Evaluation Of Open Dynamics Engine Software

R.C.Kooijman

D&C 2010.022

Traineeship report

Coach(es): dr.ir.D.Kostic

Supervisor: prof.dr.ir.H.Nijmeijer

Eindhoven University of Technology
Department of Mechanical Engineering
Dynamics & Control

Eindhoven, March, 2010

Contents

Chapter 1

Introduction and Overview

1.1 Introduction

Numerous Robocup teams have been using a particular software tool, namely "Open Dynamics Engine, (ODE or OpenDE in short), to produce simulations of humanoid robots. This tool allows a user to preform rigid body dynamics simulations by simply providing the system's parameters and initial states, and thus not have to worry about the equations of motion involved.

The Dynamics and Control department at the TU/e has no working experience with this tool and therefore has no real knowledge of its exact capabilities and subsequent limitations. This report takes an investigative look into the use of OpenDE, particulary focusing on its use in the field of robotics. The software's capabilities have been evaluated in comparison to those of Matlab, which is currently the most popular tool for dynamic simulations.

1.2 Goals and Contributions

The goals of this internship is to create a better understanding of the OpenDE tool. The capabilities of OpenDE are to be compared to Matlab by means of comparing equivalent models and simulations. In particular, this study aims to evaluates the software's performance with respect to the following two criteria's:

- Simulation error. As OpenDE implements a certain methodes of numerical integration, it is important to find out how accurately these are able to replicate simulations made in Maltab.
- User friendliness. Determine how quickly/easily OpenDE is to learn and then use relative to Matlab from the perspective of an mechanical engineering student.

Due to a combination of poor IT skills and poor documentation available, a lot of time and energy was spent just getting the software up and running. To help facilitate future work, comprehensive installation and implementation user guides have been constructed and added to the appendix. These were a written to accommodate for the needs of even the most computer illiterate mechanical engineer around.

Initially, this study was to be based on a implementation of Pieter van Zutven's model of the humanoid robot, TUlip created in Matlab[?]. This soon proved to be an overly complex task for the time frame of a short internship. The workload (and model) were simplified to an implementation of the anthropomorphic walker derived in [?] and compared with the matlab model and simulation produced by D.Karssen and M.Wisse. This model was to a large extent implemented revealing a lot about OpenDE's ease of use. Being pressed for time and needing a completed model to evaluate accuracies of solutions, two simple models were considered, namely a double pendulum and a bouncing ball.

Keeping in mind the goals, the main contributions of this thesis can be summarized as:

- 'How to' guides are produced for installation and use.
- Basic modeling and simulation principles employed by OpenDE are analyzed in detail. Although not explicitly part of the project goals, is was studied in depth to compensate for shortcomings in the openDE userguide.
- Advantages and disadvantages are discussed, as well as the pitfalls an engineering student is likely to come across when using OpenDE.

Chapter 2

Introduction to OpenDE

2.1 Introduction

This chapter will gives an overview of basic structure and fundamental principles behind OpenDE. Brief instructions on how to install OpenDE will also be given, along with a discussion on some of the problems which had occurred during that phase.
"Physics engine", "Dynamics engine","Physics SDK (Software Development Kit)", " library for simulating rigid body dynamics" are some of the descriptions one could come across when reading about OpenDE, meaning more or less the same thing. To prevent confusion, these and other important words and descriptions will first be looked into.

2.2 Definitions

There are a view definitions important enough to be mentioned explicitly. As this report is a discussion on modeling Rigid Body Dynamics, we shall look this definition[1] first.

Rigid Bodies Dynamics

The study of the motion of a rigid body under the influence of forces. A rigid body is a system of particles whose distances from one another are fixed. The general motion of a rigid body consists of a combination of translations (parallel motion of all particles in the body) and rotations (circular motion of all particles in the body about an axis). Its equations of motion can be derived from the equations of motion of its constituent particles.

The first line is the definition of 'dynamics'. The description of 'rigid bodies' translates to 'undefendable bodies'. A rigid body model is a simplification of a real body which has finite stiffens between all the the particles, based on the assumption that the deformation is negligible. Weather or not this assumption/simplification can be made depends on three things, namely:

- The body (dimension of a body, material stiffness.)
- Simulation parameters (excitation force on the body must be below plastic deformation limit, excitation frequency needs to be significantly lower than the body's first flexible mode...)
- Degree of model accuracy required.

The soft, or deformable bodies, for which this assumption does not hold, are modeled with finite element models and will not be treated further.

[1] Definitions were taken taken form internet: www.thefreedictionary.com. These may not be the official dictionary definitions, but good enough for this project

As mentioned, I came across various descriptions of OpenDE and unfamiliar terminology. To spare any confusion this may cause, a list of common terms and there definitions has been formulated:

Software development kit

Software development kit (SDK or "devkit") is typically a set of development tools that allows a software engineer to create applications for a certain software package, software framework, hardware platform, computer system, video game console, operating system, or similar platform.

Library

A library is a collection of subroutines or classes used to develop software. Libraries contain code and data that provide services to independent programs. This allows code and data to be shared and changed in a modular fashion.

Physics Engine

A computer program that simulates Newtonian[2] *physics models, using variables such as mass, velocity, friction and wind resistance.*
The term 'game engine' has occasionally been used to describe OpenDE but this subjects the inclusion of extra utilities (other than dynamics simulation) to facilitate game development.

2.3 Simulation Environment

OpenDE is often revered to being exclusively responsible for providing the library functionality to preform rigid body simulations. In fact, two additional software modules or *engines* are needed to preform meaningful simulations, namely a *collision detection* and *rendering* engine. As explained in the setup guide, OpenDE comes with two collision detection engines (OPCODE and GIMPACT), to choose from. The rendering engine, OpenGL, used by the OpenDE demos, is responsible for process of generating an image from a model.
These modules and how they interact during a of a rigid body simulation are shown in figure 2.1.

2.3.1 Objet Oriented Approach

Object Orientated programming, or OOP, is a style of programming where the source code is written in a structured form. This stuctured form can be seen as sperate blocks of code, each providing independent functionality. OOP is good practice, keeping the code easily readable, extendable and/or reusable. Although OOP can be implemented in many languages including Matlab, the C++ language syntax is especially suited for this as it was originally designed for just that purpose.

For Real time physics engines like those used for gaming, object orientated programming is needed in order to be able to dynamically change the model based on user input. This online simulation really sets the physics engine apart from Matlab simulations, which do not allow any changes to the simulation environment after the simulation has started.

2.3.2 Users

In order to help understand software and clear up confusion, its help to view it from the perspective of the users involved. There are 3 types of users involved with OpenDE, namely:

[2]Here *'Newtonian'* is describing non-relativistic classical momentum. This refers to the model's correctness depends on the velocity being much lower than the speed of light. It shouldn't be confused with *'Newtonian Mechanics'* and by extension the excluding *Lagrangian Mechanics*.

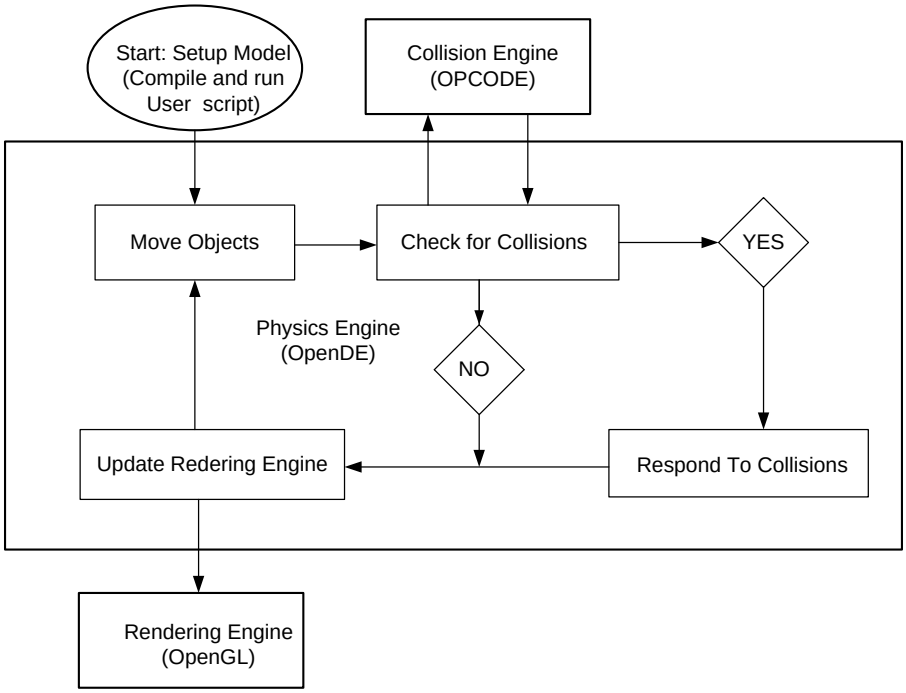


Figure 2.1: Flow diagram of a dynamics simulation showing the interaction of various software modules responsible for dynamics, collision detection and rendering

- The stimulater or a gamer.
- Application Designer. The person making the rigid body models assembled from functional modules, the so called components.
- Component Designers. Initially this was Russel Smith [**?**] , but now these are the volunteers developing OpenDE. They create the standard functional modules for the application designer.

2.3.3 Wrapper

The functional modules provided by the OpenDE library is relatively low level software. The application designer not only has to implement the parameters of a model but also the framework involving the connection of the all various modules. There are physics engine wrapper available which takes care of this framework and usually provides a form of graphics user interface, GUI, for the application designer.

A number of wrappers have been looked into but only briefly. The possibility of using a wrapper was discovered quite late in the project at which point it seemed more logical to spend the time carry on without the use of a wrapper. This decision was also based on the assumption that by using a wrapper would also limit the understanding of OpenDE. In retrospect, this was a bad decision as a lot of time was lost debugging code. Further more, in depth understanding of the OpenDE was actually outside the scope of this project however this became a necessity in order to understand how to implement various options correctly.

There were numerous OpenDE wrappers available including a number that were peruse build for robotic simulations. Many of these, as apposed to OpenDE, are no longer under development and as such are assumed to be out dated. WebBots appears to be the most in use today. This wrapper offers extra functionality such as prebuild sensor models, however it has to large disadvantages, namely it

is expensive and not open source. The Physics Abstraction Layer, PAL, wrapper was unique in that it supports a large range of popular physics engines. It was also open source and free and therefore perhaps a useful tool to test the preference of different physics engines in future work.

2.3.4 Open Source

Open source means that the source code is freely available to the public, *at least for viewing purposes*. The exact terms and conditions of copying, altering and redistributing the code are covered by a particular licence, the name of which is usually included as a header in the code. Generally this is covered by the General Public Licence (GPL), which allows for alteration of the code but in doing so keep the names of previous authors and licence.
OpenDE is open source which some major advantage over its (usually commercial) closed source counterpart, for such as PhysicsX

- If there is a problem the origin of this problem can be more easily traced and solved.
- By being able to look into the code can help with understanding how to use the applications.
- As there are many people working on the development of the software, it is easy receive technical support from newsgroups.

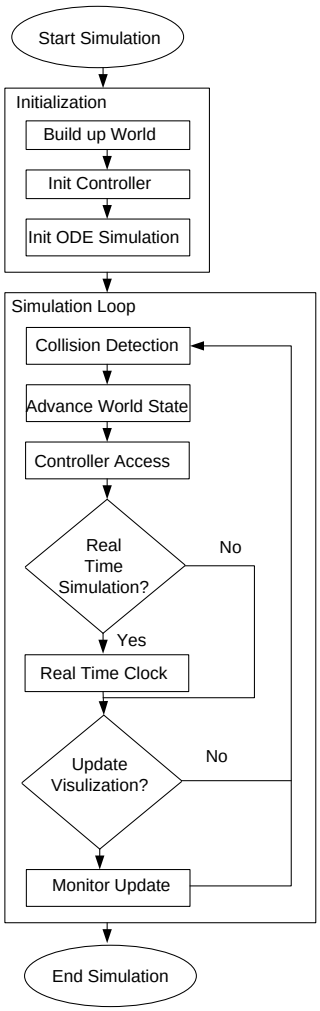
However, open source software generally also has some mager disadvantages:

- The software has not been fully tested, so bugs are usually more frequent.
- Poor documentation:
  - Often outdated. This is because a new version of the software is made public as soon as it something has been changed. In contrast, commercial software often make a new version at regular intervals, (often annually), implementing a range of new functionality in one go.
  - Just lacking in general. Open software programs written by enthusiasts are less enthusiastic about writing the user guide.

OpenDE suffers for this last issue, failing to explain a number of issues satisfactorily, which led to doubts on just how to implement certain paraments. These will be discussed when they further on.

2.4 Simulation Architecture

An overview of the simulation architecture, figure 2.2, reveals the various software components which the application designer, as described in section 2.3.2 needs to link together.
OpenDE has been written in an Object Orientated fashion. This means that the software can be split up in a number of modules. Separating the code into modules like this keeps the code readable and makes it easier to upgrade the software.



1. Create a dynamics world

2. Create bodies in the dynamics world

3. Set the state (position and velocities) of all bodies

4. Create the joints (constraints) that connect bodies.

5. Create a collision world and collision geometry objects.

6. While (time < timemax)
   (a) Apply forces to the bodies as necessary
   (b) Call collision detection.
   (c) Create a contact joint for every collision point. (Add them to the contact joint group)
   (d) Take a forward simulation step.
   (e) Remove all joints in the contact joint group.
   (f) Advance the time: time = t+t_step

7. End

Figure 2.2: Flow Diagram Of Simulation Architecture

Chapter 3

Closer Look at Open Dynamics Engine

3.1 OpenDE In Robotics

The Open Dynamics Engine is a free, industrial quality library for simulating articulated rigid body dynamics. It is known for its stability, not for its accuracy. This, however does not exclude it form being used for academic prepossess. ODE has been shown to be well suited for use in a number of key fields in robotics. Yriv Bachar, for example wrote his master thesis [**?**] on implementing ODE as a teaching tool to learn about biped motion control. There he talks about using ODE wrapper, namely Webbots, which is a user friendly robot programming Graphical User Interface, GUI . Webbots is commercial software but there are similar open source robot-GUI interfaces applications available, such as openSim, XPERSim or robosim.

D.Hein wrote his master thesis on the implementation of a learning control algorithm witch was first 'taught' in an ODE simulation [**?**]. This was only one of many papers available which used ODE simulations as a teaching ground for this kind of controller.

M.Friedmann and K.Petersen wrote a paper [**?**] on the use of ODE to produce real time simulation environments that allow for software-in-the-loop (SIL) experiments. This is especially promising for the TUlip robots they make use of the same controle framework, called roboframe, shown in figure 3.1.

3.2 ODE Features

The ODE has the following basic features:

- Articulated rigid-body structures
- Hard contacts (Non-penetrating constraints)
- Collision detection
- Joint types such as: ball-and-socket, hinge, slider, angular Motor, etc
- Collision primitives: sphere, box, cylinder, plane, ray, triangular mesh
- Motion equations: Lagrange multiplier velocity based on Trinkle/Stewart and Anitescu/Potra
- Choice of integrator methods (normal or quickstep)

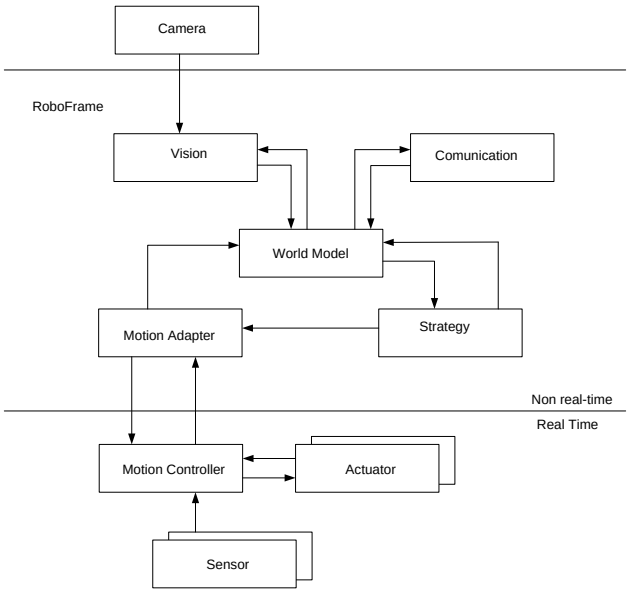


Figure 3.1: Overview of the RoboFrame architecture

- Contact and friction using Dantzig LCP solver

ODE simulation are usually run with the addition of two extra libraries, namely a collision library which is called by ODE itself, along with a separate 3D rendering library which outputs the results on the screen.

3.3 Coulomb Friction model

ODE has 2 fiction modes based on the equation

$$|f_T| \leq mu|f_N| \tag{3.1}$$

Where

- f_T =Tangential friction force
- mu = Friction coefficient
- f_N = Normal and tangential force vectors

ODE's implementation of this rule is slightly different for reasons of computational efficiency. The programmer can chose out of the following 2 implementations:

- The meaning of mu is changed so that it specifies the maximum friction (tangential) force that can be present at a contact, in either of the tangential friction directions. This is rather non physical because it is independent of the normal force, but it can be useful and it is the computationally cheapest option. Note that in this case mu is a force limit an must be chosen appropriate to the simulation.
- The friction cone is approximated by a friction pyramid aligned with the first and second friction directions. A further approximation is made: first ODE computes the normal forces assuming that all the contacts are frictionless. Then it computes the maximum limits fm for the friction (tangential) forces from $f_{max} = mu|f_N|$ and then proceeds to solve for the entire system with

these fixed limits (in a manner similar to approximation 1 above). This differs from a true friction pyramid in that the "effective" mu is not quite fixed. This approximation is easier to use as mu is a unit-less ratio the same as the normal Coloumb friction coefficient, and thus can be set to a constant value around 1.0 without regard for the specific simulation.

3.4 EPR and CFM Parameters

ODE makes use of parameters, EPR and CFM that together apply external forces on constraints. This is done for 2 reasons:

- If the user sets the position/orientation of one body without correctly setting the position/orientation of the other body. (as illustrated in 3.2)
- During the simulation, errors can creep in that result in the bodies drifting away from their required positions.

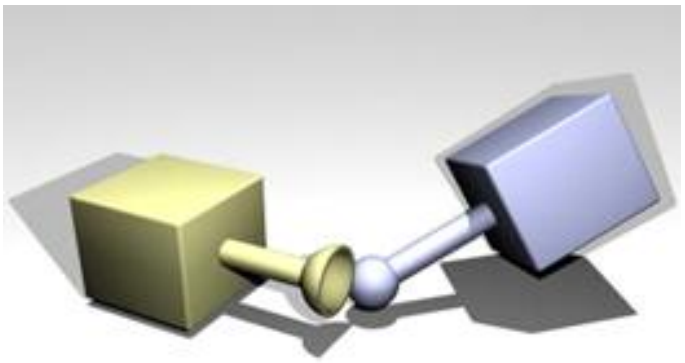


Figure 3.2: An example of error in a ball and socket joint

3.4.1 Soft constraint and constraint force mixing (CFM)

Most constraints are by nature "hard". This means that the constraints represent conditions that are never violated. For example, the ball must always be in the socket, and the two parts of the hinge must always be lined up. In practice constraints can be violated by unintentional introduction of errors into the system, but the error reduction parameter can be set to correct these errors.
Traditionally the constraint equation for every joint has the form

$$J\frac{dx}{dt} = c \tag{3.2}$$

where

- $J =$ Jacobian matrix with one row for every degree of freedom the joint removes from the system
- $c =$Righ hand side vector

At the next time step, a vector, λ,is calculated (of the same size as c) such that the forces,F_c, applied to the bodies to preserve the joint constraint are

$$F_c = J^T \lambda \tag{3.3}$$

ODE's constraint equation, however, has the form

$$J\frac{dx}{dt} = c + K_{CFM}\lambda \tag{3.4}$$

where K_{CFM} is a square diagonal matrix. K_{CFM} mixes the resulting constraint force in with the constraint that produces it. A nonzero (positive) value of K_{CFM} allows the original constraint equation to be violated by an amount proportional to K_{CFM} times the restoring force λ that is needed to enforce the constraint. For a single step in time,T_{step}, solving for λ gives

$$(JM^{-1}J^T + \frac{K_{CFM}}{T_{step}})\lambda = \frac{c}{T_{step}} \tag{3.5}$$

Where T_{step} is the stepsize.Thus K_{CFM} simply adds to the diagonal of the original system matrix. Using a positive value of K_{CFM} has the additional benefit of taking the system away from any singularity and thus improving the factorizer accuracy.

3.4.2 Using ERP and CFM

The parameters can be used to simulate spring constant kp and damping constant kd, then the corresponding ODE constants are:

$$ERP = T_{step}kp/(T_{step}k_p + kd) \tag{3.6}$$

$$CFM = 1/(T_{step}kp + kd) \tag{3.7}$$

Increasing CFM, especially the global CFM, can reduce the numerical errors in the simulation. If the system is near-singular, then this can markedly increase stability.

3.5 Sources Of Error

Initial investigation into ODE quickly revealed that it is not suited for accurate simulations. The large simulation errors which have been reported by various users,as well as the openDE user guide. This section will briefly looks into a number of sources of errors that are one may encounter when using OpenDE.

3.5.1 Numerical Integration

OpenDE allows the user to chose between for 2 methods of integration which it calls the 'big matrix' and 'Quickstep' method. Both of these methods are describe as first order integration, however the latter is described as being a lot less accurate than then the former. Unfortunately a detailed description of that implementation is not given.

3.5.2 Degrees Of Freedom

OpenDE is restricted to modeling things using a 3D Cartesian coordinate system. This is a great disadvantage when modeling for example a planar system such as a double pendulum, where a 2D Cartesian coordinate system would be sufficient. Furthermore, as constraint forces are very rarely reliant, the simulation domain can be reduced further by expressing the equations of motion in terms of generalized coordinates. Each DOF is represented by a state and the total integration error will increases linearly with each added state. Adding more states to the model then is necessary also

increases another source of error. This can be explained by looking at the general equations of motion for a manipulator 3.8

$$D(q)\ddot{q} + C(q, \dot{q}\dot{q} + G(q) = \tau \tag{3.8}$$

Increasing the number of degrease of freedom to the model also generally increases the condition number of the D(q) matrix. This, as explain in the next section, can leed to truncation of smaller values. As this matrix is inverted when numerically integrating the system ($\int ((D^{-1}(q)(\tau - C(q, \dot{q}\dot{q} - G(q))$), the error due to truncation in the (near) singular directions of D(q) will be very large.

3.5.3 Machine Precision

Open dynamics engine can be installed with single or double precision. This is roughly equivalent to a floating point precision of 10^{-7} or 10^{-16} respectively. Limited precision can cause singularity problems when system matrix is poorly conditioned. For example, the user guide states that the user should avoid using long thin bodies. The reason for this is because inertia matrix will be poorly conditioned, having much higher values for one axis of rotation than in the other. If OpenDE is being used with single precision and the difference in the singular values exceeds 10^7, these lower values would be rounded off to zero. The system matrix would therefor be singular in that direction, resulting in an unstable rotational freedom. 'Long, thin bodies' is such a case would spin around this axis in uncontrolled torsion.

3.5.4 Friction Model

Only a very basic friction model can be used. As such this could possibly be a significant source of error, as well as a significant disadvantage to a Matlab model.

3.5.5 Solutions to the error problem

Although OpenDE has some very significant accuracy problems, being open source it should be possible to solve these issues. It is possible, for example, to increase the order of integration. It would would however be essential to implement fixed step integration however, rather than much more computationally efficient variable step counterpart, due to problems that would otherwise arise in collision analysis.

Chapter 4

Simulations

4.1 Introduction

The Dynamical walker model [?], which was initially chosen to be the basis of comparison for the OpenDE and Matlab simulation was more complex than was initially suspected. As modeling is as much an art form as a science, it was decided to start with more basic simulation. These could be used to pinpoint the weakness or/and strong points of OpenDE software more easily. These simpler model would also serve a dual learning and teaching purpose and thus are explained in greater depth than the walker model.

The first model to be looked into was a double pendulum, was chosen to get a greater understanding on how articulated bodies are modeled. The was discussed in great detail in section 5.7.3, looking closely at the modeling principles involved and results of the actual case study. A second case study of a pendulum colliding with a wall is discussed in section 7. This section concentrated on the comparison of collision handeling functionality as implemented in Matlab and OpenDE

4.2 Evaluation Criterium

Evaluation of the accuracies of OpenDE for the simple models can be derived from comparison of the simulation output to that of analytically derived solutions. Analytically solving the equations of motion of the more complex walker model is however only possible numerically. In Matlab, this is done using typical ode45 solver. This, as will be shown, will give an error per time step as the 5th order function of the time step. This is 3 orders lower than that which Euler integration will give. As such, Matlab output will be considered as being the true simulation output.
Evaluation of other subjective concepts such as 'ease of use' are much more difficult. As OpenDE simulations is based on C (or C++) programming language, the ease of use is highly dependent on the users affiliation with this language. Another problem, as mentioned previously,is that a model can be implemented in many different ways. The best implementation is usually the most simplest and easy to follow, but this again, is subjective.

Chapter 5

Double Pendulum

5.1 Model Description

Figure 5.1 depicts a double pendulum, consisting of 2 links connected to a base. Two frictionless hing joints connect the $link_1$,to the base at A and the the second link, $link_2$ at B. $\theta = \begin{bmatrix}\theta_1 & \theta_2\end{bmatrix}^T$ gives the angles the links make with the horizontal. The links are homogenous, with mass $m = 1[kg]$, and have dimensions: hight $L_h = 1[m]$, width,$L_w = 0.1[m]$ and breadth,$L_b = 0.1[m]$. The initial states of the system, θ_0 and $\dot{\theta}_0$ are $\begin{bmatrix}\pi/4 & 0\end{bmatrix}^T$ and $\begin{bmatrix}0 & 0\end{bmatrix}$ respectively.

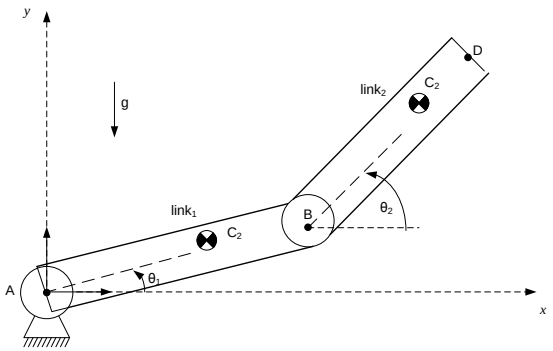


Figure 5.1: Double pendulum

5.1.1 Deriving the Equations Of Motion

There are a number of analytical methods for deriving the equations of motion (EoM) which will have the general form given in equation 5.1.

$$\underline{Q} = \mathbf{M}(\underline{q})\ddot{q} + \mathbf{C}(\underline{q}, \underline{\dot{q}})\underline{\dot{q}} + \mathbf{F}(\underline{\dot{q}}) + \mathbf{G}(\underline{q}) \tag{5.1}$$

where

- q is the vector of generalized joint coordinates describing the pose of the manipulator
- $\dot{q}$ is the vector of joint velocities;
- $\ddot{q}$ is the vector of joint accelerations
- **M** is the symmetric joint-space inertia matrix

- **C** describes Coriolis and centripetal effects ⛃ Centripetal torques are proportional to $\dot{q}_i^2$, while the Coriolis torques are proportional to $\dot{q}_i \dot{q}_j$
- **F** describes viscous and Coulomb friction and is not generally considered part of the rigidbody dynamics
- **G** is the gravity loading
- **Q** is the vector of generalized forces associated with the generalized coordinates q.

Depending on the problem, (for example Lagrangian, Newton-Euler, d'Alémbert or Kane's method, just to name a few..)a careful choice of method can simplify calculations and there by reduce computational time as well as numerical errors.
OpenDE makes use of Newton-Euler methode to derive the eom, followed by Euler integration to update the states information in time. The internal steps have been studied and have been replicated for clarity in this section. This will be followed by a second derivation witch tries to follow standard practices as closely as possibly. The aim is get a clearer understanding of OpenDE and any short comings it may have.

5.1.2 Newton-Euler with Constraints, as Implemented in OpenDE

Introduction

The method implemented by OpenDE is based on the Newton-Euler equations of motion, namely:

$$\sum F = m_c \ddot{x} \tag{5.2}$$

$$\sum M_c = I_{czz} \ddot{\theta} \tag{5.3}$$

Constraint forces produced in the joints are found by my means of Lagrangian multiplier. The second order equations for the position (linear displacement,x and rotation, θ) of the center of mass are then expressed in state-space canonical form, as coupled first order equations which are then integrated once in time.

Derivation of EOM

For analysis, the system is best described in a free body diagram, as shown in figure 5.2.

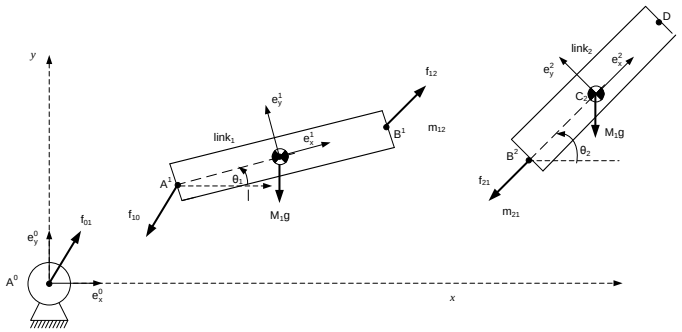


Figure 5.2: Double pendulum

Link 1: Applying equation 5.2

$$\vec{f_{12}} + m_1 \vec{g} + \vec{f_{12}} = m_1 \ddot{\vec{r}}_{com_1} \tag{5.4}$$

The internal forces must cancel each other out (in accordance to the principle of virtual displacement). Letting H_i and V_i denote the vertical and horrizontal components of the force exerted by link i on link i+1, we can express components of the internal forces in the inertial frame, as in equations 5.5, and figure 5.3

$$\begin{aligned}\vec{f}_{10} &= -\vec{f}_{01} \equiv H_1\vec{e}_x^0 + V_1\vec{e}_y^0 \\ \vec{f}_{21} &= -\vec{f}_{12} \equiv H_2\vec{e}_x^0 + V_2\vec{e}_y^0\end{aligned} \tag{5.5}$$

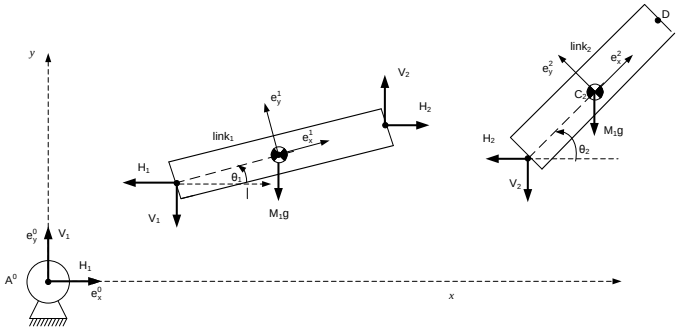


Figure 5.3: Double pendulum

Equating horizontal and vertical components, we can decompose 5.4 into 5.6 and 5.7

$$H_1 + H_2 + m_1 g = m_1 \ddot{x}_1 \tag{5.6}$$

$$V_1 + V_2 = m_1 \ddot{y}_1 \tag{5.7}$$

(Note that the subscribe, COM, has been left out for clarity)

Now by applying Euler equation, 5.3, we can write:

$$(H_1 + H_2)\frac{1}{2}lsin\theta_1 - (V_1 + V_2)\frac{1}{2}lcos\theta_1 = I_1\ddot{\theta} \tag{5.8}$$

Similarly, for the Link 2, we get:

$$\begin{aligned}&H_2 + m_2 g = m_2 \ddot{x}_2 \\ &V_2 = m_2 \ddot{y}_2 \\ &H_2\frac{1}{2}lsin\theta_2 - V_2\frac{1}{2}lcos\theta_2 = I_2\ddot{\theta}_2\end{aligned} \tag{5.9}$$

In order to solve for the 10 unknowns, (6 accelerations: $\ddot{x}_1, \ddot{y}_1, \ddot{\theta}_1, \ddot{x}_2, \ddot{y}_2, \ddot{\theta}_2$, 4 forces: H_1, V_1, H_2, V_2), we need 4 constraint equations, which can be derived from the position of the joints:

$$\begin{aligned}r_A^0 = 0 \Rightarrow \\ &x_1 - \frac{1}{2}lcos\theta_1 = 0 \\ &y_1 - \frac{1}{2}lsin\theta_1 = 0\end{aligned} \tag{5.10}$$

$$r^0_{B1} - r^0_{B2} = 0 \Rightarrow$$

$$x_1 + \frac{1}{2}cos\theta_1 - (x_2 - \frac{x}{2}cos\theta_2) = 0$$
$$y_1 + \frac{1}{2}sin\theta_1 - (y_2 - \frac{x}{2}sin\theta_2) = 0 \tag{5.11}$$

Differentiating 5.10 and 5.11 twice with respect to time now gives:

$$\begin{aligned}
\ddot{x}_1 + \frac{1}{2}l_1\ddot{\theta}_1 s_1 + \frac{1}{2}\theta_1^2 c_1 &= 0 \\
\ddot{y}_1 + \frac{1}{2}l_1\ddot{\theta}_1 c_1 + \frac{1}{2}\theta_1^2 s_1 &= 0 \\
-\ddot{x}_1 + \ddot{x}_2 + \frac{1}{2}l_1\ddot{\theta}_1 s_1 + \frac{1}{2}l_1\ddot{\theta}_2 s_2 + \frac{1}{2}\theta_1^2 c_1 + \frac{1}{2}\theta_2^2 c_2 &= 0 \\
-\ddot{y}_1 + \ddot{y}_2 + \frac{1}{2}l_1\ddot{\theta}_1 c_1 + \frac{1}{2}l_1\ddot{\theta}_2 c_2 + \frac{1}{2}\theta_1^2 s_2 + \frac{1}{2}\theta_2^2 s_2 &= 0
\end{aligned} \tag{5.12}$$

(Note that the abbreviations $s_i = sin\theta_i$ and $c_i = cos\theta_i$ have been used)
Combining above equations into a set of differential algebraic equations gives:

$$\begin{bmatrix} M & A \\ B & 0 \end{bmatrix} \begin{bmatrix} \ddot{x} \\ f_v \end{bmatrix} = \begin{bmatrix} f_z \\ a(x, \dot{x}) \end{bmatrix} \tag{5.13}$$

with $M = diag\begin{bmatrix} m_1 & m_1 & I_1 & m_2 & m_2 & I_2 \end{bmatrix}$
$\ddot{x} = \begin{bmatrix} \ddot{x}_1 & \ddot{y}_1 & \ddot{\theta}_1 & \ddot{x}_2 & \ddot{y}_2 & \ddot{\theta}_2 \end{bmatrix}$
$f_v = \begin{bmatrix} H_A & V_A & H_B & V_B \end{bmatrix}$
$f_z = \begin{bmatrix} m_1 g & 0 & 0 & m_2 g & 0 & 0 \end{bmatrix}$

Premultiplying both sides of 5.13 by the inverse of the first matrix will give the solutions for constraint forces. After adding the intermediate state $v = \dot{x}$, changing the system into a coupled frist order system, it can be integrated numerically.

5.1.3 Integration

Introduction

Finding an exact analytical solution for nonlinear equations of motion is not always possible or feasible due to the computational cost. This section will look closely into Euler integration, which is used in OpenDE and discus some key issues, namely stability, accuracy and efficiency.

Euler Integration

First order Euler integration can be explained by looking at the Taylor expansion of an arbitrary, smooth function x(t)

$$x(t_0 + \Delta t) = x(t_0) + \Delta t\dot{x}(t_0) + \frac{\Delta t^2}{2}\ddot{x}(t_0) + \frac{\Delta t^3}{3}\dddot{x}(t_0).. \tag{5.14}$$

Truncating the infinite series after the second term gives the first order approximation of $x(t_0 + \Delta t)$, known as Euler integration. For a given (constant) step size, $\Delta t = h$, this becomes

$$x(t_0 + h) = x(t_0) + h\dot{x}(t_0) \tag{5.15}$$

For small enough step size, the Taylor series is dominated by the leading terms, hence the error (the remaining terms in 5.14, can be described as $O(h^2)$ Note that the total number of integration steps required, N, depends inversely on the stepsize, h^{-1}. This means that the effective error depends linearly on step size. In practice, the error per step is also bounded by limited computational precision causing rounding off errors, a phenomenon known as numerical drift.

5.2 Implementation

5.2.1 Introduction

Every simulation will start with the *setup phase*,where the user must provide initial paraments and conditions. This consists of:

- model setup: Defining the model parameters (mass, inertia, joints,etc) and initial conditions (x and $\dot{x}$). OpenDE has numerous functions which facilitate this. For example, by either giving the mass or density along with the dimensions of a primitive object (rectangle, spheer, cylinder or capped cylinder), it calculates and stores the inertia values.

- simulation setup: Selecting type of integration, size of time step, duration of simulation..

Next, the *simulation phase* is carried out, loosely consists of 4 steps which are repeated after each time step until the simulation has ended.

.

- Calculation of forces: A variable F, called the *force accumulator*, assigned to each individual (ridgid) body is updated by summing all the forces which act upon that body.

- Time Step: The solver gathers all positions and velocities in phase space: ($x_k, \dot{x}_k$and $\ddot{x}_k = F/m$). Then by applying some kind of integration routine calculates $x_{k+1}, \dot{x}_{k+1}$

- Collision Detection: Checks to find out if any objects have collided. Often the most costly step with respect to processing time. If so, 'virtual' joints are added to points of contact containing penetration depth.

- force accumulator is cleared.

The setup phase can be understood by studying the comments next to the code. More interesting and meaningful is to study the steps taken during the simulation phase. Although OpenDE is only able to model 3D ridgid bodies, (not 2D point masses for that matter), it has been programmed according to well established methods [?]. For this reason, to understand how it works it is we shall look at how a 2D concentrated mass model of the double pendulum *would be* implemented, than build up to the 3D RB version.

As a side note, to facilitate automatic derivation and solving of the equations of motion for arbitrary models, the code is structured into sperate code blocks, or modules, each preforming it's own operation, but also often accessing information from other modules. Notably, the C++ scripting language is particularly well suited for this approach to programming (called Object Orientated Programming ,OOP,), having been originally designed with this in mind.

5.2.2 Particle object

Initially we shall only look at a single link, with it's mass concentrated at the center (as shown in figure 5.4)

The state (position and velocity ,x,v)and mass,m, are stored in a **particle object** during the initial setup phase. The particle object also stores force f,shown in the figure 5.5 .

The user interface to this object is a function which would have a function signature looking something like:

```
PARTICAL point(double m, double pos_x,double pos_y double vel_x,double vel_y)
```

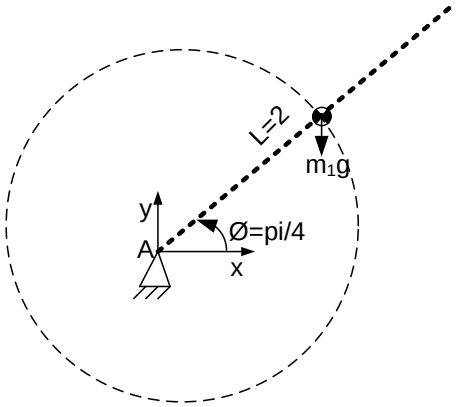


Figure 5.4: Single Pendulum, modeled as a partial constrained on a circular path

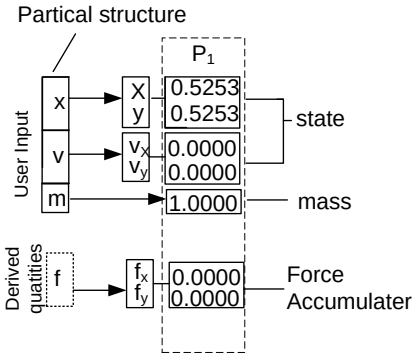


Figure 5.5: Particle Structure filled in with the initial set up data for the singel pendulum model

The function signature gives the input/output relation of a function and is useful to the end user as it indicates how to use a particular function. Here, 'POINT', tells the user there is no output. Between the brackets are the input type and names separated by commas. For the single pendulum example, the user would implement the point mass as:

```
P1=point(1,0.5253,0.5253,0,0)
```

Notice that the recorded forces are zero at this moment. Before each simulation step ($t_k \rightarrow t_{k+1}$), the forces are added to f(known as the 'force accumulator' by so called **force objects**).

5.2.3 Force object

Each force object stores a particular force law, such as force due to gravity (hard programmed), and the particle(s)(if any), it acts upon, set in the model setup phase. A spring force, for example will not generally be applied to every particles in a model. This makes the force objects heterogeneous by nature. An example of the gravity object can be seen in figure 5.6.

During the model setup phase, it needs user input for the direction and magnitude of the gravitational force along with a list of particles it affects. The latter is not strictly necessary as gravity, unlike other force objects, will acts on all particles of the system.

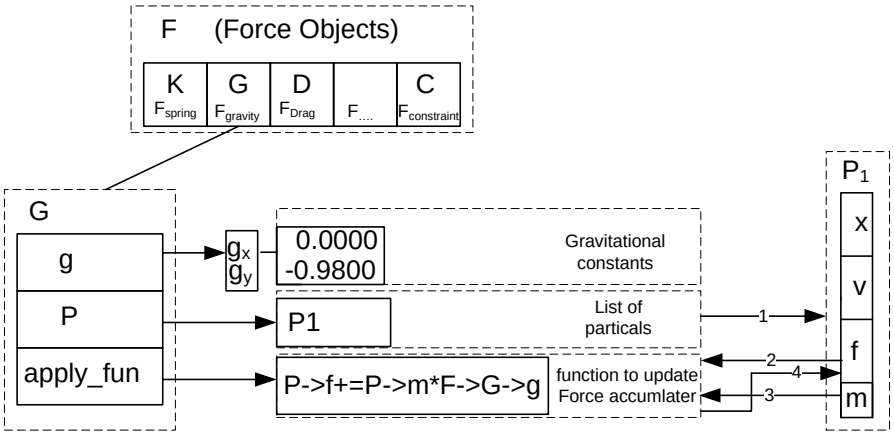


Figure 5.6: Structure of a gravity opject

The numbered arrows indicate the direction of data transfer. The force object loops over all the particle structures listed in P, (for this case just P_1). For each particle structure it runs the pseudocode

```
P->f+=P->m*F->G->g
```

The code reads: access force accumulator 'f' of particle structure, 'P'. *Add to this value* mass 'm' multiplied by constant 'g' (which is found by accessing gravity object 'G' after accessing force object 'F')
Constraint force is a special type of force in that it is dependent on the other forces acting on the particle. For this reason it's part is added to the force accumulator only after all the other forces have been added (as indicated in figure 5.6).

5.2.4 System and solver objects

During the time step ($t \Rightarrow t+\Delta T$) the system states (position and velocity of all particles) are updated ($X_k \Rightarrow X_{k+1}$) To facilitate this step, the particle objects, along with the system parameters (time,t and the time step,ΔT) are saved in a **system object**. The number of particles,n, in the systems is also stored in this object. This variable is iterated automatically when a particle is added or removed for the system. A **solver object** updates the states. The proces involves copying the states and derived state velocities (using 'n' to allocating memory). The new states at $t = t + \Delta T$ are found by applying Euler integration (as explained in section 5.16)

$$\begin{aligned} X_{k+1} &= X_k + \dot{X}_k \Delta T \\ \begin{bmatrix} x_{k+1} \\ v_{k_1} \end{bmatrix} &= \begin{bmatrix} x_k \\ v_k \end{bmatrix} + \Delta T \begin{bmatrix} v_k \\ F/m_i \end{bmatrix} \end{aligned} \tag{5.16}$$

The main stages of this process (updating the solver and system) can be viewed in figure 5.7.

5.3 Constraint Force

5.3.1 Introduction

Constraint forces require relatively complex implementation compared to, for example spring forces and have therefore been given there own section. Theoretically it would be more correct to model the the particle as if it were attached by a spring (with spring constant,k_s, equal to Youngs modules of

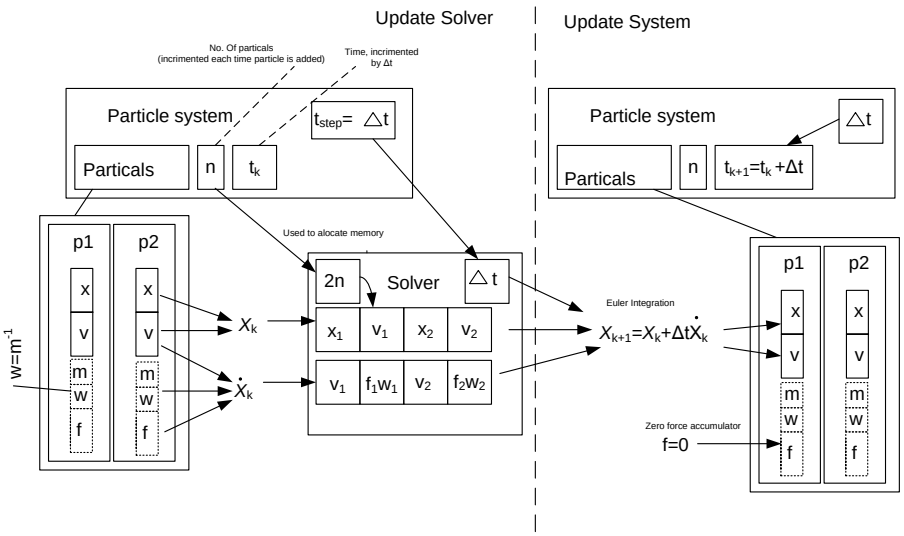


Figure 5.7: Solver Object

model material, multiplied but constrained length, divided by the cross sectional area of the pendulum). The problem with this approach is that k_s would be very large, require a subsequently very small (integration) time step to prevent integrating instability as discussed in 5.1.3.

5.3.2 Derivation of Constrain Equations

Constraint forces cancel only those parts of the applied forces which act against the constraint. As $F = ma$, it is therefore possible to think of these forces as converting a particles acceleration into "legal" accelerations, consisted with the constraint. This relation can be found by formulating implicit function for a position constraint ($C(x) = 0$), then differentiating this twice with respect to time. (Note that x is in this case a two dimensional vector)

$$C = \|x\| - r = x \cdot x - r^2 = 0 \quad Position\ constraint \tag{5.17}$$
$$\dot{C} = x \cdot \dot{x} = 0 \qquad Velocity\ constraint \tag{5.18}$$
$$\ddot{C} = \ddot{x} \cdot x + \dot{x} \cdot \dot{x} = 0 \quad Acceleration\ contraint \tag{5.19}$$

The positions and velocities that satisfy the constraints 5.17 and 5.18, are known as *legal* positions and velocities. These constraints will be maintained as long as 5.19 is satisfied.
Expressing the acceleration in terms of applied forces, f_a and (unknown) constraint force, f_c, we can write

$$\ddot{x} = \frac{f_a + f_c}{m} \tag{5.20}$$

Substituting for $\ddot{x}$ into 5.19 gives

$$\ddot{C} = \frac{f_a + f_c}{m} \cdot x + \dot{x} \cdot \dot{x} = 0$$
$$\Rightarrow f_c \cdot x = -f_a \cdot x - m\dot{x} \cdot \dot{x} \tag{5.21}$$

In order to solve for the two components of f_c a second equation is needed. This second equation comes from the requirement that constraint forces, being passive and lossless, never add to nor remove energy from the system.

Differentiating the kinetic energy, $T = \frac{m}{2}\dot{x} \cdot \dot{x}$ gives an expression of work. Substituting for $\ddot{x}$ gives

$$\dot{T} = m\ddot{x} \cdot \dot{x} = mf_a \cdot \dot{x} + mf_c \cdot \dot{x} \tag{5.22}$$

The requirement means that the second term must be zero. Combining 5.22 and 5.18gives:

$$f_c \cdot \dot{x} = 0, \forall \dot{x} \| x \cdot \dot{x} = 0 \tag{5.23}$$

Equation 5.23 states that f_c must point in the direction of x, thus we can write

$$f_c = \lambda x \tag{5.24}$$

where lambda is an unknown scalar. Substituting f_c in 5.21 and solving for λ gives

$$\lambda = \frac{-f_a \cdot x - m\dot{x} \cdot \dot{x}}{x \cdot x} \tag{5.25}$$

Having found lambda, we can solve for f_c in equation 5.24 and finally the acceleration from 5.20. For a geometric interpretation of f_c, assume particle is at rest. Solving for f_c then simplifies to 5.26

$$f_c = -\frac{f \cdot x}{x \cdot x}x \tag{5.26}$$

This shows that f_c is the orthogonal projection of f onto the circle's tangent. The force component removed by the projection is that which points out of the legal motion direction.

5.3.3 General Case

In the previous section we looked at the implementation of only one constraint acting on a single particle. This section will look at the implementation of more general system of constraints. For this analysis we lump the positions of the particles in vector q. As this is a 2D model, for n particles q would have a length of $2n$. Next we define the mass matrix, **M**, as a diagonal matrix containing the particles masses: $\mathbf{M} = diag(m_1 \quad m_1 \quad m_2 \quad m_2 \quad .. \quad m_n \quad m_n)$. We define **W** as the inverse of this matrix, such that $\mathbf{W} = diag(m_1^{-1} \quad m_1^{-1} \quad m_2^{-1} \quad m_2^{-1} \quad .. \quad m_n^{-1} \quad m_n^{-1})$. We can now write:

$$\ddot{q} = \mathbf{WQ} \tag{5.27}$$

Where **Q** contains the forces acting on the particle.

Similar to the previous section, we group all the constraints into a single vector function **C**(q). (For n, 2D particles, subject to m scalar constraints, the output would be an m-vector with $2n$-input vector.) Assume legal positions and velocities (such that $\mathbf{C} = \dot{\mathbf{C}} = 0$) Now, analogous to the previous section, we need to find the constraint forces Q_c, which are added to the applied forces, Q_a, which guarantees that $\ddot{C} = 0$

In the previous section we had an algebraic expression for C. C will now be left as an anonymous function of the states, hence the time derivative is expressed as:

$$\dot{\mathbf{C}} = \frac{d\mathbf{C}}{dq}\dot{q} \tag{5.28}$$

Matrix $\frac{d\mathbf{C}}{dq}$ is called the *Jacobian* of **C** and will denoted by variable **J**. Differentiating the constraint function, **C**, a second time 5.28 w.r.t time will then give:

$$\ddot{\mathbf{C}} = \dot{\mathbf{J}}\dot{q} + \mathbf{J}\ddot{q} \tag{5.29}$$

Using the systems equations of motion,5.27, and replacing $\ddot{q}$ by a force expression gives:

$$\ddot{\mathbf{C}} = \dot{\mathbf{J}}\dot{q} + \mathbf{JW}(\mathbf{Q}_a + \mathbf{Q}_c) \tag{5.30}$$

Setting 5.30 to zero gives the general form of equation 5.23

$$\mathbf{JWQ}_c = -\dot{\mathbf{J}}\dot{q} - \mathbf{JWQ}_a \tag{5.31}$$

Again, there are more unknowns than equations and to solve for $\mathbf{Q}_c$ we need to use the principle of virtual work. Let **x** be the 'legal velocities', which obey the constraint equation, $\dot{C}$ or $\mathbf{J}\dot{x} = 0$, to ensure that the constraint force does not preform any work, we require that

$$\mathbf{Q}_c \cdot \dot{x} = 0, \;\; \forall \dot{x} | \mathbf{J}\dot{x} = 0 \tag{5.32}$$

The solutions to equation 5.32 will be spanned by the *null space complement* of **J** and can be expressed as:

$$\mathbf{Q}_c = \mathbf{J}^T \lambda \tag{5.33}$$

where λ is a vector of dimension **C**, who's components are known as Lagrange multipliers. Note that the set of vectors, v, which span the null space of J, (such that $\mathbf{J}v = 0$ contain the displacement gradients of the *legal* displacement). Now by inserting 5.33 into 5.31, it is possible to solve for λ:

$$\mathbf{JWJ}^T \lambda = -\dot{\mathbf{J}}\dot{q} - \mathbf{JWQ}_a \tag{5.34}$$

$$\lambda = -(\mathbf{JWJ}^T)^{-1}(\dot{\mathbf{J}}\dot{q} + \mathbf{JWQ}_a) \tag{5.35}$$

After solving for λ, it can be used in equation 5.33 to find the constraint force.

5.3.4 Feedback Term

In order to prevent the accumulation of numerical drift, which would effectively cause joint segments to separate as the simulation progresses, a feedback term is introduced. Rather than solving for $\ddot{\mathbf{C}} = 0$, we solve for

$$\ddot{\mathbf{C}} = -k_s\mathbf{C} - k_d\dot{\mathbf{C}} \tag{5.36}$$

$$\tag{5.37}$$

λ is then solved for:

$$\mathbf{JWJ}^T \lambda = -\dot{\mathbf{J}}\dot{q} - \mathbf{JWQ}_a - k_s\mathbf{C} - k_d\dot{\mathbf{C}} \tag{5.38}$$

In OpenDE this paraments can be set globally and/or for each joint separately.

5.3.5 Implementation Constrained Dynamics

In order to preform the simulation, the constraint function **C**(q) must be provided. Using Matlab,the method would generally be deriving the constraint functions and their derivatives symbolically and substitute them into 5.38. *Dynamics engines*deviate from this derive and implement approach by having constraint 'objects'similar to the force objects discussed earlier, which automatically apply certain constraint functions from a given set of general constraint functions. This implementation will be discussed in this section. This computational step in the sequence of simulation steps discussed earlier, is part of the force accumulation which can now be separated into a separate step as follows:

1. Clear forces :zero the force accumulator variable of each partial
2. Calculate applied forces: loop over force objects
3. Calculate constraint forces: After the applied forces,$\mathbf{Q}_a$, are gathered in the previous step, constraint forces, $\mathbf{Q}_c$ can be determined form equations 5.38 and 5.33, and added to Q_c

4. Integrate system: Forces and states set to solver module which calculates new states

In order to preform this third step, C, $\dot{C}$, J and $\dot{J}$ are needed. The equations that describe these need to be implemented in a generic fashion, such that the models are built automatically. As joints are the physical object that impose constraints, *joint functions* is a natural choice for naming the user functions for implementing constraints. Looking at the pendulum problem, these joint functions would need the following function arguments:

- 2 (mass) points which the joint is connected to
- Position of the joint axis.

The joint function signature would therefore look something like

```
void Joint(Point P1,Point P2, pos PosJ)
```

The (hinge) joint function imposes 2 distance constraints: one between each point, $r^0_{P_i}, r^0_{P_{i+1}}$ and joint, $r^0_{J_i}$.

5.3.6 Rigid Bodies

This section will extend the description of the system to include ridged bodies in 3D space. Ridged bodies describe cluster of particles which are fixed in the relative to a coordinate frame attached to the body. The location of an arbitrary point,$\vec{r_i}$ on the body, when expressed in the initial coordinate fame, $r^0_i(t)$, can described by a *constant* vector,r^b_i, in the body-fixed coordinate frame, a rotation of the body fixed coordinate frame ($^0_bR(t)$)and the distance,($x^0_b(t)$), form that frame to the inertial frame.

$$r^0_i(t) =^0_b R(t) r^b_i + x^0_b(t) \tag{5.39}$$

To simplify allays we shall dorp the sub and superscripts where they are not essential for understanding the problem. x(t) and R(t) will now be refired to as the *position* and *orientation* of the body.

5.3.7 Linear and Angular Velocity

Differentiating 5.39 to get the velocity of arbitrary particles gives:

$$\begin{aligned} \dot{r}_i(t) &= \dot{R}(t) r_b i + \dot{x}(t) \\ &= w(t) \times R(t) r_b i + v(t) \end{aligned} \tag{5.40}$$

where $v(t) = \dot{x}$, and $\omega(t)$ are the linear and angular velocities of the rigid body. Equation 5.40 can also be rewritten as:

$$\begin{aligned} \dot{r}_i(t) &= w(t) \times (R(t) r_b i + x(t) - x(t)) + v(t) \\ &= w(t) \times (r_i - x(t)) + v(t) \end{aligned} \tag{5.41}$$

5.3.8 Force and Torque

In order to simplfy analysis, the body fixed coordinate is places at the center of mass of the rigid body (such that $\frac{\Sigma m_i r_{0i}}{M} = 0$) Let F_i denote the total force that is exerted by forces acting on an arbitrary partial p_i then the torque acting on that particle relative to the COM, is defined as

$$\tau_i(t) = (r_i(t) - x(t)) \times F_i(t) \tag{5.42}$$

The total force and torque acting on the body is now simply the sum all torques and forces

$$F(t) = \Sigma F_i \tau(t) = \Sigma (r_i(t) - x(t)) \times F_i(t) \tag{5.43}$$

5.3.9 Linear Momentum

Linear momentum,p, of a particle of mass m_i and velocity $\dot{r}_i$, is defined as $p_i = m_i\dot{r}_i$. The total linear momentum of a ridged body is therefor the sum of all linear velocities of it's particles:

$$P(t) = \Sigma m_i \dot{r}_i(t) \tag{5.44}$$

Substituting the expression for $\dot{r}_i$ from 5.41, we get

$$\begin{aligned} P(t) &= \Sigma(m_i v(t) + m_i \omega(t) \times (r_i(t) - x(t))) \\ &= \Sigma m_i v(t) + \omega(t) \times \Sigma m_i (r_i(t) - x(t)) \\ &= Mv(t) + \omega(t) \times \Sigma m_i (r_i(t) - x(t)) \end{aligned} \tag{5.45}$$

where M is the total mass the the system. Note that as the body coordinate frame is positioned at the center of mass, we can write:

$$\Sigma m_i(r_i(t) - x(t)) = \Sigma m_i(R(t)r_{oi} + x(t) - x(t)) = R(t)\Sigma m_i r_{oi} = 0 \tag{5.46}$$

Therefore

$$P(t) = Mv(t) \tag{5.47}$$

As $F(t) = M\ddot{x} = \frac{d}{dt}(M\dot{x}) = \dot{P}(t)$, this gives a similar result to that of a simple particle motion:

$$\dot{v} = \frac{F(t)}{M} \tag{5.48}$$

5.3.10 Angular Momentum

Similar to the definition of linear momentum, we can define total angular momentum, L, of a rigid body by the equation $L(t) = I(t)\omega(t)$, where $I(t)$ is 3×3 inertia tensor describing the mass distribution relative to the bodies center of mass. Analogues to the relation $\dot{P}(t) = F(t)$, the relation between the $L(t)$ and the total toque $\tau(t)$ can be shown to be

$$\dot{L}(t) = \tau(t) \tag{5.49}$$

Note that although I(t) is time dependant function, if the orientation, R(t)of the body is known it can be simply calculated by:

$$I(t) = R(t)I_{body}R(t)^T \tag{5.50}$$

Where I_{body} is the constant inertia matrix specified in body-space.

5.3.11 Rigid Body Equations Of Motion

Position,x, orientation,R, as well as linear and angular momentum,P and L, are chosen as states,X, such that:

$$X(t) = \begin{bmatrix} x(t) \\ R(t) \\ P(t) \\ L(t) \end{bmatrix} \tag{5.51}$$

During the initial set up, rigid body parameters M and I_{body} as well as the position, orientation and the linear and angular velocities are given such that

$$X(0) = \begin{bmatrix} x(0) \\ R(0) \\ P(0) \\ L(0) \end{bmatrix} = \begin{bmatrix} x(0) \\ R(0) \\ Mv(0) \\ R(0)I_{body}R(0)^T w(0) \end{bmatrix} \tag{5.52}$$

The velocity in phase state then becomes

$$\frac{d}{dt}X = \frac{d}{dt}\begin{bmatrix} x(t) \\ R(t) \\ P(t) \\ L(t) \end{bmatrix} = \begin{bmatrix} v(t) \\ \omega(t) \times R(t) \\ F(t) \\ \tau(t) \end{bmatrix} \tag{5.53}$$

$F(t)$ and $\tau(t)$ are calculated using 5.43 in a similar fashion as was shown in section 5.2.3. Auxiliary quantities $I(t)$,$\omega(t)$ and $v(t)$ are computed by

$$v(t) = \frac{P}{M} \tag{5.54}$$
$$I(t) = R(t)I_{body}R(t)^T \tag{5.55}$$
$$\omega(t) = I(t)^{-1}L(t) \tag{5.56}$$

Note that the $\omega(t) \times R(t)$ can be rewritten in terms of pure matrix multiplication, $W(t)R(t)$ where W is the skew-symmetric matrix:

$$W(t) = \begin{bmatrix} 0 & -\omega(t)_3 & \omega(t)_2 \\ \omega(t)_3 & 0 & -\omega(t)_1 \\ -\omega(t)_2 & \omega(t)_1 & 0 \end{bmatrix} \tag{5.57}$$

5.4 Quaternion

In OpenDE orientation is implemented as a quaternion,q rather than in rotation matrix form. A quaternion is made represented by a rotation angle and rotation axis of unit length, $q = [cos(\theta/2), sin(\theta/2))* u]$. As this requires only 4 numbers rather than 9,a quaternion is a beter method of implementation. Also, compensation for numerical inaccuracies can also be carried out more easily. For example, for a rotation matrix, numerical inaccuracies could require the need to preform some operation such that it regains it's orthogonal properties. For the same situation a quaternion only needs to be scaled back to unit magnitude.

5.5 OpenDE Implementation

In order to keep the code readable, the code is separated into various files which are compiled together to produce an executable (exe file), which when run preforms the simulation. A brief description of these files are:

- *pendulum_main.cpp* C.1: Calls the separate files when needed
- *pendulum_body.cpp* C.2: Contains the function for building the pendulum model
- *pendulum_body.h* C.3: Contains all the function and variable declarations used in $penulum_body.cpp$
- *pendulum_par.h* C.4: Contains all the parameters used in the simulation (eg mass,rotation angles)

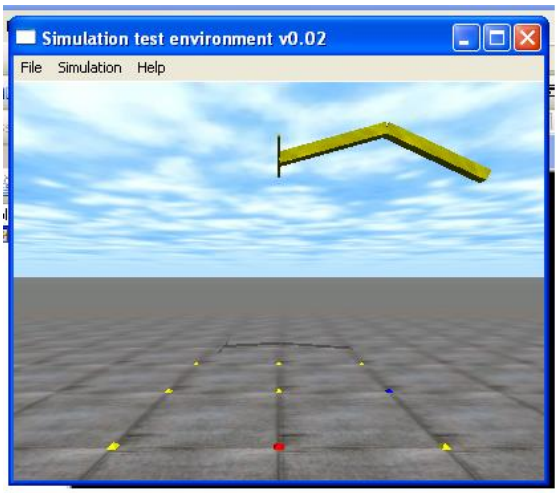


Figure 5.8: Screen shot of the double pendulum simulation

A screen shot of the double pendulum can be seen in figure 5.8
The position, orientation, linear and angular velocities are are also outputted to an m-file. The rotation angles about the x-axis (the rotation axes of the joints) are then extracted form the orientation. The angles and angular velocities have been plotted in figure 5.9, along with phase plots and the derived potential and kinetic energy of the system. Note that some data processing was needed to make the simulation output comparable to that of the matlab simulation as has is explained in appendix B.

5.6 Matlab Implementation

In DP_EOM.m, (appendix D.1), Lagrange formalism was used to construct the symbolic matrices: **D**(q),**C**(q,$\dot{q}$) and **g**(q), such that Newton-Euler equations of motion of the system are given by:

$$D(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) = \tau \tag{5.58}$$

Where τ, the externally applied joint torque in this case is zero. Also, q and the gravity vector $\vec{g}$, used to construct **g** is defined in figure 5.10.

The matrices were used to construct the equation of motion, in a function file, xD_DP.m, (in appendix D.2, computing

$$\dot{x} = \begin{bmatrix} \dot{q} \\ \ddot{q} \end{bmatrix} = \begin{bmatrix} \dot{q} \\ D^{-1}(q))(-C(q,\dot{q} - g(q)) \end{bmatrix} \tag{5.59}$$

simDP.m, in appendix D.3, integrates the system, x, from time: $t = 0$ to $t = 10$, given initial values $x(0) = \begin{bmatrix} q_1(0) & q_2(0) & \dot{q}_1(0) & \dot{q}_2(0) \end{bmatrix}^T = \begin{bmatrix} pi/4 & -pi/4 & 0 & 0 \end{bmatrix}^T$. The sum of the potential and kinetic energy is also plotted given in figure 5.12).
The system was also integrated using Euler integration, with the results shown in figure 5.11. The time simulation time step, $t = 0.001$, corresponds to that chosen for the OpenDE simulation.

5.6.1 Robotics Toolbox

5.6.2 Introduction

The model was also implemented using a robotics toolbox for Matlab, [?]. The toolbox provides functions to model serial articulated kinematic chains. "DNY" matrix used to construct "robot" objects are

used to model rigid body links in a similar manner "body" objects are used in OpenDE. The main differences between these two objects are:

- The position (of the COG) of the body object is described by a variable, in terms of an absolute (world) coordinate frame. Along with the rotation matrix (describing the rotation of a body-fixed coordinate frame w.r.t the world coordinate frame), these make up the states of the object. The position of the COG for a link object, however is described in terms of the link coordinate frame (either using the standard or modified DH convention). This makes the description an object property, rather than state. Generalis coordinates are the states of the object. For this reason, the Matlab toolbox is restricted to only modeling serial articulated chains connected to a fixed base.
- As opposed to the DNY matrix, the body object also contain geometric description of the rigid body and is used for collision detection. The Matlab toolbox has no collision detection functionality which is a great disadvantage.

An advantage for the Matlab toolbox is that it provides functions which uses Ode45 to integrate the system rather than the much less efficient and accurate Euler integration used in OpenDE.

5.6.3 Implementation using the Robotics Toolbox

The Robotics Toolbox constructs the equations of motion using recursive Newton-Euler formulation. After implementing the model and running the simulation, (see D.4), results were found to be identical to the those found previously.

5.6.4 Implementation with MEX files

A known drawback to using Matlab rather than C++ for modeling and system is that Matlab is computationally much slower. [?] document mentions speed improvements of greater than 1000 times when using C code as apposed to Matlab code. For models with many degrees of freedom, it can take days to preform a simulation in Matlab (for example Pieter van Zitven's humanoid model [?]). Matlab however is a much more user friendly language in which to program. A good alternative is to combine the two used MEX files.

Simulations carried out using the robotics toolbox call a reclusive Newton-Euler algorithm (implemented in the rne.m file) each time step which is the bottel neck of the simulation. The robotics toolbox also provides this algorithm in c code which the user can compile to a MEX file extension with the same name. Matlab functions *tic* and *toc* were used to record time needed to run the simulation before and after mex files were used.

5.7 Comparison of Results

Five different methods have been used to implement the double pendulum model as summed up in table 5.2. The first 3 implementation use the same model and solver and therefor produce the same output but differ in computational time. This will be looked into more closely in this section.

5.7.1 Computation Time

An overview of the computation time for the various simulations preformed is given in table 5.1. In order to make the simulation conditions as similar as possible both numerical and graphical output was suppressed until after the simulation had been completed. Additionally, the simulations were all performed on the same computer.

From table 5.2, case 1 and 3, it can be seen that running the simulation using the robotics toolbox was almost 20 times slower than without the help of this toolbox. This was initially unexpected as both

Table 5.1: Overview methods of implementation

Case No.	coordinate system	Platform	Toolbox / Library	Solver
1	generalized	Matlab	Robotics toolbox	Rung Kutta, (4,5)
2	generalized	Matlab / C++	Robotics toolbox (mex files	Rung Kutta, (4,5)
3	generalized	Matlab	-	Rung Kutta, (4,5)
4	generalized	Matlab	-	Euler
5	cartesian	C++	OpenDE	Euler

Table 5.2: Overview of the computation time

Case No.	Platform	Toolbox / Library	Solver	Time Steps	Sim. Time [s]
1	Matlab	Robotics toolbox	Rung Kutta, (4,5)	500	6.60
2	Matlab / C++	Robotics toolbox (mex files	Rung Kutta, (4,5)	500	1.13
3	Matlab	-	Rung Kutta, (4,5)	500	0.35
4	Matlab	-	Euler	10000	1.13
5	C++	OpenDE	Euler	10000	0.29

models and the solver used were identical. The reason for the difference lies in the implementation. For case 3, the equations of motion were already derived and implemented in a function file (see D.1). When using the robotics toolbox, before integrating the system, the equations of motion are first derived given the link parameters.

This step is implemented with the mex files (in case 2, as explained in section 5.6.4) giving an almost six fold improvement in the computation time.

Comparing case 3 and 5 show that OpenDE was only marginally faster than the Matlab, however this result is dependant on the chosen size of the time step implemented in OpenDE. Case 4 and 5 reveal that OpenDE is almost 4 times faster than the matlab implementation using equivalent solvers.

5.7.2 Energy of the system

Naturally, the total energy of the system (potential + kinetic) should in theory remain constant making it a good indicator of the presents of modeling or simulation error. Results for the OpenDE simulation (figure 5.9) show the simulated energy level of the system change with much greater amplitude then these of simulated in Matlab with both Rung Kutta and Euler integration (figures 5.12 and 5.11). The reason for such highly fluctuating simulation energy in the OpenDE can be explained by the fact that joint error feedback gains are implemented which effectively adds externally applied forces to the system.

5.7.3 Accuracy

Matlab models integrated with Rung Kutta (4,5) and Euler are compared to the OpenDE simulation in figure 5.13. Although the Matlab model integrated with Rung Kutta (4,5) will have numerical error as shown when looking at the system energy (discussed in section 5.7.2), the method of integration is known to be much more accurate than Euler and will can therefor be considered as the most accurate. On this Bases it can be concluded that the OpenDE simulation appears to be more accurate than the Matlab, Euler model, following the Rung Kutta model more closely, at least until just over 6 seconds. The large deviation present in the OpenDE simulation after 6 seconds might be caused by the sensitivity of the system's output with respect to it's initial values rather than any significant modeling

error. This hypothesis was analyzed by again simulating a second time using Rung Kutta (4,5) solver but with perturbed initial states.
The order of the magnitude of this perturbation was chosen to coincide with the magnitude of the expected error in the states due to euler integration after 6 seconds. As the expected error was shown in section 5.1.3 to be linear function of the step size, the perturbation was chosen to be $\delta = t_{step} t_{span} = 0.001\text{x}6 = 0.006$
The simulation with and without perturbed initial states can be seen in figure 5.14. Large deviations in the results for q_2 after just over 3 seconds shows that the system's states indeed appear to be sensatie to it's initial conditions. As a consequence, it may have been more meaningful to compare simulation results for a single pendulum.

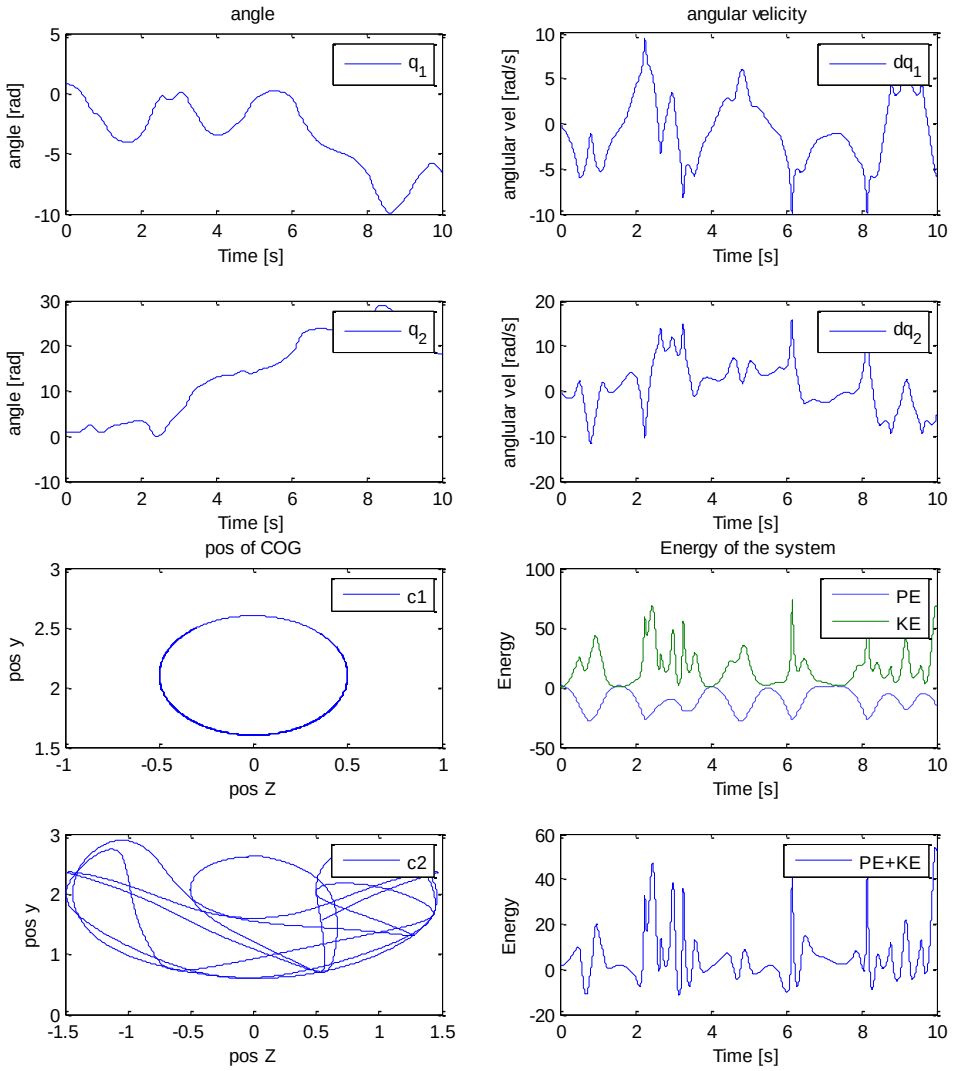


Figure 5.9: Resulting time-phase, position of COG, and energy plot of the system modeled in OpenDE (t_step=0.0001)

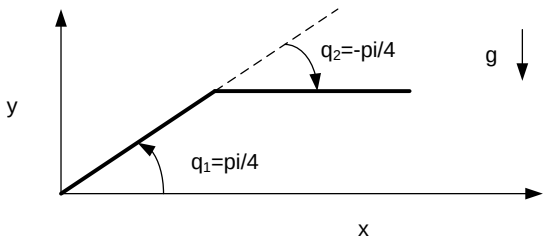


Figure 5.10: Figure showing definition of q_1, q_2 and $\vec{g}$, as well as the initial states. (note $\dot{q}_1 = \dot{q}_1 = 0$)

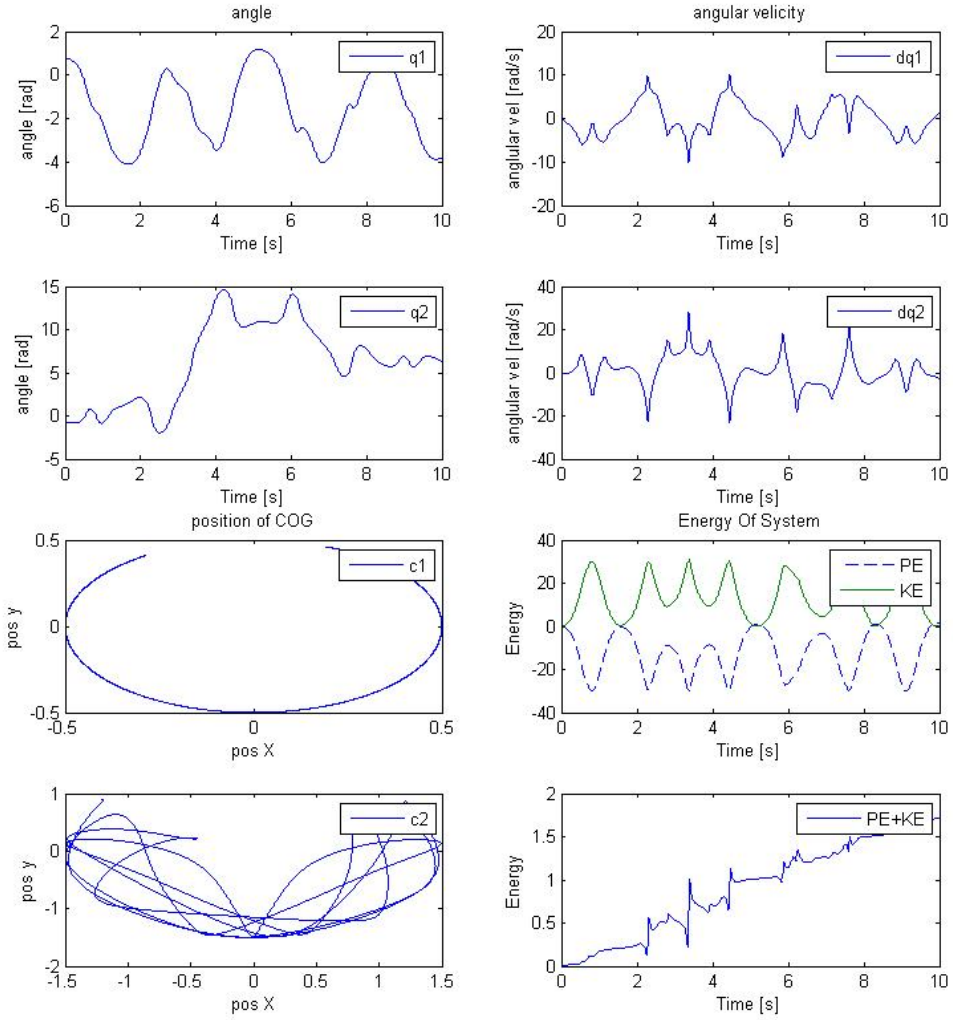


Figure 5.11: Resulting time, phase and energy plots of the system modeled in Matlab using Euler integration

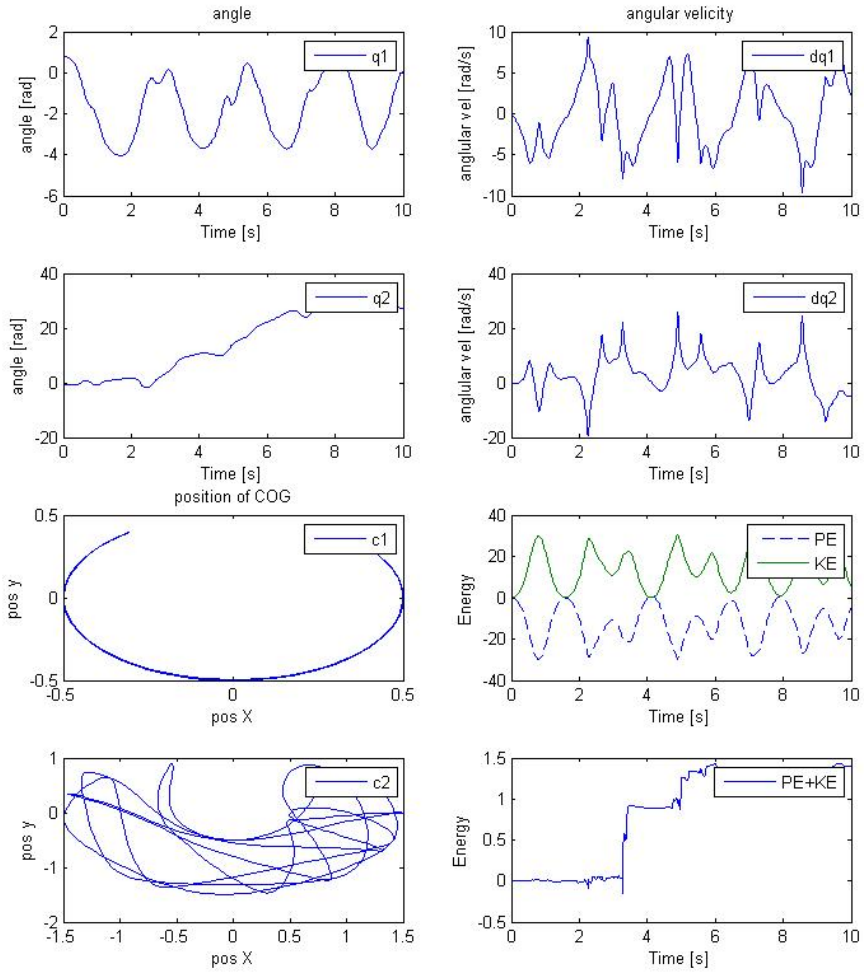


Figure 5.12: Resulting time, phase and energy plots of the system modeled in Matlab

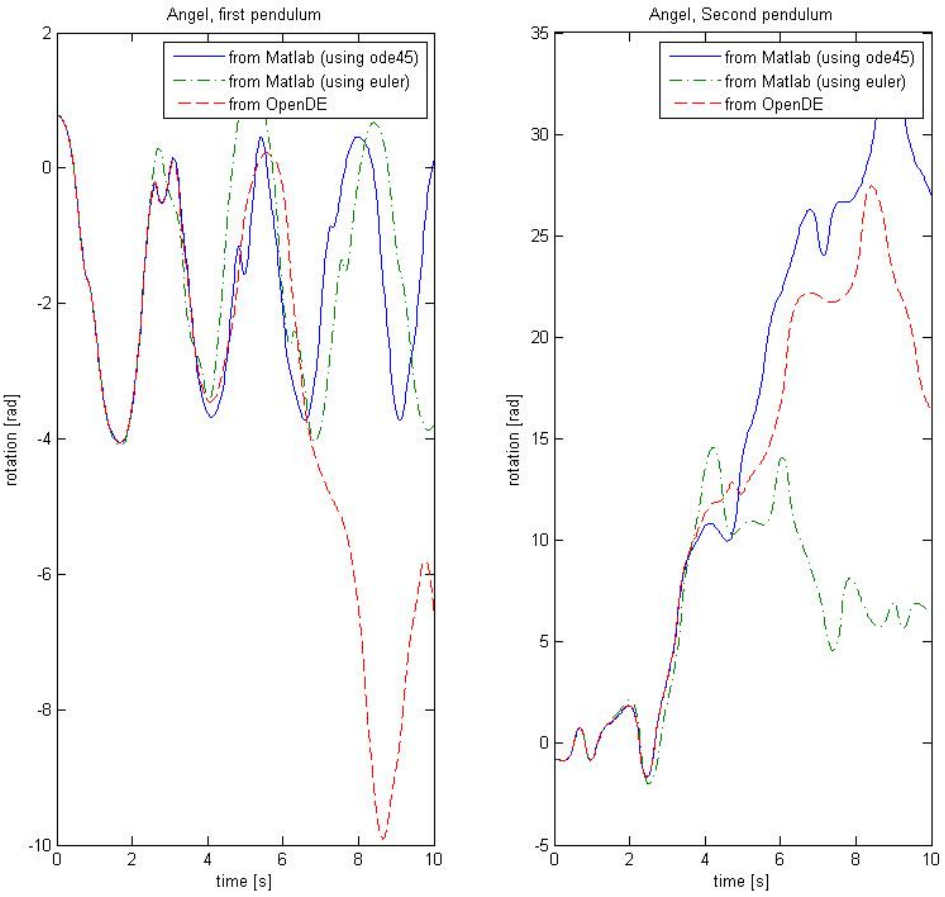


Figure 5.13: Comparison Of rotation angles from OpenDE and Matlab model

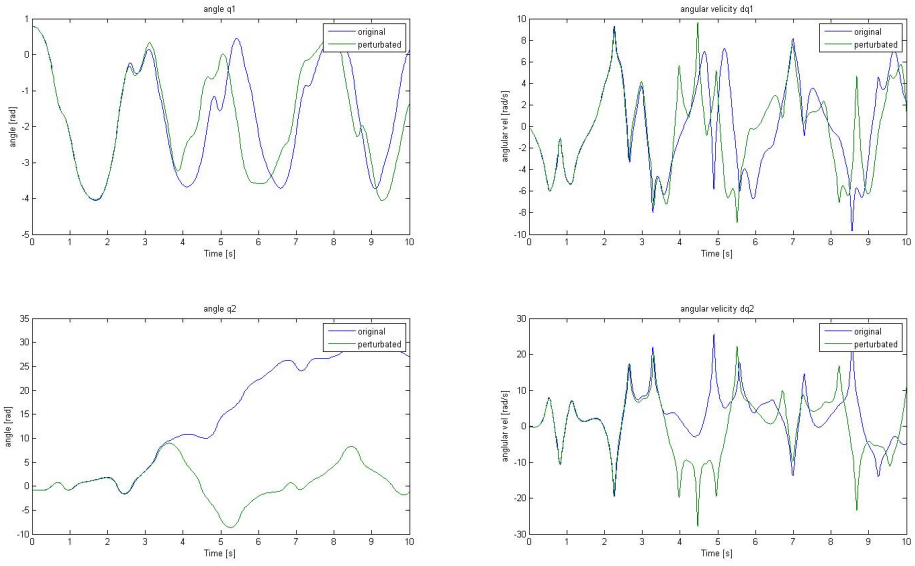


Figure 5.14: Comparison of pendulum state output for perturbed and unperturbed initial states. Both computed form the same Matlab model and integrated using Rung Kutta (4,5)

Chapter 6

Colliding Pendulum

6.1 Introduction

A single pendulum colliding with a wall was implemented in OpenDE and in Matlab (the code added to appendix: C.5 and **??**respectively). This chapter will briefly describe the method of implementing collision detection attributes in both matlab as in OpenDE.

6.2 Collision handling implementation in OpenDE

In OpenDE, ridged body object (so called 'dBody' object) is made up of two separate properties namely *mass* and *geometry*. This differs form the robotics toolbox which uses only mass properties and consequently has no collision handeling attributes.
These two properties are contained in separated objects, named 'dMassID' and 'dGeomID'. All the geometries that need to be checked for collision are added to a container. The following code illustrates how this may be implemented:

```
dSpaceID space  = dHashSpaceCreate(0);                  //create space list
dGeomID geom = dCreateBox(space,lx,ly,lz);              //creat box, put this in the 'space'
```

The model used in OpenDE differs for this reason slightly form that implemented in Matlab in that 'wall' (modeled as a box fixed to the inertia frame) needed to be position with a horizonal offset w.r.t. to the position of the joint to account for the geometry of the pendulum, as can be seen from figure 6.1.

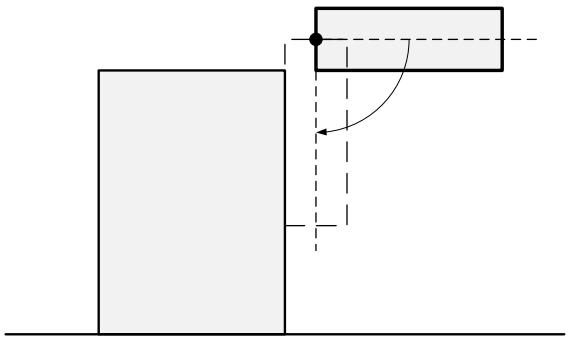


Figure 6.1: figure showing model implemented in OpenDE

The collision engine (software responsible for checking for collisions) keeps track of position of each object with respect to each other in a list. A check is made to see if any objects are touching each other before each simulation step is taken. This is doen with the 'dSpaceCollide' function, for example:

```
dSpaceCollide(space,0,&nearCollisionCallback); //  (collision detection)
```

Only objects which are located near to each other in the previous time step are checked for collision according to the the 3rd function input. This 3rd input named 'nearCollisionCallback' in the previous example, is a user defined function, for example:

```
static void nearCollisionCallback(void *data, dGeomID o1, dGeomID o2) {

//part 1) check if bodies are connecting
  dBodyID b1 = dGeomGetBody(o1), b2 = dGeomGetBody(o2);
  if (b1 && b2 && dAreConnectedExcluding(b1,b2,dJointTypeContact)) return;

//part 2) if not, add contact-joints to the bodies which are intersecting
  static const int N = 20;
  dContact contact[N];
  int n = dCollide(o1,o2,N,&contact[0].geom,sizeof(dContact));

//part 3) for each 'contact joint',define contact properties
  if (n > 0) {
    for (int i=0; i<n; i++) {
      contact[i].surface.mode = dContactBounce;
      contact[i].surface.bounce = 1.0;       //  (coefficient of restitution )
      contact[i].surface.bounce_vel = 0.0;

//part 4) add constraints to the system
    dJointID c = dJointCreateContact(world,contactgroup,&contact[i]);
      dJointAttach(c,b1,b2);
    }
  }
}
```

Note that other contact parameters such as fiction properties can be implemented in a similar fashion. Screen shots of the single pendulum simulation can be seen in figure 6.2.
A thorough explanation of how the collision handeling takes place can be found in [?].

6.3 Comparison with collision handling as implementation in Matlab

For the Matlab model the collision handeling functionality was implemented with the help of an exception handeling attribute of the solver. After stoping the integration when reaching a predetermined value (for this case for example, when $\theta = -pi$, the rotational velocity is reversed and the integration is restarted. Figure 6.3 shows that the OpenDE is detects the collision much later than it should (10 time steps into the collision, as can be seen from figure 6.4). The default EPR and CFM parameters (discussed in section 3.4) could be responsible for the delayed respons to the the collision detection. This could be investigated further by disabling these joint error reduction properties.

Implementation has shown that collision detection /response is a lot more simple and intuitive when doen in Matlab for this case study. An important thing to keep in mind here is that in this case

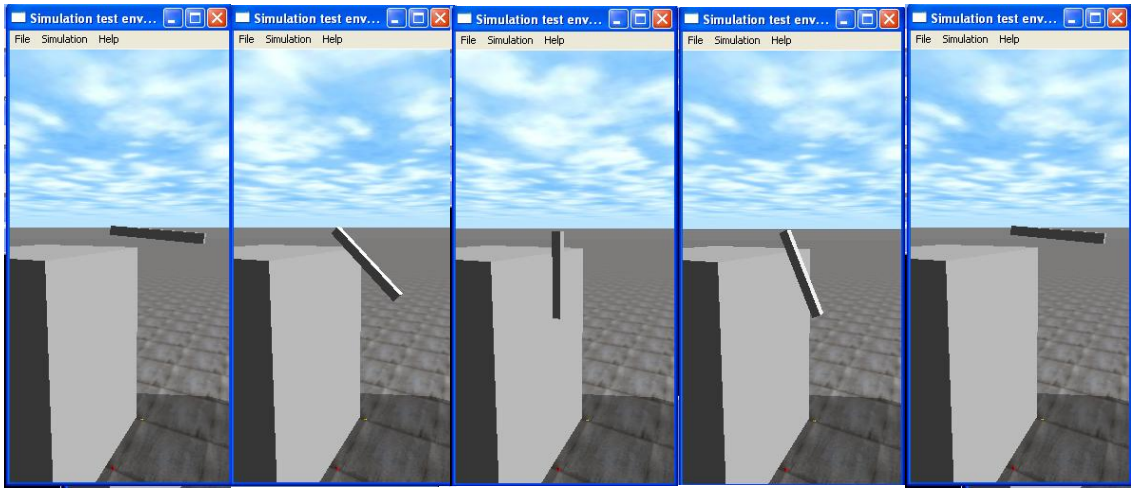


Figure 6.2: Successive screen shots of the single pendulum simulation

study collision detection was carried out between simple geometries where one of which is fixed to the world. Collision detection in a changing environment between multipel, complex geometries would be a far more difficult problem to solve in Matlab but would be implemented in more or less the same way in OpenDE. Another thing to consider is that collision detection has been described by [?] as the most computationally costly step of the simulation and Matlab has been described by [?] as being over 1000 times slower than C++. Assuming a given engineer would have the time and skills to solve this complex collision handeling problem in Matlab, the computational time necessary for such a simulation could likely make it very unpractical.

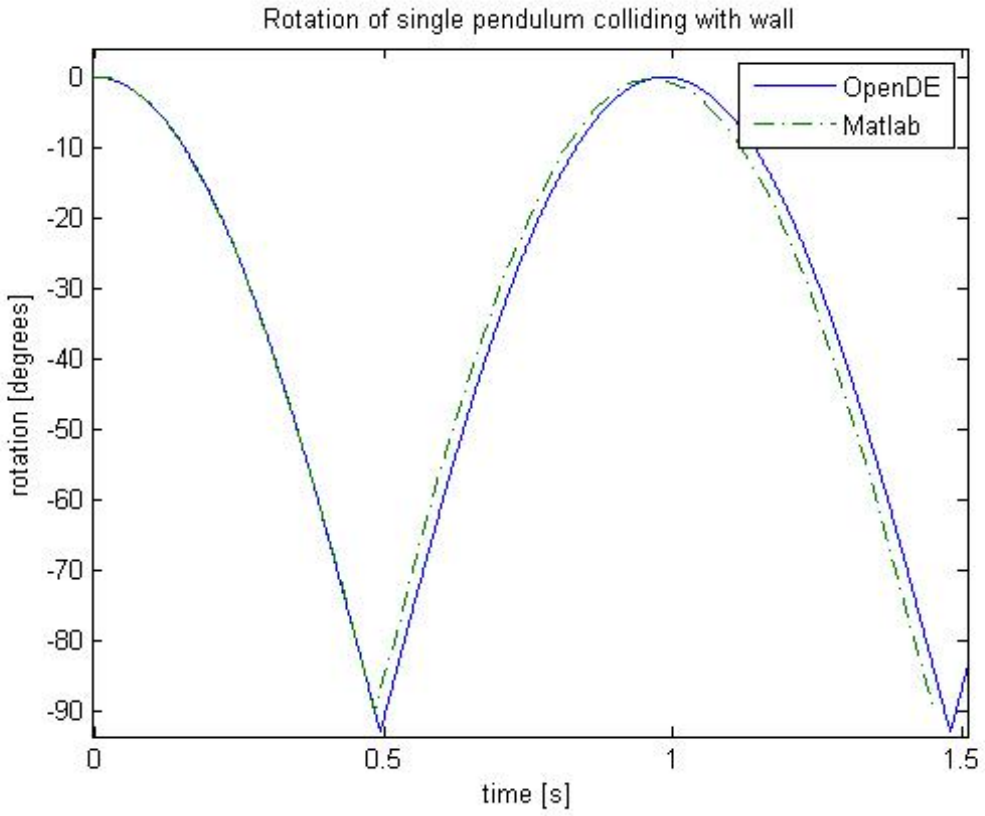


Figure 6.3: Comparison of Matlab and OpenDE simulation results

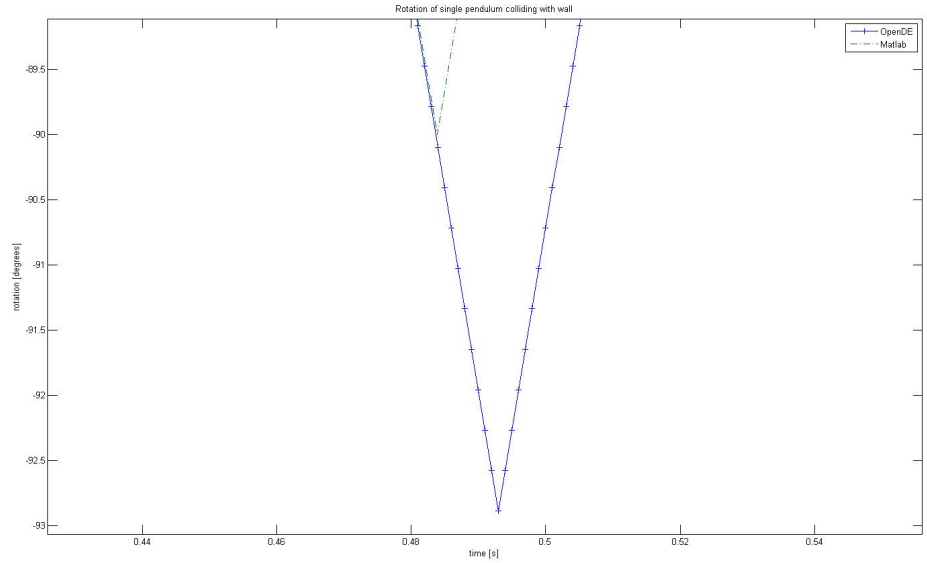


Figure 6.4: Comparison of Matlab and OpenDE simulation results at point of collision

Chapter 7

Conclusions and Recommendations

OpenDE proved more difficult to learn than expected, predominantly due to the though understanding of c++ programming language. A great benefit that Matlab has over C++ is that there is no need for declaring and defining variables before using them. This make the programming al lot more readable, less error prone and less time consuming to program. More specifically with respect to using the OpenDE library, this was found difficult and time consuming to install. Also documentation was found to be in many cases out of date and difficult to follow. Although the steep learning curve, required for mechanical engineers to learn c++, render it less suitable for simple models, the added difficulty would be less significant for the purpose of more complex models such as the Tulip model [**?**]. Furthermore it was found that the collision handeling attribute make is especially easy to use compared with Matlab for dynamically changing world models.

The fact that practically all robots are programmed in c code, does make it OpenDE a useful learning tool. Another practical application is to use OpenDE to make a robot simulator. This could be used to test for bugs in code before running then om a real robot which may cause damage.

The poor documentation mentioned earlier is a big problem from the perspective of a mechanical engineer who would be keen to know exactly what is being modeled in order to take into account any modeling error. After some research it was found that the articulated ridged body systems modeled OpenDE were found to contain many more states than necessary. Error feedback compensation for the position of the joints, an OpenDE attribute which compensates for numerical drift was shown to produce promising results.

In OpenDE, contacts are represented by contact points rather than contact surfaces. This simplification may be essential in order to keep computation time low enough for example for the purpose of robot controle. Which contact points as well as the number of contact points to use for optimally representing the physical system might be interesting subjects for future research.

The Euler solver used in OpenDE is not accurate enough for theoretical studies. As OpenDE is open source software it is possible and recommended to implement a more accurate solver. This solver would need to be fixed step solver (for example Fourth-Order Runge-Kutta method)in order to be compatible with the collision detection software.

Simulations preformed in OpenDE were shown to be computationally much faster than those in Maltab. As Matlab code is to be a lot more user-friendly with the exception of collision handeling implementation, combining the collision detection attributes with Matlab code though the use of mex files may be an good compromise for running simulations with dynamically changing world models.

Appendix A

Forward Kinematics And DH parameters

Coordinate frames are fixed to each each link to facilitate the description of their location. Denavit and Hartenberg proposed a matrix method of systematically assigning coordinate systems to each link of a serial manipulator. This convention, consisting of 4 parameters,(known as the DH-parameters), defines the geometric position of one link w.r.t the previous link.

- link length a_i the offset distance between the z_{i-1} and z_i axes along the x_i axis;
- link twist α_i the angle from the z_{i-1} axis to the z_i axis about the x_i axis;
- link offset d_i the distance from the origin of frame $i-1$ to the x_i axis along the z_{i-1} axis;
- joint angle θ_i the angle between the x_{i-1} and x_i axes about the z_{i-1} axis.

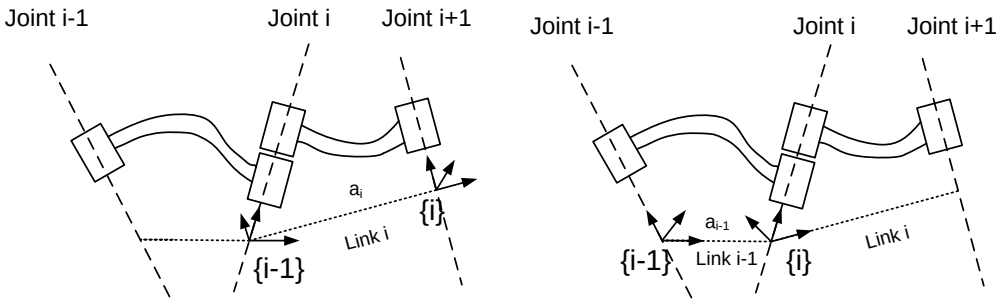
This became the standard for describing the kinematics for robotics. Later J. J. Craig. came up with a similar convention, known as the modified-DH parameters. Although this convention uses the same rules to define a_i,α_i,d_i and θ_i, the difference arise in the manner to which the joints are numbered. In the case of DH-formulism:

- Frame i has its origin along the axis of joint i+1

In the case of modified Denavit-Hartenberg (MDH) form:

- Frame i has its origin along the axis of joint i.

The two similar conventions give very different descriptions coordinate frame descriptions as can be seen in the following figures.



(a) standard DH form

(b) Modified DH form

Figure A.1: Two Different methods of assigning frames

Appendix B

Data Processing

For the OpenDE simulation, the rotation matrix describing the orientation of the bodies, global position and the rotational velocities were recorded, relative to the 3D cartesian coordinate system used. This coordinate system differed form the one used in the Matlab model as can be seen from figure B.1.[1]
Some data processing was needed to firstly extract the Euler angles (shown as θ_1 and θ_2 in figure B.1,A) from the rotation matrixes, and next to converted to relative coordinates,(shown as q_1 and q_2 in figure B.1B) for comparison.

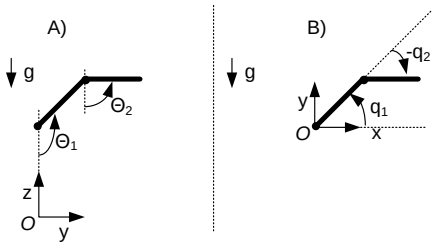


Figure B.1: The different conventions used for defining the inertial frame and pendulum rotations: A) OpenDE B)Matlab

Extraction of Euler angles, α, β, γ, where $R = R_z(\alpha)R_y(\beta)R_x(\gamma)$, is was doen as follows:
Express R in terms of α, β, γ,

$$R_z(\alpha)R_y(\beta)R_x(\gamma) = \begin{bmatrix} c\alpha & -s\alpha & 0 \\ s\alpha & c\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c\beta & 0 & s\beta \\ 0 & 1 & 0 \\ -s\beta & 0 & -c\beta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & c\gamma & -s\gamma \\ 0 & s\gamma & c\gamma \end{bmatrix} \quad \text{(B.1)}$$

$$= \begin{bmatrix} c\alpha c\beta & c\alpha s\beta - s\alpha c\gamma & c\alpha s\beta c\gamma + s\alpha s\gamma \\ s\alpha c\beta & s\alpha s\beta + c\alpha c\gamma & s\alpha s\beta c\gamma - c\alpha s\gamma \\ -s\beta & c\beta\gamma & c\beta c\gamma \end{bmatrix} \quad \text{(B.2)}$$

Equating this expression to an arbitrary expression of R (the rotation matrix recorded each time step),

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

[1] It was possible to define the direction of the gravity vector in OpenDE freely, and hence have orientated the pendulum to coincide with that of the Matlab model, following the DH convention. The orientation of the graphics output on the computer screen however is fixed according to the OpenGL standard (z-axis points upwards, x-axis point out of the screen). The gravity was therefor chosen to point (downwards) in the negative z direction, although the 'camera', or viewpoint of the simulation could have just as easily have ben rotated

This gives

$$\begin{array}{rcl} c\beta & = & \sqrt{r_{11}^2 + r_{21}^2} \\ s\beta & = & -r_{31} \end{array} \Rightarrow \beta = atan2(-r_{31}, \sqrt{r_{11}^2 + r_{21}^2}) \tag{B.3}$$

define constant: $k = c\beta$

$$\begin{array}{rcl} c\alpha & = & r_{11}/k \\ s\alpha & = & r_{12}/k \end{array} \Rightarrow \alpha = atan2(r_{12}/k, r_{11}/k) \tag{B.4}$$

$$\begin{array}{rcl} c\gamma & = & r_{33}/k \\ s\gamma & = & r_{32}/k \end{array} \Rightarrow \gamma = atan2(r_{32}/k, r_{33}/k) \tag{B.5}$$

(Note that is derivation gives the value between pi and -pi. To give the correct angle of rotation the signal was also unwrapped. We are only interested in the rotation about the x axis, $\theta_i = \gamma_i$. Converted to relative coordinates is simply

$$\begin{aligned} q_1 &= \theta_1 - \pi/2 \\ q_2 &= \theta_2 - \theta_1 + \pi/2 \end{aligned}$$

Appendix C

OpenDE Pendulum code

C.1 pendulum_main.cpp

```
//-------------------------------------------------------
// main
//-------------------------------------------------------
#include "pendulum_body.cpp"
#include <ctime>
#include <iostream>

static void setupDrawstuff();
static void start();
static void command (int cmd);
static void simLoop (int pause);

int main (int argc, char **argv)
{
  dInitODE();
  setupDrawstuff();
  time_t currentTime;
  time (&currentTime); // fill now with the current time

  myfile.open("pendulumData3.m");
  myfile << "%TIME_STAMP:  ";
  struct tm * ptm= localtime(&currentTime);
  myfile  << " % " << ((ptm->tm_mon)+1) << "/";
  myfile << ptm->tm_mday << "/";
  myfile << ptm->tm_year << " , " << ctime(&currentTime) << "\n\n";

  myfile <<"% recording the position, rotation matrix,vel,
    and omega of pendulum1 and pendulum2 \n\n";
  myfile <<"clear all;close all; clc\n";

  myfile <<"% simulation time parameters:"<<"\n";

  myfile <<"t_start = "<<t_start <<";"<<"t_step =
    "<<t_step<<";"<<"t_end = "<<t_end<<";\n\n";
```

```
  myfile <<"\n data=[";

//-- Create world. (Oject which stores RB's as well as world/global parameters
  world = dWorldCreate();
  dWorldSetGravity (world,0,0,GRAVITY); //-- Set the gravity. (Earth gravity: -9.81)
  dWorldSetCFM (world,CFM); //-- Set the CFM parameter.
  //-- Set the other ODE parameters
  dWorldSetContactMaxCorrectingVel (world,MAX_CORRECTING_VEL);
  dWorldSetContactSurfaceLayer (world,SURFACE_LAYER);

//-- Create the collision space. It contains all
// the geometries that should be check for collisions.
  space = dHashSpaceCreate (0);
//-- a*x + b*y + c*z = d, where (a,b,c) is normal vect. to plain
  dGeomID ground = dCreatePlane (space,0,0,1,0);
////-- Build the pendulum
  pendulumCreate(&pendulum1,&pendulum2,world,space);

  /** START THE SIMULATION        */
  //-- This is the main loop. All the callback functions will be called.
  dsSimulationLoop (argc,argv,400,300,&fn);
//-- The screen dimensions are passed as parameters
  myfile <<"];\n";
  myfile <<"t=data(:,1);\n";
  myfile <<"xyz1=data(:,(1+1:1+1+2));\n";
  myfile <<"R1=data(:,1+1+2+1:1+1+2+9);\n";
  myfile <<"xyz2=data(:,1+1+2+9+1:1+1+2+9+3);\n";
  myfile <<"R2=data(:,1+1+2+9+3+1:1+1+2+9+3+9);\n";
 myfile << "%Analise results\n";
 myfile << "close all\n";
 myfile << "tl=1; Rl=9; xl=3;vl=3;wl=3;\n";
 myfile << "t=data(:,tl);\n";
 myfile << "x1=data(:,[tl+1:tl+xl]);\n";
 myfile << "R1=data(:,[tl+xl+1:tl+xl+Rl]);\n";
 myfile << "v1=data(:,[tl+xl+Rl+1:tl+xl+Rl+vl]);\n";
 myfile << "w1=data(:,[tl+xl+Rl+vl+1:tl+xl+Rl+vl+wl]);\n";

 myfile << "R2=data(:,[tl+xl+Rl+vl+wl+1:tl+xl+Rl+vl+wl+Rl]);\n";
 myfile << "x1=data(:,[tl+xl+Rl+vl+wl+Rl+1:tl+xl+Rl+vl+wl+Rl+xl]);\n";
 myfile << "v1=data(:,[tl+xl+Rl+vl+wl+Rl+xl+1:tl+xl+Rl+vl+wl+Rl+xl+vl]);\n";
 myfile << "w2=data(:,[tl+xl+Rl+vl+wl+Rl+xl+vl+1:tl+xl+Rl+vl+wl+Rl+xl+vl+wl]);\n";
 myfile << "%test=data(:,[tl+xl+Rl+vl+wl+Rl+xl+vl+wl+1])\n";
 myfile << "n=length(data);\n";
 myfile << "for i=1:n\n";
 myfile << "    theta1(i)=atan2(R1(i,8),R1(i,9));\n";
 myfile << "   theta2(i)=atan2(R2(i,8),R2(i,9));\n";
 myfile << "end\n";

 myfile << "plot(t,theta1,t,theta2,'--')\n";
 myfile << "theta3=theta2-theta1;\n";
 myfile << "w3=w2-w1;\n";
 myfile << "subplot 211; plot(t,theta1,t,theta3,'--');
    title('joint angels'); xlabel('time [s]');
    ylabel('joint angle [rad]');legend('theta1','theta2');\n";
```

```
 myfile << "subplot 212; plot(t,w1(:,1),t,w3(:,1),'--');
    title('joint vel'); xlabel('time [s]');
    ylabel('joint vel[rad/s]');legend('theta1','theta2');\n";
 myfile << "figure \n";
 myfile << "subplot 211; plot(t,theta1,t,theta2,'--');
    title('body  angels'); xlabel('time [s]');
    ylabel('joint angle [rad]');legend('theta1','theta2');\n";
 myfile << "subplot 212; plot(t,w1(:,1),t,w2(:,1),'--');
    title('body rotational  vel wx'); xlabel('time [s]');
        ylabel('joint vel[rad/s]');legend('theta1','theta2');\n";


  myfile.close();
  /* END OF THE SIMULATION  */
  dSpaceDestroy (space); //-- Destroy de collision space
  dWorldDestroy (world); //-- Destroy the world!!!
  return 0;
}

/* Set the drawstuff parameters and callback functions.        */
static void setupDrawstuff(){
  fn.start = &start; //-- Set the Starting simulation Callback function
  fn.step = &simLoop;  //-- Set the Simulation step callback function
  fn.command = &command; //-- Set the key pressed callback function
  fn.version = DS_VERSION; //-- Others
  fn.stop = 0;
  fn.path_to_textures = (char *)"../../drawstuff/textures";
}

/*** Simulation start callback function. ***/
static void start()
{
   static float xyz[3] = {3.0,0.0, 1.0};//-- Camera position.
   //-- Orientation: Pan/Tilt/Roll.if (0,0,0)= dir +ve x axis
   static float hpr[3] = {180,0.0, 0.0};
   dsSetViewpoint (xyz,hpr); //-- Set camera position and orientation

   printf ("Keys: \n"); //-- Print the Menu for controlling the snake robot
   printf ("r: Reset Position \n");
   printf ("q: Quit\n");

}


/* Callback function that is called when a key is pressed  */
static void command (int cmd)
{

//-- When the 1 key is pressed, the body is set to its initial state
    if (cmd=='1') {
    //pendulumSetState (&pendulum1);
  }
  else if (cmd=='q') {
   dsStop(); //-- Finish the simulation and exit
```

```
  }
}


/* FUNCTIONS FOR PERFORMING THE SIMULATION                                    */

/*- Simulation loop. This function is called at every simulation step.      */
static void simLoop (int pause)
{
static double time=t_start;
//record output
//-- Read its position and orientation
  const dReal *pos1; pos1 = dBodyGetPosition (pendulum1.body);
  const dReal *R1; R1 = dBodyGetRotation (pendulum1.body);
  const dReal *omega1;    omega1 =dBodyGetLinearVel (pendulum1.body);
  const dReal *vel1; vel1 = dBodyGetAngularVel (pendulum1.body);

  myfile <<time << " , "<< pos1[0]<< " , "<<pos1[1]<< " , "<<pos1[2]<< " , ";
  myfile << R1[0] << " , " << R1[1] << " , " << R1[2] << " , "
         << R1[4] << " , " << R1[5] << " , " << R1[6] << " , "
 << R1[8] << " , " << R1[9] << " , " << R1[10]<< " , ";
  myfile << omega1[0]<< " , "<<omega1[1]<< " , "<<omega1[2]<< " , ";
  myfile << vel1[0]<< " , "<<vel1[1]<< " , "<<vel1[2]<< " , ";

//-- Read its position and orientation
  const dReal *pos2; pos2 = dBodyGetPosition (pendulum2.body);
  const dReal *R2; R2 = dBodyGetRotation (pendulum2.body);
  const dReal *omega2;    omega2 =dBodyGetLinearVel (pendulum2.body);
  const dReal *vel2; vel2 = dBodyGetAngularVel (pendulum2.body);

  myfile <<pos2[0]<< " , "<<pos2[1]<< " , "<<pos2[2]<< " , ";
  myfile << R2[0] << " , " << R2[1] << " , " << R2[2] << " , "
         << R2[4] << " , " << R2[5] << " , " << R2[6] << " , "
 << R2[8] << " , " << R2[9] << " , " << R2[10]<< " , ";
  myfile << omega2[0]<< " , "<<omega2[1]<< " , "<<omega2[2]<< " , ";
  myfile << vel2[0]<< " , "<<vel2[1]<< " , "<<vel2[2]<< " \n";

//step system
//  if (!pause) { //-- sincronise with wall clock
    dWorldStep(world,t_step); //-- Perform a simulation step. All the objects are updated
//  }
//Render Output
  Body_render(&pendulum1); //-- Draw the box on the screen
  Body_render(&pendulum2);
//end simulation after..
  static int k=0; k++;
  time+=k*t_step;
  if(time>t_end) dsStop(); //--stop when time>t_end
}
```

C.2 pendulum_body.cpp

```
//-------------------------------------------------------
// body
//-------------------------------------------------------

/* contains the functions needed for building the box  ODE model and drawing it*/
#include <ode/ode.h>
#include <drawstuff/drawstuff.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#include <iostream>//lib to print output to file
#include <fstream>//
using namespace std;

#include "pendulum_par.h"
#include "pendulum_body.h"
//-------------------------------------------------------


/*** Build the ODE body model.   ***/
void pendulumCreate(object *pendulum1,object *pendulum2, dWorldID world, dSpaceID space)
{
//Creat pendulum 1
  dMass m1;  //should this be global??
  pendulum1->body = dBodyCreate(world); //-- Create the body
  dMassSetBoxTotal (&m1, MASS1, W1, D1, H1); //-- Set its mass
  dBodySetMass (pendulum1->body,&m1);
  //-- Create its geometry and associate it to the body
  pendulum1->geom = dCreateBox (space, W1, D1, H1);
  dGeomSetBody (pendulum1->geom,pendulum1->body);

//set Orrintaion
  dMatrix3 R1;  dRFromAxisAndAngle(R1,1,0,0,ANGLE1);
  dBodySetRotation(pendulum1->body,R1);

//set position
  dVector3 bTop1={0.0,0.0,H1/2};  // body space vector to top pendulem 1
  dVector3 gTop1;   // global/world space vecter pointing to top of pendulum
  //from body to global coordinates
  dBodyGetRelPointPos(pendulum1->body,bTop1[0], bTop1[1], bTop1[2], gTop1);
  dBodySetPosition(pendulum1->body,(anchor1[0]-gTop1[0]),
    (anchor1[1]-gTop1[1]),(anchor1[2]-gTop1[2]));   //translate body (line up anchorVec

//add hinge 1
  dJointID hJoint1 = dJointCreateHinge(world,0);
  dJointAttach(hJoint1, pendulum1->body,0);
  dJointSetHingeAnchor (hJoint1, anchor1[0], anchor1[1], anchor1[2]);
  dJointSetHingeAxis(hJoint1,1,0,0);

// Creat pendulem 2
```

```
  dMass m2;
  pendulum2->body = dBodyCreate(world);
  dMassSetBoxTotal (&m2, MASS2, W2, D2, H2);
  dBodySetMass (pendulum2->body,&m2);
  pendulum2->geom = dCreateBox (space, W2, D2, H2);
  dGeomSetBody (pendulum2->geom,pendulum2->body);

// Set orientation
  dMatrix3 R2;  dRFromAxisAndAngle(R2,1,0,0,ANGLE2);
  dBodySetRotation(pendulum2->body,R2);
// set position
  //find world coordinates of ankor pos (bottem of pendulum 1)
  dVector3 bBottom1={0.0,0.0,-H1/2}; //vector pointing to bottem of 1st pendulum
  dVector3  anchor2; //anchor2 = bottom1
  //get position of 2nd joint, then translate pendulum to such that bottem + top line up
  dBodyGetRelPointPos(pendulum1->body,bBottom1[0],
  bBottom1[1],
  bBottom1[2], anchor2); //from body to global coordinates
  //get global coordinates of top of pendulum 2
  dVector3 bTop2={0.0,0.0,H2/2};
  dVector3  gTop2;  // vecter pointing to top of 2nd pendulum (in BODY coordinates)
   //get position of 2nd joint, then translate pendulum to such that bottem + top line up
  dBodyGetRelPointPos(pendulum2->body,bTop2[0],
  bTop2[1],
  bTop2[2], gTop2); //body to global coordinates
  //translate pendulum2
  dBodySetPosition(pendulum2->body,(anchor2[0]-gTop2[0]),
   (anchor2[1]-gTop2[1]),
   (anchor2[2]-gTop2[2]));  //translate body (line up anchorVec
//add joint 2
  dJointID  hJoint2 = dJointCreateHinge(world,0); //--Add hinge
  dJointAttach(hJoint2, pendulum1->body,pendulum2->body);
  dJointSetHingeAnchor (hJoint2, anchor2[0], anchor2[1], anchor2[2]);
  dJointSetHingeAxis(hJoint2,1,0,0);

//set rotation speed
  dBodySetAngularVel (pendulum1->body, angularVel1[0],
   angularVel1[1],
   angularVel1[2]);  //add angular vel?

  dBodySetAngularVel (pendulum2->body, angularVel2[0],
   angularVel2[1],
   angularVel2[2]);  //add angular vel?
}

/*** Draw the body on the screen. ***/
void Body_render(object *pendulum)
{
  dsSetTexture (DS_WOOD); //-- Set the body texture
  dsSetColor (1.0, 1.0, 0.0); //-- Set the color: Red, green and blue components
  drawGeom(pendulum->geom); //-- Draw the body
}

static void drawGeom (dGeomID g)
```

```
{
//referance cylinder
  const double pos_ref[3]={anchor1[0],anchor1[1],anchor1[2]};
  dMatrix3 R_ref;
  dRFromAxisAndAngle(R_ref,0,1,0,M_PI/2);     //rotate 90deg
  float length_ref = 1;
  float radius_ref = 0.01;

  //draw cylinder  (long /thin (initially Z-allined , through anchor pos
  dsDrawCylinderD(pos_ref,R_ref,length_ref,radius_ref);
//pendulum
  const dReal *pos; pos = dGeomGetPosition (g); //-- Read its position and orientation
  const dReal *R; R = dGeomGetRotation (g);
   //-- Get the type of geometry. In this example it should always be a box
  int type = dGeomGetClass (g);
  dVector3 sides;
  dGeomBoxGetLengths (g,sides);
  dsDrawBoxD (pos, R, sides);
 // myfile <<pos[0] << "," << R[0] <<  "," << R[1] << ","
            <<  R[2] << "," <<  R[3] << "," <<  R[4] << "," <<  R[5] <<  ","
                << R[6] <<  "," << R[7] <<  "," << R[8] <<  "," << R[9] <<  "," <<
                    R[10] <<  "," << R[11]<<"\n";
}
```

C.3 pendulum_body.h

```
//-------------------------------------------------------------------------------------------------
#ifndef __pendulum_body_h
#define __pendulum_body_h

#include <ode/ode.h>
#include <drawstuff/drawstuff.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>//lib to print output to file
#include <fstream>//
using namespace std;

#include "pendulum_par.h"
//-------------------------------------------------------------------------------------------------




/*-- body data structure      --*/
struct object {
  dBodyID body;       //-- The ODE body
  dGeomID geom;       //-- The body's geometry
};

void pendulumCreate(object *pendulum1,object *pendulum2, dWorldID world, dSpaceID space);
void Body_render(object *pendulum);
static void drawGeom (dGeomID g);



/*-- GLOBAL VARIABLES     */
static dWorldID world; //-- World identification
static dSpaceID space; //-- Collision space identification
static object pendulum1,pendulum2; //-- pendulums (mass+geom)

//dJointID hJoint;

ofstream myfile;
dsFunctions fn;




//-------------------------------------------------------------------------------------------------
#endif
```

C.4 pendulum_par.h

```
//-------------------------------------------
// par
//-------------------------------------------

#ifndef __pendulum_par_h
#define __pendulum_par_h

#include <ode/ode.h>
#include <drawstuff/drawstuff.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>//lib to print output to file
#include <fstream>//
using namespace std;

#include "pendulum_body.h"
//-------------------------------------------
/*-      PARAMETERS -*/

#define MASS1      1.0   //-- Body mass, in Kg.
#define H1         1.0   //-- Hight. In meters
#define W1         0.1   //-- Width. In meters
#define D1         0.1      //--Depth. In meters

#define MASS2      1.0   //-- Body mass, in Kg.
#define H2         1.0   //-- Hight. In meters
#define W2         0.1   //-- Width. In meters
#define D2         0.1      //--Depth. In meters

dVector3 anchor1={0.0,0.0,2.1};  // hing-ancher at 0,0,2

//-------------------------------------------------------------------------
// -    INITIAL STATES   ---
#define ANGLE1  3*M_PI/4 //BODY1
dVector3 angularVel1={0.0,0.0,0.0};
#define ANGLE2  M_PI/2
dVector3 angularVel2={0.0,0.0,0.0}; //BODY2
//-------------------------------------------------------------------------
// SIMULATION PARAMETERS
#define t_start 0
#define t_step      0.001
#define t_end 10000  //


/* ODE CONSTANTS     */      //collision const.
#define MU                  0.2 //-- friction coefficients (same in both directions
#define MU2                 MU
#define GRAVITY          -9.81 //-- Gravity constant (in meters/s^2)
#define CFM                 1e-5 //-- Other ODE parameters
#define ERP                 0.2
```

```
#define MAX_CORRECTING_VEL   0.1 //don't use this?
#define SURFACE_LAYER        0.0001 //tollerance1: joint error considered zero
#define BOUNCE               0.9     //v2=0.9v1 (v2 after collision...
#define BOUNCE_VEL           0.1        //tollerance2: min v to allow bounce
#define SOFT_CFM             0.01
#define MAX_CONTACTS         4




//-------------------------------------------------------------------------
#endif
```

C.5 SinglePendulum.cpp

```
//#include <stdio.h>
//#include <stdlib.h>

#include <ode/ode.h>
#include <drawstuff/drawstuff.h>
#include <iostream>
#include<fstream>

using namespace std;
#ifdef dDOUBLE
#define dsDrawBox dsDrawBoxD
#endif
#define NUM 2

dWorldID      world;
dSpaceID      space;
dGeomID       ground;
dGeomID       wall;
dJointGroupID contactgroup;
dJointID      joint[NUM];
dsFunctions   fn;

ofstream myfile;

typedef struct {
  dBodyID body;
  dGeomID geom;
} MyObject;

MyObject rlink[NUM];
dReal  THETA[NUM] = {0.0};


void  makeDP()
{
  dMass mass;
  dReal x[NUM]      = {0.5, 1.5};  // pos cog?
  dReal y[NUM]      = {0.00, 0.00};  //
  dReal z[NUM]      = {3, 3};  //

  dReal lx[NUM] = {1.0, 1.0};  //   box dimentions
  dReal ly[NUM] = {0.10, 0.10};  //
  dReal lz[NUM] = {0.1, 0.1};  //

  dReal weight[NUM] = {1.00, 1.00};  //


  dReal c_x[NUM]    = {0.00, 1.00};  // constraint postition
  dReal c_y[NUM]    = {0.00, 0.00};  //
  dReal c_z[NUM]    = {0.00, 3.00};  //

  dReal axis_x[NUM] = {0, 0};              // rot. axis
```

```
  dReal axis_y[NUM] = {1, 1};              //
  dReal axis_z[NUM] = {0, 0};              //


  for (int i = 0; i < NUM; i++) {
    rlink[i].body = dBodyCreate(world);
    dBodySetPosition(rlink[i].body, x[i], y[i], z[i]);
    dMassSetZero(&mass);
    dMassSetBoxTotal(&mass,weight[i],lx[i],ly[i],lz[i]);
    dBodySetMass(rlink[i].body, &mass);
    rlink[i].geom = dCreateBox(space,lx[i],ly[i],lz[i]);
    dGeomSetBody(rlink[i].geom,rlink[i].body);
  }

  joint[0] = dJointCreateFixed(world, 0);
  dJointAttach(joint[0], rlink[0].body, 0);
  dJointSetFixed(joint[0]);

  for (int j = 1; j < NUM; j++) {
    joint[j] = dJointCreateHinge(world, 0);
    dJointAttach(joint[j], rlink[j].body, rlink[j-1].body);
    dJointSetHingeAnchor(joint[j], c_x[j], c_y[j], c_z[j]);
    dJointSetHingeAxis(joint[j], axis_x[j], axis_y[j],axis_z[j]);
  }
}


void drawLinks()
{
   dVector3 sides;

   for (int i = 0; i < NUM; i++ ) {
 dGeomBoxGetLengths (rlink[i].geom, sides);
     dsDrawBox(dBodyGetPosition(rlink[i].body), dBodyGetRotation(rlink[i].body),sides);
   }
   dGeomBoxGetLengths (wall, sides);
   dsDrawBox(dGeomGetPosition(wall),dGeomGetRotation(wall),sides);
}

static void nearCallback(void *data, dGeomID o1, dGeomID o2) {
  dBodyID b1 = dGeomGetBody(o1), b2 = dGeomGetBody(o2);
  if (b1 && b2 && dAreConnectedExcluding(b1,b2,dJointTypeContact)) return;

  static const int N = 20;
  dContact contact[N];
  int n = dCollide(o1,o2,N,&contact[0].geom,sizeof(dContact));
  if (n > 0) {
    for (int i=0; i<n; i++) {
      contact[i].surface.mode = dContactBounce;
      contact[i].surface.bounce = 1.0;
      contact[i].surface.bounce_vel = 0.0;

    dJointID c = dJointCreateContact(world,contactgroup,&contact[i]);
      dJointAttach(c,b1,b2);
```

```
    }
  }
}



void start()
{
  float xyz[3] = {    0.0f, -5.0f, 3.0f};
  float hpr[3] = { 90.0f, 0.0f, 0.0f};
  dsSetViewpoint(xyz, hpr);
}


void command(int cmd)
{
  switch (cmd) {

    case 'q': exit(0); break;
  }
}


void simLoop(int pause)
{
  static int k=0; k++;
  if (k==5000) exit(0);

  dReal angle = dJointGetHingeAngle(joint[1]);
  myfile<< angle <<"\n";

  dSpaceCollide(space,0,&nearCallback); //  (collision detection)
  dWorldStep(world, 0.001);
  dJointGroupEmpty(contactgroup);       //  (empty jointgroup)
  drawLinks();

}


void setDrawStuff()
{
  fn.version = DS_VERSION;
  fn.start   = &start;
  fn.step    = &simLoop;
  fn.command = &command;
  fn.path_to_textures = "../textures_standard";
}

int main(int argc, char **argv)
{
  dInitODE();
  setDrawStuff();
```

```
  myfile.open("linksData.txt");
  myfile<<"q=[";

  world        = dWorldCreate();
  space        = dHashSpaceCreate(0);
  contactgroup = dJointGroupCreate(0);
  wall    = dCreateBox(space, 1.8, 2.0, 5.5);
  dWorldSetGravity(world, 0, 0, -9.8);
  makeDP();
  dsSimulationLoop(argc, argv, 640, 480, &fn);

  myfile<<"];";
  myfile.close();

  dSpaceDestroy(space);
  dWorldDestroy(world);
  dCloseODE();
  return 0;
}
```

Appendix D

Matlab Code

D.1 DP_EOM.m

```
%Derrivation of EOM Pendulum
clear all;clc; close all;
syms th1 Dth1 th2 Dth2 d3 Dd3 pi

l1=1; l2=1;
r1c1=[-l1/2 0 0]'; r2c2=[-l2/2 0 0]';
h1=1;w1=0.1;d1=0.1;m1=1;
h2=1;w2=0.1;d2=0.1;m2=1;
I1=diag([1/12*m1*(w1^2+d1^2),1/12*m1*(h1^2+d1^2), 1/12*m1*(h1^2+w1^2)]);
I2=diag([1/12*m2*(w2^2+d2^2),1/12*m2*(h2^2+d2^2), 1/12*m2*(h2^2+w2^2)]);

q=[th1; th2 ];
Dq=[Dth1;Dth2];

alf=[0; 0];
a=[l1; l2];
d=[0; 0];
th=[q(1); q(2)];

sigma=[0; 0 ];

x=[]; R=[]; A=[];
for ii=1:length(a),
    x{ii}=[a(ii)*cos(th(ii)); a(ii)*sin(th(ii)); d(ii)];
    R{ii}=[cos(th(ii)), -cos(alf(ii))*sin(th(ii)),  sin(alf(ii))*sin(th(ii))
           sin(th(ii)),  cos(alf(ii))*cos(th(ii)), -sin(alf(ii))*cos(th(ii))
                     0,              sin(alf(ii)),              cos(alf(ii))];
      A{ii}=[        R{ii},  x{ii}
           [0 0 0],      1];
end

H01=A{1}; H02=A{1}*A{2};
z00=[0 0 1]'; O00=[0 0 0]';R01=H01(1:3,1:3);
z01=H01(1:3,3); O01=H01(1:3,4); R02=H02(1:3,1:3);
```

```
r0c1_h=H01*[r1c1;1];r0c1=r0c1_h(1:3)
r0c2_h=H02*[r2c2;1];r0c2=r0c2_h(1:3)

Jv_c1=[cross(z00,(r0c1-O00)),[0 0 0]']
Jv_c2=[cross(z00,(r0c2-O00)),cross(z01,(r0c2-O01))]

Jw1=[z00,[0 0 0]'];
Jw2=[z00,z01];

DT=simple(m1*Jv_c1.'*Jv_c1+ m2*Jv_c2.'*Jv_c2);
DR=(Jw1.'*R01*I1*R01.'*Jw1+Jw2.'*R02*I2*R02.'*Jw2);
D=simple(DT+DR);
D_q=D

for kk=1:length(a);
    for ii=1:length(a);
        for jj=1:length(a);
            c_q{ii,jj,kk}=simple(0.5*( diff(D(kk,jj),q(ii)) + diff(D(kk,ii),q(jj)) - diff(D(ii,jj
        end
    end
end

% Matrix C_q appears in the term representing Coriolis and centripetal effects
C_q=0*D;
for kk=1:length(a);
    for jj=1:length(a);
        [kk,jj];
        for ii=1:length(a),
            C_q(kk,jj)=C_q(kk,jj)+c_q{ii,jj,kk}*Dq(ii);
        end
    end
end
C_q

gv=[0 9.81 0].';    %  g pointing in (-)y dir!
P1=simple(m1*gv.'*r0c1);
P2=simple(m2*gv.'*r0c2);

P=simple(P1+P2);

g_q=[];
for ii=1:length(a),
    g_q=[g_q; simple(diff(P,q(ii)))];
end
g_q

save DP_EOM D_q C_q g_q
```

D.2 xD_DP.m

```
function xD=xD_DP(t,x)
th1=x(1);
th2=x(2);
Dth1=x(3);
Dth2=x(4);
%load DP_EOM.mat
D_q =[    1001/600+cos(th2), 1/2*cos(th2)+401/1200
      1/2*cos(th2)+401/1200,              401/1200];



C_q =[                   -1/2*sin(th2)*Dth2, -1/2*sin(th2)*Dth1-1/2*sin(th2)*Dth2
                          1/2*sin(th2)*Dth1,                                    0];



g_q = [ 2943/200*cos(th1)+981/200*cos(th1+th2)
                    981/200*cos(th1+th2)  ];

q=[th1; th2];
qD=[Dth1;Dth2];

qDD=inv(D_q)*(-C_q*qD-g_q);

 xD=[qD;qDD];
```

D.3 simDP.m

```
close all;clear all; clc;
q0=[pi/4 -pi/4]';
qD0=[0 0]';
x0=[q0;qD0];
tic
[T x]=ode45(@xD_DP,[0 10],x0);
toc

n=length(T);
Q=x(:,[1,2]);
QD=x(:,[3,4]);

load robot_object
% plot(RR,Q)
% view([0,0, 1])
% figure

% Calculated KE / PE / COG
PE=zeros(1,n);
KE=zeros(1,n);
gv=[0 9.81 0].'; m1=1;m2=1;
P0=-10.4
for i=1:n
    th1=x(i,1);
    th2=x(i,2);
    Dth1=x(i,3);
    Dth2=x(i,4);
    D_q =[    1001/600+cos(th2), 1/2*cos(th2)+401/1200
          1/2*cos(th2)+401/1200,              401/1200];
    r0c1 =[ 1/2*cos(th1)  1/2*sin(th1) 0].';
    r0c2 =[ 1/2*cos(th1)*cos(th2)-1/2*sin(th1)*sin(th2)+cos(th1) 1/2*sin(th1)*cos(th2)+1/2*cos(th
    PE(i)=m1*gv.'*r0c1+m2*gv.'*r0c2+P0;
    KE(i)=1/2*[Dth1 Dth2]*D_q*[Dth1 Dth2]';
    c1(i,:)=r0c1';
    c2(i,:)=r0c2';
    icon(i)=inv(cond(D_q));
end

figure  %plot KE/PE
subplot 121; plot(T,PE,'--',T,KE);xlabel('Time [s]');ylabel('Energy');legend('PE','KE')
subplot 122; plot(T,PE+KE,T,icon,'--');xlabel('Time [s]');ylabel('Energy');legend('PE+KE','Invers

figure %plot states q/dq and cog
subplot 231; plot(T,x(:,1));xlabel('Time [s]');ylabel('angle [rad]');legend('q1');title('angle')
subplot 232; plot(T,x(:,3));xlabel('Time [s]');ylabel('anglular vel [rad/s]');legend('dq1');title
subplot 233; plot(c1(:,1),c1(:,2));xlabel('Time [s]');ylabel('pos y');xlabel('pos X');legend('c1'

subplot 234; plot(T,x(:,2));xlabel('Time [s]');ylabel('angle [rad]');legend('q2')
subplot 235; plot(T,x(:,4));xlabel('Time [s]');ylabel('anglular vel [rad/s]');legend('dq2')
subplot 236; plot(c2(:,1),c2(:,2));xlabel('Time [s]');ylabel('pos y');xlabel('pos X');legend('c2'
```

D.4 DP_RT.m

```
%Derrivation of EOM Pendulum using the ROBOTICS TOOLBOX
clear all;clc; close all;

h1=1;d1=0.1;w1=0.1;m1=1;                    %link dimentions (h_x by b_y by w_z) and mass
h2=1;d2=0.1;w2=0.1;m2=1;
r1c1=[-h1/2 0 0]'; r2c2=[-h2/2 0 0]';       %link COG wrt link coordinate frame

%%%%%%%%%%%%%%%% DNY matrix parameters:
   %  kinamatic prop.
alpha=[0; 0];                               %     link twist angle
A=[h1; h2];                                 %   link length
D=[0; 0];                                   % link offset distance
theta=[0; 0];                               % link rotation angle
sigma=[0; 0 ];                              % joint type, 0 for revolute, non-zero for prismati
   %  Dynamic prop
mass=[m1;m2];                               % mass of the link
rx=[r1c1(1);r2c2(1)];                       % link COG with respect to the link coordinate fram
ry=[r1c1(2);r2c2(2)];                       %
rz=[r1c1(3);r2c2(3)];                       %

Ixx=[1/12*m1*(w1^2+d1^2);1/12*m2*(w2^2+d2^2)];% elements of link inertia tensor about the link CO
Iyy=[1/12*m1*(h1^2+d1^2);1/12*m2*(h2^2+d2^2)];%
Izz=[1/12*m1*(h1^2+w1^2);1/12*m2*(h2^2+w2^2)];%
Ixy=[0;0];                                  %
Iyz=[0;0];                                  %
Ixz=[0;0];                                  %

Jm=[0;0];                                   % armature inertia
G=[0;0];                                    % reduction gear ratio. joint speed/link speed
B=[0;0];                                    % viscous friction, motor refered
TcP=[0;0];                                  % coulomb friction (positive rotation), motor refer
TcN=[0;0];                                  % coulomb friction (negative rotation), motor refer
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%   DYN(1  2 3  4     5     6    7  8  9  10  11  12  13  14  15 16 17 18 19 20 )
DYN=[alpha,A,D,theta,sigma,mass,rx,ry,rz,Ixx,Iyy,Izz,Ixy,Iyz,Ixz,Jm,G,B,TcP,TcN];
RR=robot(DYN);                              %  Creat robot object  (NB: grav=[0 0 9.81])
RR.gravity = [0;9.81;0];                    %  Change gravity vector to x dir
%showlink(RR)                               %  Display link properties

save robot_object RR                        %  load in pendulumStateVelocity func.

Q0=[pi/4 -pi/4]';                          %  Initial Position
QD0=[0 0]';
%plot(RR,Q')                                %  Visulise Robot in initial configuration
%drivebot(RR)                               %  Play with config. par.

%Preform Simulation:
    % loop: for time=t_start to t_end
    %0) Difine state x=[Q;QD],
    %1) Compute forward dynamics (joint accelleration, QDD) -->QDD = ACCEL(ROBOT, [Q QD TORQUE])
    %2) Intergrate state velocity xD=[QD;QDD] to get niew state
```

```
% Above steps implimented in fdyn function (using ode45 intergration):
%[T Q QD] = FDYN(ROBOT, T0, T1, TORQFUN, Q0, QD0)
tic
[T Q QD] = fdyn(RR, 0, 10, [0 0]', [pi/4 -pi/4]', [0 0]');
toc  %before mexfile: Elapsed time is 16.085133 seconds.


Fourth-Order Runge-Kutta
%n=length(T)                          %  In real time  iff (detlaT(i)-toc) > 0
% detlaT=T(2:n)-T(1:(n-1));
% for i=1:n
%     tic;
%     plot(RR,Q(i,:))
%     toc;
%     pause(detlaT(i)-toc)
% end

x=[Q,QD];
n=length(x);
PE=zeros(1,n);
KE=zeros(1,n);
gv=[0 9.81 0].'; m1=1;m2=1;
P0=-15.31+4.905; %energy constant so that ke+pe=0
for i=1:n
    q1=x(i,1);
    q2=x(i,2);
    dq1=x(i,3);
    dq2=x(i,4);
    PE(i)=cos(q1)*1/2+cos(q1)+1/2*cos(q1+q2);
    M=inertia(RR,[q1,q2]);
    KE(i)=1/2*[dq1 dq2]*M*[dq1 dq2]';

    r0c1 =[ 1/2*cos(q1)  1/2*sin(q1) 0].';    %from DP_EOM
    r0c2 =[ 1/2*cos(q1)*cos(q2)-1/2*sin(q1)*sin(q2)+cos(q1) 1/2*sin(q1)*cos(q2)+1/2*cos(q1)*sin(q
    PE(i)=m1*gv.'*r0c1+m2*gv.'*r0c2+P0;
    c1(i,:)=r0c1';
    c2(i,:)=r0c2';
    Jq=jacobn(RR,[q1,q2]);  E=eig(Jq'*Jq); Yosh(i)=min(E)/max(E);
end

subplot 121; plot(T,PE,'--',T,KE);legend('PE','KE')
subplot 122; plot(T,PE+KE,T,Yosh,'--');legend('PE+KE','inv Cond no. (JqTJq)')
figure
subplot 231; plot(T,x(:,1));xlabel('Time [s]');ylabel('angle [rad]');legend('q1');title('angle')
subplot 232; plot(T,x(:,3));xlabel('Time [s]');ylabel('anglular vel [rad/s]');legend('dq1');title
subplot 233; plot(c1(:,1),c1(:,2));xlabel('Time [s]');ylabel('pos y');xlabel('pos X');legend('c1'
subplot 234; plot(T,x(:,2));xlabel('Time [s]');ylabel('angle [rad]');legend('q2')
subplot 235; plot(T,x(:,4));xlabel('Time [s]');ylabel('anglular vel [rad/s]');legend('dq2')
subplot 236; plot(c2(:,1),c2(:,2));xlabel('Time [s]');ylabel('pos y');xlabel('pos X');legend('c2'
```

D.5 SinglePendulum.m

```
function SP_Sim
close all;clear all; clc;

x0=[0;0];
options_c = odeset('Events',@collision_detection);
tic
[T1 x1]=ode45(@xD_SP,[0 10],x0,options_c);
toc
x0=[x1(end,1);-x1(end,2)]
T0=T1(end)
[T2 x2]=ode45(@xD_SP,[T0 T0+10],x0,options_c);

q1=x1(:,1)*180/pi;
q2=x2(:,1)*180/pi;
%qd=x(:,2);
plot(T1,q1);hold
plot(T2,q2)

Tm=[T1;T2];
qm=[q1;q2];

function [value,isterminal,direction] = collision_detection(t,s)
%---------------------------------------------------------------
%floor detection
%---------------------------------------------------------------
%stop integration when pendulum touches wall (traveling in neg. dir.)

value = s(1)+pi/2;     % detect height = radius of ball
isterminal = 1;   % stop the integration
direction = -1;   % negative direction

function xD=xD_SP(t,x)
%---------------------------------------------------------------
%EOM  pendulum EOM
%---------------------------------------------------------------
th1=x(1);
Dth1=x(2);

D_q =401/1200;
C_q = 0 ;
g_q = 981/200*cos(th1);

q=[th1 ];
qD=[Dth1];

qDD=inv(D_q)*(-C_q*qD-g_q);

 xD=[qD;qDD];
```