FMU Export of a Python-driven Simulation Program

Release 1.0.0rc15

LBNL - Building Technology and Urban Systems Division

Jun 22, 2018

CONTENTS

CHAPTER

ONE

INTRODUCTION

This user manual explains how to install and use SimulatorToFMU.

SimulatorToFMU is a software package written in Python which allows users to export a Python-driven simulation program or script as a *Functional Mock-up Unit* (FMU) for model.simulator or co-simulation using the *Functional Mock-up Interface* (FMI) standard version 1.0 or 2.0. This FMU can then be imported into a variety of simulation programs that support the import of Functional Mock-up Units. In the remainder of this document, we define a Python-driven simulation program, and a Python script as a Simulator.

Note: SimulatorToFMU generates FMUs that use the Python 2.7/ C API for interfacing with the simulators.

CHAPTER

TWO

INSTALLATION AND CONFIGURATION

This chapter describes how to install, configure, and uninstall SimulatorToFMU on Windows and Linux operating systems. SimulatorToFMU is currently not supported on Mac OS.

Software requirements

To export a Simulator as an FMU, SimulatorToFMU needs:

1. Python and following dependencies:
   - jinja2
   - lxml
2. Modelica parser
3. C-Compiler

SimulatorToFMU has been tested with:

- Python 2.7.12 (Linux) and 2.7.13 (Windows)
- Three Modelica parsers
  - Dymola 2018 on Windows and Linux
  - JModelica 2.0 on Windows, and JModelica trunk version 9899 on Linux
  - OpenModelica 1.11.0 on Windows
- C-Compiler: Microsoft Visual Studio 10 Professional

Installation

To install SimulatorToFMU, proceed as follows:

1. Add following folders to your system path:
   - Python installation folder (e.g. `C:\Python27`)
   - Python scripts folder (e.g. `C:\Python27\Scripts`),
   - Dymola executable folder (e.g. `C:\Program Files(x86)\Dymola2018\bin`)
   - JModelica installation folder (e.g. `C:\JModelica.org-2.0`)

- OpenModelica executable folder (e.g. `C:\OpenModelica1.11.0-32bit\bin`)

You can add folders to your system path by performing following steps on Windows 8 or 10:

- In Search, search for and then select: System (Control Panel)
- Click the Advanced system settings link.
- Click Environment Variables. In the section System Variables, find the PATH environment variable and select it. Click Edit.
- In the Edit System Variable (or New System Variable) window, specify the value of the PATH environment variable (e.g. `C:\Python27`, `C:\Python27\Scripts`). Click OK. Close all remaining windows by clicking OK.
- Reopen Command prompt window for your changes to be active.

To check if the variables have been correctly added to the system path on Windows, type `python`, `dymola`, `pylab`, or `omc` into a command prompt to see if the right version of Python, Dymola, JModelica, or OpenModelica starts up.

Note:

- To avoid adding Dymola, JModelica, or OpenModelica to the system path, provide the path to the executables to SimulatorToFMU.py. See *Command-line use* for the lists of arguments of SimulatorToFMU.
- SimulatorToFMU sets the hidden Dymola 2018's flag `Advanced.AllowStringParametersForFMU` to `true` when exporting a simulation program/script as an FMU. The flag is not available in older versions of Dymola. The flag is required to allow a master algorithm to set the path to the configuration file of an FMU. See section *Command-line use* for more details.

---

2. Install SimulatorToFMU by running

```
> pip install --user SimulatorToFMU
```

Note: Use the `--user` command line option to install SimulatorToFMU so it can be installed in your Python 2.7 user installation directory and can write files to your disk. The Python 2.7 user installation directory is typically `C:\Users\YourUserName\AppData\Roaming\Python\Python27\site-packages` on Windows, and `/home/YourUserName/.local/lib/python2.7/site-packages` on Linux where `YourUserName` is your system login user name.

The installation directory should contain the following subdirectories:

- `bin/` (Scripts for running unit tests)
- `doc/` (Documentation sources)
- `fmus/` (FMUs folder)
- `parser/` (Python scripts, Modelica templates and XML validator files)

UnitTests

To test your installation run from the installation `bin` folder

```
> python runUnitTest.py
```

Uninstallation

To uninstall SimulatorToFMU, run

```
> pip uninstall SimulatorToFMU
```

CHAPTER

THREE

BEST PRACTICE

This section explains to users the best practice in configuring a Simulator XML input file, and implementing the Python wrapper which will interface with the Simulator.

Configuring the Simulator XML input file

To export a Simulator as an FMU, the user needs to write an XML file which contains the list of inputs, outputs and parameters of the FMU. The XML snippet below shows how a user has to write such an input file. A template named `SimulatorModeldescritpion.xml` which shows such a file is provided in the `parser/utilities` installation folder of SimulatorToFMU. This template should be adapted to create new XML input file.

The following snippet shows an input file where the user defines 1 input and 1 output variable.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <SimulatorModelDescription
3    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4    fmiVersion="2.0"
5    modelName="simulator"
6    description="Input data for a Simulator FMU"
7    generationTool="SimulatorToFMU">
8    <ModelVariables>
9      <ScalarVariable
10       name="v"
11       description="Voltage"
12       causality="input"
13           type="Real"
14           unit="V"
15           start="0.0">
16     </ScalarVariable>
17     <ScalarVariable
18       name="i"
19       description="Current"
20       causality="output"
21           type="Real"
22           unit="A">
23     </ScalarVariable>
24   </ModelVariables>
25 </SimulatorModelDescription>
```

To create such an input file, the user needs to specify the name of the FMU (Line 5). This is the `modelName` which should be unique. The user then needs to define the inputs and outputs of the FMUs. This is done by adding a `ScalarVariable` into the list of `ModelVariables`.

To parametrize the `ScalarVariable` as an input variable, the user needs to

- define the name of the variable (Line 10),
- give a brief description of the variable (Line 11)
- give the causality of the variable (`input` for inputs, `output` for outputs) (Line 12)
- define the type of variable (Currently only `Real` variables are supported) (Line 13)
- give the unit of the variable (Currently only *Modelica* units are supported) (Line 14)
- give a start value for the input variable (This is optional) (Line 15)

To parametrize the `ScalarVariable` as an output variable, the user needs to

- define the name of the variable (Line 18),
- give a brief description of the variable (Line 19)
- give the causality of the variable (`input` for inputs, `output` for outputs) (Line 20)
- define the type of variable (Currently only `Real` variables are supported) (Line 21)
- give the unit of the variable (Currently only *Modelica* units are supported) (Line 22)

Note: If *Modelica* units can't be used (Line 14 and Line 22), then remove the *unit* field from the input file when defining new `ScalarVariable`.

Configuring the Python Wrapper Simulator

To export a Simulator as an FMU, the user needs to write the Python wrapper which will interface with the Simulator. The wrapper will be embedded in the FMU when the Simulator is exported and used at runtime on the target machine.

The user needs to extend the Python wrapper provided in `parser/utilities/simulator_wrapper.py` and implements the function *.simulator*'.

The following snippet shows the Simulator function.

```
# Dummy Python-driven simulator
class Simulator():
    """
    Dummy simulator Python-driven simulator
    which increments in its doTimeSteo method the input values by 1.
    This class is for illustration purposes only.
    """
    def __init__(self, configuration_file, time, input_names,
            input_values, output_names, write_results):
        self.configuration_file = configuration_file
        self.input_values = input_values


    def doTimeStep(self, input_values):
        """
        This function increments the input variables by 1
        """

        return input_values + 1

# Main Python function to be modified to interface with a simulator which has memory.
def.simulator(configuration_file, time, input_names,
```

```
            input_values, output_names, write_results,
            memory):
    """
    Return  a list of output values from the Python-based Simulator.
    The order of the output values must match the order of the output names.

    :param configuration_file (String): Path to the Simulator model or configuration file
    :param time (Float): Simulation time
    :param input_names (Strings): Input names
    :param input_values (Floats): Input values (same length as input_names)
    :param output_names (Strings): Output names
    :param write_results (Integers): Store results to file (1 to store, 0 else)
    :param memory: Variable that stores the memory of a Python object

    """

    ######################################################################
    # EDIT AND INCLUDE CUSTOM CODE FOR TARGET SIMULATOR
    # Include body of the function used to compute the output values
    # based on the inputs received by the simulator function.
    # This will need to be adapted so it returns the correct output_values.
    # If the list of output names has only one name, then only a scalar
    # must be returned.
    # The snippet shows how a Python object should be held in the memory
    # This is done by getting the object from the.simulator function, modifying it,
    # and returning it.
    ######################################################################
    # Since master algorithms need to some time call at the same time instant
    # an FMU multiple times for event iteration. It is for efficient reasons
    # good to catch the simulator input and outputs results, along with the current
    #  and past simulation times to determine when the Simulator needs to be reinvoked.
    if memory == None:
        # Initialize the Python object
        s = Simulator(configuration_file, time, input_names,
                      input_values, output_names, write_results)
        memory = {'memory':s, 'tLast':time, 'outputs':None}
        if not (input_values is None):
            memory['inputsLast'] = input_values
            memory['outputs'] = s.doTimeStep(input_values)
        else:
            # Return default output
            memory['outputs'] = 1.0
        memory['s'] = s
    else:
        # Check if inputs values have changed
        if not (input_values is None):
            newInputs = sum([abs(m - n) for m, n in zip (input_values,
            memory['inputsLast'])])
        # Check if time has changed prior to updating the outputs
        if(abs(time - memory['tLast'])>1e-6 or newInputs > 0):
            # Updtate the outputs of the Simulator
            memory['outputs'] = memory['s'].doTimeStep(memory['outputs'])
            # Save last time
            memory['tLast'] = time
            # Save last input values
            memory['inputsLast'] = input_values
```

```
    # Handle errors
    if(memory['outputs'] < 0.0):
            raise("The memory['outpus'] cannot be null")
    # Save the output of the Simulator
    output_values = memory['outputs']
    ########################################################################
    return [output_values, memory]

if __name__ == "__main__":
    memory = None
    print.simulator("dummy.csv", 0.0, "v", None, "i", 0, memory))
```

The arguments of the functions are in the next table

Arguments	Description
configuration_file	The Path to the Simulator model or configuration file
time	The current simulation model time
input_names	The list of input names of the FMU
input_values	The list of input values of the FMU
output_names	The list of output names of the FMU
output_values	The list of output values of the FMU
write_results	A flag for writing the simulation results to a file located in the working directory of the importing tool.
memory	A variable that holds the memory of a Python object This argument is required only if the simulator has variables which have memory.

If the simulator does not have memory, then the function *.simulator'* will be defined as

```
# Dummy Python-driven simulator
class Simulator():
    """
    Dummy simulator Python-driven simulator
    which increments in its doTimeSteo method the input values by 1.
    This class is for illustration purposes only.
    """
    def __init__(self, configuration_file, time, input_names,
            input_values, output_names, write_results):
        self.configuration_file = configuration_file
        self.input_values = input_values


    def doTimeStep(self, input_values):
        """
        This function increments the input variables by 1
        """

        return input_values + 1

# Main Python function to be modified to interface with a simulator which has memory.
def.simulator(configuration_file, time, input_names,
            input_values, output_names, write_results):
    """
    Return  a list of output values from the Python-based Simulator.
    The order of the output values must match the order of the output names.

    :param configuration_file (String): Path to the Simulator model or configuration␣
↪file
```

```
    :param time (Float): Simulation time
    :param input_names (Strings): Input names
    :param input_values (Floats): Input values (same length as input_names)
    :param output_names (Strings): Output names
    :param write_results (Integers): Store results to file (1 to store, 0 else)

    """

    ###################################################################
    # EDIT AND INCLUDE CUSTOM CODE FOR TARGET SIMULATOR
    # Include body of the function used to compute the output values
    # based on the inputs received by the simulator function.
    # This will need to be adapted so it returns the correct output_values.
    # If the list of output names has only one name, then only a scalar
    # must be returned.
    ###################################################################
    # Since master algorithms need to some time call at the same time instant
    # an FMU multiple times for event iteration. It is for efficient reasons
    # good to catch the simulator outputs results, and use the current and past
    # simulation times to determine when the Simulator needs to be reinvoked

    # Call the Simulator
    s = Simulator(configuration_file, time, input_names,
                          input_values, output_names, write_results)
    if not (input_values is None):
        output_values=s.doTimeStep(input_values)
    else:
        # Return default output value
        output_values = 1.0

    # Handle errors
    if(output_values < 0.0):
            raise("The memory['outpus'] cannot be negative.")
    # Save the output of the Simulator
    ###################################################################
    return output_values

#if __name__ == "__main__":
#    print.simulator("dummy.csv", 0.0, "v", 1.0, "i", 0))
```

Note:

- The function *.simulator‘* must return a list of output values which matches the order of the output names.
- If the simulator has memory, then the function *.simulator‘* must also return the memory.
- The function *.simulator‘* can be used to invoke external programs/scripts which do not ship with the FMU. The external programs/scripts will have to be installed on the target machine where the FMU is run. See *Creating an FMU* for details on command line options.
- Once `simulator_wrapper.py` is implemented, it must be saved under a name of the form `"modelname" + "_wrapper.py"`, and its path used as required argument for `SimulatorToFMU.py`.

CHAPTER

FOUR

CREATING AN FMU

This chapter describes how to create a Functional Mockup Unit. It assumes you have followed the *Installation and Configuration* instructions, and that you have created the Simulator model description file as well as the Python script required to interface the Simulator following the *Best Practice* guidelines.

Command-line use

To create an FMU, open a command-line window (see *Notation*). The standard invocation of the SimulatorToFMU tool is:

```
> python  <scriptDir>SimulatorToFMU.py -s <python-scripts-path>
```

where `scriptDir` is the path to the scripts directory of SimulatorToFMU. This is the `parser` subdirectory of the installation directory. See *Installation and Configuration* for details.

An example of invoking `SimulatorToFMU.py` on Windows is

```
# Windows:
> python parser\SimulatorToFMU.py -s parser\utilities\simulator_wrapper.py,d:\calc.py
```

Following requirements must be met when using SimulatorToFMU

- All file paths can be absolute or relative.
- If any file path contains spaces, then it must be surrounded with double quotes.

`SimulatorToFMU.py` supports the following command-line switches:

The main functions of SimulatorToFMU are

- reading, validating, and parsing the Simulator XML input file. This includes removing and replacing invalid characters in variable names such as $*+-$ with _,
- writing Modelica code with valid inputs and outputs names,
- invoking a Modelica compiler to compile the *Modelica* code as an FMU for model.simulator or co-simulation `1.0` or `2.0`.

The next section discusses requirements of some of the arguments of SimulatorToFMU.

Simulation model or configuration file

An FMU exported by SimulatorToFMU needs in certain cases a configuration file to run. There are two ways of providing the configuration file to the FMU:

1. The path to the configuration file is passed as the command line argument `"-c"` of SimulatorToFMU.py. In this situation, the configuration file is copied in the resources folder of the FMU.
2. The path to the configuration is set by the master algorithm before initializing the FMU.

Note: The name of the configuration variable is `_configurationFileName`. This name is reserved and should not be used for FMU input and output names.

Depending on the tool used to export the FMU, following requirements/restrictions apply:

Dymola

- If the path to the configuration file is provided, then Dymola copies the file to its resources folder and uses the configuration file at runtime. In this case, the path to the configuration file can't be set and changed by the master algorithm.
- If the configuration file is not provided, then the path to the configuration file must be set by the master algorithm prior to initializing the FMU.

JModelica

- If the path to the configuration file is provided, then JModelica will not copy it to the resources folder of the FMU. Instead, the path to the configuration is hard-coded in the FMU. As a further restriction, the path to the configuration file can't be set and changed by the master algorithm.

  These are known limitations in JModelica 2.0. The workaround is to make sure that the path of the configuration file is the same on the machine where the FMU will be run.
- If the configuration file is not provided, then SimulatorToFMU will issue a warning.

OpenModelica

- If the path to a configuration file is provided, then OpenModelica will not copy it to the resources folder of the FMU. Instead, the path to the configuration is hard-coded in the FMU. However, the path to the configuration file can be set and changed by the master algorithm.

  This is a known limitation in OpenModelica 1.11.0. The workaround is to either make sure that the path of the configuration file is the same on the machine where the FMU will be run, or set the path of the configuration file when running the FMU.
- If the configuration file is not provided, then the path to the configuration file must be set by master algorithm prior to initializing the FMU.

Reserved variable names

Following variables names are not allowed to be used as FMU input, output, or parameter names.

- `_configurationFileName`: String variable name used to set the path to the Simulator model or configuration file.
- `_saveToFile`: Boolean variable used to set the flag for storing simulation results (true for storing, false else).
- `time`: Internal FMU simulation time.

If any of these variables is used for an FMU input, output, or parameter name, SimulatorToFMU will exit with an error.

Outputs of SimulatorToFMU

The main outputs from running `SimulatorToFMU.py` consist of an FMU named after the `modelName` specified in the input file, a zip file called `"modelname" + ".scripts.zip"`, and a zip file called `"modelname" + ".binaries.zip"`. That is, if the `modelName` is called `Simulator`, then the outputs of `SimulatorToFMU` will be `Simulator.fmu`, `Simulator.scripts.zip`, and `Simulator.binaries.zip`.

The FMU and the zip file are written to the current working directory, that is, in the directory from which you entered the command.

`"modelname" + ".scripts.zip"` contains the Python scripts that are needed to interface with the Simulator. The unzipped folder must be added to the `PYTHONPATH` of the target machine where the FMU will be used.

`"modelname" + ".binaries.zip"` contains subdirectories with binaries files that are needed to interface with the Simulator. Subdirectories with binaries to be supported by the FMU must be added to the system `PATH` on Windows or the `LD_LIBRARY_PATH` on Linux. That is, if the FMU is exported for Windows 32 bit, then the subdirectory `"win32"` must be added to the system `PATH` of the target machine where the FMU is run.

Any secondary output from running the SimulatorToFMU tools can be deleted safely.

Note that the FMU itself is a zip file. This means you can open and inspect its contents. To do so, it may help to change the "`.fmu`" extension to "`.zip`".

Note:

- FMUs exported using OpenModelica 1.11.0 needs significantly longer compilation/simulation time compared to the tested versions of Dymola and JModelica.
- FMUs exported using Dymola 2018 needs a Dymola runtime license to run. A Dymola runtime license is not be needed if the FMU is exported with a version of Dymola which has the `Binary Model Export` license.

CHAPTER

FIVE

DEVELOPMENT

The development site of this software is at https://github.com/LBNL-ETA/SimulatorToFMU.

To clone the master branch, type

git clone https://github.com/LBNL-ETA/SimulatorToFMU.git

CHAPTER

SIX

HELP

This chapter lists potential issues encountered when using SimulatorToFMU.

Compilation failed with Dymola

If the export of the Simulator failed when compiling the model with Dymola, comment out `"exit()"` in `parser/utilities/SimulatorModelica_Template_Dymola.mos` with `"//exit()"`, and re-run `SimulatorToFMU.py` to see why the complation has failed.

Compilation failed with OpenModelica

If the export of the Simulator failed when compiling the model with OpenModelica, check if the variable `OPENMODELICALIBRARY` is defined in the Windows `Environment Variables`.

Note: `OPENMODELICALIBRARY` is the path to the libraries which are required by OpenModelica to compile Modelica models.

Simulation failed when running `Simulator.fmu`

If the simulation failed with the exported FMU, check if the unzipped `"modelname" + ".scripts.zip"`, and the subdirectories of `"modelname" + ".binaries.zip"` were added to the `PYTHONPATH`, and the system `PATH` (Windows) /`LD_LIBRARY_PATH` (Linux) respectively as described in *Outputs of SimulatorToFMU*.

Note: Any software or Python 2.7 module which is required to run the exported FMU will need to be installed on the target machine where the FMU is run.

Simulation failed with Dymola FMUs

If an FMU exported using Dymola fails to run, check if the version of Dymola which exported the FMU had the `Binary Model Export` license. The `Binary Model Export` license is required to export FMUs which can be run without requiring a Dymola runtime license. You can also inspect the model description of the FMU to see if a Dymola runtime license is required to run the FMU.

CHAPTER

SEVEN

NOTATION

This chapter shows the formatting conventions used throughout the User Guide.

The command-line is an interactive session for issuing commands to the operating system. Examples include a DOS prompt on Windows, a command shell on Linux, and a Terminal window on MacOS.

The User Guide represents a command window like this:

```
# This is a comment.
> (This is the command prompt, where you enter a command)
(If shown, this is sample output in response to the command)
```

Note that your system may use a different symbol than ">" as the command prompt (for example, "`$`"). Furthermore, the prompt may include information such as the name of your system, or the name of the current subdirectory.

CHAPTER

EIGHT

GLOSSARY

Dymola Dymola, Dynamic Modeling Laboratory, is a modeling and simulation environment for the Modelica language.

Functional Mock-up Interface The Functional Mock-up Interface (FMI) is the result of the Information Technology for European Advancement (ITEA2) project *MODELISAR*. The FMI standard is a tool independent standard to support both model.simulator and co-simulation of dynamic models using a combination of XML-files, C-header files, C-code or binaries.

Functional Mock-up Unit A simulation model or program which implements the FMI standard is called Functional Mock-up Unit (FMU). An FMU comes along with a small set of C-functions (FMI functions) whose input and return arguments are defined by the FMI standard. These C-functions can be provided in source and/or binary form. The FMI functions are called by a simulator to create one or more instances of the FMU. The functions are also used to run the FMUs, typically together with other models. An FMU may either require the importing tool to perform numerical integration (model.simulator) or be self-integrating (co-simulation). An FMU is distributed in the form of a zip-file that contains shared libraries, which contain the implementation of the FMI functions and/or source code of the FMI functions, an XML-file, also called the model description file, which contains the variable definitions as well as meta-information of the model,additional files such as tables, images or documentation that might be relevant for the model.

Modelica Modelica is a non-proprietary, object-oriented, equation-based language to conveniently model complex physical systems containing, e.g., mechanical, electrical, electronic, hydraulic, thermal, control, electric power or process-oriented subcomponents.

MODELISAR MODELISAR is an ITEA 2 (Information Technology for European Advancement) European project aiming to improve the design of systems and of embedded software in vehicles.

PyFMI PyFMI is a package for loading and interacting with Functional Mock-Up Units (FMUs), which are compiled dynamic models compliant with the Functional Mock-Up Interface (FMI).

Python Python is a dynamic programming language that is used in a wide variety of application domains.

CHAPTER

NINE

ACKNOWLEDGMENTS

The development of this documentation was supported by the Assistant Secretary for Energy Efficiency and Renewable Energy, Office of Building Technologies of the U.S. Department of Energy, under contract No. DE-AC02-05CH11231.

The following people contributed to the development of this program:

- Thierry Stephane Nouidui, Lawrence Berkeley National Laboratory
- Michael Wetter, Lawrence Berkeley National Laboratory

CHAPTER

TEN

DISCLAIMERS

CHAPTER

ELEVEN

COPYRIGHT AND LICENSE

Copyright

If you have questions about your rights to use or distribute this software, please contact Berkeley Lab's Innovation & Partnerships Office at IPO@lbl.gov.

NOTICE. This Software was developed under funding from the U.S. Department of Energy and the U.S. Government consequently retains certain rights. As such, the U.S. Government has been granted for itself and others acting on its behalf a paid-up, nonexclusive, irrevocable, worldwide license in the Software to reproduce, distribute copies to the public, prepare derivative works, and perform publicly and display publicly, and to permit others to do so.

License Agreement

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

(1) Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

(2) Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

(3) Neither the name of the University of California, Lawrence Berkeley National Laboratory, U.S. Dept. of Energy nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

You are under no obligation whatsoever to provide any bug fixes, patches, or upgrades to the features, functionality or performance of the source code ("Enhancements") to anyone; however, if you choose to make your Enhancements available either publicly, or directly to Lawrence Berkeley National Laboratory, without imposing a separate written license agreement for such Enhancements, then you hereby grant the following license: a non-exclusive, royalty-free, perpetual license to install, use, modify, prepare derivative works, incorporate into other computer software, distribute, and sublicense such enhancements or derivative works thereof, in binary and source code form.

PYTHON MODULE INDEX

p

INDEX