PYSP Version 1.11
User Documentation

Jean-Paul Watson
Sandia National Laboratories
Discrete Math and Complex Systems Department
P.O. Box 5800, MS 1318
Albuquerque, NM 87185-1318 USA
jwatson@sandia.gov

David L. Woodruff
Graduate School of Management
University of California, Davis
Davis, CA 95616-8609 USA
dlwoodruff@ucdavis.edu

March 29, 2011

1 Overview

The pysp package extends the pyomo modeling language to support multi-stage stochastic programs with enumerated scenarios. Pyomo and pysp are Python version 2.6 programs. In order to specify a program, the user must provide a reference model and a scenario tree.

Provided and the necessary paths have been communicated to the operating system, the command to execute the pysp package is of the form:

```
runph
```

It is possible, and generally necessary, to provide command line arguments. The simplest argument causes the program to output help text:

```
runph --help
```

but notice that there are two dashes before the word "help." Command line arguments are summarized in Section 3.

The underlying algorithm in pysp is based on Progressive Hedging (PH) [5], which decomposes the problem into sub-problems, one for each scenario. The algorithm progressively computes *weights* corresponding to each variable to force convergence and also makes use of a *proximal* term that provides a penalty for the squared deviation from the mean solution from the last PH iteration.

1.1 Reference Model

The reference model describes the problem for a canonical scenario. It does not make use of, or describe, a scenario index or any information about uncertainty. Typically, it is just the model that would be used if there were only a single scenario. It is given as a pyomo file. Data from an arbitrary scenario is needed to instantiate.

The objective function needs to be separated by stages. The term for each stage should be "assigned" (i.e., constrained to be equal to) a variable. These variable names are reported in ScenarioStructure.dat so that they can be used for reporting purposes.

1.2 Scenario Tree

The scenario tree provides information about the time stages and the nature of the uncertainties. In order to specify a tree, we must indicate the time stages at which information becomes available. We also specify the nodes of a tree to indicate which variables are associated with which realization at each stage. The data for each scenario is provided in separate data files, one for each scenario.

2 File Structure

- ReferenceModel.py (A pyomo model file)
- ReferenceModel.dat (data for an arbitrary scenario)
- ScenarioStructure.dat (among other things: the scenario names: Sname)
- *Sname.dat (full data for now) one file for each scenario

In this list we use "Sname" as the generic scenario name. The file `ScenarioStructure.dat` gives the names of all the scenarios and for each scenario there is a data file with the same name and the suffix ".dat" that contains the full specification of data for the scenario.

2.1 ScenarioStructure.dat

The file ScenarioStucture.dat contains the following data:

- set Scenarios: List of the names of the scenarios. These names will subsequently be used as indices in this data file and these names will also be used as the root file names for the scenario data files (each of these will have a .dat extension) if the parameter ScenarioBasedData is set to True, which is the default.

- set Stages: List of the names of the time stages, which must be given in time order. In the sequel we will use StageName to represent a node name used as an index.

- set Nodes: List of the names of the nodes in the scenario tree. In the sequel we will use NodeName to represent a node name used as an index.

- param NodeStage: A list of pairs of nodes and stages to indicate the stage for each node.

- param Parent: A list of node pairs to indicate the parent of each node that has a parent (the root node will not be listed).

- set Children[NodeName]: For each node that has children, provide the list of children. No sets will be give for leaf nodes.

- param ConditionalProbability: For each node in the scenario tree, give the conditional probability. For the root node it must be given as 1 and for the children of any node with children, the conditional probabilities must sum to 1.

- param ScenarioLeafNode: A list of scenario and node pairs to indicate the leaf node for each scenario.

- set StageVariables[StageName]: For each stage, list the pyomo model variables associated with that stage.

Data to instantiate these sets and parameters is provided by users in the file ScenarioStructure.dat, which can be given in AMPL [1] format.

The default behavior is one file per scenario and each file has the full data for the scenario. An alternative is to specify just the data that changes from the root node in one file per tree node. To select this option, add the following line to ScenarioStructure.dat:

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

This will set it up to want a per-node file, something along the lines of what's in `examples/pysp/farmer/NODEDATA`.

Advanced users may be interested in seeing the file `coopr/pysp/utils/scenariomodels.py`, which defines the python sets and parameters needed to describe stochastic elements. This file should not be edited.

3 Command Line Arguments

The basic PH algorithm is controlled by parameters that are set as command line arguments. Note that options begin with a double dash.

- `-h`, `--help`
  Show help message and exit.

- `--model-directory`=MODEL_DIRECTORY
  The directory in which all model (reference and scenario) definitions are stored. I.e., the ".py" files. Default is ".".

- `--instance-directory`=INSTANCE_DIRECTORY
  The directory in which all instance (reference and scenario) definitions are stored. I.e., the ".dat" files. Default is ".".

- `--verbose`
  Generate verbose output for both initialization and execution. Default is False.

- `--report-solutions`
  Always report PH solutions after each iteration. Enabled if –verbose is enabled. Default is False.

- `--report-weights`
  Always report PH weights prior to each iteration. Enabled if –verbose is enabled. Default is False.

- `--report-only-statistics`
  When reporting solutions (i.e. if –report-solutions has been selected), only output per-variable statistics - not the individual scenario values. Default is False.

- `--solver`=SOLVER_TYPE
  The type of solver used to solve scenario sub-problems. Default is cplex.

- `--solver-manager`=SOLVER_MANAGER_TYPE
  The type of solver manager used to coordinate scenario sub-problem solves. Default is serial. This option is changed in parallel applications as described in Section 8.

- `--max-iterations`=MAX_ITERATIONS
  The maximal number of PH iterations. Default is 100.

- `--default-rho`=DEFAULT_RHO
  The default (global) rho for all blended variables. Default is 1.

- `--rho-cfgfile`=RHO_CFGFILE
  The name of a configuration script to compute PH rho values. Default is None.

- `--enable-termdiff`-convergence
  Terminate PH based on the termdiff convergence metric. The convergcne metric is the unscaled sum of differences between variable values and the mean. Default is True.

- `--enable-normalized`-termdiff-convergence
  Terminate PH based on the normalized termdiff convergence metric. Each term in the termdiff sum is normalized by the average value (NOTE: it is NOT normalized by the number of scenarios). Default is False.

- `--termdiff-threshold`=TERMDIFF_THRESHOLD
  The convergence threshold used in the term-diff and normalized term-diff convergence criteria. Default is 0.01, which is too low for most problems.

- `--enable-free-discrete-count-convergence`
  Terminate PH based on the free discrete variable count convergence metric. Default is False.

- `--free-discrete-count-threshold`=FREE_DISCRETE_COUNT_THRESHOLD
  The convergence threshold used in the criterion based on when the free discrete variable count convergence criterion. Default is 20.

- `--enable-ww-extensions`
  Enable the Watson-Woodruff PH extensions plugin. Default is False.

- `--ww-extension-cfgfile`=WW_EXTENSION_CFGFILE
  The name of a configuration file for the Watson-Woodruff PH extensions plugin. Default is wwph.cfg.

- `--ww-extension-suffixfile`=WW_EXTENSION_SUFFIXFILE
  The name of a variable suffix file for the Watson-Woodruff PH extensions plugin. If the file name is not given, it defaults to wwph.suffixes. Important note: this option must be used to enable slamming. Without it, all slamming priorities will be zero.

- `--user-defined-extension`=EXTENSIONFILE
  Here, "EXTENSIONFILE" is the module name, which is in either the current directory (most likely) or somewhere on your PYTHONPATH. A simple example is "testphextension" plugin that simply prints a message to the screen for each callback. The file testphextension.py can be found in the sources directory and is shown in Section 4.3. A test of this would be to specify "-user-defined-extension=testphextension", assuming testphextension.py is in your PYTHONPATH or current directory. Note that both PH extensions (WW PH and your own) can co-exist; however, the WW plugin will be invoked first.

- `--scenario-solver-options`
  The options are specified just as in pyomo, e.g., `--scenario-solver-options="mip_tolerances_mip`
  to set the mipgap for all scenario sub-problem solves to 20% for the CPLEX

solver. The options are specified in a quote deliminted string that is passed to the sub-problem solver. Whatever options specified are persistent across all solves.

- `--ef-solver-options`
  The options are specified just as in pyomo, e.g., `--scenario-solver-options="mip_tolerances_mipg` to set the mipgap for all scenario sub-problem solves to 20% for the CPLEX solver. The options are specified in a quote deliminted string that is passed to the EF problem solver.

- `--write-ef`
  Upon termination, write the extensive form of the model - accounting for all fixed variables.

- `--solve-ef`
  Following write of the extensive form model, solve it.

- `--ef-output-file`=EF_OUTPUT_FILE
  The name of the extensive form output file (currently only LP format is supported), if writing of the extensive form is enabled. Default is efout.lp.

- `--suppress-continuous-variable-output`
  Eliminate PH-related output involving continuous variables. Default: no output.

- `--keep-solver-files`
  Retain temporary input and output files for scenario sub-problem solves. Default: files not kept.

- `--output-solver-logs`
  Output solver logs during scenario sub-problem solves. Default: no output.

- `--output-scenario-tree-solution`
  Report the full solution (including the leaves) in scenario tree format upon termination. Values at non-leaf nodes represent averages. Default is False.

- `--output-ef-solver-log`
  Output solver log during the extensive form solve. Default: no output.

- `--output-solver-results`
  Output solutions obtained after each scenario sub-problem solve. Default: no output.

- `--output-times`
  Output timing statistics for various PH components. Default: no output.

- `--disable-warmstarts`
  Disable warm-start of scenario sub-problem solves in PH iterations ¿= 1. Default=False (i.e., warm starts are the default).

- `--drop-proximal-terms`
  Eliminate proximal terms (i.e., the quadratic penalty terms) from the weighted PH objective. Default=False (i.e., but default, the proximal terms are included).

- `--retain-quadratic-binary-terms`
  Do not linearize PH objective terms involving binary decision variables. Default=False (i.e., the proximal term for binary variables is linearized by default; this can have some impact on the relaxations during the branch and bound solution process).

- `--linearize-nonbinary-penalty-terms`=BPTS
  Approximate the PH quadratic term for non-binary variables with a piecewise linear function. The argument BPTS gives the number of breakpoints in the linear approximation. The default=0. Reasonable non-zero values are usually in the range of 3 to 7. Note that if a breakpoint would be very close to a variable bound, then the break point is ommited. IMPORTANT: this option requires that all variables have bounds that are either established in the reference model or by code specfied using the bounds-cfgfile command line option. See Section 7 for more information about linearizing the proximal term.

- `--breakpoint-strategy`=BREAKPOINT_STRATEGY
  Specify the strategy to distribute breakpoints on the [lb, ub] interval of each variable when linearizing. 0 indicates uniform distribution. 1 indicates breakpoints at the node min and max, uniformly in- between. 2 indicates more aggressive concentration of breakpoints near the observed node min/max.

- `--bounds-cfgfile`=BOUNDS_CFGFILE
  The argument BOUNDS_CFGFILE specifies the name of an executable pyomo file that sets bounds. The devault is that there is no file. When specified, the code in this file is executed after the initialization of scenario data so the bounds can be based on data from all scenarios. The config subdirectory of the farmer example contains a simple example of such a file (boundsetter.cfg).

- `--ef-mipgap`=MIPGAP
  Specifies the mipgap for the EF solve (if there is an ef solve).

- `--checkpoint-interval`
  The number of iterations between writing of a checkpoint file. Default is 0, indicating never.

- `--restore-from-checkpoint`
  The name of the checkpoint file from which PH should be initialized. Default is not to restore from a checkpoint.

- `--profile=PROFILE`
  Enable profiling of Python code. The value of this option is the number of functions that are summarized. The default is no profiling.

- `--enable-gc`
  Enable the python garbage collecter. The default is no garbage collection.

4 Extensions via Callbacks

Basic PH can converge slowly, so it is usually advisable to extend it or modify it. In pysp, this is done via the pyomo plug-in mechanism. The basic PH implementation provides callbacks that enable access to the data structures used by the algorithm. In §4.1 we describe extensions that are provided with the release. In §4.3, we provide information to power users who may wish to modify or replace the extensions.

4.1 Watson and Woodruff Extensions

Watson and Woodruff describe innovations for accelerating PH [6], most of which are generalized and implemented in the file `wwextension.py`, but users generally do not need to know this file name. To invoke the program with these additional features, invoke the software with a command of the form:

```
runph --enable-ww-extensions
```

Many of the examples described in §5 use this plug-in. The main concept is that some integer variables should be fixed as the algorithm progresses for two reasons:

- Convergence detection: A detailed analysis of PH algorithm behavior on a variety of problem indicates that individual decision variables frequently converge to specific, fixed values scenarios in early PH iterations. Further, despite interactions among the the variables, the value frequently does not change in subsequent PH iterations. Such variable "fixing" behaviors lead to a potentially powerful, albeit obvious, heuristic: once a particular variable has been the same in all scenarios for some number of iterations, fix it to that value. For problems where the constraints effectively limit x from both sides, these methods may result in PH encountering infeasible scenario sub-problems even though the problem is ultimately feasible.
- Cycle detection: When there are integer variables, cycling is sometimes encountered, consequently, cycle detection and avoidance mechanisms are required to force eventual convergence of the PH algorithm in the mixed-integer case. To detect cycles, we focus on repeated occurrences of the weights, implemented using a simple hashing scheme [7] to minimize impact on run-time. Once a cycle in the weight vectors associated with any decision variable is detected, the value of that variable is fixed.

Fixing variables aggressively can result in shorter solution times, but can also result in solutions that are not as good. Furthermore, for some problems, aggressive fixing can result in infeasible sub-problems even though the problem is ultimately feasible. Many of the parameters discussed in the next subsections control fixing of variables. This is discussed in a tutorial in section 6.

4.1.1 Variable Specific Parameters

The plug-in makes use of parameters to control behavior at the variable level. Global defaults (to override the defaults stated here) should be set using methods described in §4.2.1. Values for each variable should be set using methods described in §4.2.2. Note that for variable fixing based on convergence detection, iteration zero is treated separately. The parameters are as follows:

- fix_continuous_variables: True or False. If true, fixing applies to all variables. If false, then fixing applies only to discrete variables.
- Iter0FixIfConvergedAtLB: 1 (True) or 0 (False). If 1, then discrete variables that are at their lower bound in all scenarios after the iteration zero solves will be fixed at that bound.
- Iter0FixIfConvergedAtUB: 1 (True) or 0 (False). If 1, then discrete variables that are at their upper bound in all sce¡narios after the iteration zero solves will be fixed at that bound.
- Iter0FixIfConvergedAtNB: = 1 1 (True) or 0 (False). If 1, then discrete variables that are at the same value in all scenarios after the iteration zero solves will be fixed at that value, without regard to whether it is a bound. If this is true, it takes precedence. A value of zero, on the other hand, implies that variables will not be fixed at at a non-bound.
- FixWhenItersConvergedAtLB: The number of consecutive PH iterations that discrete variables must be their lower bound in all scenarios before they will be fixed at that bound. A value of zero implies that variables will not be fixed at the bound.
- FixWhenItersConvergedAtUB: The number of consecutive PH iterations that discrete variables must be their upper bound in all scenarios before they will be fixed at that bound. A value of zero implies that variables will not be fixed at the bound.
- FixWhenItersConvergedAtNB: The number of consecutive PH iterations that discrete variables must be at the same, consistent value in all scenarios before they will be fixed at that value, without regard to whether it is a bound. If this is true, it takes precedence. A value of zero, on the other hand, implies that variables will not be fixed at at a non-bound.
- FixWhenItersConvergedContinuous: The number of consecutive PH iterations that continuous variables must be at the same, consistent value in all scenarios

before they will be fixed at that value. A value of zero implies that continuous variables will not be fixed.

- CanSlamToLB: True or False. If True, then slamming can be to the lower bound for any variable.
- CanSlamToMin: True or False. If True, then slamming can be to the minimum across scenarios for any variable.
- CanSlamToAnywhere: True or False. If True, then slamming can be to any value.
- CanSlamToMax: True or False. If True, then slamming can be to the maximum across scenarios for any variable.
- CanSlamToUB: True of False. If True, then slamming can be to the upper bound for any variable.
- DisableCycleDetection: True or False. If True, then cycle detection and the associated slamming are completely disabled. This cannot be changed to False on the fly because a value of True at startup causes creation of the cycle detection storage to be bypassed.

In the event that multiple slam targets are True, then Min and Max trump LB and UB while Anywhere trumps all.

4.2 General Parameters

The plug-in also makes use of the following parameters, which should be set using methods described in §4.2.1.

- Iteration0Mipgap: Gives the mipgap to be sent to the solver for iteration zero solves. The default is zero, which causes nothing to be sent to the solver (i.e., the solver uses its default mipgap).
- InitalMipGap: Gives the mipgap to be sent to the solver for iteration one solves. The default is zero, which causes nothing to be sent to the solver (i.e., the solver uses its default mipgap). If not zero, then this gap will be used as the starting point for the mipgap to change on each PH iteration in proportion to the convergence termination criterion so that when the criterion for termination is met the mipgap will be at (or near) the parameter value of FinalMipGap. If the InitialMipGap is significantly higher than the Iteration0MipGap parameter, the PH algorithm may perform poorly. This is because the iteration k-1 solutions are used to warm start iteration k-1 solves. If the iteration 1 mipgap is much higher than the iteration 0 mipgap, the iteration zero solution, although not optimal for the iteration one objective, might be within the mipgap. Default: 0.10.

- FinalMipGap: The target for the mipgap when PH has converged. Default: 0.001.
- PH_Iters_Between_Cycle_Slams: controls the number of iterations to wait after a variable is slammed due to hash hits that suggest convergence. Zero indicates unlimited slams per cycle. Default: 1.
- SlamAfterIter: Iteration number after which one variable every other iteration will be slammed to force convergence. Default: the number of scenarios.
- hash_hit_len_to_slam: Ignore possible cycles for which the only evidence of a cycle is less than this. Also, ignore cycles if any variables have been fixed in the previous hash_hit_len_to_slam PH iterations. Default: the number of scenarios. This default is often not a good choice. For many problems with a lot of scenarios, a value like 10 or 20 might be a lot better if rapid convergence is desired.
- W_hash_history_len: This obscure parameter controls how far back the code will look to see if there is a possible cycle. Default: max(100, number of scenarios).

4.2.1 Setting Parameter Values

The parameters of PH and of any callbacks can be changed using the file wwph.cfg, which is executed by the python interpreter after PH has initialized. This is a potentially powerful and/or dangerous procedure because it gives an opportunity to change anything using the full features of python and pyomo.

4.2.2 Setting Suffix Values

Suffixes are set using the data file named `wwph.suffixes` using this syntax:

```
VARSPEC SUFFIX VALUE
```

where VARSPEC is replaced by a variable specification, SUFFIX is replaced by a suffix name and VALUE is replaced by the value of the suffix for that variable or those variables. Here is an example:

```
   Delta CanSlamToLB False
   Gamma[*,Ano1] SlammingPriority 10
   Gamma[*,Ano2] SlammingPriority 20
   ...
```

4.3 Callback Details

Most users of pysp can skip this subsection. A callback class definition named iphextension is in the file iphextension.py and can be used to implement callbacks at a variety of points in PH. For example, the method post_iteration_0_solves is called immediately after all iteration zero solves, but before averages and weights have been computed while the method post_iteration_0 is called after averages and weights based on iteration zero have been computed. The file iphextension is in the coopr/pysp directory and is not intended to be edited by users.

The user defines a class derived from SingletonPlugin that implements iphextension. Its name is given to ph as an option. This class will be automatically instantiated by ph. It has access to data and methods in the PH class, which are defined in the file ph.py. An example of such a class is in the file named testphextension.py in the pysp example directory.

A user defined extension file can be incorporated by using the command line option: `--user-defined-extension=EXTENSIONFILE`. Here, "EXTENSIONFILE" is the module name, which is in either the current directory (most likely) or somewhere on your PYTHONPATH. A simple example is "testphextension" plugin that simply prints a message to the screen for each callback. The file testphextension.py can be found in the sources directory and given in Section 4. An easy test of this would be to specify "-user-defined-extension=testphextension" and you should note the the ".py" file extension is not included on the runph command line.

Both your own extension and the WWPH extensions can co-exist; however, the WW plugin will be invoked first at each callback point if both are included.

Here are the callbacks:

- post_ph_initialization: Called after PH data structures have been intialized but before iteration zero solves.
- post_iteration_0_solves: Called after iteration zero solutions and some statistics such as averages have been computed, but before weights are updated.
- post_iteration_0: Called after all processing for iteration zero is complete.
- post_iteration_k_solves: Called after solutions some statistics such as averages have been computed, but before weights are updated for iterations after iteration zero.
- post_iteration_k: Called after all processing for each iteration after iteration 0 is complete.
- post_ph_execution: Called execution is complete.

Users interested in writing their own extensions will probably want to refer to the source file ph.py to get a deeper understanding of when the callback occur.

5 Examples

A number of examples are provided with pysp.

5.1 Farmer Example

This two-stage example is composed of models and data for the "Farmer" stochastic program, introduced in Section 1.1 of "Introduction to Stochastic Programming" by Birge and Louveaux [2].

- ReferenceModel.py: a single-scenario model for the SP
- ReferenceModel.dat: a single-scenario data file for the SP (any scenario will do - used to flush out variable and constraint index sets)
- ScenarioStructure.dat: data file defining the scenario tree.
- AboveAverageScenario.dat: one of the scenario data files.
- BelowAverageScenario.dat: one of the scenario data files.
- AverageScenario.dat: one of the scenario data files.

The command runph executes PH, assuming the ReferenceModel.* and ScenarioStructure.* files are present and correct. This example is probably in a directory with a name something like:

```
coopr\examples\pysp\farmer
```

The data is in a subdirectory called scenariodata and the model is in the models subdirectory. To invoke PH for this problem, connect to this farmer directory and use the command:

```
runph --model-directory=models --instance-directory=scenariodata
```

5.2 Sizes Example

This two-stage example is composed of models and data for the "Sizes" stochastic program [3, 4], which consists of the following files:

- wwph.cfg: replace default algorithm parameter values for the Watson and Woodruff extensions.
- wwph.suffixes: sets algorithm parameter values at the variables level for the Watson and Woodruff extensions.

- ReferenceModel.py: a single-scenario model for the SP
- ReferenceModel.dat: a single-scenario data file for the SP (any scenario will do - used to flush out variable and constraint index sets)
- ScenarioStructure.dat: data file defining the scenario tree.
- Scenario1.dat: one of the scenario data files.
- Scenario2.dat: one of the scenario data files.
- ...

This example is probably in a directory with a name something like:

```
coopr\packages\coopr\examples\pysp\sizes
```

The data for a three scenario version is in a subdirectory called SIZES3 and a ten scenario dataset is in SIZES10.

To invoke PH for the 10 scenario problem, connect to the sizes directory and use the command:

```
runph --model-directory=models --instance-directory=SIZES10
```

5.3 Forestry Example

This four-stage example is composed of models and data for the "forestry" stochastic program [], which consists of the following files:

- wwph.cfg: replace default algorithm parameter values for the Watson and Woodruff extensions.
- wwph.suffixes: sets algorithm parameter values at the variables level for the Watson and Woodruff extensions.
- ReferenceModel.py: a single-scenario model for the SP
- ReferenceModel.dat: a single-scenario data file for the SP (any scenario will do - used to flush out variable and constraint index sets)
- ScenarioStructure.dat: data file defining the scenario tree.
- Scenario1.dat: one of the scenario data files.
- Scenario2.dat: one of the scenario data files.
- ...

This example is probably in a directory with a name something like:

```
coopr\packages\coopr\examples\pysp\forestry
```

There are two families of instances: "Chile" and "Davis," each with four stages and eighteen scenarios. This is also a small two-stage, four scenario instances in the subdirectory DAVIS2STAGE.

This full 18 scenario problem instance takes too long without the Watson Woodruff extensions, but to invoke PH for this problem without them, connect to the forestry directory and use the command:

```
 runph --models-directory=models --instancs-directory=chile
```

and to run with the extensions, use

```
runph --model-directory=models --instance-directory=davis \
--enable-ww-extensions --ww-extension-cfgfile=config/wwph.cfg \
--ww-extension-suffixfile=config/wwph.suffixes
```

6 Tutorial: Parameters for Watson and Woodruff PH Extensions

The parameters for the PH extensions in WWPHExtensions.py provide the user with considerable control over how and under what conditions variables are fixed during the PH algorithm execution. Often, some variables converge to consistent values during early iterations and can be fixed at these values without affecting quality very much and without affecting feasibility at all. Also, the algorithm may need to fix some variables during execution in order to break cycles. In both cases, guidance from the user concerning which classes of variables can and/or should be fixed under various circumstances can be very helpful.

The overarching goal is to support industrial and government users who solve the same model repeatedly with different, but somewhat similar, data each time. In such settings, it behooves the modeler to consider tradeoffs between speed, solution quality and feasibility and create at least one good set of directives and parameters for the PH algorithm. In some cases, a user may want more than one set of directives and parameters depending on whether speed of execution or quality of solution are more important. Iteration zero is controlled separately because often the absence of the quadratic term results in faster solves for this iteration and fixing variables after the iteration has the maximum possible impact on speedup.

In order to discuss these issues, we consider an example.

6.1 Sizes Example

The Sizes example is simple and small. In fact, the instances that we will consider are so small that there is really no need to use the PH algorithm since the extensive form can be solved directly in a few minutes. However, it serves as a vehicle for introducing the concepts.

This description and formulation comes from the original paper by Jorjani, Scott and Woodruff [3].

If demand constraints for the current period are based on firm orders but future demands are based on forecasts or conjecture, then they should not be treated in the same fashion. We must recognize that future demand constraints are stochastic. That is to say that they should be modeled as random variables. It may not be reasonable or useful to consider the entire demand probability distribution functions. It may not be reasonable because there may not be sufficient data to estimate an entire distribution. It may not be useful because the essence of the stochastics may be captured by specifying a small number of representative *scenarios.* We assume that scenarios are specified by giving a full set of random variable realizations and a corresponding probability. We index the scenario set, $\mathcal{L}$, by ℓ and refer to the probability of occurrence of ℓ (or, more accurately, a realization "near" scenario ℓ) as $\Pr(\ell)$. We refer to solution systems that satisfy constraints with probability one as *admissible.*

In addition to modeling stochastics, we would like to model *recourse* as well. That is, we would like to model the ability of decision makers to make use of new information (e.g., orders) at the start of each planning period. We allow our solution vectors to depend on the scenario that is realized, but we do not want to assume prescience. We refer to a system of solution vectors as *implementable* if for all decision times t, the solution vector elements corresponding to period $1, \ldots, t$ are constant with respect to information that becomes available only after stage t. We refer to the set of implementable solutions as $\mathcal{N}_{\mathcal{L}}$. It is possible to require implementable solutions by adding *non-anticipatitivity constraints*, but we will instead make use of solution procedures that implicitly guarantee implementable solutions.

In the stochastic, multi-period formulation that follows the objective is to minimize expected costs. We invoke the network equivalence given earlier to drop the explicit requirement that $\boldsymbol{x}$ and $\boldsymbol{y}$ be integers. Variables and data are subscripted with a period index t that takes on values up to T. To model the idea that sleeves produced in one period can be used as-is or cut in subsequent periods, we use x_{ijt} to indicate that sleeves of length index i are to be used without cutting in period t if $i = j$ and with cutting otherwise. The $\boldsymbol{y}$ vector gives production quantities for each length in each period without regard to the period in which they will be used (and perhaps cut). The formulation is essentially an extension of ILP except that a capacity constraint must be added in the multiple period formulation. Holding costs could be added, but an additional subscript becomes necessary without the benefit of any additional insight. As an aside, note that the addition of holding costs would add a large number of continuous variables, but no new integers so the impact on computational performance would not be catastrophic.

$$\min \sum_{\ell \in \mathcal{L}} \Pr(\ell) \sum_{t}^{T} \left[\sum_{i=1}^{N} (s z_{it\ell} + p_i y_{it\ell}) + r \sum_{j<i} x_{ijt\ell} \right] \quad \text{(SMIP)}$$

subject to

$$\begin{aligned}
\sum_{j \geq i} x_{ijt\ell} &\geq d_{it\ell} \quad \ell \in \mathcal{L},\ i=1,\ldots,N,\ t=1,\ldots,T && (1)\\
\sum_{t' \leq t} \left[\sum_{j \leq i} x_{ijt'\ell} - y_{it'\ell} \right] &\leq 0 \quad \ell \in \mathcal{L},\ i=1,\ldots,N,\ t=1,\ldots,T && (2)\\
y_{it\ell} - M z_{it\ell} &\leq 0 \quad \ell \in \mathcal{L},\ i=1,\ldots,N,\ t=1,\ldots,T && (3)\\
\sum_{i=1}^{N} y_{it\ell} &\leq c_{t\ell} \quad \ell \in \mathcal{L},\ t=1,\ldots,T && (4)\\
z_{it\ell} &\in \{0,1\} \quad \ell \in \mathcal{L},\ i=1,\ldots,N,\ t=1,\ldots,T && (5)\\
\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z} &\in \mathcal{N}_{\mathcal{L}} && (6)
\end{aligned}$$

Bear in mind that solution vector elements corresponding to periods two through T are not actually intended for use, they are computed just to see the effect that period 1 (the current period) decision would have on future optimal behavior. At the start of period 2 – or at any other time – the decision maker would run the model again with updated demands for the current period and new scenario estimates.

6.2 ReferenceModel.py

Here is the single scenario reference model:

```
#
# This is the two-period version of the SIZES optimization model.
#

from coopr.pyomo import *

#
# Model
#

model = Model()

#
# Parameters
#
```

```
# the number of product sizes.
model.NumSizes = Param(within=NonNegativeIntegers)

def product_sizes_rule(model):
    ans = set()
    for i in range(1, model.NumSizes()+1):
        ans.add(i)
    return ans

# the set of sizes, labeled 1 through NumSizes.
model.ProductSizes = Set(initialize=product_sizes_rule)

# the deterministic demands for product at each size.
model.DemandsFirstStage = Param(model.ProductSizes, within=NonNegativeIntegers)
model.DemandsSecondStage = Param(model.ProductSizes, within=NonNegativeIntegers)

# the unit production cost at each size.
model.UnitProductionCosts = Param(model.ProductSizes, within=NonNegativeReals)

# the setup cost for producing any units of size i.
model.SetupCosts = Param(model.ProductSizes, within=NonNegativeReals)

# the unit penalty cost of meeting demand for size j with larger size i.
model.UnitPenaltyCosts = Param(model.ProductSizes, within=NonNegativeReals)

# the cost to reduce a unit i to a lower unit j.
model.UnitReductionCost = Param(within=NonNegativeReals)

# a cap on the overall production within any time stage.
model.Capacity = Param(within=PositiveReals)

# a derived set to constrain the NumUnitsCut variable domain.
def num_units_cut_domain_rule(model):
   ans = set()
   for i in range(1,model.NumSizes()+1):
      for j in range(1, i+1):
         ans.add((i,j))
   return ans

model.NumUnitsCutDomain = Set(initialize=num_units_cut_domain_rule, dimen=2)

#
# Variables
#

# are any products at size i produced?
model.ProduceSizeFirstStage = Var(model.ProductSizes, domain=Boolean)
model.ProduceSizeSecondStage = Var(model.ProductSizes, domain=Boolean)

# NOTE: The following (num-produced and num-cut) variables are implicitly integer
#       under the normal cost objective, but with the PH cost objective, this isn't
#       the case.

# the number of units at each size produced.
model.NumProducedFirstStage = Var(model.ProductSizes, domain=NonNegativeIntegers)
model.NumProducedSecondStage = Var(model.ProductSizes, domain=NonNegativeIntegers)

# the number of units of size i cut (down) to meet demand for units of size j.
model.NumUnitsCutFirstStage = Var(model.NumUnitsCutDomain, domain=NonNegativeIntegers)
model.NumUnitsCutSecondStage = Var(model.NumUnitsCutDomain, domain=NonNegativeIntegers)

# stage-specific cost variables for use in the pysp scenario tree / analysis.
model.FirstStageCost = Var(domain=NonNegativeReals)
model.SecondStageCost = Var(domain=NonNegativeReals)

#
# Constraints
#

# ensure that demand is satisfied in each time stage, accounting for cut-downs.
def demand_satisfied_first_stage_rule(i, model):
   return (0.0, \
           sum(model.NumUnitsCutFirstStage[j,i] \
               for j in model.ProductSizes if j >= i) - model.DemandsFirstStage[i], \
           None)

def demand_satisfied_second_stage_rule(i, model):
   return (0.0, \
           sum(model.NumUnitsCutSecondStage[j,i] \
               for j in model.ProductSizes if j >= i) - model.DemandsSecondStage[i], \
           None)

model.DemandSatisfiedFirstStage = \
                Constraint(model.ProductSizes, rule=demand_satisfied_first_stage_rule)
model.DemandSatisfiedSecondStage = \
                Constraint(model.ProductSizes, rule=demand_satisfied_second_stage_rule)

# ensure that you don’t produce any units if the decision has been made to disable production.
def enforce_production_first_stage_rule(i, model):
   # The production capacity per time stage serves as a simple upper bound for "M".
   return (None, \
           model.NumProducedFirstStage[i] - model.Capacity * model.ProduceSizeFirstStage[i], \
           0.0)

def enforce_production_second_stage_rule(i, model):
   # The production capacity per time stage serves as a simple upper bound for "M".
   return (None, \
           model.NumProducedSecondStage[i] - model.Capacity * model.ProduceSizeSecondStage[i], \
           0.0)

model.EnforceProductionBinaryFirstStage = \
                  Constraint(model.ProductSizes, rule=enforce_production_first_stage_rule)
```

```
model.EnforceProductionBinarySecondStage = \
                  Constraint(model.ProductSizes, rule=enforce_production_second_stage_rule)

# ensure that the production capacity is not exceeded for each time stage.
def enforce_capacity_first_stage_rule(model):
   return (None, \
           sum(model.NumProducedFirstStage[i] for i in model.ProductSizes) - model.Capacity, \
           0.0)

def enforce_capacity_second_stage_rule(model):
   return (None, \
           sum(model.NumProducedSecondStage[i] for i in model.ProductSizes) - model.Capacity, \
           0.0)

model.EnforceCapacityLimitFirstStage = Constraint(rule=enforce_capacity_first_stage_rule)
model.EnforceCapacityLimitSecondStage = Constraint(rule=enforce_capacity_second_stage_rule)

# ensure that you can't generate inventory out of thin air.
def enforce_inventory_first_stage_rule(i, model):
   return (None, \
           sum(model.NumUnitsCutFirstStage[i,j] \
               for j in model.ProductSizes if j <= i) - model.NumProducedFirstStage[i], \
           0.0)

def enforce_inventory_second_stage_rule(i, model):
   return (None, \
           sum(model.NumUnitsCutFirstStage[i,j] \
               for j in model.ProductSizes \
                  if j <= i) + sum(model.NumUnitsCutSecondStage[i,j] \
                     for j in model.ProductSizes if j <= i) \
               - model.NumProducedFirstStage[i] - model.NumProducedSecondStage[i], \
           0.0)

model.EnforceInventoryFirstStage = \
               Constraint(model.ProductSizes, rule=enforce_inventory_first_stage_rule)
model.EnforceInventorySecondStage = \
               Constraint(model.ProductSizes, rule=enforce_inventory_second_stage_rule)

# stage-specific cost computations.
def first_stage_cost_rule(model):
   production_costs = \
            sum(model.SetupCosts[i] * model.ProduceSizeFirstStage[i] \
                + model.UnitProductionCosts[i] * model.NumProducedFirstStage[i] \
                for i in model.ProductSizes)
   cut_costs = \
             sum(model.UnitReductionCost * model.NumUnitsCutFirstStage[i,j] \
                 for (i,j) in model.NumUnitsCutDomain if i != j)
   return (model.FirstStageCost - production_costs - cut_costs) == 0.0

model.ComputeFirstStageCost = Constraint(rule=first_stage_cost_rule)
```

```
def second_stage_cost_rule(model):
   production_costs = \
                sum(model.SetupCosts[i] * model.ProduceSizeSecondStage[i] \
                    + model.UnitProductionCosts[i] * model.NumProducedSecondStage[i] \
                    for i in model.ProductSizes)
   cut_costs = \
             sum(model.UnitReductionCost * model.NumUnitsCutSecondStage[i,j] \
                 for (i,j) in model.NumUnitsCutDomain if i != j)
   return (model.SecondStageCost - production_costs - cut_costs) == 0.0

model.ComputeSecondStageCost = Constraint(rule=second_stage_cost_rule)

#
# Objective
#

def total_cost_rule(model):
   return (model.FirstStageCost + model.SecondStageCost)

model.TotalCostObjective = Objective(rule = total_cost_rule, sense=minimize)
```

6.3 ReferenceModel.dat

This file establishes the data for a representative instance. The main thing to notice is that the indexes for sizes happen to be given as integers, which is not required: they could have been strings.

```
#ReferenceModel.dat
param NumSizes := 10 ;

param Capacity := 200000 ;

param DemandsFirstStage := 1 2500 2 7500 3 12500 4 10000 5 35000
                           6 25000 7 15000 8 12500 9 12500 10 5000 ;

param DemandsSecondStage := 1 2500 2 7500 3 12500 4 10000 5 35000
                            6 25000 7 15000 8 12500 9 12500 10 5000 ;

param UnitProductionCosts := 1 0.748 2 0.7584 3 0.7688 4 0.7792 5 0.7896
                             6 0.8 7 0.8104 8 0.8208 9 0.8312 10 0.8416 ;

param SetupCosts := 1 453 2 453 3 453 4 453 5 453 6 453 7 453 8 453 9 453 10 453 ;

param UnitReductionCost := 0.008 ;
```

6.4 ScenarioStucture.dat

Here is the data file that describes the stochastics:

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

set Stages := FirstStage SecondStage ;

set Nodes := RootNode
             Scenario1Node
             Scenario2Node
             Scenario3Node
             Scenario4Node
             Scenario5Node
             Scenario6Node
             Scenario7Node
             Scenario8Node
             Scenario9Node
             Scenario10Node ;

param NodeStage := RootNode             FirstStage
                   Scenario1Node        SecondStage
                   Scenario2Node        SecondStage
                   Scenario3Node        SecondStage
                   Scenario4Node        SecondStage
                   Scenario5Node        SecondStage
                   Scenario6Node        SecondStage
                   Scenario7Node        SecondStage
                   Scenario8Node        SecondStage
                   Scenario9Node        SecondStage
                   Scenario10Node       SecondStage ;

set Children[RootNode] := Scenario1Node
                          Scenario2Node
                          Scenario3Node
                          Scenario4Node
                          Scenario5Node
                          Scenario6Node
                          Scenario7Node
                          Scenario8Node
                          Scenario9Node
                          Scenario10Node ;

param ConditionalProbability := RootNode          1.0
                                Scenario1Node     0.10
                                Scenario2Node     0.10
                                Scenario3Node     0.10
                                Scenario4Node     0.10
                                Scenario5Node     0.10
                                Scenario6Node     0.10
                                Scenario7Node     0.10
                                Scenario8Node     0.10
                                Scenario9Node     0.10
                                Scenario10Node    0.10 ;
```

```
set Scenarios := Scenario1
                 Scenario2
                 Scenario3
                 Scenario4
                 Scenario5
                 Scenario6
                 Scenario7
                 Scenario8
                 Scenario9
                 Scenario10 ;

param ScenarioLeafNode := Scenario1  Scenario1Node
                          Scenario2  Scenario2Node
                          Scenario3  Scenario3Node
                          Scenario4  Scenario4Node
                          Scenario5  Scenario5Node
                          Scenario6  Scenario6Node
                          Scenario7  Scenario7Node
                          Scenario8  Scenario8Node
                          Scenario9  Scenario9Node
                          Scenario10 Scenario10Node ;

set StageVariables[FirstStage] := ProduceSizeFirstStage[*]
                                  NumProducedFirstStage[*]
                                  NumUnitsCutFirstStage[*,*] ;
set StageVariables[SecondStage] := ProduceSizeSecondStage[*]
                                   NumProducedSecondStage[*]
                                   NumUnitsCutSecondStage[*,*] ;

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

6.5 wwph.cfg

This file overrides default values for the suffixes used by WW PH Extension. Even when most of them will be overridden at the variable level, it makes sense to provide instance specific defaults. For this problem, it does not seem helpful or prudent to do any fixing of continuous variables since they only used to compute the objective function terms. The `ProduceSizeFirstStage` variables are binary and fixing their values would seem to determine the value of the other values. The other first stage variables are general integers. The `ProduceSizeFirstStage` can safely be fixed because provided that the largest size is not fixed at zero, there is little risk of infeasibility since larger sizes can be cut (at a cost) to meet demand for smaller sizes. Consequently, it is safe to let the algorithm slam those variables as needed. Slamming the other variables is riskier because they could get slammed to values that don't make sense given the values of the `ProduceSizeFirstStage` variables.

Fixing variables that have converged will speed the algorithm, perhaps resulting in a less desirable objective function value. It would make sense to tend to fix the `ProduceSizeFirstStage` before fixing the others to avoid inconsistencies and because the The `ProduceSizeFirstStage` variables have a strong tendency to imply values for the other variables.

Here is a sensible and internally consistent, if a bit conservative, wwph.cfg file:

```
# wwph.cfg
# python  commands that are executed by the wwphextensions.py file
# to set parameters and parameter defaults

self.fix_continuous_variables = False

self.Iteration0MipGap = 0.02
self.InitialMipGap = 0.10
self.FinalMipGap = 0.001

# for all six of these, zero means don't do it.
self.Iter0FixIfConvergedAtLB = 0 # 1 or 0
self.Iter0FixIfConvergedAtUB = 0  # 1 or 0
self.Iter0FixIfConvergedAtNB = 0  # 1 or 0 (converged to a non-bound)
self.FixWhenItersConvergedAtLB = 0
self.FixWhenItersConvergedAtUB = 25
self.FixWhenItersConvergedAtNB = 0  # converged to a non-bound
self.FixWhenItersConvergedContinuous = 0

# "default" slamming parms
# True and False are the options (case sensitive)
self.CanSlamToLB = False
self.CanSlamToMin = False
self.CanSlamToAnywhere = False
self.CanSlamToMax = True
self.CanSlamToUB = False
self.PH_Iters_Between_Cycle_Slams = 5

# the next line will try to force at least one variable to be
# fixed every other iteration after iteration 50
# if anything can be slammed
self.SlamAfterIter = 50

self.hash_hit_len_to_slam = 50
self.W_hash_history_len = 100
```

There are a number of things to notice about the contents of this file. Since it is executed as python code, the syntax matters. Users should changes values of numbers of change True to False, but everything else should not be edited with the exception of comments, which is any text after a sharp sign (sometimes called a pound sign). Changes to this file should be tested incrementally because errors are trapped by the python interpreter and may be difficult for non-python programmers to decipher.

This particular example file allows variables to be fixed if all scenarios have agreed on the upper bound for five iterations in a row. Since the `ProduceSizeFirstStage` variables are the only discrete variables in the first stage with an upper bound, they are the only variables affected by this.

This example allows slamming, but only to the max across scenarios. This is different than the upper bound, even for binary variables, because a variable could be selected for slamming for which all scenarios have agreed on the same value (which could be the lower lower bound). Data providing an override for this default as well as control over the selection priority for variables to slam are provided in the wwph.suffixes file.

6.6 wwph.suffixes

```
# Optional suffixes to help control PH

# Text triples specifying (variable/variable-index, suffix-name, suffix-value)
# tuples.

# override the defaults given in wwph.cfg as needed
# If no scenario needs the smallest size to be produced, then just forget it
ProduceSizeFirstStage[1] Iter0FixIfConvergedAtLB 1

# if all scenarios need either of the two largest sizes, just lock them in
ProduceSizeFirstStage[9] Iter0FixIfConvergedAtUB 1
ProduceSizeFirstStage[10] Iter0FixIfConvergedAtUB 1

# and be aggressive with the smallest size after iteration 0
ProduceSizeFirstStage[1] FixWhenItersConvergedAtLB 8

# if the production quantities have been fixed a long time, fix them
NumProducedFirstStage[*] FixWhenItersConvergedAtNB 20

# don’t allow slamming of variables other than ProduceSize
# for completeness, disallow all slamming destinations

NumProducedFirstStage[*] CanSlamToLB False
NumProducedFirstStage[*] CanSlamToMin False
NumProducedFirstStage[*] CanSlamToAnywhere False
NumProducedFirstStage[*] CanSlamToMax False
NumProducedFirstStage[*] CanSlamToUB False

NumUnitsCutFirstStage[*,*] CanSlamToLB False
NumUnitsCutFirstStage[*,*] CanSlamToMin False
NumUnitsCutFirstStage[*,*] CanSlamToAnywhere False
NumUnitsCutFirstStage[*,*] CanSlamToMax False
NumUnitsCutFirstStage[*,*] CanSlamToUB False

# higher priority means slam these first
```

```
ProduceSizeFirstStage[1] SlammingPriority 1
ProduceSizeFirstStage[2] SlammingPriority 2
ProduceSizeFirstStage[3] SlammingPriority 3
ProduceSizeFirstStage[4] SlammingPriority 4
ProduceSizeFirstStage[5] SlammingPriority 5
ProduceSizeFirstStage[6] SlammingPriority 6
ProduceSizeFirstStage[7] SlammingPriority 7
ProduceSizeFirstStage[8] SlammingPriority 8
ProduceSizeFirstStage[9] SlammingPriority 9
ProduceSizeFirstStage[10] SlammingPriority 10
```

The first thing to notice is that this is a data file and not a file of Python comments. Consequently, simpler messages are provided if there are errors, but the file can be verbose and a computer program or spreadsheet might be required to produce this data file. The next thing to notice is that indexes can be specified either using valid index values, or wildcards.

This file overrides the defaults to allow some fixing after iteration zero. Fixing the smallest size (index 1) to zero cannot result in infeasibility because larger sizes can be cut to satisfy demand for that size. Along the same lines, aggressively fixing the production indicator for larger sizes to 1 is also safe and perhaps not sub-optimal if all scenarios "want" those sizes anyway.

6.7 rhosetter.cfg

```
# rhosetter.cfg

# users probably want this line intact so they can use model_instance
model_instance = self._model_instance

MyRhoFactor = 0.1

for i in model_instance.ProductSizes:
   self.setRhoAllScenarios(model_instance.ProduceSizeFirstStage[i], model_instance.SetupC
   self.setRhoAllScenarios(model_instance.NumProducedFirstStage[i], model_instance.UnitPr
   for j in model_instance.ProductSizes:
      if j <= i:
         self.setRhoAllScenarios(model_instance.NumUnitsCutFirstStage[i,j], model_instanc
```

6.8 Execution

To run this example, connect to the sizes directory, which is something like:

```
coopr\examples\pysp\sizes
```

Then use

```
runph --model-directory=models --instance-directory=SIZES10 \
--enable-ww-extensions --ww-extension-cfgfile=config/wwph.cfg \
--ww-extension-suffixfile=config/wwph.suffixes \
--rho-cfgfile=config/rhosetter.cfg
```

Since this instance is so small by modern standards, the enhancement are not needed and may even increase the total solution time. It is provided to illustrate the features of the extensions, not to illustrate why you might need them. Much larger instances are are required for that.

7 Linearizing the Proximal Term

7.1 Introduction

For a decision variable x the proximal term added to the objective function for each scenario subproblem at PH iteration k is

$$\left(x - \bar{x}^{(k-1)}\right)^2$$

where $\bar{x}^{(k-1)}$ is the average over the scenarios from the last iteration. Expanding the square reveals that the only quadratic term is x^2. For binary variables, this is equal to x, although this is not the case when a relaxation is solved. For binary variables, the default behaviour is to replace the quadratic term with x, but an option allows the quadratic to be retained because this can effect the subproblem solution time due to the use of the quadratic term when the relaxations are solved as part of the branch and bound process.

For non-binary variables an option exists to replace the x^2 term with a piece-wise approximation and the number of breakpoints in the approximation is under user control. This can have a significant effect on CPU time required to solve subproblems because the presence of the quadratic term increases the solution times significantly. However, linearization results in a time-quality tradeoff because increasing the number of breakpoints increases the fidelity but each breakpoint introduces another (unseen) binary variable so solution times are generally increased.

A few strategies for placing the breakpoints are supported as command a line options: `--breakpoint-strategy`=BREAKPOINT_STRATEGY. A value of 0 indicates uniform distribution. 1 indicates breakpoints at the min and max for the node in the scenario tree, uniformly in- between. A value of 2 indicates more aggressive concentration of breakpoints near the observed node min/max.

Upper and lower bounds for variables must be specified when the linearization option is chosen. This can be done by specifying bounds in the reference model or by using the bounds-cfgfile command line option. It is, of course, best to use meaningful bounds provided in the reference model; however, the modeller must be careful not use estimated bounds that are too tight since that could preclude an optimal (or even a feasible) solution to the overall stochastic problem even though it might not cut off any optimal solutions for the particular scenario. The use of a bounds cfgfile is an advanced topic, but enables the modeller to use bounds that cannot create infeasibilities.

7.2 Bounds-cfgfile

Using a bounds cfgfile is an advanced topic. The modeler is writing python/pyomo code that will be executed by the ph.py file that is the core of the PH algorithm. The first executable line in a bounds file is typically:

```
model_instance = self._model_instance
```

This establishes a local variable called "model_instance" that can be used to refer to the model (of course, a different name can be used, such as MODINS). The object "self" on the right hand side of this assignment refers to the core PH object. The model, in turn contains the parameters and variables defined in the ReferenceModel.py file that can be accessed by name. For example, with the Farmer model, the cfg file sets the uppter bound on DevotedAcreage to be value of the paramter TOTAL_ACREAGE at intialization (since this is Python, the parentheses after TOTAL_ACREAGE cause the value in TOTAL_ACREAGE to be assigned rather than the name of the parameter object:

```
# only need to set upper bounds on first-stage variables, i.e., those being
# blended.
model_instance = self._model_instance

# the values supplied to the method
upper_bound = float(model_instance.TOTAL_ACREAGE())

for index in model_instance.CROPS:
   # the lower and upper bounds are expected to be floats, and trigger an
   # exception if not.
   self.setVariableBoundsAllScenarios("DevotedAcreage", index, 0.0, upper_bound)
```

The same thing could be accomplished by setting the upper bound in the model file, but it does serve as a simple example of a bounds cfg file.

8 Solving sub-problems in Parallel

Pyomo makes use of the pyro facility to enable sub-problems to easily be assigned to be solved in parallel. This capability is suppored by pysp. We will refer to a single master computer and multiple slave computers in the discussion here, but actually, the master computing processes can (and for synchronous parallel, should) be on a processor that also runs a slave process.

Here are the commands in order:

1. On the master: `coopr-ns`
2. On the master: `dispatch_srvr`
3. On each slave: `pyro_mip_server`
4. On the master: `runph ... --solver-manager=pyro ...`

Note that the command `coopr-ns` and the argument `solver-manger` have a dash in the middle, while the commands `dispatch_srvr` and `pyro_mip_server` have underscores. The first three commands launch processes that have no internal mechanism for termination; i.e., they will be terminated only if they crash or if they are killed by an external process. It is common to launch these processes with output redirection, such as `coopr-ns >& cooprns.log`. The `runph` command is a normal runph command with the usual arguments with the additional specification that subproblem solves should be directed to the pyro solver manager.

References

[1] AMPL home page. `http://www.ampl.com/`, 2008.

[2] J.R. Birge and F. Louveaux. *Introduction to Stochastic Programming*. Springer, 1997.

[3] S. Jorjani, C.H. Scott, and D.L. Woodruff. Selection of an optimal subset of sizes. *International journal of production research*, 37:3697–3710, 1999.

[4] A. Løkketangen and D.L. Woodruff. Progressive hedging and tabu search applied to mixed-integer (0,1) multistage stochastic programming. *Journal of Heuristics*, 2:111–128, 1996.

[5] R.T. Rockafellar and R. J-B. Wets. Scenarios and policy aggregation in optimization under uncertainty. *Mathematics of Operations Research*, pages 119–147, 1991.

[6] J-P Watson and D.L. Woodruff. Progressive hedging innovations for a class of stochastic mixed-integer resource allocation problems. *Computational Management Science*, page to appear, 2010.

[7] D.L. Woodruff and E. Zemel. Hashing vectors for tabu search. *Annals of Operations Research*, 41:123–137, 1993.

Acknowledgments

Sandia is a multipurpose laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.