PySP Documentation

Release 6.0.0

PySP

May 26, 2021

CONTENTS

Support for solving multi-stage stochastic programs based on a scenario discretization of the uncertainty.

CHAPTER

ONE

STOCHASTIC PROGRAMMING

To express a stochastic program in PySP, the user specifies both the deterministic base model and the scenario tree model with associated uncertain parameters. Both concrete and abstract model representations are supported.

Given the deterministic and scenario tree models, PySP provides multiple paths for the solution of the corresponding stochastic program. One alternative involves forming the extensive form and invoking an appropriate deterministic solver for the entire problem once. For more complex stochastic programs, we provide a generic implementation of Rockafellar and Wets' Progressive Hedging algorithm, with additional specializations for approximating mixed-integer stochastic programs as well as other decomposition methods. By leveraging the combination of a high-level programming language (Python) and the embedding of the base deterministic model in that language (Pyomo), we are able to provide completely generic and highly configurable solver implementations.

This section describes PySP: (Pyomo Stochastic Programming), where parameters are allowed to be uncertain.

1.1 Overview of Modeling Components and Processes

The sequence of activities is typically the following:

- Create a deterministic model and declare components
- Develop base-case data for the deterministic model
- Test, verify and validate the deterministic model
- Model the stochastic processes
- Develop a way to generate scenarios (in the form of a tree if there are more than two stages)
- Create the data files need to describe the stochastics
- Use PySP to solve stochastic problem

When viewed from the standpoint of file creation, the process is

- Create an abstract model for the deterministic problem in a file called `ReferenceModel.py`
- Specify the stochastics in a file called `ScenarioStructure.dat`
- Specify scenario data

1.2 Birge and Louveaux's Farmer Problem

Birge and Louveaux [BirgeLouveauxBook] make use of the example of a farmer who has 500 acres that can be planted in wheat, corn or sugar beets, at a per acre cost of 150, 230 and 260 (Euros, presumably), respectively. The farmer needs to have at least 200 tons of wheat and 240 tons of corn to use as feed, but if enough is not grown, those crops can be purchased for 238 and 210, respectively. Corn and wheat grown in excess of the feed requirements can be sold for 170 and 150, respectively. A price of 36 per ton is guaranteed for the first 6000 tons grown by any farmer, but beets in excess of that are sold for 10 per ton. The yield is 2.5, 3, and 20 tons per acre for wheat, corn and sugar beets, respectively.

1.2.1 ReferenceModel.py

So far, this is a deterministic problem because we are assuming that we know all the data. The Pyomo model for this problem shown here is in the file `ReferenceModel.py` in the sub-directory `examples/farmer/models` that is distributed with PySP.

```
#  _________________________________________________________________________
#
#  Pyomo: Python Optimization Modeling Objects
#  Copyright 2017 National Technology and Engineering Solutions of Sandia, LLC
#  Under the terms of Contract DE-NA0003525 with National Technology and
#  Engineering Solutions of Sandia, LLC, the U.S. Government retains certain
#  rights in this software.
#  This software is distributed under the 3-clause BSD License.
#  _________________________________________________________________________

# Farmer: rent out version has a scalar root node var
# note: this will minimize
#
# Imports
#

from pyomo.core import *

#
# Model
#

model = AbstractModel()

#
# Parameters
#

model.CROPS = Set()

model.TOTAL_ACREAGE = Param(within=PositiveReals)

model.PriceQuota = Param(model.CROPS, within=PositiveReals)

model.SubQuotaSellingPrice = Param(model.CROPS, within=PositiveReals)
```

(continues on next page)

(continued from previous page)

```
def super_quota_selling_price_validate (model, value, i):
    return model.SubQuotaSellingPrice[i] >= model.SuperQuotaSellingPrice[i]

model.SuperQuotaSellingPrice = Param(model.CROPS, validate=super_quota_selling_price_
↪validate)

model.CattleFeedRequirement = Param(model.CROPS, within=NonNegativeReals)

model.PurchasePrice = Param(model.CROPS, within=PositiveReals)

model.PlantingCostPerAcre = Param(model.CROPS, within=PositiveReals)

model.Yield = Param(model.CROPS, within=NonNegativeReals)

#
# Variables
#

model.DevotedAcreage = Var(model.CROPS, bounds=(0.0, model.TOTAL_ACREAGE))

model.QuantitySubQuotaSold = Var(model.CROPS, bounds=(0.0, None))
model.QuantitySuperQuotaSold = Var(model.CROPS, bounds=(0.0, None))

model.QuantityPurchased = Var(model.CROPS, bounds=(0.0, None))

#
# Constraints
#

def ConstrainTotalAcreage_rule(model):
    return summation(model.DevotedAcreage) <= model.TOTAL_ACREAGE

model.ConstrainTotalAcreage = Constraint(rule=ConstrainTotalAcreage_rule)

def EnforceCattleFeedRequirement_rule(model, i):
    return model.CattleFeedRequirement[i] <= (model.Yield[i] * model.DevotedAcreage[i])␣
↪+ model.QuantityPurchased[i] - model.QuantitySubQuotaSold[i] - model.
↪QuantitySuperQuotaSold[i]

model.EnforceCattleFeedRequirement = Constraint(model.CROPS,␣
↪rule=EnforceCattleFeedRequirement_rule)

def LimitAmountSold_rule(model, i):
    return model.QuantitySubQuotaSold[i] + model.QuantitySuperQuotaSold[i] - (model.
↪Yield[i] * model.DevotedAcreage[i]) <= 0.0

model.LimitAmountSold = Constraint(model.CROPS, rule=LimitAmountSold_rule)

def EnforceQuotas_rule(model, i):
    return (0.0, model.QuantitySubQuotaSold[i], model.PriceQuota[i])

model.EnforceQuotas = Constraint(model.CROPS, rule=EnforceQuotas_rule)
```

(continues on next page)

(continued from previous page)

```
#
# Stage-specific cost computations
#

def ComputeFirstStageCost_rule(model):
    return summation(model.PlantingCostPerAcre, model.DevotedAcreage)

model.FirstStageCost = Expression(rule=ComputeFirstStageCost_rule)

def ComputeSecondStageCost_rule(model):
    expr = summation(model.PurchasePrice, model.QuantityPurchased)
    expr -= summation(model.SubQuotaSellingPrice, model.QuantitySubQuotaSold)
    expr -= summation(model.SuperQuotaSellingPrice, model.QuantitySuperQuotaSold)
    return expr

model.SecondStageCost = Expression(rule=ComputeSecondStageCost_rule)

#
# PySP Auto-generated Objective
#
# minimize: sum of StageCosts
#
# An active scenario objective equivalent to that generated by PySP is
# included here for informational purposes.
def total_cost_rule(model):
    return model.FirstStageCost + model.SecondStageCost
model.Total_Cost_Objective = Objective(rule=total_cost_rule, sense=minimize)
```

1.2.2 Example Data

The data introduced here are in the file AverageScenario.dat in the sub-directory `examples/farmer/scenariodata` that is distributed with Pyomo. These data are given for illustration. The file ReferenceModel.dat is not required by PySP.

```
# "mean" scenario

set CROPS := WHEAT CORN SUGAR_BEETS ;

param TOTAL_ACREAGE := 500 ;

# no quotas on wheat or corn
param PriceQuota := WHEAT 100000 CORN 100000 SUGAR_BEETS 6000 ;

param SubQuotaSellingPrice := WHEAT 170 CORN 150 SUGAR_BEETS 36 ;

param SuperQuotaSellingPrice := WHEAT 0 CORN 0 SUGAR_BEETS 10 ;

param CattleFeedRequirement := WHEAT 200 CORN 240 SUGAR_BEETS 0 ;
```

(continues on next page)

(continued from previous page)

```
# can't purchase beets (no real need, as cattle don't eat them)
param PurchasePrice := WHEAT 238 CORN 210 SUGAR_BEETS 100000 ;

param PlantingCostPerAcre := WHEAT 150 CORN 230 SUGAR_BEETS 260 ;

param Yield := WHEAT 2.5 CORN 3 SUGAR_BEETS 20 ;
```

Any of these data could be modeled as uncertain, but we will consider only the possibility that the yield per acre could be higher or lower than expected. Assume that there is a probability of 1/3 that the yields will be the average values that were given (i.e., wheat 2.5; corn 3; and beets 20). Assume that there is a 1/3 probability that they will be lower (2, 2.4, 16) and 1/3 probability they will be higher (3, 3.6, 24). We refer to each full set of data as a *scenario* and collectively we call them a *scenario tree*. In this case the scenario tree is very simple: there is a root node and three leaf nodes: one corresponding to each scenario. The acreage-to-plant decisions are root node decisions because they must be made without knowing what the yield will be. The other variables are so-called *second stage* decisions, because they will depend on which scenario is realized.

1.2.3 ScenarioStructure.dat

PySP requires that users describe the scenario tree using specific constructs in a file named `ScenarioStructure.dat`; for the farmer problem, this file can be found in the pysp sub-directory `examples/farmer/scenariodata` that is distributed with PySP.

```
# IMPORTANT - THE STAGES ARE ASSUMED TO BE IN TIME-ORDER.

set Stages := FirstStage SecondStage ;

set Nodes := RootNode
             BelowAverageNode
             AverageNode
             AboveAverageNode ;

param NodeStage := RootNode         FirstStage
                   BelowAverageNode SecondStage
                   AverageNode      SecondStage
                   AboveAverageNode SecondStage ;

set Children[RootNode] := BelowAverageNode
                          AverageNode
                          AboveAverageNode ;

param ConditionalProbability := RootNode          1.0
                                BelowAverageNode  0.33333333
                                AverageNode       0.33333334
                                AboveAverageNode  0.33333333 ;

set Scenarios := BelowAverageScenario
                 AverageScenario
                 AboveAverageScenario ;
```

(continues on next page)

(continued from previous page)

```
param ScenarioLeafNode :=
                    BelowAverageScenario BelowAverageNode
                    AverageScenario      AverageNode
                    AboveAverageScenario AboveAverageNode ;

set StageVariables[FirstStage] :=  DevotedAcreage[*];

set StageVariables[SecondStage] := QuantitySubQuotaSold[*]
                                   QuantitySuperQuotaSold[*]
                                   QuantityPurchased[*];

param StageCost := FirstStage  FirstStageCost
                   SecondStage SecondStageCost ;
```

This data file is verbose and somewhat redundant, but in most applications it is generated by software rather than by a person, so this is not an issue. Generally, the left-most part of each expression (e.g. ''set Stages :=") is required and uses reserved words (e.g., `Stages`) and the other names are supplied by the user (e.g., ''FirstStage" could be any name). Every assignment is terminated with a semi-colon. We will now consider the assignments in this file one at a time.

The first assignments provides names for the stages and the words "set Stages" are required, as are the := symbols. Any names can be used. In this example, we used "FirstStage" and "SecondStage" but we could have used "EtapPrimero" and "ZweiteEtage" if we had wanted to. Whatever names are given here will continue to be used to refer to the stages in the rest of the file. The order of the names is important. A simple way to think of it is that generally, the names must be in time order (technically, they need to be in order of information discovery, but that is usually time-order). Stages refers to decision stages, which may, or may not, correspond directly with time stages. In the farmer example, decisions about how much to plant are made in the first stage and "decisions" (which are pretty obvious, but which are decision variables nonetheless) about how much to sell at each price and how much needs to be bought are second stage decisions because they are made after the yield is known.

```
set Stages := FirstStage SecondStage ;
```

Node names are constructed next. The words "set Nodes" are required, but any names may be assigned to the nodes. In two stage stochastic problems there is a root node, which we chose to name "RootNode" and then there is a node for each scenario.

```
set Nodes := RootNode
           BelowAverageNode
           AverageNode
           AboveAverageNode ;
```

Nodes are associated with time stages with an assignment beginning with the required words "param Nodestage." The assignments must make use of previously defined node and stage names. Every node must be assigned a stage.

```
param NodeStage := RootNode         FirstStage
                   BelowAverageNode SecondStage
                   AverageNode      SecondStage
                   AboveAverageNode SecondStage ;
```

The structure of the scenario tree is defined using assignment of children to each node that has them. Since this is a two stage problem, only the root node has children. The words "param Children" are required for every node that has children and the name of the node is in square brackets before the colon-equals assignment symbols. A list of children is assigned.

```
set Children[RootNode] := BelowAverageNode
                          AverageNode
                          AboveAverageNode ;
```

The probability for each node, conditional on observing the parent node is given in an assignment that begins with the required words "param ConditionalProbability." The root node always has a conditional probability of 1, but it must always be given anyway. In this example, the second stage nodes are equally likely.

```
param ConditionalProbability := RootNode          1.0
                                BelowAverageNode  0.33333333
                                AverageNode       0.33333334
                                AboveAverageNode  0.33333333 ;
```

Scenario names are given in an assignment that begins with the required words "set Scenarios" and provides a list of the names of the scenarios. Any names may be given. In many applications they are given unimaginative names generated by software such as "Scen1" and the like. In this example, there are three scenarios and the names reflect the relative values of the yields.

```
set Scenarios := BelowAverageScenario
                 AverageScenario
                 AboveAverageScenario ;
```

Leaf nodes, which are nodes with no children, are associated with scenarios. This assignment must be one-to-one and it is initiated with the words "param ScenarioLeafNode" followed by the colon-equals assignment characters.

```
param ScenarioLeafNode :=
                    BelowAverageScenario BelowAverageNode
                    AverageScenario      AverageNode
                    AboveAverageScenario AboveAverageNode ;
```

Variables are associated with stages using an assignment that begins with the required words "set StageVariables" and the name of a stage in square brackets followed by the colon-equals assignment characters. Variable names that have been defined in the file ReferenceModel.py can be assigned to stages. Any variables that are not assigned are assumed to be in the last stage. Variable indexes can be given explicitly and/or wildcards can be used. Note that the variable names appear without the prefix "model." In the farmer example, DevotedAcreage is the only first stage variable.

```
set StageVariables[FirstStage] :=  DevotedAcreage[*] ;
set StageVariables[SecondStage] := QuantitySubQuotaSold[*]
                                   QuantitySuperQuotaSold[*]
                                   QuantityPurchased[*] ;
```

Note: Variable names appear without the prefix "model."

Note: Wildcards can be used, but fully general Python slicing is not supported.

For reporting purposes, it is useful to define auxiliary variables in `ReferenceModel.py` that will be assigned the cost associated with each stage. This variables do not impact algorithms, but the values are output by some software during execution as well as upon completion. The names of the variables are assigned to stages using the "param StageCost" assignment. The stages are previously defined in `ScenarioStructure.dat` and the variables are previously defined in `ReferenceModel.py`.

```
param StageCost := FirstStage  FirstStageCost
                   SecondStage SecondStageCost ;
```

1.2.4 Scenario data specification

So far, we have given a model in the file named `ReferenceModel.py`, a set of deterministic data in the file named `ReferenceModel.py`, and a description of the stochastics in the file named `ScenarioStructure.dat`. All that remains is to give the data for each scenario. There are two ways to do that in PySP: *scenario-based* and *node-based*. The default is scenario-based so we will describe that first.

For scenario-based data, the full data for each scenario is given in a `.dat` file with the root name that is the name of the scenario. So, for example, the file named `AverageScenario.dat` must contain all the data for the model for the scenario named "AvererageScenario." It turns out that this file can be created by simply copying the file `ReferenceModel.dat` as shown above because it contains a full set of data for the "AverageScenario" scenario. The files `BelowAverageScenario.dat` and `AboveAverageScenario.dat` will differ from this file and from each other only in their last line, where the yield is specified. These three files are distributed with PySP and are in the pysp sub-directory `examples/farmer/scenariodata` along with `ScenarioStructure.dat` and `ReferenceModel.dat`.

Scenario-based data wastes resources by specifying the same thing over and over again. In many cases, that does not matter and it is convenient to have full scenario data files available (for one thing, the scenarios can easily be run independently using the `pyomo` command). However, in many other settings, it is better to use a node-based specification where the data that is unique to each node is specified in a .dat file with a root name that matches the node name. In the farmer example, the file `RootNode.dat` will be the same as `ReferenceModel.dat` except that it will lack the last line that specifies the yield. The files `BelowAverageNode.dat`, `AverageNode.dat`, and `AboveAverageNode.dat` will contain only one line each to specify the yield. If node-based data is to be used, then the `ScenarioStructure.dat` file must contain the following line:

```
param ScenarioBasedData := False ;
```

An entire set of files for node-based data for the farmer problem are distributed with PySP in the sub-directory `examples/farmer/nodedata`

1.3 Finding Solutions for Stochastic Models

PySP provides a variety of tools for finding solutions to stochastic programs.

1.3.1 runef

The `runef` command puts together the so-called *extensive form* version of the model. It creates a large model that has constraints to ensure that variables at a node have the same value. For example, in the farmer problem, all of the `DevotedAcres` variables must have the same value regardless of which scenario is ultimately realized. The objective can be the expected value of the objective function, or the CVaR, or a weighted combination of the two. Expected value is the default. A full set of options for `runef` can be obtained using the command:

```
runef --help
```

The pysp distribution contains the files need to run the farmer example in the sub-directories to the sub-directory `examples/farmer` so if this is the current directory and if CPLEX is installed, the following command will cause formation of the EF and its solution using CPLEX.

```
runef -m models -i nodedata --solver=cplex --solve
```

The option `-m models` has one dash and is short-hand for the option `--model-directory=models` and note that the full option uses two dashes. The `-i` is equivalent to `--instance-directory=` in the same fashion. The default solver is CPLEX, so the solver option is not really needed. With the `--solve` option, runef would simply write an .lp data file that could be passed to a solver.

1.3.2 runph

The runph command executes an implementation of Progressive Hedging (PH) that is intended to support scripting and extension.

The pysp distribution contains the files need to run the farmer example in the sub-directories to the sub-directory `examples/farmer` so if this is the current directory and if CPLEX is installed, the following command will cause PH to execute using the default sub-problem solver, which is CPLEX.

```
runph -m models -i nodedata
```

The option `-m models` has one dash and is short-hand for the option `--model-directory=models` and note that the full option uses two dashes. The `-i` is equivalent to `--instance-directory=` in the same fashion.

After about 33 iterations, the algorithm will achieve the default level of convergence and terminate. A lot of output is generated and among the output is the following solution information:

```
Variable=DevotedAcreage
      Index: [CORN]            (Scenarios:  BelowAverageScenario   AverageScenario  ␣
↪AboveAverageScenario   )
              Values:       79.9844      80.0000      79.9768     Max-Min=      0.0232  ␣
↪  Avg=     79.9871
      Index: [SUGAR_BEETS]             (Scenarios:  BelowAverageScenario  ␣
↪AverageScenario   AboveAverageScenario   )
              Values:      249.9848     249.9770     250.0000     Max-Min=      0.0230  ␣
↪  Avg=    249.9873
      Index: [WHEAT]           (Scenarios:  BelowAverageScenario   AverageScenario  ␣
↪AboveAverageScenario   )
              Values:      170.0308     170.0230     170.0232     Max-Min=      0.0078  ␣
↪  Avg=    170.0256

Cost Variable=FirstStageCost
      Tree Node=RootNode               (Scenarios:  BelowAverageScenario  ␣
↪AverageScenario   AboveAverageScenario   )
      Values:   108897.0836  108897.4725  108898.1476     Max-Min=      1.0640     Avg=␣
↪108897.5679
```

For problems with no, or few, integer variables, the default level of convergence leaves root-node variables almost converged. Since the acreage to be planted cannot depend on the scenario that will be realized in the future, the average, which is labeled "Avg" in this output, would be used. A farmer would probably interpret acreages of 79.9871, 249.9873, and 170.0256 to be 80, 250, and 170. In real-world applications, PH is embedded in scripts that produce output in a format desired by a decision maker.

But in real-world applications, the default settings for PH seldom work well enough. In addition to post-processing the output, a number of parameters need to be adjusted and sometimes scripting to extend or augment the algorithm is needed to improve convergence rates. A full set of options can be obtained with the command:

```
runph --help
```

Note that there are two dashes before `help`.

By default, PH uses quadratic objective functions after iteration zero; in some settings it may be desirable to linearize the quadratic terms. This is required to use a solver such as glpk for MIPs because it does not support quadratic MIPs. The directive `--linearize-nonbinary-penalty-terms=n` causes linearization of the penalty terms using n pieces. For example, to use glpk on the farmer, assuming glpk is installed and the command is given when the current directory is the `examples/farmer`, the following command will use default settings for most parameters and four pieces to approximate quadratic terms in sub-problems:

```
runph -i nodedata -m models --solver=glpk --linearize-nonbinary-penalty-terms=4
```

Use of the `linearize-nonbinary-penalty-terms` option requires that all variables not in the final stage have bounds.

1.3.3 Final Solution

At each iteration, PH computes an average for each variable over the nodes of the scenario tree. We refer to this as X-bar. For many problems, particularly those with integer restrictions, X-bar might not be feasible for every scenario unless PH happens to be fully converged (in the primal variables). Consequently, the software computes a solution system X-hat that is more likely to be feasible for every scenario and will be equivalent to X-bar under full convergence. This solution is reported upon completion of PH and its expected value is report if it is feasible for all scenarios.

Methods for computing X-hat are controlled by the `--xhat-method` command-line option. For example

```
--xhat-method=closest-scenario
```

causes X-hat to be set to the scenario that is closest to X-bar (in a z-score sense). Other options, such as `voting` and `rounding`, assign values of X-bar to X-hat except for binary and general integer variables, where the values are set by probability weighted voting by scenarios and rounding from X-bar, respectively.

1.3.4 Solution Output Control

To get the full solution, including leaf node solution values, use the `runph --output-scenario-tree-solution` option.

In both `runph` and `runef` the solution can be written in csv format using the `--solution-writer=pysp.plugins.csvsolutionwriter` option.

1.4 Summary of PySP File Names

PySP scripts such as `runef` and `runph` require files that specify the model and data using files with specific names. All files can be in the current directory, but typically, the file `ReferenceModel.py` is in a directory that is specified using `--model-directory=` option (the short version of this option is `-i`) and the data files are in a directory specified in the `--instance-directory=` option (the short version of this option is `-m`).

Note: A file name other than `ReferenceModel.py` can be used if the file name is given in addition to the directory name as an argument to the `--instance-directory` option. For example, on a Windows machine `--instance-directory=models\MyModel.py` would specify the file `MyModel.py` in the local directory `models`.

- `ReferenceModel.py`: A full Pyomo model for a singe scenario. There should be no scenario indexes in this model because they are implicit.
- `ScenarioStructure.dat`: Specifies the nature of the stochastics. It also specifies whether the rest of the data is node-based or scenario-based. It is scenario-based unless `ScenarioStructure.dat` contains the line

```
param ScenarioBasedData := False ;
```

If scenario-based, then there is a data file for each scenario that specifies a full set of data for the scenario. The name of the file is the name of the scenario with `.dat` appended. The names of the scenarios are given in the `ScenarioStructure.dat` file.

If node-based, then there is a file with data for each node that specifies only that data that is unique for the node. The name of the file is the name of the node with `.dat` appended. The names of the nodes are given in the `ScenarioStructure.dat` file.

1.5 Solving Sub-problems in Parallel and/or Remotely

The Python package called Pyro provides capabilities that are used to enable PH to make use of multiple solver processes for sub-problems and allows both `runef` and `runph` to make use remote solvers. We will focus on PH in our discussion here.

There are two solver management systems available for `runph`, one is based on a `pyro_mip_server` and the other is based on a `phsolverserver`. Regardless of which is used, a name server and a dispatch server must be running and accessible to the `runph` process. The name server is launched using the command `pyomo_ns` and then the dispatch server is launched with `dispatch_srvr`. Note that both commands contain an underscore. Both programs keep running until terminated by an external signal, so it is common to pipe their output to a file.

Solvers are controlled by solver servers. The pyro mip solver server is launched with the command `pyro_mip_server`. This command may be repeated to launch as many solvers as are desired. The `runph` then needs a `--solver-manager=pyro` option to signal that `runph` should not launch its own solver, but should send subproblems to be dispatched to parallel solvers. To summarize the commands:

- Once: `pyomo_ns`
- Once: `dispatch_srvr`
- Multiple times: `pyro_mip_server`
- Once: `runph ... --solver-manager=pyro ...`

Note: The `runph` option `--shutdown-pyro` will cause a shutdown signal to be sent to `pyomo_ns`, `dispatch_srvr` and all `pyro_mip_server` programs upon termination of `runph`.

Instead of using `pyro_mip_server`, one can use `phsolverserver` in its place. You can get a list of arguments using `pyrosolverserver --help`, which does not launch a solver server (it just displays help and terminates). If you use the phsolverserver, then use `--solver-manager=phpyro` as an argument to runph rather than `--solver-manager=pyro`.

Warning: Unlike the normal `pyro_mip_server`, there must be one `phsolverserver` for each sub-problem. One can use fewer phsolverservers than there are scenarios by adding the command-line option "–phpyro-required-workers=X". This will partition the jobs among the available workers,

1.6 Generating SMPS Input Files From PySP Models

This document explains how to convert a PySP model into a set of files representing the SMPS format for stochastic linear programs. Conversion can be performed through the command line by invoking the SMPS converter using the command `python -m pysp.convert.smps`. This command is available starting with Pyomo version 5.1. Prior to version 5.1, the same functionality was available via the command `pysp2smps` (starting at Pyomo version 4.2).

SMPS is a standard for expressing stochastic mathematical programs that is based on the ancient MPS format for linear programs, which is matrix-based. Modern algebraic modeling languages such as Pyomo offer a lot of flexibility so it is a challenge to take models expressed in Pyomo/PySP and force them into SMPS format. The conversions can be inefficient and error prone because Pyomo allows flexible expressions and model construction so the resulting matrix may not be the same for each set of input data. We provide tools for conversion to SMPS because some researchers have tools that read SMPS and exploit its limitations on problem structure; however, the user should be aware that the conversion is not always possible.

Currently, these routines only support two-stage stochastic programs. Support for models with more than two time stages will be considered in the future as this tool matures.

1.6.1 Additional Requirements for SMPS Conversion

To enable proper conversion of a PySP model to a set of SMPS files, the following additional requirements must be met:

1. The reference Pyomo model must include annotations that identify stochastic data locations in the second-stage problem.
2. All model variables must be declared in the ScenarioStructure.dat file.
3. The set of constraints and variables, and the overall sparsity structure of the objective and constraint matrix must not change across scenarios.

The bulk of this section discusses in-depth the annotations mentioned in the first point. The second point may come as a surprise to users that are not aware of the ability to *not* declare variables in the ScenarioStructure.dat file. Indeed, for most of the code in PySP, it is only critical that the variables for which non-anticipativity must be enforced need to be declared. That is, for a two-stage stochastic program, all second-stage variables can be left out of the ScenarioStructure.dat file when using commands such as `runef` and `runph`. However, conversion to SMPS format requires all variables to be properly assigned a decision stage by the user.

Note: Variables can be declared as *primary* by assigning them to a stage using the `StageVariables` assignment, or declared as *auxiliary* variables, which are assigned to a stage using `StageDerivedVariables` assignment. For algorithms such as PH, the distinction is meaningful and those variables that are fully determined by primary variables and the data should generally be assigned to `StageDerivedVariables` for their stage.

The third point may also come as a surprise, but the ability to handle a non-uniform problem structure in most PySP tools falls directly from the fact that the non-anticipativity conditions are all that is required in many cases. However, the conversion to SMPS format is based on a matrix representation of the problem where the stochastic coefficients are provided as a set of sparse matrix coordinates. This subsequently requires that the row and column dimensions as well as the sparsity structure of the problem does not change across scenarios.

1.6.2 Annotating Models for SMPS File Generation

Annotations are necessary for alerting the SMPS conversion routines of the locations of data that needs to be updated when changing from one scenario to another. Knowing these sparse locations allows decomposition algorithms to employ efficient methods for solving a stochastic program. In order to use the SMPS conversion tool, at least one of the following annotations must be declared on the reference Pyomo model:

- **StochasticConstraintBoundsAnnotation**: indicates the existence of stochastic constraint right-hand-sides (or bounds) in second-stage constraints
- **StochasticConstraintBodyAnnotation**: indicates the existence of stochastic variable coefficients in second-stage constraints
- **StochasticObjectiveAnnotation**: indicates the existence stochastic cost coefficients in the second-stage cost function

These will be discussed in further detail in the remaining sections. The following code snippet demonstrates how to import these annotations and declare them on a model.

```
from pysp import annotations
model.stoch_rhs = annotations.StochasticConstraintBoundsAnnotation()
model.stoch_matrix = annotations.StochasticConstraintBodyAnnotation()
model.stoch_objective = annotations.StochasticObjectiveAnnotation()
```

Populating these annotations with entries is optional, and simply declaring them on the reference Pyomo model will alert the SMPS conversion routines that all coefficients appearing on the second-stage model should be assumed stochastic. That is, adding the lines in the previous code snippet alone implies that: (i) all **second-stage constraints** have stochastic bounds, (ii) all **first- and second-stage variables** appearing in **second-stage constraints** have stochastic coefficients, and (iii) all **first- and second-stage variables** appearing in the objective have stochastic coefficients.

PySP can attempt to determine the *stage*-ness of a constraint by examining the set of variables that appear in the constraint expression. E.g., a first-stage constraint is characterized as having only first-stage variables appearing in its expression. A second-stage constraint has at least one second-stage variable appearing in its expression. The stage of a variable is declared in the scenario tree provided to PySP. This method of constraint stage classification is not perfect. That is, one can very easily define a model with a constraint that uses only first-stage variables in an expression involving stochastic data. This constraint would be incorrectly identified as first-stage by the method above, even though the existence of stochastic data necessarily implies it is second-stage. To deal with cases such as this, an additional annotation is made available that is named **ConstraintStageAnnotation**. This annotation will be discussed further in a later section.

It is often the case that relatively few coefficients on a stochastic program change across scenarios. In these situations, adding explicit declarations within these annotations will allow for a more sparse representation of the problem and, consequently, more efficient solution by particular decomposition methods. Adding declarations to these annotations is performed by calling the `declare` method, passing some component as the initial argument. Any remaining argument requirements for this method are specific to each annotation. Valid types for the component argument typically include:

- `Constraint`: includes single constraint objects as well as constraint containers
- `Objective`: includes single objective objects as well as objective containers
- `Block`: includes Pyomo models as well as single block objects and block containers

Any remaining details for adding declarations to the annotations mentioned thus far will be discussed in later sections. The remainder of this section discusses the semantics of these declarations based on the type for the component argument.

When the `declare` method is called with a component such as an indexed `Constraint` or a `Block` (model), the SMPS conversion routines will interpret this as meaning all constraints found within that indexed `Constraint` or on that `Block` (that have not been deactivated) should be considered. As an example, we consider the following partially declared concrete Pyomo model:

```
model = pyo.ConcreteModel()

# data that is initialized on a per-scenario basis
p = 1.0
q = 2.0

# variables declared as second-stage on the
# PySP scenario tree
model.z = pyo.Var()
model.y = pyo.Var()

# indexed constraint
model.r_index = pyo.Set(initialize=[3, 6, 9])
def r_rule(model, i):
    return pyo.inequality(p + i, 1*model.z + 5*model.y, 10 + q + i)
model.r = pyo.Constraint(model.r_index, rule=r_rule)

# singleton constraint
model.c = pyo.Constraint(expr= p*model.z >= 1)

# a sub-block with a singleton constraint
model.b = pyo.Block()
model.b.c = pyo.Constraint(expr= q*model.y >= 1)
```

Here the local Python variables `p` and `q` serve as placeholders for data that changes with each scenario.

The following are equivalent annotations of the model, each declaring all of the constraints shown above as having stochastic right-hand-side data:

- Implicit form

```
model.stoch_rhs = annotations.StochasticConstraintBoundsAnnotation()
```

- Implicit form for `Block` (model) assignment

```
model.stoch_rhs = annotations.StochasticConstraintBoundsAnnotation()
model.stoch_rhs.declare(model)
```

- Explicit form for singleton constraint with implicit form for indexed constraint and sub-block

```
model.stoch_rhs = annotations.StochasticConstraintBoundsAnnotation()
model.stoch_rhs.declare(model.r)
model.stoch_rhs.declare(model.c)
model.stoch_rhs.declare(model.b)
```

- Explicit form for singleton constraints at the model and sub-block level with implicit form for indexed constraint

```
model.stoch_rhs = annotations.StochasticConstraintBoundsAnnotation()
model.stoch_rhs.declare(model.r)
model.stoch_rhs.declare(model.c)
model.stoch_rhs.declare(model.b.c)
```

- Fully explicit form for singleton constraints as well as all indices of indexed constraint

```
model.stoch_rhs = annotations.StochasticConstraintBoundsAnnotation()
model.stoch_rhs.declare(model.r[3])
model.stoch_rhs.declare(model.r[6])
model.stoch_rhs.declare(model.r[9])
model.stoch_rhs.declare(model.c)
model.stoch_rhs.declare(model.b.c)
```

Note that the equivalence of the first three bullet forms to the last two bullet forms relies on the following conditions being met: (1) `model.z` and `model.y` are declared on the second stage of the PySP scenario tree and (2) at least one of these second-stage variables appears in each of the constraint expressions above. Together, these two conditions cause each of the constraints above to be categorized as second-stage; thus, causing them to be considered by the SMPS conversion routines in the implicit declarations used by the first three bullet forms.

Warning: Pyomo simplifies product expressions such that terms with 0 coefficients are removed from the final expression. This can sometimes create issues with determining the correct stage classification of a constraint as well as result in different sparsity patterns across scenarios. This issue is discussed further in the later section entitled Edge-Cases.

When it comes to catching errors in model annotations, there is a minor difference between the first bullet form from above (empty annotation) and the others. In the empty case, PySP will use exactly the set of second-stage constraints it is aware of. This set will either be determined through inspection of the constraint expressions or through the user-provided constraint-stage classifications declared using the **ConstraintStageAnnotation** annotation type. In the case where the stochastic annotation is not empty, PySP will verify that all constraints declared within it belong to the set of second-stage constraints it is aware of. If this verification fails, an error will be reported. This behavior is meant to aid users in debugging problems associated with non-uniform sparsity structure across scenarios that are, for example, caused by 0 coefficients in product expressions.

Annotations on AbstractModel Objects

Pyomo models defined using the `AbstractModel` object require the modeler to take further steps when making these annotations. In the `AbstractModel` setting, these assignments must take place within a `BuildAction`, which is executed only after the model has been constructed with data. As an example, the last bullet form from the previous section could be written in the following way to allow execution with either an `AbstractModel` or a `ConcreteModel`:

```
def annotate_rule(m):
    m.stoch_rhs = annotations.StochasticConstraintBoundsAnnotation()
    m.stoch_rhs.declare(m.r[3])
    m.stoch_rhs.declare(m.r[6])
    m.stoch_rhs.declare(m.r[9])
    m.stoch_rhs.declare(m.c)
    m.stoch_rhs.declare(m.b.c)
model.annotate = pyo.BuildAction(rule=annotate_rule)
```

Note that the use of `m` rather than `model` in the `annotate_rule` function is meant to draw attention to the fact that the model object being passed into the function as the first argument may not be the same object as the model outside of the function. This is in fact the case in the `AbstractModel` setting, whereas for the `ConcreteModel` setting they are the same object. We often use `model` in both places to avoid errors caused by forgetting to use the correct object inside the function (Python scoping rules handle the rest). Also note that a `BuildAction` must be declared on the model after the declaration of any components being accessed inside its rule function.

Stochastic Constraint Bounds (RHS)

If stochastic elements appear on the right-hand-side of constraints (or as constants in the body of constraint expressions), these locations should be declared using the **StochasticConstraintBoundsAnnotation** annotation type. When

components are declared with this annotation, there are no additional required arguments for the `declare` method. However, to allow for more flexibility when dealing with double-sided inequality constraints, the `declare` method can be called with at most one of the keywords `lb` or `ub` set to `False` to signify that one of the bounds is not stochastic. The following code snippet shows example declarations with this annotation for various constraint types.

```
# declare the annotation
model.stoch_rhs = annotations.StochasticConstraintBoundsAnnotation()

# equality constraint
model.c_eq = pyo.Constraint(expr= model.y == q)
model.stoch_rhs.declare(model.c_eq)

# range inequality constraint with stochastic upper bound
model.c_ineq = pyo.Constraint(expr= pyo.inequality(0, model.y, p))
model.stoch_rhs.declare(model.c_ineq, lb=False)

# indexed constraint using a BuildAction
model.C_index = pyo.RangeSet(1,3)
def C_rule(model, i):
    if i == 1:
        return model.y >= i * q
    else:
        return pyo.Constraint.Skip
model.C = pyo.Constraint(model.C_index, rule=C_rule)
def C_annotate_rule(model, i):
    if i == 1:
        model.stoch_rhs.declare(model.C[i])
    else:
        pass
model.C_annotate = pyo.BuildAction(model.C_index, rule=C_annotate_rule)
```

Note that simply declaring the **StochasticConstraintBoundsAnnotation** annotation type and leaving it empty will alert the SMPS conversion routines that all constraints identified as second-stage should be treated as having stochastic right-hand-side data. Calling the `declare` method on at least one component implies that the set of constraints considered should be limited to what is declared within the annotation.

Stochastic Constraint Matrix

If coefficients of variables change in the second-stage constraint matrix, these locations should be declared using the **StochasticConstraintBodyAnnotation** annotation type. When components are declared with this annotation, there are no additional required arguments for the `declare` method. Calling the `declare` method with the single component argument signifies that all variables encountered in the constraint expression (including first- and second-stage variables) should be treated as having stochastic coefficients. This can be limited to a specific subset of variables by calling the `declare` method with the `variables` keyword set to an explicit list of variable objects. The following code snippet shows example declarations with this annotation for various constraint types.

```
model = pyo.ConcreteModel()

# data that is initialized on a per-scenario basis
p = 1.0
q = 2.0

# a first-stage variable
model.x = pyo.Var()
```

(continues on next page)

(continued from previous page)

```
# a second-stage variable
model.y = pyo.Var()

# declare the annotation
model.stoch_matrix = annotations.StochasticConstraintBodyAnnotation()

# a singleton constraint with stochastic coefficients
# both the first- and second-stage variable
model.c = pyo.Constraint(expr= p*model.x + q*model.y == 1)
model.stoch_matrix.declare(model.c)
# an assignment that is equivalent to the previous one
model.stoch_matrix.declare(model.c, variables=[model.x, model.y])

# a singleton range constraint with a stochastic coefficient
# for the first-stage variable only
model.r = pyo.Constraint(expr=pyo.inequality(0, p*model.x - 2.0*model.y, 10))
model.stoch_matrix.declare(model.r, variables=[model.x])
```

As is the case with the **StochasticConstraintBoundsAnnotation** annotation type, simply declaring the **StochasticConstraintBodyAnnotation** annotation type and leaving it empty will alert the SMPS conversion routines that all constraints identified as second-stage should be considered, and, additionally, that all variables encountered in these constraints should be considered to have stochastic coefficients. Calling the `declare` method on at least one component implies that the set of constraints considered should be limited to what is declared within the annotation.

Stochastic Objective Elements

If the cost coefficients of any variables are stochastic in the second-stage cost expression, this should be noted using the **StochasticObjectiveAnnotation** annotation type. This annotation uses the same semantics for the `declare` method as the **StochasticConstraintBodyAnnotation** annotation type, but with one additional consideration regarding any constants in the objective expression. Constants in the objective are treated as stochastic and automatically handled by the SMPS code. If the objective expression does not contain any constant terms or these constant terms do not change across scenarios, this behavior can be disabled by setting the keyword `include_constant` to `False` in a call to the `declare` method.

```
# declare the annotation
model.stoch_objective = annotations.StochasticObjectiveAnnotation()

model.FirstStageCost = pyo.Expression(expr= 5.0*model.x)
model.SecondStageCost = pyo.Expression(expr= p*model.x + q*model.y)
model.TotalCost = pyo.Objective(expr= model.FirstStageCost + model.SecondStageCost)

# each of these declarations is equivalent for this model
model.stoch_objective.declare(model.TotalCost)
model.stoch_objective.declare(model.TotalCost, variables=[model.x, model.y])
```

Similar to the previous annotation type, simply declaring the **StochasticObjectiveAnnotation** annotation type and leaving it empty will alert the SMPS conversion routines that all variables appearing in the single active model objective expression should be considered to have stochastic coefficients. *Edge Cases*

The section discusses various points that may give users some trouble, and it attempts to provide more details about the common pitfalls associated with translating a PySP model to SMPS format.

- *Moving a Stochastic Objective to the Constraint Matrix*

It is often the case that decomposition algorithms theoretically support stochastic cost coefficients but the software

implementation has not yet added support for them. This situation is easy to work around in PySP. One can simply augment the model with an additional constraint and variable that *computes* the objective, and then use this variable in the objective rather than directly using the second-stage cost expression. Consider the following reference Pyomo model that has stochastic cost coefficients for both a first-stage and a second-stage variable in the second-stage cost expression:

```
>>> # suppress duplicate object warning
>>> del model.TotalCost
```

```
# define the objective as the sum of the stage-cost expressions
model.TotalCost = pyo.Objective(expr= model.FirstStageCost + model.SecondStageCost)

# declare that model.x and model.y have stochastic cost
# coefficients in the second stage
model.stoch_objective = annotations.StochasticObjectiveAnnotation()
model.stoch_objective.declare(model.TotalCost, variables=[model.x, model.y])
```

The code snippet below re-expresses this model using an objective consisting of the original first-stage cost expression plus a second-stage variable `SecondStageCostVar` that represents the second-stage cost. This is enforced by restricting the variable to be equal to the second-stage cost expression using an additional equality constraint named `ComputeSecondStageCost`. Additionally, the **StochasticObjectiveAnnotation** annotation type is replaced with the **StochasticConstraintBodyAnnotation** annotation type.

```
# set the variable SecondStageCostVar equal to the
# expression SecondStageCost using an equality constraint
model.SecondStageCostVar = pyo.Var()
model.ComputeSecondStageCost = pyo.Constraint(expr= model.SecondStageCostVar == model.
↪SecondStageCost)

# declare that model.x and model.y have stochastic constraint matrix
# coefficients in the ComputeSecondStageCost constraint
model.stoch_matrix = annotations.StochasticConstraintBodyAnnotation()
model.stoch_matrix.declare(model.ComputeSecondStageCost, variables=[model.x, model.y])
```

- *Stochastic Constant Terms*

The standard description of a linear program does not allow for a constant term in the objective function because this has no weight on the problem solution. Additionally, constant terms appearing in a constraint expression must be lumped into the right-hand-side vector. However, when modeling with an AML such as Pyomo, constant terms very naturally fall out of objective and constraint expressions.

If a constant terms falls out of a constraint expression and this term changes across scenarios, it is critical that this is accounted for by including the constraint in the **StochasticConstraintBoundsAnnotation** annotation type. Otherwise, this would lead to an incorrect representation of the stochastic program in SMPS format. As an example, consider the following:

```
# a param initialized with scenario-specific data
model.p = pyo.Param(mutable=True)

# a second-stage constraint with a stochastic upper bound
# hidden in the left-hand-side expression
def d_rule(m):
    return (m.x - m.p) + m.y <= 10
model.d = pyo.Constraint(rule=d_rule)
```

Note that in the expression for constraint `c`, there is a fixed parameter `p` involved in the variable expression on the left-hand-side of the inequality. When an expression is written this way, it can be easy to forget that the value of this parameter will be pushed to the bound of the constraint when it is converted into linear canonical form. Remember to declare these constraints within the **StochasticConstraintBoundsAnnotation** annotation type.

A constant term appearing in the objective expression presents a similar issue. Whether or not this term is stochastic, it must be dealt with when certain outputs expect the problem to be expressed as a linear program. The SMPS code in PySP will deal with this situation for you by implicitly adding a new second-stage variable to the problem in the final output file that uses the constant term as its coefficient in the objective and that is fixed to a value of 1.0 using a trivial equality constraint. The default behavior when declaring the **StochasticObjectiveAnnotation** annotation type will be to assume this constant term in the objective is stochastic. This helps ensure that the relative scenario costs reported by algorithms using the SMPS files will match that of the PySP model for a given solution. When moving a stochastic objective into the constraint matrix using the method discussed in the previous subsection, it is important to be aware of this behavior. A stochastic constant term in the objective would necessarily translate into a stochastic constraint right-hand-side when moved to the constraint matrix.

- *Stochastic Variable Bounds*

Although not directly supported, stochastic variable bounds can be expressed using explicit constraints along with the **StochasticConstraintBoundsAnnotation** annotation type to achieve the same effect.

- *Problems Caused by Zero Coefficients*

Expressions that involve products with some terms having 0 coefficients can be problematic when the zeros can become nonzero in certain scenarios. This can cause the sparsity structure of the LP to change across scenarios because Pyomo simplifies these expressions when they are created such that terms with a 0 coefficient are dropped. This can result in an invalid SMPS conversion. Of course, this issue is not limited to explicit product expressions, but can arise when the user implicitly assigns a variable a zero coefficient by outright excluding it from an expression. For example, both constraints in the following code snippet suffer from this same underlying issue, which is that the variable `model.y` will be excluded from the constraint expressions in a subset of scenarios (depending on the value of `q`) either directly due to a 0 coefficient in a product expressions or indirectly due to user-defined logic that is based off of the values of stochastic data.

```
q = 0

model.c1 = pyo.Constraint(expr= p * model.x + q * model.y == 1)

def c2_rule(model):
    expr = p * model.x
    if q != 0:
        expr += model.y
    return expr >= 0
model.c2 = pyo.Constraint(rule=c2_rule)
```

The SMPS conversion routines will attempt some limited checking to help prevent this kind of situation from silently turning the SMPS representation to garbage, but it must ultimately be up to the user to ensure this is not an issue. This is in fact the most challenging aspect of converting PySP's AML-based problem representation to the structure-preserving LP representation used in the SMPS format.

One way to deal with the 0 coefficient issue, which works for both cases discussed in the example above, is to create a *zero* `Expression` object. E.g.,

```
model.zero = pyo.Expression(expr=0)
```

This component can be used to add variables to a linear expression so that the resulting expression retains a reference to them. This behavior can be verified by examining the output from the following example:

```
# an expression that does NOT retain model.y
>>> print((model.x + 0 * model.y).to_string())
x

# an equivalent expression that DOES retain model.y
>>> print((model.x + model.zero * model.y).to_string())
x + 0.0*y

# an equivalent expression that does NOT retain model.y (so beware)
>>> print((model.x + 0 * model.zero * model.y).to_string())
x
```

1.6.3 Generating SMPS Input Files

This section explains how the SMPS conversion utilities available in PySP can be invoked from the command line. Starting with Pyomo version 5.1, the SMPS writer can be invoked using the command `python -m pysp.convert.smps`. Prior to version 5.1, this functionality was available via the `pysp2smps` command (starting at Pyomo version 4.2). Use the `--help` option with the main command to see a detailed description of the command-line options available:

```
$ python -m pysp.convert.smps --help
```

Next, we discuss some of the basic inputs to this command.

Consider the baa99 example inside the `baa99` subdirectory that is distributed with the PySP examples (`examples/baa99`). Both the reference model and the scenario tree structure are defined in the file `ReferenceModel.py` using PySP callback functions. This model has been annotated to enable conversion to the SMPS format. Assuming one is in this example's directory, SMPS files can be generated for the model by executing the following shell command:

```
$ python -m pysp.convert.smps -m ReferenceModel.py --basename baa99 \
   --output-directory sdinput/baa99
```

Assuming successful execution, this would result in the following files being created:

- `sdinput/baa99/baa99.cor`
- `sdinput/baa99/baa99.tim`
- `sdinput/baa99/baa99.sto`
- `sdinput/baa99/baa99.cor.symbols`

The first file is the core problem file written in MPS format. The second file indicates at which row and column the first and second time stages begin. The third file contains the location and values of stochastic data in the problem for each scenario. This file is generated by merging the individual output for each scenario in the scenario tree into separate BLOCK sections. The last file contains a mapping for non-anticipative variables from the symbols used in the above files to a unique string that can be used to recover the variable on any Pyomo model. It is mainly used by PySP's solver interfaces to load a solver solution.

To ensure that the problem structure is the same and that all locations of stochastic data have been annotated properly, the script creates additional auxiliary files that are compared across scenarios. The command-line option `--keep-auxiliary-files` can be used to retain the auxiliary files that were generated for the template scenario used to write the core file. When this option is used with the above example, the following additional files will appear in the output directory:

- `sdinput/baa99/baa99.mps.det`
- `sdinput/baa99/baa99.sto.struct`

- sdinput/baa99/baa99.row
- sdinput/baa99/baa99.col

The .mps.det file is simply the core file for the reference scenario with the values for all stochastic coefficients set to zero. If this does not match for every scenario, then there are places in the model that still need to be declared on one or more of the stochastic data annotations. The .row and the .col files indicate the ordering of constraints and variables, respectively, that was used to write the core file. The .sto.struct file lists the nonzero locations of the stochastic data in terms of their row and column location in the core file. These files are created for each scenario instance in the scenario tree and placed inside of a subdirectory named scenario_files within the output directory. These files will be removed removed unless validation fails or the --keep-scenario-files option is used.

The SMPS writer also supports parallel execution. This can significantly reduce the overall time required to produce the SMPS files when there are many scenarios. Parallel execution using PySP's Pyro-based tools can be performed using the steps below. Note that each of these commands can be launched in the background inside the same shell or in their own separate shells.

1. Start the Pyro name server:

```
$ pyomo_ns -n localhost
```

2. Start the Pyro dispatch server:

```
$ dispatch_srvr -n localhost --daemon-host localhost
```

3. Start 8 ScenarioTree Servers (for the 625 baa99 scenarios)

```
$ mpirun -np 8 scenariotreeserver --pyro-host=localhost
```

4. Run python -m pysp.convert.smps using the Pyro ScenarioTree Manager

```
$ python -m pysp.convert.smps -m ReferenceModel.py --basename baa99 \
   --output-directory sdinput/baa99 \
   --pyro-required-scenariotreeservers=8 \
   --pyro-host=localhost --scenario-tree-manager=pyro
```

An annotated version of the farmer example is also provided. The model file can be found in the examples/farmer/smps_model examples subdirectory. Note that the scenario tree for this model is defined in a separate file. When invoking the SMPS writer, a scenario tree structure file can be provided via the --scenario-tree-location (-s) command-line option. For example, assuming one is in the farmer subdirectory, the farmer model can be converted to SMPS files using the command:

```
$ python -m pysp.convert.smps -m smps_model/ReferenceModel.py \
   -s scenariodata/ScenarioStructure.dat --basename farmer \
   --output-directory sdinput/farmer
```

Note that, by default, the files created by the SMPS writer use shortened symbols that do not match the names of the variables and constraints declared on the Pyomo model. This is for efficiency reasons, as using fully qualified component names can result in significantly larger files. However, it can be useful in debugging situations to generate the SMPS files using the original component names. To do this, simply add the command-line option --symbolic-solver-labels to the command string.

The SMPS writer supports other formats for the core problem file (e.g., the LP format). The command-line option --core-format can be used to control this setting. Refer to the command-line help string for more information about the list of available format.

1.7 Generating DDSIP Input Files From PySP Models

PySP provides support for creating DDSIP inputs, and some support for reading DDSIP solutions back into PySP is under development. Use of these utilties requires additional model annotations that declare the location of stochastic data coefficients. See the section on converting PySP models to SMPS for more information.

To access the DDSIP writer via the command line, use `python -m pysp.convert.ddsip`. To access the full solver interface to DDSIP, which writes the input files, invokes the DDSIP solver, and reads the solution, use `python -m pysp.solvers.ddsip`. For example, to get a list of command arguments, use:

```
$ python -m pysp.convert.ddsip --help
```

Note: Not all of the command arguments are relevant for DDSIP.

For researchers that simply want to write out the files needed by DDSIP, the `--output-directory` option can be used with the DDSIP writer to specifiy the directory where files should be created. The DDSIP solver interface creates these files in a temporary directory. To have the DDSIP solver interface retain these files after it exits, use the `--keep-solver-files` command-line option. The following example invokes the DDSIP solver on the networkflow example that ships with PySP. In order to test it, one must first `cd` into to the networkflow example directory and then execute the command:

```
$ python -m pysp.solvers.ddsip \
   -s 1ef10 -m smps_model --solver-options="NODELIM=1"
```

The `--solver-options` command line argument can be used set the values of any DDSIP options that are written to the DDSIP configuration file; multiple options should be space-separated. See DDSIP documentation for a list of options.

Here is the same example modified to simply create the DDSIP input files in an output directory named `ddsip_networkflow`:

```
$ python -m pysp.convert.ddsip \
   -s 1ef10 -m smps_model --output-directory ddsip_networkflow \
   --symbolic-solver-labels
```

The option `--symbolic-solver-labels` tells the DDSIP writer to produce the file names using symbols that match names on the original Pyomo model. This can significantly increase file size, so it is not done by default. When the DDSIP writer is invoked, a minimal DDSIP configuration file is created in the output directory that specifies the required problem structure information. Any additional DDSIP options must be manually added to this file by the user.

As with the SMPS writer, the DDSIP writer and solver interface support PySP's Pyro-based parallel scenario tree management system. See the section on the SMPS writer for a description of how to use this functionality.

1.8 PySP in scripts

See *rapper: a PySP wrapper* for information about putting Python scripts around PySP functionality.

1.9 Introduction to Using Concrete Models with PySP

The concrete interface to PySP requires a function that can return a concrete model for a given scenario. Optionally, a function that returns a scenario tree can be provided; however, a `ScenarioStructure.dat` file is also an option. This very terse introduction might help you get started using concrete models with PySP.

1.9.1 Scenario Creation Function

There is a lot of flexibility in how this function is implemented, but the path of least resistance is

```
>>> def pysp_instance_creation_callback(scenario_tree_model,
...                                     scenario_name,
...                                     node_names):
...    pass
```

In many applications, only the `scenario_name` argument is used. Its purpose is almost always to determine what data to use when populating the scenario instance. Note that in older examples, the `scenario_tree_model` argument is not present.

An older example of this function can be seen in `examples/farmer/concrete/ReferenceModel.py`

Note that this example does not have a function to return a scenario tree, so it can be solved from the `examples/farmer` directory with a command like:

:: runef -m concrete/ReferenceModel.py -s scenariodata/ScenarioStructure.dat –solve

---

Note: If, for some reason, you want to use the concrete interface for PySP for an `AbstractModel`, the body of the function might be something like:

```
>>> instance = model.create_instance(scenario_name+".dat")
>>> return instance
```

assuming that `model` is defined as an `AbstractModel` in the namespace of the file.

1.9.2 Scenario Tree Creation Function

There are many options for a function to return a scenario tree. The path of least resistance is to name the function `pysp_scenario_tree_model_callback` with no arguments. One example is shown in `examples/farmer/concreteNetX/ReferenceModel.py`

It can be solved from the `examples/farmer` directory with a command like:

:: runef -m concreteNetX/ReferenceModel.py –solve

CHAPTER

TWO

RAPPER: A PYSP WRAPPER

This is an advanced topic.

The *pysp.util.rapper* package is built on the Pyomo optimization modeling language ([PyomoJournal], [PyomoBookII]) to provide a thin wrapper for some functionality of PySP [PySPJournal] associated with the *runef* and *runph* commands. The package is designed mainly for experienced Python programmers who are users of a Pyomo *ConcreteModel* in PySP and who want to embed the solution process is simple scripts. There is also support for users of a Pyomo *AbstractModel*. Note that callback functions are also supported for some aspects of PySP, which is somewhat orthogonal to the functionality provided by *pysp.util.rapper*.

2.1 Demonstration of rapper Capabilities

In this section we provide a series of examples intended to show different things that can be done with rapper.

```
Imports:

>>> import pyomo.pysp.util.rapper as rapper
>>> import pyomo.pysp.plugins.csvsolutionwriter as csvw
>>> from pyomo.pysp.scenariotree.tree_structure_model import␣
↪CreateAbstractScenarioTreeModel
>>> import pyomo.environ as pyo

The next line establishes the solver to be used.

>>> solvername = "cplex"

The next two lines show one way to create a concrete scenario tree. There are
others that can be found in `pyomo.pysp.scenariotree.tree_structure_model`.

>>> abstract_tree = CreateAbstractScenarioTreeModel()
>>> concrete_tree = \
...     abstract_tree.create_instance("ScenarioStructure.dat")
```

2.1.1 Emulate some aspects of *runef*

Create a *rapper* solver object assuming there is a file named *ReferenceModel.py* that has an appropriate *pysp_instance_creation_callback* function.

```
>>> stsolver = rapper.StochSolver("ReferenceModel.py",
...            tree_model = concrete_tree)
```

This object has a *solve_ef* method (as well as a *solve_ph* method)

```
>>> ef_sol = stsolver.solve_ef(solvername)
```

The return status from the solver can be tested.

```
>>> if ef_sol.solver.termination_condition != \
...            pyo.TerminationCondition.optimal:
...     print ("oops! not optimal:",ef_sol.solver.termination_condition)
```

There is an iterator to loop over the root node solution:

```
>>> for varname, varval in stsolver.root_Var_solution():
...    print (varname, str(varval))
```

There is also a function to compute compute the objective function value.

```
>>> obj = stsolver.root_E_obj()
>>> print ("Expecatation take over scenarios=", obj)
```

Also, *stsolver.scenario_tree* has the solution. The package csvw is imported from PySP as shown above.

```
>>> csvw.write_csv_soln(stsolver.scenario_tree, "testcref")
```

It is also possible to add arguments for chance constraints and CVaR; see *rapper API* for details.

2.1.2 Again, but with mip gap reported

Now we will solve the same problem again, but we cannot re-use the same *rapper.StochSolver* object in the same program so we must construct a new one; however, we can re-used the scenario tree.

```
>>> stsolver = rapper.StochSolver("ReferenceModel.py",
...            tree_model = concrete_tree)
```

We add a solver option to get the mip gap

```
>>> sopts = {"mipgap": 1} # I want a gap
```

and we add the option to *solve_ef* to return the gap and the *tee* option to see the solver output as well.

```
>>> res, gap = stsolver.solve_ef(solvername, sopts = sopts, tee=True, need_gap␣
↪= True)
>>> print ("ef gap=",gap)
```

2.1.3 PH

We will now do the same problem, but with PH and we will re-use the scenario tree in *tree_model* from the code above. We put sub-solver options in *sopts* and PH options (i.e., those that would provided to *runph*) Note that if options are passed to the constructor (and the solver); they are passed as a dictionary where options that do not have an argument have the data value *None*. The constructor really only needs to some options, such as those related to bundling.

```
>>> sopts = {}
>>> sopts['threads'] = 2
>>> phopts = {}
>>> phopts['--output-solver-log'] = None
>>> phopts['--max-iterations'] = '3'
```

```
>>> stsolver = rapper.StochSolver("ReferenceModel.py",
...                               tree_model = concrete_tree,
...                               phopts = phopts)
```

The *solve_ph* method is similar to *solve_ef*, but requires a *default_rho* and accepts PH options:

```
>>> ph = stsolver.solve_ph(subsolver = solvername, default_rho = 1,
...                        phopts=phopts)
```

With PH, it is important to be careful to distinguish x-bar from x-hat.

```
>>> obj = stsolver.root_E_obj()
```

We can compute and x-hat (using the current PH options):

```
>>> obj, xhat = rapper.xhat_from_ph(ph)
```

There is a utility for obtaining the x-hat values:

```
>>> for nodename, varname, varvalue in rapper.xhat_walker(xhat):
...     print (nodename, varname, varvalue)
```

2.2 rapper API

A class and some utilities to wrap PySP. In particular to enable programmatic access to some of the functionality in runef and runph for ConcreteModels Author: David L. Woodruff, started February 2017

class pysp.util.rapper.**StochSolver**(*fsfile*, *fsfct=None*, *tree_model=None*, *phopts=None*)

Bases: `object`

A class for solving stochastic versions of concrete models and abstract models. Inspired by the IDAES use case and by daps ability to create tree models. Author: David L. Woodruff, February 2017

Parameters

- **fsfile** (*str*) – is a path to the file that contains the scenario callback for concrete or the reference model for abstract.
- **fsfct** (*str, or fct, or None*) –

  str: callback function name in the file

fct: callback function (fsfile is ignored)
None: it is a AbstractModel

- **tree_model** (*concrete model, or networkx tree, or path*) – gives the tree as a concrete model (which could be a fct) or a valid networkx scenario tree or path to AMPL data file.
- **phopts** – dictionary of ph options; needed during construction if there is bundling.

scenario_tree
scenario tree object (that includes data)

make_ef(*verbose=False*, *generate_weighted_cvar=False*, *cvar_weight=None*, *risk_alpha=None*, *cc_indicator_var_name=None*, *cc_alpha=0.0*)
Make an ef object (used by solve_ef); all Args are optional.

Parameters

- **verbose** (*boolean*) – indicates verbosity to PySP for construction
- **generate_weighted_cvar** (*boolean*) – indicates we want weighted CVar
- **cvar_weight** (*float*) – weight for the cvar term
- **risk_alpha** (*float*) – alpha value for cvar
- **cc_indicator_var_name** (*string*) – name of the Var used for chance constraint
- **cc_alpha** (*float*) – alpha for chance constraint

Returns the ef object

Return type ef_instance

root_E_obj()
post solve Expected cost of the solution in the scenario tree (xbar)

Returns the expected costs of the solution in the tree (xbar)

Return type float

root_Var_solution()
Generator to loop over x-bar

Yields name, value pair for root node solution values

solve_ef(*subsolver*, *sopts=None*, *tee=False*, *need_gap=False*, *verbose=False*, *generate_weighted_cvar=False*, *cvar_weight=None*, *risk_alpha=None*, *cc_indicator_var_name=None*, *cc_alpha=0.0*)
Solve the stochastic program directly using the extensive form. All Args other than subsolver are optional.

Parameters

- **subsolver** (*str*) – the solver to call (e.g., 'ipopt')
- **sopts** (*dict*) – solver options
- **tee** (*bool*) – indicates dynamic solver output to terminal.
- **need_gap** (*bool*) – indicates the need for the optimality gap
- **verbose** (*boolean*) – indicates verbosity to PySP for construction
- **generate_weighted_cvar** (*boolean*) – indicates we want weighted CVar
- **cvar_weight** (*float*) – weight for the cvar term
- **risk_alpha** (*float*) – alpha value for cvar

- **cc_indicator_var_name** (*string*) – name of the Var used for chance constraint
- **cc_alpha** (*float*) – alpha for chance constraint

Returns: (*Pyomo solver result*, *float*)

solve_result is the solver return value.

absgap is the absolute optimality gap (might not be valid); only if requested

---

Note: Also update the scenario tree, populated with the solution. Also attach the full ef instance to the object. So you might want obj = pyo.value(stsolver.ef_instance.MASTER) This needs more work to deal with solver failure (dlw, March, 2018)

---

solve_ph(*subsolver*, *default_rho*, *phopts=None*, *sopts=None*)
Solve the stochastic program given by this.scenario_tree using ph

Parameters

- **subsolver** (*str*) – the solver to call (e.g., 'ipopt')
- **default_rho** (*float*) – the rho value to use by default
- **phopts** – dictionary of ph options (optional)
- **sopts** – dictionary of subsolver options (optional)

Returns: the ph object

Note: Updates the scenario tree, populated with the xbar values; however, you probably want to do obj, xhat = ph.compute_and_report_inner_bound_using_xhat() where ph is the return value.

---

pysp.util.rapper.**xhat_from_ph**(*ph*)
a service fuction to wrap a call to get xhat

Parameters **ph** – a post-solve ph object

Returns: (float, object)

float: the expected cost of the xhat solution for the scenarios

xhat: an object with the solution tree

pysp.util.rapper.**xhat_walker**(*xhat*)
A service generator to walk over a given xhat

Parameters **xhat** (*dict*) – an xhat solution (probably from xhat_from_ph)

Yields (nodename, varname, varvalue)

2.3 Abstract Constructor

In *Demonstration of rapper Capabilities* we provide a series of examples intended to show different things that can be done with rapper and the constructor is shown for a *ConcreteModel*. The same capabilities are available for an *AbstractModel* but the construction of the *rapper* object is different as shown here.

```
Import for constructor:

>>> import pysp.util.rapper as rapper
```

The next line constructs the *rapper* object that can be used to emulate *runph* or *runef*.

```
>>> stsolver = rapper.StochSolver(ReferencePath,
...                               fsfct = None,
...                               tree_model = scenariodirPath,
...                               phopts = None)
```

2.4 The rap

As homage to the tired cliché based on the rap-wrap homonym, we provide the lyrics to our rap anthem:

A PySP State of Mind (The Pyomo Hip Hop)

By Woody and https://www.song-lyrics-generator.org.uk

```
Yeah, yeah
Ayo, modeller, it's time.
It's time, modeller (aight, modeller, begin).
Straight out the scriptable dungeons of rap.

The cat drops deep as does my map.
I never program, 'cause to program is the uncle of rap.
Beyond the walls of scenarios, life is defined.
I think of optimization under uncertainty when I'm in a PySP state of mind.

Hope the resolution got some institution.
My revolution don't like no dirty retribution.
Run up to the distribution and get the evolution.

In a PySP state of mind.

What more could you ask for? The fast cat?
You complain about unscriptability.
I gotta love it though - somebody still speaks for the mat.

I'm rappin' with a cape,
And I'm gonna move your escape.

Easy, big, indented, like a solution
Boy, I tell you, I thought you were an institution.

I can't take the unscriptability, can't take the script.
```

(continues on next page)

(continued from previous page)

```
I woulda tried to code I guess I got no transcript.

Yea, yaz, in a PySP state of mind.

When I was young my uncle had a nondescript.
I waz kicked out without no manuscript.
I never thought I'd see that crypt.
Ain't a soul alive that could take my uncle's transcript.

An object oriented fox is quite the box.

Thinking of optimization under uncertainty.
Yaz, thinking of optimization under uncertainty
(optimization under uncertainty).
```

CHAPTER

THREE

BIBLIOGRAPHY

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[BirgeLouveauxBook] J.R. Birge and F. Louveaux. Introduction to Stochastic Programming. Springer Series in Operations Research. New York. Springer, 1997

[PyomoBookII] W. E. Hart, C. D. Laird, J.-P. Watson, D. L. Woodruff, G. A. Hackebeil, B. L. Nicholson, J. D. Siirola. Pyomo - Optimization Modeling in Python, 2nd Edition. Springer Optimization and Its Applications, Vol 67. Springer, 2017.

[PyomoJournal] William E. Hart, Jean-Paul Watson, David L. Woodruff. "Pyomo: modeling and solving mathematical programs in Python," Mathematical Programming Computation, Volume 3, Number 3, August 2011

[PySPJournal] Jean-Paul Watson, David L. Woodruff, William E. Hart. "Pyomo: modeling and solving mathematical programs in Python," Mathematical Programming Computation, Volume 4, Number 2, June 2012, Pages 109-142

PYTHON MODULE INDEX

p

INDEX