PySimulator – A Simulation and Analysis Environment in Python with Plugin Infrastructure

A. Pfeiffer, M. Hellerer, S. Hartweg, M. Otter, M. Reiner
DLR Institute of System Dynamics and Control, Oberpfaffenhofen, Germany
{Andreas.Pfeiffer, Matthias.Hellerer, Stefan.Hartweg, Martin.Otter, Matthias.Reiner}@dlr.de

Abstract

A new simulation and analysis environment in Python is introduced. The environment provides a graphical user interface for simulating different model types (currently Functional Mockup Units and Modelica Models), plotting result variables and applying simulation result analysis tools like Fast Fourier Transform. Additionally advanced tools for linear system analysis are provided that can be applied to the automatically linearized models. The modular concept of the software enables easy development of further plugins for both simulation and analysis.

Keywords: PySimulator; Python; Simulator; FMI; FMU; Modelica; Plugin; Simulation; Analysis; Linear System Analysis

1 Introduction

In this article the open source environment *PySimulator*[1] is introduced and its design is discussed. The central idea is to provide a generic framework

- to perform simulations with different simulation engines in a convenient way,
- organize the persistent storage of results,
- provide plotting and other post-processing feature such as signal processing or linear system analysis, and
- export simulation and analysis results to other environments.

1.1 Design

From an end-user's point of view, PySimulator consists of a convenient graphical user interface so that all these operations can be defined mostly with the mouse. This is similar to many other, usually commercial, simulation environments.

However, the major innovation is that PySimulator is constructed as a *plugin system*: Nearly all operations are defined as plugins with defined interfaces. Several useful plugins are already provided, but anyone can extend this environment by his/her own plugins and there is no formal difference to plugins already provided by the authors of the paper.

Introducing a new plugin means to copy a template and adapt it by writing Python code. Hereby it is possible to build upon the results of other plugins and provide own results to other plugins. All plugin functionality available via the graphical user interface shall also be easily accessible in Python scripts. This will allow a modeler to define and automatically execute Python scripts.

1.2 Related Work

There are several existing Python packages that aim to simulate dynamic systems of standardized physical models like Modelica or FMI [MC10]:

- The software package BuildingsPy [LBN+12] provides functions in Python to start simulations of Modelica models in Dymola [DS12]. Furthermore the result file can be read to process the signals.
- The OMPython package [GFR+12] interfaces the OpenModelica environment with Python. Hence, many functionalities of OpenModelica can be controlled by Python scripts.
- The packages PyFMI and Assimulo [AAF+12] provide Python interfaces for calling functions of a general Functional Mock-Up Unit. Moreover, sophisticated numerical integration algorithms are interfaced or implemented in Assimulo. The user mainly interacts with the packages by Python scripts. A graphical user interface for plotting simulation results is currently available.

These packages concentrate the functionalities on a specific type of model and simulation engine and do not provide a wide range of post-processing features.

[1] PySimulator builds on other Python packages with different license conditions. The most restrictive used is LGPL. Non-GUI functions are under the BSD license.

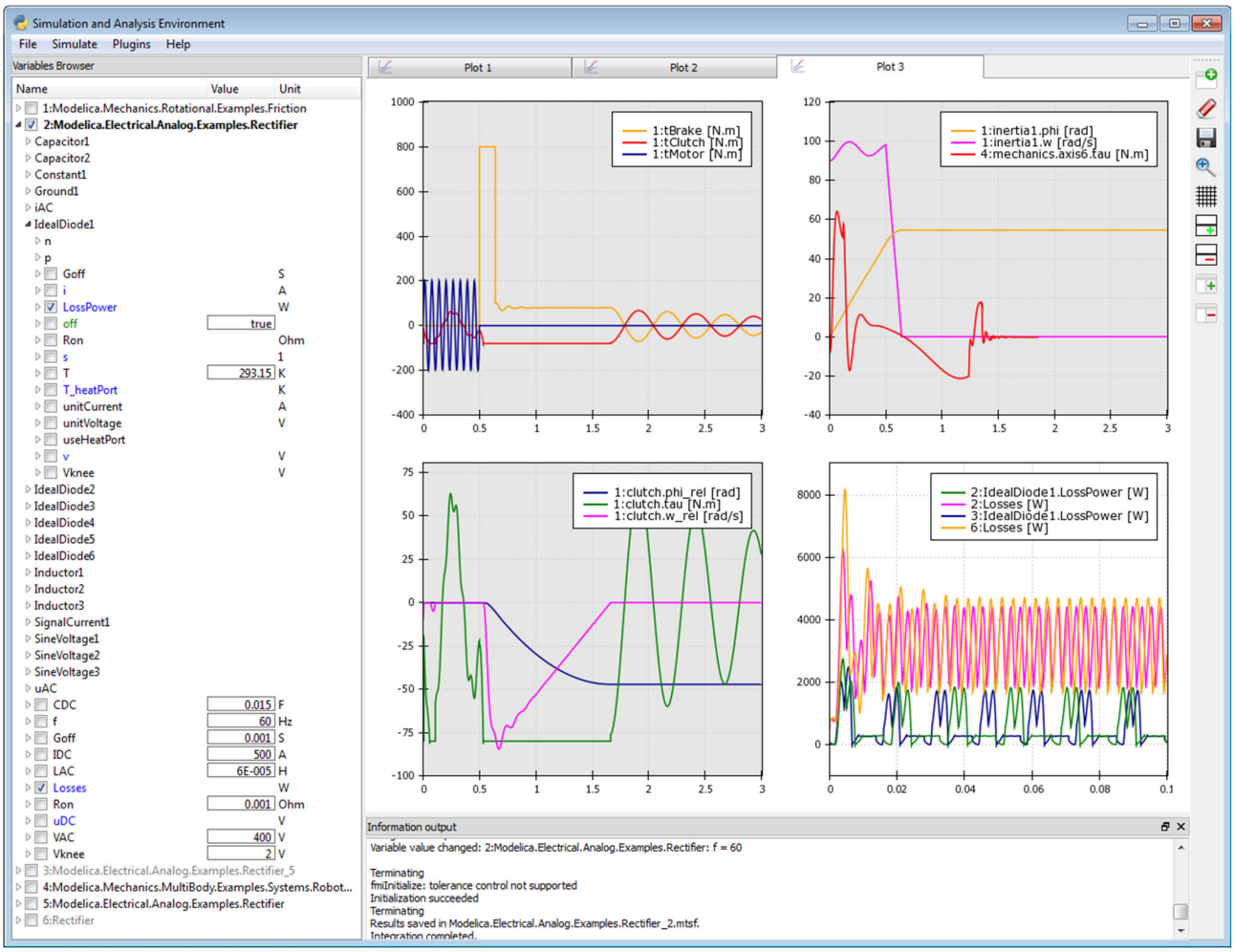

Figure 1: Main Graphical User Interface of PySimulator.

Also, the license conditions are partially restrictive since, e.g., GPL is used. In the Python package index (http://pypi.python.org) about 130 Python simulation packages are listed. Most of these packages are dedicated to the simulation of specific models (like neuron networks, biological systems, discrete event systems) or are low level generic packages that require to define a model as Python code (like Assimulo, pyDDE, ScipySim).

2 Architecture and GUI

The environment PySimulator is implemented in Python and depends on several Python packages. The Graphical User Interface (GUI) is built by PySide [P12], a Qt-Interface to Python. Plotting features are realized by integrating Chaco [E12] into the Qt [NC12] framework.

The main GUI of PySimulator (see Figure 1) has a *menu bar* on the top, the *Variable Browser*, a *plotting area* and an *Information output* window. The menu bar shown in Figure 2 provides functionalities for opening models, opening and conversion of result files, running the simulation and starting analysis tools. In the Variable Browser all models and simulation results are managed to get access to the variables, their attributes and their numeric data. Structured plots show the numeric data in the plotting area.

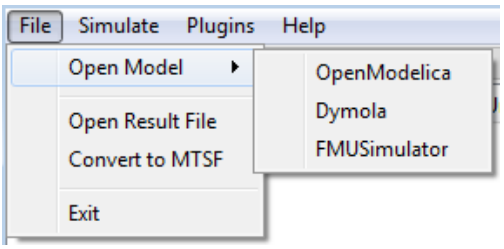

Figure 2: Menu bar of PySimulator GUI.

The environment is intended to provide features for two kinds of users. In a first step the user interactively works with the graphical user interface by loading models, simulating them, plotting variables, applying

DOI 10.3384/ecp12076523

analysis tools and inspecting the results. An advanced user can profit from Python's scripting features because it is possible to load, simulate and analyze models by API function calls in custom scripts.

The implemented software is structured as shown in Figure 3. Some main modules are hosted in the top level directory *PySimulator*. Under *Plugins* all code and data of plugins is organized. Plugin interfaces for model simulators and simulation analysis tools are provided. This plugin concept leads to very modular software that can be easily extended by further plugins. In the following subsections the main GUI elements and the modular plugin structure are presented.

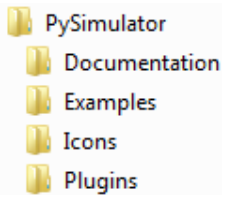

Figure 3: Main directory structure of PySimulator.

2.1 Model and Result Management

A central element of the PySimulator GUI is the *Variable Browser*, see Figure 1. It can show several variable trees of different models. Such a variable tree is either generated by opening a simulation result file, or by opening a model. To open a model the Simulator plugin has firstly to be selected in the menu *Open Model* (see in Figure 2; for more details see Section 3). Secondly the model file itself is to be specified. Each top level item in the Variable Browser has an ID number followed by a colon and the name of the model. The ID number also marks variables uniquely in plot windows.

By selecting *Open Result File* the user can load a result file of different formats into the Variable Browser. In such a case there is no model to be simulated and the item is displayed in grey color like items number 3 and 6 in Figure 5. Currently two result formats are supported: the proposed standard time series file format MTSF [PBO12] in HDF5, and the binary format generated by Dymola's [DS12] simulation executable in Matlab 4 format [M12].

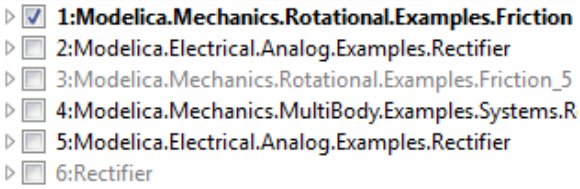

Figure 5: Top level items in the Variable Browser. Black color: Model and result file; grey color: only result file.

For each top level item an information text window (tool tip) is displayed when the user holds the mouse pointer some moment over the name of the item. The text informs about properties of the model. Its structure depends on the model type. An example for an FMU [MC10] is displayed in Figure 4.

After opening a model the variable tree is constructed according to the names of all model variables. New tree branches are introduced by variable names containing dots '.' representing a hierarchy or squared brackets '[', ']' representing arrays. The unit

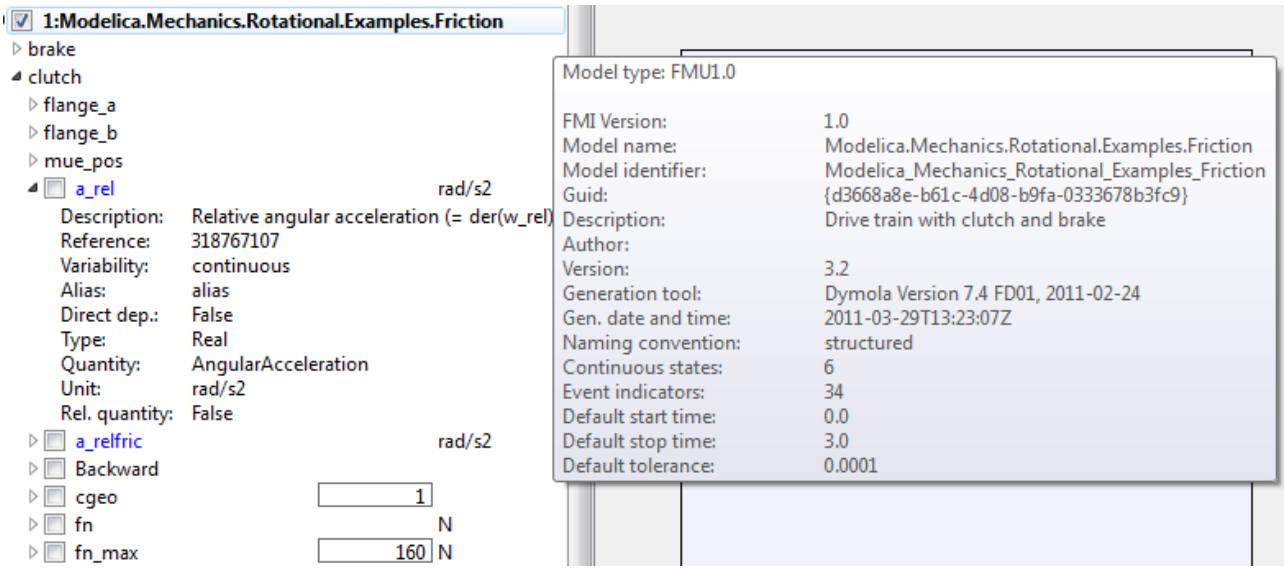

Figure 4: Variable Browser with information text (on the right) for the FMU model.

of the variable is shown if there is any. The values of independent parameters or the initial values of state variables may be edited in the Variable Browser e.g. for `1:clutch.cgeo` or `1:clutch.fn_max`. Furthermore, variables to be plotted can be defined before the numerical integration starts. The attributes of each variable depending on the model or result file type can be displayed by opening the leaf in the variable tree, e.g. for the variable `1:clutch.a_rel` in Figure 4.

During the numerical integration of a model a result file is generated that is associated with the model. A context menu *Results* for the top level items in the Variable Browser informs about the associated result file, see Figure 6. By selecting the context menu *Model* one can close a model and the associated result file. Also, the user can duplicate a loaded model. Each duplicate has its own top level item in the Variable Browser like any other model. It is based on the same model file (e.g. Friction.fmu or Friction.mo), but has a separate result file, separate settings for the numerical integration and separate values for parameters or initial values set before the numerical integration. For example, the top level item 5 in Figure 5 is a duplicate of model 2 (Rectifier model).

This approach has the advantage that comparing a reference simulation with a tuning simulation of the same original model is very easily possible. The user just duplicates the reference version of the model and experiments on the duplicate. The effects can be directly inspected in plots for the reference and the tuned version of the model.

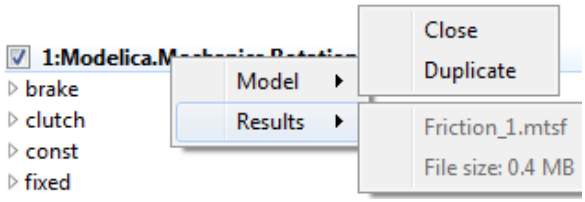

Figure 6: Standard context menus for a top level item in the Variable Browser.

2.2 Plotting Features

Simulation data and analysis results must be processed to make them comprehensible for humans. In PySimulator, the data is visualized using graphical plots. For this purpose PySimulator provides a plotting framework, based on the 2D plotting library *Chaco* [C12]. Chaco was chosen because it is compatible with the Qt/PySide UI-framework, it is licensed under the new BSD license, and it is a native Python library. The last feature not only facilitates the integration but also allows making full use of its object oriented structure at all levels of the inheritance tree for manipulation and extendibility to fit our needs. The primary advantage of Chaco and what sets it apart from other plotting solutions for Python lies in its focus on interactivity.

Plots are shown in a designated area within the main window as shown in the upper right part of Figure 1. Within this area plots can be arranged on a higher level within tabs. The tabs are then subdivided into a

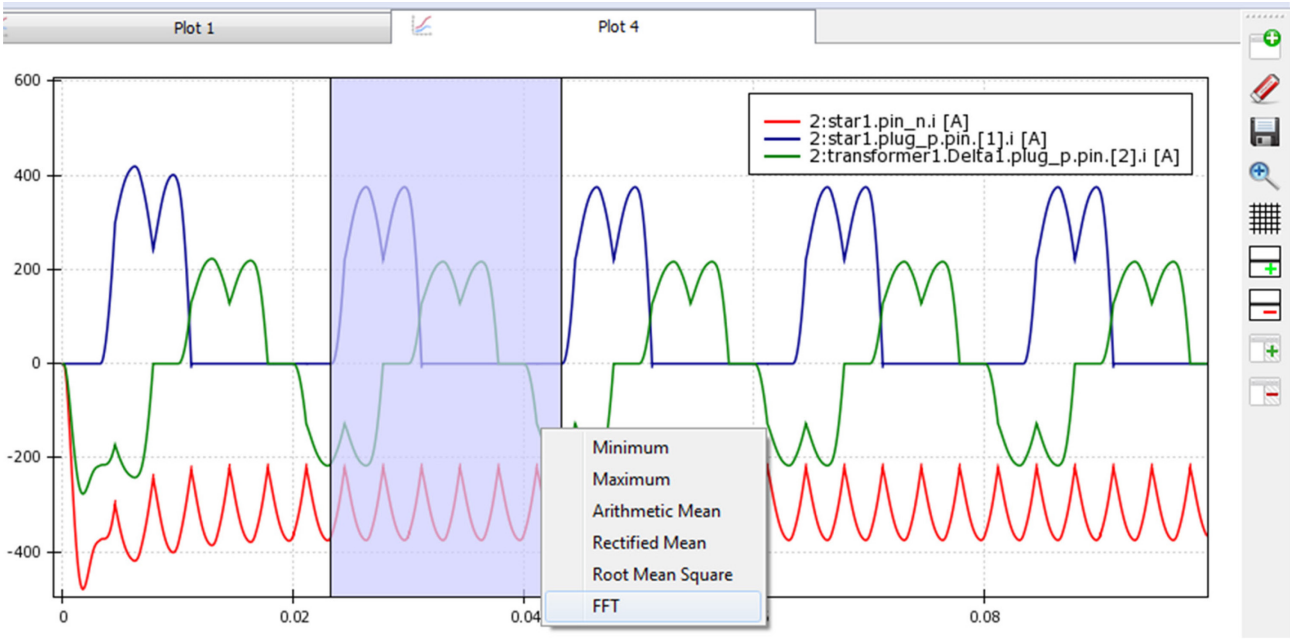

Figure 7: A plot widget displaying three variables, with a selected time interval and open context menu.

DOI 10.3384/ecp12076523

grid in which the plots are arranged. To achieve all this, a base class called *PlotWidget* was implemented that acts as an adapter between Chaco and our application or respectively Qt/PySide. All plots are supposed to be implemented as extensions of this base class.

Based on the Chaco framework and PlotWidget a default plot widget (*DefaultPlotWidget*) for displaying a variable value over time was implemented while paying special attention to the fact that future plugin developers can both easily use the existing material and still have access to Chaco's full versatility. Marking a variable in the Variable Browser plots the variable value over the time of the simulation in the currently active plot, unmarking it removes the plot line. An example of three variables of a simulated model plotted in a default plot widget can be seen in Figure 7. The following features are based on default Chaco elements and can easily be used individually on any plot, specifically plots by plugins, either out-of-the-box as described here or derived from them to fit special needs:

- *Panning*: Left clicking and dragging within the plot pans the view.
- *Zooming*: Turning the mouse wheel while hovering over the plot zooms in and out. Hovering over an axis only zooms along the respective axis. Zooming and panning works very fast, and is even reasonably fast with millions of points in the plot window.
- *Selecting*: Left clicking and dragging on the X-axis selects a time period. Double clicking the axis opens a menu for textual input of selection limits.
- *Context menu*: Right clicking on the plot opens a context menu. In the DefaultPlotWidget it shows callbacks for plugins.
- *Marker*: While hovering over one plot, the plot's time stamp under the mouse is displayed as a vertical line in this and all related plots.

Additionally, plots can be saved as images, either as bitmaps in PNG format or as vector graphics in SVG or PDF format.

2.3 Plugin Structure

Currently, infrastructure for two kinds of plugins is available in PySimulator: *Simulator* and *Analysis* plugins. The plugin interfaces are designed to easily integrate own simulator and analysis code.

Simulator plugins are intended to provide the infrastructure to simulate a certain kind of model and write/read the result file of the simulation. In principle all types of simulation engines can be included, provided time series are produced as results and variables and parameters are identified with a hierarchical naming structure. Currently, plugins are available for FMUs [FC10], for Dymola [DS12], and for OpenModelica [GFR+12].

The name of each Simulator plugin appears in the main menu bar (Figure 2) under *Open Model*. To include a Simulator plugin only the plugin code has to be inserted in a new directory of *Plugins/Simulator*, e.g. *FMUSimulator* in Figure 8.

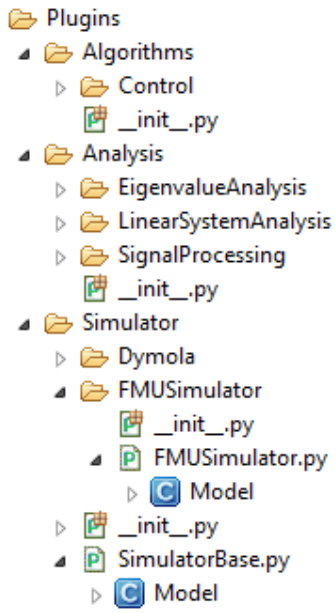

Figure 8: Directory and main Python class structure for plugins in PySimulator.

The main Python code of a Simulator plugin has to be inside a class `Model` that is derived from the class `Plugins.Simulator.SimulatorBase.Model`. Important variables, classes and function of the main class `Model` are:

- `modelType`: String, e.g. '*FMU1.0*', '*Dymola*'.
- `integrationSettings`: Class including start, stop time, algorithm name, etc.
- `integrationStatistics`: Class including number of events, grid points, elapsed real time, etc.
- `integrationResults`: Class including result file access.
- `setVariableTree()`: Function to generate data for a variable tree.

- `getAvailableIntegrationAlgorithms()`: Function to get a list of available integration algorithms.
- `simulate()`: Function to start the numerical integration of the model.
- `initialize(t)`: Function to initialize the model.
- `getDerivatives(t,x)`: Function to evaluate the right hand side of the system.
- `getEventIndicators(t,x)`: Function to evaluate the event indicators (= switching functions to detect events) of the system.
- `getStates()`: Function to get the values of all continuous model states.
- `getStateNames()`: Function to get a list of all names of the continuous model states.
- `getValue(name)`: Function to retrieve the value of a certain variable.

For example, the file *FMUSimulator.py* has a Python class `Model` that provides model typical methods and data as listed for an FMU.

Analysis plugins provide functionality for analyzing the model or result data in the post-processing stage of a simulation. They contain functions which work on variables, models and plots after a model is loaded or a simulation is finished. In order to integrate the Analysis plugins, they are automatically loaded by PySimulator from the *Analysis* folder. An initialization function is called for every plugin to enable the initial setup, like declaration of variables or own classes. The Analysis plugin is further able to register callback functions in the main program which allows access to the plugin's functions. The call of a plugin's function from the GUI takes place by either pull-down menus, a custom button bar or a context menu appearing when the user clicks on an appropriate GUI element like the model's name.

For processing the data, the plugins can implement own algorithms or use shared functionality stored in the *Algorithms* folder. It is furthermore possible for such a plugin to initialize a model or to start a simulation, as this might be necessary for some functionality like linearization of the model. In this case, the features of the Simulator plugins are utilized. The feedback of the Analysis plugin can be sent to the textual Information output window, a plot window or stored in every other way Python allows, e.g. in a file on disk.

It follows a simple example for an Analysis plugin to find the maximum value of a time trajectory and plot a label at the maximum point:

```
def findMax(widget):
  for plot in widget.plots:
    data = plot.data
    maxVal = data[0]
    for time, value in data:
      if value > maxVal[1]:
        maxVal = (time, value)
    maxLabel = DataLabel(
       component=plot,
       data_point=maxVal,
       label_format=str('(%(x)f, %(y)f)'))
    plot.overlays.append(maxLabel)

def getPlotCallbacks():
  return [["Find Maximum", findMax]]
```

3 Simulator Plugins

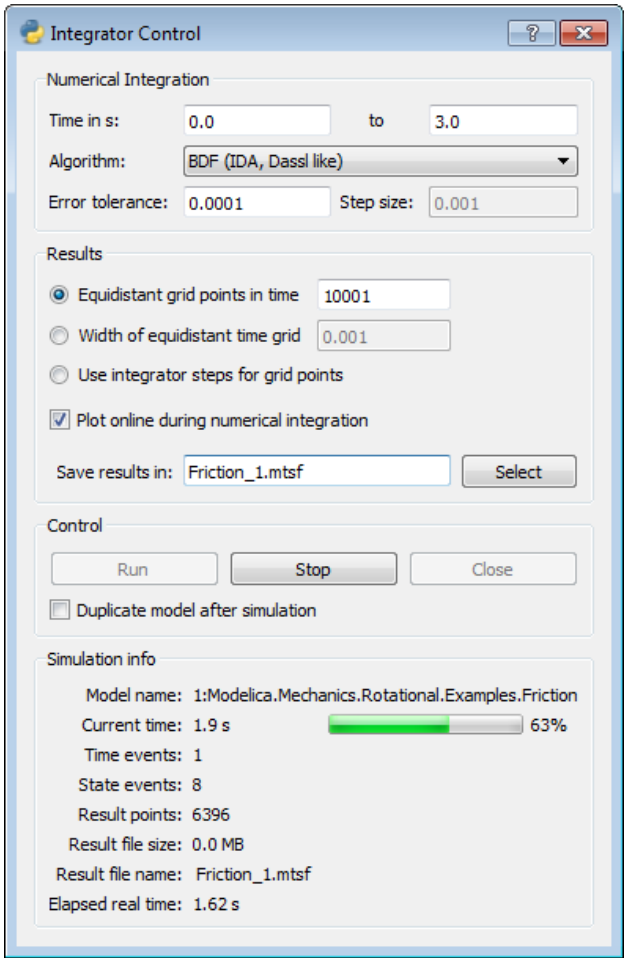

Figure 9: Integrator control GUI in PySimulator.

One of the main features of PySimulator is running and controlling the numerical integration of different types of models (= simulation). Those models require different simulation engines interfaced by the

DOI 10.3384/ecp12076523

Simulator plugins in PySimulator. All the Simulator plugins are controlled by the same Integrator Control GUI, see Figure 9. Some menu entries depend on properties of the Simulator plugin.

Start and stop time for the integration may be edited and one of the integration algorithms available for the Simulator plugin can be selected. Depending on the property of the algorithm the user can edit the error tolerance or the fixed step size. The simulation results are mainly discretized, time depending trajectories. The discretization points (= grid points, dense output points) of the time can be given either by the number of equidistant grid points or by the width of an equidistant time grid. A third option is to use the steps of the integration algorithm for the grid points. The name of the result file can also be specified. If *Plot online* is selected in the GUI, the plots of the simulation results are updated during the integration process. This may increase the elapsed real time for the integration, but gives information about the results at once. This feature is especially intended for model simulations that take some time.

The simulation is run in a separate thread, so Variable Browser and Plot area are still available for user interactions. During the numerical integration several statistical parameters inform about the progress: current simulation time, number of time and state events, number of computed result points, the size of the result file and the elapsed real time so far. In some cases it is very helpful to see that for example lots of events are generated and therefore the integration is getting stuck, or the result settings lead to a huge result file and therefore the simulation is slowing down.

3.1 FMU Simulator

The FMU simulator provides an interface to models exported as a Functional Mockup Unit for Model Exchange (FMU, see [MC10]). This interface is supported by more than 30 simulation environments (www.functional-mockup-interface.org/tools.html).
An FMU is basically composed of two components: Firstly, a description file in XML-format holds all information about the variables of the model and other model information. Secondly, binaries for one or several target machines are contained, such as Windows dynamic link libraries (.dll) or Linux shared object libraries (.so). They contain the code for evaluating the model's equations.

This way, the FMU interface allows the evaluation of the right hand side f of the governing equations of a model, as well as its outputs y and its event indicator signals z. They depend in generally on the model's states x, its parameters p, inputs u and the time t. Additionally, time events can be triggered by the FMU. The event indicator signals are used to detect state events, which may occur in many physical models. With this information, it is possible for a numerical integration solver to perform the time integration of the model to obtain a solution, see Figure 10.

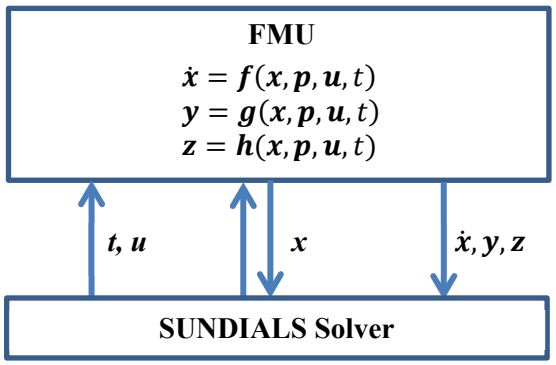

Figure 10: Interface from the FMU model to the SUNDIALS solver.

The single steps performed by PySimulator are the following. First, the XML description file of the selected model is parsed. The information from this file is visualized in the Variable Browser of the main GUI. The Variable Browser can thus also be used independently as an FMU description viewer.

Next, the Functional Mockup Interface (FMI) functions in the shared library are interfaced to make them available in PySimulator. This way, it is possible for the integrator to call the model functions. While these parts are sufficient for some basic operations like initialization, the time integration itself utilizes the Sundials Solver Suite [HBG05]. Sundials provides solvers for explicit and implicit dynamical systems: CVODE and IDA. CVODE numerically integrates ordinary differential equations by linear multistep methods. Depending on the solution CVODE switches between solvers for stiff and non-stiff problems. IDA uses BDF (Backwards Differentiation Formulas) to solve systems of differential-algebraic equations. Sundials supports root finding during the numerical integration. In summary, the Sundials solvers are prepared to be applied to FMUs. The Sundials integrator suite is implemented in C and is accessed from PySimulator via the *python-sundials* [T12] interface.

The FMU Simulator can be interfaced both by code from e.g. an Analysis plugin as well as by the GUI elements described in Section 3. In both cases the important simulation parameters can be adjusted by the user to the specific problem. After simulation, the results are stored in the MTSF file format, the proposed standard time series file format [PBO12] that is based on HDF5 [THG12]. This format offers a way to read and write variable information and numeric data in a convenient and standardized way. The format is especially designed to support both small and very large files. In [PBO12] MTSF files up to 200 GBytes have been generated and variables have been read from the file. Most simulation programs do not support generating and plotting result files of such a size.

For example, a result file for the full robot model from the Modelica Standard Library (FMU generated by Dymola) is generated with 30 Mio. result points. The result file has a size of 171 GBytes. When plotting signals from this file, the loaded signal is downsampled to 5 Mio. points to get acceptable plotting performance.

3.2 Dymola Simulator

The second Simulator plugin is based on the simulation executable (*dymosim[.exe]*) generated by the commercial Modelica environment Dymola [DS12] from Dassault Systèmes. PySimulator supports selecting a Modelica model by asking for the package file and the model name. Then, the Modelica model is automatically compiled by Dymola in the background if there is a version of Dymola installed. The executable includes object code for both the model equations and the numerical integration algorithms.

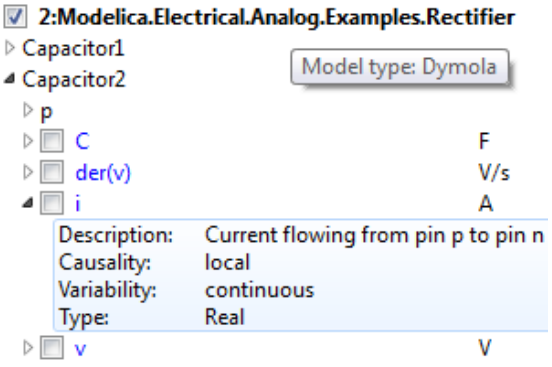

Figure 11: Variable tree in PySimulator based on Dymola's simulation executable.

The list of all variables and the values for editable parameters and initial values are generated when loading the model, see Figure 11 for an example. If the user wants to start the numerical integration the function `model.simulate` of the Dymola Simulator plugin generates a new initialization file from the integration settings in the Integrator Control GUI and the changed parameters and initial values. After these preparations the simulation executable is started. During the numerical integration process the current simulation time is read and displayed in the Integrator Control GUI to be up to date about the simulation progress.

The result file in Matlab's 4 binary MAT-format can be read by PySimulator. The corresponding result object in PySimulator enables to get access to the numeric data, the description string and the unit by a Modelica variable name. A conversion of Dymola's result file (MAT) to the proposed Standard Time Series File Format (MTSF) is supported by a separate menu entry shown in Figure 2.

3.3 OpenModelica Simulator

A third Simulator plugin for PySimulator is shipped with the open source OpenModelica environment. Details about this plugin are given in [GFR+12].

4 Analysis Plugins

The result of a simulation mainly consists of time series data that can be plotted. Signal processing plugins can access the plot data, can extract more information and can visualize it. Several simple functionalities are already provided to compute minimum, maximum, and other signal properties in a selectable time window. Furthermore, an involved functionality is available to perform Fast Fourier Transformations.

The nonlinear model of a Simulator plugin can be linearized around the initialization point or another time point of the simulation (provided the Simulator plugin supports the required interface for linear models). Afterwards, linear system analysis plugins can operate on such a linear system. Already available plugins compute and plot eigenvalues, provide eigenmode analysis, and perform frequency and step responses.

4.1 Signal Processing Plugin

The Signal Processing plugin provides operations on result signals displayed in a plot window. When right clicking on a plot window, together with an optional

DOI 10.3384/ecp12076523

selection of a time range, a window (see Figure 7) pops up to select the desired signal processing operation on the selected time range.

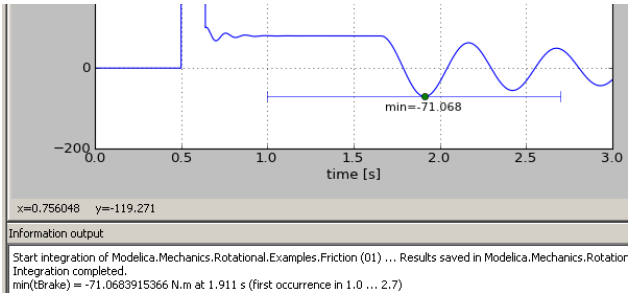

Figure 12: Example for marking of a minimum.

An example of how the result of an operation is shown in a plot is given in Figure 12, where the minimum of a signal is determined in the range $t \in [1.0, 2.7]$.

The operations to be carried out have the following mathematical definition:

Name	**Operation on $y(t)$** with $t_{min} \le t \le t_{max}$, $T = t_{max} - t_{min}$
Minimum	$y_{min} = \min y(t)$
Maximum	$y_{max} = \max y(t)$
Arithmetic Mean	$y_{DC} = \frac{1}{T} \cdot \int_{t_{min}}^{t_{max}} y(t) \cdot dt$
Rectified Mean	$y_{RM} = \frac{1}{T} \cdot \int_{t_{min}}^{t_{max}} \lvert y(t) \rvert \cdot dt$
Root Mean Square	$y_{RMS} = \sqrt{\frac{1}{T} \cdot \int_{t_{min}}^{t_{max}} y(t)^2 \cdot dt}$
FFT	$f_s = \frac{n-1}{T}$, $f = \left[0, \frac{f_s}{n}, \frac{2f_s}{n}, \cdots, \frac{f_s}{2}\right]$, $\Delta y_r = y(t_r) - y_{DC}$, $y_{FFT,k}(f_k) = \frac{1}{n_f} \sum_{r=0}^{n_f-1} \Delta y_r \, e^{-i2\pi k \frac{r}{n_f}}$

The integrals in the operations are computed by using the trapezoidal integration rule on the selected signal y (basically, the result points of y are linearly interpolated and then exactly integrated).

The Fast Fourier Transform (FFT, [RKH10]) is used to analyze which frequencies with which amplitudes are contained in a periodic result signal. For this, a complex vector y_{FFT} is computed as function of a real frequency vector f. Since an FFT requires equidistant time points, the (potentially) non-equidistant result points of a signal, $y = y(t)$, are linearly interpolated and mapped to an equidistant grid of the desired number of points n. The frequency vector f consists of $n_f = \text{div}(n,2) + 1$ points. For even n, the last point of vector f is $f_s/2$, otherwise it is $f_s/2 \cdot (n\text{-}1)/n$ (with $f_s = (n\text{-}1)/T$ and T as the selected time range). The variant of FFT is used, that subtracts the arithmetic mean of y from the signal y itself and normalizes the FFT result with n_f (in order that amplitudes of y_{FFT} correspond to the amplitudes in the underlying result signal).

The core FFT calculation is performed with Python function `numpy.fft.rfft` which in turn is an interface to the Fortran package `fftpack` [Swa82]. This package computes the FFT of an equidistant vector y of any length n in $O(n^2)$ and if n is expressed as a multiple of 2, 3, 4, or 5, that is $n = 2^i 3^j 4^k 5^l$ in $O(n \cdot \log(n))$ operations. Note, the non-prime factor 4 gives a speed-up with respect to purely 2 factors [Tem83].

A natural question is what number n to select. There are two requirements: (1) all frequencies up to a desired frequency should be included, and (2) the distance between two frequency points should be small enough. With (1) the number of points n can be computed as (T is the time range on which the FFT is applied):

$$f_{max} = \frac{f_s}{2} = \frac{n-1}{2T} \quad \rightarrow \quad n \approx 2T f_{max}.$$

The distance d between two frequency points of vector f for an even number of n is computed as (for an odd n the result is the same, but with a slightly different derivation):

$$d = \frac{f_{max}}{n_f - 1} = \frac{f_s/2}{n_f - 1} = \frac{\frac{n-1}{2T}}{\frac{n}{2} + 1 - 1} = \frac{1}{T}\frac{n-1}{n} \approx \frac{1}{T}.$$

This means that the frequency resolution depends only on the examined time interval T and can therefore only be enlarged by enlarging this interval (and it is not related to the number of points used in the FFT calculation). For example, if the base frequency is f_0 and the examined time interval T is over k periods of this base frequency, then the distance d is:

$$d = \frac{1}{k/f_0} = \frac{f_0}{k}.$$

In other words, in order to get at least a resolution of 10 % of the base frequency, the examined time interval should have at least a range of 10 base periods.

As a simple example consider the following addition of two sines with different amplitudes ($A_1 = 1, A_2 = 0.2$) and frequencies ($f_1 = 5, f_2 = 20$):

$$y(t) = A_1 \sin(2\pi f_1 t) + A_2 \sin(2\pi f_2 t).$$

If 10 periods of f_1 are analyzed, the FFT-plot up to $2f_2$ ($n \approx 2 \cdot \frac{10}{5} \cdot 40 + 1 \rightarrow n = 160$) results in Figure 13.

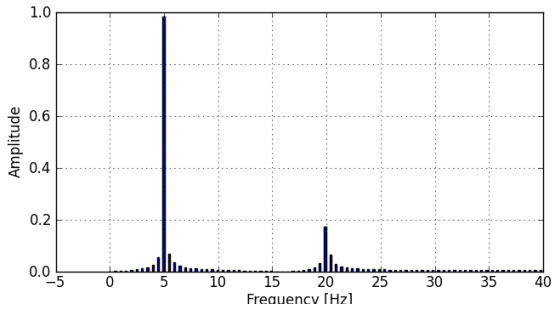

Figure 13: FFT of example with *n* = 160.

As can be seen the 5 and 20 Hz frequencies are correctly identified with small errors in the amplitudes. (the width of the plot bars are selected as $2/5 \cdot d$). Extending the frequency range to $10f_2$ does not change the resolution ($d = 5/10 = 0.5\,Hz$), but reduces the amplitude errors as seen in Figure 14.

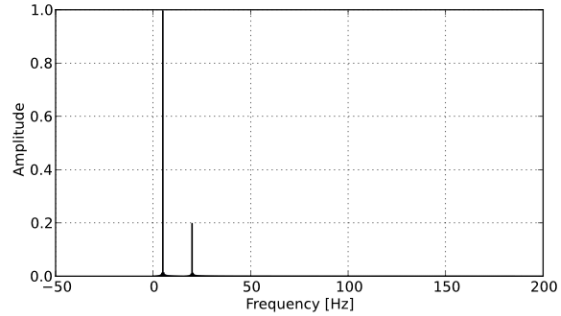

Figure 14: FFT of example with *n* = 800.

4.2 Linear System Analysis Plugin

For many control applications it is necessary to have a linear approximation of a nonlinear system. In addition a linear representation of a nonlinear system can be helpful to analyze specific properties of the system, for example local stability.

The Linear System Analysis plugin allows to automatically linearize a model that is loaded into PySimulator. If the plugin is loaded, its functionality can be accessed by right-clicking a loaded model in the GUI of PySimulator. If a loaded model is linearized using the GUI the parameter set $p \in \mathbb{R}^{n_p}$, as defined in the Variable Browser is used for the linearization around the operating point. If it is called from a Python-script, a set (Python dictionary) of parameters and values can be used. A model (nonlinear dynamic system) can be represented as a set of equations:

$$\dot{x} = f(x, p, u, t), \qquad x(t_0) = x_0,$$
$$y = g(x, p, u, t).$$

For the plugin it is necessary that a set of inputs $u \in \mathbb{R}^{n_u}$ and outputs $y \in \mathbb{R}^{n_y}$ are defined in the model, where $n_u \in \mathbb{N}$ is the number of inputs and $n_y \in \mathbb{N}$ is the number of outputs of the system.

The linearization procedure uses a numerical central difference quotient for the calculation of the Jacobians. For a function $q(v)$ depending on a scalar v we use the approximation:

$$q_v(v) \approx \frac{q(v+\delta) - q(v-\delta)}{2\delta}$$

with a step size $\delta = \sqrt[3]{\varepsilon}\max(|v|, 1)$ and the machine precision ε. The step size is computed to find a compromise between a minimum discretization error and a minimum numerical error.

The central difference quotient is successively applied to every component of x and u at a steady state point $w_{ss} := (x_{ss}, p, u_{ss}, t_0)$. The linear approximation of the nonlinear system is a linear time invariant (LTI) system that is represented by the matrices $A \in \mathbb{R}^{n_x \times n_x}$, $B \in \mathbb{R}^{n_x \times n_u}$, $C \in \mathbb{R}^{n_y \times n_x}$ and $D \in \mathbb{R}^{n_y \times n_u}$:

$$A = f_x(w_{ss}), \qquad B = f_u(w_{ss}),$$
$$C = g_x(w_{ss}), \qquad D = g_u(w_{ss}).$$

The default case is $x_{ss} := x_0 \in \mathbb{R}^{n_x}$ and $u_{ss} = 0$. If no user defined steady state point is given, x_{ss} is calculated by calling the simulator's initialization function. It is also possible to linearize around an arbitrary user-defined steady state x_{ss}.

DOI 10.3384/ecp12076523

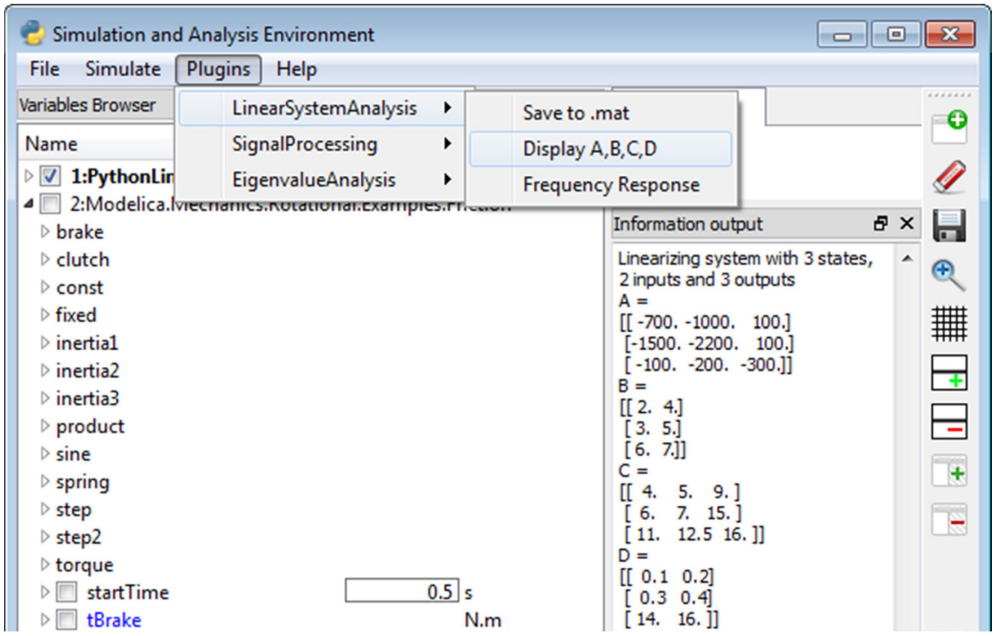

Figure 15: Linear System Analysis plugin inside PySimulator.

The linear system is generated as an instance of a Python class inside the Linear System Analysis plugin, and can be accessed by other plugins inside PySimulator for further analysis. The class provides functions to return the matrices *A*, *B*, *C*, *D*, names and sizes of the input, output and state vectors. In addition it allows writing the matrices along with the state, input and output names to a file in Matlab's MAT-format, see Figure 15, so that they can be directly used for controller synthesis inside Matlab [M12].

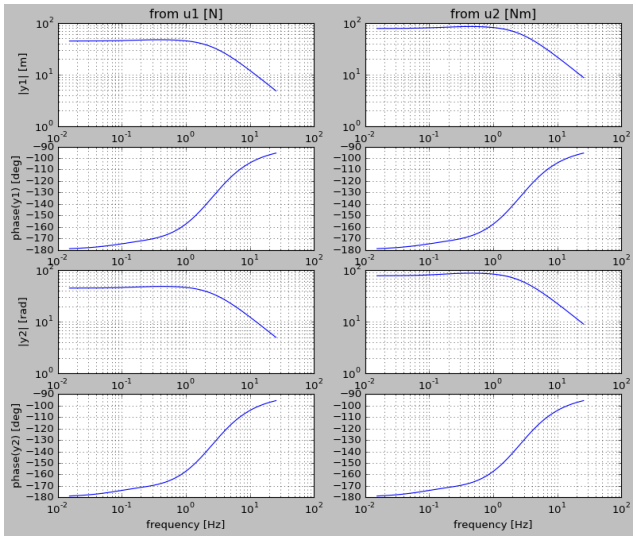

Figure 16: Frequency responses of a 2x2 system.

Furthermore, the plugin provides various analysis operations on the linear input/output system. Most important, the frequency responses from the inputs to the outputs are computed and plotted. An example of the frequency responses of a system with 2 inputs and 2 outputs is shown in Figure 16.

4.3 Eigenvalue Analysis Plugin

For the analysis of many systems, the eigenvalues and eigenmodes are of special interest. They support the understanding of the system by providing damping and frequency information when eigenmodes or states are excited.

The Eigenvalue Analysis plugin needs the functionality to linearize a system as a starting point for further analysis. For this, the Linear System Analysis plugin from Section 4.2 is utilized. Based on this, functions for the visualization of both eigenvalues and eigenmodes can be called, see Figure 17.

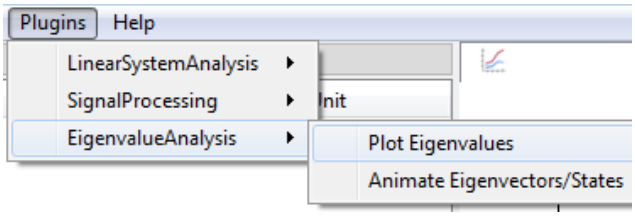

Figure 17: Menu of the Eigenvalue Analysis plugin.

The eigenvalues are plotted in the complex domain as can be seen in Figure 18. This provides information about the stability in the point of linearization as well as about the dynamics of the corresponding eigenmodes. When clicking with the left mouse button on an eigenvalue, additional information to this

eigenvalue is displayed such as frequency, damping and controllability.

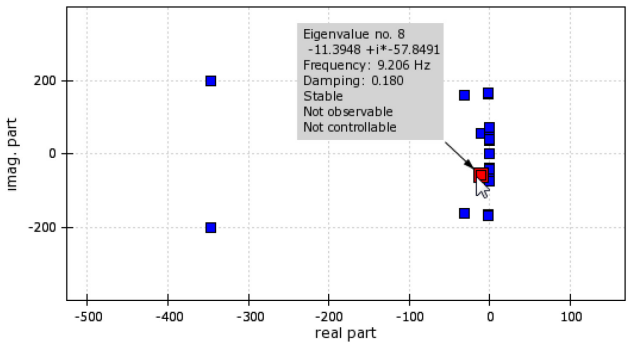

Figure 18: Plot of eigenvalues and frequency response with additional information.

The eigenmodes themselves can be visualized if the model has been exported with an own visualization routine. This is e.g. the case, if a Modelica model is exported with the DLR Visualization library [Bel09]. The eigenmodes are a linear combination of the model's states. Therefore, they can be visualized if the states have some form of visualization. The selected eigenmodes, see Figure 19, are excited by a periodic sine, making it possible to see their impact on the system, not only in a figure, but in a dynamic way.

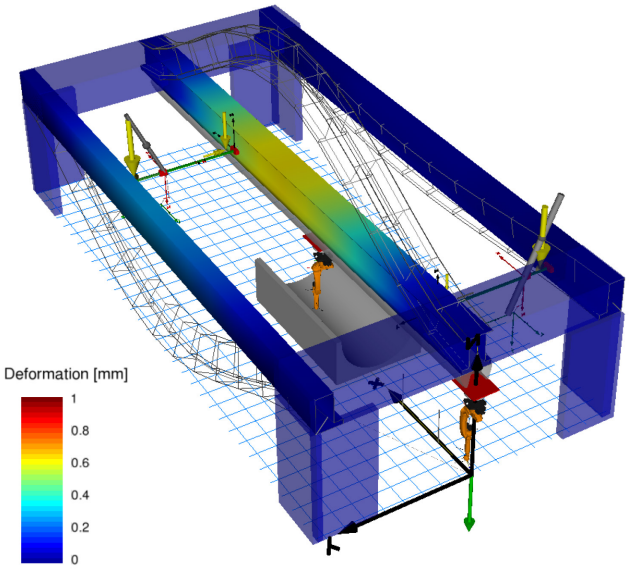

Figure 19: GUI and animation of the 8th eigenmode, showing a clear coupling of the flexible states.

The shown example is a mechanical model of a multi-robot cell of the DLR Center of Lightweight Production Technology. The visualized Eigenmode 8 shows a clear coupling of the left and middle beam due to the portal shown in the upper left part of the figure. The GUI in Figure 20 shows some dynamic properties which can also be seen in the eigenvalue plot in Figure 18. As an additional possibility, the user can furthermore visualize the states of the system.

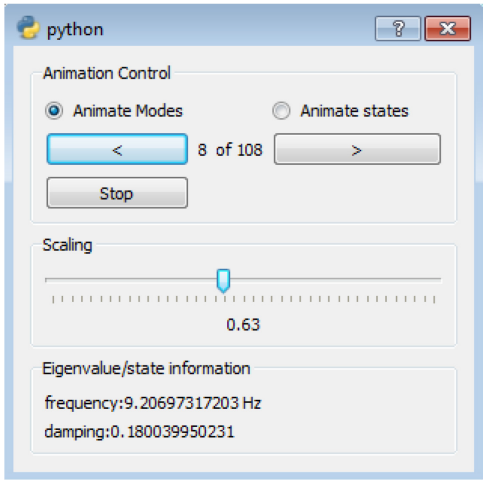

Figure 20: GUI to control the visualization of eigenmodes and states.

The combination of the two abilities *Plot Eigenvalues* and *Animate Eigenvectors/States* enables the engineer to understand and visualize the dynamics of the system. This might help to adapt parameters of the system to e.g. stabilize it or reduce the impact of a periodic disturbance.

5 Algorithms

The algorithms used in the plugins are mostly based on the standard Python packages `numpy` and `scipy`. However, several new algorithms had to be implemented that seemed to be not yet available in other Python packages. These algorithms are provided under directory *Plugins/Algorithms*. All functions in this directory can be used also in any other context, since there is no relationship to PySimulator (just that plugins from PySimulator are calling these functions). Especially, in this directory functions are provided for the Signal Processing and the Linear System Analysis plugins.

For example, class LTI in file *Algorithms/Control/lti.py* provides various functions for multi-input-multi-output Linear Time Invariant systems. In the current version, two representations of continuous linear systems are supported:

DOI 10.3384/ecp12076523

- **LTI – State Space** (derived by linearization from the nonlinear model, see Section 4.2):

$$\dot{x}(t) = Ax(t) + Bu(t),$$
$$y(t) = Cx(t) + Du(t).$$

- **LTI – Zeros and Poles**:

$$y(s) = \begin{bmatrix} g_{11} & \cdots & g_{1m} \\ \vdots & \ddots & \vdots \\ g_{n1} & \cdots & g_{nm} \end{bmatrix} \cdot u(s),$$

$$g_{ij}(s) = k_{ij} \cdot \frac{\prod_l (s - z_{ij,l})}{\prod_l (s - p_{ij,l})} = k_{ij} \cdot \frac{\prod_l (s + n_{1,ij,l})}{\prod_l (s + d_{1,ij,l})} \cdot \frac{\prod_l (s^2 + n_{2,ij,l} s + n_{3,ij,l})}{\prod_l (s^2 + d_{2,ij,l} s + d_{3,ij,l})}$$

An LTI object is initialized by either defining a state space representation with a tuple of matrices (*A*, *B*, *C*, *D*), or by defining a zeros and poles representation by a matrix of tuples (*k, z, p*). Such a tuple is defined with a gain $k \in \mathbb{R}$, and z and p vectors of real or conjugate complex zeros and poles. Internally in the class, a second representation is computed and stored consisting of first and second order transfer functions described by coefficients $n_{q,ij,l}$, $d_{q,ij,l} \in \mathbb{R}$ with $q = 1, 2, 3$. Depending on the selected operation, one of the two representation forms is used to perform the calculation. For example, evaluating a zeros and poles object on a given *s*-value is performed with the second representation form, since then a real-valued *s* will result in a real-valued result. Otherwise, due to numerical errors, the result might be complex-valued.

Besides pure data, also meta-information can be associated to an LTI object, consisting of signal names, units and description texts. When generating an LTI object from the Linear System Analysis plugin, this meta information is automatically generated from the corresponding information stored in the underlying model. When plotting or printing an LTI object, the meta-information is utilized to improve the representation for the user.

Currently, only a few operations on LTI objects are provided. Most importantly, a frequency response object can be computed. If the LTI object is in a state space representation, it is internally first transformed to a zeros and poles object and this object is then evaluated on the desired $s = j\omega$ values. By default, these values are selected on a logarithmic scale and the smallest and largest frequency values are deduced from the poles and zeros. The transformation to zeros and poles form is performed in a numerically reliable way by computing the eigenvalues of *A* and the generalized eigenvalues of (*A*, *B*, *C*, *D*) for selected columns of *B* and selected rows of *C* and *D*.

6 Conclusions

PySimulator is provided as an open source environment to conveniently perform simulations with different simulation engines and to analyze the results with a wide range of Analysis plugins. The environment has been designed to cope with large problems. For example, result files with sizes larger than 100 GByte can be handled, as well as several million points in one plot window. We hope that many other people will contribute with Simulator and Analysis plugins. We plan to include plugins from other developers in future PySimulator distributions, provided the plugin adds useful functionality, and the most restrictive license used in the plugin is LGPL. The copyright remains with the developers.

7 Acknowledgement

Important inputs for the design of the Simulator plugin interfaces have been given by Anand Kalaiarasi Ganeson and Peter Fritzson (PELAB) during the fruitful cooperation to integrate the OpenModelica Simulator plugin into PySimulator.

References

[AAF+12] Andersson C., Andreasson, J., Führer C. and Åkesson J.: *A Workbench for Multibody Systems ODE and DAE Solvers*. In Proc. of 2nd Joint International Conference on Multibody System Dynamics, Stuttgart, Germany, 2012.

[Bel09] Bellmann T.: *Interactive Simulations and advanced Visualization with Modelica*. Proceedings of 7th International Modelica Conference, Como, Italy, Sep. 20-22, 2009.

[DS12] Dassault Systèmes AB: *Dymola*, www.dymola.com.

[E12] Enthough, Inc.: *Chaco*. code.enthought.com/chaco.

[GFR+12] Ganeson A. K., Fritzson P., Rogovchenko O., Asghar A., Sjölund M. and Pfeiffer A.: *An OpenModelica Python Interface and its*

use in PySimulator. Accepted for publication in the Proceedings of 9th International Modelica Conference, Munich, Germany, Sept. 2012.

[HBG05] Hindmarsh A. C., Brown P. N., Grant K. E., Lee S. L., Serban R., Shumaker D. E. and Woodward C. S.: *SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers*. ACM Transactions on Mathematical Software, 31(3), pp. 363-396, 2005.

[LBN+12] Lawrence Berkeley National Laboratory: *BuildingsPy*. simulationresearch.lbl.gov/modelica.

[M12] MathWorks: *Matlab*. www.mathworks.com/products/matlab.

[MC10] MODELISAR consortium: *Functional Mock-up Interface for Model Exchange, Version 1.0*, 2010. www.functional-mockup-interface.org.

[NC12] Nokia Corporation: *Qt.* www.qt.nokia.com.

[P12] *PySide*. www.pyside.org.

[PBO12] Pfeiffer A., Bausch-Gall I. and Otter M.: *Proposal for a Standard Time Series File Format in HDF5*. Accepted for publication in the Proceedings of 9th International Modelica Conference, Munich, Germany, Sept. 2012.

[RKH10] Rao K. R., Kim D. N. and Hwang J.-J.: *Fast Fourier Transform: Algorithms And Applications*. Springer, Dordrecht, Heidelberg, London, 2010.

[Swa82] Swarztrauber P.N.: *Vectorizing the FFTs.* In: Parallel Computations, Ed. G. Rodrigue, Academic Press, 1982, pp. 51-83. www.netlib.org/fftpack

[T12] Tenfjord R.: *Python-sundials*. www.code.google.com/p/python-sundials.

[Tem83] Temperton C.: *Self-Sorting Mixed-Radix Fast Fourier Transforms*. Journal of Computational Physics, 52, pp. 1-23, 1983. www.sciencedirect.com/science/article/pii/0021999183900013X.

[THG12] The HDF Group. www.hdfgroup.org.

 DOI 10.3384/ecp12076523