BOPTEST Tutorial #1

An Introduction to BOPTEST Software for Test Case Development and Interaction

By David Blum, Javier Arroyo, and Donghun Kim

Based on a tutorial delivered to IBPSA Project 1 Expert Meeting Work Package 1.2 in Rome, Italy on August 31, 2019.

Updated: April 6, 2023, compatible with BOPTEST v0.4.0

Introduction

The Building Optimization Test Framework (BOPTEST) is developed to enable the fair comparison and benchmarking of advanced control strategies in buildings. The framework is composed of three main parts:

1. Test Cases
2. Key Performance Indicators
3. Software Runtime Environment

Test Cases are complete building emulators. They not only include detailed physical models of buildings and their systems, but also all boundary condition data necessary to test a controller for the building. This includes weather, schedules, energy and fuel prices, occupant comfort definitions, and documentation. A test case is ultimately represented by a Functional Mockup Unit (FMU), known as the test case FMU. A repository of test cases of various building and system types is available at https://github.com/ibpsa/project1-boptest/tree/master/testcases.

Key Performance Indicators (KPI) are metrics to evaluate controller performance. A set of core KPIs have been identified and defined by the IBPSA Project WP1.2 team, which get calculated for every tested controller for each test case.

The Software Runtime Environment is designed to create a controlled testing environment that is efficiently deployed. It makes use of Docker containerization to create a well-defined simulation environment which contains Python modules to manage simulation of the test case FMU, deliver boundary condition forecasts, store results, calculate KPIs, and deliver information about the test case. This functionality is accessed by a user through a RESTful HTTP API.

The purpose of this tutorial is to provide a basic introduction of the BOPTEST framework and associated software components to potential test case developers and controller testers. Therefore, the tutorial is split into three main parts: I) Software Requirements, II) Test Case Development and III) Test Case Interaction.

More information at https://github.com/ibpsa/project1-boptest and in [1].

Table of Contents

[1] D. Blum, J. Arroyo, S. Huang, J. Drgona, F. Jorissen, H.T. Walnum, Y. Chen, K. Benne, D. Vrabie, M. Wetter, and L. Helsen. (2021). "Building optimization testing framework (BOPTEST) for simulation-based benchmarking of control strategies in buildings." *Journal of Building Performance Simulation*, 14(5), 586-610.

Part I: Software Requirements

The following is a list of software needed for this tutorial. Note that items 1 and 2 are only needed for test case development, and not for test case interaction or controller testing.

1. Modelica Buildings Library (7.0.0)
2. Dymola (2020x or 2021)
3. Python (≥2.7)
4. Docker
5. BOPTEST Repository Software
6. Cygwin (for Windows only)

Please follow the steps below to obtain links for downloading and installing the proper versions.

A. Download the Modelica Buildings Library Version 7.0.0 from https://simulationresearch.lbl.gov/modelica/downloads/archive/modelica-buildings.html. Extract the zip file to a directory location of your choice.

B. Install Dymola from https://www.3ds.com/products-services/catia/products/dymola/. A demo version will suffice for this tutorial.

C. Install Python (≥2.7) from https://www.python.org/downloads/ and the "pandas" package via pip.

D. Install Docker by following the instructions for your system:

   Windows:
   https://docs.docker.com/docker-for-windows/install/
   Ubuntu or Other Linux (see side panel at link):
   https://docs.docker.com/install/linux/docker-ce/ubuntu/
   MacOS:
   https://docs.docker.com/docker-for-mac/install/

Be sure to test your Docker installation by opening a terminal and running a test example container stored in an image on the Docker Hub (For Windows, try PowerShell. For Linux and MacOS, you may need to add sudo permission):

$ docker run hello-world

If you are using Windows and getting a permission error, you may have to add yourself to the "docker-users" group in computer management. See here for more information and how to resolve: https://icij.gitbook.io/datashare/faq/you-are-not-allowed-to-use-docker-you-must-be-in-the-docker-users-group-.-what-should-i-do

Additional information for Windows users can be found at a knowledge base established here: https://github.com/ibpsa/project1-boptest/issues/142.

E. Download the BOPTEST Repository Software v0.4.0 from https://github.com/ibpsa/project1-boptest. Extract the zip file to a directory location of your choice and add the root directory for the repository to your PYTHONPATH environmental variable.

Note: All path references throughout the tutorial will be made with reference to the root directory for this repository!

F. For Windows users only, install Cygwin from http://www.cygwin.com/.

Note: Make sure to include the "make" command when installing Cygwin (this one is not installed by default), and also to add "C:\cygwin64\bin" to your path environmental variable. Notice that the previous absolute path could vary depending on your choice of Cygwin version and installation folder.

Part II: Test Case Development

This part will step through the process of creating the test case FMU. It begins with an already-developed Modelica model of a simple single zone building defined by the BESTEST Case 900 envelope with heating and cooling provided by a fan coil unit. There are four main steps in this part:

1. Develop and document the test case building model. We will use Dymola as the development environment.
2. Add and configure signal exchange blocks to define inputs and outputs for interacting with an external controller to be tested, and variables needed for KPI calculations.
3. Create boundary condition data to include with the building model.
4. Compile the test case FMU.

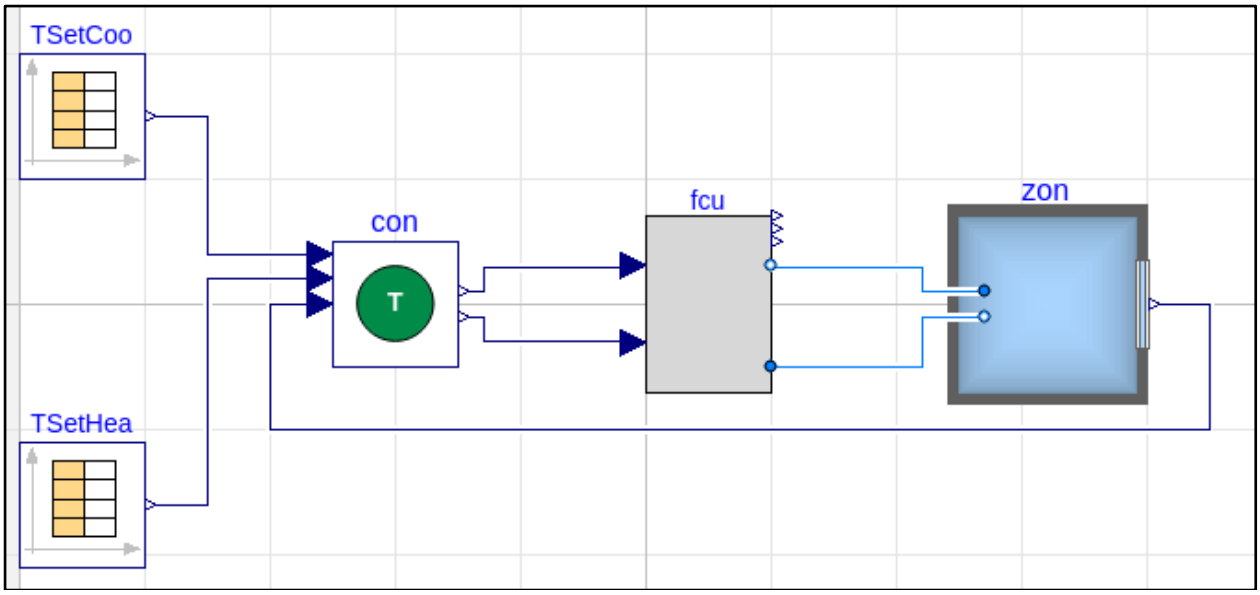
1. Develop and Document Test Case Building Model

For this tutorial, this step is already completed. However, let's take some time to get familiar with the building design and model.

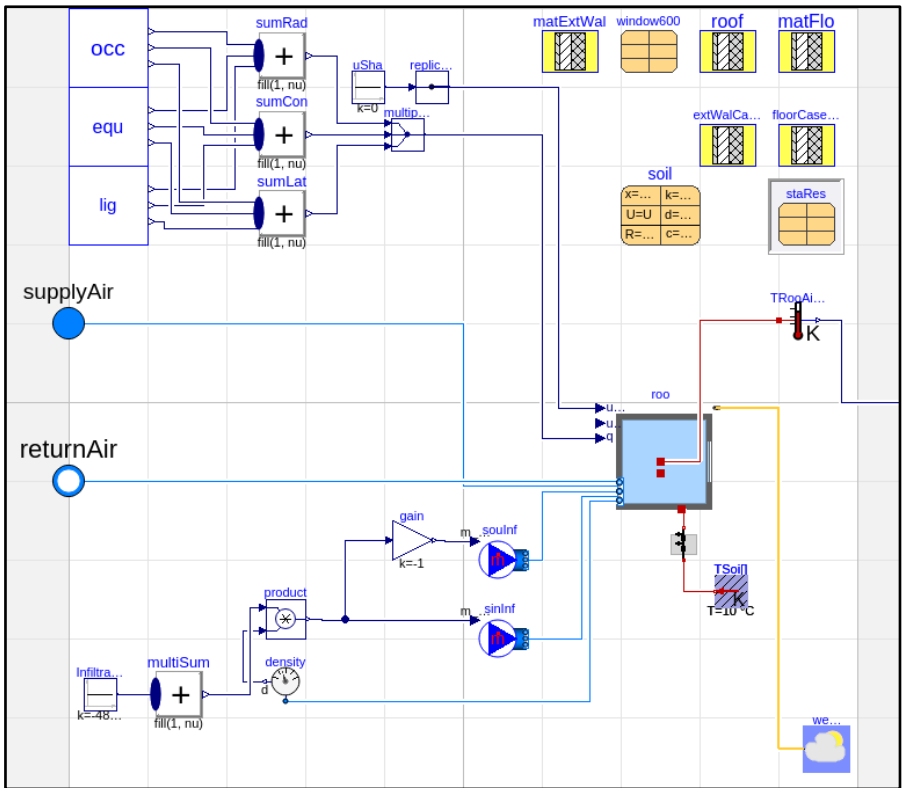
A. First, copy the folder `docs/tutorials/tutorial1_developer` to `testcases/tutorial1_developer`.

B. In Dymola, load the Modelica Buildings Library and the test case model at `testcases/tutorial1_developer/models/BESTESTAir/package.mo`.

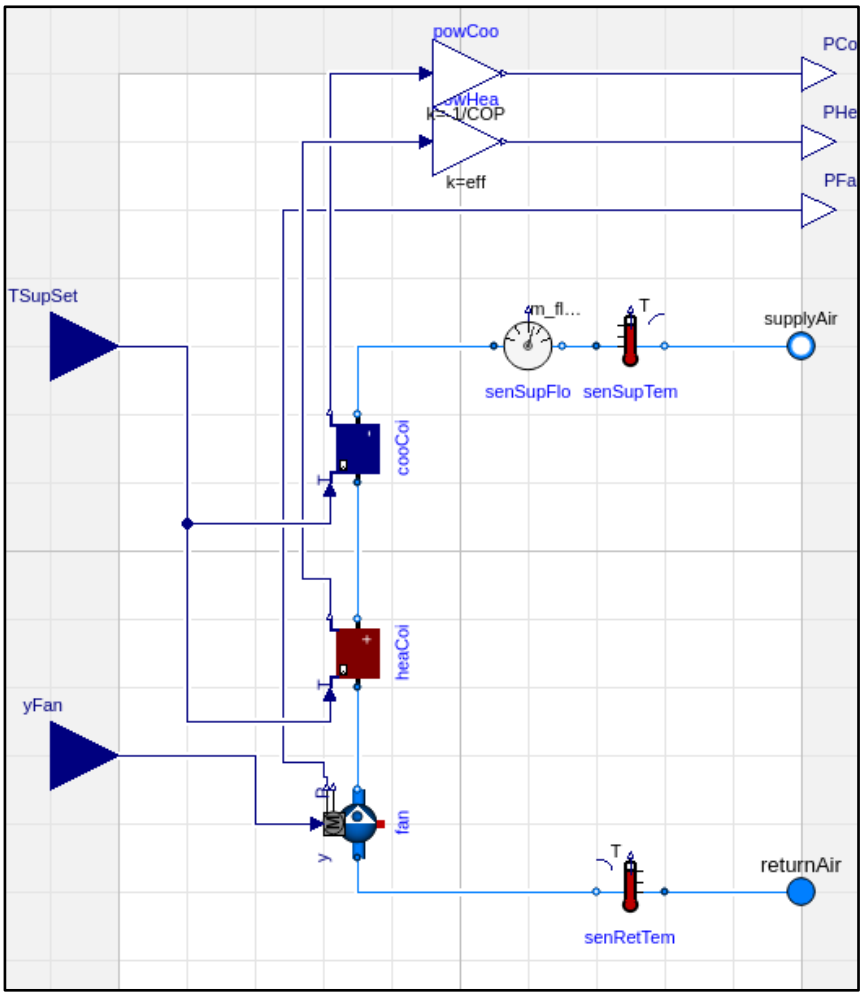
C. Open the model `BESTESTAir.TestCase` using Dymola. Notice the zone envelope model (named "zon"), fan coil unit model ("fcu"), controller ("con"), and zone temperature setpoint schedules for heating and cooling ("TSetHea" and "TSetCoo").



D. Open the model `BESTESTAir.Testcase.zon` (right-click on "zon" and select "Open Class in New Tab" in the drop-down menu) to investigate the zone heat balance implementation, materials and construction definitions, internal load models, infiltration model, weather reader, and supply and return air ports.

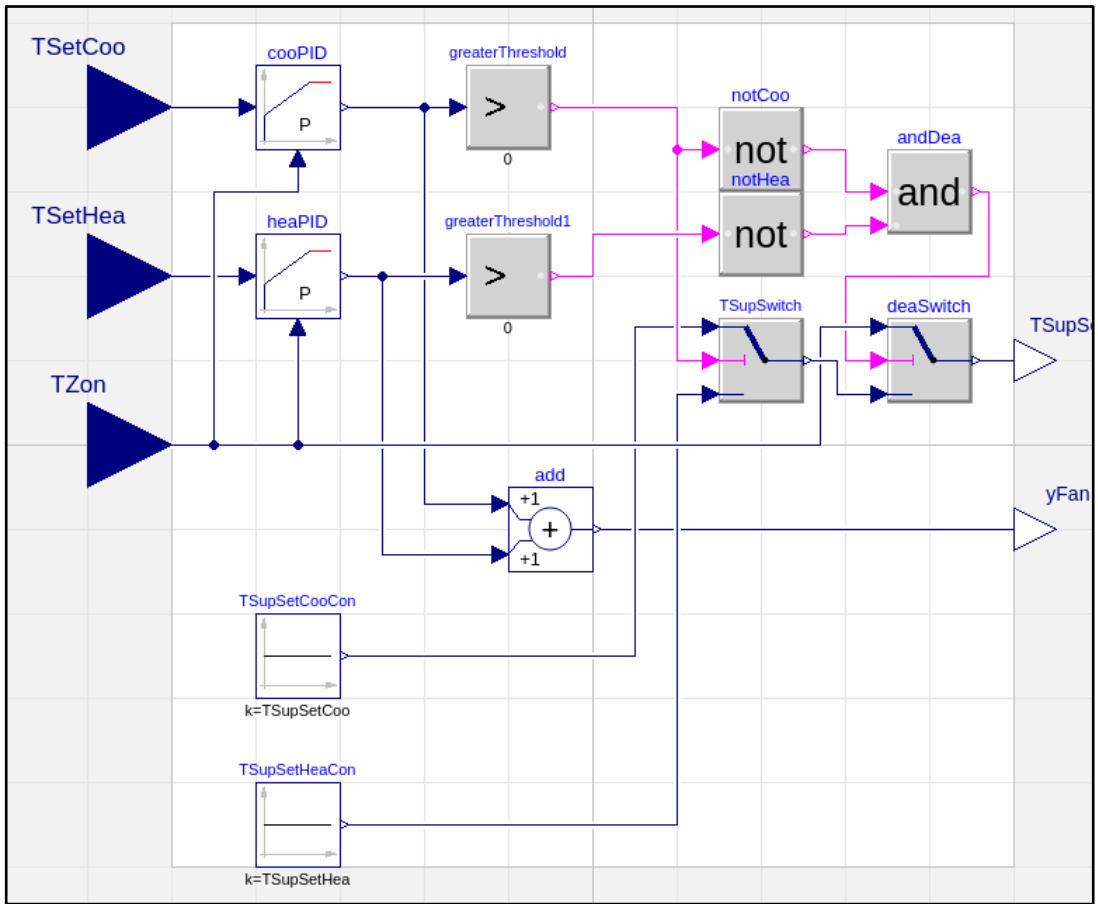


E. Open the model `BESTESTAir.Testcase.fcu` to investigate the models for supply fan ("fan"), heating coil ("heaCoi"), cooling coil ("cooCoi"), and supply and return air ports. For this fan coil unit model, cooling electrical power is calculated using the thermal load on the cooling coil, and a constant COP (defined in the "powCoo" model). Likewise, heating thermal power is calculated using the thermal load on the heating coil and a constant efficiency (defined in the "powHea" model). Fan electrical power is calculated using fan performance curves specified in the fan model.



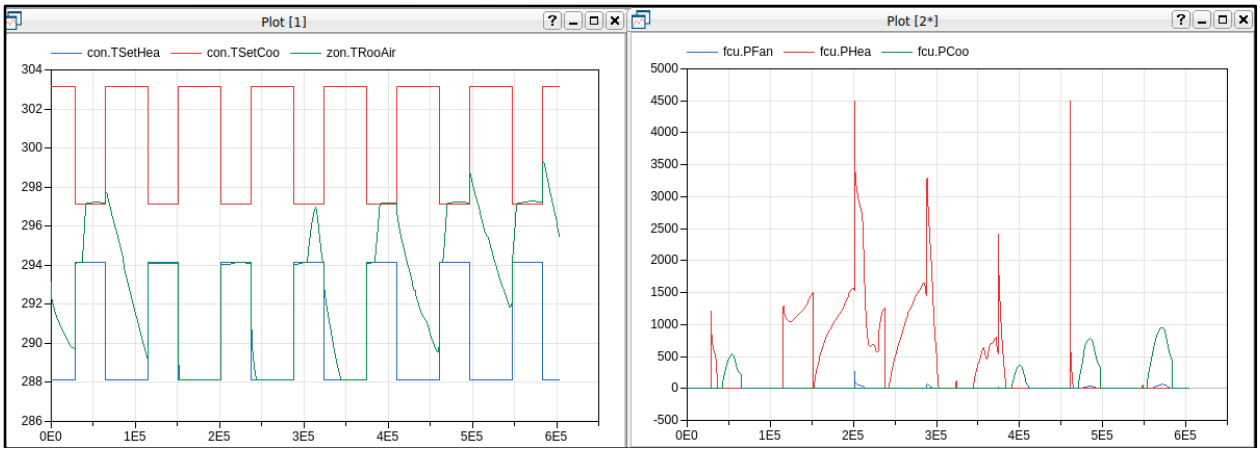
F. Open the model `BESTESTAir.Testcase.con` to investigate the logic used to control the fan speed and supply air temperature setpoint. In this controller model, the fan speed is controlled by proportional-only feedback controllers ("cooPID" and "heaPID") using zone temperature setpoints for heating and cooling and the measured zone air temperature. If the zone is

in heating mode, the heating supply air temperature setpoint is used. If the zone is in cooling mode, the cooling supply air temperature setpoint is used. If the zone is in deadband (neither heating nor cooling), the supply air temperature setpoint is set to the zone air temperature to indicate no heating and cooling required.



G. Double-click the models `BESTESTAir.Testcase.TSetCoo` and `BESTESTAir.Testcase.TSetHea` to investigate the heating and cooling setpoint schedules for the zone air temperature.

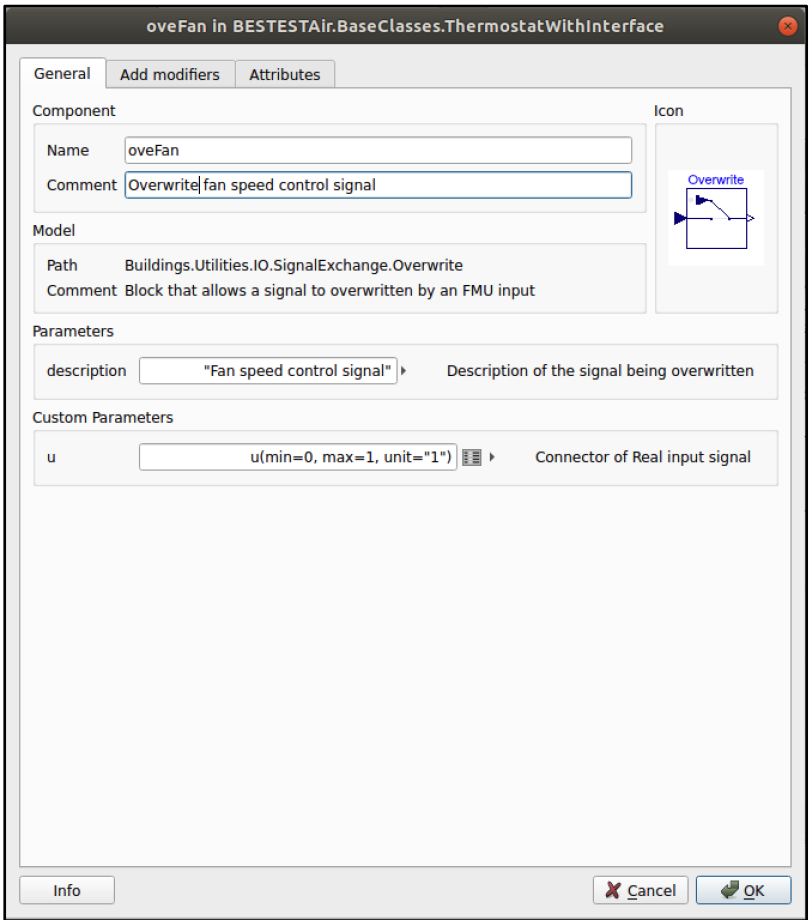
H. Simulate the model (not possible with Demo version) and notice the behavior of the FCU and controller to provide heating and cooling to the zone in order to maintain the zone temperature setpoints. To retrieve the figure below, go to the “Simulation” tab in the Dymola GUI, click “Commands” and select “TestCase.mos”.

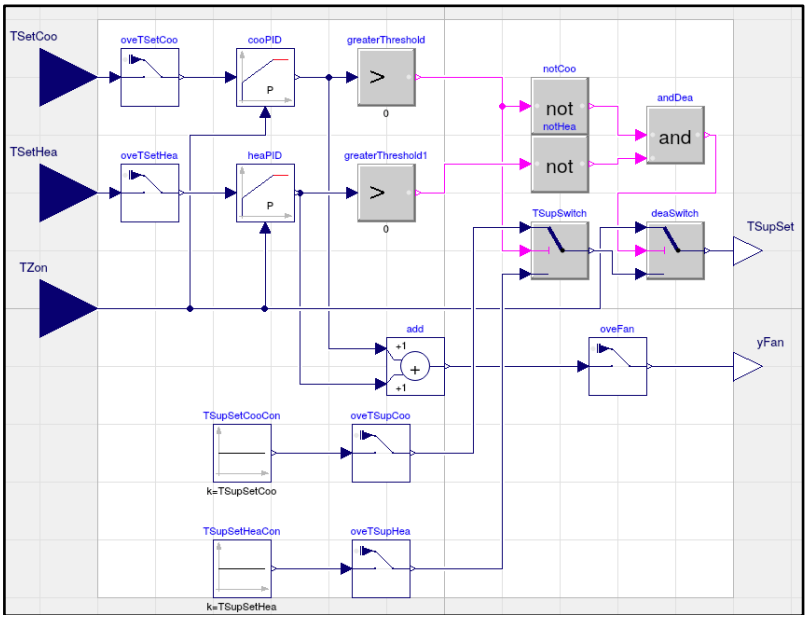


2. Add and Configure Signal Exchange Blocks

The previous Modelica model is stand-alone which does not have an interface to interact with an external controller to be tested. Therefore, in this step, we will modify the Modelica model to add the interface using "SignalExchange" blocks defined in Modelica Buildings Library. The blocks help identify model control signals that can be overwritten by an external controller, model measurements that can be read by an external controller, and model measurements that are needed for BOPTEST to calculate KPIs. For user's convenience, a complete, modified model with signal exchange blocks added is provided in the model `BESTESTAir.TestCase_Ideal`.

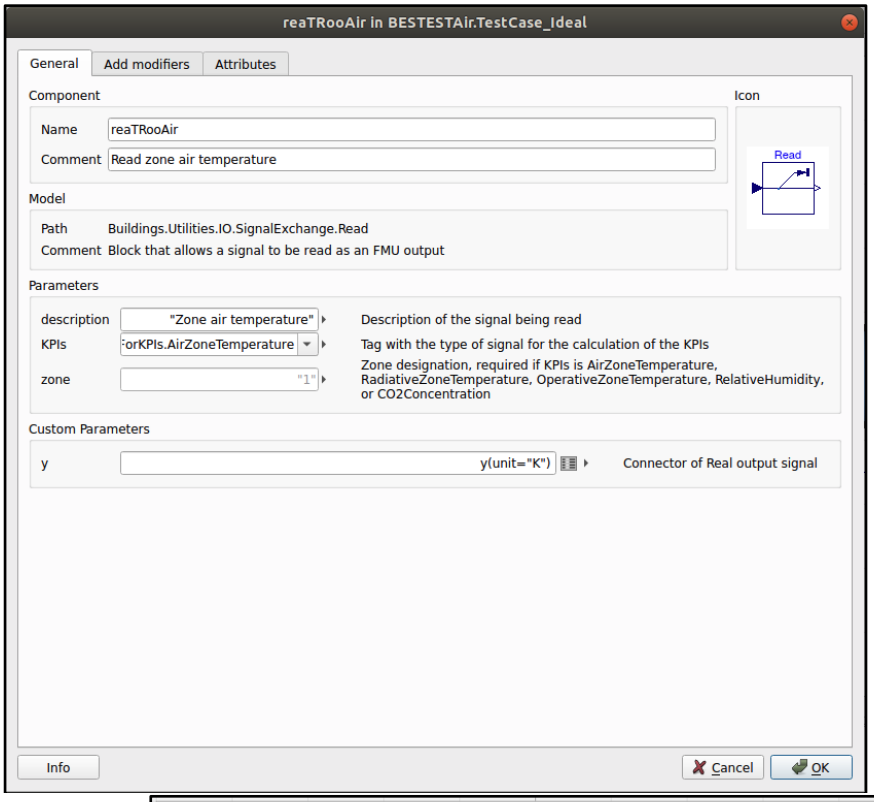
A. Use `Buildings.Utilities.IO.SignalExchange.Overwrite` to implement and configure signal exchange blocks for each control signal we'd like to be able to overwrite with an external controller. Configuration is done through editing the parameters of the block and defining min, max, and unit attributes to the block input, variable `u` (can use the "Add modifiers" tab for this). An example is shown below for `con.yFan`, where the overwrite block added is `oveFan`. Then, repeat this step for the variables in the table below.

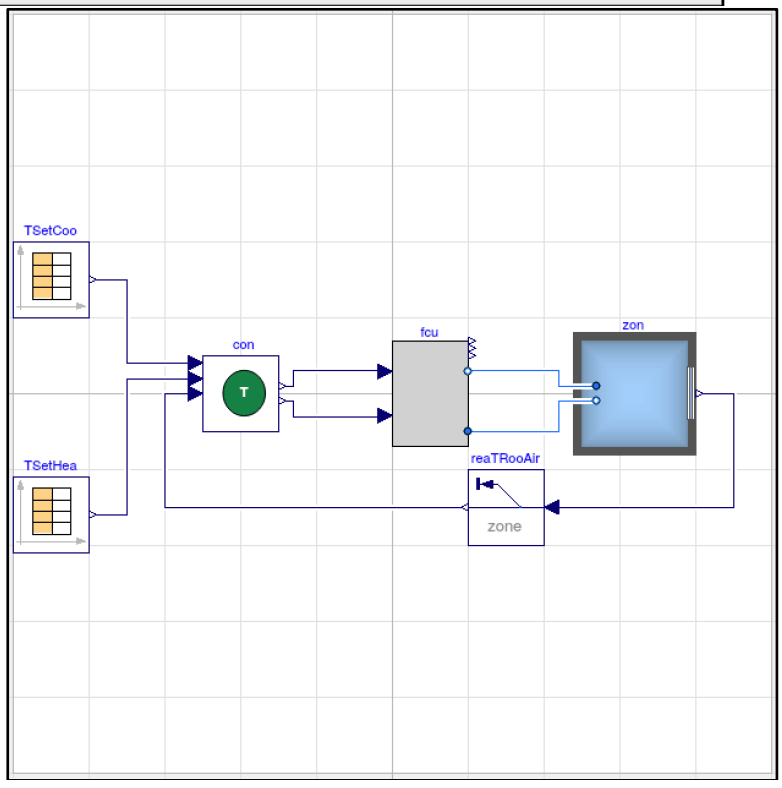

oveFan in BESTESTAir.BaseClasses.ThermostatWithInterface
General
Add modifiers
Attributes
Component
Name
oveFan
Comment
Overwrite fan speed control signal
Icon
Overwrite
Model
Path
Buildings.Utilities.IO.SignalExchange.Overwrite
Comment
Block that allows a signal to overwritten by an FMU input
Parameters
description
"Fan speed control signal"
Description of the signal being overwritten
Custom Parameters
u
u(min=0, max=1, unit="1")
Connector of Real input signal
Info
Cancel
OK

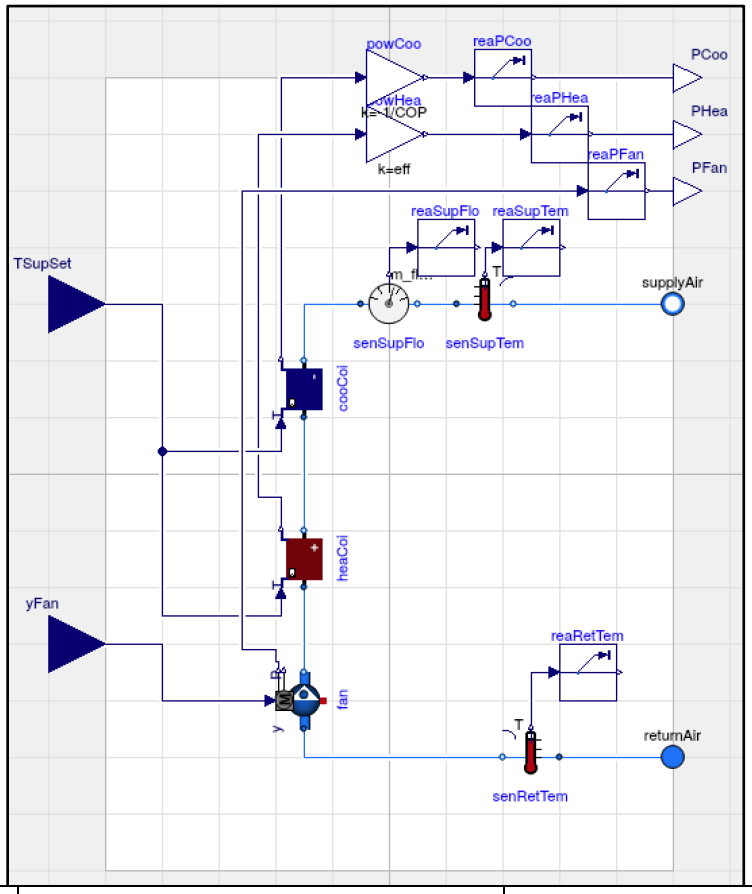

TSetCoo
oveTSetCoo
cooPID
greaterThreshold
notCoo
not
notHea
andDea
and
TSetHea
oveTSetHea
heaPID
greaterThreshold1
TSupSwitch
deaSwitch
TSupSet
TZon
add
oveFan
yFan
TSupSetCooCon
oveTSupCoo
k=TSupSetCoo
TSupSetHeaCon
oveTSupHea
k=TSupSetHea

Variable	Description	Min	Max	Unit
`con.yFan`	Fan speed control signal	0	1	“1”
`con.TSupSwitch.u1`	Supply air temperature setpoint for cooling	273.15+12	273.15+18	“K”
`con.TSupSwitch.u3`	Supply air temperature setpoint for heating	273.15+30	273.15+40	“K”
`con.TSetCoo`	Zone temperature setpoint for cooling	273.15+23	273.15+30	“K”
`con.TSetHea`	Zone temperature setpoint for heating	273.15+15	273.15+23	“K”

B. Use `Buildings.Utilities.IO.SignalExchange.Read` to implement and configure signal exchange blocks for each measurement signal we’d like to be able to read with an external controller. Configuration is done through editing the parameters of the block and defining unit attributes to the block output, variable `y`. One of these parameters can be used to identify the measurement as being needed for KPI calculation. An example is shown below for `zon.TRooAir`. Then, repeat this step for the variables in the table below. Note that specification of KPIs that require zone specifiers will require specification of a parameter called “zone” with a string value. Leaving as “1” is ok.


reaTRooAir in BESTESTAir.TestCase_Ideal
General
Add modifiers
Attributes
Component
Name
reaTRooAir
Comment
Read zone air temperature
Icon
Read
Model
Path
Buildings.Utilities.IO.SignalExchange.Read
Comment
Block that allows a signal to be read as an FMU output
Parameters
description
"Zone air temperature"
Description of the signal being read
KPIs
orKPIs.AirZoneTemperature
Tag with the type of signal for the calculation of the KPIs
zone
"1"
Zone designation, required if KPIs is AirZoneTemperature, RadiativeZoneTemperature, OperativeZoneTemperature, RelativeHumidity, or CO2Concentration
Custom Parameters
y
y(unit="K")
Connector of Real output signal
Info
Cancel
OK


TSetCoo
con
T
fcu
zon
TSetHea
reaTRooAir
zone



Variable	Description	KPI	Unit
`zon.TRooAir`	Zone air temperature	Air Zone Temperature	“K”
`fcu.PCoo`	Cooling electrical power consumption	Electric Power From Grid	“W”
`fcu.PHea`	Heating gas power consumption	Thermal Power From Natural Gas	“W”
`fcu.PFan`	Supply fan electrical power consumption	Electric Power From Grid	“W”
`fcu.senSupFlo.m_flow`	Supply air mass flowrate	None	“kg/s”
`fcu.senSupTem.T`	Supply air temperature	None	“K”
`fcu.senRetTem.T`	Return air temperature	None	“K”

3. Create Boundary Condition Data

This step will create CSV files that represent the necessary boundary condition data to run the test case and calculate KPIs, such as weather, internal load schedules, energy and fuel prices, and occupant comfort ranges. A Python module has been created to help with this process, `data/data_generator.py`. However, the CSV files can be created in any way if more customization is needed, likely for more complex test cases, as long as their column names adhere to the naming conventions defined by Section 4.6 of the BOPTEST Design Requirements and Guide (located in `docs/DesReqGui`), they are saved in the appropriate Resources directory, and they accurately represent the intended test case

A. Create the directory `testcases/tutorial1_developer/models/Resources`, which is used to store the boundary condition data. In this tutorial, the folder was already created for convenience.

B. Copy the weather file used for the test case into the new `Resources` directory. The weather file is located in the Modelica Buildings Library at `Buildings/Resources/weatherdata/DRYCOLD.mos`. In this tutorial, the weather file is already copied there for convenience.

C. Save the Python script below to `testcases/tutorial1_developer/models/generate_data.py`. The script generates the data using the BOPTEST `data/data_generator.py` module at an interval of 15 minutes. Note that the function arguments are based on the building model we explored earlier. We will use this module to create the following test case data:

    a. Weather
    b. Energy and fuel prices
    c. Energy and fuel emission factors
    d. Occupancy
    e. Internal heat gains
    f. Thermal comfort setpoint ranges
    g. IAQ upper limit for CO_2 ppm

```
from data.data_generator import Data_Generator
import os
```

```
# Set the location of the Resource directory relative to this file location
file_dir = os.path.dirname(os.path.realpath(__file__))
resources_dir = os.path.join(file_dir, 'Resources')
# Create data generator object with time interval to 15 minutes
gen = Data_Generator(resources_dir, period=900)
# Generate weather data from .mos in Resources folder with default values
gen.generate_weather()
# Generate prices data with default values
gen.generate_prices()
# Generate emission factors data with default values
gen.generate_emissions()
# Generate occupancy data for our case
gen.generate_occupancy(2,
                start_day_time = '08:00:00',
                end_day_time = '18:00:00')
# Generate internal gains data for our case
gen.generate_internalGains(start_day_time = '08:00:00',
                  end_day_time   = '18:00:00',
                  RadOcc = 10.325*48,
                  RadUnocc = 0.85*48,
                  ConOcc = 7.71667*48,
                  ConUnocc = 0.65*48,
                  LatOcc = 1.875*48,
                  LatUnocc = 0*48)
# Generates comfort range data for our case
gen.generate_setpoints(start_day_time = '08:00:00',
                end_day_time = '18:00:00',
                THeaOcc  = 21+273.15,
                THeaUnocc = 15+273.15,
                TCooOcc  = 24+273.15,
                TCooUnocc = 30+273.15)
```

The function `gen.generate_weather()` requires the compilation and simulation of a boundary condition model, specifically the model from the IBPSA Modelica library `IBPSA.BoundaryConditions.WeatherData.ReaderTMY3`. We will use a Docker image that has been developed with JModelica.org installed on

it to run the `generate_data.py` script.

D. First, we must build the Docker image we will use to generate the test case data. To do this, change the working directory to `testing/`. Then, use the following command:

```
$ make build_jm_image
```

This may take a minute or two. Check that the image built successfully by the command:

```
$ docker images
```

There should be a listed image called `jm`.

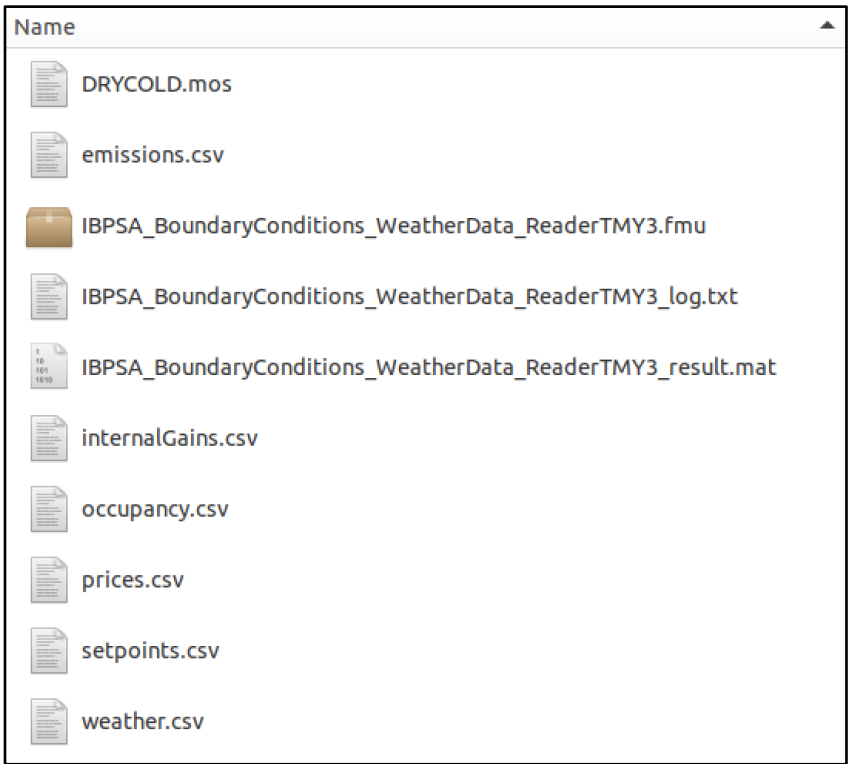
E. Now, to generate the data, change the working directory to `testing/` and use the following command:

```
$ make generate_testcase_data TESTCASE=tutorial1_developer
```

Note: that if an error occurred during generation of the test case data, before making any fixes and trying the above command again, run the following command:

```
$ docker stop jm
```

F. After running this command successfully, there should be nine new files within `testcases/tutorial1_developer/models/Resources`. The six which we wish to keep are `emissions.csv`, `prices.csv`, `occupancy.csv`, `weather.csv`, `ingernalGains.csv`, and `setpoints.csv`. Each csv file should have a time column to define the emulation time in seconds at 15 minute (900 second) intervals and the appropriate column header names according to the naming conventions defined by Section 4.6 of the BOPTEST Design Requirements and Guide (located in `docs/DesReqGui`). If no errors occurred, it is safe to delete the `.fmu`, `.txt`, and `.mat` files that were created.



4. Create JSON Files

We need to create three json files that provide additional information about the test case to the BOPTEST framework. They are the `config.json`, `days.json`, and `library_versions.json` each described in more detail below.

A. Create the file `testcases/tutorial1_developer/models/config.json` as shown below. The `config.json` file holds default configuration parameters for when we deploy and use the test case in the next section.

```
{
"name"        : "tutorial1_developer",
"area"        : 48.0,
"start_time"    : 0.0,
"warmup_period" : 0.0,
"step"        : 3600.0,
"scenario"     : {"electricity_price":"constant",
                "time_period":null}
}
```


B. Create the file `testcases/tutorial1_developer/models/days.json` as shown below. The `days.json` file specifies reference days during the year to define specific time period scenarios for tests, which can be set by the user during interaction with BOPTEST. A python module has been prepared in BOPTEST which defines time periods related to peak, typical, and mixed heating and cooling periods, located at `data/find_days.py`. This module can be used with annual simulation data from the test case in .csv format that contains system heating and cooling loads. For convenience in this tutorial, this data has been provided in `testcases/tutorial1_developer/models/heating_cooling.csv`. Finally, a script has been provided in this tutorial at `testcases/tutorial1_developer/models/find_days.py` that uses the BOPTEST algorithm and test case simulation data to create the `days.json`. This script is also shown below. This script can be run from the directory `testcases/tutorial1_developer/models/` using the command:

```
$ python find_days.py
```


```
{
"peak_heat_day": 341,
 "peak_cool_day": 289,
 "typical_heat_day": 92,
```

```
 "typical_cool_day": 135,
 "mix_day": 48
}
```

```
from data.find_days import find_days
import json

days = find_days(heat='fcu.heaCoi.Q_flow', cool='fcu.cooCoi.Q_flow', data='heating_cooling.csv')

with open('days.json', 'w') as f:
   json.dump(days, f)
```

C. Create the file `testcases/tutorial1_developer/models/library_versions.json` as shown below. The `library_versions.json` file specifies which version (specifically, Git commit) of dependent Modelica libraries to use when compiling the model into the test case FMU.

```
{
"BUILDINGS_COMMIT": "v7.0.0"
}
```

5. Compile Test Case FMU

This final step will compile the building model with signal exchange blocks, the boundary condition data, the `config.json`, and the `days.json` into our test case FMU, which will be ready for us to use for controller testing! Since the run-time environment Docker container has a Linux-based OS (Ubuntu), the test case FMU needs to be compiled with binaries suitable for simulation in Linux environments. Therefore, a Docker image and process has been developed for Windows users to be able to compile the test case FMU. Linux/MacOS users can also use this same process with Docker, or compile the test case FMU in a native environment. Such compilation in a native environment without Docker requires the BOPTEST `parsing/parser.py` module and the JModelica.org open source Modelica compiler to be installed, though these will not be detailed in this

tutorial.

D. Create a Python module as below and save it as `testcases/tutorial1_developer/models/compile_fmu.py`.

```
from parsing import parser

def compile_fmu():
  '''Compile the fmu.

  Returns
  -------
  fmupath : str
     Path to compiled fmu.

  '''

  # DEFINE MODEL
  mopath = 'BESTESTAir/package.mo'
  modelpath = 'BESTESTAir.TestCase_Ideal'

  # COMPILE FMU
  fmupath = parser.export_fmu(modelpath, [mopath])

  return fmupath

if __name__ == "__main__":
  fmupath = compile_fmu()
```

E. To compile the test case FMU, make sure the working directory is `testing/` and use the following command:

   ```
   $ make compile_testcase_model TESTCASE=tutorial1_developer
   ```

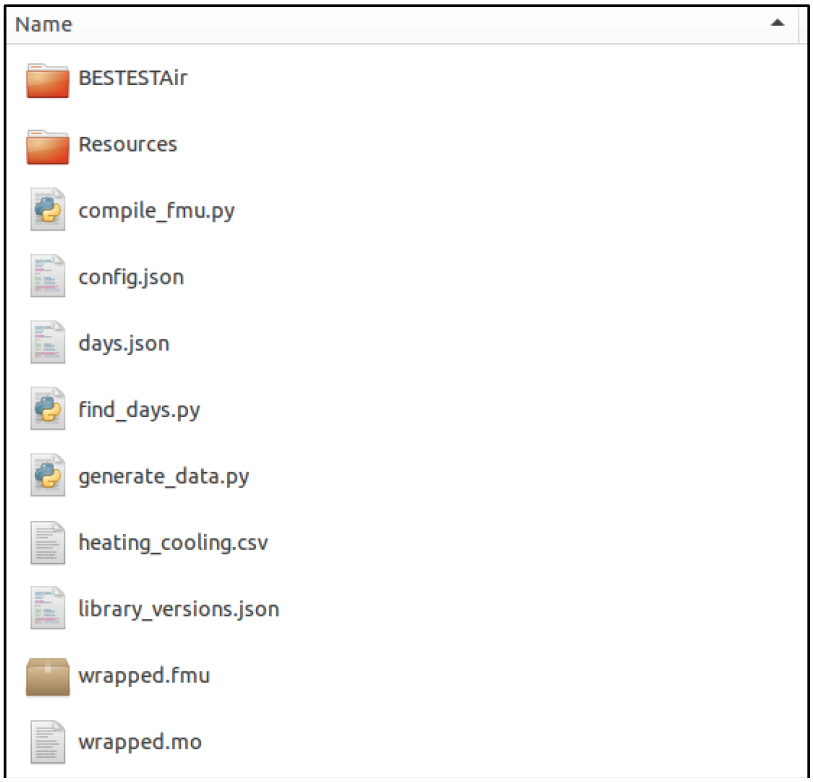
   Once complete, there should be a new file `testcases/tutorial1_developer/models/wrapped.fmu`, which is the test case FMU.

Note: that if an error occurred during compilation of the test case, before making any fixes and trying the above command again, run the following command:

```
$ docker stop jm
```

F. The final step in creating the test case is to add documentation. Create a directory `testcases/tutorial1_developer/doc` to hold any documentation. We will hold off on creating content to fill to this directory for this tutorial, however, create the directory nonetheless. Documentation should follow the requirements and template provided in Section 4.8 of the BOPTEST Design Requirements and Guide (located in `docs/DesReqGui`). Also look at `testcases/bestest_air/doc` for an example.

The final test case directory `testcases/tutorial1_developer/models`, including the test case FMU (`wrapped.fmu`), that will be needed for deployment in the next section look like (it is ok if there are extra files from FMU compilation):



Part III: Test Case Interaction

This part will step through how to deploy the software run-time environment with the test case, explore the ways in which we can interact with the test case using the HTTP RESTful API, and perform a simple example controller comparison test.

1. Deploy Test Case

This step will use Docker to deploy the test case we have just created.

Linux or macOS:
```
$ TESTCASE=tutorial1_developer docker-compose up
```
Windows PowerShell:
```
> ($env:TESTCASE="tutorial1_developer") -and (docker-compose up)
```

A couple notes:

- The first time this command is run, the image `boptest_base` will be built. This takes about a minute or so. Subsequent usage will use the already-built image and deploy much faster.
- If you update your BOPTEST repository, use the command `docker rmi boptest_base` to remove the image so it can be re-built with the updated repository upon next deployment.

The test case is successfully running if you see the following message in your terminal window:

```
* Serving Flask app "restapi" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

A. At this point, the test case is waiting for more instructions by listening to the `localhost:5000` port for HTTP RESTful API commands.

2. Using the HTTP RESTful API

The HTTP RESTful API commands for the test case are displayed in the Figure below. Let's create a Python script to demonstrate some interactions with our test case. We will use the `requests` package for this. In general, the test case can be utilized by any software that can send HTTP requests, we are only using a script written in Python here as an example.

Interaction	Request
Advance simulation with control input and receive measurements.	POST `advance` with optional json data "{<input_name>:}"
Initialize simulation to a start time using a warmup period in seconds. Also resets point data history and KPI calculations.	PUT `initialize` with required arguments `start_time=<value>`, `warmup_period=<value>`
Receive communication step in seconds.	GET `step`
Set communication step in seconds.	PUT `step` with required argument `step=<value>`
Receive sensor signal point names (y) and metadata.	GET `measurements`
Receive control signal point names (u) and metadata.	GET `inputs`
Receive test result data for the given point names between the start and final time in seconds.	PUT `results` with required arguments `point_names=<list of strings>`, `start_time=<value>`, `final_time=<value>`
Receive test KPIs.	GET `kpi`
Receive test case name.	GET `name`
Receive boundary condition forecast from current communication step for the given point names for the horizon and at the interval in seconds.	PUT `forecast` with required arguments `point_names=<list of strings>`, `horizon=<value>`, `interval=<value>`
Receive boundary condition forecast available point names and metadata.	GET `forecast_points`
Receive current test scenario.	GET `scenario`
Set test scenario. Setting the argument `time_period` performs an initialization with predefined start time and warmup period and will only simulate for predefined duration.	PUT `scenario` with optional arguments `electricity_price=<string>`, `time_period=<string>`. See README in /testcases for options and test case documentation for details.
Receive BOPTEST version.	GET `version`
Submit KPIs, other test information, and optional string tags (up to 10) to online dashboard. Requires a formal test scenario to be completed, initialized using the PUT `scenario` API.	POST `submit` with required argument `api_key=<string>` and optional arguments `tag#=<string>` where # is an integer between 1 and 10. The API key can be obtained from the user account registered with the online dashboard.

A. Create a new file from the code below in the root of the BOPTEST repository called `api_example.py`. Then, run the code and analyze the output. Try running the simulation for longer and/or plotting the results.

```
import requests
import pprint
pp = pprint.PrettyPrinter(indent=4)

# Set URL for testcase
url = 'http://127.0.0.1:5000'


# DEMONSTRATE API FOR GETTING TEST CASE INFORMATION
# -------------------------------------------
pp.pprint('\nTEST CASE INFORMATION\n----------------------')
# Get the test case name
name = requests.get('{0}/name'.format(url)).json()
print('\nName is:http://0.0.0.0:5000/')
pp.pprint(name)
# Inputs available
inputs = requests.get('{0}/inputs'.format(url)).json()
print('\nControl Inputs are:')
pp.pprint(inputs)
# Measurements available
measurements = requests.get('{0}/measurements'.format(url)).json()
print('\nMeasurements are:')
# Forecasts available
measurements = requests.get('{0}/forecast_points'.format(url)).json()
print('\Forecast points are:')
pp.pprint(measurements)
# Default simulation step
step_def = requests.get('{0}/step'.format(url)).json()
print('\nDefault Simulation Step is:\t{0}'.format(step_def))

# DEMONSTRATE API FOR SIMULATING TEST CASE
# ----------------------------------------
# Set simulation step
print('\nSetting simulation step to 3600.')
res = requests.put('{0}/step'.format(url), json={'step':3600})
print(res)
# Advance simulation
print('\nAdvancing emulator one step...')
y = requests.post('{0}/advance'.format(url), json={}).json()
print('\nMeasurements at next step are:')
pp.pprint(y)
# Get a forecast of boundary conditions
print('\Fetching forecast...')
w = requests.put('{0}/forecast'.format(url), data={'point_names':['TDryBul'], 'horizon':3600*6,
'interval':3600}).json()
print('Forecast is:')
pp.pprint(w)

# DEMONSTRATE API FOR VIEWING RESULTS
# -----------------------------------
# Get simulation results for the point TrooAir_y
data = requests.put('{0}/results'.format(url), json={'point_names':['reaTRooAir_y'],
'start_time':0, 'final_time':7200}).json()
print('\nSimulation results are:')
pp.pprint(data)
# Report KPIs
kpi = requests.get('{0}/kpi'.format(url)).json()
print('\nKPIs are:')
```

```
pp.pprint(kpi)

# Reset test case to beginning
print('\nResetting test case to beginning.')
res = requests.put('{0}/initialize'.format(url), json={'start_time':0, 'warmup_period':0})
print(res)
```

3. Example Controller Test

Let's now edit the `api_example.py` script developed in the previous section to simulate the model for 1 day (24 steps of 3600 seconds) and implement control signals to overwrite those within the model. We will implement new zone heating and cooling setpoint temperatures that are constant and make a narrower deadband range. Save the Python code below to a file named `control_example.py` and run it.

Feel free to edit control signals or parameters to explore how the KPIs and results change. Try implementing a feedback controller using the measurements that are taken at each timestep. Remember that stability may require the use of a smaller control time step.

```
import requests
import pprint
pp = pprint.PrettyPrinter(indent=4)

# Set URL for testcase
url = 'http://127.0.0.1:5000'


# DEMONSTRATE API FOR GETTING TEST CASE INFORMATION
# -------------------------------------------
pp.pprint('\nTEST CASE INFORMATION\n---------------------')
# Get the test case name
name = requests.get('{0}/name'.format(url)).json()['payload']
print('\nName is:http://0.0.0.0:5000/')
pp.pprint(name)
# Inputs available
inputs = requests.get('{0}/inputs'.format(url)).json()['payload']
print('\nControl Inputs are:')
pp.pprint(inputs)
# Measurements available
measurements = requests.get('{0}/measurements'.format(url)).json()['payload']
print('\nMeasurements are:')
pp.pprint(measurements)
# Default simulation step
step_def = requests.get('{0}/step'.format(url)).json()['payload']
print('\nDefault Simulation Step is:\t{0}'.format(step_def))

# DEMONSTRATE API FOR SIMULATING TEST CASE
# ----------------------------------------
# Set simulation step
```

```
print('\nSetting simulation step to 3600.')
res = requests.put('{0}/step'.format(url), json={'step':3600}).json()['payload']
print(res)
# Advance simulation in loop
for i in range(24):
    u = {'con_oveTSetHea_u':273.15+22, # Set zone heating setpoint to 22 C
         'con_oveTSetHea_activate':1, # Activate overwrite of zone heating setpoint
         'con_oveTSetCoo_u':273.15+23, # Set zone cooling setpoint to 23 C
         'con_oveTSetCoo_activate':1 # Activate overwrite of zone cooling setpoint
         }
    print('\nAdvancing emulator one step with control data...')
    y = requests.post('{0}/advance'.format(url), json=u).json()['payload']
    print('\nMeasurements at next step are:')
    pp.pprint(y)
    # Get a forecast of boundary conditions
    print('\nFetching forecast...')
    w = requests.put('{0}/forecast'.format(url), json={'point_names':['TDryBul'],
'horizon':3600*6, 'interval':3600}).json()['payload']
    print('Got forecast.')


# DEMONSTRATE API FOR VIEWING RESULTS
# -----------------------------------
# Get simulation results for the point TrooAir_y
data = requests.put('{0}/results'.format(url), json={'point_names':['reaTRooAir_y'],
'start_time':0, 'final_time':86400}).json()['payload']
print('\nSimulation results are:')
pp.pprint(data)
# Report KPIs
kpi = requests.get('{0}/kpi'.format(url)).json()['payload']
print('\nKPIs are:')
pp.pprint(kpi)

# Reset test case to beginning
print('\nResetting test case to beginning.')
res = requests.put('{0}/initialize'.format(url), json={'start_time':0,
'warmup_period':0}).json()['payload']
pp.pprint(res)
```