PyMbs Documentation

Release 2

Carsten Knoll, Christian Schubert, Jens Frenkel, Sebastian Voigt

November 09, 2012

CONTENTS

Contents:

CHAPTER

ONE

ABOUT

PyMbs is being developed at Dresden University of Technology at the Institute of Processing Machines and Mobile Machinery by a questing quartet:

Figure 1.1: Carsten Knoll & Christian Schubert & Jens Frenkel & Sebastian Voigt

We startetd working on PyMbs in July 2009, as we faced the need of a convenient way to obtain the equations of motion of complex mechanical systems to feed a real time simulatiuon environment based on the Modelica modeling language. By the release of PyMbs 0.1 these aim could be achieved. Nevertheless PyMbs is subject to ongoing development to enhance its features and increase numeric performance. It is published under the LGPL license.

If you face any problems using PyMbs, feel free to contact us. As Christian's got a heart for PyMbs users, he will be your primary contact person. Of course you may consult any other developer as well.

Dresden University of Technology

Institute of Processing Machines and Mobile Machinery

Tel.: +49 (351) 463-39278

Fax: +49 (351) 463-37731

D-01062 Dresden, Germany

http://tu-dresden.de/bft

- christian.schubert@tu-dresden.de
- jens.frenkel@tu-dresden.de
- carstenknoll@gmx.de
- sebastian.voigt@mailbox.tu-dresden.de

CHAPTER

TWO

GETTING STARTED

This section is ment to be a quick start guide and shall provide an overview of the basic features and capabilites of the PyMbs module. For a more detailed description please consult the reference section of the PyMbs user guide.

2.1 What is PyMbs?

PyMbs is a tool to generate the equations of motion of holonomic multibody systems. The equations can be exported to several target languages like Matlab, Modelica, Python or C++. Thus the output of PyMbs is a mathematical model of the mechanical system which can be simulated by timestep integration [1].

The description of the mechanical system is text-based but held as easy and convenient as possible. The user assembles the system from a range of predefined objects like bodies, joints, loops, sensors etc. Anyway this method may provoke (typing) errors, especially when modelling more complex systems. Therefore it is possible to create a visualisation for the mechanical system from CAD-files or geometric primitives to check the consistency of the model. It is also possible to manipulate the degrees of freedom interactively.

2.2 Installation

PyMbs is completely written in Python and therefore it is platform-independent. It was tested on Python 2.6.5 and is based on the sympy module featuring symbolic calculation for Python. In this section the required modules are specified and afterwards a step-by-step instruction is given on how to install PyMbs on Windows machines, which is intended to help users who never got in touch with Python before.

2.2.1 Dependencies

- sympy – (at least 0.6.7)
- numpy
- scipy
- lxml – not necessary for general usage
- pymat – optional, allows PyMbs to use MATLAB's symbolic simplification capabilites, if present
- VTK – tested with 5.4.0

[1] Timestep integration is not (yet) a feature of PyMbs.

2.2.2 Windows

This guide assumes that no Python distribution is installed on your system. If Python is already installed, you may go directly to section *A First Example*. We strongly recommend the installation of a prebundled Python distribution to avoid troubles regarding dependencies.

- Download Python(x,y) from http://www.pythonxy.com. At the time, this guide was written, version 2.6.6.1 was the latest.
- Install Python(x,y). Make sure you selected *'Full'* as *'type of install'* (see figure *Python(x,y) installer*). This will install Python 2.6.5 to *'C:\Python26'* and Python(x,y) to *'C:\Program Files\pythonxy'* per default.

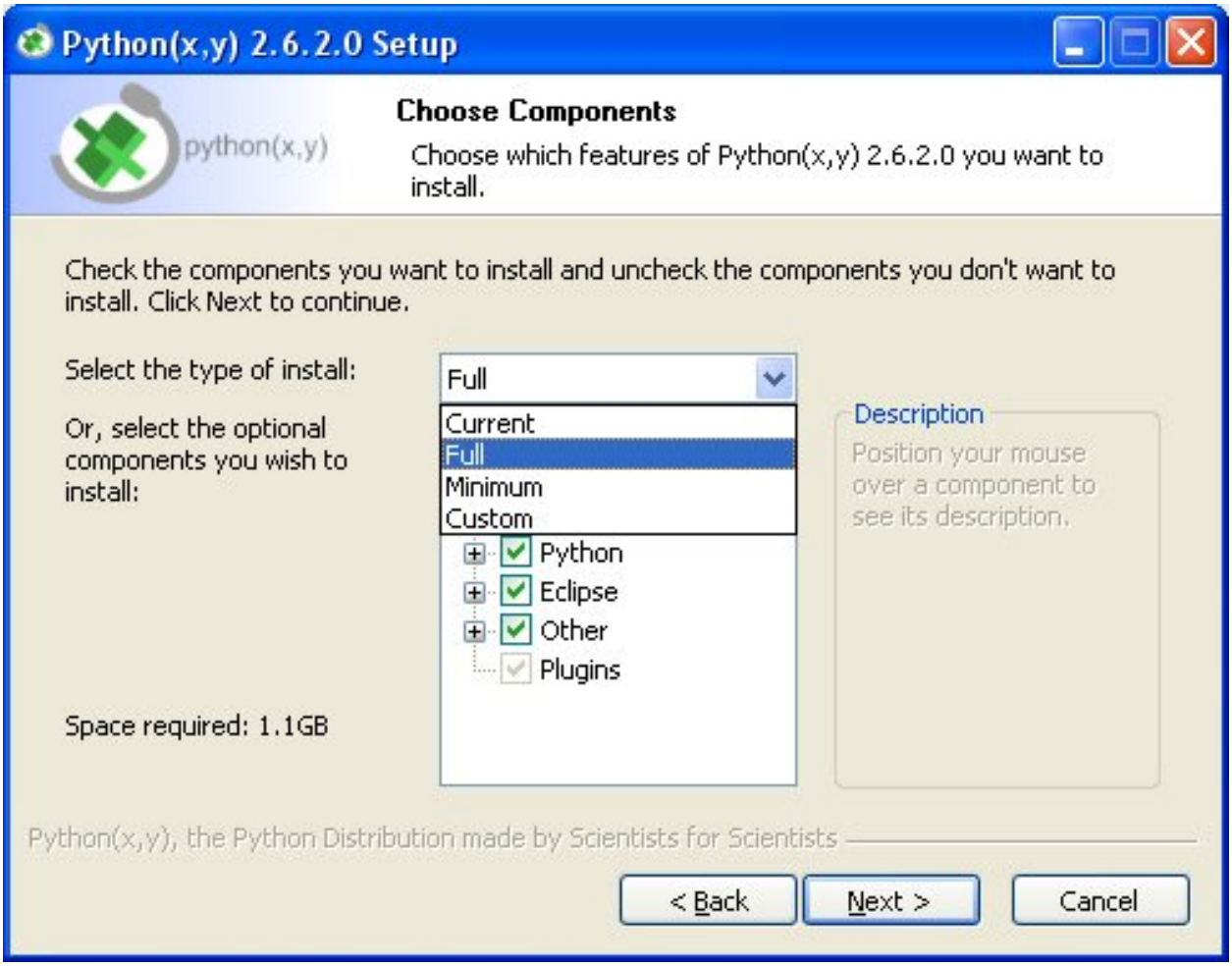

Figure 2.1: Python(x,y) installer

- Install PyMbs using the installer PyMbs-0.1.5.win32.exe. Note that this programme must be run with administrator privileges. Please make sure that PyMbs uses the correct Python Version which is in *'C:\Python26'* per default (see figure *PyMbs installer*). This is especially important, if there are more than one Python distributions installed.
- You may use any text editor to write your PyMbs models. However, we recommend the use of PyScripter as it supports syntax highlighting and code completion. It is freely available at http://code.google.com/p/pyscripter/. At the time this document was written version 2.4.1 was the latest. Please install it.
- You can check your installation by starting Python(x,y) from your desktop and pressing the button next to

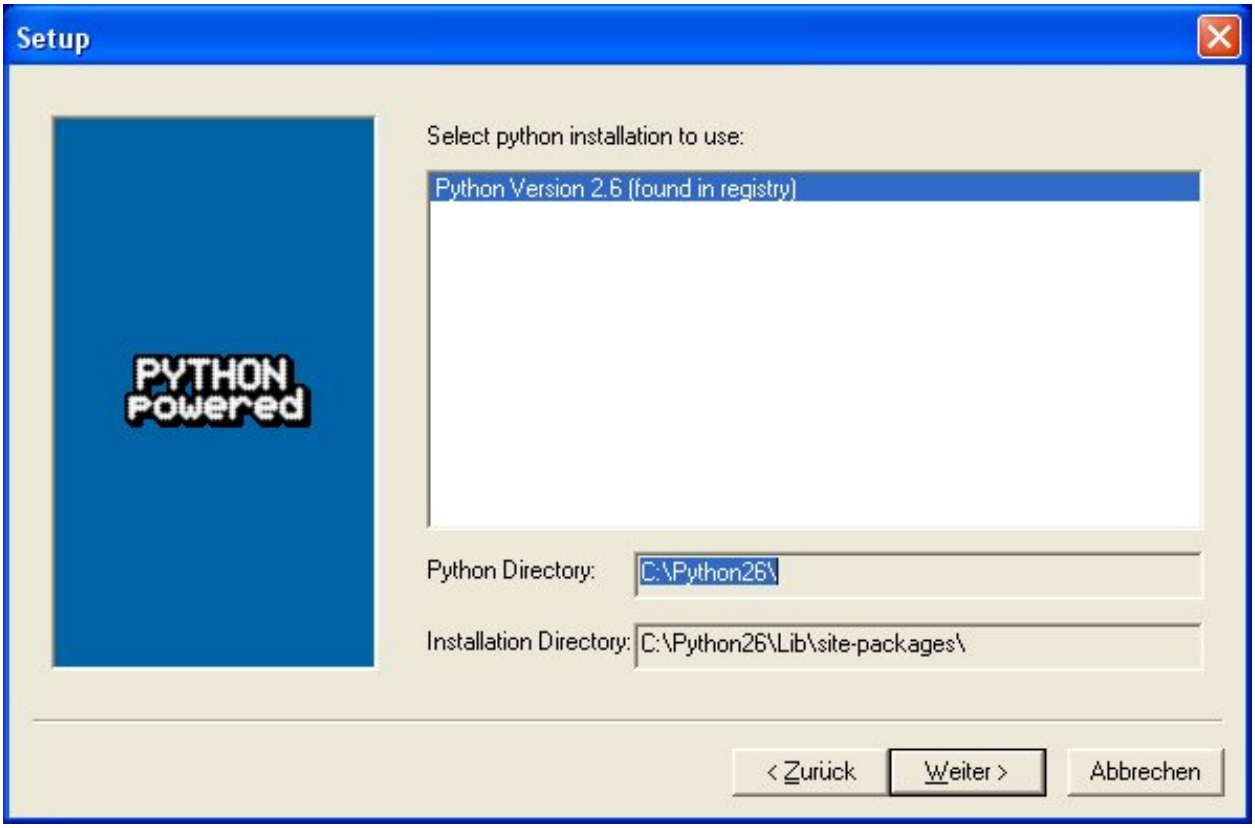

Figure 2.2: PyMbs installer

IPython(x,y) which will start IPython (see figure *Check your installation*, left). As soon as IPython has started a line saying 'In [1]: ' will appear. Now type: *'from PyMbs.Input import *'* (without the ''). If there is no error message and it jumps directly to 'In [2]: ' then the installation was successful (see figure *Check your installation*, right).

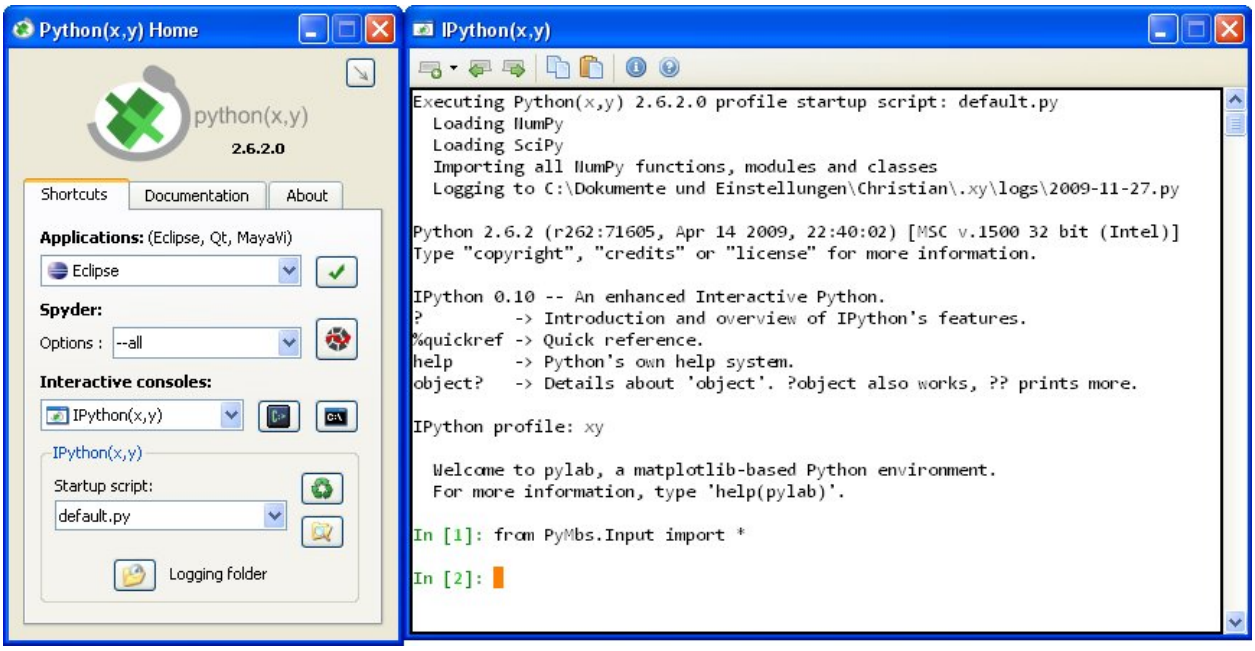

Figure 2.3: Check your installation

2.2.3 Linux

There is a Python(x,y) Ubuntu project, which provides a repository including a bundled Python distribution. Anyway there might be repositories for Python and its modules to be installed by using the package manager of your specific distribution. Just make shure that all required modules are installed on your system. Copy the PyMbs package to a folder of your choice and add it to the PYTHONPATH.

2.2.4 Mac OS

As we don't have a Mac available for testing, we can't give any instructions for installing. Basically, if you have Python and its modules running, you only need to copy the PyMbs package to a folder of your choice and add it to the PYTHONPATH.

2.3 A First Example

2.3.1 Mechanical System

In order to demonstrate the usage of PyMbs a simple exemplary system shall be modeled. Consider the system of a crane crab given in figure *PyMbs visualisation of the crane crab*.

2.3.2 Model Description

Next the model of the Crane Crab is generated using PyMbs.

- Start PyScripter (or your favourite Python editor) and copy the following code into the editor window. Please note that some systems mess up the apostrophe '?! If that is the case it is marked as red and has to be replaced manually by a proper one.:

```
# import PyMbs
from PyMbs.Input import *

# set up inertial frame
world=MbsSystem([0,0,-1])

# add inputs and parameters
F=world.addInput('DrivingForce', 'F')
m1=world.addParam('mass 1', 'm1', 1.0)
m2=world.addParam('mass 2', 'm2', 1.0)
l2=world.addParam('lengthOfRod 2', 'l2', 1.0)
I2=world.addParam('inertia 2', 'I2', m2*l2**2/12)

# add bodies
crab=world.addBody('Crab', mass=m1)
pend=world.addBody('Pendulum', mass=m2, inertia=diag([0,I2,0]))
pend.addFrame('joint' , [0, 0, l2])
pend.addFrame('middle', [0, 0, l2/2], rotMat(pi/2,'x'))

# add joints
world.addJoint('TransCrab', world, crab, 'Tx', 1.0)
world.addJoint('RotPendulum', crab, pend.joint, 'Ry', -1.0)

# add load element and sensor
world.addLoad.PtPForce(crab, world, F)
world.addSensor.Distance(crab, world, 'd')

# add visualisation
world.addVisualisation.Box(crab, 1, 0.5, 0.1)
world.addVisualisation.Cylinder(pend.middle, 0.01, 1)
world.addVisualisation.Sphere(pend, 0.1)

# generate equations
world.genEquations(explicit=True)

# generate simulation code
world.genCode('mo', 'CraneCrab')
world.genCode('m', 'CraneCrab')
world.genCode('py', 'CraneCrab')

# show system
world.show('CraneCrab')
```

- Once you have done this, you can run the model by clicking on the button with a green arrow inside. After a short moment you should see a screen showing the crane crab (figure *PyMbs visualisation of the crane crab*). You may use the sliders on the left to move the crane crab and the pendulum which can be used for checking the kinematics of your assembly. In case you receive a Syntax Error you might have to replace the inverted commas by 'proper ones'. Also note that, due to the fact that PyScripter does not properly reinitialise its Python engine, it might help to restart it as soon as you receive errors you cannot explain.

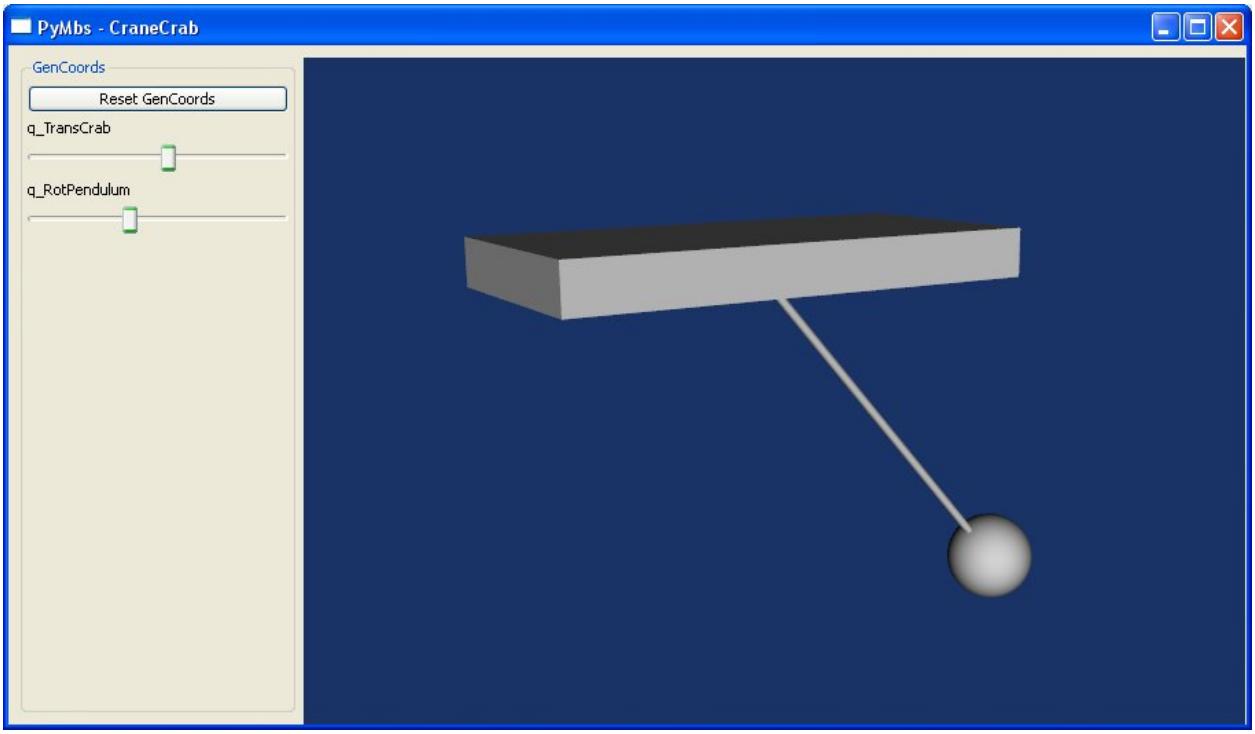

Figure 2.4: PyMbs visualisation of the crane crab

2.3.3 Code Export

Python

The command `world.genCode('py', 'CraneCrab')` is used to export the equations of motion into Python format. The generated module *CraneCrab_der_state.py* includes the function *CraneCrab_der_state(t, y)* which calculates the state derivative from a given state. This can be used in combination with any standard numerical integrator, which is able to solve differential equations of the form

$$\dot{y} = f(y, t)$$

where y is the state vector and t the time.

Modelica

The Modelica code generator, accessible through `world.genCode('mo','CraneCrab')`, creates a file called *CraneCrab_der_state.mo*. It can be used in combination with any Standard Modelica tool such as OpenModelica (http://www.openmodelica.org) or JModelica (http://www.jmodelica.org). Note, that this special model is defined partial since no equation for the input F is given. Usually, the driving force F is calculated directly inside Modelica using the Modelica Standard Library (MSL). In order to combine this model with the MSL, it is recommended to write another Modelica model by hand as given in the listing below which inherits from the automatically generated model and simply extends it by a mechanical connector:

```
model CraneCrab
extends CraneCrab_der_state;
        import Modelica.Mechanics.Translational.*;
        Interfaces.Flange_b flange;
equation
```

```
        flange.f = F;
        flange.s = d[1];
end CraneCrab;
```

Matlab

The MATLAB code generation is more involved since five different files are generated

CraneCrab_sim.m Basic simulation file. It defines the initial values, start time and stop time and calls the solver.

CraneCrab_der_state.m This file features the calculation of all state derivatives of the form

$$\dot{y} = f\left(t, y\right)$$

CraneCrab_inputs.m This function is called from CraneCrab_der_state.m. Here one can implement its own algorithms to generate inputs to the system.

CraneCrab_sensors.m PyMbs separates the calculation of sensor values from the state derivatives. The function within CraneCrab_sensors returns a struct containing all sensor values if passed a state vector, i.e.

$$S = g\left(t, y\right)$$

CraneCrab_visual.m This file can be used to visualise the system according to the description in the PyMbs model description. It features a function that takes the result from the MATLAB Solvers, i.e. a vector T containing all time values and a matrix Y where each row is a state vector. The column corresponds to the time value in T. There is third parameter fak which can be used to slow the visualisation down if chosen greater than one.

CHAPTER

THREE

COMMAND REFERENCE

3.1 General

3.1.1 PyMbs.Input

PyMbs.Input.MbsSystem provides the user interface for PyMbs.

The task of setting up a multibody system is divided into six parts, to which the corresponding functions are given below.

1. Initialisation
   - PyMbs.Input.MbsSystem
   - PyMbs.Input.MbsSystem.addParam()
   - PyMbs.Input.MbsSystem.addInput()
   - PyMbs.Input.MbsSystem.setGravity()
2. Create Bodies and Coordinate Systems, i.e. Frames or Connectors
   - PyMbs.Input.MbsSystem.addBody()
   - PyMbs.Input.MbsSystem.addFrame()
3. Connect Bodies
   - PyMbs.Input.MbsSystem.addJoint()
   - PyMbs.Input.MbsSystem.addConstraint()
   - PyMbs.Input.MbsSystem.addLoop
4. Add Load Elements, Sensors and Expressions
   - PyMbs.Input.MbsSystem.addLoad
   - PyMbs.Input.MbsSystem.addSensor
   - PyMbs.Input.MbsSystem.addExpression()
5. Add Visualisation
   - PyMbs.Input.MbsSystem.addVisualisation
6. Calculate Equations of Motion and Generate Code
   - PyMbs.Input.MbsSystem.genEquations()
   - **PyMbs.Input.MbsSystem.genCode()**

– PyMbs.Input.MbsSystem.show()

In the following the members of the module PyMbs.Input is explained in more detail.

PyMbs.Input.MbsSystem

class PyMbs.Input.**MbsSystem**(*gravity_vect=[0, 0, -1], name='world'*)

the "world" which is a kind of a body

manages (i.e. mainly collects) all these Elements: bodies, joints, loads, loops, sensors, constraints, expressions

gives access to the transformation module which is the user interface to the functionality of PyMbs

addBody(*mass*, *cg=matrix(zeros((3, 1)))*, *inertia=matrix(zeros((3, 3)))*, *name=None*)

With *addBody* you can insert a rigid body to your multibody system. TODO: Mentioning of body fixed frame.

Parameters

- **name** (*String.*) – Name of the body.
- **mass** (float/int or PyMbs.Symbolics.Symbol as returned by PyMbs.Input.MbsSystem.addParam()) – Mass of the body in kg.
- **cg** (3x1 Vector (PyMbs.Symbolics.Matrix or list of float/int).) – Vector, describing the position of the Centre of Gravity with respect to the body fixed frame.
- **inertia** (You can either specify a matrix (which must be symmetric) or pass a 6x1 vector having the elements [I_{xx}, I_{xy}, I_{yy}, I_{xz}, I_{yz}, I_{zz}]) – Tensor of inertia at the centre of gravity for this body written with respect to the body fixed frame.

Returns Reference to the generated Body object

Return type PyMbs.Input.Body

Note: A newly generated body needs to be connected to other bodies via a joint. (PyMbs.Input.MbsSystem.addJoint())

---

addConstraint(*CS1, CS2, transLock=[0, 0, 0], rotLock=[0, 0, 0], name=None*)

One way of closing kinematic loops is through the use constraints. This will introduce constraint forces at both coordinate systems as well as a set of constraint equations inhibiting certain relative movements which are specified by *transLock* and *rotLock*. This will ultimately lead to formulation as a DAE (Differential Algebraic Equations).

Parameters

- **name** (*String.*) – Name of the constraint.
- **CS1** (*MbsSystem, Body or Coordinate System.*) – Parent coordinate system on the parent body.
- **CS2** (*MbsSystem, Body or Coordinate System.*) – Child coordinate system on the child body.
- **transLock** – Vector [x,y,z] that defines which relative translations are forbidden. Use 0 to allow relative motion along the corresponding axis and 1 to inhibit it.
- **rotLock** – Vector [x,y,z] that defines which relative rotations are forbidden. Use 0 to allow relative motion around the corresponding axis and 1 to inhibit it.

addContactSensorsLoads(*name*, *CS_C*)

Add all sensors and force elements needed for a contact: * Position * Orientation * Velocity * Angular Velocity * CmpForce with 3x1 vector input * CmpTorque with 3x1 vector input

Parameters

- **name** – Name of this sensors (using r_, T_, v_, om_ as prefixes)
- **CS_C** (*Body or Coordinate System*) – Contact-Frame

addExpression(*symbol_str*, *exp*, *name=None*, *category=64*, *positive=False*, *negative=False*, *limits=None*)

Add an expression to the system of equations. Expressions allow the user to include calculations of, say, forces from sensors and inputs, or to define own sensors. All symbols known to PyMbs may be used in such an expression.

Parameters

- **name** (*String.*) – Name of the expression.
- **symbol** (*String.*) – Symbol that shall be used for the parameter.
- **exp** (`PyMbs.Symbolics.Basic`) – Term to calculate the expression, like c*x[0]+d*x[1]
- **category** (*(List of) Strings.*) – (Optional) Each expression may be categorised. Use VarKind.Sensor, to let PyMbs recognise it as a sensor, for example.
- **positive** (*Boolean.*) – (Optional) Let PyMbs.Symbolics assume that it is positive, which makes it easier to simplify.

Returns Symbol of the expression.

Return type `PyMbs.Symbolics.Symbol`

addGraphicSensors(*cs*, *name=None*, *category=256*)

You may use this function to manually add a Position and a Orientation sensor for a Coordinate System.

Parameters

- **name** (*String.*) – Name of the visualisation sensors (r_ and T_ are used as prefix for the position and orientation sensor, respectively)
- **cs** (*MbsSystem, Body, Frame.*) – Coordinate System from which the position and orientation shall be measured.
- **category** (*list of Strings.*) – May be used to manually override the category (not recommended)

Returns list of symbols for position and orientation.

Return type list of `PyMbs.Symbolics.Symbol`

addHydCylSensorsLoads(*name*, *CS1*, *CS2*)

Add all Sensors and Load Elements needed for a hydraulic cylinder: * DistanceSensor (distance and velocity at which the distance changes) * PtP-Force with scalar input

Parameters

- **CS1** (*Body or Coordinate System*) – Coordinate System at the bottom of a hydraulic cylinder
- **CS2** (*Body or Coordinate System*) – Coordinate System at the end of the rod

addInput (*symbol_str*, *shape=None*, *limits=None*, *name=None*)

Add an input to the system. It may be used for values that are calculated externally, like tire forces. Its implementation depends on the format the generated code is exported. Look at the corresponding documentation for more information.

Parameters

- **name** (*String.*) – Name of the input.
- **symbol_str** (*String.*) – Symbol that shall be used for the input.
- **shape** (*List of two numbers (int/float).*) – Optional parameter, defining the shape of the input. Use (m,n) if it is a m x n Matrix. By default (1,1), i.e. a scalar input is assumed.
- **limits** (*List of (int/float), e.g. [-10, 10]*) – Limits for the corresponding slider in the simulation tab. Default is [-1,1]

Returns Symbol representing the input, which may consequently be used in expressions (`PyMbs.Input.MbsSystem.addExpression()`) and loads (ref:'Loads').

Return type `PyMbs.Symbolics.Symbol`

addJoint (*CS1*, *CS2*, *dofList=None*, *startVals=None*, *startVals_d=None*, *name=None*)

addJoint introduces a joint linking two bodies together. It is defined using two coordinate systems which define the position on each body. The parent coordinate system (body) should be the body which is closest to the inertial frame (world/MbsSystem). The movements allowed by this joint are defined by supplying a *dofList*. It is a list of strings where each element consists of the type: * Translational (T) * Rotational (R) and the axis of the parent coordinate system: * x, y, z. Thus a translational joint along the x-axis would be abbreviated by 'Tx'. A rotational joint around the z-axis would be 'Rz'. By supplying a list of such elementary rotations and translations more complex joints may be defined. A 6dof joint using cardan-angles, for example, is characterised by *dofList* = ['Tx', 'Ty', 'Tz', 'Rx', 'Ry', 'Rz']. Note that each elementary movement is carried out consecutively. In addition initial values, both for the position and the velocity, may be supplied for each elementary movement.

Parameters

- **name** (*String.*) – Name of the joint.
- **CS1** (*MbsSystem, Body or Coordinate System.*) – Parent coordinate system on the parent body.
- **CS2** (*Body or Coordinate System.*) – Child coordinate system on the child body.
- **dofList** (*List of Strings.*) – List of strings describing the allowed elementary movements. 'Ty' stands for translation along the y-axis. 'Rx' allows rotations around the x-axis. dofList = [], creates a fixed joint.
- **startVals** (List of int/float. Dimension must be the same as *dofList*) – Initial position for each elementary movement.
- **startVals_d** (List of int/float. Dimension must be the same as *dofList*) – Initial velocity for each elementary movement.

Returns Reference to the newly generated joint.

Return type `PyMbs.Input.Joint`

Note: A joint may not be used to close a kinematic loop. This can only be achieved by using Loops (*Explicit Handling of Kinematic Loops*) or Constraints (`PyMbs.Input.MbsSystem.addConstraint()`)

---

addJointLoad(*loadSymb*, *thejoint*, *name=None*)
Adds a load object to the joint. The load is a force or a torqe, depending on the type of the joint

Returns None

addMotionSystemSensors(*CS_D*, *name='BWS'*)
Add all sensors that are needed for the motion platform, i.e. * Position * Orientation * Velocity * Angular Velocity * Acceleration * Angular Acceleration

Parameters **CS_D** (*Body, Coordinate System*) – Coordinate System where the driver sits

addParam(*symbol_str*, *defaultValue*, *positive=False*, *negative=False*, *name=None*)
Creates a new parameter. A parameter is considered to be a value which is constant during simulation but may vary between different simulation runs. The dimension of a parameter is determined by its *defaultValue*. Furthermore, it is possible to supply a *positive* flag which, when set, allows simplifications like $\sqrt{x^2} = x$, otherwise it would be $|x|$.

Parameters

- **name** (*String.*) – Name of the parameter.
- **symbol_str** (*String.*) – Symbol that shall be used for the parameter.
- **defaultValue** (*int/float, list of int/float, symbolic term.*) – Value which is used by default. Note that this option defines the dimension of this parameter. The defaultValue may also be formula consisting of parameters.
- **positive** (*Boolean.*) – Will the parameter always be positive, which holds true for a mass, for example.

Returns Symbol to the parameter.

Return type `PyMbs.Symbolics.Symbol`

addTyreSensorsLoads(*name*, *CS_C*, *CS_R*)
Add all sensors and force elements needed for a tire model: * Position * Orientation * Velocity * Angular Velocity * CmpForce with 3x1 vector input * CmpTorque with 3x1 vector input

Parameters

- **name** – Name of this sensors (using r_, T_, v_, om_ as prefixes)
- **CS_C** (*Body or Coordinate System*) – Carrier-Frame - Wheel centre, attached to the vehicle (does not rotate with the wheel)
- **CS_R** – Rim-Frame - Wheel centre, attached to the wheel (does rotate with the wheel)

exportGraphReps(*fileName*)
Create a text file for saving the information / parameters of the graphical representations of the system. Data is stored plain text in a *.cfg file which can be used by other tools.

Parameters **fileName** (*String.*) – Path and file name for saving the generated ouput.

genMatlabAnimation(*modelname*, *dirname='.'*, ***kwargs*)
Use *genMatlabAnimation* to generate a MATLAB Animation file which will visualise your graphical objects using the simulation results from the MATLAB script which has been generated using *genCode*.

Parameters

- **modelname** (*String.*) – Name of the model - will be used as filename.
- **dirname** (*String.*) – Directory where the MATLAB-Files will be put.
- **kwargs** (*Dictionary.*) – Additional Settings may be passed as kwargs: * axislimits = list of six values [Xmin, Xmax, Ymin, Ymax, Zmin, Zmax]

genSarturisXml(*modelname*, *dirname='.'*)
This function writes an XML-File for the Simulation Framework SARTURIS (ref:'http://www.sarturis.de') developed at the institute of mobile machinery at Dresden University. The XML-File includes a sample definition of the HDAE-System as well as all graphical nodes for OpenSceneGraph. In order to use this output lxml needs to be installed.

Parameters

- **modelname** (*String.*) – Name of the model - will be used as filename.
- **dirname** (*String.*) – Directory where the XML-File will be put.

setGravity(*value*)
setGravity may be used to change the value of the gravitational acceleration which is set by default to 9.81.

Parameters **value** (*float or integer*) – New value of gravitational acceleration

Returns Nothing

Note: With this function only the value but not the direction of gravity can be changed. Changing direction can be achieved through the constructor of `PyMbs.Input.MbsSystem`

---

setJointRange(*joint*, *minVal*, *maxVal*)
Use this function to set the range in which the joint coordinate can be adjusted by the corresponding slider in the GUI.

Parameters

- **joint** (*Joint-Object*) – Reference to joint
- **minVal** (*int/float*) – Minimum Value
- **maxVal** (*int/float*) – Maximum Value

show(*modelname=None*, ***kwargs*)
Produces a GUI (graphical user interface) in which your model will be displayed as long as you provided graphic objects using addVisualisation (ref:'addVisualisation'). Next to the model there will be sliders with which you can manipulate each joint coordinate directly and thus test your assembly.

Please note that you must call genEquations (:ref:`PyMbs.Input.MbsSystem.genCode()`) before you may call show!

Parameters **modelname** (*String.*) – Name of the model - will be displayed in the title bar.

3.2 Kinematic Loops

3.2.1 Loops

class `PyMbs.Input.MbsSystem.`**AddLoop**(*world*)
Class that provides functions to create loops

CrankSlider(*CS1*, *CS2*, *name=None*)
Close a loop which we call a CrankSlider. It is a planar mechanism consisting of a series of three revolute and one translational joints. Its best example is the piston in an engine.

Parameters

- **CS1** (*Coordinate System, Body or MbsSystem.*) – Reference to parent coordinate system / parent frame.

- **CS2** (*Coordinate System, Body or MbsSystem.*) – Reference to child coordinate system / child frame.
- **name** (*string*) – A name may be assigned to each loop. If no name is given, then a name like loop_1 is generated automatically. The name is used for code generation only, i.e. the symbols connected with this force will contain the name.

ExpJoint(*j*, *exp*, *name=None*)

The ExpJoint allows the user to provide an expression for the joint coordinate. Currently, only kinematic calculations is supported.

Parameters

- **j** (*Joint-Object.*) – joint.
- **exp** (*Expression.*) – Expression for joint coordinate
- **name** (*string*) – A name may be assigned to each loop. If no name is given, then a name like loop_1 is generated automatically. The name is used for code generation only, i.e. the symbols connected with this force will contain the name.

FourBar(*CS1*, *CS2*, *posture=1*, *name=None*)

Handles a classic planar four bar linkage mechanism comprising four revolute joints. The *posture* parameter specifies which solution shall be used (crossing beams or not).

Parameters

- **CS1** (*Coordinate System, Body or MbsSystem.*) – Reference to parent coordinate system / parent frame.
- **CS2** (*Coordinate System, Body or MbsSystem.*) – Reference to child coordinate system / child frame.
- **posture** (*int - either 1 or -1*) – Specifying the solution which shall be used - most time it means, if beams are crossed
- **name** (*string*) – A name may be assigned to each loop. If no name is given, then a name like loop_1 is generated automatically. The name is used for code generation only, i.e. the symbols connected with this force will contain the name.

FourBarTrans(*CS1*, *CS2*, *posture=1*, *name=None*)

The FourBarTrans is an extension to the FourBarLinkage. Whereas the classical four bar linkage consists of only four revolute joints, the FourBarTrans is extended by a translational joint. This mechanism is often found in wheel loaders.

Parameters

- **CS1** (*Coordinate System, Body or MbsSystem.*) – Reference to parent coordinate system / parent frame.
- **CS2** (*Coordinate System, Body or MbsSystem.*) – Reference to child coordinate system / child frame.
- **name** (*string*) – A name may be assigned to each loop. If no name is given, then a name like loop_1 is generated automatically. The name is used for code generation only, i.e. the symbols connected with this force will contain the name.

Hexapod(*CS1*, *CS2*, *name=None*)

Close a loop which we call a ThreeBarTrans. It is a planar mechanism consisting of a series of three revolute and one translational joints. It often occurs in conjunction with hydraulic cylinders.

Parameters

- **CS1** (*Coordinate System, Body or MbsSystem.*) – Reference to parent coordinate system / parent frame.
- **CS2** (*Coordinate System, Body or MbsSystem.*) – Reference to child coordinate system / child frame.
- **name** (*string*) – A name may be assigned to each loop. If no name is given, then a name like loop_1 is generated automatically. The name is used for code generation only, i.e. the symbols connected with this force will contain the name.

Hexapod_m_AV(*CS1*, *CS2*, *name=None*)

Close a loop which we call a ThreeBarTrans. It is a planar mechanism consisting of a series of three revolute and one translational joints. It often occurs in conjunction with hydraulic cylinders.

Parameters

- **CS1** (*Coordinate System, Body or MbsSystem.*) – Reference to parent coordinate system / parent frame.
- **CS2** (*Coordinate System, Body or MbsSystem.*) – Reference to child coordinate system / child frame.
- **name** (*string*) – A name may be assigned to each loop. If no name is given, then a name like loop_1 is generated automatically. The name is used for code generation only, i.e. the symbols connected with this force will contain the name.

Steering(*CS1*, *CS2*, *setUp=1*, *name=None*)

Close a loop which we call Steering. More detailed explanation follows.

Parameters

- **CS1** (*Coordinate System, Body or MbsSystem.*) – Reference to parent coordinate system / parent frame.
- **CS2** (*Coordinate System, Body or MbsSystem.*) – Reference to child coordinate system / child frame.
- **setUp** (*int - either 1 or -1*) – Specifying the solution which shall be used - most time it means, if beams are crossed
- **name** (*string*) – A name may be assigned to each loop. If no name is given, then a name like loop_1 is generated automatically. The name is used for code generation only, i.e. the symbols connected with this force will contain the name.

ThreeBarTrans(*CS1*, *CS2*, *name=None*)

Close a loop which we call a ThreeBarTrans. It is a planar mechanism consisting of a series of three revolute and one translational joints. It often occurs in conjunction with hydraulic cylinders.

Parameters

- **CS1** (*Coordinate System, Body or MbsSystem.*) – Reference to parent coordinate system / parent frame.
- **CS2** (*Coordinate System, Body or MbsSystem.*) – Reference to child coordinate system / child frame.
- **name** (*string*) – A name may be assigned to each loop. If no name is given, then a name like loop_1 is generated automatically. The name is used for code generation only, i.e. the symbols connected with this force will contain the name.

Transmission(*j1*, *j2*, *ratio=1*, *name=None*)

The transmission loop introduces a relation between two joints, such that their joint coordinates q1 and q2

are related by the following equation: j1 = *ratio* * v2; This can either be used to synchronise joints or to create a fixed joint by choosing *ratio* =0.

Parameters

- **j1** (*Joint-Object.*) – First joint.
- **j2** (*Joint-Object.*) – Second Joint.
- **ratio** (*int/float.*) – Ratio between two joints.
- **name** (*string*) – A name may be assigned to each loop. If no name is given, then a name like loop_1 is generated automatically. The name is used for code generation only, i.e. the symbols connected with this force will contain the name.

3.3 Loads (Forces and Torques)

In order to model internal as well as external forces and torques, the concept of LoadElements has been introduced. In the following all LoadElements are described which are currently available.

3.3.1 Loads

class PyMbs.Input.MbsSystem.**AddLoad**(*world*)

Class that provides functions to create load elements

CmpForce(*symbol*, *CS1*, *CS2*, *name=None*, *CSref=None*)

Use *addLoad.CmpForce* to add a vectorial force, acting between two coordinate systems. The force, specified with respect to the parent or reference frame, acts in positive direction on the parent coordinate system (CS1) and in negative direction on the child coordinate system.

Parameters

- **CS1** (*Coordinate System, Body or MbsSystem.*) – Reference to parent coordinate system / parent frame.
- **CS2** (*Coordinate System, Body or MbsSystem.*) – Reference to child coordinate system / child frame.
- **symbol** (*Expression as returned by addInput, addExpression or addSensor.*) – Symbol representing a three dimensional vector variable whose components are interpreted as force values in x, y, and z-direction. The direction of x,y and z is given by the parent frame (CS1) or by the reference frame (CSref).
- **CSref** (*Coordinate System, Body or MbsSystem.*) – Reference to reference coordinate system / reference frame.
- **name** (*string*) – A name may be assigned to each force. If no name is given, then a name like load_1 is generated automatically. The name is used for code generation only, i.e. the symbols connected with this force will contain the name.

Returns Reference to the generated LoadElement

Return type LoadElement

CmpTorque(*symbol*, *CS1*, *CS2*, *CSref=None*, *name=None*)

Use *addLoad.CmpTorque* to add a vectorial torque, acting between two coordinate systems. The torque, specified with respect to the parent or reference frame, acts in positive direction on the parent coordinate system (CS1) and in negative direction on the child coordinate system.

Parameters

- **CS1** (*Coordinate System, Body or MbsSystem.*) – Reference to parent coordinate system / parent frame.
- **CS2** (*Coordinate System, Body or MbsSystem.*) – Reference to child coordinate system / child frame.
- **symbol** (*Expression as returned by addInput, addExpression or addSensor.*) – Symbol representing a three dimensional vector variable whose components are interpreted as torque values around the x, y, and z-axis. The direction of x,y and z is given by the parent frame (CS1) or by the reference frame (CSref).
- **CSref** (*Coordinate System, Body or MbsSystem.*) – Reference to reference coordinate system / reference frame.
- **name** (*string*) – A name may be assigned to each force. If no name is given, then a name like load_1 is generated automatically. The name is used for code generation only, i.e. the symbols connected with this force will contain the name.

Returns Reference to the generated LoadElement

Return type LoadElement

`Joint` (*symbol*, *joint*, *name=None*)

Use *addLoad.Joint* to add a torque, acting on a joint. In case of a translational joint, a force has to be supplied. In case of a rotational joint, the load represents a torque.

Parameters

- **joint** (*Joint.*) – Reference to joint.
- **symbol** (*Expression as returned by addInput, addExpression or addSensor.*) – Symbol representing a scalar. Force or Torque depending on whether it is a translational or rotational joint.
- **name** (*string*) – A name may be assigned to each force. If no name is given, then a name like load_1 is generated automatically. The name is used for code generation only, i.e. the symbols connected with this force will contain the name.

Returns Reference to the generated LoadElement

Return type LoadElement

`PtPForce` (*symbol*, *CS1*, *CS2*, *name=None*)

Use *addLoad.PtPForce* to add a scalar force, acting between two coordinate systems along a connecting line. A positive force means that the coordinate systems are pushed apart.

Parameters

- **CS1** (*Coordinate System, Body or MbsSystem.*) – Reference to parent coordinate system / parent frame.
- **CS2** (*Coordinate System, Body or MbsSystem.*) – Reference to child coordinate system / child frame.
- **symbol** (*Expression as returned by addInput, addExpression or addSensor.*) – Symbol representing a scalar variable whose value is taken as force between the two coordinate systems.
- **name** (*string*) – A name may be assigned to each force. If no name is given, then a name like load_1 is generated automatically. The name is used for code generation only, i.e. the symbols connected with this force will contain the name.

Returns Reference to the generated LoadElement

Return type LoadElement

3.4 Sensors

3.4.1 Sensors

class PyMbs.Input.MbsSystem.**AddSensor**(*world*)
Class that provides functions to create sensors

Acceleration(*symbol*, *CS1*, *CS2*, *CSref=None*, *name=None*, *category=256*)
The acceleration sensor measures the relative acceleration between *CS1* and *CS2* with respect to the reference frame specified by *CSref*. If *CSref* has been omitted then *CS1* is used.

Parameters

- **CS1** (*MbsSystem, Body, Coordinate System.*) – Parent Coordinate System.
- **CS2** (*MbsSystem, Body, Coordinate System.*) – Child Coordinate System.
- **CSref** (*MbsSystem, Body, Coordinate System.*) – Reference Frame.
- **symbol** (*String.*) – Symbol representing a the sensor result.
- **name** (*String.*) – A name may be assigned to each sensor. If no name is given, then a name like sensor_1 is generated automatically.

Returns Symbol of the sensor

Return type PyMbs.Symbolics.Symbol

AngularAcceleration(*symbol*, *CS1*, *CS2*, *CSref=None*, *name=None*, *category=256*)
The AngularAcceleration sensor measures the relative angular acceleration between *CS1* and *CS2* with respect to the reference frame specified by *CSref*. If *CSref* has been omitted then *CS1* is used.

Parameters

- **CS1** (*MbsSystem, Body, Coordinate System.*) – Parent Coordinate System.
- **CS2** (*MbsSystem, Body, Coordinate System.*) – Child Coordinate System.
- **CSref** (*MbsSystem, Body, Coordinate System.*) – Reference Frame.
- **symbol** (*String.*) – Symbol representing a the sensor result.
- **name** (*String.*) – A name may be assigned to each sensor. If no name is given, then a name like sensor_1 is generated automatically.

Returns Symbol of the sensor

Return type PyMbs.Symbolics.Symbol

AngularVelocity(*symbol*, *CS1*, *CS2*, *CSref=None*, *name=None*, *category=256*)
The AngularVelocity sensor measures the relative angular velocity between *CS1* and *CS2* with respect to the reference frame specified by *CSref*. If *CSref* has been omitted then *CS1* is used.

Parameters

- **CS1** (*MbsSystem, Body, Coordinate System.*) – Parent Coordinate System.
- **CS2** (*MbsSystem, Body, Coordinate System.*) – Child Coordinate System.
- **CSref** (*MbsSystem, Body, Coordinate System.*) – Reference Frame.

- **symbol** (*String.*) – Symbol representing a the sensor result.
- **name** (*String.*) – A name may be assigned to each sensor. If no name is given, then a name like sensor_1 is generated automatically.

Returns Symbol of the sensor

Return type `PyMbs.Symbolics.Symbol`

CenterofMass(*symbol_str*, *Bodies=None*, *name=None*, *category=256*)

The CenterofMass sensor measures the total mass and the position of it. If *body* is the world (MbsSystem) then the sensor measures the total mass of the system.

Parameters

- **Bodies** – Bodies of which the total mass shall be measured. Pass the world/ MbsSystem if the total mass of all bodies shall be measured.
- **symbol** (*String.*) – Symbol representing a the sensor result.

Returns Symbol of the sensor

Return type `PyMbs.Symbolics.Symbol`

ConstraintForce(*symbol*, *joint*, *name=None*, *category=256*)

The Joint sensor returns the joint's constraint force

Parameters

- **joint** (*Joint-Object.*) – Joint of which the constraint force shall be measured.
- **symbol** (*String.*) – Symbol representing a the sensor result.

Returns Symbol of the sensor

Return type `PyMbs.Symbolics.Symbol`

ConstraintTorque(*symbol*, *joint*, *name=None*, *category=256*)

The Joint sensor returns the joint's constraint torque

Parameters

- **joint** (*Joint-Object.*) – Joint of which the constraint torque shall be measured.
- **symbol** (*String.*) – Symbol representing a the sensor result.

Returns Symbol of the sensor

Return type `PyMbs.Symbolics.Symbol`

Distance(*symbol*, *CS1*, *CS2*, *name=None*, *category=256*)

A distance sensor measures the scalar distance (always positive) between two coordinate systems. Furthermore it measures the velocity, i.e. the rate at which the distance changes with time. If you want to measure the distance vector have a look at the PositionSensor (ref:'Position').

Parameters

- **CS1** (*MbsSystem, Body, Coordinate System.*) – Parent Coordinate System.
- **CS2** (*MbsSystem, Body, Coordinate System.*) – Child Coordinate System.
- **symbol** (*String.*) – Symbol representing a the sensor result.
- **name** (*String.*) – A name may be assigned to each sensor. If no name is given, then a name like sensor_1 is generated automatically.

Returns Symbol of the sensor

Return type PyMbs.Symbolics.Symbol

Energy(*symbol_str*, *body*, *name=None*, *category=256*)

The Energy sensor measures the potential and the kinetic energy of *body*. If *body* is the world (MbsSystem) then the sensor measures the sum of the energies of each body in the system.

Parameters

- **body** (*MbsSystem, Body, Coordinate System*) – Body of which the energy shall be measured. Pass the world/ MbsSystem if the energy of all bodies shall be measured.
- **symbol** (*String.*) – Symbol representing a the sensor result.

Returns Symbol of the sensor

Return type PyMbs.Symbolics.Symbol

Joint(*symbol*, *joint*, *name=None*, *category=256*)

The Joint sensor returns the joint coordinate as well as the joint velocity.

Parameters

- **joint** (*Joint-Object.*) – Joint of which the coordinate shall be extracted.
- **symbol** (*String.*) – Symbol representing a the sensor result.

Returns Symbol of the sensor

Return type PyMbs.Symbolics.Symbol

Orientation(*symbol*, *CS1*, *CS2*, *name=None*, *category=256*)

The Orientation sensor measures the relative orientation between *CS1* and *CS2*, described by a transformation matrix.

Parameters

- **CS1** (*MbsSystem, Body, Coordinate System.*) – Parent Coordinate System.
- **CS2** (*MbsSystem, Body, Coordinate System.*) – Child Coordinate System.
- **symbol** (*String.*) – Symbol representing a the sensor result.
- **name** (*String.*) – A name may be assigned to each sensor. If no name is given, then a name like sensor_1 is generated automatically.

Returns Symbol of the sensor

Return type PyMbs.Symbolics.Symbol

Position(*symbol*, *CS1*, *CS2*, *CSref=None*, *name=None*, *category=256*)

A position sensor measuring the distance vector between *CS1* and *CS2* with the arrow pointing to *CS2*. The vector is written with respect to the reference frame specified by *CSref*. If *CSref* has been omitted then *CS1* is used.

Parameters

- **CS1** (*MbsSystem, Body, Coordinate System.*) – Parent Coordinate System.
- **CS2** (*MbsSystem, Body, Coordinate System.*) – Child Coordinate System.
- **CSref** (*MbsSystem, Body, Coordinate System.*) – Reference Frame.
- **symbol** (*String.*) – Symbol representing a the sensor result.
- **name** (*String.*) – A name may be assigned to each sensor. If no name is given, then a name like sensor_1 is generated automatically.

Returns Symbol of the sensor

Return type PyMbs.Symbolics.Symbol

Velocity(*symbol*, *CS1*, *CS2*, *CSref=None*, *name=None*, *category=256*)

The velocity sensor measures the relative velocity between *CS1* and *CS2* with respect to the reference frame specified by *CSref*. If *CSref* has been omitted then *CS1* is used.

Parameters

- **CS1** (*MbsSystem, Body, Coordinate System.*) – Parent Coordinate System.
- **CS2** (*MbsSystem, Body, Coordinate System.*) – Child Coordinate System.
- **CSref** (*MbsSystem, Body, Coordinate System.*) – Reference Frame.
- **symbol** (*String.*) – Symbol representing a the sensor result.
- **name** (*String.*) – A name may be assigned to each sensor. If no name is given, then a name like sensor_1 is generated automatically.

Returns Symbol of the sensor

Return type PyMbs.Symbolics.Symbol

3.5 Visualisation

3.5.1 Visualisation

class PyMbs.Input.MbsSystem.**AddVisualisation**(*world*)

Class that provides functions to create visual shapes

Arrow(*cs*, *size=1*, *name=None*)

Add a arrow graphical representation to a coordinate system given by *cs*.

Parameters

- **cs** (*MbsSystem, Body, Coordinate System.*) – Coordinate System to which the Graphical Object shall be attached.
- **radius** (*int/float.*) – Radius of the circular groundplane.
- **length** – Length of the Arrow.
- **name** (*String.*) – A name may be assigned to each visualistion. If no name is given, then a name like visual_1 is generated automatically. The name is used for the sensor and thus for code generation.

Box(*cs*, *length=1*, *width=1*, *height=1*, *name=None*)

Add a line object which whose dimensions are called: * x - *length* * y - *width* * z - *height* The origin of this graphical representation lies in the centre of the objet.

Parameters

- **cs** (*MbsSystem, Body, Coordinate System.*) – Coordinate System to which the Graphical Object shall be attached.
- **length** (*int/float.*) – Length (x-dimension) of the box.
- **width** (*int/float.*) – Width (y-dimension) of the box.
- **height** (*int/float.*) – Height (z-dimension) of the box.

- **name** (*String.*) – A name may be assigned to each visualistion. If no name is given, then a name like visual_1 is generated automatically. The name is used for the sensor and thus for code generation.

Cylinder(*cs*, *radius=1*, *height=1*, *res=20*, *name=None*)

Add a cylindrical graphical representation to a coordinate system given by *cs*. Its groundplane forms a circle with the radius given by *radius* and lies in the x-y-plane. The origin lies directly in the centre, at half of the height.

Parameters

- **cs** (*MbsSystem, Body, Coordinate System.*) – Coordinate System to which the Graphical Object shall be attached.
- **radius** (*int/float.*) – Radius of the circular groundplane.
- **height** (*int/float.*) – Height of the cylinder.
- **name** (*String.*) – A name may be assigned to each visualistion. If no name is given, then a name like visual_1 is generated automatically. The name is used for the sensor and thus for code generation.

File(*cs*, *fileName*, *scale=1*, *name=None*)

Attach a File-Object given by *fileName* to the Coordinate System given by *cs*. Since the python visualisation is based on VTK, all file formats that are compatible to VTK are supported. MATLAB Animations, however, only support STL-Files.

Parameters

- **cs** (*MbsSystem, Body, Coordinate System.*) – Coordinate System to which the Graphical Object shall be attached.
- **fileName** (*String.*) – Filename of the graphics file.
- **scale** (*int/float.*) – Can be used to scale down the Graphic. Use 1000 if the graphic information is stored in millimeters, for example.
- **name** (*String.*) – A name may be assigned to each visualistion. If no name is given, then a name like visual_1 is generated automatically. The name is used for the sensor and thus for code generation.

Frame(*cs*, *size=1*, *name=None*)

Attaches a graphical representation of a coordinate system. It consists of three lines, each pointing in the direction of an axis. Use size to vary the length of the lines.

The axes can be distinguished by their color: x-axis: red y-axis: yellow z-axis: green

Parameters

- **cs** (*MbsSystem, Body, Coordinate System.*) – Coordinate System to which the Graphical Object shall be attached.
- **size** (*int/float.*) – length of an axis of this coordinate system.
- **name** (*String.*) – A name may be assigned to each visualistion. If no name is given, then a name like visual_1 is generated automatically. The name is used for the sensor and thus for code generation.

Line(*cs*, *length=1*, *name=None*)

Add a line object which starts at [0,0,0] and ends at [*length*,0,0], i.e. spans in x-direction.

Parameters

- **cs** (*MbsSystem, Body, Coordinate System.*) – Coordinate System to which the Graphical Object shall be attached.
- **length** (*int/float.*) – Length of the line.
- **name** (*String.*) – A name may be assigned to each visualistion. If no name is given, then a name like visual_1 is generated automatically. The name is used for the sensor and thus for code generation.

Sphere(*cs*, *radius=1*, *res=50*, *name=None*)

Attaches a sphere with the radius *radius* to a given coordinate system *cs*. The origin lies directly in the middle of the sphere.

Parameters

- **cs** (*MbsSystem, Body, Coordinate System.*) – Coordinate System to which the Graphical Object shall be attached.
- **radius** (*int/float.*) – Radius of the sphere.
- **name** (*String.*) – A name may be assigned to each visualistion. If no name is given, then a name like visual_1 is generated automatically. The name is used for the sensor and thus for code generation.

CHAPTER

FOUR

BACKGROUND

This chapter is supposed to provide the reader with a brief overview of the mathematical and physical background.

4.1 Deriving Equations of Motion

4.1.1 Format

PyMbs translates the description based on bodies, joints and load elements into the following system of differential equations describing the equations of motion:

$$\begin{aligned} M\dot{qd} + h &= f + W\lambda \\ \dot{q} &= qd \\ \Phi(q) &= 0 \end{aligned} \tag{4.1}$$

where q is vector of the generalised positions; q_d is the vector of generalised velocities; λ is the vector of constraint forces; M is the generalised Mass Matrix; h is a vector containing all centrifugal and Coriolis forces; f comprises all internal and external loads; $\Phi\,(q)$ describes all holonomic constraints.

Currently, PyMbs features an explicit and a recursive scheme to derive the equations of motion. The explicit scheme is best used for small systems with up to four degrees of freedom. It obtains expressions for M, h, f, Phi directly. This code is very compact for small systems but the expressions become extremely long for larger systems. For that reason a recursive scheme has been implemented which exploits the kinematic structure of the system leading to many but simple equations.

4.1.2 Explicit Scheme

The explicit scheme is based on the Newton-Euler-Algorithm and the symbolic calculation capabilities of PyMbs through the use of sympy. Since an exhaustive presentation would go far beyond the scope of this document, only the essential highlights will be discussed. newline Given a rigid body i which is able to move freely in a three dimensional space, following equations describe the trajectory of its centre of gravity.

$$\begin{aligned} \underbrace{\begin{bmatrix} M_{1,i} & 0 \\ 0 & M_{2,i} \end{bmatrix}}_{M_i} \begin{bmatrix} \dot{v}_i \\ \dot{\omega}_i \end{bmatrix} + \underbrace{\begin{bmatrix} 0 \\ \tilde{\omega}_i M_{2,i} \omega_i \end{bmatrix}}_{h_i} &= \underbrace{\begin{bmatrix} f_{e,i} \\ m_{e,i} \end{bmatrix}}_{f_i} \\ v_i &= \dot{r}_i \\ \omega_i &= H_i \dot{\alpha}_i \end{aligned} \tag{4.2}$$

Where r_i and v_i describe the position and the velocity of the centre of gravity with respect to the inertial frame; ω_i represents its angular velocity with respect to a body-fixed frame; α_i describes the bodies orientation using Cardan Angles, for example. $M_{1,i}$ and $M_{2,i}$ are sub matrices of the block diagonal matrix M_i, given by

$$M_{1,i} = \begin{bmatrix} m_i & 0 & 0 \\ 0 & m_i & 0 \\ 0 & 0 & m_i \end{bmatrix}$$
$$M_{2,i} = I_i = \begin{bmatrix} I_{xx,i} & I_{xy,i} & I_{xz,i} \\ I_{xy,i} & I_{yy,i} & I_{yz,i} \\ I_{xz,i} & I_{yz,i} & I_{zz,i} \end{bmatrix}$$

with m_i being the mass and I_i being the symmetric tensor of inertia. The vector h_i contains centrifugal and Coriolis-forces; f_i comprises all internal and external forces and torques.

Usually, a single body without any constraints will not be sufficient for a model of a technical system. Multiple bodies, connected by joints or governed by constraints are needed. Based on equation (4.2) one can find a more general description for a system consisting of N bodies.

$$\begin{aligned} \underbrace{\begin{bmatrix} M_1 & 0 \\ 0 & M_2 \end{bmatrix}}_{M} \begin{bmatrix} \dot{zd}_1 \\ \dot{zd}_2 \end{bmatrix} + \underbrace{\begin{bmatrix} 0 \\ \tilde{zd}_2 M_2 zd_2 \end{bmatrix}}_{h} &= \underbrace{\begin{bmatrix} f_e \\ m_e \end{bmatrix}}_{f} \\ zd_1 &= \dot{z}_1 \\ zd_2 &= H\dot{z}_2 \end{aligned} \tag{4.3}$$

Although equation (4.3) looks very similar to equation (4.2), the definitions of the used symbols does differ

$$z_1 = \begin{bmatrix} r_1^T & r_2^T & \dots & r_N^T \end{bmatrix}^T$$
$$zd_2 = \begin{bmatrix} \omega_1^T & \omega_2^T & \dots & \omega_N^T \end{bmatrix}^T$$
$$M_1 = blkdiag\left(M_{1,1}, M_{1,2}, \dots, M_{1,N}\right)$$

M_2, f_e, m_e, α and H are generalised in a similar way.

Introducing joints between bodies leads to restrictions of their relative movements for which algebraic constraints can be found

$$\Phi\left(z_1, z_2\right) = 0.$$

These constraints are ensured by introducing corresponding constraint forces into the equations of motion acting perpendicular to the direction of allowed displacements. Thus equation (4.3) changes to

$$\begin{aligned} M \begin{bmatrix} \dot{zd}_1 \\ \dot{zd}_2 \end{bmatrix} + h &= f + \left(\frac{\partial \Phi}{\partial z}\right)^T \lambda \\ zd_1 &= \dot{z}_1 \\ zd_2 &= H\dot{z}_2 \\ \Phi &= 0 \end{aligned} \tag{4.4}$$

with

$$z = \begin{bmatrix} z_1^T & z_2^T \end{bmatrix}^T$$

where λ are the Lagrange Multipliers representing a vector of generalised forces.

In many applications, a set of minimal coordinates q can be found. They can be characterised by three essential properties

- There exist some functions F_1 and F_2 such that $z_1 = F_1(q)$ and $z_2 = F_2(q)$
- All elements of q are independent, i.e. they form a basis
- All constraints are satisfied by the q, i.e. $\Phi(q) = 0 \; \forall q$

If q exists, the following relations hold for z_1, zd_1 and its derivatives

$$\begin{aligned} z_1 &= F_1(q) \\ zd_1 &= \dot{z}_1 \\ &= \frac{\partial F_1}{\partial q}\dot{q} = J_T qd \\ \dot{zd}_1 &= J_T \dot{qd} + \dot{J}_T qd \end{aligned} \tag{4.5}$$

and for z_2, zd_2 and its derivatives

$$\begin{aligned} z_2 &= F_2(q) \\ zd_2 &= H\dot{z}_2 \\ &= H\frac{\partial F_2}{\partial q}\dot{q} \\ &= J_R qd \\ \dot{zd}_2 &= J_R \dot{qd} + \dot{J}_R qd \end{aligned} \tag{4.6}$$

Substituting $\dot{zd}_1$ and $\dot{zd}_2$ in equation (4.4) by the newly found relations, gives

$$M \underbrace{\begin{bmatrix} J_T \\ J_R \end{bmatrix}}_{J} \dot{qd} + M \begin{bmatrix} \dot{J}_T \\ \dot{J}_R \end{bmatrix} qd + h = f + \left(\frac{\partial \Phi}{\partial z}\right)^T \lambda \tag{4.7}$$

Multiplication by J^T from the left transforms the equation into the space of the minimal coordinates, eliminating all influences of constraint forces. This is due to the fact that no work is performed by constraint forces since they always act perpendicular to the allowed displacements. Thus equation (4.7) becomes

$$\underbrace{\left(J^T M J\right)}_{M_{red}} qd + \underbrace{J^T\left(M\dot{J}\dot{q} + h\right)}_{h_{red}} = \underbrace{J^T f}_{f_{red}} \tag{4.8}$$

which can be written using the newly introduced variables

$$M_{red}\dot{qd} + h_{red}\left(q, qd\right) = f_{red}\left(q, qd\right) \tag{4.9}$$

which corresponds to the general equation of motion for multibody systems (4.1).

4.1.3 Recursive Scheme

Since the explicit scheme is only suitable for small multibody systems an explicit scheme has been implemented. This has been done according to [FS] where a detailed explanation is given. Therefore only the implemented equations will be given here. For each body i following information must be provided * Mass: m_i * Inertia: I_i * Position of Centre of Gravity: l_i * Distance to all child bodies h= p_{hi}^z

The underlying assumption is that every joint has exactly one degree of freedom. More complex joints can be obtained through a combination of such simple joints. In general each joint i belonging to the body i has two parameters: * Axis of Translation: Ψ_i * Axis of Rotation: Φ_i

The equations are generated in three steps. First there is a forward loop, where all positions and velocities of each body is obtained. Below you find all equations (taken from [FS]) with i denoting the current body and h denoting its parent body. The index k runs over all ancestor bodies.

$$\begin{aligned}
\omega_i &= \omega_h + qd_i\Phi_i \\
\dot{\omega}_i^C &= \dot{\omega}_h^c + \tilde{\omega}_i\Phi_i qd_i \\
\beta_i^C &= \tilde{\omega}_i\tilde{\omega}_i + \tilde{\dot{\omega}}_i^C \\
\alpha_i^C &= \alpha_h^C + \beta_h^C p_{hi}^z + 2\tilde{\omega}_i\Psi_i qd_i \\
O_{i,k}^M &= O_{h,k}^M + \delta_{k,i}\Psi_i \\
A_{i,k}^M &= A_{h,k}^M + \tilde{O}_{h,k}^M p_{h,i}^z + \delta_{k,i}\Psi_i
\end{aligned}$$

All values are initialised with 0 except for $\alpha_0^C = -g$ with g denoting the gravity vector.

Second there is a backward loop in which all joint forces are calculated, with r denoting all direct successors (child bodies) of the body i.

$$\begin{aligned}
G_i^C &= m_i\left(\alpha_i^C + \beta_i^C l_i\right) - F_{ext,i} \\
F_i^C &= \sum F_r^C + G_i^C \\
L_i^C &= \sum\left(\left(L_r^C + \tilde{p}_{i,r}^z F_r^C\right)\right) \\
G_{i,k}^M &= m_i\left(A_{i,k}^M + \tilde{O}_{i,k}^M l_i\right) \\
F_{i,k}^M &= \sum F_{r,k}^M + G_{i,k}^M \\
L_{i,k}^M &= \sum\left(L_{r,k}^M + \tilde{p}_{i,r}^z F_{r,k}^M\right) + \tilde{l}_i G_{i,k}^M + I_i O_{i,k}^M
\end{aligned}$$

Third, there is another forward loop in which expressions for the generalised accelerations are calculated.

$$\begin{aligned}
C_i &= \Psi_i F_i^C + \Phi_i L_i^C \\
M_{i,k} = M_{k,i} &= \Psi_i F_{i,k}^M + \Phi_i L_{i,k}^M
\end{aligned}$$

The so found C_i and $M_{k,i}$ are the elements of the Mass Matrix M and a C-Vektor which satisfies the following relation

$$C = f + W\lambda - h$$

Thus, the form of equation (4.1) has been achieved.

4.2 Explicit Handling of Kinematic Loops

A unique feature of PyMbs is its ability to deal with certain kinematic loops in an explicit manner. This consequently leads to a formulation based on minimal coordinates and thereby to explicit ordinary differential equations without algebraic constraints. The underlying concept shall be presented in this section.

Currently, there are four different kinematic loops implemented as shown in Figure *PyMbs visualisation of the crane crab*.

Handling kinematic loops is based on the coordinate partitioning method [WH]. Given the vectors of generalised positions q a set of independent coordinates u and a set of dependent coordinates v are chosen according to the definitions made in the kinematic loops.

$$q = \begin{bmatrix} u \\ v \end{bmatrix}$$

First, an explicit relation between the dependent coordinates v and the independent coordinates u must be provided for each loop.

$$v = H(u)$$

Exploiting the (usually much simpler) implicit relation

$$h(u, v)$$

of the loop j, corresponding relations on velocity and acceleration level may be derived.

$$\begin{aligned} \dot{h} &= h_u \dot{u} + h_v \dot{v} \\ \dot{v} &= -h_v^{-1} h_u \dot{u} \\ &= B_{vu} \dot{u} \end{aligned}$$

with $B_{vu} = -h_v^{-1} h_u$. The same can be achieved for finding a relationship on acceleration level.

$$\begin{aligned} \ddot{h} &= h_u \ddot{u} + h_v \ddot{v} + \dot{h}_u \dot{u} + \dot{h}_v \dot{v} \\ \ddot{v} &= -h_v^{-1} \left(h_u \ddot{u} + \dot{h}_u \dot{u} + \dot{h}_v \dot{v} \right) \\ &= B_{vu} \ddot{u} + b' \end{aligned} \tag{4.10}$$

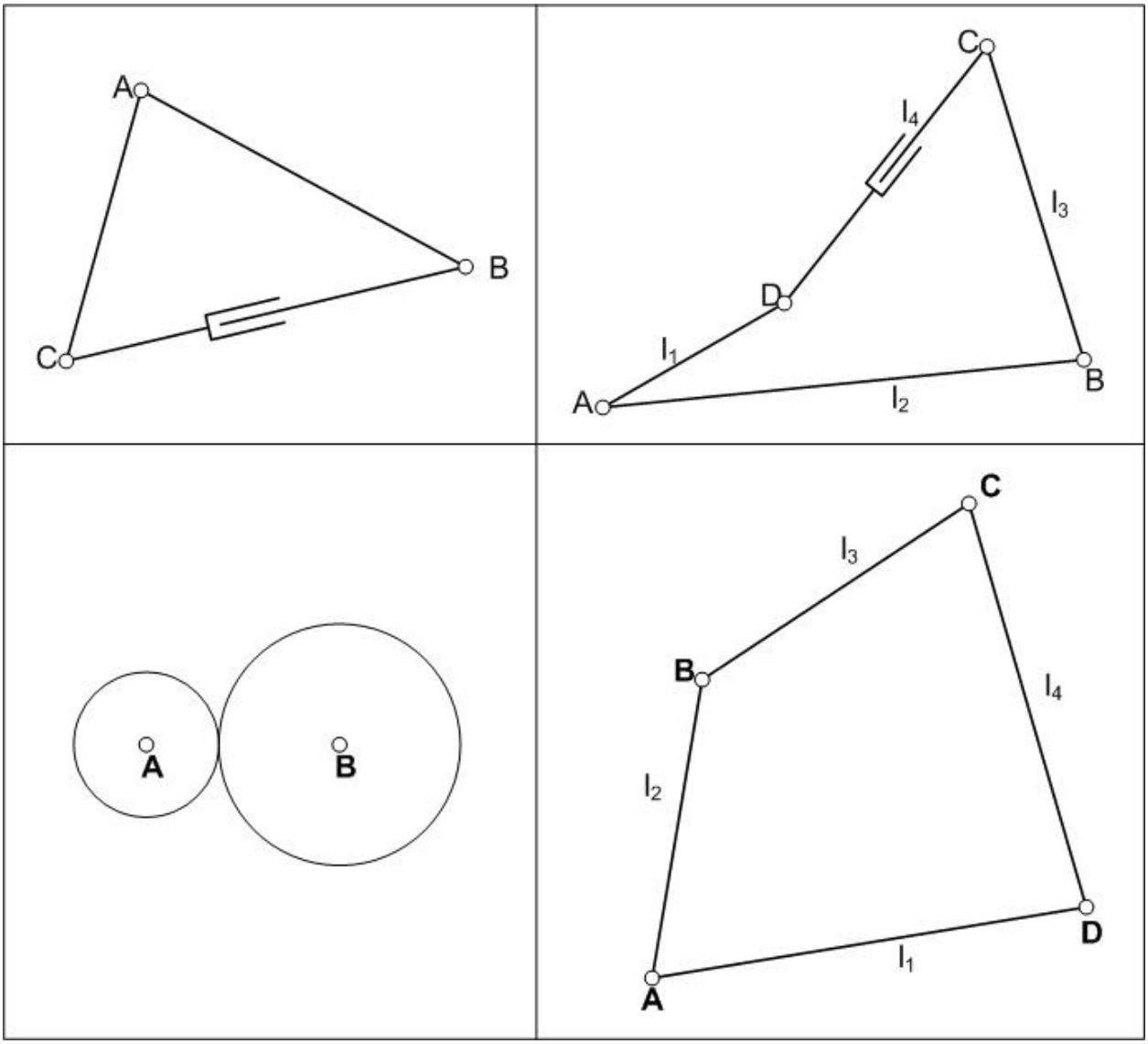

Figure 4.1: PyMbs visualisation of the crane crab

with $b' = -h_v^{-1}\left(\dot{h}_u\dot{u} + \dot{h}_v\dot{v}\right)$. Thus, the equations the equations of motion can be reduced even further, so that they only need to be solved for the set of independent coordinates. Equation (4.1) may be rewritten using equation (4.10).

$$M\left(\underbrace{\begin{bmatrix} I \\ B_{vu} \end{bmatrix}}_{J}\ddot{u} + \underbrace{\begin{bmatrix} 0 \\ b' \end{bmatrix}}_{b}\right) + h = F + W\lambda$$
$$MJ\ddot{u} + Mb + h = F + W\lambda$$

Multiplying this equation with J^T from the left yields

$$\begin{aligned} J^TMJ\ddot{u} + J^T\left(Mb + h\right) &= J^TF + \underbrace{J^TW}_{=0}\lambda \\ M_{red}\ddot{u} + h_{red} &= F_{red} \end{aligned} \tag{4.11}$$

with $M_{red} = J^TMJ^T$; $h_{red} = J^T\left(Mb + h\right)$; $F_{red} = J^TF$. Clearly, equation (4.11) has the form of (4.1) and is therefore compatible.

In order to implement explicit handling for a new kinematic loop, the following information must be provided:

- partitioning of joint coordinates into u and v
- H, such that $v = H\left(v\right)$
- B_{vu}, b', such that $\dot{v} = B_{vu}\dot{u}$ and $\ddot{v} = B_{vu}\ddot{u} + b'$

Note, that saving B_{vu} and b_{prime} directly, although they can be derived from $h\left(u, v\right) = 0$ automatically, saves time when assembling a model and leaves room for optimising these expressions towards brevity.

CHAPTER

FIVE

PYMBS ARCHITECTURE AND DESIGN

5.1 Overview

PyMbs consists of three main components: input, processing / analysis and output (see figure *Basic structure of PyMbs*), whereas the processing and analysis layer can be considered to be the heart of PyMbs.

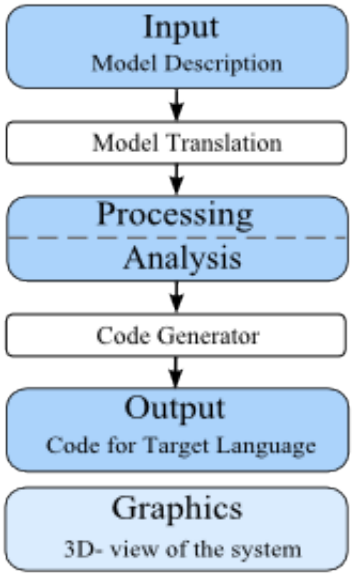

Figure 5.1: Basic structure of PyMbs

The input layer provides the user interface for the description of the mechanical system. In release 0.1, the model description is completely text based. The script created by the user is translated into an abstract syntax suitable for the generation of equations of motion by the processing module. This intermediate step prepares PyMbs for other user interfaces, i.e. graphical modeling, as any kind of model description can be used as long as an appropriate translator is provided. The analysis module simplifies and reduces the set of equations generated by the processing module and passes them to the code generator. The code generator provides a set of writers for several target languages and creates the output files. Optionally, the graphical output module evaluates a function created by the python code generator to calculate the positions and orientations of the elements of the mechanical system. The system can be visualised in an interactive scenegraph.

%TODO: Ueberarbeiten Rechtschreibung / Grammatik

5.2 Analysis

5.2.1 Introduction

The equations generated by the processing module are unsuitable for the calculation process. The generated system of equation has many equations which are not usefull to calculate the outputs of the system. Outputs of the system are this values who are interesting for the modeller. In case of PyMbs outputs are sensors. A example for "not usefull" is that many equations are equal to zero ore equal to one ore consist of only one variable. The goal is to speed up the calculation by reducing the efford. The efford can be reduced by an analyses of the system of equations on the equation level and the system level. The resultant system of equation contains only the essential steps to calculate the outputs. More precisely, the variables which are equal to zero ore equal to one may be replaced in all equations. All variables used only once may be replace with its equation. To give an example look at picture (Graph1.pdf). Given system of equations

$$\begin{aligned} a &= 1 \\ b &= 0 \\ c &= 5 \cdot k \\ d &= a \cdot c \\ e &= 2 \cdot a + 2 \\ f &= b \cdot k \\ g &= d + e + f \end{aligned} \tag{5.1}$$

with the output g, seven steps needed to calculate g. In order to reduce the system all zeros are eleminated. The second step is to do the same for all variables equal one. The last step is to replace all variables used only once. The resultant system of equations reads as follows

$$\begin{aligned} c &= 5 \cdot k \\ g &= c + 4 \end{aligned} \tag{5.2}$$

Now, only two steps are required to calculate g.

Furthermore an abstract syntax tree of the system of equations is useful to calculate the dimensions from all variables and to sort the equations. Not all supported languages and corresponding simulation tools are able to derive the dimension of an variable from the equation. To sort the equations is necesary because not all supported simulation tools are able to do this. Note, that the implemented algorithm is not able to handle loops in the system of equations. For example the given system of equation

$$\begin{aligned} a &= 5 \\ b &= c + 2a \\ c &= b + a \\ d &= b + c \end{aligned} \tag{5.3}$$

contains a loop for the variables b and c. The methods used in implementing the aformentioned tasks are described in the sequal.

5.2.2 Generating the Abstract Syntax Tree

The input from the Analyses Layer is a list of Expression objects. These objects have the following important information.

- The name of the variable called *symbol*, which represend left hand side of the expression.
- The equation, which is the right hand side of the expression.

All information is stored in an node. A node is a vertices of the abstract syntax tree. A list of all required variables for each node can be generated using a sympy function. With the list of required variables from each node the edges of the tree can be cunstructed. To store the edges every node has two list. One for the parent nodes and one for the child nodes. A parent node is a node, which is used to calculate the expression of the node. In addition a global list of all nodes exists to avoid to itterated throw the tree when searching nodes.

5.2.3 Reducing the System of Equation

The reduce algorithm has to find all nodes, which are

1. equal to zero or one or
2. have only one parent.

Because this is one of the main parts of the analysing module it should work efficient and fast.

The algorithm for the first point is divided in the following four steps. The first step is to find all nodes with no parents to extract all constants. This can be done simple by looking on the list of parents from each node. The second step is to check whether the node is replaceable. Not all nodes are replaceable, for example parameters ore sensors should not disappear. To decide wheter a node is replaceable ore not categorys were established. There are categorys for state variables, parameter variables or sensor variables. The third step is to check whether the node is equal to one or zero. If this conditions are true the last step starts and replace the node in the equations of the children and rearrange the abstract syntax tree by removing the corresponding edges. After this changes it is possible that a child node equation become equal to zero or one. Also this node should be replaced to accomplish a equation system thats suitable for calculating the outputs efficient. This four steps are executed in a recursive way for each child node.

The algorithm for the second point look similar to the algorithm for the first point. The difference is that this algorithm does not work in a recursive way. This algorithm is executed for each node of the global list of nodes.

5.2.4 Obtaining Sorted Equations

To sort the equations the nodes within the abstract syntax tree have to be marked with a level of depth. The level of depth is the longest path from a child to parent with no parents. This is done by a recursive algorithm who is executed for a list of nodes. The first list contains all nodes with no parent node. The level of depth is default set to 0 for each node. This means all nodes have the same level of depth. The algorithm starts and try to increas the level of depth for all childs from a node of the list. If no level of depth from a child is increased, the algorithm takes the next node from the list. The level of dept can be increased if the new value is grather then the old one. In addition this algorithm is used to find loops inside the system of equations. Therefore each node have a flag that indicates if the node is curently visited. If the flag is active and it will be visited again a loop is detected and a error message is thrown. If the level of depth is set for all nodes the next step is to collect all nodes according to the desired category. Therefor a loop runs throw the global list of nodes and collect the required nodes in a list. The next step is to collect all parents from the nodes inside the list. This is done by an recursive algorithm and a following loop. The recursive algorithm get the list of required nodes. It steps throw all parents of the nodes and set the required flag. Afterwards the loop checks the global list of nodes and collect all nodes with aktiv requred flag. With the list of all required nodes the third step starts to arrange the required nodes in a way that every expression of a node can be calculated. Therefor every variable inside the expression has to be calculated befor. This problem is solved by the level of depth. Alle nodes with equal level of

depth are collected inside one list. Afterwards the lists are ordered by increasing level of depht and concatenate. The last step is to calculate the dimension of each expression and collect the expressions in a list according to the sequence.

5.2.5 Calculating Value and Dimension of an Equation

For some layers there is need to know the dimension of a variable. For example Models design in MODELICA or C code need the dimension for a variable. To get the dimension of a variable the expression need to be evaluated. To evaluate a expression all variables need to be known. This process is done recursive until a expression can be evaluate because it is an constant value. The expression is evaluted using sympy built in function. Also the information about the dimension of the variable is get from sympy built in functions.

5.3 Output

The output module is used to create code from a given set of equations for a specific target language. PyMbs 0.1 is capable of generating code for

- Python
- Matlab
- Modelica
- C++.

This section will explain the mechanism of code generation and enable you to create custom writers for languages not listed above. Therefore we strongly advise you to take a look at the source code of the involved classes to get a better understanding of how things work.

5.3.1 The Code Generation Process

The output module is made up of four classes: *CodeGenerator*, *Writer*, *Syntax* and *Grammar*, whereas *Writer* is the base class for all inherited language specific writers. Basically, the actual code is generated by the writer objects which know the syntax and grammar of the target language. These writer objects are instantiated by *CodeGenerator* (which passes the equations to the writer) and create lists containing the lines of code as strings. Finally, these lists are written to a file by the code generator (figure *PyMbs code generation process*).

5.3.2 Create Custom Target Writer

This section will show you how to create a custom writer class for targets not listed above. Depending on the syntax and semantics of the target language, this might be rather straightforward or require some more effort. Anyway you should have a look at the *Classdocs: PyMbs.Output* of the ouput section to get an overview of the methods and functions provided by the writer base class. You should also study the writers shipped with PyMbs to get an idea of how things work. Begin with the PyhtonWriter as this is the simplest.

First of all you need a custom writer class inheriting from *Writer*. It stores the code linewise in a list called *code*, where each line is a string and an item of the list. The code is written in the order of the list items. Your job is now to fill this list with code. To help you doing this, a template is shipped with PyMbs: Writer_temp.py. We suggest to define syntax and grammar of the target language at first. Then you may add your own methods to the writer, whereas each method should be used to create code for a single output file. Use the methods of the writer base class to write variables, constants, equations etc. Use *writeLine()*, *writeComment()* and *writeLineBreak()* to write anything else to the code.

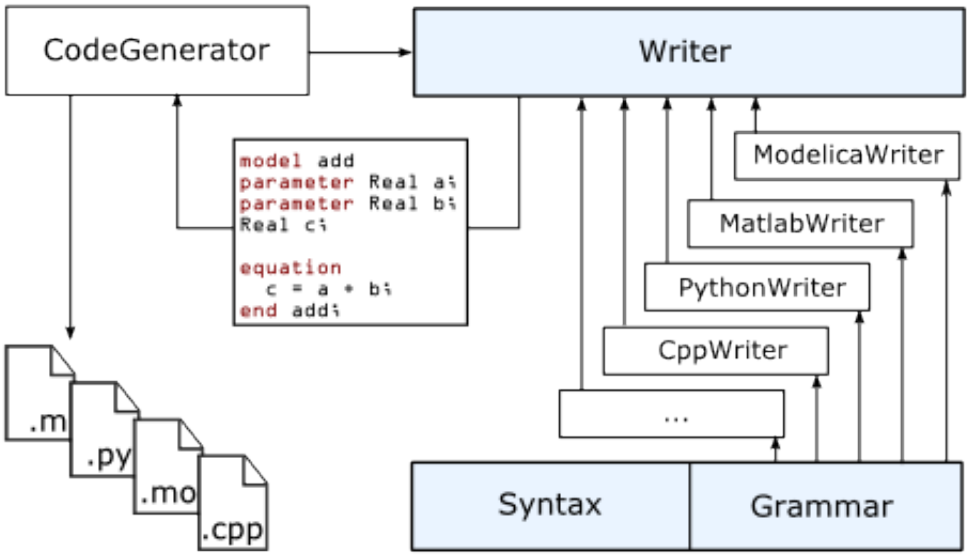

Figure 5.2: PyMbs code generation process

Note: You have to take care of correct indentation if required!

Once you're finished go to the *codeGenerator* class and add your writer to the constructor and the file extension of your target to the dialect list. Create a method ala *genMyCode(self)*, where you should create a function- and a filename. Call the write method of your writer and the code generators saveToFile() method. That's it. You may now acces your writer by the genCode() command from your script, providing the specified file extension as dialect.

5.3.3 Classdocs: `PyMbs.Output`

`PyMbs.Output.CodeGenerator`

`PyMbs.Output.Writer`

`PyMbs.Output.Syntax`

`PyMbs.Output.Grammar`

CHAPTER

SIX

TROUBLESHOOTING

In this chapter we tried to gather all problems which might occur during the usage of PyMbs anf how they can be solved.

6.1 Error: sympy could not be found

Sympy is only installed if full has been selected when installing Python(x,y). Please reinstall Python(x,y) in full mode.

6.2 Error: PyScripter throws strange errors in Gui.py

Unfortunately, PyScripter does not reinitialise its Python engine properly which can cause very strange errors. You have the following options to resolve this:

- Restart PyScripter
- Use remote Python Engine (Run -> Python Engine -> Remote) which can be reinitialised
- Use another editor (like eclipse)

These strange errors are most commonly caused by a module that has been already imported, has been changed (automatically or manually) as is the case with some parts of the GUI. The Python engine does not recognise this and does not reload the module. Hence you are still working with the old version which can lead to unexpected behaviours.

6.3 Anything Else

Please contact Christian christian.schubert@tu-dresden.de. He will be happy to assist you.

CHAPTER

SEVEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

BIBLIOGRAPHY

[FS] Fisette, P; Samin, J. C.: Symbolic generation of large multibody system dynamic equations using a new semi-explicit Newton/Euler recursive scheme. In: Archive of applied mechanics Vol. 66, No 3. (1996) pp. 187-199

[WH] Wehage, R. A.; Hung, E. I.: Generalized coordinate partitioning for dimension reduction in analysis of constrained dynamic systems. I, Mech. Design 134 (1982) pp. 247-255

PYTHON MODULE INDEX

p

INDEX