MPCPy User Guide

Release 0.1.0

Lawrence Berkeley National Laboratory

Jul 23, 2020

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 General

MPCPy is a python package that facilitates the testing and implementation of occupant-integrated model predictive control (MPC) for building systems. The 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:

- `exodata` classes collect external data and process it for use within MPCPy. This includes data for weather, internal loads, control signals, grid signals, model parameters, optimization constraints, and miscellaneous inputs.
- `system` classes represent real or emulated systems to be controlled, collecting measurements from and providing control inputs to the systems. For example, these include detailed simulations or real data collected for zone thermal response, HVAC performance, or ground-truth occupancy.
- `models` classes represent system models for MPC, managing model simulation, estimation, and validation. For example, these could represent an RC zone thermal response model, simplified HVAC equipment performance models, or occupancy models.
- `optimization` classes formulate and solve the MPC optimization problems using `models` objects.

Three other modules provide additional, mainly internal, functionality to MPCPy:

- `variables` and `units` 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.

1.2 Third-Party Software

While MPCPy provides an integration platform, it relies on free, open-source, third-party software packages for model implementation, simulators, parameter estimation algorithms, and optimization solvers. This includes python packages for scripting and data manipulation as well as other more comprehensive software packages for specific purposes. In particular, modeling and optimization for physical systems rely on the Modelica language specification (https://www.modelica.org/) and FMI standard (http://fmi-standard.org/) in order to leverage model library and tool development on these standards occurring elsewhere within the building and other industries. Two examples of these third-party tools are:

- **JModelica.org** (http://jmodelica.org/) is used for simulation of FMUs, compiling FMUs from Modelica models, parameter estimation of Modelica models, and control optimization using Modelica models.
- **EstimationPy** (http://lbl-srg.github.io/EstimationPy/) is used for implementing the Unscented Kalman Filter for parameter estimation of FMU models.

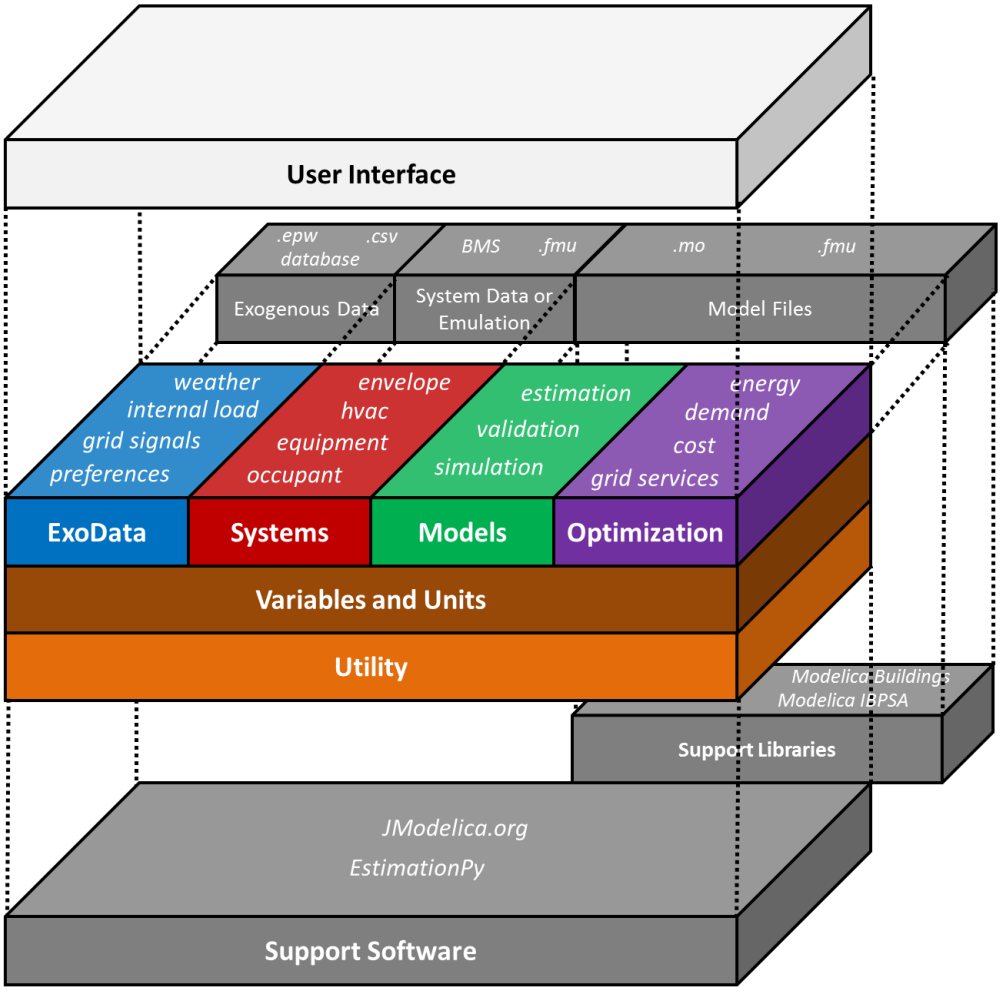

Fig. 1: Software architecture diagram for MPCPy. Note that a user interface has not been developed.

1.3 Contributing

Research has shown that MPC can address emerging control challenges faced by buildings. However, there exists no standard practice or methods for implementing MPC in buildings. Implementation is defined here as model structure, complexity, and training methods, data resolution and amount, optimization problem structure and algorithm, and transfer of optimal control solution to real building control. In fact, different applications likely require different implementations. Therefore, the aim is for MPCPy to be flexible enough to accommodate different and new approaches to MPC in buildings.

If you are interested in contributing to this project, please contact the developers and visit the development site at https://github.com/lbl-srg/MPCPy.

1.4 Cite

To cite MPCPy, please use:

Blum, D. H. and Wetter, M. "MPCPy: An Open-Source Software Platform for Model Predictive Control in Buildings." Proceedings of the 15th Conference of International Building Performance Simulation, Aug 7 – 9, 2017. San Francisco, CA.

CHAPTER

TWO

GETTING STARTED

To get started with MPCPy, first follow the installation instructions below. Then, checkout the introductory tutorial to get a feel for the workflow of MPCPy. You can always consult the user guide for more information.

2.1 Installation Instructions For Linux (Ubuntu 16.04 LTS)

1. Install Python packages:
   - MPCPy uses Python 2.7.
   - using pip, install the following packages:

     matplotlib >= 2.0.2

     numpy >= 1.16.6

     pandas >= 0.20.3

     python-dateutil >= 2.6.1

     pytz >= 2017.2

     scikit-learn >= 0.18.2

     sphinx >= 1.6.3

     numpydoc >= 0.7.0

     tzwhere = 2.3

     pyDOE >= 0.3.8

     netCDF4 == 1.4.2

     cftime == 1.0.4.2

     pvlib == 0.6.0

     siphon == 0.8.0
2. Install libgeos-dev with command:

   ```
   > sudo apt-get install libgeos-dev
   ```
3. Install JModelica 2.0 (for Modelica compiling, optimization, and fmu simulation)
4. Create JModelica environmental variables
   - add the following lines to your bashrc script and replace the "..." with the JModelica install directory:

```
export JMODELICA_HOME=".../JModelica"
export IPOPT_HOME=".../Ipopt-3.12.4-inst"
export SUNDIALS_HOME="$JMODELICA_HOME/ThirdParty/Sundials"
export SEPARATE_PROCESS_JVM="/usr/lib/jvm/java-8-openjdk-amd64/"
export JAVA_HOME="/usr/lib/jvm/java-8-openjdk-amd64/"
```

5. Create the MODELICAPATH environmental variable
   - add the following lines to your bashrc script (assumes 4. above sets JMODELICA_HOME):

```
export MODELICAPATH="$JMODELICA_HOME/ThirdParty/MSL"
```

6. Download or Clone EstimationPy
   - go to https://github.com/lbl-srg/EstimationPy and clone or download repository into a directory (let's call it `.../EstimationPy`).
7. Download or Clone MPCPy
   - go to https://github.com/lbl-srg/MPCPy and clone or download repository into a directory (let's call it `.../MPCPy`).
8. Edit PYTHONPATH environmental variable
   - add the following lines to your bashrc script (assumes 4. above sets JMODELICA_HOME) and replace the "..." with the appropriate directory:

```
export PYTHONPATH=$PYTHONPATH:"$JMODELICA_HOME/Python"
export PYTHONPATH=$PYTHONPATH:"$JMODELICA_HOME/Python/pymodelica"
export PYTHONPATH=$PYTHONPATH:".../EstimationPy"
export PYTHONPATH=$PYTHONPATH:".../MPCPy"
```

9. Test the installation
   - Run the introductory tutorial example. From the command-line, use the commands:

```
> cd doc/userGuide/tutorial
> python introductory.py
```

10. Optional, for developers only
    - to pass all unit tests, the Modelica Buildings Library must also be on the MODELICAPATH. Download the library and add the appropriate directory path to the MODELICAPATH variable.
    - to generate the user guide pdf, latex must be installed. Use the following commands to install texlive and latexmk:

```
> sudo apt-get install texlive
> sudo apt-get install texlive-formats-extra
> sudo apt-get install latexmk
```

2.2 Introductory Tutorial

This tutorial will introduce the basic concepts and workflow of mpcpy. By the end, we will train a simple model based on emulated data, and use the model to optimize the control signal of the system. All required data files for this tutorial are located in doc/userGuide/tutorial.

The model is a simple RC model of zone thermal response to ambient temperature and a singal heat input. It is written in Modelica:

```
model RC "A simple RC network for example purposes"
  Modelica.Blocks.Interfaces.RealInput weaTDryBul(unit="K") "Ambient temperature";
  Modelica.Blocks.Interfaces.RealInput Qflow(unit="W") "Heat input";
  Modelica.Blocks.Interfaces.RealOutput Tzone(unit="K") "Zone temperature";
  Modelica.Thermal.HeatTransfer.Components.HeatCapacitor heatCapacitor(C=1e5)
  "Thermal capacitance of zone";
  Modelica.Thermal.HeatTransfer.Components.ThermalResistor thermalResistor(R=0.01)
  "Thermal resistance of zone";
  Modelica.Thermal.HeatTransfer.Sources.PrescribedTemperature preTemp;
  Modelica.Thermal.HeatTransfer.Sensors.TemperatureSensor senTemp;
  Modelica.Thermal.HeatTransfer.Sources.PrescribedHeatFlow preHeat;
equation
  connect(senTemp.T, Tzone)
  connect(preHeat.Q_flow, Qflow)
  connect(heatCapacitor.port, senTemp.port)
  connect(heatCapacitor.port, preHeat.port)
  connect(preTemp.port, thermalResistor.port_a)
  connect(thermalResistor.port_b, heatCapacitor.port)
  connect(preTemp.T, weaTDryBul)
end RC;
```

2.2.1 Variables and Units

First, lets get familiar with variables and units, the basic building blocks of MPCPy.

```
>>> from mpcpy import variables
>>> from mpcpy import units
```

Static variables contain data that is not a timeseries:

```
>>> setpoint = variables.Static('setpoint', 20, units.degC)
>>> print(setpoint) # doctest: +NORMALIZE_WHITESPACE
Name: setpoint
Variability: Static
Quantity: Temperature
Display Unit: degC
```

The unit assigned to the variable is the display unit. However, each display unit quantity has a base unit that is used to store the data in memory. This makes it easy to convert between units when necessary. For example, the degC display unit has a quantity temperature, which has base unit in Kelvin.

```
>>> # Get the data in display units
>>> setpoint.display_data()
20.0
>>> # Get the data in base units
>>> setpoint.get_base_data()
293.15
>>> # Convert the display unit to degF
>>> setpoint.set_display_unit(units.degF)
>>> setpoint.display_data() # doctest: +NORMALIZE_WHITESPACE
68.0
```

Timeseries variables contain data in the form of a `pandas` Series with a datetime index:

```
>>> # Create pandas Series object
>>> import pandas as pd
>>> data = [0, 5, 10, 15, 20]
>>> index = pd.date_range(start='1/1/2017', periods=len(data), freq='H')
>>> ts = pd.Series(data=data, index=index, name='power_data')
```

Now we can do the same thing with the timeseries variable as we did with the static variable:

```
>>> # Create mpcpy variable
>>> power_data = variables.Timeseries('power_data', ts, units.Btuh)
>>> print(power_data) # doctest: +NORMALIZE_WHITESPACE
Name: power_data
Variability: Timeseries
Quantity: Power
Display Unit: Btuh
>>> # Get the data in display units
>>> power_data.display_data()
2017-01-01 00:00:00+00:00     0.0
2017-01-01 01:00:00+00:00     5.0
2017-01-01 02:00:00+00:00    10.0
2017-01-01 03:00:00+00:00    15.0
2017-01-01 04:00:00+00:00    20.0
Freq: H, Name: power_data, dtype: float64
>>> # Get the data in base units
>>> power_data.get_base_data()
2017-01-01 00:00:00+00:00    0.000000
2017-01-01 01:00:00+00:00    1.465355
2017-01-01 02:00:00+00:00    2.930711
2017-01-01 03:00:00+00:00    4.396066
2017-01-01 04:00:00+00:00    5.861421
Freq: H, Name: power_data, dtype: float64
>>> # Convert the display unit to kW
>>> power_data.set_display_unit(units.kW)
>>> power_data.display_data()
2017-01-01 00:00:00+00:00    0.000000
2017-01-01 01:00:00+00:00    0.001465
2017-01-01 02:00:00+00:00    0.002931
2017-01-01 03:00:00+00:00    0.004396
2017-01-01 04:00:00+00:00    0.005861
Freq: H, Name: power_data, dtype: float64
```

There is additional functionality with the units that may be useful, such as setting new data and getting the units. Consult the documentation on these classes for more information.

2.2.2 Collect model weather and control signal data

Now, we would like to collect the weather data and control signal inputs for our model. We do this using exodata objects:

```
>>> from mpcpy import exodata
```

Let's take our weather data from an EPW file. We instantiate the weather exodata object by supplying the path to the EPW file:

```
>>> weather = exodata.WeatherFromEPW('USA_IL_Chicago-OHare.Intl.AP.725300_TMY3.epw')
```

Note that using the weather exodata object assumes that weather inputs to our model are named a certain way. Consult the documentation on the weather exodata class for more information. In this case, the ambient dry bulb temperature input in our model is named weaTDryBul.

Let's take our control input signal from a CSV file. The CSV file looks like:

```
Time,Qflow_csv
01/01/17 12:00 AM,3000
01/01/17 01:00 AM,3000
01/01/17 02:00 AM,3000
...
01/02/17 10:00 PM,3000
01/02/17 11:00 PM,3000
01/03/17 12:00 AM,3000
```

We instantiate the control exodata object by supplying the path to the CSV file as well as a map of the names of the columns to the input of our model. We also assume that the data in the CSV file is given in the local time of the weather file, and so we supply this optional parameter, tz_name, upon instantiation as well. If no time zone is supplied, it is assumed to be UTC.

```
>>> variable_map = {'Qflow_csv' : ('Qflow', units.W)}
>>> control = exodata.ControlFromCSV('ControlSignal.csv',
...                                  variable_map,
...                                  tz_name = weather.tz_name)
```

Now we are ready to collect the exogenous data from our data sources for a given time period.

```
>>> start_time = '1/1/2017'
>>> final_time = '1/3/2017'
>>> weather.collect_data(start_time, final_time) # doctest: +ELLIPSIS
-etc-
>>> control.collect_data(start_time, final_time)
```

Use the `display_data()` and `get_base_data()` functions for the weather and control objects to get the data in the form of a pandas dataframe. Note that the data is given in UTC time.

```
>>> control.display_data() # doctest: +ELLIPSIS
                            Qflow
Time
2017-01-01 06:00:00+00:00  3000.0
2017-01-01 07:00:00+00:00  3000.0
2017-01-01 08:00:00+00:00  3000.0
-etc-
```

2.2.3 Simulate as Emulated System

The model has parameters for the resistance and capacitance set in the modelica code. For the purposes of this tutorial, we will assume that the model with these parameter values represents the actual system. We now wish to collect measurements from this 'actual system.' For this, we use the systems module of mpcpy.

```
>>> from mpcpy import systems
```

First, we instantiate our system model by supplying a measurement dictionary, information about where the model resides, and information about model exodata.

The measurement dictionary holds information about and data from the variables being measured. We start with defining the variables we are interested in measuring and their sample rate. In this case, we have two, the output of the

model, called 'Tzone' and the control input called 'Qflow'. Note that 'heatCapacitor.T' would also be valid instead of 'Tzone'.

```
>>> measurements = {'Tzone' : {}, 'Qflow' : {}}
>>> measurements['Tzone']['Sample'] = variables.Static('sample_rate_Tzone',
...                                                    3600,
...                                                    units.s)
>>> measurements['Qflow']['Sample'] = variables.Static('sample_rate_Qflow',
...                                                    3600,
...                                                    units.s)
```

The model information is given by a tuple containing the path to the Modelica (.mo) file, the path of the model within the .mo file, and a list of paths of any required libraries other than the Modelica Standard. For this example, there are no additional libraries.

```
>>> moinfo = ('Tutorial.mo', 'Tutorial.RC', {})
```

Ultimately, the modelica model is compiled into an FMU. If the emulation model is already an FMU, than an fmupath can be specified instead of the modelica information tuple. For more information, see the documentation on the systmems class.

We can now instantiate the system emulation object with our measurement dictionary, model information, collected exogenous data, and time zone:

```
>>> emulation = systems.EmulationFromFMU(measurements,
...                                      moinfo = moinfo,
...                                      weather_data = weather.data,
...                                      control_data = control.data,
...                                      tz_name = weather.tz_name)
```

Finally, we can collect the measurements from our emulation over a specified time period and display the results as a pandas dataframe. The `collect_measurements()` function updates the measurement dictionary with timeseries data in the `'Measured'` field for each variable.

```
>>> # Collect the data
>>> emulation.collect_measurements('1/1/2017', '1/2/2017') # doctest: +ELLIPSIS
-etc-
>>> # Display the results
>>> emulation.display_measurements('Measured').applymap('{:.2f}'.format) # doctest:␣
↪+ELLIPSIS
                           Qflow   Tzone
Time
2017-01-01 06:00:00+00:00  3000.00  293.15
2017-01-01 07:00:00+00:00  3000.00  291.01
2017-01-01 08:00:00+00:00  3000.00  291.32
-etc-
```

2.2.4 Estimate Parameters

Now assume that we do not know the parameters of the model. Or, that we have measurements from a real or emulated system, and would like to estimate parameters of our model to fit the measurements. For this, we use the models module from mpcpy.

```
>>> from mpcpy import models
```

In this case, we have a Modelica model with two parameters that we would like to train based on the measured data from our system; the resistance and capacitance.

We first need to collect some information about our parameters and do so using a parameters exodata object. The parameter information is stored in a CSV file that looks like:

```
Name,Free,Value,Minimum,Maximum,Covariance,Unit
heatCapacitor.C,True,40000,1.00E+04,1.00E+06,1000,J/K
thermalResistor.R,True,0.002,0.001,0.1,0.0001,K/W
```

The name is the name of the parameter in the model. The Free field indicates if the parameter is free to be changed during the estimation method or not. The Value is the current value of the parameter. If the parameter is to be estimated, this would be an initial guess. If the parameter's Free field is set to False, then the value is set to the parameter upon simulation. The Minimum and Maximum fields set the minimum and maximum value allowed by the parameter during estimation. The Covariance field sets the covariance of the parameter, and is only used for unscented kalman filtering. Finally, the Unit field specifies the unit of the parameter using the name string of MPCPy unit classes.

```
>>> parameters = exodata.ParameterFromCSV('Parameters.csv')
>>> parameters.collect_data()
>>> parameters.display_data() # doctest: +NORMALIZE_WHITESPACE
                  Covariance  Free Maximum Minimum Unit  Value
Name
heatCapacitor.C         1000  True   1e+06   10000  J/K  40000
thermalResistor.R     0.0001  True     0.1   0.001  K/W  0.002
```

Now, we can instantiate the model object by defining the estimation method, validation method, measurement dictionary, model information, parameter data, and exogenous data. In this case, we use JModelica optimization to perform the parameter estimation and will validate the parameter estimation by calculating the root mean square error (RMSE) between measurements from the model and emulation.

```
>>> model = models.Modelica(models.JModelicaParameter,
...                         models.RMSE,
...                         emulation.measurements,
...                         moinfo = moinfo,
...                         parameter_data = parameters.data,
...                         weather_data = weather.data,
...                         control_data = control.data,
...                         tz_name = weather.tz_name)
```

Let's simulate the model to see how far off we are with our initial parameter guesses. The `simulate()` function updates the measurement dictionary with timeseries data in the `'Simulated'` field for each variable.

```
>>> # Simulate the model
>>> model.simulate('1/1/2017', '1/2/2017') # doctest: +ELLIPSIS
-etc-
>>> # Display the results
>>> model.display_measurements('Simulated').applymap('{:.2f}'.format) # doctest:␣
↪+ELLIPSIS
                            Qflow   Tzone
Time
2017-01-01 06:00:00+00:00  3000.00  293.15
2017-01-01 07:00:00+00:00  3000.00  266.95
2017-01-01 08:00:00+00:00  3000.00  267.44
-etc-
```

Now, we are ready to estimate the parameters to better fit the emulated measurements. In addtion to a training period, we must supply a list of measurement variables for which to minimize the error between the simulated and measured data. In this case, we only have one, `'Tzone'`. The `estimate()` function updates the Value field for the parameter data in the model.

```
>>> model.parameter_estimate('1/1/2017', '1/2/2017', ['Tzone']) # doctest: +ELLIPSIS
-etc-
```

Let's validate the estimation on the training period. The `validate()` method will simulate the model over the specified time period, calculate the RMSE between the simulated and measured data, and generate a plot in the working directory that shows the simulated and measured data for each measurement variable.

```
>>> # Perform validation
>>> model.validate('1/1/2017', '1/2/2017', 'validate_tra', plot=1) # doctest:␣
↪+ELLIPSIS
-etc-
>>> # Get RMSE
>>> print("%.3f" % model.RMSE['Tzone'].display_data()) # doctest: +NORMALIZE_
↪WHITESPACE
0.041
```

Now let's validate on a different period of exogenous data:

```
>>> # Define validation period
>>> start_time_val = '1/2/2017'
>>> final_time_val = '1/3/2017'
>>> # Collect new measurements
>>> emulation.collect_measurements(start_time_val, final_time_val) # doctest:␣
↪+ELLIPSIS
-etc-
>>> # Assign new measurements to model
>>> model.measurements = emulation.measurements
>>> # Perform validation
>>> model.validate(start_time_val, final_time_val, 'validate_val', plot=1) # doctest:␣
↪+ELLIPSIS
-etc-
>>> # Get RMSE
>>> print("%.3f" % model.RMSE['Tzone'].display_data()) # doctest: +NORMALIZE_
↪WHITESPACE
0.047
```

Finally, let's view the estimated parameter values:

```
>>> for key in model.parameter_data.keys():
...     print(key, "%.2f" % model.parameter_data[key]['Value'].display_data())
('heatCapacitor.C', '119828.30')
('thermalResistor.R', '0.01')
```

2.2.5 Optimize Control

We are now ready to optimize control of our system heater using our calibrated MPC model. Specificlaly, we would like to maintain a comfortable temperature in our zone with the minimum amount of heater energy. We can do this by using the optimization module of MPCPy.

```
>>> from mpcpy import optimization
```

First, we need to collect some constraint data to add to our optimization problem. In this case, we will constrain the heating input to between 0 and 4000 W, and the temperature to a comfortable range, between 20 and 25 degC. We collect contraint data from a CSV using a constraint exodata data object. The constraint CSV looks like:

```
Time,Qflow_min,Qflow_max,T_min,T_max
01/01/17 12:00 AM,0,4000,20,25
01/01/17 01:00 AM,0,4000,20,25
01/01/17 02:00 AM,0,4000,20,25
...
01/02/17 10:00 PM,0,4000,20,25
01/02/17 11:00 PM,0,4000,20,25
01/03/17 12:00 AM,0,4000,20,25
```

The constraint exodata object is used to determine which column of data matches with which model variable and whether it is a less-than-or-equal-to (LTE) or greater-than-or-equal-to (GTE) constraint:

```
>>> # Define variable map
>>> variable_map = {'Qflow_min' : ('Qflow', 'GTE', units.W),
...                 'Qflow_max' : ('Qflow', 'LTE', units.W),
...                 'T_min' : ('Tzone', 'GTE', units.degC),
...                 'T_max' : ('Tzone', 'LTE', units.degC)}
>>> # Instantiate constraint exodata object
>>> constraints = exodata.ConstraintFromCSV('Constraints.csv',
...                                         variable_map,
...                                         tz_name = weather.tz_name)
>>> # Collect data
>>> constraints.collect_data('1/1/2017', '1/3/2017')
>>> # Get data
>>> constraints.display_data() # doctest: +ELLIPSIS
                           Qflow_GTE  Qflow_LTE  Tzone_GTE  Tzone_LTE
Time
2017-01-01 06:00:00+00:00        0.0     4000.0       20.0       25.0
2017-01-01 07:00:00+00:00        0.0     4000.0       20.0       25.0
2017-01-01 08:00:00+00:00        0.0     4000.0       20.0       25.0
-etc-
```

We can now instantiate an optimization object using our calibrated MPC model, selecting an optimization problem type and solver package, and specifying which of the variables in the model to treat as the objective variable. In this case, we choose an energy minimization problem (integral of variable over time horizon) to be solved using JModelica, and Qflow to be the variable we wish to minimize the integral of over the time horizon.

```
>>> opt_problem = optimization.Optimization(model,
...                                         optimization.EnergyMin,
...                                         optimization.JModelica,
...                                         'Qflow',
...                                         constraint_data = constraints.data)
```

The information provided is used to automatically generate a .mop (optimization model file for JModelica) and transfer the optimization problem using JModelica. Using the `optimize()` function optimizes the variables defined in the control data of the model object and updates their timeseries data with the optimal solution for the time period specified. Note that other than the constraints, the exogenous data within the model object is used, and the control interval is assumed to be the same as the measurement sampling rate of the model. Use the `get_optimization_options()` and `set_optimization_options()` to see and change the options for the optimization solver; for instance number of control points, maximum iteration number, tolerance, or maximum CPU time. See the documentation for these functions for more information.

```
>>> opt_problem.optimize('1/2/2017', '1/3/2017') # doctest: +ELLIPSIS
-etc-
```

We can get the optimization solver statistics in the form of (return message, # of iterations, objective value, solution time in seconds):

```
>>> opt_problem.get_optimization_statistics() # doctest: +ELLIPSIS
('Solve_Succeeded', 12, -etc-)
```

We can retrieve the optimal control solution and verify that the constraints were satisfied. The intermediate points are a result of the direct collocation method used by JModelica.

```
>>> opt_problem.display_measurements('Simulated').applymap('{:.2f}'.format) #␣
↪doctest: +ELLIPSIS
                                    Qflow   Tzone
Time
2017-01-02 06:00:00+00:00          669.93  298.15
2017-01-02 06:09:18.183693+00:00  1512.95  293.15
2017-01-02 06:38:41.816307+00:00  2599.01  293.15
2017-01-02 07:00:00+00:00         1888.28  293.15
-etc-
```

Finally, we can simulate the model using the optimized control trajectory. Note that the `model.control_data` dictionary is updated by the `opt_problem.optimize()` function.

```
>>> model.control_data['Qflow'].display_data().loc[pd.to_datetime('1/2/2017  06:00:00
↪'):pd.to_datetime('1/3/2017 06:00:00')].map('{:.2f}'.format) # doctest: +ELLIPSIS
2017-01-02 06:00:00+00:00            669.93
2017-01-02 06:09:18.183693+00:00    1512.95
2017-01-02 06:38:41.816307+00:00    2599.01
2017-01-02 07:00:00+00:00           1888.28
-etc-
>>> model.simulate('1/2/2017', '1/3/2017') # doctest: +ELLIPSIS
-etc-
>>> model.display_measurements('Simulated').applymap('{:.2f}'.format) # doctest:␣
↪+ELLIPSIS
                              Qflow   Tzone
Time
2017-01-02 06:00:00+00:00   669.93  293.15
2017-01-02 07:00:00+00:00  1888.28  291.41
2017-01-02 08:00:00+00:00  2277.67  293.03
-etc-
```

Note there is some mismatch between the simulated model output temperature and the raw optimal control solution model output temperature output. This is due to the interpolation of control input results during simulation not aligning with the collocation polynomials and timestep determined by the optimization solver. We can solve the optimization problem again, this time updating the `model.control_data` with a greater time resolution of 1 second. Some mismatch will still occur due to the optimization solution using collocation being an approximation of the true dynamic model.

```
>>> opt_problem.optimize('1/2/2017', '1/3/2017', res_control_step=1.0) # doctest:␣
↪+ELLIPSIS
-etc-
>>> model.control_data['Qflow'].display_data().loc[pd.to_datetime('1/2/2017 06:00:00
↪'):pd.to_datetime('1/3/2017 06:00:00')].map('{:.2f}'.format) # doctest: +ELLIPSIS
2017-01-02 06:00:00+00:00     669.93
2017-01-02 06:00:01+00:00     671.66
2017-01-02 06:00:02+00:00     673.38
-etc-
>>> model.simulate('1/2/2017', '1/3/2017') # doctest: +ELLIPSIS
-etc-
>>> model.display_measurements('Simulated').applymap('{:.2f}'.format) # doctest:␣
↪+ELLIPSIS
```

(continues on next page)

(continued from previous page)

```
                             Qflow   Tzone
Time
2017-01-02 06:00:00+00:00   669.93  293.15
2017-01-02 07:00:00+00:00  1888.28  292.67
2017-01-02 08:00:00+00:00  2277.67  293.13
-etc-
```

2.3 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:

```
> python bin/runUnitTests
```

To run only unit tests in the module test_models from command-line, use the command:

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

To run only unit tests in the class SimpleRC from the module test_models from the command-line, use the command:

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

CHAPTER

THREE

VARIABLES AND UNITS

`variables` classes together with `units` classes form 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

Single 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.

3.1 Classes

class mpcpy.variables.**Static**(*name*, *data*, *display_unit*)

Variable class with data that is not a timeseries.

Parameters

name [string] Name of variable.

data [float, int, bool, list, `numpy` array] Data of variable

display_unit [mpcpy.units.unit] Unit of variable data being set.

Attributes

name [string] Name of variable.

data [float, int, bool, list, `numpy` array] Data of variable

display_unit [mpcpy.units.unit] Unit of variable data when returned with `display_data()`.

quantity_name [string] Quantity type of the variable (e.g. Temperature, Power, etc.).

variability [string] Static.

display_data(***kwargs*)
Return the data of the variable in display units.

Parameters

geography [list, optional] Latitude [0] and longitude [1] in degrees. Will return timeseries index in specified timezone.

tz_name [string, optional] Time zone name according to `tzwhere` package. Will return timeseries index in specified timezone.

Returns

data [data object] Data object of the variable in display units.

get_base_data()
Return the data of the variable in base units.

Returns

data [data object] Data object of the variable in base units.

get_base_unit()
Returns the base unit of the variable.

Returns

base_unit [mpcpy.units.unit] Base unit of variable.

get_base_unit_name()
Returns the base unit name of the variable.

Returns

base_unit_name [string] Base unit name of variable.

get_display_unit()
Returns the display unit of the variable.

Returns

display_unit [mpcpy.units.unit] Display unit of variable.

get_display_unit_name()
Returns the display unit name of the variable.

Returns

display_unit_name [string] Display unit name of variable.

set_data(*data*)
Set data of Static variable.

Parameters

data [float, int, bool, list, `numpy` array] Data to be set for variable.

Yields

data [float, int, bool, list, `numpy` array] Data attribute.

set_display_unit(*display_unit*)
Set the display unit of the variable.

Parameters

display_unit [mpcpy.units.unit] Display unit to set.

class mpcpy.variables.**Timeseries**(*name*, *timeseries*, *display_unit*, *tz_name='UTC'*, ***kwargs*)

Variable class with data that is a timeseries.

Parameters

name [string] Name of variable.

timeseries [pandas Series] Timeseries data of variable. Must have an index of timestamps.

display_unit [mpcpy.units.unit] Unit of variable data being set.

tz_name [string] Timezone name according to tzwhere.

geography [list, optional] List specifying [latitude, longitude] in degrees.

cleaning_type [dict, optional] Dictionary specifying {'cleaning_type' : mpcpy.variables.Timeseries.cleaning_type, 'cleaning_args' : cleaning_args}.

Attributes

name [string] Name of variable.

data [float, int, bool, list, numpy array] Data of variable

display_unit [mpcpy.units.unit] Unit of variable data when returned with display_data().

quantity_name [string] Quantity type of the variable (e.g. Temperature, Power, etc.).

variability [string] Timeseries.

cleaning_replace()

Cleaning method to replace values within timeseries.

Parameters

to_replace Value to replace.

replace_with Replacement value.

Returns

timeseries Timeseries with data replaced according to to_replace and replace_with.

display_data(***kwargs*)

Return the data of the variable in display units.

Parameters

geography [list, optional] Latitude [0] and longitude [1] in degrees. Will return timeseries index in specified timezone.

tz_name [string, optional] Time zone name according to tzwhere package. Will return timeseries index in specified timezone.

Returns

data [data object] Data object of the variable in display units.

get_base_data()

Return the data of the variable in base units.

Returns

data [data object] Data object of the variable in base units.

get_base_unit()

Returns the base unit of the variable.

Returns

base_unit [mpcpy.units.unit] Base unit of variable.

get_base_unit_name()
Returns the base unit name of the variable.

Returns

base_unit_name [string] Base unit name of variable.

get_display_unit()
Returns the display unit of the variable.

Returns

display_unit [mpcpy.units.unit] Display unit of variable.

get_display_unit_name()
Returns the display unit name of the variable.

Returns

display_unit_name [string] Display unit name of variable.

set_data(*timeseries*, *tz_name='UTC'*, ***kwargs*)
Set data of Timeseries variable.

Parameters

data [`pandas` Series] Timeseries data of variable. Must have an index of timestamps.

tz_name [string] Timezone name according to `tzwhere`.

geography [list, optional] List specifying [latitude, longitude] in degrees.

cleaning_type [dict, optional] Dictionary specifying {'cleaning_type' : mpcpy.variables.Timeseries.cleaning_type, 'cleaning_args' : cleaning_args}.

Yields

data [`pandas` Series] Data attribute.

set_display_unit(*display_unit*)
Set the display unit of the variable.

Parameters

display_unit [mpcpy.units.unit] Display unit to set.

CHAPTER

FOUR

EXODATA

`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.

4.1 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. 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 list found in 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}
```

4.1.1 Classes

class mpcpy.exodata.**WeatherFromEPW**(*epw_file_path*, *standard_time=False*)

Collects weather data from an EPW file.

Parameters

epw_file_path [string] Path of epw file.

standard_time [boolean] False to localize data timestamps to EPW file location. True to treat data timestamps in standard time. Default is False.

Attributes

data [dictionary] {"Weather Variable Name" : mpcpy.Variables.Timeseries}.

lat [mpcpy.variables.Static] Latitude in degrees.

lon [mpcpy.variables.Static] Longitude in degrees.

tz_name [string] Timezone name.

file_path [string] Path of epw file.

calculate_solar_radiation(*method='Zhang-Huang'*)

Calculate the global solar horizontal irradiation with already-collected variables.

This function adds the 'weaHGloHor' variable to the data dictionary in W/m^2.

The available method is the 'Zhang-Huang': Zhang-Huang Solar Model. Reference to the ZH model: https://www.energyplus.net/sites/default/files/docs/site_v8.3.0/EngineeringReference/05-Climate/index.html#zhang-huang-solar-model Original paper: https://pdfs.semanticscholar.org/7b8e/7ea72db78f99939ce2d7c2890dacfcb0dc5a.pdf For this method, the data dictionary variables needed to calculate the solar radiation are:

- weaSolAlt : solar altitude angle
- weaNTot : cloud cover
- weaRelHum : relative humidity
- weaWinSpe : wind speed

The `collect_data()` method should be used before calling this method.

Parameters

method [str, optional] Method of calculating the solar irradiation. Only one option exists, 'Zhang-Huang'. Default is 'Zhang-Huang'.

Returns

None

collect_data(*start_time*, *final_time*)
Collect data from specified source and update data attribute.

Parameters

start_time [string] Start time of data collection.

final_time [string] Final time of data collection.

Yields

data [dictionary] Data attribute.

display_data()
Get data in display units as pandas dataframe.

Returns

df [pandas dataframe] Timeseries dataframe in display units.

get_base_data()
Get data in base units as pandas dataframe.

Returns

df [pandas dataframe] Timeseries dataframe in base units.

class mpcpy.exodata.**WeatherFromCSV**(*csv_file_path*, *variable_map*, *geography*, ***kwargs*)
Collects weather data from a csv file.

Parameters

csv_file_path [string] Path of csv file.

variable_map [dictionary] {"Column Header Name" : ("Weather Variable Name", mpcpy.Units.unit)}.

geography [[numeric, numeric]] List of [Latitude, Longitude] in degrees.

Attributes

data [dictionary] {"Weather Variable Name" : mpcpy.Variables.Timeseries}.

lat [mpcpy.variables.Static] Latitude in degrees.

lon [mpcpy.variables.Static] Longitude in degrees.

tz_name [string] Timezone name.

file_path [string] Path of csv file.

calculate_solar_radiation(*method='Zhang-Huang'*)
Calculate the global solar horizontal irradiation with already-collected variables.

This function adds the 'weaHGloHor' variable to the data dictionary in W/m^2.

The available method is the 'Zhang-Huang': Zhang-Huang Solar Model. Reference to the ZH model: https://www.energyplus.net/sites/default/files/docs/site_v8.3.0/EngineeringReference/05-Climate/index.html#zhang-huang-solar-model Original paper: https://pdfs.semanticscholar.org/7b8e/7ea72db78f99939ce2d7c2890dacfcb0dc5a.pdf For this method, the data dictionary variables needed to calculate the solar radiation are:

- weaSolAlt : solar altitude angle
- weaNTot : cloud cover
- weaRelHum : relative humidity
- weaWinSpe : wind speed

The `collect_data()` method should be used before calling this method.

Parameters

method [str, optional] Method of calculating the solar irradiation. Only one option exists, 'Zhang-Huang'. Default is 'Zhang-Huang'.

Returns

None

collect_data(*start_time*, *final_time*)
Collect data from specified source and update data attribute.

Parameters

start_time [string] Start time of data collection.

final_time [string] Final time of data collection.

Yields

data [dictionary] Data attribute.

display_data()
Get data in display units as pandas dataframe.

Returns

df [`pandas` dataframe] Timeseries dataframe in display units.

get_base_data()
Get data in base units as pandas dataframe.

Returns

df [`pandas` dataframe] Timeseries dataframe in base units.

class mpcpy.exodata.**WeatherFromDF**(*df*, *variable_map*, *geography*, ***kwargs*)
Collects weather data from a pandas DataFrame object.

Parameters

df [pandas DataFrame] DataFrame of data. The index must be a datetime object.

variable_map [dictionary] {"Column Header Name" : ("Weather Variable Name", mpcpy.Units.unit)}.

geography [[numeric, numeric]] List of [Latitude, Longitude] in degrees.

Attributes

data [dictionary] {"Weather Variable Name" : mpcpy.Variables.Timeseries}.

lat [mpcpy.variables.Static] Latitude in degrees.

lon [mpcpy.variables.Static] Longitude in degrees.

tz_name [string] Timezone name.

calculate_solar_radiation(*method='Zhang-Huang'*)
Calculate the global solar horizontal irradiation with already-collected variables.

This function adds the 'weaHGloHor' variable to the data dictionary in W/m^2.

The available method is the 'Zhang-Huang': Zhang-Huang Solar Model. Reference to the ZH model: https://www.energyplus.net/sites/default/files/docs/site_v8.3.0/EngineeringReference/05-Climate/index.html#zhang-huang-solar-model Original paper: https://pdfs.semanticscholar.org/7b8e/7ea72db78f99939ce2d7c2890dacfcb0dc5a.pdf For this method, the data dictionary variables needed to calculate the solar radiation are:

- weaSolAlt : solar altitude angle
- weaNTot : cloud cover
- weaRelHum : relative humidity
- weaWinSpe : wind speed

The `collect_data()` method should be used before calling this method.

Parameters

method [str, optional] Method of calculating the solar irradiation. Only one option exists, 'Zhang-Huang'. Default is 'Zhang-Huang'.

Returns

None

collect_data(*start_time*, *final_time*)
Collect data from specified source and update data attribute.

Parameters

start_time [string] Start time of data collection.

final_time [string] Final time of data collection.

Yields

data [dictionary] Data attribute.

display_data()
Get data in display units as pandas dataframe.

Returns

df [`pandas` dataframe] Timeseries dataframe in display units.

get_base_data()
Get data in base units as pandas dataframe.

Returns

df [`pandas` dataframe] Timeseries dataframe in base units.

class `mpcpy.exodata.`**WeatherFromNOAA**(*geography*, *method*, ***kwargs*)
Collects weather data from NOAA.

It can either be historical or predicted weather data, depends on the start_time and final_time. Based on the weather forecast function of pvlib version 6.0, https://pvlib-python.readthedocs.io/en/v0.6.0/

Parameters

geography: [numeric, numeric] List of [Latitude, Longitude] in degrees. The timezone will be inferred automatically from the input geography. When speficying the period for data collection, ONLY local time is allowed.

method: string Weather forecast model. Options are:

'GFS': Global Forecast System model, available for the entire globe and for 7 days ahead, supports historical data, updated every 6 hours, time resolution: 3 hours, geographical resolution: 0.25 and 0.5 deg

'HRRR': High Resolution Rapid Refresh model, available for U.S. and for ~15 hours ahead, DOES NOT support historical data, updated every hour, time resolution: 1 hour, geographical resolution: 3 km

'RAP': Rapid Refresh model, available for the U.S. and for 18 hours, supports historical data, updated every hour, time resolution: 1 hour, geographical resolution: 20, 40 km

'NAM': North American Mesoscale model, available for the whole North America and for 3 days ahead, supports historical data, updated every 6 hours, time resolution: 1 hour, geographical resolution: 20 km

Attributes

data [dictionary] {"Weather Variable Name" : mpcpy.Variables.Timeseries}.

lat [mpcpy.variables.Static] Latitude in degrees.

lon [mpcpy.variables.Static] Longitude in degrees.

tz_name [string] Timezone name.

calculate_solar_radiation(*method='Zhang-Huang'*)

Calculate the global solar horizontal irradiation with already-collected variables.

This function adds the 'weaHGloHor' variable to the data dictionary in W/m^2.

The available method is the 'Zhang-Huang': Zhang-Huang Solar Model. Reference to the ZH model: https://www.energyplus.net/sites/default/files/docs/site_v8.3.0/EngineeringReference/05-Climate/index.html#zhang-huang-solar-model Original paper: https://pdfs.semanticscholar.org/7b8e/7ea72db78f99939ce2d7c2890dacfcb0dc5a.pdf For this method, the data dictionary variables needed to calculate the solar radiation are:

- weaSolAlt : solar altitude angle
- weaNTot : cloud cover
- weaRelHum : relative humidity
- weaWinSpe : wind speed

The `collect_data()` method should be used before calling this method.

Parameters

method [str, optional] Method of calculating the solar irradiation. Only one option exists, 'Zhang-Huang'. Default is 'Zhang-Huang'.

Returns

None

collect_data(*start_time*, *final_time*)

Collect data from specified source and update data attribute.

Parameters

start_time [string] Start time of data collection.

final_time [string] Final time of data collection.

Yields

data [dictionary] Data attribute.

display_data()
Get data in display units as pandas dataframe.

Returns

df [pandas dataframe] Timeseries dataframe in display units.

get_base_data()
Get data in base units as pandas dataframe.

Returns

df [pandas dataframe] Timeseries dataframe in base units.

4.2 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. 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 input names 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`.

4.2.1 Classes

class mpcpy.exodata.**InternalFromCSV**(*csv_file_path*, *variable_map*, ***kwargs*)
Collects internal data from a csv file.

Parameters

csv_file_path [string] Path of csv file.

variable_map [dictionary] {"Column Header Name" : ("Zone Name", "Internal Variable Name", mpcpy.Units.unit)}.

Attributes

data [dictionary] {"Zone Name" : {"Internal Variable Name" : mpcpy.Variables.Timeseries}}.

lat [mpcpy.variables.Static] Latitude in degrees. For timezone.

lon [mpcpy.variables.Static] Longitude in degrees. For timezone.

tz_name [string] Timezone name.

file_path [string] Path of csv file.

collect_data(*start_time*, *final_time*)
Collect data from specified source and update data attribute.

Parameters

start_time [string] Start time of data collection.

final_time [string] Final time of data collection.

Yields

data [dictionary] Data attribute.

display_data()
Get data in display units as pandas dataframe.

Returns

df [pandas dataframe] Timeseries dataframe in display units.

get_base_data()
Get data in base units as pandas dataframe.

Returns

df [pandas dataframe] Timeseries dataframe in base units.

class mpcpy.exodata.**InternalFromOccupancyModel**(*zone_list*, *load_list*, *unit*, *occupancy_model_list*, ***kwargs*)
Collects internal data from an occupancy model.

Parameters

zone_list [[string]] List of zones.

load_list [[[numeric, numeric, numeric]]] List of load per person lists for [convective, radiative, latent] corresponding to zone_list.

unit [mpcpy.Units.unit] Unit of loads.

occupancy_model_list [[mpcpy.Models.Occupancy]] List of occupancy model objects corresponding to zone_list.

Attributes

data [dictionary] {"Zone Name" : {"Internal Variable Name" : mpcpy.Variables.Timeseries}}.

lat [mpcpy.variables.Static] Latitude in degrees. For timezone.

lon [mpcpy.variables.Static] Longitude in degrees. For timezone.

tz_name [string] Timezone name.

collect_data(*start_time*, *final_time*)
Collect data from specified source and update data attribute.

Parameters

start_time [string] Start time of data collection.

final_time [string] Final time of data collection.

Yields

data [dictionary] Data attribute.

display_data()
Get data in display units as pandas dataframe.

Returns

df [pandas dataframe] Timeseries dataframe in display units.

get_base_data()
Get data in base units as pandas dataframe.

Returns

df [pandas dataframe] Timeseries dataframe in base units.

4.3 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. 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.

4.3.1 Classes

class mpcpy.exodata.**ControlFromCSV**(*csv_file_path*, *variable_map*, ***kwargs*)
Collects control data from a csv file.

Parameters

csv_file_path [string] Path of csv file.

variable_map [dictionary] {"Column Header Name" : ("Control Variable Name", mpcpy.Units.unit)}.

Attributes

data [dictionary] {"Control Variable Name" : mpcpy.Variables.Timeseries}.

lat [mpcpy.variables.Static] Latitude in degrees. For timezone.

lon [mpcpy.variables.Static] Longitude in degrees. For timezone.

tz_name [string] Timezone name.

file_path [string] Path of csv file.

collect_data(*start_time*, *final_time*)
Collect data from specified source and update data attribute.

Parameters

start_time [string] Start time of data collection.

final_time [string] Final time of data collection.

Yields

data [dictionary] Data attribute.

display_data()
Get data in display units as pandas dataframe.

Returns

df [pandas dataframe] Timeseries dataframe in display units.

get_base_data()
Get data in base units as pandas dataframe.

Returns

df [pandas dataframe] Timeseries dataframe in base units.

class mpcpy.exodata.**ControlFromDF**(*df*, *variable_map*, ***kwargs*)
Collects control data from a pandas DataFrame object.

Parameters

df [pandas DataFrame] DataFrame of data. The index must be a datetime object.

variable_map [dictionary] {"Column Header Name" : ("Control Variable Name", mpcpy.Units.unit)}.

Attributes

data [dictionary] {"Control Variable Name" : mpcpy.Variables.Timeseries}.

lat [mpcpy.variables.Static] Latitude in degrees.

lon [mpcpy.variables.Static] Longitude in degrees.

tz_name [string] Timezone name.

collect_data(*start_time*, *final_time*)
Collect data from specified source and update data attribute.

Parameters

start_time [string] Start time of data collection.

final_time [string] Final time of data collection.

Yields

data [dictionary] Data attribute.

display_data()
Get data in display units as pandas dataframe.

Returns

df [pandas dataframe] Timeseries dataframe in display units.

get_base_data()
Get data in base units as pandas dataframe.

Returns

df [pandas dataframe] Timeseries dataframe in base units.

4.4 Other Input

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. 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.

4.4.1 Classes

class mpcpy.exodata.**OtherInputFromCSV**(*csv_file_path*, *variable_map*, ***kwargs*)
Collects other input data from a CSV file.

Parameters

csv_file_path [string] Path of csv file.

variable_map [dictionary] {"Column Header Name" : ("Other Input Variable Name", mpcpy.Units.unit)}.

Attributes

data [dictionary] {"Other Input Variable Name" : mpcpy.Variables.Timeseries}.

lat [mpcpy.variables.Static] Latitude in degrees. For timezone.

lon [mpcpy.variables.Static] Longitude in degrees. For timezone.

tz_name [string] Timezone name.

file_path [string] Path of csv file.

collect_data(*start_time*, *final_time*)
Collect data from specified source and update data attribute.

Parameters

start_time [string] Start time of data collection.

final_time [string] Final time of data collection.

Yields

data [dictionary] Data attribute.

display_data()
Get data in display units as pandas dataframe.

Returns

df [pandas dataframe] Timeseries dataframe in display units.

get_base_data()
Get data in base units as pandas dataframe.

Returns

df [pandas dataframe] Timeseries dataframe in base units.

class mpcpy.exodata.**OtherInputFromDF**(*df*, *variable_map*, ***kwargs*)
Collects other input data from a pandas DataFrame object.

Parameters

df [pandas DataFrame] DataFrame of data. The index must be a datetime object.

variable_map [dictionary] {"Column Header Name" : ("Other Input Variable Name", mpcpy.Units.unit)}.

Attributes

data [dictionary] {"Other Input Variable Name" : mpcpy.Variables.Timeseries}.

lat [mpcpy.variables.Static] Latitude in degrees.

lon [mpcpy.variables.Static] Longitude in degrees.

tz_name [string] Timezone name.

collect_data(*start_time*, *final_time*)
Collect data from specified source and update data attribute.

Parameters

start_time [string] Start time of data collection.

final_time [string] Final time of data collection.

Yields

data [dictionary] Data attribute.

display_data()
Get data in display units as pandas dataframe.

Returns

df [pandas dataframe] Timeseries dataframe in display units.

get_base_data()
Get data in base units as pandas dataframe.

Returns

df [pandas dataframe] Timeseries dataframe in base units.

4.5 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. For demand cost minimization, price exodata may also contain information regarding an estimated peak demand that has already been acheived or projected to be acheived. 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
- pi_d - electrical demand price for multi-period
- P_est - estimated peak power consumption for multi-period
- pi_d_c - electrical demand price for coincedent. Must be constant for all time.
- P_est_c - estimated peak power consumption for coincedent. Must be constant for all time.

4.5.1 Classes

class mpcpy.exodata.**PriceFromCSV**(*csv_file_path*, *variable_map*, ***kwargs*)
Collects price data from a csv file.

Parameters

csv_file_path [string] Path of csv file.

variable_map [dictionary] {"Column Header Name" : ("Price Variable Name", mpcpy.Units.unit)}.

Attributes

data [dictionary] {"Price Variable Name" : mpcpy.Variables.Timeseries}.

lat [mpcpy.variables.Static] Latitude in degrees. For timezone.

lon [mpcpy.variables.Static] Longitude in degrees. For timezone.

tz_name [string] Timezone name.

file_path [string] Path of csv file.

collect_data(*start_time*, *final_time*)
Collect data from specified source and update data attribute.

Parameters

start_time [string] Start time of data collection.

final_time [string] Final time of data collection.

Yields

data [dictionary] Data attribute.

display_data()
Get data in display units as pandas dataframe.

Returns

df [pandas dataframe] Timeseries dataframe in display units.

get_base_data()
Get data in base units as pandas dataframe.

Returns

df [pandas dataframe] Timeseries dataframe in base units.

class mpcpy.exodata.**PriceFromDF**(*df*, *variable_map*, ***kwargs*)
Collects price data from a pandas DataFrame object.

Parameters

df [pandas DataFrame object] DataFrame of data. The index must be a datetime object.

variable_map [dictionary] {"Column Header Name" : ("Price Variable Name", mpcpy.Units.unit)}.

Attributes

data [dictionary] {"Price Variable Name" : mpcpy.Variables.Timeseries}.

lat [mpcpy.variables.Static] Latitude in degrees. For timezone.

lon [mpcpy.variables.Static] Longitude in degrees. For timezone.

tz_name [string] Timezone name.

file_path [string] Path of csv file.

collect_data(*start_time*, *final_time*)
Collect data from specified source and update data attribute.

Parameters

start_time [string] Start time of data collection.

final_time [string] Final time of data collection.

Yields

data [dictionary] Data attribute.

display_data()
Get data in display units as pandas dataframe.

Returns

df [pandas dataframe] Timeseries dataframe in display units.

get_base_data()
Get data in base units as pandas dataframe.

Returns

df [pandas dataframe] Timeseries dataframe in base units.

4.6 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 as follows. Exogenous constraint data has the following organization:

‘‘**constraints.data = {“State or Control Variable Name”** [{“Constraint Variable Type”][{“Value”][mpcpy.Variables.Timeseries/Static
“Weight” : mpcpy.Variables.Static or None}}‘‘

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

- LTE - less than or equal to (Timeseries)
- sLTE - less than or equal to with slack variable (Timeseries). This means that the constraint is implemented by adding a slack variable s to the LTE constraint, the objective function with weight w, and set of variables to be optimized as follows, where J is the specified objective variable in the optimization:

$$
\begin{aligned}
&min_{u(t),s(t)} J(x(t), u(t)) + w * s(t)^2 \\
&s.t. \\
&x(t) - s(t) \leq b(t)
\end{aligned}
$$

- GTE - greater than or equal to (Timeseries)
- sGTE - greater than or equal to with slack variable (Timeseries). This means that the constraint is implemented by adding a slack variable s to the GTE constraint, the objective function with weight w, and set of variables to be optimized as follows, where J is the specified objective variable in the optimization:

$$
\begin{aligned}
&min_{u(t),s(t)} J(x(t), u(t)) + w * s(t)^2 \\
&s.t. \\
&x(t) + s(t) \geq b(t)
\end{aligned}
$$

- E - equal to (Timeseries)
- Initial - initial value (Static)
- Final - final value (Static)

- Cyclic - initial value equals final value (Static - Boolean)

Note that the "Weight" is only used for sLTE and sGTE.

4.6.1 Classes

class mpcpy.exodata.**ConstraintFromCSV**(*csv_file_path*, *variable_map*, ***kwargs*)

Collects constraint data from a csv file.

Parameters

csv_file_path [string] Path of csv file.

variable_map [dictionary] {"Column Header Name" : ("State or Control Variable Name", "Constraint Variable Type", mpcpy.Units.unit, <weight>[optional])} Note that <weight> is float or int and is only needed if "Constraint Variable Type" is 'sLTE' or 'sGTE'.

Attributes

data [dictionary]

{"State or Control Variable Name" [{"Constraint Variable Type"][{"Value"][mpcpy.Variables.Timeseries/Static,] "Weight" : mpcpy.Variables.Static or None}}}''

lat [mpcpy.variables.Static] Latitude in degrees. For timezone.

lon [mpcpy.variables.Static] Longitude in degrees. For timezone.

tz_name [string] Timezone name.

file_path [string] Path of csv file.

collect_data(*start_time*, *final_time*)

Collect data from specified source and update data attribute.

Parameters

start_time [string] Start time of data collection.

final_time [string] Final time of data collection.

Yields

data [dictionary] Data attribute.

display_data()

Get data in display units as pandas dataframe.

Returns

df [pandas dataframe] Timeseries dataframe in display units.

get_base_data()

Get data in base units as pandas dataframe.

Returns

df [pandas dataframe] Timeseries dataframe in base units.

class mpcpy.exodata.**ConstraintFromDF**(*df*, *variable_map*, ***kwargs*)

Collects constraint data from a pandas DataFrame object.

Parameters

df [pandas DataFrame object] DataFrame of data. The index must be a datetime object.

variable_map [dictionary] {"Column Header Name" : ("State or Control Variable Name", "Constraint Variable Type", mpcpy.Units.unit, <weight>[optional])} Note that <weight> is float or int and is only needed if "Constraint Variable Type" is 'sLTE' or 'sGTE'.

Attributes

data [dictionary]

{"State or Control Variable Name" [{"Constraint Variable Type"][{"Value"][mpcpy.Variables.Timeseries/Static,] "Weight" : mpcpy.Variables.Static or None}}}''

lat [mpcpy.variables.Static] Latitude in degrees. For timezone.

lon [mpcpy.variables.Static] Longitude in degrees. For timezone.

tz_name [string] Timezone name.

file_path [string] Path of csv file.

collect_data(*start_time*, *final_time*)

Collect data from specified source and update data attribute.

Parameters

start_time [string] Start time of data collection.

final_time [string] Final time of data collection.

Yields

data [dictionary] Data attribute.

display_data()

Get data in display units as pandas dataframe.

Returns

df [pandas dataframe] Timeseries dataframe in display units.

get_base_data()

Get data in base units as pandas dataframe.

Returns

df [pandas dataframe] Timeseries dataframe in base units.

class mpcpy.exodata.**ConstraintFromOccupancyModel**(*state_variable_list*, *values_list*, *constraint_type_list*, *unit_list*, *occupancy_model*, ***kwargs*)

Collects constraint data from an occupancy model.

Parameters

state_variable_list [[string]] List of variable names to be constrained. States with multiple constraints should be listed once for each constraint type.

values_list [[[numeric or boolean, numeric or boolean]]] List of values for [Occupied, Unoccupied] corresponding to state_variable_list.

constraint_type_list [[string]] List of contraint variable types corresponding to state_variable_list.

unit_list [[mpcpy.Units.unit]] List of units corresponding to each contraint type in constraint_type_list.

occupancy_model [mpcpy.Models.Occupancy] Occupancy model object to use.

Attributes

data [dictionary] {"State or Control Variable Name" : {"Constraint Variable Type" : mpcpy.Variables.Timeseries/Static}}.

lat [mpcpy.variables.Static] Latitude in degrees. For timezone.

lon [mpcpy.variables.Static] Longitude in degrees. For timezone.

tz_name [string] Timezone name.

collect_data(*start_time*, *final_time*)
Collect data from specified source and update data attribute.

Parameters

start_time [string] Start time of data collection.

final_time [string] Final time of data collection.

Yields

data [dictionary] Data attribute.

display_data()
Get data in display units as pandas dataframe.

Returns

df [`pandas` dataframe] Timeseries dataframe in display units.

get_base_data()
Get data in base units as pandas dataframe.

Returns

df [`pandas` dataframe] Timeseries dataframe in base units.

4.7 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. Exogenous parameter data has the following organization:

{"Parameter Name" : {"Parameter Key Name" : mpcpy.Variables.Static}}

The parameter name must match that which is in the model. The parameter key 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
- Unit - unit string of parameter

4.7.1 Classes

class mpcpy.exodata.**ParameterFromCSV**(*csv_file_path*)

Collects parameter data from a csv file.

Parameters

csv_file_path [string] Path of csv file. The csv file rows must be named as the parameter names and the columns must be named as the parameter key names.

Attributes

data [dictionary] {"Parameter Name" : {"Parameter Key Name" : mpcpy.Variables.Static}}.

file_path [string] Path of csv file.

append_data(*name*, *value*, *free*, *minimum*, *maximum*, *covariance*, *unit*)

Append a new parameter to existing parameters.

Parameters

name [str] Name of parameter.

value [float | int] Value for the parameter.

free [boolean] True if parameter is free for parameter estimation.

minimum [float | int] Minimum for the parameter.

maximum [float | int] Maximum for the parameter.

covariance [float | int] Covariance for the parameter.

unit [mpcpy Units object] Unit of parameter.

collect_data()

Collect parameter data from csv file into data dictionary.

Yields

data [dictionary] Data attribute.

display_data()

Get data as pandas dataframe in display units.

Returns

df [pandas dataframe] Dataframe in display units.

get_base_data()

Get data as pandas dataframe in base units.

Returns

df [pandas dataframe] Dataframe in base units.

set_data(*name*, *value=None*, *free=None*, *minimum=None*, *maximum=None*, *covariance=None*, *new_name=None*)

Set new data for existing parameter.

All data must be in display units of parameter. No changes are made to arguments that are None.

Parameters

name [str] Name of parameter for which to set data.

value [float | int, optional] Set a new value for the parameter. Default is None.

free [boolean, optional] True if parameter is free for parameter estimation. Default is None.

minimum [float | int, optional] Set a new minimum for the parameter. Default is None.

maximum [float | int, optional] Set a new maximum for the parameter. Default is None.

covariance [float | int, optional] Set a new covariance for the parameter. Default is None.

new_name [str, optional] Set a new name for the parameter. Default is None.

class mpcpy.exodata.**ParameterFromDF**(*df*)

Collects parameter data from a pandas DataFrame object.

Parameters

df [pandas DataFrame object] DataFrame of data. The DataFrame index values must be named as the parameter names and the columns must be named as the parameter key names.

Attributes

data [dictionary] {"Parameter Name" : {"Parameter Key Name" : mpcpy.Variables.Static}}.

append_data(*name*, *value*, *free*, *minimum*, *maximum*, *covariance*, *unit*)

Append a new parameter to existing parameters.

Parameters

name [str] Name of parameter.

value [float | int] Value for the parameter.

free [boolean] True if parameter is free for parameter estimation.

minimum [float | int] Minimum for the parameter.

maximum [float | int] Maximum for the parameter.

covariance [float | int] Covariance for the parameter.

unit [mpcpy Units object] Unit of parameter.

collect_data()

Collect parameter data from DataFrame into data dictionary.

Yields

data [dictionary] Data attribute.

display_data()

Get data as pandas dataframe in display units.

Returns

df [pandas dataframe] Dataframe in display units.

get_base_data()

Get data as pandas dataframe in base units.

Returns

df [pandas dataframe] Dataframe in base units.

set_data(*name*, *value=None*, *free=None*, *minimum=None*, *maximum=None*, *covariance=None*, *new_name=None*)

Set new data for existing parameter.

All data must be in display units of parameter. No changes are made to arguments that are None.

Parameters

name [str] Name of parameter for which to set data.

value [float | int, optional] Set a new value for the parameter. Default is None.

free [boolean, optional] True if parameter is free for parameter estimation. Default is None.

minimum [float | int, optional] Set a new minimum for the parameter. Default is None.

maximum [float | int, optional] Set a new maximum for the parameter. Default is None.

covariance [float | int, optional] Set a new covariance for the parameter. Default is None.

new_name [str, optional] Set a new name for the parameter. Default is None.

4.8 Estimated States

Estimated state data represents data for states of models which may need to be estimated using system measurement data. Exogenous estimated state data has the following organization:

{"Estimated State Name" : {"Estimated State Key Name" : mpcpy.Variables.Static}}

The estimated state name must match that which is in the model. The estimated state key names should be chosen from the following list:

- Value - value of the estimated state, which is also used as an initial guess for state estimation algorithms
- Unit - unit string of estimated state

4.8.1 Classes

class mpcpy.exodata.**EstimatedStateFromCSV**(*csv_file_path*)

Collects estimated state data from a csv file.

Parameters

csv_file_path [string] Path of csv file. The csv file rows must be named as the estimated state names and the columns must be named as the estimated state key names.

Attributes

data [dictionary] {"Estimated State Name" : {"Estimated State Key Name" : mpcpy.Variables.Static}}.

file_path [string] Path of csv file.

append_data(*name*, *value*, *unit*, *parameter*)

Append a new estimated state to existing estimated states.

Parameters

name [str] Name of estimated state.

value [float | int] Value for the estimated state.

unit [mpcpy Units object] Unit of estimated state.

parameter [str] Name of parameter representing intial value of estimated state.

collect_data()

Collect estimated state data from csv file into data dictionary.

Yields

data [dictionary] Data attribute.

display_data()
Get data as pandas dataframe in display units.

Returns

df [pandas dataframe] Dataframe in display units.

get_base_data()
Get data as pandas dataframe in base units.

Returns

df [pandas dataframe] Dataframe in base units.

set_data(*name*, *value=None*, *new_name=None*, *parameter=None*)
Set new data for existing estimated state.

All data must be in display units of estimated state. No changes are made to arguments that are None.

Parameters

name [str] Name of estimated state for which to set data.

value [float | int, optional] Set a new value for the estimated state. Default is None.

new_name [str, optional] Set a new name for the estimated state. Default is None.

parameter [str, optional] Name of parameter representing intial value of estimated state. Default is None.

class mpcpy.exodata.**EstimatedStateFromDF**(*df*)
Collects estimated state data from a pandas DataFrame object.

Parameters

df [pandas DataFrame object] DataFrame of data. The DataFrame index values must be named as the estimated state names and the columns must be named as the estimated state key names.

Attributes

data [dictionary] {"Estimated State Name" : {"Estimated State Key Name" : mpcpy.Variables.Static}}.

append_data(*name*, *value*, *unit*, *parameter*)
Append a new estimated state to existing estimated states.

Parameters

name [str] Name of estimated state.

value [float | int] Value for the estimated state.

unit [mpcpy Units object] Unit of estimated state.

parameter [str] Name of parameter representing intial value of estimated state.

collect_data()
Collect estimated state data from DataFrame into data dictionary.

Yields

data [dictionary] Data attribute.

display_data()
Get data as pandas dataframe in display units.

Returns

df [pandas dataframe] Dataframe in display units.

get_base_data()
Get data as pandas dataframe in base units.

Returns

df [pandas dataframe] Dataframe in base units.

set_data(*name*, *value=None*, *new_name=None*, *parameter=None*)
Set new data for existing estimated state.

All data must be in display units of estimated state. No changes are made to arguments that are None.

Parameters

name [str] Name of estimated state for which to set data.

value [float | int, optional] Set a new value for the estimated state. Default is None.

new_name [str, optional] Set a new name for the estimated state. Default is None.

parameter [str, optional] Name of parameter representing intial value of estimated state. Default is None.

CHAPTER

FIVE

SYSTEMS

`systems` classes represent the controlled systems, with methods to collect measurements from or set control inputs to the system. This representation can be real or emulated using a detailed simulation model. A common interface to the controlled system in both cases allows for algorithm development and testing on a simulation with easy transition to the real system. Measurement data can then be passed to `models` objects to estimate or validate model parameters. Measurement data has a specified variable organization in the form of a Python dictionary in order to aid its use by other objects. It is as follows:

system.measurements = {"Measurement Variable Name" : {"Measurement Key" : mpcpy.Variables.Timeseries/Static}}.

The measurement variable name should match the variable that is measured in a model in the emulation case, or match the point name that is measured in a real system case. The measurement keys are from the following list:

- Simulated - timeseries variable for simulated measurement (yielded by `models` objects)
- Measured - timeseries variable for real measurement (yielded by `systems` objects)
- Sample - static variable for measurement sample rate
- SimulatedError - timeseries variable for simulated standard error
- MeasuredError - timeseries variable for measured standard error

5.1 Emulation

Emulation objects are used to simulate the performance of a real system and collect the results of the simulation as measurements. Models used for such simulations are often detailed physical models and are not necessarily the same as a model used for optimization. A model for this purpose should be instantiated as a `models` object instead of a `systems` object.

5.1.1 Classes

class mpcpy.systems.**EmulationFromFMU**(*measurements*, *save_parameter_input_data=False*, ***kwargs*)

System emulation by FMU simulation.

Parameters

measurements [dictionary] {"Measurement Name" : {"Sample" : mpcpy.Variables.Static}}.

fmupath [string, required if not moinfo] FMU file path.

moinfo [tuple or list, required if not fmupath] (mopath, modelpath, libraries). *mopath* is the path to the modelica file. *modelpath* is the path to the model to be compiled within the package specified in the modelica file. *libraries* is a list of paths directing to extra libraries required to compile the fmu.

zone_names [list, optional] List of zone name strings.

weather_data [dictionary, optional] `exodata` weather object data attribute.

internal_data [dictionary, optional] `exodata` internal object data attribute.

control_data [dictionary, optional] `exodata` control object data attribute.

other_inputs [dictionary, optional] `exodata` other inputs object data attribute.

parameter_data [dictionary, optional] `exodata` parameter object data attribute.

tz_name [string, optional] Name of timezone according to the package `tzwhere`. If `'from_geography'`, then geography kwarg is required.

geography [list or tuple, optional] List or tuple with (latitude, longitude) in degrees.

save_parameter_input_data: boolean True to output the parameter and input data set for simulations and optimizations. Saved files are: "mpcpy_simulation_parameters_system.csv" "mpcpy_simulation_inputs_system.csv" Times will be in UTC. Default is False.

Attributes

measurements [dictionary] {"Measurement Name" : {"Measurement *Key*" : mpcpy.Variables.Timeseries/Static}}.

fmu [pyfmi fmu object] FMU respresenting the emulated system.

fmupath [string] Path to the FMU file.

lat [numeric] Latitude in degrees. For timezone.

lon [numeric] Longitude in degrees. For timezone.

`collect_measurements`(*start_time*, *final_time*)

Collect measurement data for the emulated system by simulation using any given exodata inputs.

Parameters

start_time [string] Start time of measurements collection. Set to 'continue' in order to continue the emulation simulation from the final time of the previous simulation. Exodata input objects must contain values for the continuation timestamp. The measurements in a continued simulation replace previous values. They do not append to a previous simulation's measurements.

final_time [string] Final time of measurements collection. Must be greater than the start time.

Yields

Updates the "'Measured'" key for each measured variable in the

measurements dictionary attribute.

`display_measurements`(*measurement_key*)

Get measurements data in display units as pandas dataframe.

Parameters

measurement_key [string] The measurement dictionary key for which to get the data for all of the variables where the data exists.

Returns

df [pandas dataframe] Timeseries dataframe in display units containing data for all measurement variables.

get_base_measurements(*measurement_key*)
Get measurements data in base units as pandas dataframe.

Parameters

measurement_key [string] The measurement dictionary key for which to get the data for all of the variables where the data exists.

Returns

df [pandas dataframe] Timeseries dataframe in base units containing data for all measurement variables.

5.2 Real

Real objects are used to find and collect measurements from a real system.

5.2.1 Classes

class mpcpy.systems.**RealFromCSV**(*csv_file_path*, *measurements*, *variable_map*, ***kwargs*)
System measured data located in csv.

Parameters

csv_file_path [string] Path of csv file.

measurements [dictionary] {"Measurement Name" : {"Sample" : mpcpy.Variables.Static}}.

variable_map [dictionary] {"Column Header Name" : ("Measurement Variable Name", mpcpy.Units.unit)}.

tz_name [string, optional] Name of timezone according to the package tzwhere. If 'from_geography', then geography kwarg is required.

geography [list or tuple, optional] List or tuple with (latitude, longitude) in degrees.

Attributes

measurements [dictionary] {"Measurement Variable Name" : {{"Measurement *Key*" : mpcpy.Variables.Timeseries/Static}}.

file_path [string] Path of csv file.

lat [numeric] Latitude in degrees. For timezone.

lon [numeric] Longitude in degrees. For timezone.

collect_measurements(*start_time*, *final_time*)
Collect measurement data for the real system.

Parameters

start_time [string] Start time of measurements collection.

final_time [string] Final time of measurements collection.

Yields

Updates the ‘‘’Measured’‘‘ key for each measured variable in the

measurements dictionary attribute.

display_measurements(*measurement_key*)

Get measurements data in display units as pandas dataframe.

Parameters

measurement_key [string] The measurement dictionary key for which to get the data for all of the variables where the data exists.

Returns

df [pandas dataframe] Timeseries dataframe in display units containing data for all measurement variables.

get_base_measurements(*measurement_key*)

Get measurements data in base units as pandas dataframe.

Parameters

measurement_key [string] The measurement dictionary key for which to get the data for all of the variables where the data exists.

Returns

df [pandas dataframe] Timeseries dataframe in base units containing data for all measurement variables.

class mpcpy.systems.**RealFromDF**(*df*, *measurements*, *variable_map*, ***kwargs*)

System measured data located in DataFrame.

Parameters

df [pandas DataFrame object] DataFrame of data. The index must be a datetime object.

measurements [dictionary] {“Measurement Name” : {“Sample” : mpcpy.Variables.Static}}.

variable_map [dictionary] {“Column Header Name” : (“Measurement Variable Name”, mpcpy.Units.unit)}.

tz_name [string, optional] Name of timezone according to the package `tzwhere`. If `'from_geography'`, then geography kwarg is required.

geography [list or tuple, optional] List or tuple with (latitude, longitude) in degrees.

Attributes

measurements [dictionary] {“Measurement Variable Name” : {{“Measurement *Key*” : mpcpy.Variables.Timeseries/Static}}.

df [pandas DataFrame object] DataFrame of data.

lat [numeric] Latitude in degrees. For timezone.

lon [numeric] Longitude in degrees. For timezone.

collect_measurements(*start_time*, *final_time*)

Collect measurement data for the real system.

Parameters

start_time [string] Start time of measurements collection.

final_time [string] Final time of measurements collection.

Yields

Updates the ‘’’Measured’‘‘ key for each measured variable in the

measurements dictionary attribute.

display_measurements(*measurement_key*)

Get measurements data in display units as pandas dataframe.

Parameters

measurement_key [string] The measurement dictionary key for which to get the data for all of the variables where the data exists.

Returns

df [pandas dataframe] Timeseries dataframe in display units containing data for all measurement variables.

get_base_measurements(*measurement_key*)

Get measurements data in base units as pandas dataframe.

Parameters

measurement_key [string] The measurement dictionary key for which to get the data for all of the variables where the data exists.

Returns

df [pandas dataframe] Timeseries dataframe in base units containing data for all measurement variables.

CHAPTER

SIX

MODELS

`models` classes are models that are used for performance prediction in MPC. This includes models for physical systems (e.g. thermal envelopes, HVAC equipment, facade elements) and occupants at the component level or at an aggregated level (e.g. zone, building, campus).

6.1 Modelica

`Modelica` model objects utilize models represented in Modelica or by an FMU.

6.1.1 Classes

class mpcpy.models.**Modelica**(*parameter_estimate_method*, *validate_method*, *measurements*, *state_estimate_method=None*, *save_parameter_input_data=False*, ***kwargs*)

Class for models of physical systems represented by Modelica or an FMU.

Parameters

parameter_estimate_method [parameter estimation method class from mpcpy.models] Method for performing the parameter estimation.

validate_method [validation method class from mpcpy.models] Method for performing the parameter validation.

measurements [dictionary] Measurement variables for the model. Same as the measurements attribute from a `systems` class. See documentation for `systems` for more information.

state_estimate_method [state estimation method class from mpcpy.models, optional] Method for performing the state estimation. Default is models.UKFState

moinfo [tuple or list] Modelica information for the model. See documentation for `systems.EmulationFromFMU` for more information.

zone_names [list, optional] List of zone name strings.

weather_data [dictionary, optional] `exodata` weather object data attribute.

internal_data [dictionary, optional] `exodata` internal object data attribute.

control_data [dictionary, optional] `exodata` control object data attribute.

other_inputs [dictionary, optional] `exodata` other inputs object data attribute.

parameter_data [dictionary, optional] `exodata` parameter object data attribute.

estimated_state_data [dictionary, optional] `exodata` estimated state object data attribute.

tz_name [string, optional] Name of timezone according to the package `tzwhere`. If `'from_geography'`, then geography kwarg is required.

geography [list or tuple, optional] List or tuple with (latitude, longitude) in degrees.

save_parameter_input_data: boolean True to output the parameter and input data set for simulations and optimizations Saved files are: "mpcpy_simulation_parameters_model.csv" "mpcpy_simulation_inputs_model.csv" "mpcpy_simulation_parameters_optimization_initial.csv" "mpcpy_simulation_inputs_optimization_initial.csv" "mpcpy_optimization_parameters.csv" "mpcpy_optimization_inputs.csv" Times will be in UTC. Default is False.

Attributes

measurements [dictionary] `systems` measurement object attribute.

fmu [pyfmi fmu object] FMU respresenting the emulated system.

fmupath [string] Path to the FMU file.

lat [numeric] Latitude in degrees. For timezone.

lon [numeric] Longitude in degrees. For timezone.

tz_name [string] Timezone name.

display_measurements(*measurement_key*)

Get measurements data in display units as pandas dataframe.

Parameters

measurement_key [string] The measurement dictionary key for which to get the data for all of the variables where the data exists.

Returns

df [`pandas` dataframe] Timeseries dataframe in display units containing data for all measurement variables.

get_base_measurements(*measurement_key*)

Get measurements data in base units as pandas dataframe.

Parameters

measurement_key [string] The measurement dictionary key for which to get the data for all of the variables where the data exists.

Returns

df [`pandas` dataframe] Timeseries dataframe in base units containing data for all measurement variables.

parameter_estimate(*start_time*, *final_time*, *measurement_variable_list*, *global_start=0*, *seed=None*, *use_initial_values=True*)

Estimate the parameters of the model.

The estimation of the parameters is based on the data in the `'Measured'` key in the measurements dictionary attribute, the parameter_data dictionary attribute, and any exodata inputs.

An optional global start algorithm where multiple estimations are preformed with different initial guesses within the ranges of each free parameter provided. It is implemented as tested in Blum et al. (2019). The algorithm uses latin hypercube sampling to choose the initial parameter guess values for each iteration and the iteration with the lowest estimation problem objective value is chosen. A user-provided guess is included by default using initial values given to parameter data of the model, though this option can be turned off to use only sampled initial guesses.

Blum, D.H., Arendt, K., Rivalin, L., Piette, M.A., Wetter, M., and Veje, C.T. (2019). "Practical factors of envelope model setup and their effects on the performance of model predictive control for building heating, ventilating, and air conditioning systems." Applied Energy 236, 410-425. https://doi.org/10.1016/j.apenergy.2018.11.093

Parameters

start_time [string] Start time of estimation period. Setting to 'continue' will result in error.

final_time [string] Final time of estimation period.

measurement_variable_list [list] List of strings defining for which variables defined in the measurements dictionary attirubute the estimation will try to minimize the error.

global_start [int, optional] Number of iterations of a global start algorithm. If 0, the global start algorithm is disabled and the values in the parameter_data dictionary are used as initial guesses. Default is 0.

seed [numeric or None, optional] Specific seed of the global start algorithm for the random selection of initial value guesses. Default is None.

use_initial_values [boolean, optional] True to include initial parameter values in the estimation iterations. Default is True.

Yields

Updates the ‘‘’Value’‘‘ key for each estimated parameter in the

parameter_data attribute.

set_parameter_estimate_method(*parameter_estimate_method*)
Set the parameter estimation method for the model.

Parameters

estimate_method [estimation method class from mpcpy.models] Method for performing the parameter estimation.

set_state_estimate_method(*state_estimate_method*)
Set the state estimation method for the model.

Parameters

state_estimate_method [estimation method class from mpcpy.models] Method for performing the state estimation.

set_validate_method(*validate_method*)
Set the validation method for the model.

Parameters

validate_method [validation method class from mpcpy.models] Method for performing the parameter validation.

simulate(*start_time*, *final_time*)
Simulate the model with current parameter estimates and any exodata inputs.

Parameters

start_time [string] Start time of validation period. Set to 'continue' in order to continue the model simulation from the final time of the previous simulation, estimation, or validation. Continuous states from simulation and validation are saved. Exodata input objects must contain values for the continuation timestamp. The measurements in a continued simulation replace previous values. They do not append to a previous simulation's measurements.

final_time [string] Final time of simulation period. Must be greater than the start time.

Yields

Updates the ‘‘’Simulated’‘‘ key for each measurement in the

measurements attribute.

state_estimate(*start_time*, *final_time*, *measurement_variable_list*)

Estimate the states of the model.

The estimation of the states is based on the data in the `'Measured'` key in the measurements dictionary attribute, the estimated_state_data dictionary attribute, and any exodata inputs.

Parameters

start_time [string] Start time of estimation period. Setting to ‘continue’ will result in error.

final_time [string] Final time of estimation period.

measurement_variable_list [list] List of strings defining for which variables defined in the measurements dictionary attribute the estimation will use.

Yields

Updates the ‘‘’Value’‘‘ key for each estimated state in the

estimated_state_data attribute.

In the case of using the JModelicaState state estimation method, which

implements a moving horizon state estimator, the ‘‘’Value’‘‘ key for

each parameter corresponding to an estimated state in the

parameter_data attribute is also updated with the optimal result.

Note that this is not the estimated state value at the current time,

rather at the initial (historic) time of the state estimation.

validate(*start_time*, *final_time*, *validate_filename*, *plot=1*)

Validate the estimated parameters of the model.

The validation of the parameters is based on the data in the `'Measured'` key in the measurements dictionary attribute, the parameter_data dictionary attribute, and any exodata inputs.

Parameters

start_time [string] Start time of validation period. Set to ‘continue’ in order to continue the model simulation from the final time of the previous simulation, estimation, or validation. Continuous states from simulation and validation are saved. Exodata input objects must contain values for the continuation timestamp. The measurements in a continued simulation replace previous values. They do not append to a previous simulation’s measurements.

final_time [string] Final time of validation period.

validate_filepath [string] File path without an extension for which to save validation results. Extensions will be added depending on the file type (e.g. .png for figures, .txt for data).

plot [[0,1], optional] Plot flag for some validation or estimation methods. Default = 1.

Yields

Various results depending on the validation method. Please check the

documentation for the validation method chosen.

6.1.2 Parameter Estimate Methods

class mpcpy.models.**JModelicaParameter**(*Model*)
Parameter Estimation method using JModelica optimization.

This estimation method sets up a parameter estimation problem to be solved using JModelica.

class mpcpy.models.**UKFParameter**(*Model*)
Parameter estimation method using the Unscented Kalman Filter.

This estimation method uses the UKF implementation EstimationPy.

6.1.3 State Estimate Methods

class mpcpy.models.**JModelicaState**(*Model*)
State Estimation method using JModelica optimization.

This estimation method sets up a simple moving horizon state estimation problem to be solved using JModelica. The method uses a parameter estimation with altered parameter data to estimate only initial states. Given a time period of observed state data, the method sets up and solves an optimization problem to find the optimal values of the estimated states at the initial time of the time period that minimizes the error between the measured observed states and modeled observed states over the time period. Then, the final value of the estimated states are taken as the current state estimates.

Based on the state estimator implemented in R. De Coninck and L. Helsen (2016). "Practical implementation and evaluation of model predictive control for an office building in Brussels." Energy and Buildings 111. 290-298. https://doi.org/10.1016/j.enbuild.2015.11.014

class mpcpy.models.**UKFState**(*Model*)
State estimation method using the Unscented Kalman Filter.

This estimation method uses the UKF implementation EstimationPy.

6.1.4 Validate Methods

class mpcpy.models.**RMSE**(*Model*)
Validation method that computes the RMSE between estimated and measured data.

Only modeled values with measurements corresponding to the same time are considered in the calculation of RMSE. If a measurement is detected as missing, a warning is printed.

Yields

RMSE [dictionary] {"Measurement Name" : mpcpy.Variables.Static}. Attribute of the model object that contains the RMSE for each measurement variable used to perform the validation in base units.

6.2 Occupancy

Occupancy models consider when occupants arrive and depart a space or building as well as how many occupants are present at a particular time.

6.2.1 Classes

class mpcpy.models.**Occupancy**(*occupancy_method*, *measurements*, ***kwargs*)

Class for models of occupancy.

Parameters

occupancy_method [occupancy method class from mpcpy.models]

measurements [dictionary] Measurement variables for the model. Same as the measurements attribute from a `systems` class. See documentation for `systems` for more information. This measurement dictionary should only have one variable key, which represents occupancy count.

tz_name [string, optional] Name of timezone according to the package `tzwhere`. If `'from_geography'`, then geography kwarg is required.

geography [list or tuple, optional] List or tuple with (latitude, longitude) in degrees.

Attributes

measurements [dictionary] `systems` measurement object attribute.

parameter_data [dictionary] `exodata` parameter object data attribute.

lat [numeric] Latitude in degrees. For timezone.

lon [numeric] Longitude in degrees. For timezone.

tz_name [string] Timezone name.

display_measurements(*measurement_key*)

Get measurements data in display units as pandas dataframe.

Parameters

measurement_key [string] The measurement dictionary key for which to get the data for all of the variables where the data exists.

Returns

df [`pandas` dataframe] Timeseries dataframe in display units containing data for all measurement variables.

estimate(*start_time*, *final_time*, ***kwargs*)

Estimate the parameters of the model using measurement data.

The estimation of the parameters is based on the data in the `'Measured'` key in the measurements dictionary attribute of the model object.

Parameters

start_time [string] Start time of estimation period.

final_time [string] Final time of estimation period.

estimate_options [dictionary, optional] Use the `get_estimate_options` method to obtain and edit.

Yields

parameter_data [dictionary] Updates the `'Value'` key for each estimated parameter in the parameter_data attribute.

get_base_measurements(*measurement_key*)

Get measurements data in base units as pandas dataframe.

Parameters

measurement_key [string] The measurement dictionary key for which to get the data for all of the variables where the data exists.

Returns

df [pandas dataframe] Timeseries dataframe in base units containing data for all measurement variables.

get_constraint(*occupied_value*, *unoccupied_value*)
Get a constraint timeseries based on the predicted occupancy.

Parameters

occupied_value [mpcpy.variables.Static] Value of constraint during occupied times.

unoccupied_value [mpcpy.variables.Static] Value of constraint during unoccupied times.

Returns

constraint [mpcpy.variables.Timeseries] Constraint timeseries.

get_estimate_options()
Set the estimation options for the model.

Returns

estimate_options [dictionary] Options for estimation of occupancy model parameters. Please see documentation for specific occupancy model for more information.

get_load(*load_per_person*)
Get a load timeseries based on the predicted occupancy.

Parameters

load_per_person [mpcpy.variables.Static] Scaling factor of occupancy prediction to produce load timeseries.

Returns

load [mpcpy.variables.Timeseries] Load timeseries.

get_simulate_options()
Get the simulation options for the model.

Returns

simulate_options [dictionary] Options for simulation of occupancy model. Please see documentation for specific occupancy model for more information.

set_estimate_options(*estimate_options*)
Set the estimation options for the model.

Parameters

estimate_options [dictionary] Options for estimation of occupancy model parameters. Please see documentation for specific occupancy model for more information.

set_occupancy_method(*occupancy_method*)
Set the occupancy method for the model.

Parameters

occupancy_method [occupancy method class from mpcpy.models]

set_simulate_options(*simulate_options*)
Set the simulation options for the model.

Parameters

simulate_options [dictionary] Options for simulation of occupancy model. Please see documentation for specific occupancy model for more information.

simulate(*start_time*, *final_time*, ***kwargs*)
Simulate the model with current parameter estimates.

Parameters

start_time [string] Start time of simulation period.

final_time [string] Final time of simulation period.

simulate_options [dictionary, optional] Use the `get_simulate_options` method to obtain and edit.

Yields

measurements [dictionary] Updates the `'Simulated'` key for each measurement in the measurements attribute. If available by the occupancy method, also updates the `'SimulatedError'` key for each measurement in the measurements attribute.

validate(*start_time*, *final_time*, *validate_filename*, *plot=1*)
Validate the estimated parameters of the model with measurement data.

The validation of the parameters is based on the data in the `'Measured'` key in the measurements dictionary attribute of the model object.

Parameters

start_time [string] Start time of validation period.

final_time [string] Final time of validation period.

validate_filepath [string] File path without an extension for which to save validation results. Extensions will be added depending on the file type (e.g. .png for figures, .txt for data).

plot [[0,1], optional] Plot flag for some validation or estimation methods.

Yields

Various results depending on the validation method. Please check the

documentation for the occupancy model chosen.

6.2.2 Occupancy Methods

class mpcpy.models.**QueueModel**
Occupancy presence prediction based on a queueing approach.

Based on Jia, R. and C. Spanos (2017). "Occupancy modelling in shared spaces of buildings: a queueing approach." Journal of Building Performance Simulation, 10(4), 406-421.

See `occupant.occupancy.queueing` for more information.

Attributes

estimate_options [dictionary] Specifies options for model estimation with the following keys: -res : defines the resolution of grid search for the optimal breakpoint placement -margin : specifies the minimum distance between two adjacent breakpoints -n_max : defines the upper limit of the number of breakpoints returned by the algorithm

simulate_options [dictionary] Specifies options for model simulation. -iter_num : defines the number of iterations for monte-carlo simulation.

CHAPTER

SEVEN

OPTIMIZATION

`Optimization` objects setup and solve mpc control optimization problems. The optimization uses `models` objects to setup and solve the specified optimization problem type with the specified optimization package type. Constraint information can be added to the optimization problem through the use of the constraint `exodata` object. Please see the `exodata` documentation for more information.

7.1 Classes

class mpcpy.optimization.**Optimization**(*Model*, *problem_type*, *package_type*, *objective_variable*, ***kwargs*)

Class for representing an optimization problem.

Parameters

Model [mpcpy.model object] Model with which to perform the optimization.

problem_type [mpcpy.optimization.problem_type] The type of poptimization problem to solve. See specific documentation on available problem types.

package_type [mpcpy.optimization.package_type] The software package used to solve the optimization problem. The model is translated into an optimization problem accoding to the problem_type to be solved in the specified package_type. See specific documentation on available package types.

objective_variable [string] The name of the model variable to be used in the objective function.

constraint_data [dictionary, optional] `exodata` constraint object data attribute.

demand_periods [int, optional, but required if problem_type includes demand.] Maximum number of different demand periods expected to be represented in price data. This should include coincident demand if needed.

Attributes

Model [mpcpy.model object] Model with which to perform the optimization.

objective_variable [string] The name of the model variable to be used as the objective variable.

constraint_data [dictionary] `exodata` constraint object data attribute.

get_optimization_options()

Get the options for the optimization solver package.

Returns

opt_options [dictionary] The options for the optimization solver package. See specific documentation on solver package for more information.

get_optimization_statistics()
Get the optimization result statistics from the solver package.

Returns

opt_statistics [dictionary] The options for the optimization solver package. See specific documentation on solver package for more information.

optimize(*start_time*, *final_time*, ***kwargs*)
Solve the optimization problem over the specified time horizon.

Consult the documentation for the solver package type for available kwargs.

Parameters

start_time [string] Start time of estimation period.

final_time [string] Final time of estimation period.

Yields

Upon solving the optimization problem, this method updates the ‘‘Model.control_data‘‘ dictionary with the optimal control timeseries for each control variable for the time period of optimization. If the optimization horizon extends past the final time of ‘‘Model.control_data‘‘, then the extra data is appended. Also creates the Optimization.measurements dictionary with the optimization solution measurements under the ‘‘’Simulated’‘‘ key. This is created for the variables defined in ‘‘Model.measurements‘‘.

set_optimization_options(*opt_options*)
Set the options for the optimization solver package.

Parameters

opt_options [dictionary] The options for the optimization solver package. See specific documentation on solver package for more information.

set_package_type(*package_type*)
Set the solver package type of the optimization.

Parameters

package_type [mpcpy.optimization.package_type] New software package type to use to solve the optimization problem. See specific documentation on available package types.

set_problem_type(*problem_type*)
Set the problem type of the optimization.

Note that optimization options will be reset.

Parameters

problem_type [mpcpy.optimization.problem_type] New problem type to solve. See specific documentation on available problem types.

7.2 Problem Types

class mpcpy.optimization.**EnergyMin**

Minimize the integral of the objective variable, $P(t)$, over the time horizon from time t_s to time t_f.

$$minJ = \int_{t_s}^{t_f} Pdt$$

class mpcpy.optimization.**EnergyCostMin**

Minimize the integral of the objective variable, $P(t)$, multiplied by a time-varying weighting factor, $\pi_e(t)$, over the time horizon from time t_s to time t_f.

$$minJ = \int_{t_s}^{t_f} \pi_e * Pdt$$

class mpcpy.optimization.**EnergyPlusDemandCostMin**

Minimize the integral of the objective variable, $P(t)$, multiplied by a time-varying weighting factor, $\pi_e(t)$, over the time horizon from time t_s to time t_f plus the incremental maximum of the objective variable over a time period, P_τ, over a previously observed or estimated maximum for the same time period, $P_{est,\tau}$, with period-specific demand costs, $\pi_{d,\tau}$. Note that τ represents ranges of time corresponding to period-specific demand costs.

$$minJ = \int_{t_s}^{t_f} \pi_e * Pdt + \sum_{\tau} \pi_{d,\tau} * (max(P_\tau) - P_{est,\tau})$$

This formulation was compared with other formulations considering demand in [1] and was found to improve responsiveness to energy shifting without increasing customer utility bills. For implementation in continuous time and for use with NLP solvers, the problem can be implemented with the following transformation.

$$minJ_{[z_\tau]} = \int_{t_s}^{t_f} \pi_e * Pdt + \sum_{\tau} \pi_{d,\tau} * z_\tau$$

$$s.t.$$

$$P \leq z_\tau + \hat{z}_\tau \qquad \forall \tau$$

$$z_\tau \geq 0 \qquad \forall \tau$$

$$where$$

$$\hat{z}_\tau = P_{est,\tau} \qquad \forall t \in \tau$$

$$\hat{z}_\tau = M >> 1 \qquad \forall t \notin \tau$$

References:

[1] O. V. Cutsem, D. H. Blum, M. Kayal, and M. Pritoni. (2019). "Comparison of MPC Formulations for Building Control under Commercial Time-of-Use Tariffs." Proc. of the 13th IEEE PES PowerTech, Jun 23-27. Milan, Italy.

7.3 Package Types

class mpcpy.optimization.**JModelica**(*Optimization*)

Use JModelica to solve the optimization problem.

This package is compatible with `models.Modelica` objects. Please consult the JModelica user guide for more information regarding optimization options and solver statistics.

The option 'n_e' is overwritten by default to equal the number of points as calculated using the model measurements sample rate and length of optimization horizon (same as if model is simulated). However, editing this option will overwrite this default.

Notes

`optimize()` kwargs:

res_control_step [int, optional] The time interval in seconds at which the model.control_data is updated with the optimal control results. The control data comes from evaluating the optimal input collocation polynomials at the specified time interval. The default value is the interval returned by JModelica according to the 'result_mode' option. See JModelica documentation for more details.

price_data [dictionary] `exodata` price object data attribute. For EnergyCostMin problems only.

CHAPTER

EIGHT

TESTING

Testing in MPCPy ensures stable software development. The directory `/unittests` contains modules for tests and testing functionality, as well as directories that hold testing resources and reference results. Reference results are stored for tests that require comparisons of complex data, including DataFrames and json structures. The `/unittests/testing.py` module contains functionality for testing of these complex structures. Other tests on more simple data are carried out with reference values hard-coded within the test. Please consult the classes below for more information about testing complex structures in MPCPy.

The script `/bin/runUnitTests.py` is used to manage the running of all of the tests. Consult the Getting Started section of the user guide for information on how to use this script to run tests. There is a test module for each of the main MPCPy modules named `test_*`. Within each test module there are classes for testing the different components of MPCPy, while the specific test functions, `test_*`, are executed within each test class.

8.1 Classes

class unittests.testing.**TestCaseMPCPy**(*methodName='runTest'*)

General test methods for testing in mpcpy.

check_df(*df_test*, *ref_file_name*, *timeseries=True*)

Compares DataFrame test data to reference data with tolerance.

If the reference data file does not exist, a reference data file is using the test DataFrame.

If a failure is found, a message is printed indicating if the failure is from index, key, or value checking. If the failure is index or key, the location will be specified. If the failure is value, a plot will be generated and saved within the reference file folder for inspection. The plot will show the reference and testing values, along with the location of the maximum error between the two.

Parameters

df_test [pandas DataFrame] DataFrame of timeseries data to test

ref_file_name [string] Path to csv file containing reference data

timeseries [boolean] True if the index of df_test is timestamps.

check_json(*json_test*, *ref_file_name*)

Compares json test data to reference data.

Uses method `assertEqual()`. If the reference data file does not exist, a reference data file is created.

Parameters

json_test [dictionary] Dictionary of data to test

ref_file_name [string] Path to csv file containing reference data

get_ref_path()
Returns the path to the test data reference file.

get_unittest_path()
Returns the path to the unittest directory.

CHAPTER

NINE

ACKNOWLEDGMENTS

This research 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.

This work is funded by the U.S.-China Clean Energy Research Center (CERC) 2.0 on Building Energy Efficiency (BEE).

Thank you to all who have provided guidance on the development of this program. The following people have contributed directly to the development of this program (in alphabetical order):

- Krzysztof Arendt, University of Southern Denmark
- David H. Blum, Lawrence Berkeley National Laboratory
- Ruoxi Jia, University of California, Berkeley
- Lisa Rivalin, Engie Axima
- Zhe Wang, Lawrence Berkeley National Laboratory
- Michael Wetter, Lawrence Berkeley National Laboratory

CHAPTER

TEN

DISCLAIMERS

CHAPTER

ELEVEN

COPYRIGHT AND LICENSE

11.1 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.

11.2 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

d

m

u

INDEX

E

G

I

J

M

O

P

Q

R

S

T

U

V

W