ISSN 0280-5316
ISRN LUTFD2/TFRT--5738--SE

A Spherical Pendelum Modeling & Control

Marcel Meerstetter

Department of Automatic Control
Lund Institute of Technology
January 2005

Department of Automatic Control **Lund Institute of Technology** **Box 118** **SE-221 00 Lund Sweden**	*Document name* MASTER THESIS *Date of issue* January 2005 *Document Number* ISRN LUTFD2/TFRT--5738--SE
Author(s) Marcel Meerstetter	*Supervisor* Karl-Erik Årzén at LTH Prof. Morari at ETH in Zurich. *Sponsoring organization*

Title and subtitle

A Spherical Pendulum Modeling & Control (Modellering och reglering av en sfärisk pendel).

Abstract

The goal of this semester project was to set up the mathematical model for the simplified process of the spherical pendulum, develop a simulation model of the process in Modelica that could then also be animated in 3D, and finally, design a controller for the process. The simplification is that the pendulum cannot turn about its own axis. For the Modelica model, a Pendulum Library, with the essential parts, was created that could then just be selected and connected to assemble the desired model. Additionally, the parts from the Modelica MultiBody Library were simplified so that the user does not have to calculate various vectors and inertia tensors. With this Pendulum Library, a two, three and four wheel variation of the spherical pendulum was built, of which finally the three wheel model was of greatest interest. To verify the correctness of the Modelica model, it was linearized and its state-space matrices were compared to the ones resulting from linearizing the equivalent mathematical model. The comparison yielded only slight deviations between the two models, allowing the conclusion that the Modelica model is physically correct. For the controller, a state-feedback controller was implemented. Using Matlab, the L vector for the controller was calculated. A variety of different L vectors corresponding to different pole placements as well as various reference signals were tested. The results were as follows. To achieve the most ideal reference tracking with the least error and actuator effort,

- The poles should have the same frequency as the natural frequency of the spherical pendulum.
- Increasing the damping of the poles beyond 45° decreases the error minimally.
- The frequency of the reference trajectory to track should also be equal to the natural frequency of the spherical pendulum.
- The amplitude of the reference trajectory should not be too large since the model has been linearized around the stable equilibrium of the spherical pendulum.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title 0280-5316		*ISBN*
Language English	*Number of pages* 50	*Recipient's notes*
Security classification		

Table of Contents

Illustration Index

Index of Tables

Abstract

The goal of this semester project was to set up the mathematical model for the simplified process of the spherical pendulum, develop a simulation model of the process in Modelica that could then also be animated in 3D, and finally, design a controller for the process. The simplification is that the pendulum cannot turn about its own axis. For the Modelica model, a Pendulum Library, with the essential parts, was created that could then just be selected and connected to assemble the desired model. Additionally, the parts from the Modelica MultiBody Library were simplified so that the user does not have to calculate various vectors and inertia tensors. With this Pendulum Library, a two, three and four wheel variation of the spherical pendulum was built, of which finally the three wheel model was of greatest interest. To verify the correctness of the Modelica model, it was linearized and its state-space matrices were compared to the ones resulting from linearizing the equivalent mathematical model. The comparison yielded only slight deviations between the two models, allowing the conclusion that the Modelica model is physically correct. For the controller, a state-feedback controller was implemented. Using Matlab, the L vector for the controller was calculated. A variety of different L vectors corresponding to different pole placements as well as various reference signals were tested. The results were as follows. To achieve the most ideal reference tracking with the least error and actuator effort,

- The poles should have the same frequency as the natural frequency of the spherical pendulum.
- Increasing the damping of the poles beyond 45° decreases the error minimally.
- The frequency of the reference trajectory to track should also be equal to the natural frequency of the spherical pendulum.
- The amplitude of the reference trajectory should not be too large since the model has been linearized around the stable equilibrium of the spherical pendulum.

Introduction

The spherical pendulum is actually the most simple pendulum to build. One only needs a string and a bob attached to its end and has a simple version of the spherical pendulum. The pop will swing around as if it is moving on the surface of a sphere whose center is the opposite endpoint of the string and has a radius equal to the length of the string. Constructing a pendulum that is only able to swing in one direction is more complex since it requires eliminating the movement in the other dimension. However, when viewed from a control point of view, the scenario changes. Building and controlling a pendulum, that is only able to swing about one axis with an inertia wheel attached to its end to control its motion, is the simplest to realize. The situation becomes more complex for the spherical pendulum. In this setup, there are two categories. One is where the pendulum will be able to turn about its own axis and the other is where this movement is prohibited. The latter is the easier of the two to control.

For this project, the goal was to setup a mathematical model for the spherical pendulum with no rotation about its own axis. Further, a model of this pendulum was to be created with the modeling language Modelica that could then be visualized as a 3D animation. Finally, a controller should be developed that will control the motion of this pendulum to follow a desired trajectory.

The following chapters describe the proceedings and achieved results of the project.

- Chapter 1 – The Models: A description of the Modelica components that were created and explanations to the models built.
- Chapter 2 – Model Verification: The mathematical verification of the Modelica Model
- Chapter 3 – The Controller: details how the controller was designed and tuned as well as presenting the results of the simulations.

Chapter 1 - The Models

In the beginning stages of the project, several aspects of the spherical pendulum were still unclear. These included the question of how many inertia wheel the pendulum should actually have and whether or not it should be able to turn on its own axis. To accommodate for these unknowns, the Pendulum Library was created. This custom made library contains all the essential building blocks of the spherical pendulum, including the controller. The parts are very flexible and easily adjusted through a few parameters. This way, it becomes quite easy to model virtually any desired arrangement of the spherical pendulum using the same basic building blocks. An additional advantage to the Pendulum Library is that if an essential change has to be made to one of the components, it only needs to be applied in the library and all models will automatically be updated with this new change.

Using this Pendulum Library, three version of the spherical pendulum were built: a two, three, and four wheel model. For each variation, there are two versions. One without the controller where the torques are directly connected to input ports and the pendulum angles are connected to output ports. This model was used to linearize the process around the stable equilibrium and then calculate the feedback gain values in Matlab. Its filename includes the word "**input**". The other model is the one with the controller, where the inputs have been replaced with the reference signal source and the outputs have been removed. The filename of this model contains the word **"controller"**.

Both model variations contain the following components: **InertialSystem**, **Joint_2D**, **Rod**, **MotorPlacebo**, **Revolute Joint**, **Torque**, **InertiaWheel**, **AngleSensor** and **SpeedSensor** (angular) (see Ill. 1).

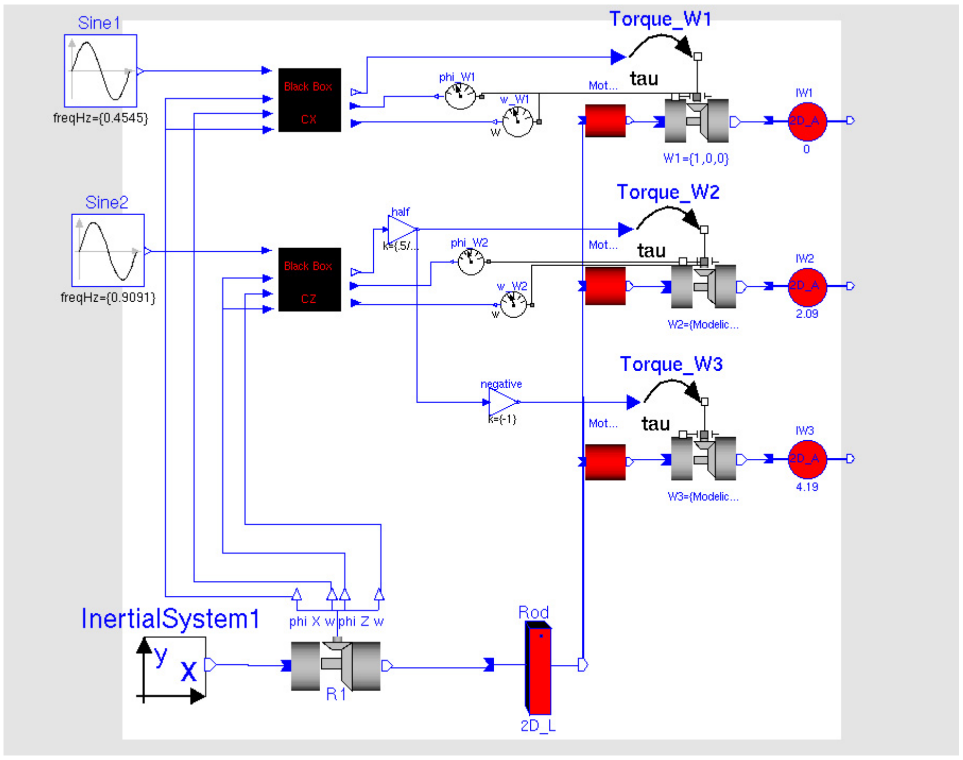


Illustration 1 Model of the three wheel pendulum with controller

Each variation of the spherical pendulum has its advantages and disadvantages. The two wheel arrangement is the easiest of all the arrangements with each inertia wheel being responsible for one of the oscillation directions. When designing the controller, there will be one for each wheel and therefore for each direction. The down side of this arrangement is that the pendulum will not be balanced in the center instead will be hanging at an angle in its natural position (see Ill. 2).

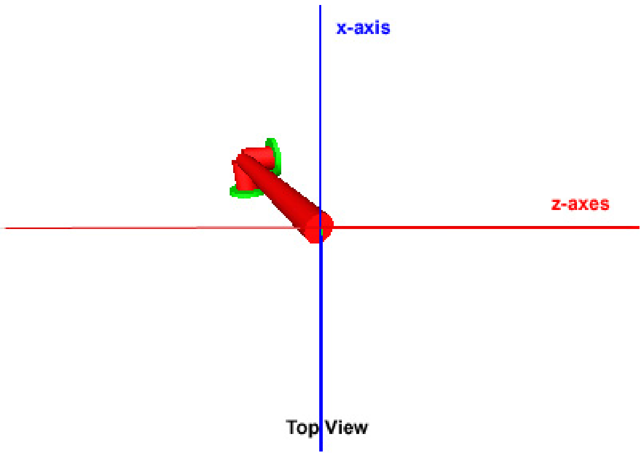


Illustration 2 Two wheel spherical pendulum hangs at an angle in stable equilibrium

The three wheel configuration eliminates this offset problem, however, now the wheels are not all oriented along the intended swing axis. The four wheel configuration takes care of the balance problem as well as that all the motors are arranged along one of the two swing axes. However, the use of four motors is not very efficient. In the end it was decided to continue with the three wheel arrangement. Two of the motors would just have to be controlled in such a way that their torque components in the direction of the third motor would cancel each other while adding up in the perpendicular direction. As a result, the two motors could be viewed as one single motor, simplifying the task of designing the controller (see Ill. 3).

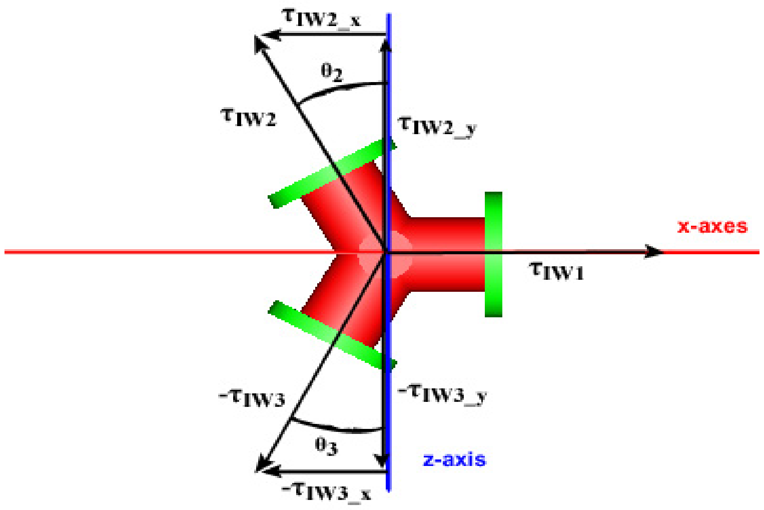


Illustration 3 Torque due to the inertia wheels

The mathematical justification is as follows. Assume that the controller gives the control signal u to motors 1 and 2 but -u to motor 3. Then the torques provided by inertia wheels will be

$$|\tau_{IW1}| = |\tau_{IW2}| = |\tau_{IW3}| = u \quad (1)$$

and

$$\tau_{IW2-x} = -\tau_{IW3-x} \quad \text{and} \quad \tau_{IW2-z} = \tau_{IW3-z} \tag{2}$$

if

$$\theta_2 = \theta_3 \tag{3}$$

Further, the torque in x and in z should be equal in magnitude

$$|\tau_x| = |\tau_z| \tag{4}$$

in order that the identical controller can be used for both directions. And since

$$\tau_x = \tau_{IW1} \tag{5}$$

then with equation 1

$$|\tau_x| = u \tag{6}$$

and

$$|\tau_z| = u \tag{7}$$

Totaling the torque in z gives

$$\tau_{IW2-z} + \tau_{IW3-z} = \tau_z \tag{8}$$

and since

$$\tau_{IW2-y} = \tau_{IW2} \cos\theta_2 \quad \text{and} \quad \tau_{IW3-y} = \tau_{IW3} \cos\theta_3 \tag{9}$$

and with equation 8

$$2 * \tau_{IW2} \cos\theta = \tau_z \tag{10}$$

Substituting with u

$$|2 * \tau_{IW2} \cos\theta| = u \tag{11}$$

Solving for τ_{IW2} gives

$$|\tau_{IW2}| = \frac{u}{2\cos\theta} \tag{12}$$

Therefore, dividing the control signal given by the controller by $2cos\,\theta$ before supplying it to motors two and three, will ensure that these two motors will behave as if there was one motor along the z-axis with the control signal u.

The motors were not modeled but instead a placebo that mimics their physical properties were used. The reason is that the model of a motor would include and inductor which adds an undesired fifth state to the whole model. The torque that would have been provided by the motor is now controlled directly.

Pendulum Library

Below is a description of the Pendulum Library components and their parameters. Some components are identical copies from the Modelica Standard Library and ModelicaAdditions Library. These parts have been included in the Pendulum Library for the comfort of having all essential parts in one place and will only be mentioned briefly. Some of them have been explained in more detail in the Modelica Appendix and the remaining are self-explanatory. There is also a comprehensive online explanation to each non-custom component.

All the custom-made components are constructed mostly from components already existing in the Modelica Standard Library or ModelicaAdditions Library. The only components that are an exception to this are the **ShapeBody2** and **Body2**. These two have minor amendments from their original design, **ShapeBody** and **Body** from the ModelicaAdditions MultiBody **Parts** sub-library. The changes pertain the six inertia tensors. These will be explained in further detail later.

Components of the Pendulum Library

- **Rod** (see Ill. 4): This is the main pendulum rod. It contains the parameters length, mass, and radius. Using these values, all the other parameter values of **ShapeBody2** are calculated. For the center of mass vector, **rCM**, it is assumed that the rod has a homogeneous weight distribution and that the center of mass is located at half the length along the central axis. The **width** and the **height** are the diameter of the rod and therefore, in both cases, two times the **radius** (***2*radius***). Finally, the **r** vector is along the vector **LengthDirection** with its magnitude equal to the parameter **length**

  $r=(0,-length,0)$

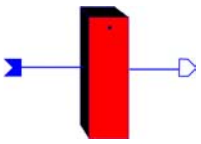


Illustration 4 Rod icon

- **Joint_2D** (see Ill. 5 and 6): This is joint is made from two revolute joints, one that is able to turn around the x and the other around the z axis. An angle and

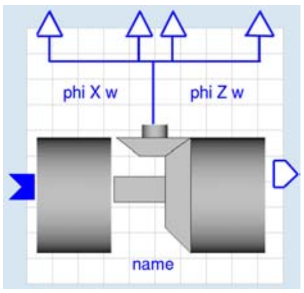


Illustration 6 Joint_2D icon

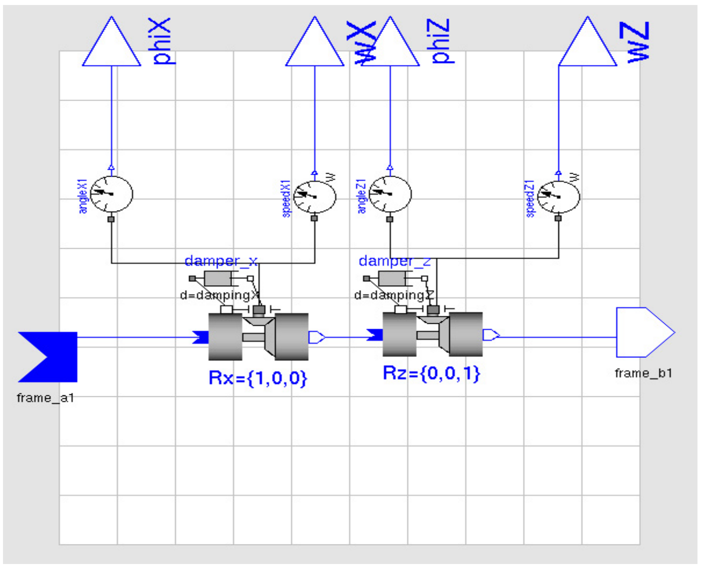


Illustration 5 Diagram of Joint_2D

  angle speed sensor are included for each joint and their reading are outputted at the top. Dampers are also included to make the component more realistic. Their values can be entered in the parameter window. This joint was built to be the fixed point joint of the pendulum.

- **Joint_3D**: This is the same joint as the **Joint_2D** only that it has been extended to include an revolute joint that turns about the y-axis. It can be implemented as a 2D joint by adding a lot of damping to the y-axis joint.

- **MotorPlacebo** (see Ill. 7): The actual motor with its dynamics is not modeled but its physical properties, such as weight and size are. This component makes use of the customized models **BodyShape2** and **Body2**. **Length**, **Mass**, **Radius** and **Theta** are the four parameters that have to be set. **Theta** is the angle that the motor makes with the positive x-axis. Its value may range anywhere between 0 and 6.28 ($2*\pi$). The equation section contains several if clauses to determine which quadrant **theta** is in and to calculate the proper values for the inertia tensors **I11**, **I22**, and **I33**. The **LengthDirection** vector and **r** vector are set in the direction of the angle **theta** with a magnitude equal to the parameter **length**. The center of mass is placed half way along these vectors. The **width** and **height** values are the diameter and therefore equal to twice the **radius**.

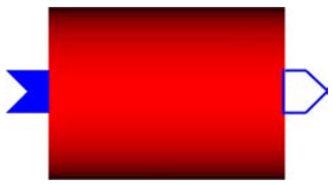


Illustration 7 MotorPlacebo icon

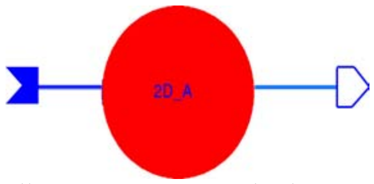


Illustration 8 InertiaWheel icon

- **InertiaWheel** (see Ill. 8): This component is identical to **MotorPlacebo** only that is is meant to represent the actual inertia wheel instead of the motor.

- **Controller** (see Ill. 9): The **controller** is a full state-feedback capable controller in which individual components such as the integrator or certain states can be turned off. Several items have been specifically programmed for the spherical pendulum model. For example, the expected state for the angle velocity is the derivative of the reference signal, which at the same time is the expected state of the actual angle. For models in which the stable equilibrium is not at θ_x=0, θ_z=0, the offset can be used to compensate for the deviation. Also, a limiter is placed before the output port to contain the control signal within its allowable bounds. Finally, the component **Add6** is a custom made adder that adds the six input signals together. For the controller implementation, the last three inputs to the adder are subtracted since they are feedback signals that have not been negated yet.

  To turn off individual components of the controller, the proper gain needs to be set to zero. The following is a list of the gains and what they influence:

  - **r_on_off**: the gain on the reference signal.
  - **sp_on_off1**: the gain on the set point of state one, the pendulum angle. In this case, state one is equal to the reference signal.
  - **L_phi**: the feedback gain on $x_{expected}(1) - x_{actual}(1)$ of the first state.
  - **sp_on_off2**: same as **sp_on_off1** only for state two, the angular velocity of the pendulum angle.
  - **L_w**: the feedback gain on $x_{expected}(2) - x_{actual}(2)$ of the second state.

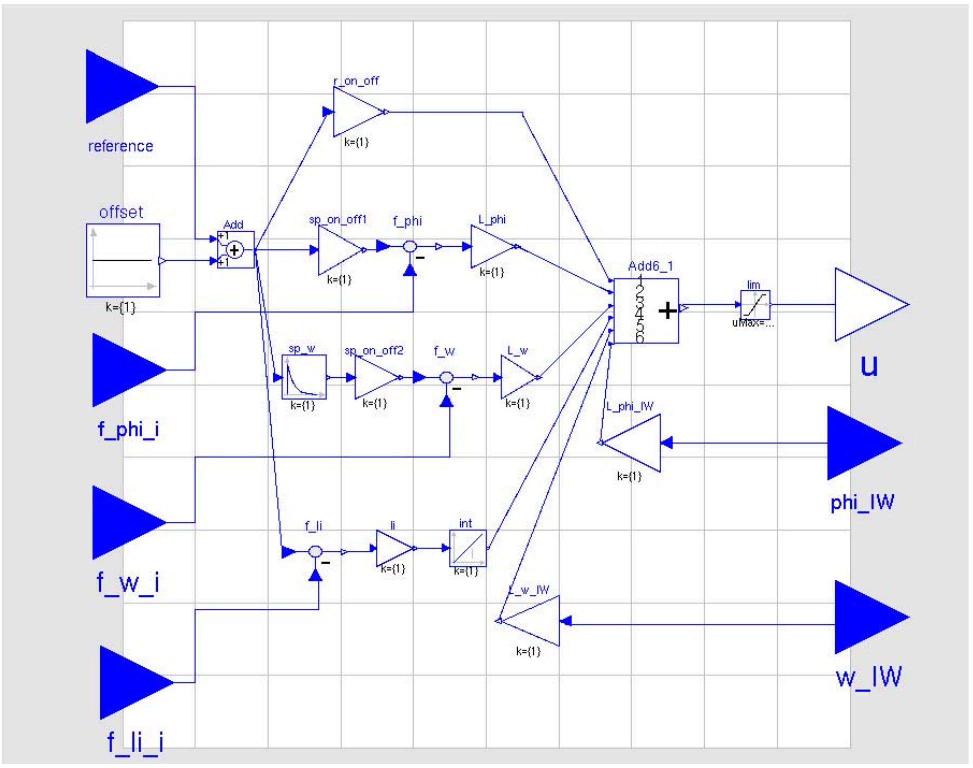


Illustration 9 The controller

- **li**: the gain for the integrator.
- **int.k[1]**: switch for integral action. 0=off, 1=on.
- **L_phi_IW**: feedback gain for inertia wheel angle.
- **L_w_IW**: feedback gain for inertia wheel velocity.

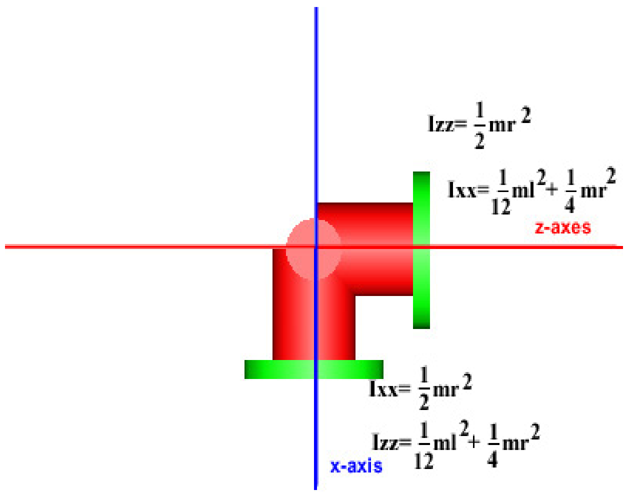
The values of these gains should not be set by adding modifiers in the parameter box but by running a script file after the model has been translated. More about the script file later.

- Other Components copied from the Modelica Standard Library and the ModelicaAdditions Library: **Gain**, **sin**, **AngleSensor**, **SpeedSensor** (angular), **InertialSystem**, and **Revolute** (joint)

The Inertia Tensors

In order for the model being build to have any physical and real world meaning, the inertia tensors values of the building block **ShapeBody** should approximately be correct. To remove this responsibility from the user and to increase the flexibility and ease-of-use of components in the Pendulum Library containing this building block, the six inertia tensors have been redeclared as regular variables. This amended version of the **ShapeBody** component was renamed to **ShapeBody2**. The same amendment was also applied to the building block **Body** and renamed to **Body2**. The tensors of the affected library components are now calculated from four other parameters given by the user. These are: **length**, **radius**, **mass** and **angle**, between the component's length direction and the positive x-axis.

When looking at the different tensors, several assumptions can be made about them. First, the Ixx (**I11**) and Izz (**I33**) tensors have their extreme values when the component lies parallel and perpendicular to these axes (see Ill. 10). In these orientations, the values of the tensors are easy to calculate.



$Izz=\frac{1}{2}mr^2$
$Ixx=\frac{1}{12}ml^2+\frac{1}{4}mr^2$
z-axes
$Ixx=\frac{1}{2}mr^2$
$Izz=\frac{1}{12}ml^2+\frac{1}{4}mr^2$
x-axis

Illustration 10 The inertia tensors

The inertia tensor Ixx, when the component lies along the x-axis, is

$$Ixx=\frac{1}{2}mr^2 \tag{13}$$

where m is the mass and r is the radius. Izz for the same component is

$$Izz=\frac{1}{12}mh^2+\frac{1}{4}mr^2 \tag{14}$$

where h is the height, in this case the length. In the case where the component lies along the z-axis, the two equations switch (see Ill. 10).

However, when the component lies at an angle theta (θ) in between these axes (see Ill. 11), calculating the exact values of these two tensors becomes more difficult. To ease the calculation, a linear approximation is made between the two extreme values. When moving the components along a circular path from the x-axis to the z-axis, the Ixx and Izz tensors switch values.

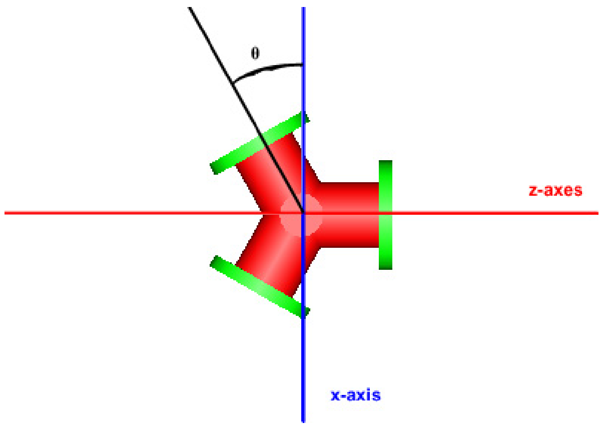


Illustration 11 Three wheel pendulum from top view

This means that in between the two axes, the values Ixx and Izz must lie between Ixx (eq. 13) and Izz (eq. 14). Assuming a linear change of the values makes it easy to calculate the values for Ixx and Izz. Even though, the values calculated this way are not exact, the controller should be robust enough to compensate for this model uncertainty.

Equations for Inertia Tensor Calculations

In order to use the linear approximation, the slope between the two extreme values needs to be calculated. This variable is called **increment** and is calculated as follows:

$$increment = \ldots$$
$$\ldots \frac{\left(\left(\frac{1}{12} * mass * length^2 + \frac{1}{4} * mass * radius^2\right) - \left(\frac{1}{2} * mass * radius^2\right)\right)}{ninety} \quad (15)$$

ninety is a constant with a value of 1.57, which is the radian equivalent of 90°. The top of the quotient calculates the difference between the two extreme values. Dividing by *ninety* results in the slope.

Now, the two tensors can be calculated.

$$Ixx = \frac{1}{2} * mass * radius^2 + increment * theta2 \quad (16)$$

$$Izz = \left(\frac{1}{12} * mass * length^2 + \frac{1}{4} * mass * radius^2\right) - (increment * theta2) \quad (17)$$

The user does not actually enter the relative angle *theta2* between the two axes, but rather the absolute angle **theta** measured from the positive x-axis. The angle is then automatically reduced to the relative angle *theta2* between the positive (or negative) x-axis and the component.

In regards to the Iyy (**I22**) tensor, it is always the same regardless how the component is placed, as long as it is along the horizontal plane. The equation is:

$$Iyy = \frac{1}{12} * mass * length^2 + \frac{1}{4} * mass * radius^2 \quad (18)$$

The remaining three elements Ixy (**I21**), Ixz (**I31**), and Iyz(**I32**) are assumed to be zero.

The Parameter Values

The following are the values used for the various parameters. They correspond to the values of the components used for the real physical experiment.

Inertia wheel: mass= 0.075kg, radius=0.025m, length=0.006m

Motor: mass=0.040kg, radius=0.015m, length= 0.035m

Rod: mass=0.1kg, radius=0.01m, length=0.5m

Linearizing

After having built and simulated the input model, it was linearized to then develop a controller for the model using Matlab. When linearizing the model, Modelica establishes the following states: position and rotational speed for each inertia wheel and pendulum rod angle and rotational speed around the fixed point for each oscillation direction. The order of these states in the state-space matrices of the linearized model dependents on the order of declaration of the respective parts in the text-based mode of the model. In order to properly use the custom m-files in Matlab, it was important that these declarations follow a special order. The

Revolute joint **R1** had to be declared before all of the **Revolute** joints of the inertia wheels and these had to be declared in the counterclock order, starting with the wheel in the direction of the positive x-axis.

Script file and Simulation with Controller

With the L vector from Matlab, the controller was tested in Modelica using the controller based version of the model. To ease the setting of the values of the various controller components, a script file was used. The following is a list of all the controller components that could be set using the script file:

- **Sine1.amplitude[1]**
- **Sine2.amplitude[1]**
- **Sine1.phase[1]**
- **Sine2.phase[1]**
- **Sine1.freqHz[1]**
- **Sine2.freqHz[1]**
- **CX.r_on_off.k[1]**
- **CZ.r_on_off.k[1]**
- **CX.sp_on_off1.k[1]**
- **CX.sp_on_off2.k[1]**
- **CZ.sp_on_off1.k[1]**
- **CZ.sp_on_off2.k[1]**
- **CX.li.k[1]**
- **CZ.li.k[1]**
- **CX.int.k[1]**
- **CZ.int.k[1]**
- **CX.L_phi.k[1]**
- **CX.L_w.k[1]**
- **CX.L_phi_IW.k[1]**
- **CX.L_w_IW.k[1]**
- **CZ.L_phi.k[1]**
- **CZ.L_w.k[1]**
- **CZ.L_phi_IW.k[1]**
- **CZ.L_w_IW.k[1]**

The **limiter** value cannot be set with a script file. It had to be set in the model itself before translating it.
Having loaded the script file, the simulation was ready to be initiated. Sometimes the simulator took a long time and the Linux shell window would display the message "...a large amount of work has been expected (about 500 steps) in the integrator...". The cause for this was that integrator value of the controller was too

big. The simulation had to be killed, the integrator value decreased and the simulation then re-started. When the simulation was completed successfully, the animation was run to visually observe the performance and effectiveness of the controller.

Chapter 2 - Model Verification

Since it is possible to model any process with Modelica, there needs to be some sort of model verification to ensure that the Modelica model does truly represent the desired process. The physical parameters especially such as the center of mass vector **rCM** and the inertia tensors need to be verified for their correctness since it is these parameters that determine the physical behavior of the model.

There are a variety of methods to choose among on how to verify the model. The obvious option would be to build the process and compare the reaction, to certain inputs, of the physical process to that of the Modelica model. Another possibility is to manually derive the mathematical model of the process, linearize it, and compare it to the linearized version of the Modelica model. For the project, this second option was selected for the verification.

The model of the Inverted Pendulum served as the starting point for the mathematical model. It consists of a rod oriented along the negative y-axis attached to a joint that is able to rotate around one axis, say x. At the bottom of the rod, an inertia wheel along with a motor to turn it is attached. The rotational axis of the inertia wheel is oriented parallel to the x-axis (see Ill.12).

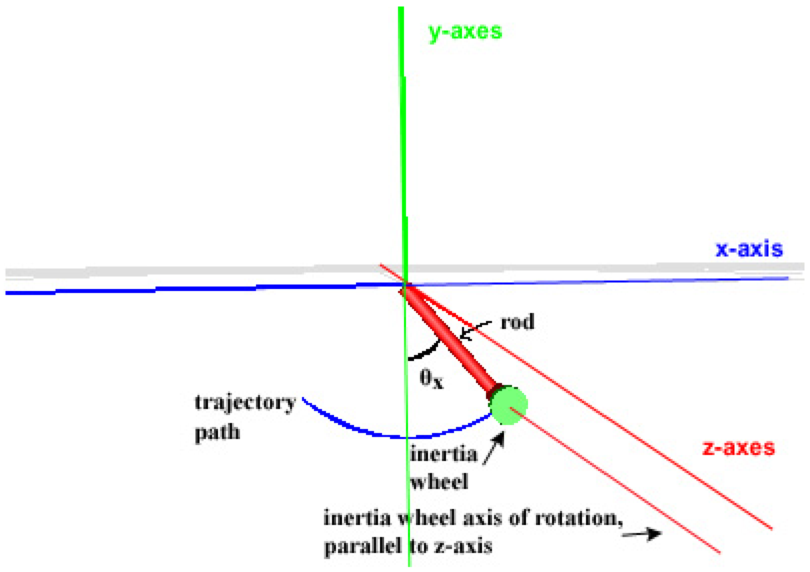


Illustration 12 Inverted Pendulum

Basically, this pendulum is quite similar to the spherical pendulum with the only difference being that the spherical pendulum has an additional degree of freedom for a total of two.

The mathematical model of the inverted pendulum is

$$J\ddot{\theta}_x + mgl\sin\theta_x = -\tau \tag{19}$$

$$\tau = J_{IW}\ddot{\theta}_{IW} \tag{20}$$

where the variable and parameters are defined as follows:

- θ_x is the angle the pendulum rod makes with negative y-axes.
- J is the moment of inertia of the pendulum about the pivot point.

- τ is the torque supplied by inertia wheel due to motor accelerating it.
- m is the mass of the pendulum.
- l is the distance from the pivot to the center of mass of the pendulum.
- θ_{IW} is the angle of rotation of the inertia wheel, relative to the fixed base, not relative to the pendulum rod. $\theta_{IW} = \theta_x + \phi$
- J_{IW} is the moment of inertia of the inertia wheel about its center of mass.

This results in the four states $\theta_x,\ \dot{\theta}_x,\ \theta_{IW}, \dot{\theta}_{IW}$.

However, since Modelica measures the angle θ_{IW} relative to the pendulum rod and not relative to the fixed base, it becomes difficult to verify the second equation and it will be omitted in the verification.

In the spherical pendulum, the rod is now also able to swing around the z-axes (see Ill. 13). As a result, the torque provided by an inertia wheels will not be oriented entirely along its intended direction when swinging about the axis perpendicular to it.

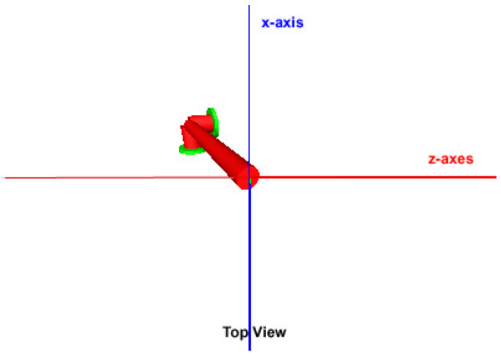


Illustration 13 Top view of spherical pendulum

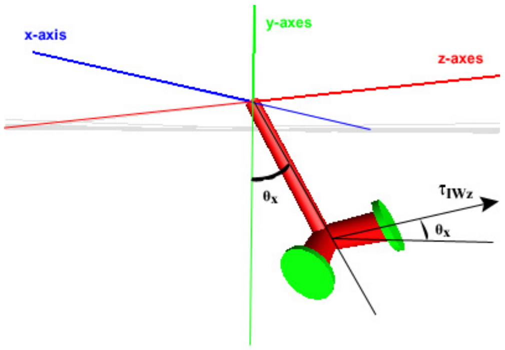


Illustration 14 Torque from inertia wheel z acting about z axis depends on angle theta x

The torque acting in the intended direction is reduced by the factor $\cos(\theta_p)$, where θ_p is the angle of the pendulum rod in the perpendicular oscillating plane. In case of the inertia wheel x it is the angle θ_z and in case of the inertia wheel z it is the angle θ_x (see Ill. 14).

$$x: \qquad J_x \ddot{\theta}_x + mgl \sin\theta_x = -\tau_{IWx} \cos\theta_z \tag{21}$$

$$z: \qquad J_z \ddot{\theta}_z + mgl \sin\theta_z = -\tau_{IWz} \cos\theta_x \tag{22}$$

Therefore, if the pendulum does not swing in the perpendicular direction, the angle θ_p will be zero, $\cos(\theta_p)$ will be one, and the torque will remain unchanged.

Since the two dimensions are identical to each other, further calculations will be carried out only in x.

Dividing equation 21 by J results in

$$\ddot{\theta}_x + \frac{mgl}{J_x} \sin\theta_x = -\frac{\tau_{IWx}}{J_x} \cos\theta_z \tag{23}$$

Further, solving for $\ddot{\theta}_x$ gives

$$\ddot{\theta}_x = -\frac{mgl}{J_x}\sin\theta_x - \frac{\tau_{IWx}}{J_x}\cos\theta_z \tag{24}$$

To be able to linearize the model, the stable equilibrium needs to be defined.

$$\ddot{\theta}_x = 0 \tag{25}$$

$$0 = -\frac{mgl}{J_x}\sin\theta_x - \frac{\tau_{IWx}}{J_x}\cos\theta_z \tag{26}$$

$$x^0: \quad \tau_{x0} = 0, \quad \theta_{x0} = 0, \quad \theta_{z0} = 0 \tag{27}$$

The next step is to linearize equation 24 about x^0.

$$\begin{aligned}\ddot{\theta}_x = 0 + (-\frac{mgl}{J_x}\cos\theta_x)\,|_{(x^0)}(\theta_x - \theta_{x0}) + (\frac{\tau_{IWx}}{J_x}\sin\theta_z)\,|_{(x^0)}(\theta_z - \theta_{z0}) + \ldots \\ \ldots + (-\frac{1}{J_x}\cos\theta_z)\,|_{(x^0)}(\tau_{IWx} - \tau_{IWx0})\end{aligned} \tag{28}$$

which gives the final equation

$$\ddot{\theta}_x = -\frac{mgl}{J_x}\theta_x - \frac{1}{J_x}\tau_{IWx} \tag{29}$$

For $\ddot{\theta}_z$ the equation then is

$$\ddot{\theta}_z = -\frac{mgl}{J_z}\theta_z - \frac{1}{J_z}\tau_{IWz} \tag{30}$$

To be able to compare this result to the linearized Modelica model, the parameters $-\frac{mgl}{J_x}$ and $-\frac{1}{J_x}$ need to be calculated.

For the following calculations, the values of the parameter are those presented in chapter 1, The Model.

For J_x

$$J_x = J_{Rod} + J^x_{Mx} + J^x_{Mz} + (m_{Mx} + m_{Mz})\,l_1^2 + J^x_{IWx} + J^x_{IWz} + (m_{IWx} + m_{IWz})\,l_2^2 \tag{31}$$

where

- J_{rod} is the moment of inertia of the rod about its end.

$$J_{Rod} = \frac{1}{3}ml^2 = 8.33 * 10^{-3}\,\text{kg m}^2 \tag{32}$$

- $J_{Mx}{}^x$ is the moment of inertia of motor x about the x-axis going through its center of mass.

$$J^x_{Mx} = \frac{1}{2}mr^2 = 4.5 * 10^{-6}\,\text{kg m}^2 \tag{33}$$

- $J_{Mz}{}^x$ is the moment of inertia of motor z about the x-axis going through its center of mass.

$$J^x_{Mz} = \frac{1}{4} mr^2 + \frac{1}{12} ml^2 = 6.33 * 10^{-6}\,\text{kg m}^2 \tag{34}$$

- $J_{IWx}{}^x$ is the moment of inertia of inertia wheel x about the x-axis going through its center of mass.

$$J^x_{IWx} = \frac{1}{2} mr^2 = 2.34 * 10^{-5}\,\text{kg m}^2 \tag{35}$$

- $J_{IWz}{}^x$ is the moment of inertia of inertia wheel z about the x-axis going through its center of mass.

$$J^x_{IWz} = \frac{1}{4} mr^2 + \frac{1}{12} ml^2 = 1.19 * 10^{-5}\,\text{kg m}^2 \tag{36}$$

- m_{Mx}, m_{Mz}, m_{IWx}, and m_{IWz} are the masses of the respective parts.
- l_1 is the distance from the center of mass of the motors to the pivot point. Approximately 0.5 m.
- l_2 is the distance from the center of mass of the inertia wheels to the pivot point. Approximately 0.5 m.

J_x and J_z are equal due to the symmetry of the pendulum.

For *ml*

$$ml = m_{Rod}\, l_{CMRod} + (m_{Mx} + m_{Mz} + m_{IWx} + m_{IWz})\, l_{CM} \tag{37}$$

where

- m_{Rod} is the mass of the pendulum rod
- l_{CMRod} is the distance from the pivot point to the center of mass of the pendulum rod. In this case 0.25 m.
- l_{CM} is the distance from the pivot point to the center of mass of the motors and inertia wheels. Due to their close proximity to each other and relative great distance to the pivot point, this distance is just assumed to be equal for all four parts and equal to the length of the pendulum rod, 0.5 m.

These equations result in

$$J = 6.58 * 10^{-2}\,\text{kg m}^2 \quad \text{and} \quad -\frac{1}{J} = 15.2\ \text{kg}^{-1}\,\text{m}^{-2}$$
$$-\frac{mgl}{J} = -20.9 \tag{38}$$

When linearizing the Modelica model, the following state-space matrices A and B are returned

$$A = \begin{Bmatrix} 0 & 1 & 0 & 0 \\ -18.2872 & -0.0128 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 18.2872 & 0.0128 & 0 & 0 \end{Bmatrix} \qquad B = \begin{Bmatrix} 0 \\ -0.0013 \\ 0 \\ 4.2679 \end{Bmatrix} * 10^4 \tag{39}$$

where $A_{2,1}$ correspondence to $-\frac{mgl}{J_x}$ and $B_{2,1}$ to $-\frac{1}{J_x}$.

Even though the values are not exactly the same, they are close enough to conclude that the Modelica model does adequately represent the real process.

Chapter 3 - The Controller

A variety of possible controller designs exist to be implemented in the spherical pendulum. There is the choice between linear and non-linear controller, and within each of these categories, there is another range of possibilities. Each one has its advantages and disadvantages, the key is to find the controller that will meet the job requirements as efficiently as possible.

Before deciding what type of controller to use some simplifications were made. First, the spherical pendulum would operate close to the stable equilibrium allowing the model to be linearized around this point (see Ill. 15). This decision assured the use of a linear controller. Further, the pendulum would be unable to rotate around

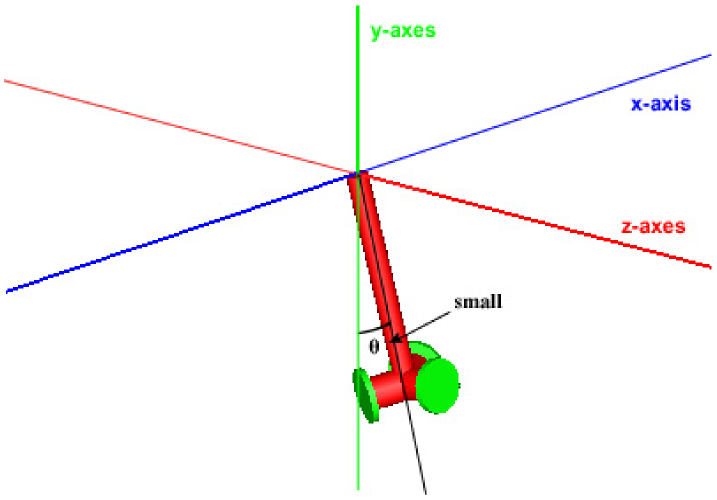


Illustration 15 Spherical Pendulum near stable equilibrium

its own axis. This permits the two direction of oscillation to be viewed independent of each other and allows the use of two identical decoupled controllers, one for each direction. With these simplifications, the state-feedback controller seemed to be the best choice.

The strength of the state-feedback controller is that it can move the poles of the closed-loop system. If the open-loop system has unstable right-hand-plane poles, a state-feedback controller is able to move these poles into the left-hand-plane. The only requirement for the controller to achieve this is that it needs measurements or estimations of all the values of the states of the process. This is no problem when working with Modelica since any state can easily be measured by just connecting the proper sensor to the appropriate joint. Therefore, caution is required when working with Modelica since it is possible to measure variables that in the actual process are not accessible to be tracked with a sensor. Of course, it is possible to build an observer for the application that will estimate the desired states from other measured states. For example, if in the spherical pendulum the motors do not have any optical sensors to relay how much and how fast each inertia wheel has turned, this information could be deduced knowing the applied voltage and motor characteristics.

Combined with the Matlab Control Systems Toolbox, the task of calculating the state-feedback controller parameters for a Modelica model was eased. When linearizing the model about the operating point in Modelica, the state-space matrices of the linearized model were saved in a mat file. This mat-file was then loaded into Matlab. Using a series of m-files provided by Dymola, the state-space matrices A, B, C and D were extracted from the mat file. With these matrices, using some self-written m-files and Matlab functions, the optimal L vector was calculated for the state-feedback controller. This procedure is elaborated below.

Calculating the Controller Parameters

The results of the linearization in Modelica are saved in the file called **`dslin.mat`**. There are two ways to load this file into Matlab. One option is to just use the function **`load('dslin.mat')`** which will load one huge matrix that includes the state-space matrices A, B, C, and D. The other command **`tloadlin('dslin.mat')`** returns the four state-space matrices separately. In order to use the commands that follow, it was important to use the latter.

Since the two oscillation directions are viewed independent of each other, it became necessary to decouple the two directions in the state-space matrices. In other words, the proper elements of the state-space matrices, loaded with one of the above commands, had to be extracted and saved as their own matrices. The loaded matrices either had the eight or twelve states depending on the Modelica model. The extracted matrices, however, should only have the following four states: angle in swing direction x, angular speed in swing direction x, angle of inertia wheel x, and angular speed of inertia wheel x. Due to model symmetry, the state-space matrices for z would be the same and did not have to be extracted separately. Also, there would be no model with ten states as might be expected from the three wheel model. This is since two of the wheels will be controlled such that they act as one; see chapter 1 The Models for an explanation. Therefore, when designing a controller for the three wheel model, the linearization of the two wheel model was used.

To ease the task of extracting the proper elements of the state-space matrices, the m-files **`extract_2D.m`** was written. The inputs to this file are the four state space matrices loaded earlier, and the outputs are the reduced decoupled state-space matrices. As an example, the command is applied as follows:

```
[A_dc,B_dc,C_dc,D_dc]=extract_2D(A,B,C,D)
```

The next step was to calculate the desired poles of the closed-loop system. This was achieved using the m-file **`getPoles`**. The input arguments to the function are: number of poles (states), frequency, and damping. The frequency is given as rad/s, and the damping is given as the angle between the positive y-axis and the pole. The poles are returned as a vector.

```
P=getPoles[states, frequency, damping];
```

How to choose the proper frequency and damping is explained below.

The final step was to compute the state-feedback gain vector L to get the desired closed-loop poles. This was accomplished with the Matlab command **`place`**. The input arguments are the decoupled matrices A and B and the pole vector P.

```
L=place(A_dc, B_dc, P);
```

The components of the L vector were then entered into the Modelica script file and tested.

Choosing the Parameters

The effectiveness of the controller is dependent on various factors. These include frequency and damping of the chosen poles, frequency and amplitude of the reference signal and actuator strength. The focus of the following sections will be on how to choose the right poles and what reference signals are feasible to track.

Poles

The elements of the feedback vector L are dependent on the chosen poles. The placement of these poles determine the frequency and damping characteristics of the controller. Traditionally, increased damping reduces the overshoot and increased frequency decreases the rise time of the step response. However, for the spherical pendulum, the step response has little meaning. Instead, it is tracking the reference trajectory with minimal error that is of interest. Therefore, the objective is to find the poles that will reduce this error the most.

Further, tests have shown that the pole placement has a significant influence when working with integral action, an essential factor when trying to minimize the error. When just using a regular state-feedback controller without integral action, solely changing the frequency and damping has little influence on the error (see Table 1).

Frequency	Damping	Direction	Integral	Error
4.27	45 (see Ill.2)	x	0	0.003
		z		0.104
	90	x		0.003
		z		0.097
4.27x2=8.54	45	x	0	0.004
		z		0.091
	90	x		0.004
		z		0.093

Table 1 Damping only with default reference signal specified in Table 2

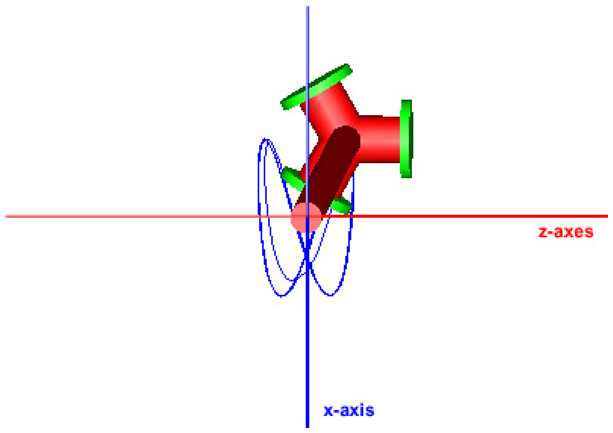


Illustration 16 Pole frequency: 4.27 Damping: 45 Integral: 0

The effects of the frequency and damping are clearly seen when using an integrator to decrease the tracking error. For a given controller, a certain maximum amount of integral action can be added before the error cannot be reduce any further and the system becomes unstable. Increasing the frequency and or damping of the controller allows more integral action to be added.

To start with, the poles where chosen so that the frequency was equal the natural frequency of the pendulum (4.27 rad/s) and the damping was 45°.

The reference signals for the tests in this section were the following (see Table 2)

Controller	Signal form	Amplitude	Frequency	Phase
x	sine	0.1 rad	4.33 rad/s	0
z	sine	0.1 rad	2.17 rad/s	0

Table 2 Default reference signal

Proper tracking of these reference signals results in a figure eight when the motion is projected onto the the x-z plane.

Focusing on the frequency, tests have shown that while increasing the frequency allows for more integral action, it also increases the error. Even increasing the damping and integral action to their maximum at the increased frequency is not enough to reduce the error to what it was before. Table 3 below summarizes the results when doubling the frequency from 4.27 to 8.57 rad/s and then increasing the damping and integral action.

Frequency	Damping	Direction	Integral	Error
4.27	45	x	25	0.001
		z	40	0.028
4.27x2=8.57	45	x	25	0.003
		z	40	0.080
4.27x2=8.57	90	x	100	0.002
		z	130	0.061

Table 3 Effects of doubling pole frequency

Even with the increased damping and integral action, the error at the higher frequency was more than twice that at the lower frequency (see Ill. 16 and 17).

Lowering the frequency below the natural frequency results in only a slight increase in error but makes the system much more sensible to integral action forcing integral action to be drastically lowered in order for the system to stay stable. For example, when lowering the frequency to half the natural frequency, the integral action in z had to be lowered from 40 to 9. The table 4 shows this.

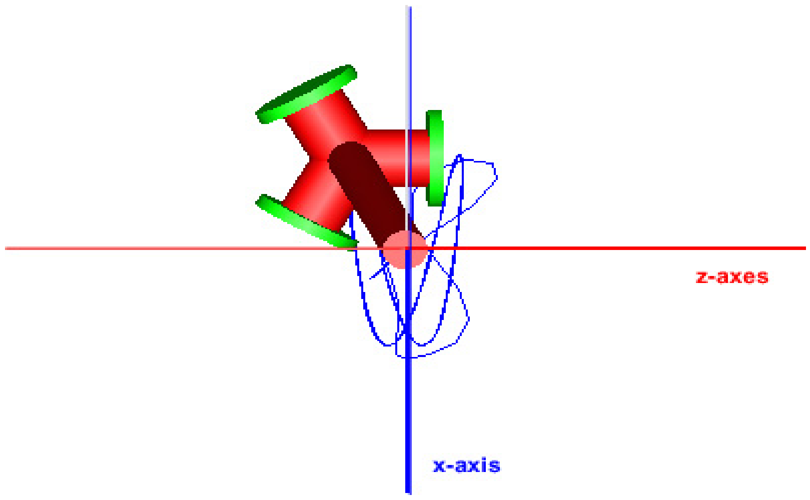


*Illustration 17 Pole frequency: 4.27*2 Damping: 90 Integral: x=100, z=130*

Frequency	Damping	Direction	Integral	Error
4.27	45	x	25	0.001
		z	40	0.028
4.27x0.5=2.135	45	x	0	0.002
		z	9	0.037

Table 4 Effects of lowering pole frequency

In respect to the settling time, there seemed to be minimal influence from the frequency. Irregardless of the controller frequency, the settling time swayed between 1.5 to 2 seconds.

In conclusion, the ideal frequency to place the poles at is the natural frequency of the pendulum.

The natural frequency can be obtained in two ways. One method is to time the period of pendulum by just letting it swing free in space. The other method is to use the Matlab command `eig(A_dc)` to get the eigenvalues of the decoupled A matrix and use the eigenvalues to calculate the period of the pendulum. As seen in table 5, both methods give very similar results.

Method	Frequency
Timing the period	4.33
Eigenvalue	4.27

Table 5 Timed period vs. eigenvalue

In regard to the controller however, the difference in periods is negligible and the same results are achieved irregardless of which value is used.

Alternating the damping has shown to be much more effective in decreasing the error. Setting the damping to 90° allows for the maximum possible integral action. This is especially useful when trying to decrease a larger error. For example, when the frequency is equal to the natural frequency and the damping is 45°, the error in z without integral action is 0.104. With a maximum integral action of 40 at 45°, the error is reduced to 0.028, a reduction of 76%. Increasing the damping to 90°, permits increasing the integral action to 75 and reduces the error to 0.020, 8% more for a total of 84% error reduction. The error in x for the same controllers changed from 0.003 to only 0.001, a change of only 2%. The results also show that, even though increasing damping from 45° to 90° allows for much greater integral action, the end effect on minimizing tracking error is minimal. Table 6 summarizes the results. Illustration 18 shows the best possible tracking that was achieved of the figure eight reference trajectory.

Frequency	Damping	Direction	Integral	Error	% Error
4.27	45	x	0	0.003	3
		z	0	0.104	104
4.27	45	x	25	0.001	1
		z	40	0.028	28
4.27	90	x	75	0.001	1
		z	75	0.020	20

Table 6 Effects of increasing damping and integral action

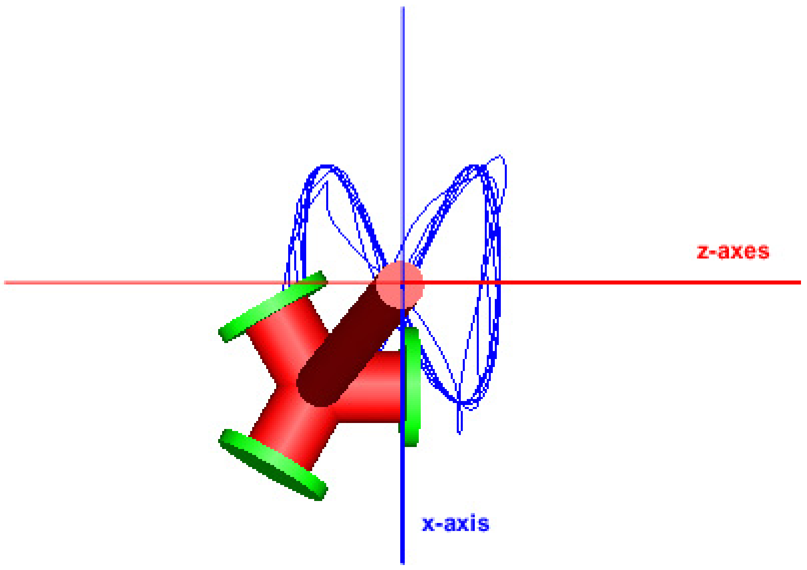


Illustration 18 Pole frequency: 4.27 Damping: 90 Integral: 75

Additionally, unlike what the case was when increasing the frequency, solely increasing damping does not result in a greater error.

Reference Signal

Comparable to the poles, the reference signal also has a substantial influence on how effective the controller is. The easiest trajectory to follow is one with a frequency equal to the natural frequency of the pendulum. Trying to follow a trajectory that has a frequency other than the natural frequency leads to tracking errors that cannot be reduced to zero or even close to zero. This is clearly demonstrated in the above results, where the error in z at its best is still 20%. Trying to track a reference signal where the frequencies were 5.65 and 2.83 rad/s produced comparable minimal errors (see Table 7 and Ill. 20 and 19). Even though, it does not appear as if there is an error in illustration 19, the amplitude in x is too big and too small in z.

Controller Frequency=4.27 and Damping=90					
Ref. Amp.	Ref. Freq.	Direction	Integral	Error	% Error
0.1	4.33 rad/s	x	100	0.001	1
	2.17 rad/s	z	75	0.020	20
0.1	5.65 rad/s	x	100	0.027	27
	2.83 rad/s	z	75	0.016	16

Table 7 Effects of reference signal frequency on tracking error

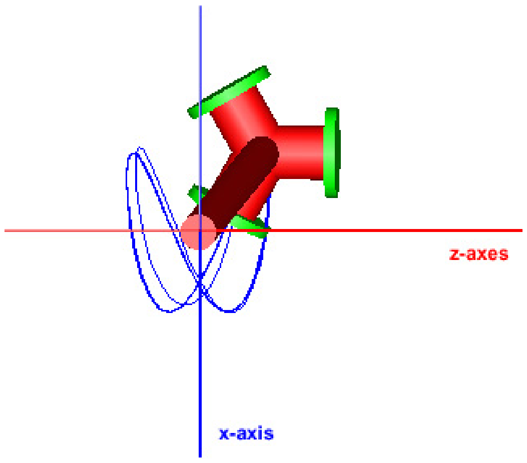


Illustration 20 Pole frequency: 4.27 Damping: 45 Integral: 0 Ref. Freq.: 5.65/2.83 rad/s

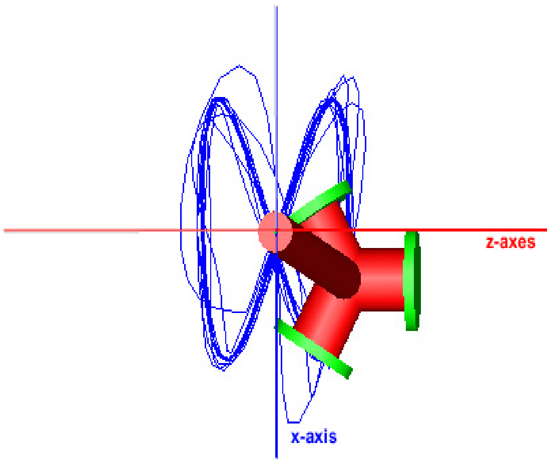


Illustration 19 Pole frequency: 4.27 Damping: 90 Integral:x=100, z=75 Ref. Freq.: 5.65/2.83 rad/s

Aside from not only leading to an increased error, the control signal to achieve these "minimal yet high" errors is also larger when compared to tracking a reference signal with its frequency equal to the natural frequency of the pendulum (see Table8). This means that stronger actuators and more energy is necessary to achieve these results.

Controller Frequency=4.27 and Damping=90					
Ref. Amp.	Ref. Freq.	Direction	Integral	Error	Torque (Nm)
0.1	4.33 rad/s	x	100	0.001	0.006-0.01
	2.17 rad/s	z	75	0.020	0.07
0.1	5.65 rad/s	x	100	0.027	0.17
	2.83 rad/s	z	75	0.016	0.07

Table 8 Effect on control torque due to non-ideal reference signal

The amplitude of the reference signal, in addition to the frequency, also plays a role in the effectiveness of the controller. Since the controller was designed based on a model linearized around the stable equilibrium of the pendulum, large amplitudes will lead to large deviations between the model and the actual process. When the controller was tested with a amplitude of 0.2 radians (11.5°), a slight increase in percent error was already observed (see Table 9).

Controller Frequency=4.27 and Damping=90					
Ref. Amp.	Ref. Freq.	Direction	Integral	Error	% Error

Controller Frequency=4.27 and Damping=90					
0.1	4.33 rad/s	x	100	0.001	1
	2.17 rad/s	z	75	0.020	20
0.2	4.33 rad/s	x	100	0.005	2.5
	2.17 rad/s	z	75	0.045	22.5

Table 9 Effects of reference signal amplitude on tracking error

Due to non-linearities, greater amplitudes will induce unproportionally greater errors.

The ideal reference signal to track is therefore one with its frequency equal to the natural frequency of the pendulum and its amplitude held small.

Conclusion

Modelica

Modelica is a comfortable modeling language to work with, especially for persons with some programming knowledge. There is an extensive library that virtually allows anything to be modeled by simply selecting the proper elements and connecting them in the workspace. Alternatively, it is possible to also program in the underlying text layer. The challenging part is to properly implement the various parts when working with the MultiBody Library. Setting the proper values for numerous vectors and inertia tensors is essential for a legitimate real world simulation.

When working with complex models, it is wise to keep them uncluttered from numerous components in order to maintain a clear overview. One way to achieve this is to create subcomponents that contain components that are related to each other, either functionally or spatially. Additionally, creating a library of custom components greatly facilitates the task of creating various versions of ones model.

Matlab

Designing the state-feedback controller with the aid of Matlab is mostly an untroublesome process. The only complication was the inability for the **`place`** function to place the poles for the steady-space matrices that were extended for integral action. The reason for this is still unknown. It was therefore rather cumbersome trying to find the optimal integral action by just trial and error simulations. A personal recommendation is to always check the Matlab results to see if it needs to be multiplied with a factor. For example, failing to notice that the resulting matrix was scaled by 10,000 may lead to unnecessary hours of error searching.

Results

In terms of the controller, there are several things to consider. First of all, the poles of the system should be chosen such that their frequency is equal to the natural frequency of the spherical pendulum. Second, the frequency of the reference signal should also be equal to the natural frequency of the pendulum. This will keep control error and actuator effort at a minimum. If the reference frequency needs to be something other than the natural frequency, then it becomes essential to use integral action to minimize the error. Depending on the deviation between the two frequencies, the error may or may not be reduced to its optimal minimum. Further, damping beyond 45° has little influence anymore on the error. Finally, keeping the amplitude of the reference trajectory small, will also keep the error smaller.

Future

The next step would be to expand the model to a spherical pendulum that is able to turn about its own axis and develop a controller for this version. The main problem here is to compensate for the spin of the pendulum about its own axis. Otherwise the two main swing directions cannot be viewed as decoupled from each other anymore since each inertia wheels will then influence both swing directions. Not only that, but to be then able to properly measure the orientation of the inertia wheel relative to the origin and to split up the torque into its correct x, y, and z components is greatly complicated.

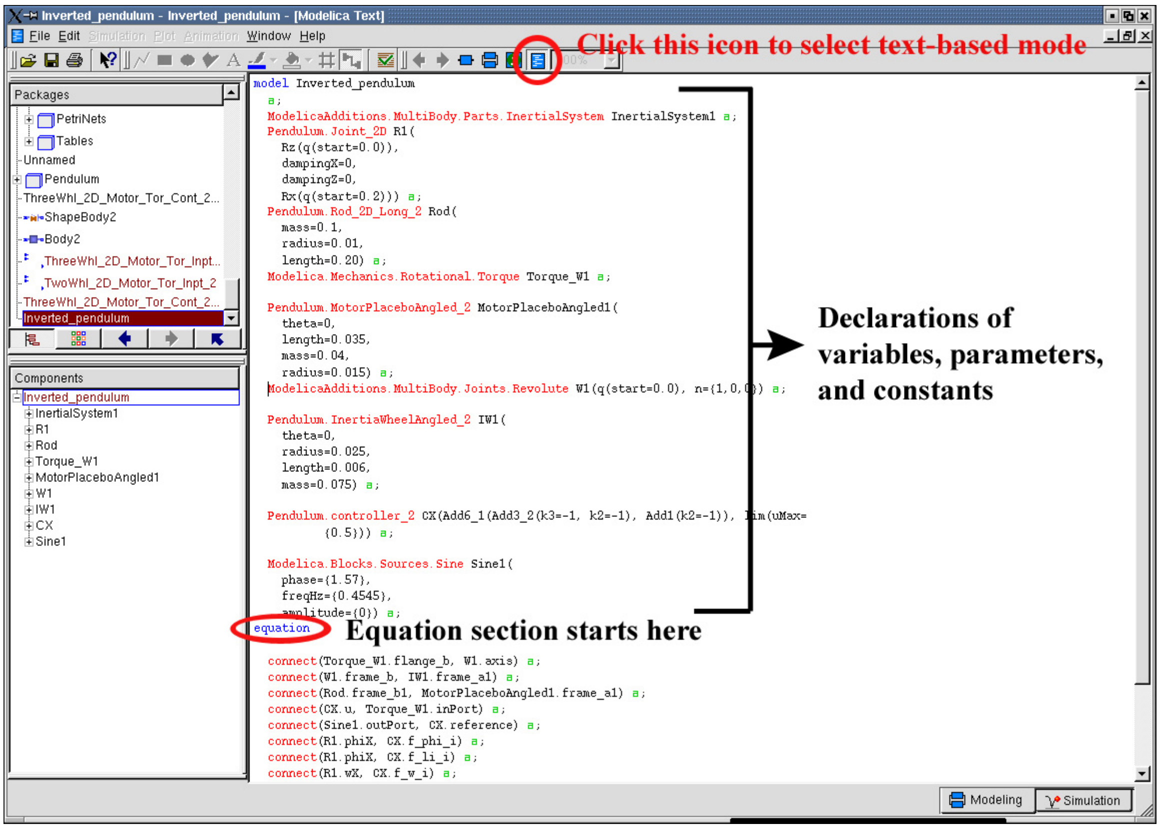
Appendix A – Modelica, a Quick Tutorial

To model the spherical pendulum on the computer, it was decided to use Modelica due to its many advantages. First of all, Modelica is a domain neutral modeling language. This means that it is not restricted to only one modeling domain such as electrical or mechanical as many other modeling languages. Secondly, Modelica has the advantage of allowing the user to solve problems that are expressed in terms of differential-algebraic equations (DAEs) rather than ordinary differential equations (ODEs). A further advantage is that Modelica allows for the simultaneous simulation of both continuous and discrete behavior in a model. Additionally, Modelica provides several extensive libraries one of which is the Modelica Standard Library (MLS). This library offers all the fundamental mechanical, electrical, thermal, and mathematical elements to model almost any desired process. There is even the possibility to establish ones own custom library of self-built components. Finally, the processes modeled with the MultiBody library can be animated in 3-D. Modelica is only the programing language and there are several vendors that offer easy-to-use software packages. For this project the Dymola version from Dynasim was chosen. The following is a quick tutorial on how to use Modelica in relevance to this project

Text- or Diagram-Based

There are three ways to build a model in Modelica, either text-, diagram-based, or a combination of the two. The text-based programming is similar to programming in a regular language such as Java or C++. Variables, constants, and parameters have to be declared and defined in the first half of the model and then be used below in the equation or algorithm section (see Ill. 21). In the diagram-based modeling, components from the various libraries can just be selected, be dragged onto the workspace, and be connected to create the desired model (see Ill. 22). The available libraries are quite extensive and allow practically anything to be modeled.

The library components are actually models of their own with input and/or output connectors. When connecting components, it is important to be aware that different types of interfaces exist and that an interface of one type cannot be connected to that of another. For example, it would make little sense to connect a translational speed sensor to a rotational joint. While programming in the diagram-based mode, Dymola generates the proper lines of code in the text layer. Later on, if it is necessary to exchange one component with another, it is easy to do so in the text layer. Especially, if the two parts have identical inputs and outputs, since then all the connection lines in the diagram-based mode will not have to be manually re-drawn.


Inverted_pendulum - Inverted_pendulum - [Modelica Text]
File Edit Simulation Plot Animation Window Help
Click this icon to select text-based mode
Packages
PetriNets
Tables
Unnamed
Pendulum
ThreeWhl_2D_Motor_Tor_Cont_2...
ShapeBody2
Body2
ThreeWhl_2D_Motor_Tor_Inpt...
TwoWhl_2D_Motor_Tor_Inpt_2
ThreeWhl_2D_Motor_Tor_Cont_2...
Inverted_pendulum
Components
Inverted_pendulum
InertialSystem1
R1
Rod
Torque_W1
MotorPlaceboAngled1
W1
IW1
CX
Sine1
Declarations of variables, parameters, and constants
Equation section starts here
Modeling
Simulation

```
model Inverted_pendulum
  a;
  ModelicaAdditions.MultiBody.Parts.InertialSystem InertialSystem1 a;
  Pendulum.Joint_2D R1(
    Rz(q(start=0.0)),
    dampingX=0,
    dampingZ=0,
    Rx(q(start=0.2))) a;
  Pendulum.Rod_2D_Long_2 Rod(
    mass=0.1,
    radius=0.01,
    length=0.20) a;
  Modelica.Mechanics.Rotational.Torque Torque_W1 a;

  Pendulum.MotorPlaceboAngled_2 MotorPlaceboAngled1(
    theta=0,
    length=0.035,
    mass=0.04,
    radius=0.015) a;
  ModelicaAdditions.MultiBody.Joints.Revolute W1(q(start=0.0), n={1,0,0}) a;

  Pendulum.InertiaWheelAngled_2 IW1(
    theta=0,
    radius=0.025,
    length=0.006,
    mass=0.075) a;

  Pendulum.controller_2 CX(Add6_1(Add3_2(k3=-1, k2=-1), Add1(k2=-1)), lim(uMax=
          {0.5})) a;

  Modelica.Blocks.Sources.Sine Sine1(
    phase={1.57},
    freqHz={0.4545},
    amplitude={0}) a;
equation
  connect(Torque_W1.flange_b, W1.axis) a;
  connect(W1.frame_b, IW1.frame_a1) a;
  connect(Rod.frame_b1, MotorPlaceboAngled1.frame_a1) a;
  connect(CX.u, Torque_W1.inPort) a;
  connect(Sine1.outPort, CX.reference) a;
  connect(R1.phiX, CX.f_phi_i) a;
  connect(R1.phiX, CX.f_li_i) a;
  connect(R1.wX, CX.f_w_i) a;
```

Illustration 21 Screen shot of text-based mode

Bottom-up or Top-down

Modelica provides for bottom-up and top-down programming. For bottom-up, models can be saved and incorporated as components of another larger model. Models can even be saved as components of a custom library. Whenever a library component is modified, all models that incorporate this part are updated automatically. It is also possible to change the parameters of a single instance of a library components by adding the proper modifier in the model where it is used. The bottom-up approach is very helpful when working with complex models in order to keep a good overview. For top-down programming Modelica offers such constructors as **partial block, extends, replaceable** and **redeclare.**

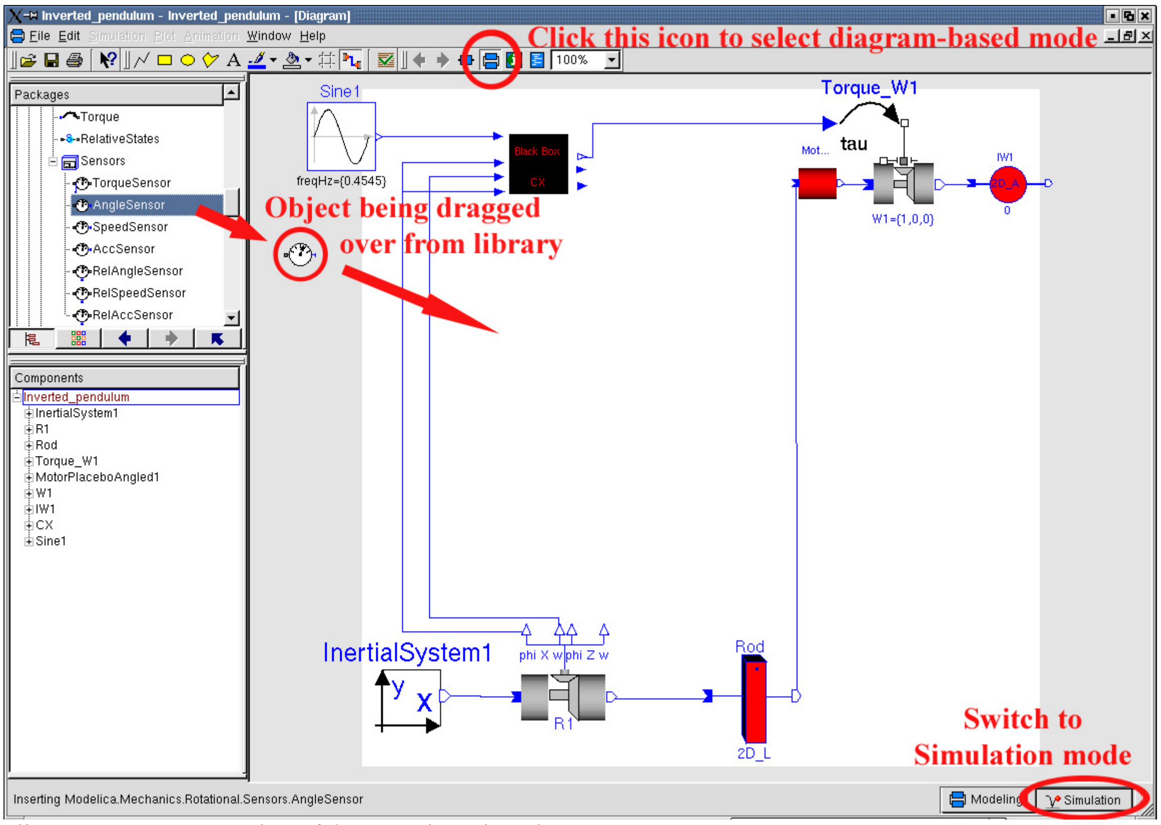


Illustration 22 Screen shot of diagram-based mode

Modifiers

Modifiers to a component can be added in both the diagram- and text-based view. In the diagram-based view, right click on the component and select **Parameters...** or just double click on the icon (see Ill. 23). Then, in the new window that appears, click on the **Add Modifiers** tab (see Ill. 24). There will be an input text field to enter the parameter name with its new desired value. If the desired parameter is part of a subcomponent, then the name of the subcomponent has to be entered first,

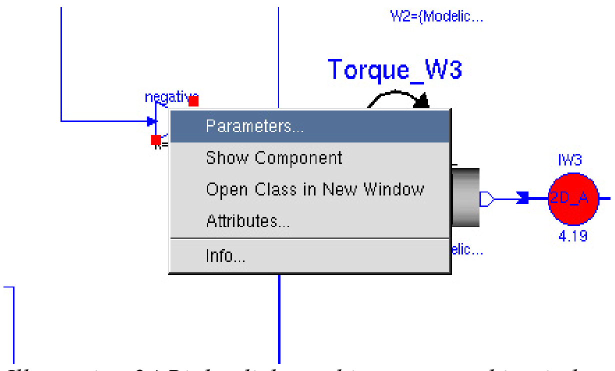
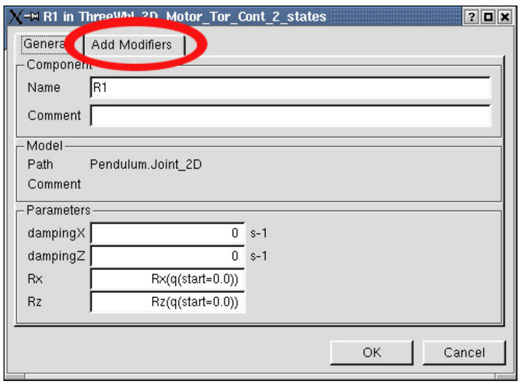


Illustration 24 Right click on object to open this window



Component
Name R1
Comment

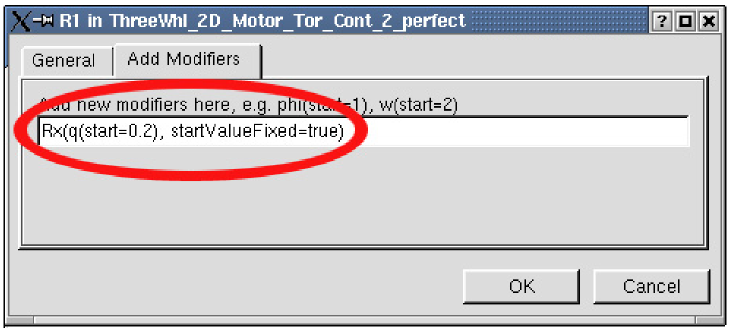
Model
Path Pendulum.Joint_2D
Comment

Parameters
dampingX	0	s-1
dampingZ	0	s-1
Rx	Rx(q(start=0.0))	
Rz	Rz(q(start=0.0))	

Illustration 23 Click on Add Modifiers

followed by a parenthesis, "(", and then the parameter name and its new value. For example, the component **Joint_2D** from the Pendulum Library contains two revolute joints (**Rx** and **Rz**) from the MultiBody Library. Each of these contains a parameter **q** that determines the initial starting angle of the joint for the simulation. By default, this value is zero. To change it to some non-zero value (e.g. 0.2 radians) for one of the joints (e.g. **Rx**) the modifier ***Rx(q(start=0.2))*** has to be entered. If more than one parameter of a component has to be changed, then all parameters have to be added to the same modifier but separated by commas, for example:

Rx(q(start=0.2), startValueFixed=true) (see Ill.25).



Add new modifiers here, e.g. phi(start=1), w(start=2)

```
Rx(q(start=0.2), startValueFixed=true)
```

Illustration 25 Screen shot of adding modifier

In the text-based mode, the modifiers can be entered directly after the main component declaration. Taking the previous example, it would look as follows in the text-based mode:

***Pendulum.Joint_2D R1(Rz(q(start=0.2)), Rx(q(start=0.2)))*.**

Multi-Body Library

The MultiBody Library, part of the ModelicaAdditions Library, contains the main building block that were used to construct the spherical pendulum. As already mentioned, processes modeled with this library may be animated in 3-D. Below is a description of some of the important and essential parts of this library, specifically in respect to the spherical pendulum. Even though, a quite comprehensive HTML-based documentation with a description of each Modelica library part and its parameters is provided with Dymola.

- **InertialSystem** - location: **Parts** sub-library- This component defines the absolute origin of the modeling room (see Ill. 26).
- **Joints** sub-library: This sub-library contains all possible joints to connect any two pieces from the MultiBody Library and define their movement relative to each other. Joints may also be connected together to increase their degree of freedom. The names of the individual joints are self-explanatory that an individual description of each one is superfluous (see Ill.27).

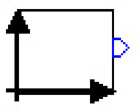


Illustration 27 InertialSystem Icon

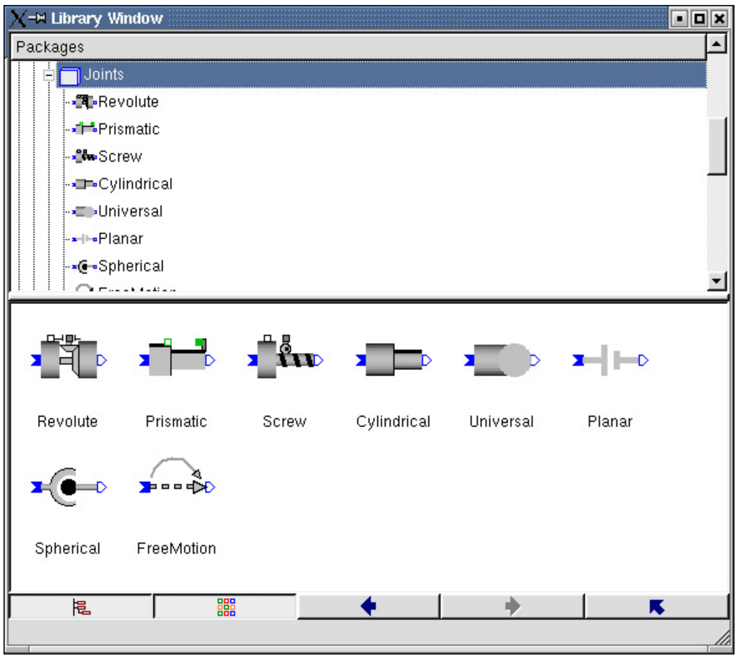


Illustration 26 Screen shot of Joints Library

- **ShapeBody** - location: **Parts** sub-library - This block is used to represent the physical shape and characteristics of the modeled process. Many standard shapes are already defined and further custom shapes can be imported as dxf files. It consists of the two parts **Shape** and **Body** that are also found in the **Parts** sub-library. **Shape** contains all the visual parameters while **Body** defines all the physical characteristics of the **ShapeBody** (see Ill. 28 and 29).

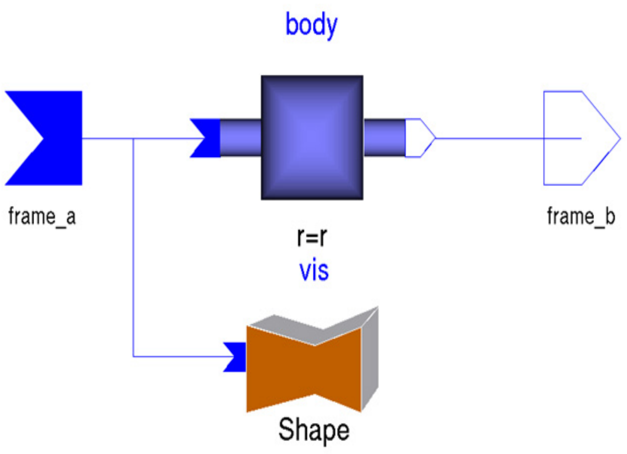


Illustration 28 Diagram of ShapeBody

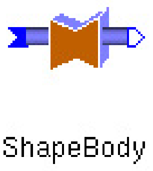
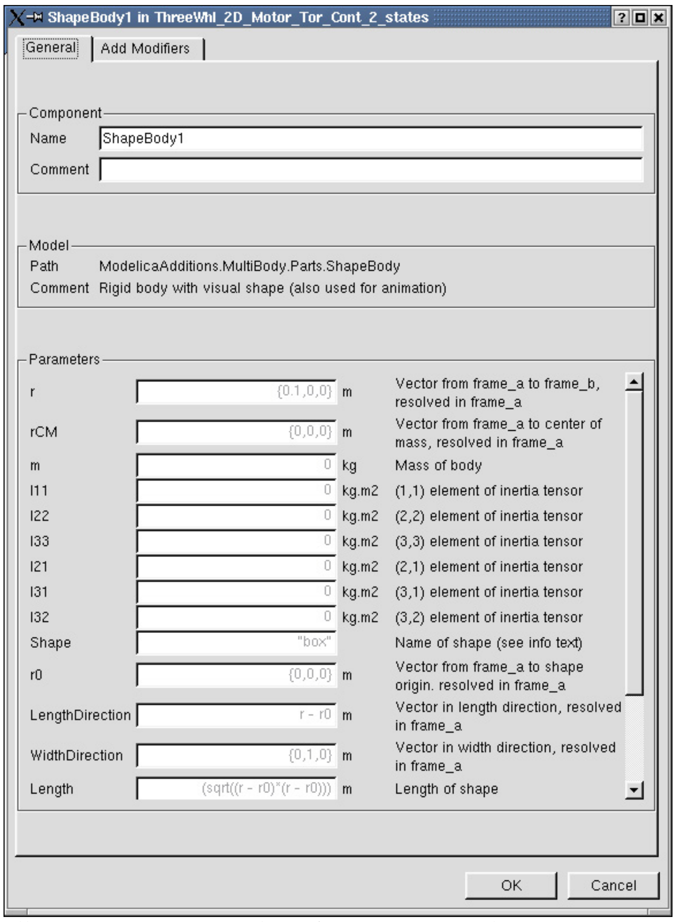


Illustration 29 ShapeBody Icon

The visual parameters are those that determine how the 3D animation will appear. In order for the model to have any practical meaning, the physical parameters have to be set as properly as possible. If the meaning of these parameters are correctly understood and calculated, building a complicated model becomes quite easy since Modelica takes care of all the complex calculations. The various parameters are explained in more detail below (see Ill. 29). All parameters have SI units.

- **Frame A**. The origin of this frame is one of the two places where the **ShapeBody** connects to other parts. It can be viewed as the "head" of the body.
- **Frame B**. The origin of this frame is the other place where the **ShapeBody** connects to other parts. It can be viewed as the "tail" of the body.
- **r**: This is the vector from the origin of **Frame A** to the origin of **Frame B**. It therefore determines where other parts attached to the "tail" of the **ShapeBody** would connect relative to **Frame A**. The magnitude as well as the direction of this vector are relevant.
- **rCM**: This vector determines the position of the center of mass of the **ShapeBody** relative to the origin of **Frame A**. The magnitude as well as the direction of this vector are relevant.
- **m**: This is the mass of the **ShapeBody**.
- **I11**: This is the element (1,1) of the inertia tensor. Assume that the inertial system underwent a translation and is now located at the center of mass of the **ShapeBody**. This component is then the moment of inertia of the **ShapeBody** around the x-axis going through the center of mass.
- **I22**: This is the element (2,2) of the inertia tensor. The same assumption as above is valid only that now it is the moment of inertia around the y-axis.
- **I33**: This is the element (3,3) of the inertia tensor. The same assumption as above is valid only that now it is the moment of inertia around the z-axis.
- **I21**, **I31**, **I32**: These are the non-diagonal elements of the inertia tensor and are called products of inertia. For parts in which the center of mass has a symmetrical position within the body, these components of the inertia tensor are zero.
- **Shape**: The following shapes can be assigned to a **ShapeBody**: box, sphere, cylinder, cone, pipe, beam, gearwheel and wirebox. It is also possible to assign externally created shapes.
- **r0**: This is the vector from the origin of **Frame A** to the origin of the visualization. So it is basically possible to connect two parts that visually do not appear to be connected at all (see Ill.31).
- **LengthDirection**: This vector specifies in which direction the length of the object is oriented relative to **Frame A**. Here the magnitude of the vector has no relevance but the direction does.
- **WidthDirection**: This vector specifies in which direction the width of the object is oriented relative to **Frame A**. Here the magnitude of the vector has no relevance but the direction does.


ShapeBody1 in ThreeWhl_2D_Motor_Tor_Cont_2_states

General | Add Modifiers

Component
Name: ShapeBody1
Comment:

Model
Path: ModelicaAdditions.MultiBody.Parts.ShapeBody
Comment: Rigid body with visual shape (also used for animation)

Parameters

r	{0.1,0,0}	m	Vector from frame_a to frame_b, resolved in frame_a
rCM	{0,0,0}	m	Vector from frame_a to center of mass, resolved in frame_a
m	0	kg	Mass of body
I11	0	kg.m2	(1,1) element of inertia tensor
I22	0	kg.m2	(2,2) element of inertia tensor
I33	0	kg.m2	(3,3) element of inertia tensor
I21	0	kg.m2	(2,1) element of inertia tensor
I31	0	kg.m2	(3,1) element of inertia tensor
I32	0	kg.m2	(3,2) element of inertia tensor
Shape	"box"		Name of shape (see info text)
r0	{0,0,0}	m	Vector from frame_a to shape origin. resolved in frame_a
LengthDirection	r - r0	m	Vector in length direction, resolved in frame_a
WidthDirection	{0,1,0}	m	Vector in width direction, resolved in frame_a
Length	(sqrt((r - r0)*(r - r0)))	m	Length of shape

OK | Cancel

Illustration 30 Screen shot of parameter window

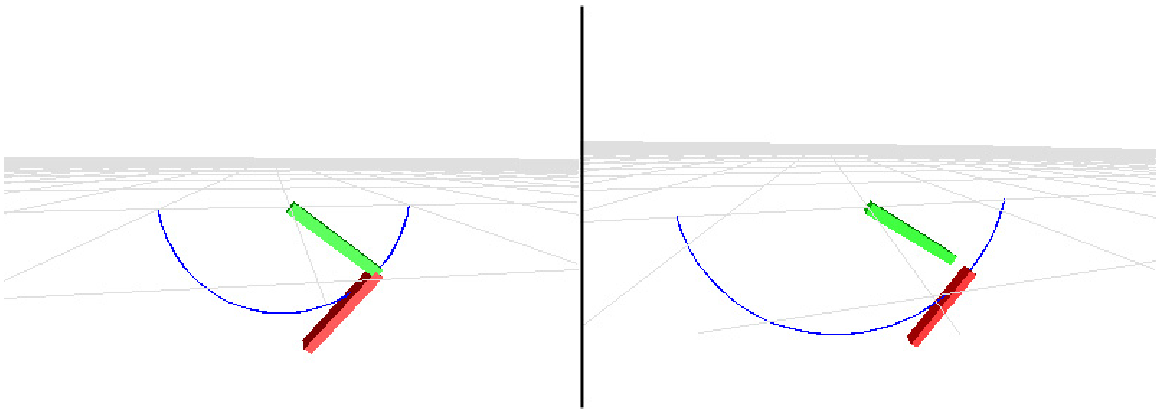


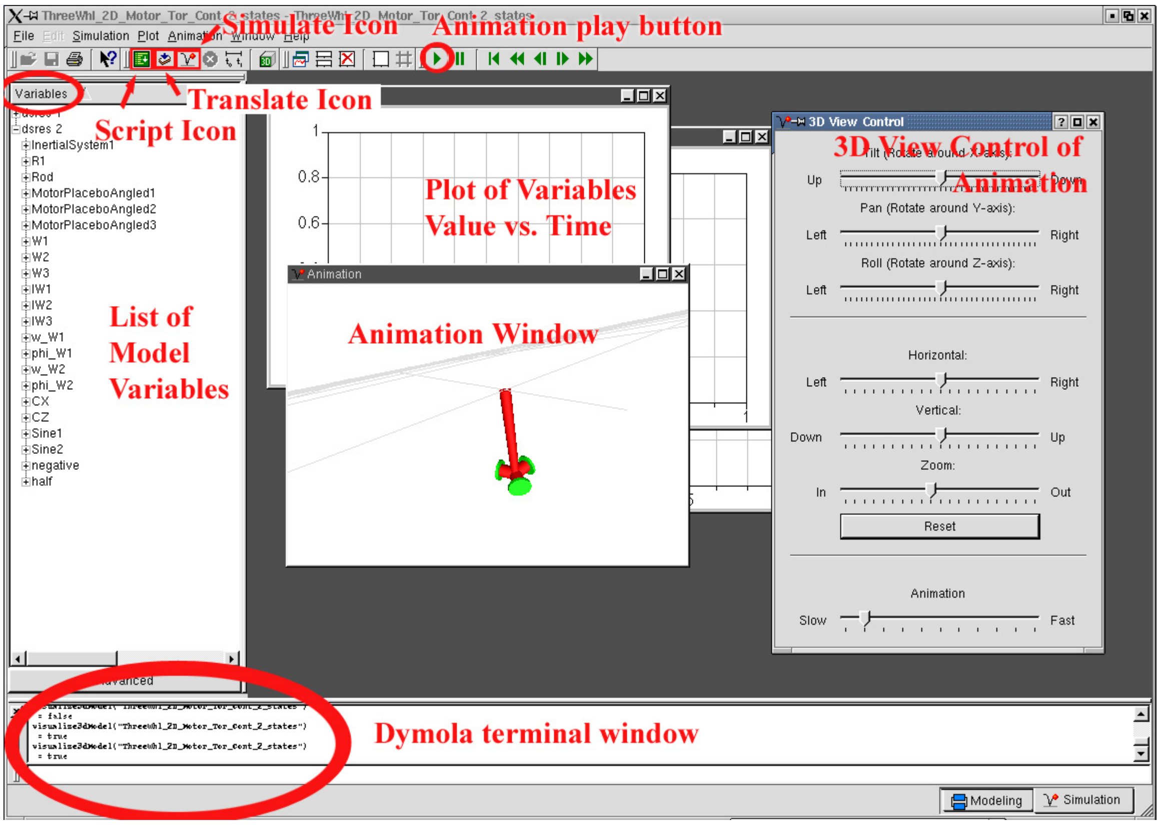
Illustration 31 Even though the two bars in the figure to the right do not appear connected, they are and behave physically the same as the construction in the figure to the left.

- **Length**: This number specifies how long the visual length of the object is in direction of the **LengthDirection** starting at the origin of the visualization as determined by **r0**.
- **Width**: This number specifies the width of the visual representation. The width is distributed equally in both width directions starting from the origin of the visualization. In case of a cylinder or sphere, it may be viewed as the diameter.
- **Height**: This specifies the height of the visual representation. The direction of the height is perpendicular to both the length and the width directions.

- **Material**: With this four element vector, the color and specular of the **ShapeBody** can be determined. The first three values are the RGB values and the fourth is the specular value. Specular =1 gives a metallic appearance.
- **Additional**: There are additional parameters for the pipe and cone shapes.

Building a Model

The easiest way to build the model of the desired process is to connect the necessary library components in the diagram-based view. This procedure could almost be compared to building a Lego model. Of course, there might be instances when it is necessary to change to the text-based view, for example when a loop structure is required. Also, when adding library components that have parameters



```
 = false
visualise3dModel("ThreeWhl_2D_Motor_Tor_Cont_2_states")
 = true
visualise3dModel("ThreeWhl_2D_Motor_Tor_Cont_2_states")
 = true
```

Illustration 32 Screen Shot of Simulation mode

that need to be set, such as the **k** value of a gain, it is wiser to do so later with the use of a script file (see following section).

Once the model has been completed, it is ready to be translated. This is done by changing to the simulation mode of Dymola by clicking on the simulation tab on the lower right of the screen (see Ill. 22). To translate the model, press the translate icon (see Ill. 32). If the translation was successful, **true** will appear in the terminal window (see Ill. 32) on the bottom and all the components will be listed on the left hand side of the Dymola window (see Ill. 32).

The next step is to create and run the script file mentioned earlier. This is basically a text file that lists all the parameters of the different components with their desired values. For example, if there is a gain component **Negate** that should be set to negative one, the entry in the script file would be as follows:

Negate.k[1]=-1;

It is important to save this text file with an **.mos** extension so that Dymola recognizes it. To execute the script file, click on the script icon (see Ill. 32) and select the script file in the dialog box. If there were no problems with executing the script file, **true** will again appear in the terminal window on the bottom. Otherwise, **false** will be displayed which means that one of the declared parameters in the script file was either misspelled or does not exist and Dymola could not find it. Subcomponents are separated by a period from their supercomponent.

The advantage of a script file is that it eases the changing of parameter values between simulations. Instead of always going back to the model-mode of Dymola, changing the parameter value at its source, returning to the simulation-mode and re-translating the entire model, one can just change the parameter value in the script file and re-load it.

Finally, when the translations was successful and the script file was loaded without any errors, the model can be simulated by clicking on the simulate icon (see Ill. 32). Depending on the complexity of the model, the simulation time will vary. Sometimes the simulator will say "...a large amount of work has been expected (about 500 steps) in the integrator...". This is most likely the result of non-ideal values for some of the parameters. For example, when simulating the spherical pendulum with the controller, this message would occur if some of the feedback gains were too large. In this case the simulator does not stop calculating but just slows down drastically. The wise thing is to kill the dymosin simulator in the Linux shell window, re-tune the parameter values in the script file, and retry simulating the model in hopes that it will work fine this time.

If the model was built using the MultiBody Library, then the 3D animation model will appear in the animation window once the simulation is complete. By pressing on the green play button (see Ill. 32), the animation of the process will start. Clicking on a body part, the path it takes in space is drawn. Additionally, the view position of the animation can be rotated and translated in any desired way making it possible to view animation from any position and angle. At the same time, by selecting the variables that appear on the left hand side of the window, their progression during the simulation can visually be observed on a graph. For example, when tuning a controller, one could check the performance of the controller, when trying different values for the controller parameters, by plotting the error between the reference and actual signal.

After having simulated the model, it is also possible to linearize the model around its initial starting position. This is especially useful when working with a non-linear model, such as the spherical pendulum, and wanting to create a linear controller for a specific operating point. Modelica calculates the state-space matrices of the linearized model and saves the results in the file **dslin.mat**. This gives the added advantage that the linearized model may then be loaded into Matlab for further calculations, as will be shown later in chapter 3, The Controller. To linearize the model, go to menu heading Simulation and select Linearize.

Appendix B – Matlab m-files

extract_2D.m

```
%extracts the proper matrices

function [A2,B2,C2,D2]=extract_2D(A,B,C,D)

A2=zeros(4);
A2(1,1)=A(1,1);
A2(1,2)=A(1,2);
A2(1,3)=A(1,5);
A2(1,4)=A(1,6);

A2(2,1)=A(2,1);
A2(2,2)=A(2,2);
A2(2,3)=A(2,5);
A2(2,4)=A(2,6);

A2(3,1)=A(5,1);
A2(3,2)=A(5,2);
A2(3,3)=A(5,5);
A2(3,4)=A(5,6);

A2(4,1)=A(6,1);
A2(4,2)=A(6,2);
A2(4,3)=A(6,5);
A2(4,4)=A(6,6);

B2=[B(1,1);B(2,1);B(5,1);B(6,1)];

C2=[C(1,1),C(1,2),C(1,5),C(1,6)];

D2=D(1,1);
```

getPoles.m

```
% inputs: radius is actually the desired frequency
%         damping is the desired damping (between 0 and 1)
%         states is the number of states (poles) of the system
% output: P is the vector with the desired poles.  They
%         all have the same frequency and are spaced equiangle

function [P]=getPoles(radius, damping, states);

%angle in first quadrant
theta=asin(damping);

%angle in second quadrant
```

```
theta=theta+(pi/2);

%first pole
p1x=radius*cos(theta);
p1y=radius*sin(theta);
p1=complex(p1x,p1y);

p2=conj(p1);

%angle between outer most poles
difference=2*pi+angle(p2)-angle(p1);

theta2=theta;

P=[p1,p2];

if states>2

   %divide space between outer most poles intoequiangle regions
   delta=difference/(states-2+1);

   %even number of poles desired
   if mod(states,2)==0
      disp('even number of states');

      %since poles come in pairs, only need to calculate one of them
      stop=(3+((states-4)/2));
      for i=3:stop,

         theta2=theta2+delta;

         tempX=radius*cos(theta2);
         tempY=radius*sin(theta2);
         temp=complex(tempX,tempY);
         temp2=conj(temp);

         P=[P,temp, temp2];
      end;
   end;

   %odd number of poles desired
   if mod(states,2)~=0
      disp('odd number of states');

      %since poles come in pairs, only need to calculate one of them
      stop=(3+(floor((states-2)/2)));
      for i=3:stop,

         theta2=theta2+delta;
```

```
            tempX=radius*cos(theta2);
            tempY=radius*sin(theta2);
            temp=complex(tempX,tempY);

            if i<stop
               temp2=conj(temp);
               P=[P,temp, temp2];

            else
               %no conjugate because on real axis

               P=[P,tempX];

            end;
         end;
      end;
   end;
```