made with the actual robot. For our validation, we selected test run number four, in which the terrain used has the profile shown in figure 22.

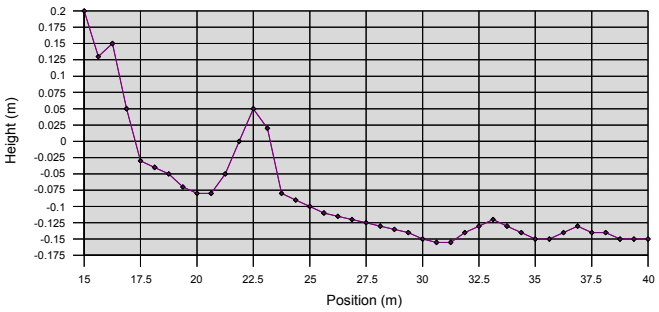

Figure 22: WorkPartner test terrain.

The test runs were made outside during the winter season. This means that the ground was snowy and uneven. The figure mentioned above only covers two dimensions so we can only model the elevation of the ground. Furthermore, the force measurements in the test data were rather noisy, so in order to discern features a 10-sample sliding median filter was used with every 10th data point plotted. The filtered test data is shown in Figure 23. The figure shows three distinct events that took place during the drive. The first one is the WorkPartner driving down the slope just before the 20 meter mark. The next one takes place just after the second bump. The third one cannot be completely explained by the height graph but the referenced paper notes that at this points some planks had been placed on the ground. This is the cause for the third event.

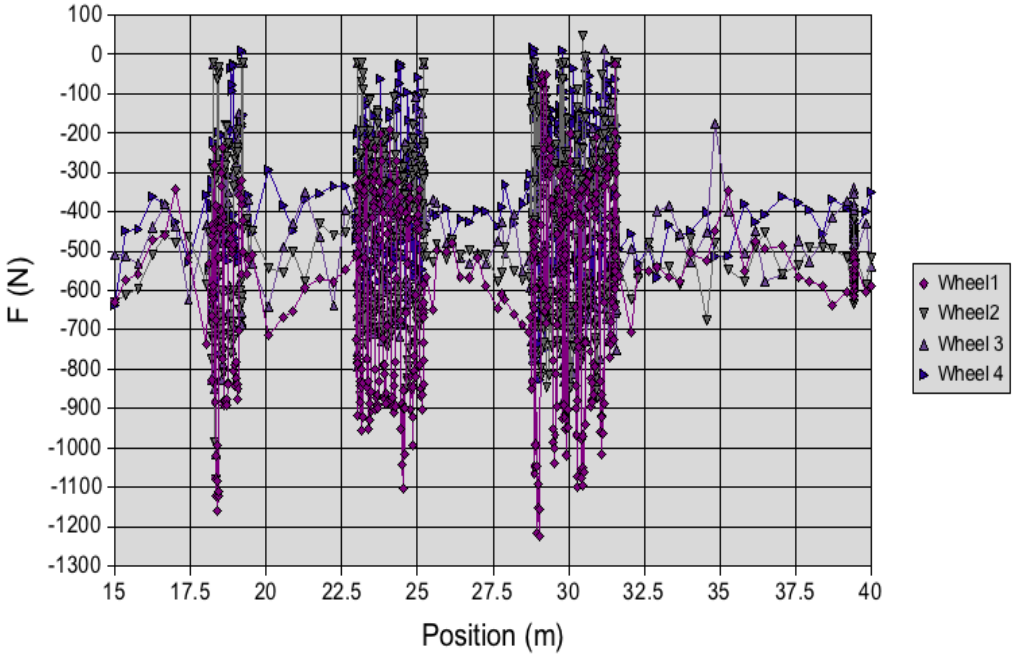

Figure 23: WorkPartner wheel forces during the test.

4.4.3 Test velocities

When the WorkPartner robot is traversing the test track, it is using active control for its wheels and legs for energy efficient locomotion. This behavior was not perfectly modeled in the client software as this is not within the scope of this work. Figure 24 shows the average velocity of the robot during the test drive. It was calculated from

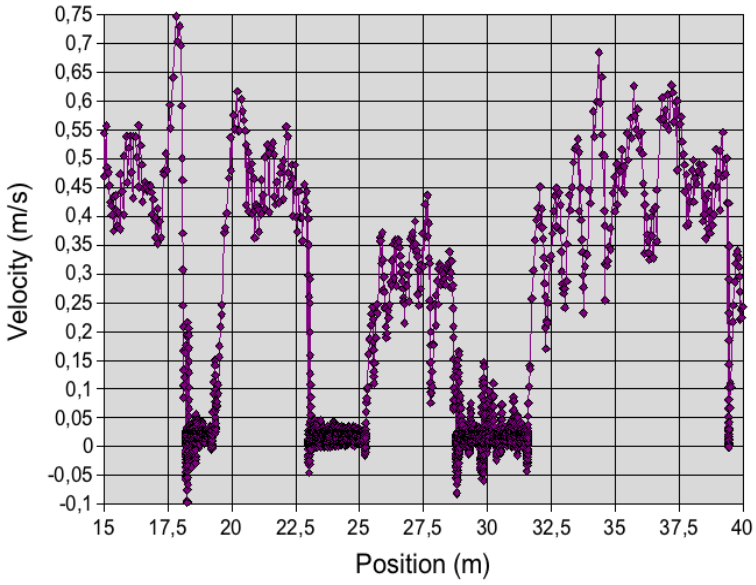

Figure 24: WorkPartner velocity during the test run.

the filtered odometry data and time stamps obtained by Ilkka Leppänen. The events corresponding to the high force value oscillations can clearly be seen. This data provides the basis for the velocities in the SimPartner simulation of the test drive.

4.4.4 Simulation velocities

After adjusting the mass distribution of the simulation model to match with the actual robot a heightfield was created that corresponds with the terrain used in the test run with the actual robot. The wheel velocities were then set to a constant velocity of 1.5 radians per second. The simulation ran for 104.75 seconds and the robot traversed a total of 34.5 meters. Figure 25 shows the velocity of the simulated robot, calculated as the average of the velocities of all body parts. It is significantly smoother than the test run made with the actual robot due to the fact that there is no active control for wheel velocities: the wheels try to maintain constant velocity regardless of the pose of the robot.

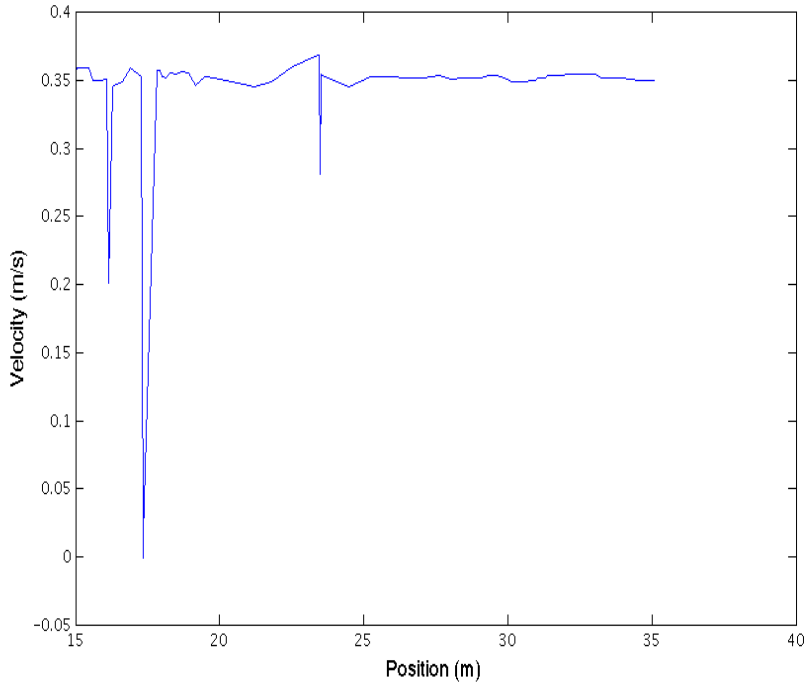

Figure 25: SimPartner velocity during the simulation.

The velocity of the robot slows down considerably in the same positions as the original test run. It can clearly be seen that these positions correspond to the locations where the terrain causes the robot to slow down. The simulation is thus similar to the actual test drive in this aspect.

4.4.5 Simulation wheel forces

The wheel forces from the simulation are shown in Figure 26. The magnitude of the forces is the same as in the test run with the actual robot, when taking into account the fact that the forces measured from the simulation incorporate approximately 200 newtons of leg and wheel weight. The number of measurements in the simulation is smaller but the spikes in the forces can clearly be seen in the same positions as in the actual test run. The small height variations in the 30 meter range cause small force spikes even though the planks mentioned before are not modeled. The data cannot be compared quantitatively due to the different control algorithms, but it can be said that the simulation resembles the actual test with an accuracy that is good enough to validate the simulator.

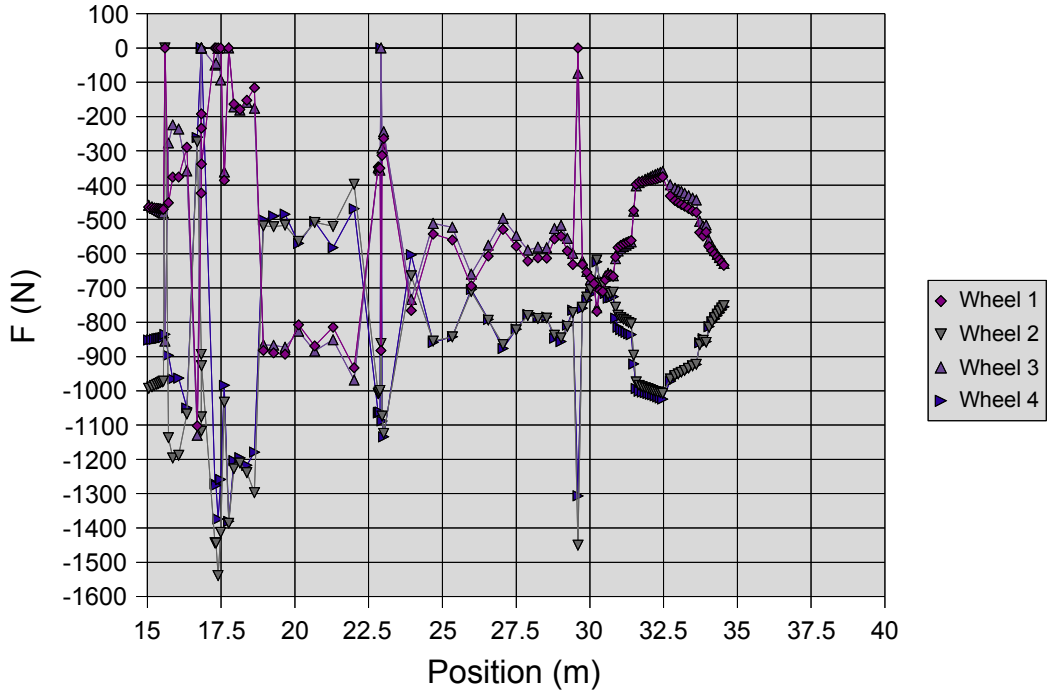

Figure 26: Wheel forces in the simulation.

4.5 Other Considerations

As stated before, ODE is not at full maturity at this point. There are still several inconsistencies and imperfections in the physics engine that must be taken into account. These imperfections also affect the SimPartner framework, and the user must understand and accept these limitations until ODE matures beyond these problems. Fortunately, depending on the type of simulation the user wants to achieve, some of these problems can be remedied by careful tuning of the simulation parameters.

4.5.1 Object-object penetration

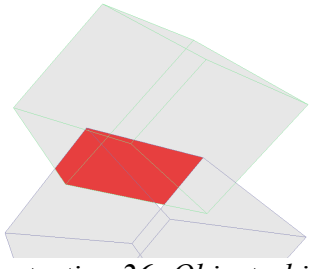

Illustration 26: Object-object penetration.

A phenomenon that can be observed, especially with box-shaped objects, is object-object penetration. This is basically a failure in the collision detection engine and it is

inherent to fixed time-step deterministic physics engines. This interpenetration does not occur when the collision is perpendicular and the contact surfaces are parallel. The chance of interpenetration increases when the contact angle becomes smaller and when the object size increases. This can be remedied by adjusting different simulation parameters which will cause jittering in stacked objects as they never come to rest because of the small amount of penetration and the resulting force that tries to keep the objects apart.

4.5.2 Object-ground penetration

Object-ground penetration is based on the same phenomenon as object-object penetration. The main difference is that in this case the other participant of the

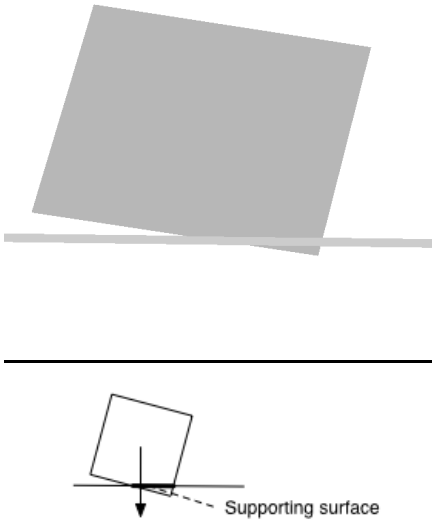

Illustration 27: Object-ground penetration.

interpenetration is the ground plane, which defines a non-usable half-space in the physics engine. An implication of this is that the object can come to rest in this position as the supporting surface is calculated from the penetration points. In this case the supporting surface is not formed by the corner points of the object but rather by the intersection points of the ground plane and the penetrated area of the object.

This phenomenon is also most frequently seen on box type objects and it can be remedied by adjusting the surface layer thickness parameter. Setting this higher will prevent penetration but might also cause jittering as objects are never fully at rest.

4.5.3 Physics engine numerical instabilities

Since there are several numerical problems that have to be solved for every simulation step, numerical problems may sometimes occur. One example of these is the ODE error message that appears to the console:

```
ODE Message 3: LCP internal error, s <= 0 (s=0.0000e+00).
```

LCP stands for linear complementarity problem, a linear algebra problem of finding two vectors that satisfy a certain set of equations based on a square matrix and a column vector. This problem is quite common in optimization, physics simulation and mathematical programming. ODE uses a method developed by (Cottle & Dantzig 1968) .

In ODE, the LCP error surfaces when objects collide, applying too much force for the solver. It does not cause the simulation to crash but may cause data inconsistencies or non-physical behavior.

4.5.4 Clock inaccuracy

The multitude of clock calls made by the SimPartner cause the PC clock to drift. This behavior will cause errors of several seconds per one minute of simulation time. This causes severe problems for analyzing the results, although the simulation itself runs well. This problem can be addressed by using NTP to actively keep the clock of the computer in correct time.

4.5.5 Rolling friction

Currently ODE does not implement rolling friction. This can be observed by the fact that a rolling object, such as a sphere, will not come to a complete stop without external forces. If a simulation is initiated including a sphere with a given initial velocity it will proceed with a constant velocity starting from time step 1.

4.5.6 ODE version dependency

Since the open dynamics engine is not yet at full maturity and the software is still being developed there is a tendency to use the most up-to-date version at all times. Herein, however, lies a danger. ODE version 0.9 (revision 1441) was used during the development of the framework. To achieve better performance the engine was updated to its latest version (revision 1468) in the final stages of the project. This caused all the simulations to become unstable. After this event ODE's official 0.9.0 release was used. This goes to show that the results can depend on the version of the physics library used.

4.6 SimPartner results analysis

4.6.1 Realization of identified good features

The conclusion of the state-of-the-art study was a list of design features that can be seen in high quality robotic simulator software. It is important to assess the quality of created software by comparing it against the feature list.

- **On target**

  This will ultimately be judged by the number of users this software will have. This feature was targeted to be achieved by the continuous dialogue between the author and the instructor of this thesis. Thus the user's viewpoint was constantly present in the software development, which is a key issue in all software projects.

- **Open Source**

  The developed software is completely open source, and it was even developed using open source tools. This was an unforeseen advantage as people in the open source community were very willing to hint and advise on the development as the end result would be open source. A problem in this development model is the licensing. Even now, the SimPartner framework contains components that have different licenses. Boost libraries are licensed under the Boost license, ODE is under BSD license, MySQL under GPL, etc. Thus developers need to keep track of the limitations of different licenses.

- **Modular**

  SimPartner fulfills this requirement well. Certain interconnections exist in the framework but if the user wants to substitute some parts of the simulator it is perfectly realizable. For example, visualization information sharing with the window manager is achieved purely by passing homogenous transformation matrices and object ID numbers through the main program. If the user wants to create his own window manager with some other rendering engine (such as OpenSceneGraph[1] or OGRE[2]), all relevant data can be found in the main program in a documented format. The same principle applies to database management and robot description modules.

- **Flexible**

  This feature is realized by the use of human-readable configuration files of the SimPartner framework. Granted, the framework is rather specialized and it is not foreseeable that it could be used for any applications outside the field

1 http://www.openscenegraph.org
2 http://www.ogre3d.org/

of robotics simulation but in its field it is rather versatile. Simulation parameters can easily be manipulated to achieve stable simulations. Robots, control code and environments can be easily modeled and modified. Furthermore, as the name of the file is always specified as a parameter, storing different configurations and switching between them is easy and fast.

- **Parametrized**

  As mentioned above, the operation of the framework can easily be manipulated. This was essential even in the validation phase when robot instabilities and performance issues could be addressed rapidly by altering simulation parameters. Without this feature, the completion of this thesis would have been impossible.
- **Platform independent**

  As the framework was originally programmed on Mac OS X there were some complications in porting the code to Ubuntu Linux. However, all the libraries used are portable and no OS-specific code is used in the framework. The software itself has been shown to operate in Linux and Mac operating systems, and will most probably work in Windows as well. This feature is therefore realized in the software.
- **Real-time ready**

  As discussed earlier, the framework can be used with the gamepad controller in real time. This can be extended to other real-time HMI controllers such as joysticks or the manipulator control vest.
- **Connected to actual hardware**

  This feature is not currently implemented. The command structure of the WorkPartner robot is currently too complex to be integrated to the framework. When the GIMNet framework is extended to the WorkPartner it would be viable to use the simulator to guide the robot. This would require GIMNet to be integrated to the SimPartner framework, which was not achieved within the scope of this project.
- **Verifiable**

  As this thesis presents, it is possible to validate the simulation model by comparing the simulated data to analytically calculated or otherwise inferred values. The values stored in the database are always in SI-units, which makes it easier to analyze the data and to create new validation scenarios.

4.6.2 Stability

The stability of the simulation framework remains an issue. At certain configurations, such as a case where all the joints are locked with a sufficient maximum force and a zero angle setting, the robot may become unstable. This means that the simulation literally “explodes” and crashes. This can be remedied with parameter manipulation or robot model modifications. There is no simple solution to these problems, which arise with all dynamic simulators. The ODE manual lists some issues that affect simulation stability:

Manual	Notes
Stiff springs / stiff forces are bad.	
Hard constraints are good.	
Dependence on integration timestep.	Timestep is very delicate, simulation can be stable with 0.01 s but unstable with 0.02 s.
Use powered joint, joint limits, built-in springs as much as possible, avoid explicit forces.	Setting forces manually to objects may cause undesired behavior in the integration step. This can in turn cause unstability.
Mass ratios - e.g. a whip. Joints that connect large and small masses together will be inherently susceptible to higher errors	Objects that are connected together should have about the same mass. Otherwise numerical errors in the integration may cause the simulation to become unstable.
If bodies move faster than is reasonable for the timestep	Velocities that are too high cause problems in collision detection because penetration occurs.
Inertias with long axes	Small and light objects are usually stable.
Increasing the global CFM will make the system more numerically robust and less susceptible to stability problems.	Increasing correction coefficients also cause the simulation to become less realistic. The tradeoff here is between stability and realism.
Redundant constraints (two or more constraints that ``try to do the same job'') will oppose each other and cause stability problems.	Redundant constraints causes singularities in the system matrix, creating unnatural forces that can be greater than the “real” forces affecting the bodies, thus causing aberrant behavior.

Table 2: Methods to increase simulation stability, adapted from (Smith n.d.) .

4.6.3 Performance

The SimPartner framework requires a lot of computing power, especially when database logging is used. If too many other applications are open, and/or the computer does not possess the required computing capabilities, the performance may not be good enough for real-time operation. Figures 27 and 28 show this performance degradation clearly. FPS rates shown are 5 sample moving averages.

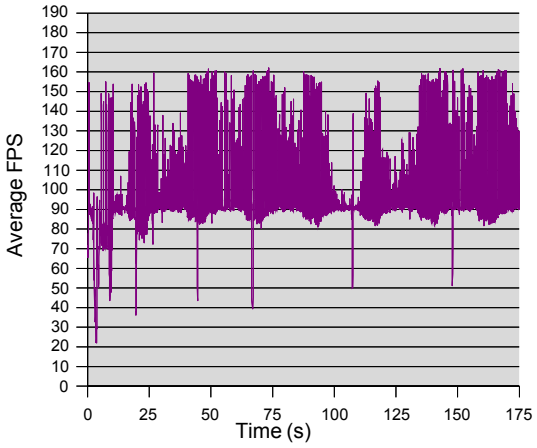

Figure 27: SimPartner performance without extra applications.

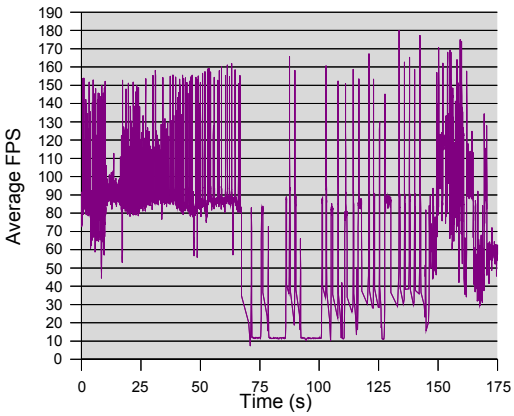

Figure 28: SimPartner performance with extra applications.

In Figure 27, the SimPartner framework and a sequencer client were run without any other software on the development computer (MacBook Core2Duo 2.0 GHz, 2 GB 667MHz DDR2 SDRAM, OSX 10.4.11). The performance is good and stays within real-time operating limits (FPS is greater than 20). In Figure 28, SimPartner is run in parallel with a C++ development engine, word processor, internet browser, e-mail client, etc. The real-time performance degradation is clear. The frame rate of SimPartner falls to the non-real-time domain and stays there, oscillating wildly. In this scenario the simulator is unusable.

As stated above, in some cases the simulation speed may not be adequate for real-time operation. There are many causes for this, including the fact that SimPartner uses the full accuracy stepping function provided by the ODE. The parameter file also allows the usage of the limited accuracy “quickStep”-function. However, this can be unstable when the system configuration is near-singular. This is the case when there is a multi-legged robot standing on the ground. Therefore, this option cannot be relied on.

4.6.4 Open source software development

An important part of every software project is to document and analyze what was good and what could have been done better to improve future work. Programming the SimPartner framework was an important learning experience with several points that can be considered to be lessons learned.

The whole project was built with and on top of open source applications. In application development, several tools such as Eclipse CDT integrated development environment, SVN source code control and versioning environment, and Doxygen documentation tool were used. The gain is that the operator and anyone contributing to the software are able to use the tools without charge. Even when these tools evolve, the basic functionality often stays the same, ensuring that future work on your project is easier.

Communication with the community is good. News groups, IRC channels and mailing lists offer great help when the development is in difficulty. People are generally eager to help, especially when writing open source software. There even exists a best practices manual on “How To Ask Questions The Smart Way” [1].

Web-based software for collaboration is effective. Related to the point above, web-based source code repositories help others to help you. When all the programs produced are available online, it is easy to refer to them for everyone to see. Furthermore, popular internet-based video sites provide the opportunity to record a screen capture when visual documentation is necessary.

1 http://www.catb.org/~esr/faqs/smart-questions.html

5 Conclusions

This thesis described the design and validation process of a dynamic mobile robot simulator. The work was based on the state-of-the-art study and literature review of existing simulators and their properties. General software engineering practices and standards of the open source community were also taken into consideration.

The most important validation test was to simulate a full test run done by the actual WorkPartner robot and to compare the results with the data from the actual robot. The test was a success as the results showed that the simulator was able to model the speed and force magnitudes of the real WorkPartner robot. It was also shown that the simulation errors can be identified and mathematically explained to a certain degree.

The software framework follows the identified common features of robotic simulators, furthermore the software has been verified to follow mathematical models and validated against actual test data. It can be thus said that the simulator is usable and trustworthy for control code development for mobile robot applications, as long as all the constraints of simulation in general and SimPartner in particular are taken into account.

It was also shown that real-time performance in dynamic robot simulations is achievable with consumer grade computers. SimPartner is a real-time simulator that can model complex mobile systems in a variable terrain with good enough real time characteristics. This, however, requires fine tuning the simulator parameters so that the performance level is acceptable.

5.1 Future work

The framework is in itself usable but there still exist several areas that could be improved.

- **Code clarification**

  The framework was programmed and designed solely by the author. This is generally not a very good way to design software. There are several redundant functions and data types in the software that could be simplified and clarified. This is also true on a general level, as overall complexity tends to increase when the program is, even partially, designed in parallel with the actual programming.

- **Geometry primitives**

  The library of possible object geometries is currently rather limited. This is a limitation that hinders robot control code development, as new environments are harder to build. However, adding new primitives is a rather straightforward task, explained in the SimPartner manual that accompanies the software.

- **GIMNet**

  The GIMNet framework was not used in the software in favor of a simple TCP/IP based communication scheme. Adding GIMNet to SimPartner would be a good improvement with added connectivity and usability. The framework itself is not sensitive to the communication method and since the base communication type is the same adding GIMNet should not be very difficult.

- **Robot editor**

  The SimPartner robot definition files tend to get rather lengthy when complex robots are designed. For example the fourth generation WorkPartner model was over 1400 lines long. A visual editor would therefore be a great aid when altering the designs or creating new robots.

- **Wheel-soil interaction**

  Wheel-soil interaction models do not currently exist in any major open source robotic simulators. Adding this feature to SimPartner was tested but rejected due to the fact that while the ODE library provides the necessary wheel forces they can only be obtained at the end of the simulation step. This means that all the forces affecting to the wheels can only be manipulated for the next simulation step. This means that the wheel-soil model would be always one step behind the actual simulation. If this issue is solved, adding the interaction model would be possible as the ODE allows the manipulation of all forces and velocities during the simulation.

6 References

Aarnio, P., 2002. *Simulation of a Hybrid Locomotion Robot Vehicle*, Licensiate Thesis PB2003-101519.

Aarnio, P., Koskinen, K., & Salmi, S., 2000. Simulation of the hybtor robot. In *Proceedings of the 3rd International Conference on Climbing and Walking Robots*. Madrid, Spain: Professional Engineering Publishing Ltd, p. 267-274.

Aarnio, P., Koskinen, K., & Ylönen, S., 2001. Using simulation during development of combined manipulator and hybrid locomotion platform. In *Proceedings of International Conference on Field and Service Robotics*., p. 287-294.

Bauer, R., Leung, W., & Barfoot, T., 2005. DEVELOPMENT OF A DYNAMIC SIMULATION TOOL FOR THE EXOMARS ROVER. *i-SAIRAS 2005'-The 8th International Symposium on Artificial Intelligence, Robotics and Automation in Space. Edited by B. Battrick. ESA SP-603. European Space Agency, 2005. Published on CDROM., p. 15.1*.

Biesiadecki, J., Jain, A., & James, M.L., 1997. Advanced Simulation Environment for Autonomous Spacecraft. *International Symposium on Artificial Intelligence, Robotics and Automation in Space*. Available at: http://dshell.jpl.nasa.gov/postscript/isairas97.ps.

Biesiadecki, J. et al., 1997. A reusable, real-time spacecraft dynamics simulator. In *Digital Avionics Systems Conference, 1997. 16th DASC., AIAA/IEEE*., p. 8.2-8-8.2-14 vol.2.

Buehler, M. et al., 1999. Stable open loop walking in quadruped robots with stick legs. In *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*., p. 2348-2353 vol.3.

Collett, T.H., MacDonald, B.A., & Gerkey, B.P., 2005. Player 2.0: Toward a Practical Robot Programming Framework. *Proceedings of the Australasian Conference on Robotics and Automation (ACRA 2005)*.

Cottle, R.W. & Dantzig, G.B., 1968. Complementary pivot theory of mathematical programming. *Linear Algebra and Its Applications*, 1, p.103-125.

Curto, P.A., 1997 NASA Software of the Year Competition. Available at: http://icb.nasa.gov/swy97win.html [Accessed February 13, 2008].

Erleben, K., 2004. Ph.D. Thesis TStable, Robust, and Versatile Multibody Dynamics Animation.

Fraczek, J. & Morecki, A., 1999. Modelling of contact in walking machines. In *IEEE SMC '99 Conference Proceedings.* IEEE SMC, p. 948 - 952.

Gerkey, B., Vaughan, R.T., & Howard, A., 2003. The player/stage project: Tools for multi-robot and distributed sensor systems. *Proceedings of the 11th International Conference on Advanced Robotics*, p.317–323.

Jain, A. et al., 2004. Recent Developments in the ROAMS Planetary Rover Simulation Environment. *2004 IEEE Aerospace Conference, Big Sky, MT, March 20, 2004*.

Jain, A. et al., 2003. ROAMS: planetary surface rover simulation environment. In *i-SAIRAS 2003*.

Jain, A., Jet Propulsion Laboratory: DARTS Home Page. Available at: http://dshell.jpl.nasa.gov/DARTS/index.php [Accessed February 13, 2008].

Koenig, N. & Howard, A., 2004. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *Intelligent Robots and Systems, 2004. (IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on.*, p. 2149-2154 vol.3.

Kovo, H., 1999. Tietokonesimulointi (AS-74.1101 Computer simulation) - Lecture notes, Helsinki University of Technology.

Leonard, J. et al., 2007. Team MIT Urban Challenge Technical Report.

Leppänen, I., 2007. Automatic locomotion mode control of wheel-legged robots, Dissertation, Helsinki University of Technology.

Michaud, S. et al., 2004. RCET: ROVER CHASSIS EVALUATION TOOLS. In *Proceeding of the 8th ESA Workshop on Advanced Space Technologies for Robotics and Automation*. Noordwijk.

Michaud, S. et al., ROVER CHASSIS EVALUATIONAND DESIGN OPTIMISATION USING THE RCET. *The 9th ESA Workshop on Advanced Space Technologies for Robotics and Automation (ASTRA'06)*.

Michel, O., 2004. WebotsTM: Professional Mobile Robot Simulation. *cs/0412052*. Available at: http://arxiv.org/abs/cs/0412052 [Accessed January 3, 2008].

NASA, 2007. Mars Global Surveyor (MGS) Spacecraft Loss of Contact. Available at: http://www.nasa.gov/pdf/174244main_mgs_white_paper_20070413.pdf [Accessed January 24, 2008].

Patel et al., 2004. Rover Mobility Performance Evaluation Tool (RMPET): A Systematic Tool for Rover Chassis Evaluation via Application of Bekker Theory. In *Proceeding of the 8th ESA Workshop on Advanced Space Technologies for Robotics and Automation*. Noordwijk.

Poulakis, P. & Joudrier, L., 2006. Port-based Modeling and Simulation of Planetary Rover Locomotion on Rough Terrain.

Smith, R., *ODE manual*, Available at: http://www.ode.org/ode-latest-userguide.html [Accessed March 9, 2008].

Vaughan, R., Gerkey, B., & Howard, A., 2003. On device abstractions for portable, reusable robot code. In *Intelligent Robots and Systems, 2003. (IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on.*, p. 2421-2427 vol.3.

Yen, J., Jain, A., & Balaram, J., 1999. ROAMS: Rover Analysis Modeling and Simulation. In *Fifth International Symposium on Artificial Intelligence, Robotics and Automation in Space*. SP. Noordwijk: ESA, p. 249.

7 Appendices

7.1 Appendix 1 – Physics engines

	SimMechanics	Vortex	20sim	ODE
Physics modeling	Unknown	Unknown	Port-based	Hybrid (Constraint-based + Penalty)
Physical Domains	Kinematics, Mechanics	Dynamics, Kinematics, mechanics	Electrical, mechanical, hydraulic, thermal, hybrid	Dynamics, Kinematics, mechanics
Accuracy	Adjustable with a tradeoff of speed	Real-time fidelity	Adjustable with a tradeoff of speed	"not for quantitative engineering"
Body types	Rigid (flexible extension available)	Rigid	Rigid	Rigid
Real-time	Code generation	Yes	Code generation	Yes
Integration	All possibilities Matlab offers	First order	One-step, multi-step or multi-order	First order Fixed-step Euler.
Objects	Function blocks (similar to Simulink)	Geometric primitives, trimeshes, planes and composites	Equations, state space descriptions, bond graphs, block diagrams, iconic diagrams	Several geometric primitives + trimeshes and planes
Collision detection	None, website reference	Contact Points (position, normal, depth)	None	Contact Points (position, normal, depth)
Surface properties	None (ibid)	"Detailed friction models"	None	Coulomb friction
License, Platform	Commercial, Matlab extension	Commercial, Windows and Linux	Commercial, Windows	BSD, platform independent

Table 3: Comparison of different physics engines.

7.2 Appendix 2 - UML sketch

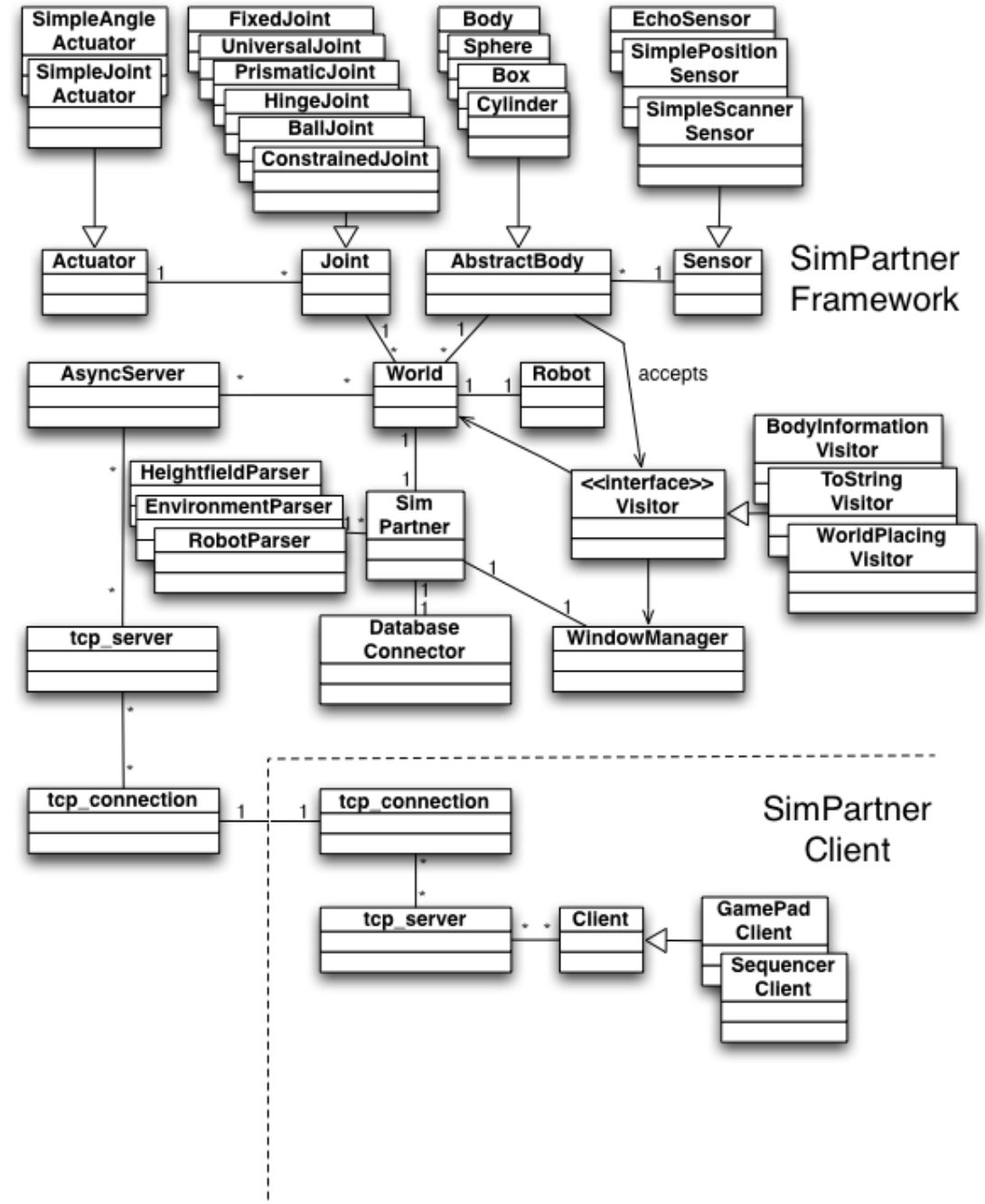

Illustration 28: SimPartner software structure.

7.3 Appendix 3 - Software versions

Library name	Abbreviation	Version number	URL	Notes
Open Dynamics Engine	ODE	0.9	http://sourceforge.net/project/showfiles.php?group_id=24884&package_id=18585&release_id=542627	
Boost		1.34.1	http://www.boost.org/users/download/	
Boost		1.35	http://www.boost.org/users/download/	For Boost.ASIO.
Simple DirectMedia Layer	SDL	1.2	http://www.libsdl.org/download-1.2.php	Including SDL-image and SDL-ttf
libxml++		2.6	http://libxmlplusplus.sourceforge.net/	
libmysql++		2.3.2		Apple version number indicated.
OpenGL Utility Toolkit	GLUT	3	http://www.opengl.org/resources/libraries/glut/	

Table 4: Different software libraries used.

7.4 Appendix 4 - Database structure

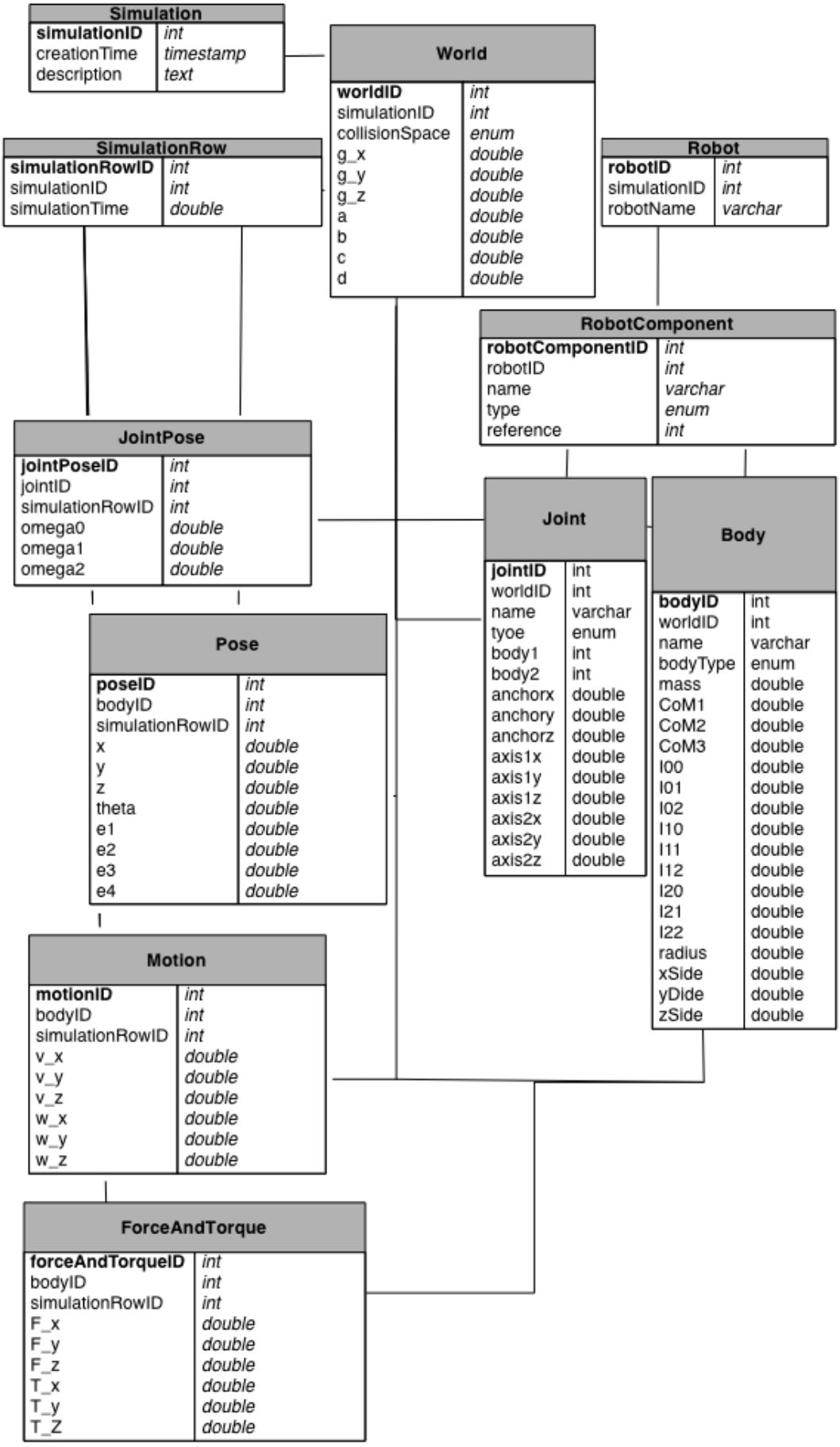

Illustration 29: Database structure

7.5 Appendix 5 - Selected SQL queries

```
SELECT s.simulationtime,p.bodyID,p.x,p.z
FROM pose p, simulationrow s WHERE bodyID IN(
      SELECT reference
      FROM RobotComponent
      WHERE robotID IN(
            SELECT robotID
            FROM Robot
            WHERE simulationID=4045)
      AND type='BODY'
      AND name LIKE 'Calf%')
AND p.simulationRowID = s.simulationRowID;
```

Query for getting the leg positions of the robot in a given simulation.

```
SELECT s.simulationTime,p.anchorx,p.anchory,p.anchorz
FROM jointPose p, simulationrow s
WHERE p.jointID IN(
      SELECT reference
      FROM robotComponent
      WHERE robotID IN(
            SELECT robotid
            FROM Robot
            WHERE Simulationid = 4282)
      AND name ='Turner')
AND p.simulationRowID = s.simulationRowID;
```

Query for getting the turner joint position in a given simulation.

```
SELECT s.simulationTime, sum(f.F_y)
FROM ForceAndTorque f, simulationrow s
WHERE bodyID IN(
      SELECT reference
      FROM RobotComponent
      WHERE robotID IN(
            SELECT robotID
            FROM Robot
            WHERE simulationID=4939)
      AND type='BODY'
      AND name like 'Wheel%')
AND s.simulationrowID = f.simulationrowID
GROUP BY simulationTime
ORDER BY simulationTime ASC;
```

Query for getting the accumulated forces affecting the wheels in a given simulation

```
SELECT SimulationRow.simulationTime,avg(Pose.x),avg(Pose.y)
FROM Pose, SimulationRow
WHERE Pose.simulationRowID = simulationRow.simulationrowID
GROUP BY Pose.simulationRowID;
```

Query for getting the location of the center of mass of the robot.

7.6 Appendix 6 - Distance sensor measurements

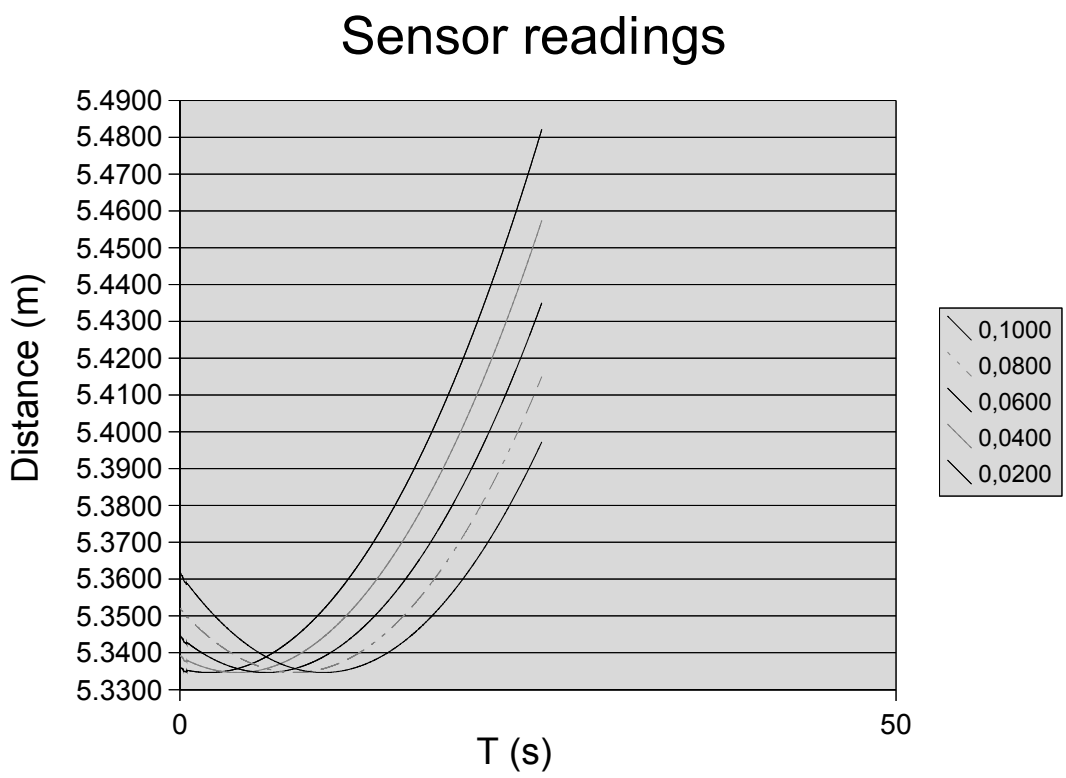

Figure 29: Sensor readings opposite the turning direction (legend in radians).

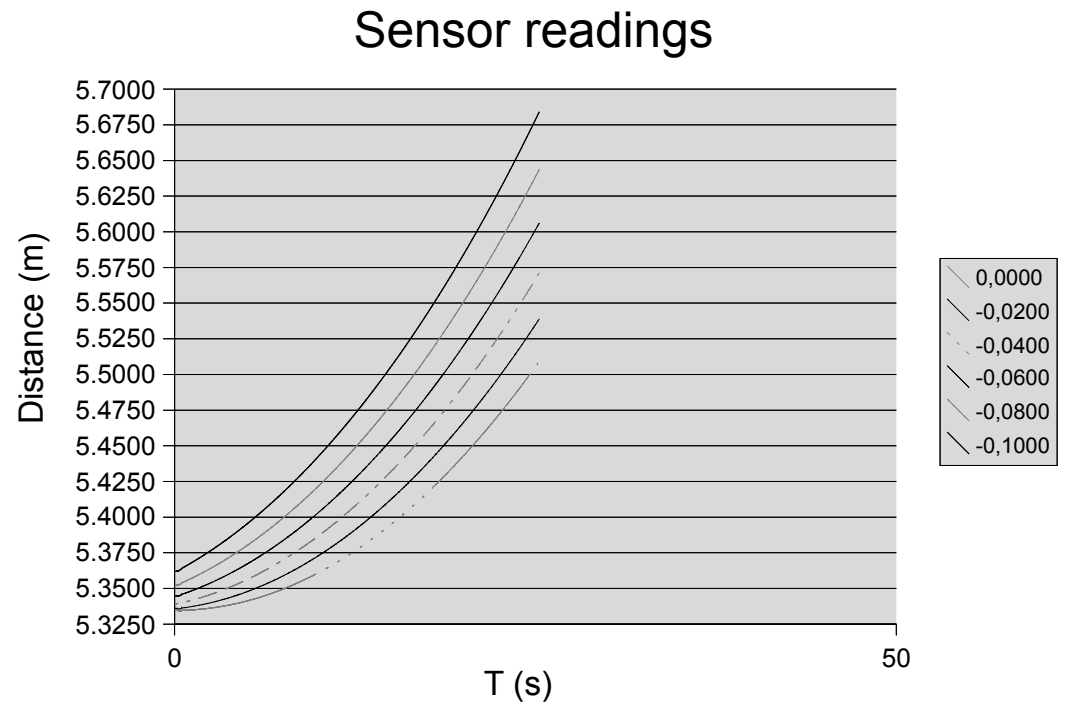

Figure 30: Sensor readings in the turning direction(legend in radians).

7.7 Appendix 7 - Motion sequence in rolling walking

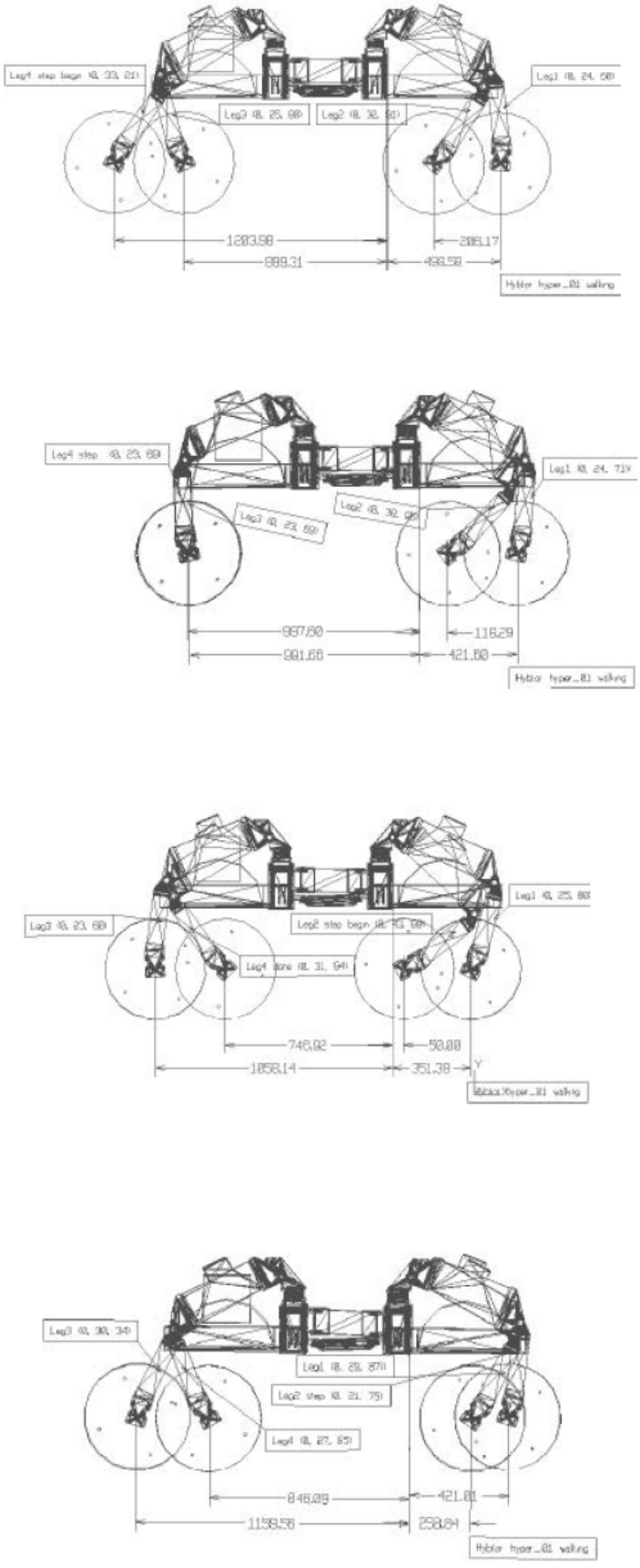

Illustration 30: Rolling Walking leg movements, From (P. Aarnio 2002)