# Instructions to use Pyomo and Gurobi

## What is Pyomo?

Pyomo is a Python-based, open-source optimization modeling language with a diverse set of optimization capabilities. All the required resources can be accessed through its website: *http://www.pyomo.org/*

## Installation

Instructions to install Pyomo can be found here: *http://www.pyomo.org/installation*. Simply execute the following command in the shell once Python is installed on the machine:

```
pip install --user pyomo
```

## Solvers

Pyomo does not include any stand-alone optimization solvers. Solvers to analyze optimization models built with Pyomo need to be installed separately. The user environment needs to be configured in a way that Pyomo can detect and execute the third-party solvers. Instructions are given here for a range of different solvers: *https://software.sandia.gov/downloads/pub/pyomo/PyomoInstallGuide.html#Solvers*
and here: *http://numberjack.ucc.ie/doc/install.html#building-additional-solver-interfaces*

The optimizer I used is called Gurobi. The GUROBI_HOME and GRB_LICENSE_FILE environment variables need to be set properly so that Pyomo can find the Gurobi install location (see link above).
Download Gurobi from this website: *http://www.gurobi.com/downloads/download-center*
Gurobi can also be installed into Anaconda (`conda install gurobi`):
*https://www.gurobi.com/documentation/6.5/quickstart_mac/installing_the_anaconda_py.html*

Gurobi can be used by faculty, staff, or students at a degree-granting academic institution for free for one year with an academic license, see here: *https://user.gurobi.com/download/licenses/free-academic*
For anyone interested in learning more about the algorithms used by Gurobi to solve Mixed-Integer Programming problems, we refer to the following documentation:
*http://www.gurobi.com/resources/getting-started/mip-basics*

## Getting started

The Pyomo documentation is very useful to get started:
*https://software.sandia.gov/downloads/pub/pyomo/PyomoOnlineDocs.html#_pyomo_overview*
Very simplified, the model consists of decision variable, parameter data, an objective expression that is either minimized or maximized and one or several constraints (restrictions on variable values).
A Python interface allows us to use Gurobi directly without having to install Pyomo, see here:
*http://www.gurobi.com/documentation/6.5/quickstart_windows/py_python_interface.html*
However, Pyomo is recommended as it allows to very easily test out different solvers without having to make huge modifications to the main code.

## Forum

Pyomo has a very useful and responsive forum: *https://groups.google.com/forum/#!forum/pyomo-forum*
Gurobi's discussion board can be found here: *http://groups.google.com/group/gurobi*
Check if your question has already been answered there before you create a new topic.

## Citation

Please do not forget to cite Pyomo if you use it for your work: *http://www.pyomo.org/citing-pyomo*
The BibTeX citation for Gurobi is given here (question 10): *http://www.gurobi.com/support/faqs*

*Nadja Herger, UNSW, 2016 (nadja.herger@student.unsw.edu.au)*

## Sample script

Here, I share a sample script to show how easy it is to formulate and solve optimization models using the optimization modeling language Pyomo and the solver Gurobi.
The aim is to select K model runs from a total number of N model runs so that the mean of the climatologies of those K simulations minimises the RMSE compared to a given observational climatology field.

```
# ------------------------------------------------------------------
#   THIS SCRIPT CONTAINS THE MAIN COMPONENTS OF PYOMO IN ORDER TO FIND
#   THE SUBSET (FROM THE MULTI-MODEL ENSEMBLE) WHICH MINIMIZES THE RMSE
#
#    Nadja Herger, UNSW, 2016 (nadja.herger@student.unsw.edu.au)
# ------------------------------------------------------------------

#####################################################
## Import necessary modules
#####################################################
from pyomo.environ import *
from pyomo.opt import TerminationCondition
import numpy as np

#####################################################
## Load your data (here: randomly create arrays)
#####################################################
# Define constants
solver = 'gurobi' # Use solver Gurobi
K_models = 10 # Size of the subset
N_models = 20 # Total number of available model runs
nlat = 36; lat = np.linspace(-90,90,nlat)
nlon = 72

# Create arrays of model climatology, observation climatology and mask
model_clim = np.random.rand(N_models,nlat,nlon)
obs_clim = np.random.rand(nlat,nlon)
obs_mask = np.array(np.random.rand(nlat,nlon)>0.5)

# Create array which contains area-weights
wgtmat = np.cos(np.tile(abs(lat[:,None])*np.pi/180, (1,nlon)))
masked_wgtmat = np.ma.array(wgtmat * ~obs_mask, mask = obs_mask)

#####################################################
## Vectorize model and observation climatology array
#####################################################
v_model = model_clim[:, ~obs_mask]  # ( N_models x time-space-dimension )
v_obs = obs_clim[~obs_mask]         # ( 1 x time-space-dimension )
v_wgtmat = wgtmat[~obs_mask]        # ( 1 x time-space-dimension )

#####################################################
## Define the solver
#####################################################
opt = SolverFactory(solver)

#####################################################
## Define the concrete model
#####################################################
model = ConcreteModel()
model.x = Var(range(N_models), domain=Boolean)
model.Constraint1 = Constraint(expr = summation(model.x) == K_models)
model.OBJ = Objective(expr = np.sum((v_obs - (np.sum([v_model[i,:] * model.x[i]
for i in range(N_models)],axis=0) / (K_models)))**2 * v_wgtmat ) /
(np.sum(v_wgtmat)))
#####################################################
## Extract the results
#####################################################
results = opt.solve(model, load_solutions=False)
if results.solver.termination_condition == TerminationCondition.optimal:
    model.solutions.load_from(results)
    solution = [model.x[i].value for i in range(N_models)]# list with 0s and 1s
    ensmember = np.where(np.array(solution)==1.0)[0]
else:
    print '!!! The solution part of the results object has not been loaded into
the model.'

mse_min = results.solution.objective.values()[0]['Value']
```

Some more information on the main components of the model:

- `model.x = Var(range(N_models), domain=Boolean)`
  The vector *x* is our variable vector. Its length is the number of available model runs, *N_models*. The elements of the variables are of type boolean (1.0 or 0.0). Model simulations with variable value 1.0 are going to be part of the optimal ensemble.

- `model.Constraint1 = Constraint(expr = summation(model.x) == K_models)`
  This line defines the constraint. We constrain the elements in the variable vector (*model.x*) to sum up to *K_models*, which is the size of our model subset.

- `model.OBJ = Objective(expr = np.sum((v_obs - (np.sum([v_model[i,:] * model.x[i] for i in range(N_models)],axis=0) / (K_models)))**2 * v_wgtmat ) / (np.sum(v_wgtmat)))`
  Our objective/cost function which we are trying to minimize is defined within *Objective()*. What it essentially does it minimize the mean squared error (MSE). Minimizing a function which describes the MSE leads to the same solution as minimizing a root mean squared error (RMSE) function.
  If you are interested in minimizing or maximising any other cost function, you would need to modify the *Objective()* function.

- `solution = [model.x[i].value for i in range(N_models)]`
  The list called *solution* might look something like this:
  [0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0]
  It has the length *N_models* and the values 1.0 indicate that this particular model run is part of the optimal ensemble. There are a total number of *K_models* values of 1.0, as dictated by our constraint.

- `ensmember = np.where(np.array(solution)==1.0)[0]`
  This is our array of the indices of model runs which are part of the optimal ensemble. For the example above, that array would look as follows: array([ 1,  4,  7,  8,  9, 11, 12, 13, 18, 19]). Remember that Python's indexing starts with the value 0.

- `mse_min = results.solution.objective.values()[0]['Value']`
  *mse_min* is the minimum MSE value that Pyomo managed to find using the ensemble members stored in the array *ensmember*. It is the optimal objective function value.