# MPCPy User Guide

***Release 0.1***

**Lawrence Berkeley National Laboratory**

July 31, 2017

# CHAPTER ONE

# INTRODUCTION

## 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.

## 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, emulation and MPC models for real systems rely heavily 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 building research and in 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.

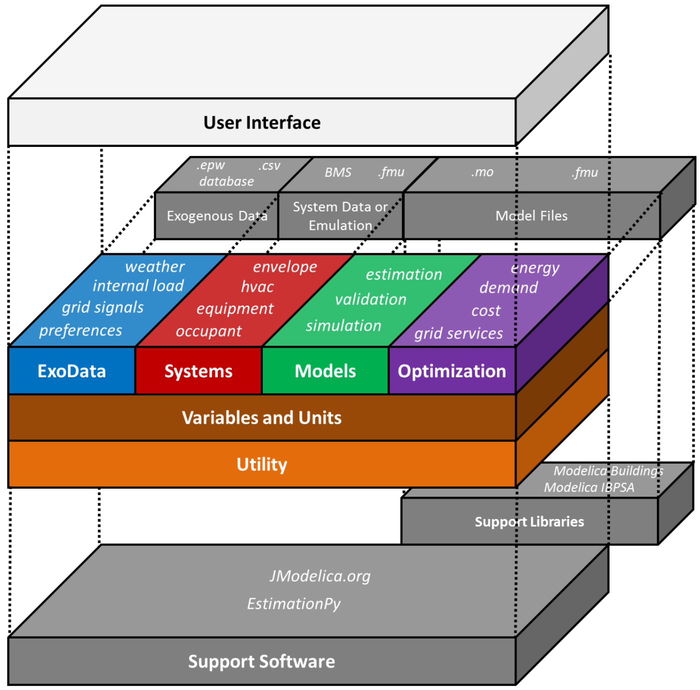


Fig. 1.1: Software architecture diagram for MPCPy. Note that a user interface has not been developed.

# 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. Infact, 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.

# CHAPTER TWO

# GETTING STARTED

To get started with MPCPy, first follow the installation instructions below. Then, checkout the introductory tutorial or explore the ipython notebooks in the `examples/` directory to get a feel for the workflow of MPCPy. You can always consult the user guide for more information.

## Installation Instructions For Linux (Ubuntu 16.04 LTS)

1. Install Python Packages
   - matplotlib 1.5.1
   - numpy 1.11.0
   - pandas 0.17.1
   - python-dateutil 2.4.2
   - pytz 2014.10
   - scikit-learn 0.18.1
   - tzwhere 2.3
   - sphinx 1.3.6
2. Install JModelica 2.0 (for Modelica compiling, optimization, and fmu simulation)
3. Create JModelica environmental variables
   - add the following lines to your bashrc script:

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

4. Download or Clone EstimationPy-KA
   - go to https://github.com/krzysztofarendt/EstimationPy-KA and clone or download repository into a directory (let's call it `.../EstimationPy-KA`).
5. 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`).

6. Edit PYTHONPATH environmental variable

   - add the following lines to your bashrc script (assumes 3. above sets JMODELICA_HOME):

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

7. Test the installation

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

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

## 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.

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;
```

### 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)
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()
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)
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
2017-01-01 01:00:00+00:00     5
2017-01-01 02:00:00+00:00    10
2017-01-01 03:00:00+00:00    15
2017-01-01 04:00:00+00:00    20
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. 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)
-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()
                             Qflow
Time
2017-01-01 06:00:00+00:00   3000
2017-01-01 07:00:00+00:00   3000
2017-01-01 08:00:00+00:00   3000
-etc-
```

## 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 only have one, and it is the output of the model, called ‘Tzone’. Note that ‘heatCapacitor.T’ would also be valid.

```
>>> measurements = {'Tzone' : {}}
>>> measurements['Tzone']['Sample'] = variables.Static('sample_rate_Tzone',
...                                                     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')
-etc-
>>> # Display the results
>>> emulation.display_measurements('Measured')
                                    Tzone
Time
```

```
2017-01-01 06:00:00+00:00  293.150000
2017-01-01 07:00:00+00:00  291.015800
2017-01-01 08:00:00+00:00  291.335967
-etc-
```

## 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()
                  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.JModelica,
...                         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')
```

```
-etc-
>>> # Display the results
>>> model.display_measurements('Simulated')
                                Tzone
Time
2017-01-01 06:00:00+00:00  293.150000
2017-01-01 07:00:00+00:00  266.964432
2017-01-01 08:00:00+00:00  267.440054
-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.estimate('1/1/2017', '1/2/2017', ['Tzone'])
-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)
-etc-
>>> # Get RMSE
>>> print(model.RMSE['Tzone'].display_data())
0.0555918208623
```

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)
-etc-
>>> # Assign new measurements to model
>>> model.measurements = emulation.measurements
>>> # Perform validation
>>> model.validate(start_time_val, final_time_val, 'validate_val', plot=1)
-etc-
>>> # Get RMSE
>>> print(model.RMSE['Tzone'].display_data())
0.0556106422491
```

Finally, let's view the estimated parameter values:

```
>>> for key in model.parameter_data.keys():
...     print(key, model.parameter_data[key]['Value'].display_data())
('heatCapacitor.C', 121756.56210192)
('thermalResistor.R', 0.0100114401286855)
```

## 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)
>>> # Collect data
>>> constraints.collect_data('1/1/2017', '1/3/2017')
>>> # Get data
>>> constraints.display_data()
                           Qflow_GTE  Qflow_LTE  Tzone_GTE  Tzone_LTE
Time
2017-01-01 00:00:00+00:00          0       4000         20         25
2017-01-01 01:00:00+00:00          0       4000         20         25
2017-01-01 02:00:00+00:00          0       4000         20         25
-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.

```
>>> opt_problem.optimize('1/2/2017', '1/3/2017')
-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()
('Solve_Succeeded', 12, -etc-)
```

Finally, 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.Model.control_data['Qflow'].display_data()
2017-01-02 06:00:00+00:00           645.563337
2017-01-02 06:09:18.183693+00:00   1501.595902
2017-01-02 06:38:41.816307+00:00   2603.388448
2017-01-02 07:00:00+00:00          1879.949971
-etc-
>>> opt_problem.Model.display_measurements('Simulated')
                                   Tzone
Time
2017-01-02 06:00:00+00:00         298.15
2017-01-02 06:09:18.183693+00:00  293.15
2017-01-02 06:38:41.816307+00:00  293.15
2017-01-02 07:00:00+00:00         293.15
-etc-
```

## 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.

## 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. |

### Methods

| | |
|---|---|
| *display_data*(**kwargs) | Return the data of the variable in display units. |
| *get_base_data*() | Return the data of the variable in base units. |
| *get_base_unit*() | Returns the base unit of the variable. |
| *get_base_unit_name*() | Returns the base unit name of the variable. |
| *get_display_unit*() | Returns the display unit of the variable. |
| *get_display_unit_name*() | Returns the display unit name of the variable. |
| *set_data*(data) | Set data of Static variable. |
| *set_display_unit*(display_unit) | Set the display unit of the variable. |

**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. |

### Methods

| | |
|---|---|
| *cleaning_replace*((to_replace, replace_with)) | Cleaning method to replace values within timeseries. |
| *display_data*(**kwargs) | Return the data of the variable in display units. |
| *get_base_data*() | Return the data of the variable in base units. |
| *get_base_unit*() | Returns the base unit of the variable. |
| *get_base_unit_name*() | Returns the base unit name of the variable. |
| *get_display_unit*() | Returns the display unit of the variable. |
| *get_display_unit_name*() | Returns the display unit name of the variable. |
| *set_data*(timeseries[, tz_name]) | Set data of Timeseries variable. |
| *set_display_unit*(display_unit) | Set the display unit of the variable. |

**cleaning_replace**(*(to_replace, replace_with)*)

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.

## 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}`

## Classes

**class** `mpcpy.exodata.`**`WeatherFromEPW`**(*epw_file_path*)

Collects weather data from an EPW file.

**Parameters** **epw_file_path** : string

Path of epw file.

### Attributes

| | |
|---|---|
| data | (dictionary) {"Weather Variable Name" : mpcpy.Variables.Timeseries}. |
| lat | (numeric) Latitude in degrees. |
| lon | (numeric) Longitude in degrees. |
| tz_name | (string) Timezone name. |
| file_path | (string) Path of epw file. |

### Methods

| | |
|---|---|
| *collect_data*(start_time, final_time) | Collect data from specified source and update data attribute. |
| *display_data*() | Get data in display units as pandas dataframe. |
| *get_base_data*() | Get data in base units as pandas dataframe. |

**`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*, ***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)}.

### Attributes

| | |
|---|---|
| data | (dictionary) {"Weather Variable Name" : mpcpy.Variables.Timeseries}. |
| lat | (numeric) Latitude in degrees. |
| lon | (numeric) Longitude in degrees. |
| tz_name | (string) Timezone name. |
| file_path | (string) Path of csv file. |

### Methods

| | |
|---|---|
| *collect_data*(start_time, final_time) | Collect data from specified source and update data attribute. |
| *display_data*() | Get data in display units as pandas dataframe. |
| *get_base_data*() | Get data in base units as pandas dataframe. |

**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.

## 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`.

### 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 | (numeric) Latitude in degrees. For timezone. |
| lon | (numeric) Longitude in degrees. For timezone. |
| tz_name | (string) Timezone name. |
| file_path | (string) Path of csv file. |

#### Methods

| | |
|---|---|
| *collect_data*(start_time, final_time) | Collect data from specified source and update data attribute. |
| *display_data*() | Get data in display units as pandas dataframe. |
| *get_base_data*() | Get data in base units as pandas dataframe. |

**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 | (numeric) Latitude in degrees. For timezone. |
| lon | (numeric) Longitude in degrees. For timezone. |
| tz_name | (string) Timezone name. |

### Methods

| | |
|---|---|
| `collect_data`(start_time, final_time) | Collect data from specified source and update data attribute. |
| `display_data`() | Get data in display units as pandas dataframe. |
| `get_base_data`() | Get data in base units as pandas dataframe. |

**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.

## 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.

### 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 | (numeric) Latitude in degrees. For timezone. |
| lon | (numeric) Longitude in degrees. For timezone. |
| tz_name | (string) Timezone name. |
| file_path | (string) Path of csv file. |

### Methods

| | |
|---|---|
| `collect_data`(start_time, final_time) | Collect data from specified source and update data attribute. |
| `display_data`() | Get data in display units as pandas dataframe. |
| `get_base_data`() | Get data in base units as pandas dataframe. |

**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.

## 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.

### 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 | (numeric) Latitude in degrees. For timezone. |
| lon | (numeric) Longitude in degrees. For timezone. |
| tz_name | (string) Timezone name. |
| file_path | (string) Path of csv file. |

#### Methods

| | |
|---|---|
| `collect_data`(start_time, final_time) | Collect data from specified source and update data attribute. |
| `display_data`() | Get data in display units as pandas dataframe. |
| `get_base_data`() | Get data in base units as pandas dataframe. |

**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.

## 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. Exogenous price data has the following organization:

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

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

- pi_e - electrical energy price

## Classes

**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 | (numeric) Latitude in degrees. For timezone. |
| lon | (numeric) Longitude in degrees. For timezone. |
| tz_name | (string) Timezone name. |
| file_path | (string) Path of csv file. |

### Methods

| | |
|---|---|
| *collect_data*(start_time, final_time) | Collect data from specified source and update data attribute. |
| *display_data*() | Get data in display units as pandas dataframe. |
| *get_base_data*() | Get data in base units as pandas dataframe. |

**`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.

# 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. Exogenous constraint data has the following organization:

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

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

## Classes

**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

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

### Attributes

| | |
|---|---|
| data | (dictionary) {"Column Header Name" : ("State or Control Variable Name", "Constraint Variable Type", mpcpy.Units.unit)}. |
| lat | (numeric) Latitude in degrees. For timezone. |
| lon | (numeric) Longitude in degrees. For timezone. |
| tz_name | (string) Timezone name. |
| file_path | (string) Path of csv file. |

### Methods

| | |
|---|---|
| *collect_data*(start_time, final_time) | Collect data from specified source and update data attribute. |
| *display_data*() | Get data in display units as pandas dataframe. |
| *get_base_data*() | Get data in base units as pandas dataframe. |

**`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 | (numeric) Latitude in degrees. For timezone. |
| lon | (numeric) Longitude in degrees. For timezone. |
| tz_name | (string) Timezone name. |

### Methods

| | |
|---|---|
| `collect_data`(start_time, final_time) | Collect data from specified source and update data attribute. |
| `display_data`() | Get data in display units as pandas dataframe. |
| `get_base_data`() | Get data in base units as pandas dataframe. |

**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.

## 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

### Classes

**class** `mpcpy.exodata.`**ParameterFromCSV**(*csv_file_path*)
Collects parameter data from a csv file.

The csv file rows must be named as the parameter names and the columns must be named as the parameter key names.

**Parameters** **csv_file_path** : string

Path of csv file.

### Attributes

| | |
|---|---|
| data | (dictionary) {"Parameter Name" : {"Parameter Key Name" : mpcpy.Variables.Static}}. |
| file_path | (string) Path of csv file. |

### Methods

| | |
|---|---|
| `collect_data()` | Collect parameter data from csv file into data dictionary. |
| `display_data()` | Get data as pandas dataframe in display units. |
| `get_base_data()` | Get data as pandas dataframe in base units. |

**`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.

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

## 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.

### Classes

**class** `mpcpy.systems.`**`EmulationFromFMU`**(*measurements*, ***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.

### 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. |

### Methods

| | |
|---|---|
| *`collect_measurements`*(start_time, final_time) | Collect measurement data for the emulated system by simulation using any gi |
| *`display_measurements`*(measurement_key) | Get measurements data in display units as pandas dataframe. |
| *`get_base_measurements`*(measurement_key) | Get measurements data in base units as pandas dataframe. |

**`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.

**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.

**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.

**Returns df** : `pandas` dataframe

Timeseries dataframe in base units containing data for all measurement variables.

## Real

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

### 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

| | |
|---|---|
| measure-ments | (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. |

### Methods

| | |
|---|---|
| *collect_measurements*(start_time, final_time) | Collect measurement data for the real system. |
| *display_measurements*(measurement_key) | Get measurements data in display units as pandas dataframe. |
| *get_base_measurements*(measurement_key) | Get measurements data in base units as pandas dataframe. |

**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.

**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.

**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).

## Modelica

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

### Classes

**class** `mpcpy.models.`**`Modelica`**(*estimate_method*, *validate_method*, *measurements*, ***kwargs*)

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

**Parameters** **estimate_method** : 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.

**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.

**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. |
| 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. |

### Methods

| | |
|---|---|
| `display_measurements`(measurement_key) | Get measurements data in display units as pandas dataframe. |
| `estimate`(start_time, final_time, ...) | Estimate the parameters of the model. |
| `get_base_measurements`(measurement_key) | Get measurements data in base units as pandas dataframe. |
| `set_estimate_method`(estimate_method) | Set the estimation method for the model. |
| `set_validate_method`(validate_method) | Set the validation method for the model. |
| `simulate`(start_time, final_time) | Simulate the model with current parameter estimates and any exodata inputs. |
| `validate`(start_time, final_time, ...[, plot]) | Validate the estimated parameters of the model. |

**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.

**Returns** **df** : `pandas` dataframe

Timeseries dataframe in display units containing data for all measurement variables.

**estimate**(*start_time*, *final_time*, *measurement_variable_list*)

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.

**Parameters** **start_time** : string

Start time of estimation period.

**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.

**Yields** 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.

**Returns** **df** : `pandas` dataframe

Timeseries dataframe in base units containing data for all measurement variables.

**`simulate`**(*start_time*, *final_time*)

Simulate the model with current parameter estimates and any exodata inputs.

**Parameters** **start_time** : string

Start time of simulation period.

**final_time** : string

Final time of simulation period.

**Yields** Updates the `'Simulated'` key for each measurement in the

measurements attribute.

**`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.

**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.

### Estimate Methods

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

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

**class** `mpcpy.models.UKF`(*Model*)
Estimation method using the Unscented Kalman Filter.

This estimation method uses the UKF implementation EstimationPy-KA, which is a fork of EstimationPy that allows for parameter estimation without any state estimation.

### Validate Methods

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

**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.

## 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.

### 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. |

### Methods

| | |
|---|---|
| *display_measurements*(measurement_key) | Get measurements data in display units as pandas dataframe. |
| *estimate*(start_time, final_time, **kwargs) | Estimate the parameters of the model using measurement data. |
| *get_base_measurements*(measurement_key) | Get measurements data in base units as pandas dataframe. |
| *get_constraint*(occupied_value, unoccupied_value) | Get a constraint timeseries based on the predicted occupancy. |
| *get_estimate_options*() | Set the estimation options for the model. |
| *get_load*(load_per_person) | Get a load timeseries based on the predicted occupancy. |
| *get_simulate_options*() | Get the simulation options for the model. |
| *set_estimate_options*(estimate_options) | Set the estimation options for the model. |
| *set_occupancy_method*(occupancy_method) | Set the occupancy method for the model. |
| *set_simulate_options*(simulate_options) | Set the simulation options for the model. |
| *simulate*(start_time, final_time, **kwargs) | Simulate the model with current parameter estimates. |
| *validate*(start_time, final_time, ...[, plot]) | Validate the estimated parameters of the model with measurement data. |

**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.

**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.

**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.

## 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 informaiton can be added to the optimization problem through the use of the constraint `exodata` object. Please see the `exodata` documentation for more information.

## 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.

### Attributes

| | |
|---|---|
| Model | ( mpcpy.model object) Model with which to perform the optimization. |
| objective_variable | (string) The name of the model variable to be used in the objective function. |
| constraint_data | (dictionary) `exodata` constraint object data attribute. |

### Methods

| | |
|---|---|
| `get_optimization_options`() | Get the options for the optimization solver package. |
| `get_optimization_statistics`() | Get the optimization result statistics from the solver package. |
| `optimize`(start_time, final_time, **kwargs) | Solve the optimization problem over the specified time horizon. |
| `set_optimization_options`(opt_options) | Set the options for the optimization solver package. |
| `set_package_type`(package_type) | Set the solver package type of the optimization. |
| `set_problem_type`(problem_type) | Set the problem type of the optimization. |

**`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.

**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 and the Model.measurements

dictionary with the optimal measurements under the `'Simulated'` key.

**`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.

## Problem Types

**class** `mpcpy.optimization.EnergyMin`

Minimize the integral of the objective variable over the time horizon.

**class** `mpcpy.optimization.EnergyCostMin`

Minimize the integral of the objective variable multiplied by a time-varying weighting factor over the time horizon.

## 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.

CHAPTER

# EIGHT

# 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
- Michael Wetter, Lawrence Berkeley National Laboratory

# CHAPTER NINE

# DISCLAIMERS

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor The Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or The Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or The Regents of the University of California.

CHAPTER

TEN

# COPYRIGHT AND LICENSE

## Copyright



## License Agreement

# PYTHON MODULE INDEX

## d



## m

## C



## D



## E



## G

## I



## J



## M



## O



## P



## Q



## R



## S



## T



## U

## V



## W