PLECS

*Tutorial*

# PLECS XML-RPC Interface and Controller Design in Python

## Using the Python Controls Library for Current Control Design of a 3-Phase Grid-Connected VSI

Tutorial version 1.1

# 1 Introduction

The XML-RPC interface of PLECS Standalone allows you to send and receive data to and from PLECS Standalone using an external program. Many programming languages such as Python, Java, C++, or Ruby offer standard XML-RPC Libraries to set up an XML-RPC client. In this tutorial Python 3 is used to launch PLECS simulations and retrieve the simulation results to be post-processed. In this tutorial you will learn:

- how the XML-RPC interface is embedded into PLECS Standalone and the simulation flow.
- how to design a simple current control scheme using the Python Control System Library.
- how to use the XML-RPC interface to interact with PLECS (starting a simulation and reading back data).

Before you start this tutorial please make sure that you have the following files in your working directory that are included in the distribution:

- `tutorial_functions_library.py`
- `xmlrpc_controller_design.plecs`
- `xmlrpc_controller_design_1.py`
- `xmlrpc_controller_design_2.py`
- `xmlrpc_controller_design.py`

## 1.1 Python Environment

The XML-RPC client library (`xmlrpc.client`) is shipped with Python 3 by default so that no additional software needs to be installed to establish a connection from Python to PLECS over XML-RPC. For this specific tutorial, additional libraries are required due to the additional focus on controller design. These libraries are:

- **os:** A module that provides a portable way of using operating system-dependent functionality. It is included by default in the Python 3 distribution.
- **matplotlib:** A plotting library for Python based on numpy.
- **control:** A package that implements basic operations for analysis and design of feedback control systems.
- **scipy.io:** A Scipy module to read data from and read data to a variety of file formats.
- **io:** A module that provides main facilities for dealing with various types of I/O.

The matplotlib and control libraries (which are not native Python 3 libraries) can be installed using the Python 3 package manager `pip3`

```
pip3 install matplotlib control
```

Please note that in this tutorial the `tutorial_functions_library.py` imports the mentioned libraries:

```
import matplotlib.pyplot as plt
import control as ct
import os
import scipy.io as sio
import io
```

## 1.2 PLECS XML-RPC Interface

PLECS has an built-in XML-RPC server that listens for simulation commands from clients on a specific XML-RPC port. The complete set of available XML-RPC commands is listed in the PLECS user manual.

The main command to start a PLECS simulation with a specific set of parameters is:

```
plecs.simulate('mdlName',optStruct)
```

The optional argument *optStruct* allows to override the parameters defined in the PLECS model. It contains the field *ModelVars* which again contains the individual simulation parameters. Values can be numerical scalars, vectors, matrices or 3d arrays, or strings. Additionally, an optional field *SolverOpts* can be passed to a simulation if the solver parameters should also be adapted (not implemented in this tutorial). Overriding model parameters using the *optStruct* allows to run several simulations in a row with different parameters without having to modify the model file.

Overriding parameter values means that the model initialization commands are executed first, then the parameters specified inside the *optStruct* are applied, and at the end the component parameters are evaluated.

### Example

The following Fig. 1 gives a simple example of the simulation flow:

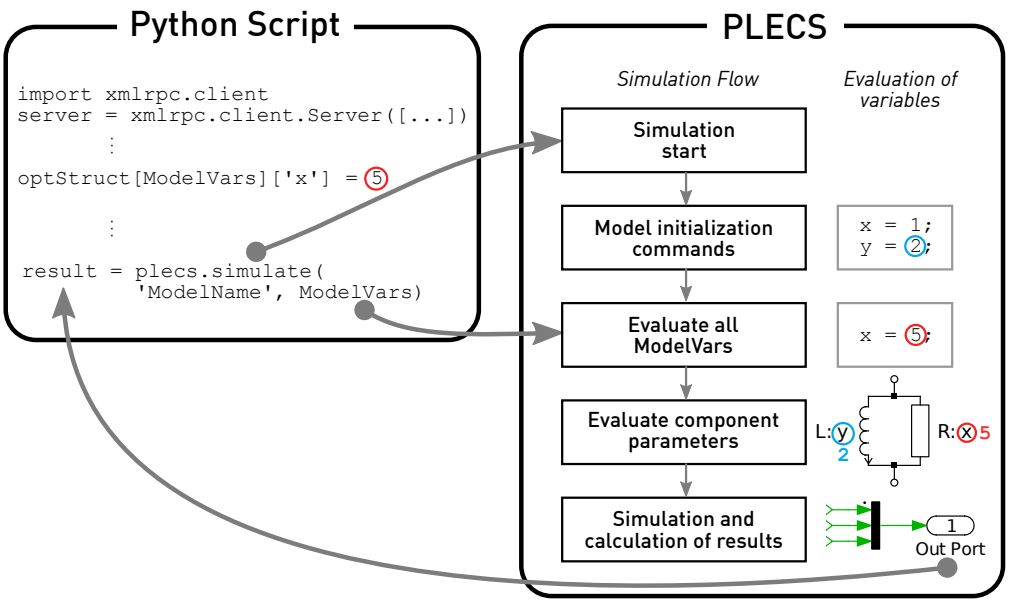


**Figure 1: Overview of the PLECS Standalone XML-RPC interface with a simple example of setting parameter values and reading back simulation results**

**1** In a Python script the struct `ModelVars` is defined with a parameter `x` and a value of 5.

**2** Using the XML-RPC command `simulate` starts a PLECS simulation of the specified model and transmits the specified parameters in the `ModelVars` struct to PLECS.

**3** The PLECS simulation starts by evaluating the initialization commands of the model where variables `x` and `y` are defined.

**4** Then the `ModelVars` struct is evaluated and the value of 1 assigned to variable `x` is overwritten by the value 5, as is defined in the external Python script. The value of `y` is not modified.

**5** Next, all circuit parameters are evaluated. The inductance takes the value 2 H and the resistance the value 5 Ω.

**6** Finally, a simulation is executed. After the simulation has finished, all the data that was passed to the Out Port is transmitted back to the Python script over XML-RPC. This simulation data is then available in the return value `result`.

This example shows how parameters in a PLECS simulation are defined over XML-RPC. This tutorial uses the same concepts, but different variable names and values.

### Setting up the XML-RPC interface

**Your Task:** Load modules and setup the XML-RPC connection.

**1** First, we need to create a new Python script in our editor or IDE of choice. Then, we include the following code to import modules and functions we will use in the tutorial:

```
1 from tutorial_functions_library import *
```

**2** We need to launch PLECS before executing the Python script. To enable the built-in XML-RPC interface of PLECS, go to the **General** tab of the **PLECS Preferences** window. Click the **Enable** checkbox of the **XML-RPC interface** option to enable the XML-RPC interface on the standard port 1080. If the port 1080 is already taken by another application the port number needs to be changed to another value. In order to start the XML-RPC Server PLECS needs to be restarted.

Keep in mind that XML-RPC connections to PLECS are only allowed from clients running on the same machine as PLECS, i.e. the Python script and the PLECS simulation have to run on the same machine. Therefore, the XML-RPC connection should always be initiated using `localhost` in the server URL.

**3** Import the XML-RPC module and start the client server connection with PLECS.

```
2 import xmlrpc.client
3 server = xmlrpc.client.Server("http://localhost:1080/RPC2")
```

# 2 Current Control Design with Python Control System Library

The technical content of this tutorial will follow the analysis shown in [1]. The analysis shows the modeling of a grid-connected 3-phase voltage source inverter through an RL filter. The system is depicted in Fig. 2. Once the modeling is performed a current controller in the dq synchronous reference frame is designed. In this tutorial, we will use the design guidelines provide in [1] and the Python Control System Library to design the current controller and obtain the expected dynamic responses of the system for a given set of parameters.

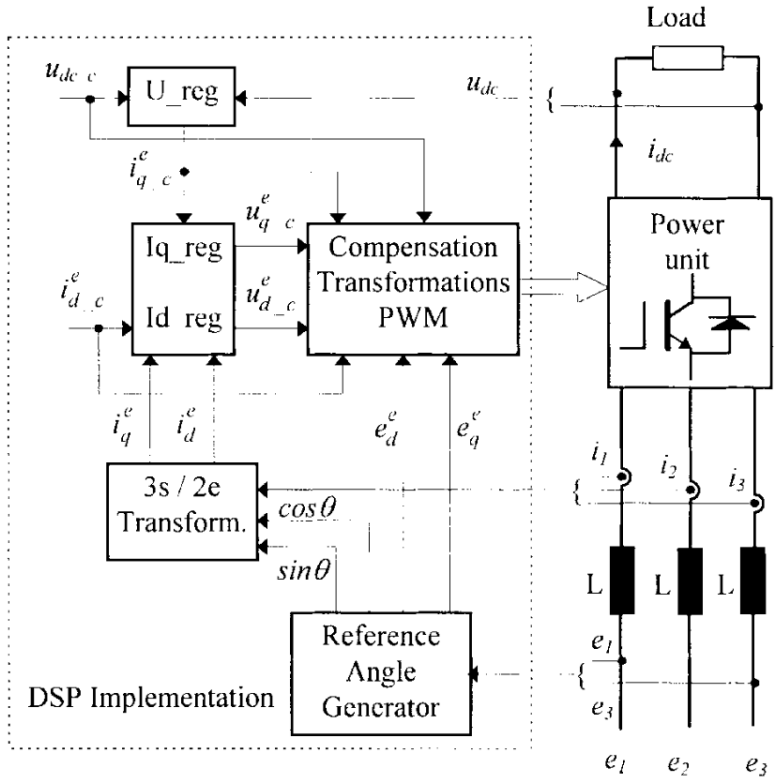


**Figure 2: Modeled system of a grid-connected 3-phase voltage source inverter through an RL filter (Fig. 6 in [1])**

For this tutorial the PLECS model with the implementation of the system shown in Fig. 2 is provided in the file `xmlrpc_controller_design.plecs`. In [1], an analysis for the control design of the DC voltage regulator is also carried out, however, this is out of the scope of the tutorial. In order to simplify the PLECS model, the DC link capacitance and DC load have been substituted by a DC voltage source.

**Your Task:** Your task is to use the Python Control System Library to calculate the transfer functions (TFs) that model the dynamics of the system, and obtain the expected frequency and transient responses. For the system modeling and controller design we will follow the guidelines shown in [1].

**1** The parameters used in the system need to be defined. For this you can use our user-defined function `SystemParameters()` where we have already defined all the parameters needed:

```
4 Param = SystemParameters()
```

You can change the values by modifying the `SystemParameters()` function in `tutorial_functions_library.py` file. This function returns a dictionary that contains the following parameters values:

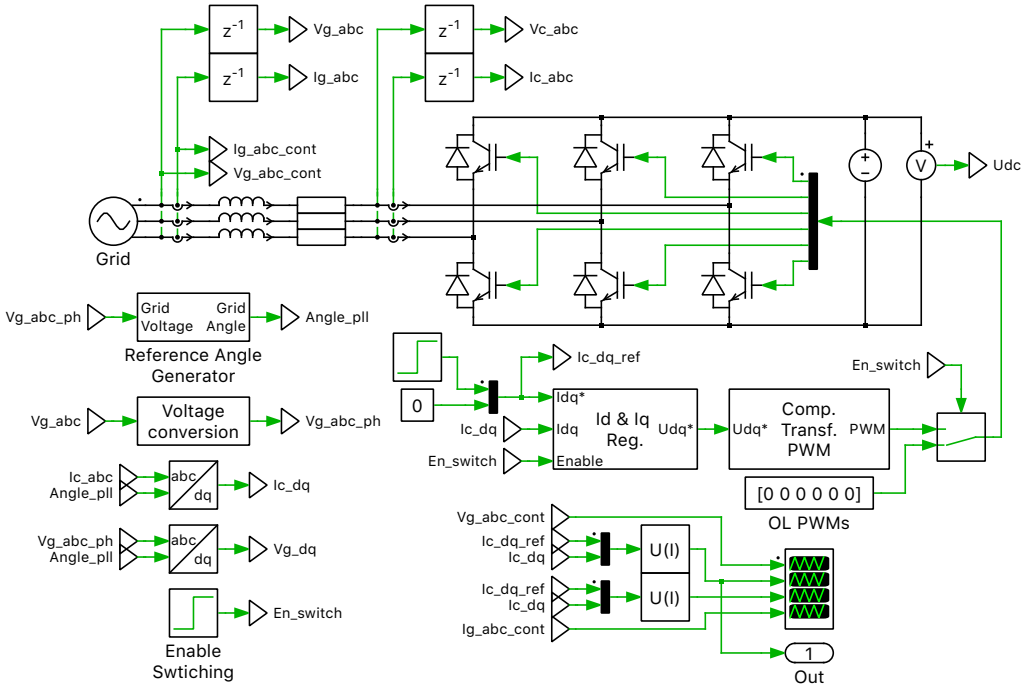


**Figure 3: Main schematic of the PLECS model of a grid-connected 3 phase voltage source inverter through a RL filter**

- $T_\mathrm{s} = 5\mathrm{e}{-5}$ is the switching period.
- $T_\mathrm{ei} = 1.5 \cdot T_\mathrm{s}$ is the approximated time constant of the grouped TFs of the Sample and Hold and PWM modulator.
- $T_\mathrm{rl} = L/R$ is the plant time constant. The plant in this system is the RL grid filter.
- $K_\mathrm{rl} = 1/R$ is the plant gain.
- $K_\mathrm{PWM} = 1$ is the PWM modulator gain.
- $D = \sqrt{2}/2$ is the damping factor of the closed-loop transfer function.
- $K_\mathrm{i} = T_\mathrm{rl}/(4D^2 \cdot T_\mathrm{ei} \cdot K_\mathrm{PWM} \cdot K_\mathrm{rl})$ is the PI proportional gain.
- $T_\mathrm{i}$ is the PI integral constant.

As [1] explains, two different values of $T_\mathrm{i}$ can be chosen depending on the design objective. $T_\mathrm{i} = T_\mathrm{rl}$ is used for optimum current reference tracking, and $T_\mathrm{i} = 10 \cdot T_\mathrm{ei}$ provides an improved disturbance rejection.

**2** In order to obtain dynamic performance of the designed closed-loop current control, you need to calculate the transfer functions of the system. Create a transfer function variable ($s$) to ease the definition of the transfer functions. You can use the following code:

```
5 s = ct.tf('s')
```

**3** You can now define the transfer functions of the Sample and Hold and PWM delays, and RL filter. As shown in [1], the delays are the smallest time constant of the system. For simplification, they can be grouped together and we can use a first order approximation with an equivalent time constant $T_\mathrm{ei}$. The RL filter transfer function can be calculated as $Plant\_TF(s) = \frac{i_\mathrm{L}(s)}{v_\mathrm{L}(s)} = \frac{1}{L*s+R} = \frac{K_\mathrm{rl}}{T_\mathrm{rl}*s+1}$. For these calculations you can use the following code:

```
6 Delays_TF = 1/(1+s*Param['Tei'])
7 Plant_TF = Param['Krl']/(1+s*Param['Trl'])
```

**4** We will analyze the dynamics of the system with both variants of the PI regulators, the one designed for optimal current reference tracking, and the one for improved disturbance rejection. This means that we will need to calculate two different sets of the three TFs: closed-loop TF $H_{\mathrm{ci}}(s) = i_{\mathrm{c}}(s)/i_{\mathrm{c}}^{*}(s)$, open-loop TF $H_{\mathrm{oi}}(s) = i_{\mathrm{c}}(s)/i_{\mathrm{c}}^{*}(s)$, and disturbance rejection TF $H_{\mathrm{di}} = i_{\mathrm{c}}(s)/v_{\mathrm{g}}(s)$. We have developed a Python function (`LoopTFsCalculation_PIgains()`) that calculates the transfer functions with the PI gains, and transfer functions of loop delays and plant, as inputs. The function returns a dictionary with the entries "`PI`", "`Hoi`", "`Hci`", "`Hdi`", "`Plant`", and "`Delays`", and the associated transfer functions calculated. You can use the following code:

```
8 TFs = LoopTFsCalculation_PIgains(Param['Ki'], Param['Ti'], Plant_TF, Delays_TF)
```

**5** As explained in [1], you can also calculate a 1st-order approximation of the closed-loop transfer function, given by the equation $H_{\mathrm{ci}}(s) \approx \frac{1}{s \cdot T_{\mathrm{et}}+1}$, with $T_{\mathrm{et}} = 2 \cdot T_{\mathrm{ei}}$. The following code lets you do that:

```
9 Param['Tet'] = 2*Param['Tei']
10 TFs['Hci'].append(1/(1+s*Param['Tet']))
```

**6** Now that the transfer functions of the system have been calculated, we can obtain the frequency responses of the closed-loop current control. For this purpose we can use the function `bode()` included in the Control System Library. We have developed a Python function, `PlotFreqResponses()`, that creates a plot with the step responses of the transfer function provided. The function also requires the title of the figure as a string, and a list of strings containing the names to be displayed in the legend. You can use the following code for this purpose:

```
11 FigNameFreq1 = 'Fig: Frequency response of closed-loop current control transfer functions'
12 LegendFreq1 = ['Hci Optimum tracking','Hci Improved rejection','Hci 1st order approx.']
13 PlotFreqResponses(TFs['Hci'],FigNameFreq1,LegendFreq1)
14 FigNameFreq2 = 'Fig: Frequency response of disturbance rejection transfer functions'
15 LegendFreq2 = ['Hdi Optimum tracking','Hdi Improved rejection']
16 PlotFreqResponses(TFs['Hdi'],FigNameFreq2,LegendFreq2)
```

The frequency responses are shown in Fig. 4. It can be seen that both systems achieve a similar current regulation bandwidth of approximately 1000 Hz (for a switching frequency of 20 kHz), while the system with the PI tuned for improved disturbance rejection will have larger overshoots. However, this poorer current tracking capability is compensated by a significantly better disturbance rejection capability. We can expect that the current regulator tuned for improved disturbance rejection is able to clear the voltage perturbations 8 times faster than the current regulator tuned for optimum tracking.

**7** Continuing the analysis, from the transfer functions we can also obtain the system transient response to a change in the current reference of the controller, or the grid voltage. For this purpose we can use the function `step_response()` included in the control library. Similarly, as in the previous exercise, we have developed a Python function, `PlotStepResponses()`, that creates a plot with the step responses of the transfer function provided. The function also requires the title of the figure as a string, a list of strings containing the names to be displayed in the legend, and a list with the first value being the time at which the step occurs, and the second value being the magnitude of the input step change. Additionally, the function returns the data used for the plots in a list. We will need this data to later compare this results with the PLECS simulation results. You can use the following code for this purpose:

```
17 TitleStep1 = 'Fig: Step response of closed-loop current control transfer functions'
18 LegendStep1 = ['Hci Optimum tracking','Hci Improved rejection','Hci 1st order approx.']
19 XYLabels1 = ['time (s)','Inductor Current (A)']
20 StepData_Hci = PlotStepResponses(TFs['Hci'],TitleStep1,LegendStep1,XYLabels1, ...
21     [Param['t_stepRef'],Param['Istep']])
```

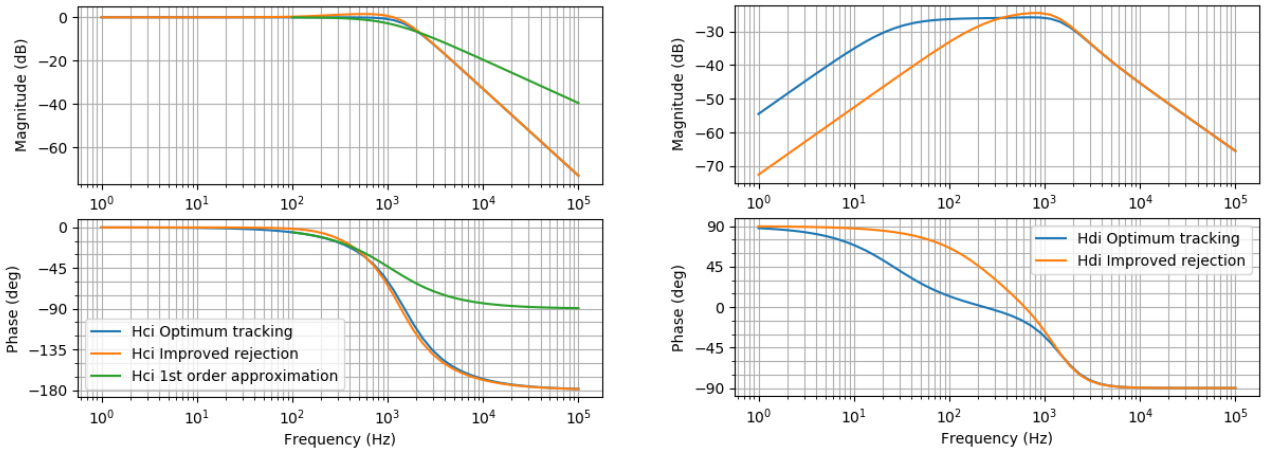


**Figure 4: Frequency responses comparison of closed-loop $H_{ci}(s)$ and $H_{di}(s)$ for the different current controller designs**

```
TitleStep2 = 'Fig: Step response of disturbance rejection transfer functions'
LegendStep2 = ['Hdi Optimum tracking','Hdi Improved rejection']
XYLabels2 = ['time (s)','Inductor Current (A)']
StepData_Hdi = PlotStepResponses(TFs['Hdi'],TitleStep2,LegendStep2,XYLabels2,[0,1])
```

The expected step responses of the designed current controllers are shown in Fig. 5. We can more clearly see here that while the the PI regulator optimized for disturbance rejection has the poorer current tracking performance, it provides significant improvement over the other design in mitigating disturbances.

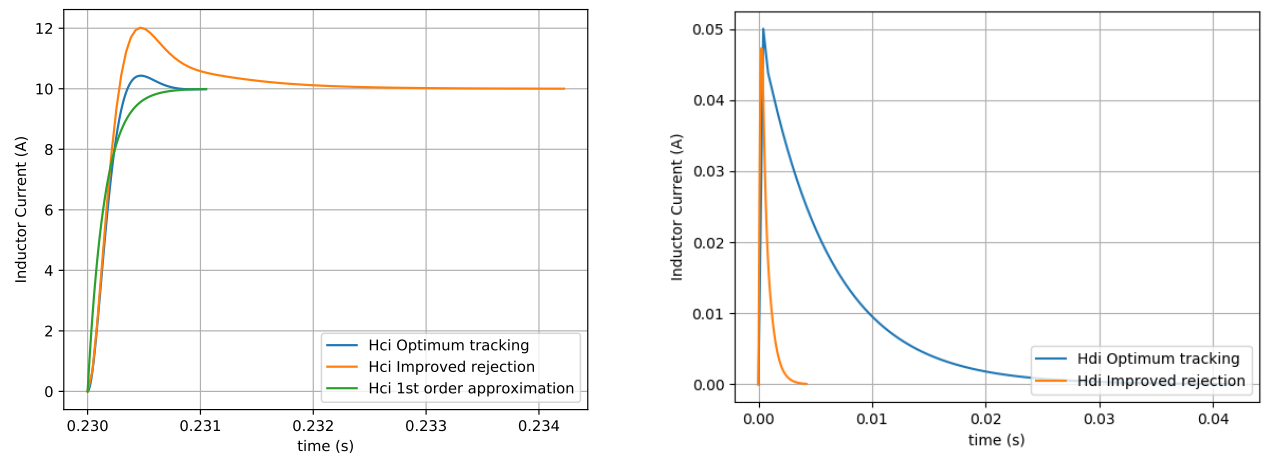


**Figure 5: Step responses comparison of closed-loop $H_{\mathrm{ci}}(s)$ and $H_{\mathrm{di}}(s)$ for the different current controller designs**

At this stage your code should be the same as in the Python file `xmlrpc_controller_design_1.py`.

# 3 Launching a PLECS Simulation and Retrieving Data in Python

In this section we will show how to launch the simulation of the PLECS model of the system, and retrieve the simulation data for post-processing.

**Your Task:** Your task consists of loading the PLECS model to be simulated, defining the simulation parameters to be used, simulating the PLECS model, and closing the PLECS model.

**1** Once the XML-RPC server has been started and the connection is established, we can load the PLECS model we will simulate. If you have the current working Python script and the PLECS model in the same folder, we can use the `os` module for this purpose. The function we need to use to obtain the current working directory are the `path.abspath()` and `getcwd()`. Then, we can build the PLECS file path with the current working directory and the file name:

```
26 full_path = os.path.abspath(os.getcwd())
27 file_name = 'xmlrpc_controller_design'
28 server.plecs.load(full_path+'/'+file_name)
```

**2** Before launching the simulation, we need to define the simulation parameters we will use. We have designed two different PI regulators, with different $T_i$ gains. Therefore, in order to simulate both regulators, we will use the same PLECS model, but we will pass to the simulation different values for $T_i$. You can use our Python function `LoadSimParameters()` to create the struct variable with the model parameters (“`Param`”). These are the same values previously defined and used for the theoretical analysis. The function returns a dictionary, with the variable structure defined in Section 1.2 to be used in the XML-RPC simulation commands:

```
29  opts = LoadSimParameters(Param)
```

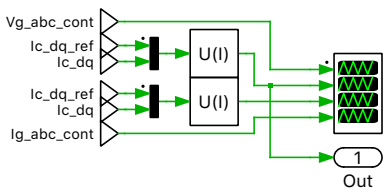


**Figure 6: Top level schematic of simulated model containing an outport for accessing the data in Python**

**3** If any outports exist on the top level schematic of the simulated model, the simulation command returns a struct consisting of two fields, “Time" and “Values”. “Time” is a vector that contains the simulation time values, in seconds, for each simulation step. The rows of the array “Values" consist of the signal value magnitudes seen at the outports. The order of the signals is determined by the port numbers. As you can see if Fig. 6, the Signal Outport block in the PLECS model contains the signals for the current control reference in the d-axis, and the measured d-axis current. We need to define variables where we will store the simulation results for both the signals and time information. We can use the following code for that:

```
30 Time = []
31 Ic_d_ref_sim = []
32 Ic_d_sim = []
```

**4** The simulation can be launched with the command `Data = server.plecs.simulate('mdlName', optStruct)`. We need to repeat this process twice, where we will pass to the PLECS model a different *optStruct* that contains the different values of $T_i$ we have previously calculated. In this command *mdlName* is the name of the model to be simulated, *optStruct* is the variable struct containing the model parameters, and the format type of the simulation data. 'MatFile' data format is used because the process for the data transmission is faster. `Data_sim1 Data_sim2` store the PLECS simulation results in a dictionary. Your code should look like the following:

```
33 opts['ModelVars']['Ti'] = Param['Ti'][0]
34 opts['OutputFormat'] = 'MatFile'
35 Data_raw = server.plecs.simulate(file_name, opts).data
36 Data_sim1 = sio.loadmat(io.BytesIO(Data_raw))
37 Time.append( Data_sim1['Time'][0] )
38 Ic_d_ref_sim.append( Data_sim1['Values'][0] )
39 Ic_d_sim.append( Data_sim1['Values'][1] )
40 opts['ModelVars']['Ti'] = Param['Ti'][1]
41 Data_raw = server.plecs.simulate(file_name, opts).data
42 Data_sim2 = sio.loadmat(io.BytesIO(Data_raw))
43 Time.append( Data_sim2['Time'][0] )
44 Ic_d_ref_sim.append( Data_sim2['Values'][0] )
45 Ic_d_sim.append( Data_sim2['Values'][1] )
```

**5** We have now obtained the simulation results, so we can use the following command to close the simulated model: `server.plecs.close('mdlName')`, where *mdlName* is the name of the model that you want to close. In this script the name is stored in the variable `file_name`:

```
46 server.plecs.close(file_name)
```

At this stage your code should be the same as in the Python file `xmlrpc_controller_design_2.py`.

**Your Task:** Now that the simulated results have been obtained, you can compare the PLECS simulation results with the theoretical responses previously obtained, in order to assess how accurate the system modeling and controller design has been.

**1** In order to compare the theoretical transient response and the simulated one you can use our function `PlotSimResults()`, which requires a list of [X,Y] pair values containing the data to be plotted. Additionally, you need to provide the figure title as a string, a list of strings containing the names to be displayed in the legend, and a list of strings containing the labels of the x and y axes. The function returns the figure handle.

Now we are ready to obtain the plots comparing the theoretical and simulated step responses for the designed PI current regulators:

```
47 Plot_XY_Pairs = [[Time[0],Ic_d_ref_sim[0]],[Time[0],Ic_d_sim[0]],StepData_Hci[0]]
48 TitleResults = 'Fig: Theoretical responses Vs Simulated model, PI reg. 1'
49 LegendResults = ['Reference','Simulated','Theoretical']
50 LabelResults = ['time (s)','Id (A)']
51 plt1 = PlotSimResults(Plot_XY_Pairs,TitleResults,LegendResults,LabelResults,Param)
52 Plot_XY_Pairs = [[Time[1],Ic_d_ref_sim[1]],[Time[1],Ic_d_sim[1]],StepData_Hci[1]]
53 TitleResults = 'Fig: Theoretical responses Vs Simulated model, PI reg. 2'
54 LegendResults = ['Reference','Simulated','Theoretical']
55 LabelResults = ['time (s)','Id (A)']
56 plt2 =PlotSimResults(Plot_XY_Pairs,TitleResults,LegendResults,LabelResults,Param)
57 plt2.show()
```

The obtained figures with the responses comparison are shown in Fig. 7.

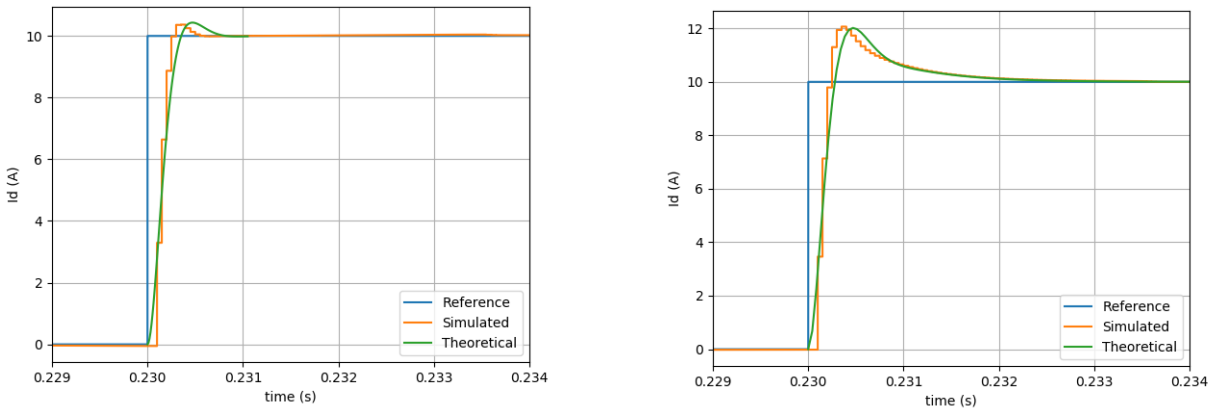


**Figure 7: Transient response comparison to a 10 A current reference step for both designed current regulators. On the left side is the PI regulator design for optimum current tracking, and on the right side is the PI regulator designed for improved disturbance rejection**

At this stage your code should be the same as in the Python file `xmlrpc_controller_design.py`.

# References

[1] Blasko, V., & Kaura, V., "A new mathematical model and control of a three-phase AC-DC voltage source converter," *IEEE Transactions on Power Electronics*, Jan. 1997, pp. 116–123.

**Revision History:**

| | |
|---|---|
| Tutorial Version 1.0 | First release |
| Tutorial Version 1.1 | Use binary format to transfer data to PLECS |

**How to Contact Plexim:**


| | | |
|---|---|---|
| ☎ | +41 44 533 51 00 | Phone |
| | +41 44 533 51 01 | Fax |
| ✉ | Plexim GmbH<br>Technoparkstrasse 1<br>8005 Zurich<br>Switzerland | Mail |
| @ | `info@plexim.com` | Email |
| | `http://www.plexim.com` | Web |


*PLECS Tutorial*