MPCPy User Guide

Release 0.1

David H. Blum

March 01, 2017

CONTENTS

CHAPTER

ONE

INTRODUCTION

MPCPy facilitates the testing and implementation of occupant-integrated model predictive control (MPC) for building systems. The software package focuses on the use of data-driven simplified physical or statistical models to predict building performance and optimize control. Four main modules contain object classes to import data, interact with real or emulated systems, estimate and validate data-driven models, and optimize control inputs. Three other modules contain classes to help track units and provide additional, mainly internal, functionality to MPCPy.

- **ExoData** classes collect external data and process it for use within MPCPy.
- **System** classes represent real or emulated systems to be controlled, collecting measurements from and providing control inputs to the systems.
- **Models** classes represent system models for MPC, managing model simulation, estimation, and validation.
- **Optimization** classes formulate and solve the MPC optimization problems using Models objects.
- **Variable** and **Unit** classes together maintain the association of static or timeseries data with units.
- **Utility** classes provide functionality needed across modules and for interactions with external components.

While MPCPy provides an integration platform, it relies on third-party software packages for model implementation, simulators, parameter estimation algorithms, and optimization solvers. See Section 2 for a dependencies list of the current release.

CHAPTER

TWO

GETTING STARTED

Dependencies

MPCPy takes advantage of many third-party software packages, listed below. It has been tested on Ubuntu 16.04.

Python Packages

- matplotlib 1.5.1
- numpy 1.11.0
- pandas 0.17.1
- python-dateutil 2.4.2
- pytz 2014.10
- scikit-learn 0.18.1
- tzwhere 2.3
- sphinx 1.3.6
- estimationpy

Modelica Compiler and Optimizer, FMU Simulator

- JModelica 1.17

Modelica Packages

- Modelica Standard Library 3.2.2
- Modelica Buildings Library 3.0.0

Installation

1. Install all dependencies listed above according to their respective processes.
2. Create the following environmental variables, where ".../" is replaced by the full directory:
   - JMODELICA_HOME = ".../Jmodelica-1.17"
   - IPOPT_HOME = ".../Ipop-3.12.5"
   - SUNDIALS_HOME = ".../Jmodelica-1.17/ThirdParty/Sundials"
   - CPPAD_HOME = ".../Jmodelica-1.17/ThirdParty/CppAD/"
   - SEPARATE_PROCESS_JVM = ".../jvm/java-8-openjdk-amd64/"

- JAVA_HOME = ".../jvm/java-8-openjdk-amd64/"

3. Add the following to the PYTHONPATH environmental variable, where ".../" is replaced by the full directory:
   - ".../Jmodelica-1.17/Python"
   - ".../Jmodelica-1.17/Python/pymodelica"
   - ".../MPCPy"
4. Add the following to the MODELICAPATH environmental variable:
   - Modelica Standard Library
   - Modelica Buildings Library
5. Test the installation and explore MPCPy use-cases by running the unit tests.

Run Unit Tests

The script bin/runUnitTests.py runs the unit tests of MPCPy. By default, all of the unit tests are run. An optional argument -s [module.class] will run only the specified unit tests module or class.

To run all unit tests from command-line, use the command (shown from the parent directory):

```
> python bin/runUnitTests
```

To run only unit tests in the module test_models from command-line, use the command (shown from the parent directory):

```
> python bin/runUnitTests -s test_models
```

To run only unit tests in the class Estimate_Jmo from the module test_models from the command-line, use the command (shown from the parent directory):

```
> python bin/runUnitTests -s test_models.Estimate_Jmo
```

CHAPTER

THREE

VARIABLES AND UNITS

General

The `variables` and `units` modules are the fundamental building blocks of data management in MPCPy. They provide functionality for assigning and converting between units as well as processing timeseries data.

Generally speaking, variables in MPCPy contain three components:

name

A descriptor of the variable.

data

Constant value or a timeseries.

unit

Assigned to variables and act on the data depending on the requested functionality, such as converting between between units or extracting the data.

A unit assigned to a variable is called the display unit and is associated with a quantity. For each quantity, there is a predefined base unit. The data entered into a variable with a display unit is automatically converted to and stored as the quantity base unit. This way, if the display unit were to be changed, the data only needs to be converted to the new unit upon extraction. For example, the unit Degrees Celsius is of the quantity temperature, for which the base unit is Kelvin. Therefore, data entered with a display unit of Degrees Celsius would be converted to and stored in Kelvin. If the display unit were to be changed to Degrees Fahrenheit, then the data would be converted from Kelvin upon extraction.

Instantiation

Variables are instantiated by defining the variable type and the three components listed in the previous section. If the data of the variable does not change with time, the variable must be instantiated using the `variables.Static` class. Data supplied to static variable may be a single value, a list, or a numpy array. If the data of the variable is a timeseries, the variable must be instantiated using the `variables.Timeseries` class. Data supplied to a timeseries variable must be in the form of a pandas series object with a datetime index. This brings to MPCPy all of the functionality of the pandas package. The unit assigned is a class chosen from the `units` module.

```
# Instantiate a static variable with units in Degrees Celsius
var = variables.Static('var', 20, units.degC)
```

Timeseries variables have capabilities to manage the the timezone of the data as well as clean the data upon instantiation with the following optional keyword arguments:

tz_name

The name of the timezone as defined by `tzwhere`. By default, the UTC timezone is assigned to the data. If a different timezone is assigned, the data is converted to a stored in UTC. Similar to the treatment of data units, the timezone is only converted to the assigned timezone upon data extraction.

geography

Tuple containing (latitude,longitude) in degrees. If geography is defined, the timezone associated with that location will be assigned to the variable.

cleaning_type

The type of cleaning to be performed on the data. This should be a class selected from `variables.Timeseries`.

cleaning_args

Arguments of the cleaning_type defined.

Variable Management

Accessing Data

Data may be extracted from a variable by using the `display_data()` and `get_base_data()` methods. The former will extract the data in the assigned unit, while the latter will extract the data in the base unit.

Setting Display Unit

The display unit of a variable may be changed using the `set_display_unit()` method. This requires a class of the `units` module as an argument.

Setting Data

The data of a variable may be changed using the `set_data()` method. This requires a single value or pandas series object as an argument, depending on the variable type.

Operations

Variables with the same display unit can be added and subtracted using the "+" and "-" operands. The result is a third variable with the resulting data, same display unit, and name as "var1_var2".

CHAPTER

FOUR

EXODATA

General

`exodata` classes are responsible for the representation of exogenous data, with methods to collect this data from various sources and process it for use within MPCPy. This data comes from sources outside of MPCPy and are not measurements of the system of interest. The data is split into categories, or types, in order to standardize the organization of variables within the data for a particular type, in the form of a python dictionary, and to allow for any specific data processing that may be required. This allows exogenous data objects to be used throughout MPCPy regardless of their data source. To add a data source, one only need to create a class that can convert the data format in the source to that standardized in MPCPy. The following is a list of exogenous data types:

- Weather
- Internal
- Control
- Other Input
- Parameter
- Constraint
- Price

Instantiation

`exodata` objects may be instantiated using classes of the naming convention `TypeFromSource`. For example, the class for collecting weather data from an EPW file is called `WeatherFromEPW`. Required arguments for instantiation will differ among the classes depending on the data type and source. However, all `exodata` classes have the following optional keyword arguments upon instantiation:

geography

Tuple containing (latitude,longitude) in degrees. Is required for weather data other than from an EPW. May also be used to detect time zone of data.

tz_name

The name of the timezone as defined by `tzwhere`. If "from_geography" is specified, the latitude and longitude coordinates are used to specify the timezone.

time_format

Timestamp format of the data in a timespec string. Timestamps naturally read by `pandas` do not have to be specified.

time_header

Name of the column or variable containing timestamps of the data. The names “Time”, “time”, “Timestamp”, and “timestamp” do not have to be specified.

clean_data

Dictionary of the form `{ "columnHeader" : "cleaning_type" = mpcpy.Variables.cleaning_type, "cleaning_args" = (cleaning_args)}`. See the `Variables` section for more information on data cleaning.

Collecting Data

Once instantiated, an `exodata` object may collect data using the `collect_data()` method.

```
# Collect data from start_time to final_time
exodata_class.collect_data(start_time, final_time);
```

Accessing Data

Once data is collected, the data dictionary is located at `exodata_object.data`. Note that this data dictionary may also be defined and set in the python environment, without the use of `collect_data`, as long as it conforms to the data type structures described in this section.

The data dictionary may also be converted to a `pandas` dataframe using the `display_data()` and `get_base_data()` methods, with similar differentiation as in MPCPy `Variable` classes.

```
# Make dataframe in display units
exodata_display_df = exodata_object.display_data();
# Make dataframe in base units
exodata_base_df = exodata_object.get_base_data();
```

Weather

Weather data represents the conditions of the ambient environment. Weather data objects have special methods for checking the validity of data and use supplied data to calculate data not directly measured, for example black sky temperature, wet bulb temperature, and sun position.

Structure

Exogenous weather data has the following organization:

```
weather.data = {"Weather Variable Name" : mpcpy.Variables.Timeseries}
```

The weather variable names should match those input variables in the model and be chosen from the following list:

- weaPAtm - atmospheric pressure
- weaTDewPoi - dew point temperature
- weaTDryBul - dry bulb temperature
- weaRelHum - relative humidity

- weaNOpa - opaque sky cover
- weaCelHei - cloud height
- weaNTot - total sky cover
- weaWinSpe - wind speed
- weaWinDir - wind direction
- weaHHorIR - horizontal infrared irradiation
- weaHDirNor - direct normal irradiation
- weaHGloHor - global horizontal irradiation
- weaHDifHor - diffuse horizontal irradiation
- weaIAveHor - global horizontal illuminance
- weaIDirNor - direct normal illuminance
- weaIDifHor - diffuse horizontal illuminance
- weaZLum - zenith luminance
- weaTBlaSky - black sky temperature
- weaTWetBul - wet bulb temperature
- weaSolZen - solar zenith angle
- weaCloTim - clock time
- weaSolTim - solar time
- weaTGnd - ground temperature

Ground temperature is an exception to the data dictionary format due to the possibility of different temperatures at multiple depths. Therefore, the dictionary format for 'ground temperature' is:

```
weather.data["weaTGnd"] = {"Depth" : mpcpy.Variables.Timeseries}
```

Classes

Weather data may be collected using the following classes:

WeatherFromEPW

Collects weather data from an EPW file.

WeatherFromCSV

Collects weather data from a CSV file. This class requires a variable map to match CSV column headers with weather variable names. The variable map is a python dictionary of the form:

```
variable_map = {"Column Header Name" : ("Weather Variable Name",
                                        mpcpy.Units.unit)}
```

Internal

Internal data represents zone heat gains that may come from people, lights, or equipment. Internal data objects have special methods for sourcing these heat gains from a predicted occupancy model.

Structure

Exogenous internal data has the following organization:

```
internal.data = {"Zone Name" : {
                    "Internal Variable Name" : mpcpy.Variables.Timeseries}}
```

The internal variable names should be chosen from the following list:

- intCon - convective internal load
- intRad - radiative internal load
- intLat - latent internal load

The internal variable names input in the model should follow the convention `internalVariableName_zoneName`. For example, the convective load input for the zone "west" should have the name `intCon_west`.

Classes

Internal data may be collected using the following classes:

InternalFromCSV

Collects internal data from a CSV file. This class requires a variable map to match CSV column headers with internal variable names. The variable map is a python dictionary of the form:

```
variable_map = {"Column Header Name" : ("Zone Name",
                                        "Internal Variable Name",
                                        mpcpy.Units.unit)}
```

InternalFromOccupancyModel

Generates internal load data from an occupancy prediction model. This class requires a zone list in the form ["Zone Name 1", "Zone Name 2", "Zone Name 3"], a list of numeric values representing the loads per person in the form [Convective, Radiative, Latent] for each zone and collected in a list, the units of the indicated loads from `mpcpy.Units.unit`, and a list of occupancy model objects with predicted occupancy, one for each zone.

Control

Control data represents control inputs to a system or model. The variables listed in a Control data object are special in that they are considered optimization variables during model optimization.

Structure

Exogenous control data has the following organization:

```
control.data = {"Control Variable Name" : mpcpy.Variables.Timeseries}
```

The control variable names should match the control input variables of the model.

Classes

Control data may be collected using the following classes:

ControlFromCSV

Collects control data from a CSV file. This class requires a variable map to match CSV column headers with control variable names. The variable map is a python dictionary of the form:

```
variable_map = {"Column Header Name" : ("Control Variable Name",
                                        mpcpy.Units.unit)}
```

Other Inputs

Other Input data represents miscellaneous inputs to a model. The variables listed in an Other Inputs data object are not acted upon in any special way.

Structure

Other input data has the following organization:

```
other_input.data = {"Other Input Variable Name" : mpcpy.Variables.Timeseries}
```

The other input variable names should match those of the model.

Classes

Other input data may be collected using the following classes:

OtherInputFromCSV

Collect other input data from a CSV file. This class requires a variable map to match CSV column headers with other input variable names. The variable map is a python dictionary of the form:

```
variable_map = {"Column Header Name" : ("Other Input Variable Name",
                                        mpcpy.Units.unit)}
```

Price

Price data represents price signals from utility or district energy systems for things such as energy consumption, demand, or other services. Price data object variables are special because they are used for optimization objective functions involving price signals.

Structure

Exogenous price data has the following organization:

```
price.data = {"Price Variable Name" : mpcpy.Variables.Timeseries}
```

The price variable names should be chosen from the following list:

- pi_e - electrical energy price

Classes

Price data may be collected using the following classes:

PriceFromCSV

Collects price data from a CSV file. This class requires a variable map to match CSV column headers with price variable names. The variable map is a python dictionary of the form:

```
variable_map = {"Column Header Name" : ("Price Variable Name",
                                        mpcpy.Units.unit)}
```

Constraints

Constraint data represents limits to which the control and state variables of an optimization solution must abide. Constraint data object variables are included in the optimization problem formulation.

Structure

Exogenous constraint data has the following organization:

```
constraint.data = {"State or Control Variable Name" : {
                        "Constraint Variable Name" : mpcpy.Variables.Timeseries/Static}}
```

The state or control variable name must match those that are in the model. The constraint variable names should be chosen from the following list:

- LTE - less than or equal to (Timeseries)
- GTE - greater than or equal to (Timeseries)
- E - equal to (Timeseries)
- Initial - initial value (Static)
- Final - final value (Static)
- Cyclic - initial value equals final value (Static - Boolean)

Classes

Constraint data may be collected using the following classes:

ConstraintFromCSV

Collects timeseries constraint data from a CSV file. Static constraint data must be added by editing the data dictionary directly. This class requires a variable map to match CSV column headers with constraint variable names. The variable map is a python dictionary of the form:

```
variable_map = {"Column Header Name" : ("State or Control Variable Name",
                                        "Constraint Variable Name",
                                        mpcpy.Units.unit)}
```

ConstraintFromOccupancyModel

Generates LTE, GTE, and E constraint data from an occupancy prediction model by implementing occupied and unoccupied values. This class requires a state or control variable list in the form [”Variable Name 1”, “Variable Name 2”, “Variable Name 3”], a list of numeric values representing the occupied and unoccupied constraint values in the form [Occupied, Unoccupied] for each variable collected in a list, a list of constraint variable names, one for each variable, and a list of the units of the indicated numeric values from `mpcpy.Units.unit`.

Parameters

Parameter data represents inputs or coefficients of models that do not change with time during a simulation, which may need to be learned using system measurement data. Parameter data object variables are set when simulating models, and are estimated using model learning techniques if flagged to do so.

Structure

Exogenous parameter data has the following organization:

```
parameter.data = {"Parameter Name" : {
                    "Parameter Variable Name" : mpcpy.Variables.Static}}
```

The parameter name must match that which is in the model. The parameter variable names should be chosen from the following list:

- Free - boolean flag for inclusion in model learning algorithms
- Value - value of the parameter, which is also used as an initial guess for model learning algorithms
- Minimum - minimum value of the parameter for model learning algorithms
- Maximum - maximum value of the parameter for model learning algorithms
- Covariance - covariance of the parameter for model learning algorithms

Classes

Parameter data may be collected using the following classes:

ParameterFromCSV

Collects parameter data from a CSV file. The CSV file rows must be named as the parameter names and the columns must be named as the parameter variable names.

CHAPTER

FIVE

SYSTEMS

CHAPTER

SIX

MODELS

CHAPTER

SEVEN

OPTIMIZATION

CHAPTER

EIGHT

OCCUPANT