Math. Prog. Comp. (2012) 4:109–149
DOI 10.1007/s12532-012-0036-1

FULL LENGTH PAPER

PySP: modeling and solving stochastic programs in Python

Jean-Paul Watson · David L. Woodruff · William E. Hart

Received: 7 September 2010 / Accepted: 2 February 2012 / Published online: 7 March 2012

Abstract Although stochastic programming is a powerful tool for modeling decision-making under uncertainty, various impediments have historically prevented its wide-spread use. One factor involves the ability of non-specialists to easily express stochastic programming problems as extensions of their deterministic counterparts, which are typically formulated first. A second factor relates to the difficulty of solving stochastic programming models, particularly in the mixed-integer, non-linear, and/or multi-stage cases. Intricate, configurable, and parallel decomposition strategies are frequently required to achieve tractable run-times on large-scale problems. We simultaneously address both of these factors in our PySP software package, which is part of the Coopr open-source Python repository for optimization; the latter is distributed as part of IBM's COIN-OR repository. To formulate a stochastic program in PySP, the user specifies both the deterministic base model (supporting linear, non-linear, and mixed-integer components) and the scenario tree model (defining the problem stages and the nature of uncertain parameters) in the Pyomo open-source algebraic modeling language. Given these two models, PySP provides two paths for solution of the corresponding stochastic program. The first alternative involves passing an extensive

J.-P. Watson (✉)
Discrete Math and Complex Systems Department, Sandia National Laboratories,
PO Box 5800, MS 1326, Albuquerque, NM 87185-1326, USA
e-mail: jwatson@sandia.gov

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

W. E. Hart
Computer Science and Informatics Department, Sandia National Laboratories,
PO Box 5800, MS 1327, Albuquerque, NM 87185-1327, USA
e-mail: wehart@sandia.gov

form to a standard deterministic solver. For more complex stochastic programs, we provide an implementation of Rockafellar and Wets' Progressive Hedging algorithm. Our particular focus is on the use of Progressive Hedging as an effective heuristic for obtaining approximate solutions to multi-stage stochastic programs. 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. PySP has been used by a number of research groups, including our own, to rapidly prototype and solve difficult stochastic programming problems.

Mathematics Subject Classification (2000) 90C15 · 90C90 · 90C59

1 Introduction

The modeling of uncertainty is widely recognized as an integral component in most real-world decision problems. Typically, uncertainty is associated with the problem input parameters, e.g., consumer demands or construction times. In cases where parameter uncertainty is independent of the decisions, stochastic programming is an appropriate and widely studied mathematical framework to express and solve uncertain decision problems [8,34,55,48]. However, stochastic programming has not yet seen widespread, routine use in industrial applications—despite the significant benefit such techniques can confer over deterministic mathematical programming models. The growing practical importance of stochastic programming is underscored by the recent proposals for and additions of capabilities in many commercial algebraic modeling languages [1,41,52,59].

Over the past decade, two key impediments have been informally recognized as central in inhibiting the widespread industrial use of stochastic programming. First, modeling systems for mathematical programming have only recently begun to incorporate extensions for specifying stochastic programs. Without an integrated and accessible modeling capability, practitioners are forced to implement custom techniques for specifying stochastic programs. Second, stochastic programs are often extremely difficult to solve—especially in contrast to their deterministic counterparts. There exists no analog to CPLEX [12], GUROBI [25], or XpressMP [60] for stochastic programming, principally because the algorithmic technology is still under active investigation and development, particularly in the multi-stage, non-linear, and mixed-integer cases.

Some commercial vendors have recently introduced modeling capabilities for stochastic programming, e.g., LINDO [38], FrontLine [20], XpressMP [60], Maximal [41], and AIMMS [1]. On the open-source front, the options are even more limited. FLOPC++ (part of COIN-OR) [15] provides an algebraic modeling environment in C++ that allows for specification of stochastic linear programs. APLEpy provides similar functionality in a Python programming language environment. In either case, the available modeling extensions have not yet seen widespread adoption.

The landscape for solvers (open-source or otherwise) targeting classes of stochastic program is extremely sparse. Those modeling packages that do provide stochastic programming facilities with few exceptions rely on the translation of the

problem into an *extensive form*—a deterministic mathematical program encoding of a stochastic program in which all scenarios are explicitly and simultaneously represented. The extensive form can then be supplied as input to a solver for the corresponding deterministic class of mathematical program, e.g., linear mixed-integer or non-linear. Unfortunately, direct solution of the extensive form is impractical in all but the simplest cases. Some combination of either the number of scenarios, the number of decision stages, or the presence of discrete decision variables typically leads to extensive forms that are either too difficult to solve or exhaust available system memory. Iterative decomposition strategies such as the L-shaped method [54] or Progressive Hedging [46] (as described in Sect. 6) directly address both of these scalability issues. Other approaches include coordinated branch-and-cut procedures [2]. While effective in certain contexts, the success of these methods is generally sensitive to choices of fundamental algorithm parameters and algorithm configuration. Further, there is little understanding of which algorithms perform best for a particular problem class. In general, the solution of difficult stochastic programs requires both experimentation with and customization of alternative algorithmic paradigms—necessitating the need for generic and configurable solvers.

In this paper, we describe an open-source software package—PySP—that begins to address the issue of the availability of generic and customizable stochastic programming solvers. At the same time, we describe modeling capabilities for expressing stochastic programs. Beyond the obvious need to somehow express a problem instance to a solver, we identify a fundamental characteristic of the modeling language that is in our opinion necessary to achieve the objective of generic and customizable stochastic programming solvers. In particular, the modeling layer must provide mechanisms for accessing components via direct introspection [45], such that solvers can generically and programmatically access and manipulate model components. Further, from a user standpoint, the modeling layer should differentiate between abstract models and model instances—following best practices from the deterministic mathematical modeling community [16].

To express a stochastic program in PySP, the user specifies both the deterministic base model and the scenario tree model with associated uncertain parameters in the Pyomo open-source algebraic modeling language [26,28]. This separation of deterministic and stochastic problem components is similar to the mechanism proposed in SMPS [7,24]. Historically, SMPS has served as the de facto interchange format for specifying stochastic programs—mirroring the early role of the MPS format in deterministic mathematical programming. Pyomo is a Python-based modeling language, and provides the ability to model both abstract problems and concrete problem instances. The embedding of Pyomo in Python enables model introspection and the construction of generic solvers.

Given deterministic and scenario tree models, PySP provides two paths for the solution of the corresponding stochastic program. The first alternative involves passing an extensive form to an appropriate deterministic solver. For more complex stochastic programs, we provide a generic implementation of Rockafellar and Wets' Progressive Hedging (PH) algorithm, with additional specializations for approximating mixed-integer stochastic programs. 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. Karabuk and Grant [37] describe the benefits of Python for such model-building and solving in more detail.

The remainder of this paper is organized as follows. We begin in Sect. 2 with a brief overview of stochastic programming, focusing on multi-stage notation and expression of the scenario tree. In Sect. 3, we describe the PySP approach to modeling a stochastic program, illustrated using a well-known introductory model. PySP capabilities for writing and solving an extensive form are described in Sect. 4. In Sect. 5, we compare and contrast PySP with other open-source software packages for modeling and solving stochastic programs. Our generic implementation of Progressive Hedging is described in Sect. 6, while Sect. 7 details mechanisms for customizing the behavior of PH. By basing PySP on Coopr and Python, we are able to provide straightforward mechanisms to support distributed parallel solves in the context of PH and other decomposition algorithms; these facilities are detailed in Sect. 8. Finally, we conclude in Sect. 9, with a brief discussion of the use of PySP on a number of active research projects.

2 Stochastic programming: definition and notations

We concern ourselves with stochastic optimization problems where uncertain parameters (data) can be represented by a set of scenarios, each of which specifies both (1) a full set of random variable (parameter) realizations and (2) a corresponding probability of occurrence. The random variables in question specify the evolution of uncertain parameters over stages, where the stages usually correspond to time. We index the scenario set by s and refer to the entire index set as $\mathcal{S}$. To improve locution, we often refer to "scenario s" with the understanding that we mean the scenario indexed by s. The probability of occurrence of s (or, more accurately, a realization "near" the scenario indexed by s) is written as $\Pr(s)$. The source of these scenarios does not concern us in this paper, although we observe that they are frequently obtained via simulation or formed from expert opinions. We assume that the decision process of interest consists of a sequence of discrete stages, the set of which is denoted $\mathcal{T}$; we index $\mathcal{T}$ by t.

Although PySP supports specification of non-linear constraints and objectives, we develop notation for the linear case in the interest of simplicity. For each scenario s and stage t, $t \in \mathcal{T} = \{1, \ldots, |\mathcal{T}|\}$, we are given a row vector $c(s, t)$ of length $n(t)$, a $m(t) \times n(t)$ matrix $A(s, t)$, and a column vector $b(s, t)$ of length $m(t)$. Let $N(t)$ be the index set $\{1, \ldots, n(t)\}$ and $M(t)$ be the index set $\{1, \ldots, m(t)\}$. For notational convenience, let $A(s)$ denote $(A(s, 1), \ldots, A(s, |\mathcal{T}|))$ and let $b(s)$ denote $(b(s, 1), \ldots, b(s, |\mathcal{T}|))^\top$.

The decision variables in a stochastic program consist of a set of $n(t)$ vectors $x(t)$; one vector for each scenario $s \in \mathcal{S}$. Let $X(s)$ be $(x(s, 1), \ldots, x(s, |\mathcal{T}|))^\top$. We will use X as shorthand for the entire solution system of x vectors, i.e., $X = (x(1, 1), \ldots, x(|\mathcal{S}|, |\mathcal{T}|))$.

If we were prescient enough to know which scenario $s \in \mathcal{S}$ would be ultimately realized, our optimization objective would be to minimize

$$f_s(X(s)) \equiv \sum_{t \in T} \sum_{i \in N(t)} [c_i(s, t) x_i(s, t)] \qquad (\mathrm{P}_s)$$

subject to the constraint

$$X \in \Omega_s.$$

We use Ω_s as an abstract notation to express all constraints for scenario s, including requirements that some decision vector elements are discrete, or more general requirements such as

$$A(s)X(s) \geq b(s).$$

The notation $A(s)X(s)$ is used to capture the usual sorts of single period and inter-period linking constraints that one typically finds in multi-stage mathematical programming formulations, e.g., see Chen et al. [10, pp. 32–35].

We refer to a solution that satisfies constraints for all scenarios as *admissible*. We refer to a solution vector as *implementable* if for all pairs of scenarios s and s' that are indistinguishable up to stage t, $x_i(s, t') = x_i(s', t')$ for all $1 \leq t' \leq t$ and each i in each $N(t)$. We refer to the set of all implementable solutions as $\mathcal{N}_\mathcal{S}$ for a given set of scenarios, indexed by $s \in \mathcal{S}$.

We must obtain solutions that do not require foreknowledge and that will be feasible independent of which scenario is ultimately realized. In particular, lacking prescience, only solutions that are implementable are practically useful. Solutions that are not admissible, on the other hand, may have some value because while some constraints may represent laws of physics, others may be violated slightly without serious consequence.

To achieve admissible and implementable solutions, the expected value minimization problem then becomes:

$$\min \sum_{s \in \mathcal{S}} [\Pr(s) f_s(X(s))] \quad \text{(P)}$$

subject to

$$\begin{aligned} X(s) &\in \Omega_s && \forall s \in \mathcal{S} \\ X(s) &\in \mathcal{N}_\mathcal{S} && \forall s \in \mathcal{S}. \end{aligned}$$

Formulation (P) is known as a stochastic mathematical program. If all decision variables are continuous, we refer to the problem simply as a continuous stochastic program. If at least some of the decision variables are discrete, we refer to the problem as a mixed-integer stochastic program. Both continuous and mixed-integer stochastic programs can be linear or non-linear, depending on the nature of the problem constraints and objectives. More comprehensive introductions to both the theoretical foundations and the range of potential applications of stochastic programming can be found in Birge and Louveaux [8], Shapiro et al. [48], and Wallace and Ziemba [55].

In practice, the parameter uncertainty in stochastic programs is often encoded via a *scenario tree*, in which a node specifies the parameter values $b(s, t)$, $c(s, t)$, and

$A(s,t)$ for all scenarios indexed by $s = s'$ and $s = s''$ in $\mathcal{S}$ such that s' and s'' are indistinguishable up to stage t. Scenario trees are discussed in more detail in Sect. 6.

3 Modeling in PySP

Modern algebraic modeling languages (AMLs) such as AMPL and GAMS provide powerful mechanisms for expressing and solving deterministic mathematical programs [3,21]. AMLs allow non-specialists to easily formulate and solve mathematical programming models, avoiding the need for a deep understanding of the underlying algorithmic technologies. We desire to deploy the same capabilities for stochastic mathematical programming models. Many AMLs differentiate between the abstract, symbolic model and a concrete instance of the model, i.e., a model instantiated with data. The advantages of this approach are widely known [16, p. 35], in particular (1) the improved maintainability and readability of the core algebraic model, (2) the ability to specify constraint and objective data via symbolic indexing (as opposed to concrete values), and (3) the separation (and therefore substitutability) of data sources from specification of the model. Thus, one of our core design objectives is to retain this differentiation in our approach to modeling stochastic mathematical programs in PySP. Finally, our ultimate objective is to develop and maintain software under an open-source distribution license. Consequently, PySP itself must be based on an open-source AML.

These design requirements have led us to select Pyomo [28] as the AML upon which PySP is based. Pyomo is an open-source AML developed and maintained by Sandia National Laboratories (co-developed by authors of this paper), and is distributed as part of IBM's COIN-OR initiative [11]. Pyomo is written in the Python high-level programming language [44], which possesses several features that enable the development of generic solvers. AML alternatives to Pyomo (many of which are also written in Python) do exist, but the discussion of the pros and cons of particular packages are beyond the present scope; we defer to Hart et al [28] for such arguments. One key differentiating feature of Pyomo with respect to open-source AMLs is that it supports the specification of linear *and* non-linear mathematical programs, with combinations of continuous and discrete variables. Consequently, PySP fully supports the modeling of linear and nonlinear stochastic programs, with or without discrete variables. Methods for solution of the resulting stochastic programs depends on capabilities of deterministic mathematical programming solvers available to a user, as we discuss in Sects. 4, 6, and 7.

In this section, we discuss the use of Pyomo to formulate and express stochastic programs in PySP. As a motivating example, we consider the well-known "farmer" stochastic program [8]. Mirroring several other approaches to modeling stochastic programs (e.g., see [51]), we require the specification of two related components: the deterministic base model and the scenario tree model. In Sect. 3.1 we discuss the specification of the deterministic base model and associated data; Sect. 3.2 then details the specification of scenario tree models in PySP. The mechanisms for specifying scenario parameter data are discussed in Sect. 3.3. We briefly discuss the programmatic compilation of a scenario tree specification into objects appropriate for use by

solvers in Sect. 3.4. Finally, we discuss the availability of additional PySP examples in Sect. 3.5.

3.1 The deterministic reference model

The starting point for developing a stochastic programming model in PySP is the specification of an abstract *reference* model, which describes the deterministic multi-stage problem for an representative scenario. The reference model does not make use of, or describe, any information relating to parameter uncertainty or the scenario tree. Typically, it is simply the model that would be used in single-scenario analysis, i.e., the model that is commonly developed before stochastic aspects of an optimization problem are considered. PySP requires that the reference model—specified in Pyomo—is contained in a file named **ReferenceModel.py**. As an illustrative example, the complete reference model for Birge and Louveaux's farmer problem is shown Fig. 1. When looking at this figure, it is useful to note that the backslash is the Python line continuation character.

For details concerning the syntax and use of the Pyomo modeling language, we defer to Hart et al. [26,28]. Here, we simply observe that a detailed knowledge of Python is *not* necessary to develop a reference model in Pyomo; users are often unaware that a Pyomo model specifies executable Python code, or that they are using a class library. Relative to the AMPL formulation of the farmer problem, the Pyomo formulation is somewhat more verbose—primarily due to its embedding in an high-level programming language.

While the reference model is independent of any stochastic components of the problem, PySP does require that the objective function cost component for each decision stage of the stochastic program be assigned to a distinct variable or variable index. In the farmer reference model, we simply label the first and second stage cost variables as **FirstStageCost** and **SecondStageCost**, respectively. The corresponding values are computed via the constraints **ComputeFirstStageCost** and **ComputeSecondStageCost**. We initially imposed the requirement concerning specification of per-stage cost variables (which is not a common feature in other stochastic programming software packages) primarily to facilitate various aspects of solution reporting. However, the convention has additionally proved very useful in implementation of various PySP solvers.

To create a concrete instance from the abstract reference model, a Pyomo data file must also be specified. The data can correspond to an arbitrary scenario, and must completely specify all parameters in the abstract reference model. The reference data file must be named **ReferenceModel.dat**. An example data file corresponding to the farmer reference model is shown in Fig. 2. Although Pyomo supports various data file formats (including raw ASCII tables, spreadsheets, and databases), the example illustrates a file that uses Pyomo data commands—the most commonly used data file format in Pyomo. Pyomo data commands include commands for specifying set and parameter data that are consistent with AMPL's data commands. Our adoption of this convention minimizes the effort required to translate deterministic reference models expressed in AMPL into Pyomo.

```
from coopr.pyomo import *
model=Model()

# Parameters
model.CROPS=Set()
model.TOTAL_ACREAGE=Param(within=PositiveReals)
model.PriceQuota=Param(model.CROPS, within=PositiveReals)
model.SubQuotaSellingPrice=Param(model.CROPS, within=PositiveReals)
model.SuperQuotaSellingPrice=Param(model.CROPS)
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))
model.FirstStageCost=Var()
model.SecondStageCost=Var()

# Constraints
def total_acreage_rule(model):
    return summation(model.DevotedAcreage) <= model.TOTAL_ACREAGE
model.ConstrainTotalAcreage=Constraint(rule=total_acreage_rule)

def cattle_feed_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=cattle_feed_rule)

def limit_amount_sold_rule(model, i):
    return model.QuantitySubQuotaSold[i] + \
           model.QuantitySuperQuotaSold[i] <= \
           (model.Yield[i] * model.DevotedAcreage[i])
model.LimitAmountSold=Constraint(model.CROPS, \
                                 rule=limit_amount_sold_rule)

def enforce_quotas_rule(model, i):
    return (0.0, model.QuantitySubQuotaSold[i], model.PriceQuota[i])
model.EnforceQuotas=Constraint(model.CROPS, \
                               rule=enforce_quotas_rule)

# Stage-specific cost computations
def first_stage_cost_rule(model):
   return model.FirstStageCost == \
      summation(model.PlantingCostPerAcre, model.DevotedAcreage)
model.ComputeFirstStageCost=Constraint(rule=first_stage_cost_rule)

def second_stage_cost_rule(model):
   expr=summation(model.PurchasePrice, model.QuantityPurchased)
   expr -= summation(model.SubQuotaSellingPrice, \
                     model.QuantitySubQuotaSold)
   expr -= summation(model.SuperQuotaSellingPrice, \
                     model.QuantitySuperQuotaSold)
   return (model.SecondStageCost - expr) == 0.0
model.ComputeSecondStageCost=Constraint(rule=second_stage_cost_rule)

# Objective
def total_cost_rule(model):
    return (model.FirstStageCost + model.SecondStageCost)
model.Total_Cost_Objective=Objective(rule=total_cost_rule, \
                                     sense=minimize)
```

Fig. 1 ReferenceModel.py. The deterministic reference Pyomo model for Birge and Louveaux's farmer problem

```
set CROPS := WHEAT CORN SUGAR_BEETS ;

param TOTAL_ACREAGE := 500 ;

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 ;

param PurchasePrice := WHEAT 238 CORN 210 SUGAR_BEETS 100000 ;

param PlantingCostPerAcre := WHEAT 150 CORN 230 SUGAR_BEETS 260 ;

param Yield := WHEAT 3.0 CORN 3.6 SUGAR_BEETS 24 ;
```

Fig. 2 ReferenceModel.dat. The Pyomo reference model data file for Birge and Louveaux's farmer problem

Finally, we observe that general deterministic mathematical programs can be specified using Pyomo, including those containing non-linear and mixed-integer (or both) components. Leveraging this capability, users are able to specify general stochastic programs in PySP. However, as we discuss subsequently, the ability of PySP to solve a particular stochastic program depends strongly on the availability of base solvers for the corresponding class of deterministic mathematical program.

3.2 The scenario tree

Given a deterministic reference model, the second step in developing a stochastic program in PySP involves specification of the scenario tree structure and associated parameter data. A PySP scenario tree specification supplies all information concerning the stages, the mapping of decision variables to stages, how various scenarios are temporally related to one another (i.e., scenario tree nodes and their inter-relationships), and the probabilities of various scenarios. As discussed below, the scenario tree does not directly specify uncertain parameter values; rather, it specifies references to input files containing such data.

As with the abstract reference model, the abstract scenario tree model in PySP is expressed in Pyomo. However, the contents of the scenario tree model—called **ScenarioStructure.py** and shown in Fig. 3 for reference—are fixed. The model is built into and distributed with PySP; the user does not edit this file. Instead, the user must supply values of each of the parameters specified in the scenario tree model, using the same Pyomo mechanisms used to specify data in the case of (for example) the deterministic reference model. Finally, we observe that the scenario tree model is simply a collection of data that is specified using Pyomo parameter and set objects, i.e., a very restricted form of a Pyomo model (lacking variables, constraints, and an objective).

The precise semantics for each of the parameters (or sets) indicated in Fig. 3 are as follows:

```
from coopr.pyomo import *

scenario_tree_model=Model()

scenario_tree_model.Stages=Set(ordered=True)
scenario_tree_model.Nodes=Set()

scenario_tree_model.NodeStage=Param(scenario_tree_model.Nodes, \
                                    within=scenario_tree_model.Stages)
scenario_tree_model.Children=Set(scenario_tree_model.Nodes, \
                                 within=scenario_tree_model.Nodes, \
                                 ordered=True)
scenario_tree_model.ConditionalProbability= \
                               Param(scenario_tree_model.Nodes)

scenario_tree_model.Scenarios=Set(ordered=True)
scenario_tree_model.ScenarioLeafNode= \
             Param(scenario_tree_model.Scenarios, \
                   within=scenario_tree_model.Nodes)

scenario_tree_model.StageVariables=Set(scenario_tree_model.Stages)
scenario_tree_model.StageCostVariable= \
             Param(scenario_tree_model.Stages)

scenario_tree_model.ScenarioBasedData=Param(within=Boolean, \
                                            default=True)
```

Fig. 3 ScenarioStructure.py. The PySP model for specifying the structure of the scenario tree

- *Stages*. An ordered set containing the names (specified as arbitrary strings) of the stages. The order corresponds to the order of the stages, which is typically determined by the time corresponding to the stage.
- *Nodes*. A set of the names (specified as arbitrary strings) of the nodes in the scenario tree.
- *NodeStage*. An indexed parameter mapping node names to stage names. Each node in the scenario tree must be assigned to a unique stage.
- *Children*. An indexed set mapping node names to sets of node names. For each non-leaf node in the scenario tree, a set of child nodes must be specified. This set implicitly defines the overall branching structure of the scenario tree. Using this set, the parent nodes are computed internally to PySP. There can only be one node in the scenario tree with no parents, i.e., the tree must be singly rooted.
- *ConditionalProbability*. An indexed, real-valued parameter mapping node names to their conditional probability, relative to their parent node. The conditional probability of the root node must be equal to 1, and for any node with children, the conditional probabilities of those children must sum to a value x such that $|1 - x| <= 1e - 6$ (allowing for minor slack due to numerical tolerance issues). Numeric values for node conditional probabilities must be contained within the interval [0, 1].
- *Scenarios*. An ordered set containing the names (specified as arbitrary strings) of the scenarios. These names are used for two purposes: reporting and data specification (see Sect. 3.3). The ordering is provided as a convenience for the user, to organize reporting output. The scenario names are distinct from the names of the scenario leaf nodes, introduced below. The separation (in contrast to simply labeling the

scenarios by their leaf node name) derives from the naming conventions associated with loading scenario and node-specific data, as we discuss in Sect. 3.3.

- *ScenarioLeafNode*. An indexed parameter mapping scenario names to their leaf node name. Facilitates linkage of a scenario to its list of defining nodes in the scenario tree.
- *StageVariables*. An indexed set mapping stage names to sets of variable names in the reference model. The sets of variables names indicate those variables that are associated with a given stage. Implicitly defines the non-anticipativity constraints that should be imposed when generating and/or solving the PySP model.
- *ScenarioBasedData*. A Boolean parameter specifying how the instances for each scenario are to be constructed. The semantics for this parameter are detailed below in Sect. 3.3.

Data to instantiate these parameters and sets must be provided by a user in a file named **ScenarioStructure.dat**. The scenario tree data for the farmer problem is shown in Fig. 4, specified using the AMPL data file format.

When loading the **ScenarioStructure.dat** file, PySP performs extensive consistency checks on the input data. In particular, checks are executed to ensure that the specified structure is actually a tree (with no loops, forming a single component), and that parent–child node pairs reside in adjacent decision stages. PySP assumes that the scenario tree is of uniform depth. Presently, PySP performs limited checking regarding variable to stage assignments, i.e., it is possible to leave variables unassigned to a stage. Variables cannot be assigned to multiple stages.

We observe that PySP provides a simple "slicing" syntax to specify subsets of indexed variables. In particular, the "*" character is used to match all values in a particular dimension of an indexed parameter. In more complex PySP examples, variables are typically indexed by stage. In these cases, the slice syntax allows for very concise specification of the stage-to-variable mapping. While the slicing syntax is incompatible with the AMPL.dat file syntax, this is in practice not a concern because **ScenarioStructure.dat** files are intended strictly for use by PySP.

Finally, we observe that PySP makes no assumptions regarding the linkage between stages and variable index structure. In particular, the stage need not explicitly be referenced within a variable's index set. While this is often the case in multi-stage formulations, the convention is not universal, e.g., as in the case of the farmer problem.

3.3 Scenario parameter specification

Data files specifying the deterministic and stochastic parameters for each of the scenarios in a PySP model can be specified in one of two ways. The simplest approach is "scenario-based", in which a single data file containing a complete parameter specification is provided for each scenario. In this case, the file naming convention is as follows: If the scenario is named **ScenarioX**, then the corresponding data file for the scenario must be named **ScenarioX.dat**. This approach is often expedient—especially if the scenario data are generated via simulation, as is often the case in practice. However, there is necessarily redundancy in the encoding. Depending on the problem size and number of scenarios, this redundancy may become excessive in terms of disk

```
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       0.33333333
                                AboveAverageNode  0.33333333 ;

set Scenarios := BelowAverageScenario
                 AverageScenario
                 AboveAverageScenario ;

param ScenarioLeafNode := BelowAverageScenario BelowAverageNode
                          AverageScenario      AverageNode
                          AboveAverageScenario AboveAverageNode ;

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

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

Fig. 4 ScenarioStructure.dat. The PySP data file for specifying the scenario tree for Birge and Louveaux's farmer problem

storage and access time. Scenario-based data specification is the default behavior in PySP, as indicated by the default value of the **ScenarioBasedData** parameter in Fig. 3. We note that the listing in Fig. 2 is an example of a scenario-based data specification.

Node-based parameter specification is provided as an alternative to the default scenario-based approach, principally to eliminate storage redundancy. With node-based specification, parameter data specific to each *node* in the scenario tree is specified in a distinct data file. The naming convention is as follows: If the node is named **NodeX**, then the corresponding data file for the node must be named **NodeX.dat**. To create a scenario instance, data for all nodes associated with a scenario are accessed (via the ScenarioLeafNode parameter in the scenario tree specification and the computed parent node linkages). Node-based parameter encoding eliminates redundancy, although typically at the expense of a slightly more complex instance generation process. To enable node-based scenario initialization, a user needs to simply add the following line to **ScenarioStructure.dat**:

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

In the case of the farmer problem, all parameters except for **Yield** are identical across scenarios. Consequently, these parameters can be placed in a file named **RootNode.dat**. Then, files containing scenario-specific **Yield** parameter values are specified for each second-stage leaf node (in files named **AboveAverageNode.dat**, **AverageNode.dat**, and **BelowAverageNode.dat**).

3.4 Compilation of the scenario tree model

The PySP scenario tree model is a declarative entity, merely specifying the data associated with a scenario tree. PySP internally uses the information contained in this model to construct a **ScenarioTree** object, which in turn is composed of **ScenarioTreeNode**, **Stage**, and **Scenario** objects. In aggregate, these Python objects allow programmatic navigation, query, manipulation, and reporting of the scenario tree structure. While hidden from the typical user, these objects are crucial in the processes of generating the extensive form (Sect. 4) and generic solvers (Sect. 6).

3.5 Additional PySP examples

Several PySP examples are installed automatically with Coopr, in the **examples/pysp** sub-directory under the root install directory. In each of these directories, there is a **README.txt** file discussing various command-line script options to exercise the solver capabilities described subsequently. Numerous other PySP examples are available via download from the following web page: http://www.software.sandia.gov/trac/coopr/wiki/PySP.

4 Generating and solving an extensive form

Given a stochastic program encoded in accordance with the PySP conventions described in Sect. 3, the next immediate issue of concern is its solution. The most straightforward method to solve a stochastic program involves generating an *extensive form* (also known as a *deterministic equivalent*) and then invoking an appropriate (for the particular problem class) standard deterministic programming solver, e.g., CPLEX in the case of stochastic mixed-integer programs, or Ipopt in the case of stochastic non-linear programs. An extensive form given as problem (P) in Sect. 2 completely specifies all scenarios and the coupling non-anticipativity constraints at each node in the scenario tree. For a variety of reasons (primarily related to our scenario-oriented specification of stochastic programming models, and our subsequent emphasis on scenario-based decomposition algorithms), we internally employ an *explicit* representation of the extensive form, in which non-anticipativity constraints are constructed to enforce variable equality across scenarios at each node in the tree. More specifically, for each non-anticipative variable we create a "master" variable, and post constraints requiring equality between the value of this master variable and the value of the corresponding variable in each participating scenario; for additional detail, we defer

to Sect. 4.2. An alternative approach is to implicitly represent the non-anticipativity constraints by introducing variables that are referenced by multiple scenarios.

In many cases, particularly when small numbers of scenarios are involved or the decision variables are all continuous, the extensive form can be effectively solved with off-the-shelf solvers [42]. Further, although decomposition techniques may ultimately be needed for large, more complex models, the extensive form is usually the first attempted method to solve a stochastic program.

In this section, we describe the use and design of facilities in PySP for generating and solving the extensive form. Section 4.1 describes a user script for generating and solving the extensive form; an overview of the implementation of this script is then provided in Sect. 4.2.

4.1 The **runef** script

PySP provides an easy-to-use script—**runef**—to both generate and solve the extensive form of a stochastic program. We now briefly describe the primary command-line options for this script; note that all options begin with a double dash prefix:

`--help`
Display all command-line options, with brief descriptions, and exit.

`--verbose`
Display verbose output to the standard output stream, above and beyond the usual status output. Disabled by default.

`--model-directory=MODEL_DIRECTORY`
Specifies the directory in which the reference model (**ReferenceModel.py**) is stored. Defaults to ".", the current working directory.

`--instance-directory=INSTANCE_DIRECTORY`
Specifies the directory in which all reference model and scenario model data files are stored. Defaults to ".", the current working directory.

`--output-file=OUTPUT_FILE`
Specifies the name of the output file to which the extensive form is written. Defaults to "efout.lp".

`--solve`
Directs the script to solve the extensive form after writing it. Disabled by default.

`--solver=SOLVER_TYPE`
Specifies the type of solver for solving the extensive form, if a solve is requested. Defaults to "cplex".

`--solver-options=SOLVER_OPTIONS`
Specifies solver options in keyword-value pair format, if a solve is requested.

`--output-solver-log`
Specifies that the output of the solver is to be echoed to the standard output stream. Disabled by default. Useful to ascertain status for extensive forms with long solve times.

For example, to write and solve the farmer problem (provided with the Coopr installation, in the directory **coopr/examples/pysp/farmer**) using the GLPK solver, the user simply executes:

```
runef --model-directory=models \
      --instance-directory=scenariodata \
      --solve \
      --solver=glpk
```

The forward-slash characters in the above listing are simply continuation characters in Unix, used (here and elsewhere in this article) to restrict the width of the example inputs and outputs. A single back-slash is similarly used as a continuation character in Python, and in our example code fragments.

Following solver execution, the resulting solution is loaded and displayed. The solution output is split into two distinct components: variable values and stage/scenario costs. For the farmer example, the per-node variable values are given as:

```
Tree Nodes:

        Name=AboveAverageNode
        Stage=SecondStage
        Parent=RootNode
        Variables:
                QuantitySubQuotaSold[CORN]=48.0
                QuantitySubQuotaSold[SUGAR_BEETS]=6000.0
                QuantitySubQuotaSold[WHEAT]=310.0

        Name=AverageNode
        Stage=SecondStage
        Parent=RootNode
        Variables:
                QuantitySubQuotaSold[SUGAR_BEETS]=5000.0
                QuantitySubQuotaSold[WHEAT]=225.0

        Name=BelowAverageNode
        Stage=SecondStage
        Parent=RootNode
        Variables:
                QuantitySubQuotaSold[SUGAR_BEETS]=4000.0
                QuantitySubQuotaSold[WHEAT]=140.0
                QuantityPurchased[CORN]=48.0

        Name=RootNode
        Stage=FirstStage
        Parent=None
        Variables:
                DevotedAcreage[CORN]=80.0
                DevotedAcreage[SUGAR_BEETS]=250.0
                DevotedAcreage[WHEAT]=170.0
```

Similarly, the per-node stage cost and per-scenario overall costs are given as follows:

```
Tree Nodes:

        Name=AboveAverageNode
        Stage=SecondStage
        Expected node cost=-275900.0000

        Name=AverageNode
        Stage=SecondStage
        Expected node cost=-218250.0000

        Name=BelowAverageNode
        Stage=SecondStage
        Expected node cost=-157720.0000

        Name=RootNode
        Stage=FirstStage
        Expected node cost=-108390.0000
```

```
Scenarios:

        Name=AboveAverageScenario
        Stage=FirstStage        Cost=108900.0000
        Stage=SecondStage       Cost=-275900.0000
        Total scenario cost=-167000.0000

        Name=AverageScenario
        Stage=FirstStage        Cost=108900.0000
        Stage=SecondStage       Cost=-218250.0000
        Total scenario cost=-109350.0000

        Name=BelowAverageScenario
        Stage=FirstStage        Cost=108900.0000
        Stage=SecondStage       Cost=-157720.0000
        Total scenario cost=-48820.0000
```

All of the above information, in addition to various run-time statistics, are reported following solves during execution of the **runef** script.

PySP currently supports output of the extensive form in two solver input file formats: the AMPL solver library "NL" format and the CPLEX "LP" file format. While originally defined for CPLEX, the latter is read (subject to some vendor-noted exceptions) by a variety of commercial and open-source linear and mixed-integer solvers. The default is the CPLEX LP format; the NL format is indicated by supplying (through the --**output-file** option) a file with a **.nl** suffix. In practice, nearly all commercial and open-source solvers support one of these two input file formats. Specifically, the LP file format supports specification of linear and mixed-integer linear problems, with some extensions to account for specific classes of quadratic objective and constraint. In contrast, the NL file format fully supports linear, non-linear, and mixed-integer problems. The LP file format is most commonly used with CPLEX, Gurobi, and GLPK, while

the NL file format is generally used in conjunction with non-linear solvers providing an AMPL Solver Library (ASL)-based interface. Coopr provides a generic interface to ASL-based solvers (e.g., Ipopt and Couenne), in addition to CPLEX, Gurobi, and GLPK.

Various other command-line options are available in the **runef** script, including those related to performance profiling and Python garbage collection. Further, the **runef** script is capable of writing and solving the extensive form augmented with a weighted Conditional Value-at-Risk term in the objective [47].

We conclude by noting that the **runef** script, as with the decomposition-based solver script described in Sect. 6, relies on significant functionality from the Coopr Python optimization library in which PySP is embedded. This includes a wide range of solver interfaces (both commercial and open-source), problem writers, solution readers, and distributed solvers. For further details, we refer to Hart et al. [26,28].

4.2 Under the hood: generating the extensive form

We now provide an overview of the implementation of the **runef** script. In doing so, our objectives are to (1) illustrate the use of Python to create generic writers and solvers and (2) to provide some indication of the programmatic-level functionality available in PySP.

The high-level process executed by the **runef** script to generate the extensive form in PySP is as follows:

1. Load the scenario tree data; create the corresponding instance.
2. Create the ScenarioTree object from the scenario tree Pyomo model.
3. Load the reference model; create the corresponding instance from reference data.
4. Load the scenario instance data; create the corresponding instances.
5. Create the master "binding" instance; instantiate per-node variable objects.
6. Add master-to-scenario instance equality constraints to enforce non-anticipativity.

Steps 1 and 2 simply involve the process of creating the Pyomo instance specifying all data related to the scenario tree structure, and creating the corresponding ScenarioTree object to facilitate programmatic access of scenario tree attributes. In Step 3, the core deterministic abstract model is loaded. The abstract model is then used in Step 4, in conjunction with the ScenarioTree object, to create a concrete model instance for each scenario in the stochastic program. The scenario instances are at this point—and remain so—completely independent of one another. This approach differs from that of some of the software packages described in Sect. 5, in which variables are instantiated for each node of the scenario tree and shared across the relevant scenarios. While our approach does introduce redundancy, the replication introduces only moderate memory overhead and confers significant practical advantages when implementing generic decomposition-based solvers, e.g., as illustrated below in Sect. 6. In particular, we note that scenario-based decomposition solvers gradually and incrementally enforce non-anticipativity, such that replicated variables are required.

Next, a master "binding" instance is created in Steps 5 and 6. The purpose of the master binding instance is to enforce the required non-anticipativity constraints at each

node in the scenario tree. Using the ScenarioTree object, the tree is traversed and the collection of variable names (including indices, if necessary) associated with each node is identified—initially specified via the *StageVariables* attribute of the scenario tree model. The corresponding variable objects are then identified in the reference model instance, cloned, and attached to the binding instance. This step critically requires the Python capability of *introspection*: the ability, at run-time, to gather information about objects and manipulate them.

To illustrate how introspection is used to develop generic algorithms, consider again the farmer example from Sect. 3. By specifying the line

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

the user is communicating the requirement to impose non-anticipativity constraints on (all indices of) the first stage variable *DevotedAcreage*. The variable is specified simply as a string, which can be programmatically split into the corresponding root name and index template. Using the root name, Python can query (via the **getattr** built-in function) the reference model for the corresponding variable, validate that it exists, and if so, return the corresponding variable object. This loose coupling between the user data and algorithm code is facilitated by this simple, yet powerful, introspection mechanism.

The final primary step in the **runef** script involves construction of constraints to enforce non-anticipativity between the newly created variables in the master binding instance and the variables in the relevant scenario instances. This process again relies on introspection to achieve a generic implementation.

Overall, the core functionality of the **runef** script is expressed in approximately 700 lines of Python code—including all white-space and comments. This includes both the code for creating the relevant Pyomo instances, generating the master binding instance via the processed described above, and controlling the output of the LP file.

Finally, we observe that despite our explicit approach to writing the extensive form through the introduction of master variables and non-anticipativity constraints, we find that the impact on run-time is typically negligible. The presolvers in commercial packages such as CPLEX, Gurobi, or XpressMP (and those available with some open-source solvers) are able to quickly identify and eliminate most of the redundant variables and constraints.

5 Related proposals and software packages

We now briefly survey prior and on-going efforts to develop software packages supporting the specification and solution of stochastic programs, with the objective of placing the capabilities of PySP in this broader context. Numerous extensions to existing AMLs to support the specification of stochastic programs have been proposed in the literature; Gassmann and Ireland [23] is an early example. Similarly, various solver interfaces have been proposed, with the dominant mechanism being the direct solution of the extensive form. Here, we primarily focus on specification and solver efforts associated with open-source and academic initiatives, which generally share the same

distribution goals, user community targets, and design objectives (e.g., experimental, generic, and configurable solvers) as PySP.

StAMPL Fourer and Lopes [18] describe an extension to AMPL, called StAMPL, whose goal is to simplify the modeling process associated with stochastic program specification. One key objective of StAMPL is to explicitly avoid the use of scenario and stage indices when specifying the core algebraic model, separating the specification of the stochastic process from the underlying deterministic optimization model. The authors describe a preprocessor that translates a StAMPL problem description into the fully indexed AMPL model, which in turn is written in SMPS format for solution. PySP differs from StAMPL in that it provides a straightforward mechanism to specify a stochastic program, and does not strive to advance the state-of-the-art in modeling. Rather, our primary focus is on developing generic and configurable solvers, and discussed in Sects. 4, 6, and 7.

STRUMS STRUMS is a system for performing and managing decomposition and relaxation strategies in stochastic programming [17]. Input problems are specified in the SMPS format, and the package provides mechanisms for writing the extensive form, performing basic and nested Benders decomposition (i.e., the L-shaped method), and implementing Lagrangian relaxation; only stochastic linear programs are considered. The design objective of STRUMS—to provide mechanisms facilitating automatic problem decomposition—is consistent with the design of PySP. However, PySP currently provides mechanisms for scenario-based decomposition, in contrast to stage-oriented decomposition. In contrast to STRUMS, PySP is integrated with an AML.

DET2STO Thénié et al. [51] describe an extension of AMPL to support the specification of stochastic programs, noting that (at the time the effort was initiated) no AMLs were available with stochastic programming support. In particular, they provide a script—called DET2STO, available from http://www.apps.ordecsys.com/det2sto—taking an augmented AMPL model as input and generating the extensive form via an SMPS output file. The research focus is on the automated generation of the extensive form, with the authors noting: "We recall here that, while it is relatively easy to describe the two base components—the underlying deterministic model and the stochastic process—it is tedious to define the contingent variables and constraints and build the deterministic equivalent" [51, p.35]. While subtle modeling differences do exist between DET2STO and PySP (e.g., in the way scenario-based and transition-based representations are processed), they provide identical functionality in terms of ability to model stochastic programs and generate the extensive form.

SMI Part of COIN-OR, the stochastic modeling interface (SMI) [49] provides a set of C++ classes to (1) either to programmatically create a stochastic program or load a stochastic program specified in SMPS, and (2) to write the extensive form of the resulting program. SMI provides no solvers, instead focusing on generation of the extensive form for solution by external solvers. Connections to FLOPC++ [15] do exist, providing a mechanism for problem description via an AML. While providing

a subset of PySP functionality, the need to express models in a compiled, technically sophisticated programming language (C++) is a significant drawback for many users.

APLEpy Karabuk [36] describes the design of classes and methods to implement stochastic programming extensions to his Python-based APLEpy [35] environment for mathematical programming, with a specific emphasis on stochastic linear programs. Karabuk's primary focus is on supporting relaxation-based decompositions in general, and the L-shaped method in particular, although his design would create elements that could be used to construct other algorithms as well. The vision expressed in Karabuk [36] is one where the boundary between model and algorithm must be crossed so that the algorithm can be expressed in terms of model elements. This approach is also possible using Pyomo and PySP, but it is not the underlying philosophy of PySP. Rather, we are interested in enabling the separation of model, data, and algorithm except when the users wish to create model specific algorithm enhancements.

SPInE SPInE [53] provides an integrated modeling and solver environment for stochastic programming. Stochastic models are specified either in an extension to AMPL called SAMPL, or an extension of MPL called SMPL. Both language extensions provide similar functionality, and are integrated with a number of built-in solvers. PySP, SAMPL, and SMPL provide similar mechanisms for modeling stochastic programs. In contrast to PySP, the solvers are not specifically designed to be customizable, and are generally limited to specific problem classes. For example, multi-stage stochastic linear programs are solved via nested Benders decomposition, while Lagrangian relaxation is the only option for two-stage mixed-integer stochastic programs. SPInE is primarily focused on providing an out-of-the-box solution for stochastic linear programs, which is consistent with the lack of emphasis on customizable solution strategies.

SLP-IOR Similar to SPInE, SLP-IOR [33] is an integrated modeling and solver environment for stochastic programming, with a strong emphasis on the linear case. In contrast to SPInE, SLP-IOR is based on the GAMS AML, and provides a broader range of solvers. However, as with SPInE, the focus is not on easily customizable solvers (most of the solver codes are written in FORTRAN). Further, solvers for the integer case are largely ignored.

SUTIL SUTIL [50] is a C++ library for loading stochastic programs defined in the SMPS format, and subsequently allowing for programmatic manipulation. Examples of manipulations include generation of the deterministic equivalent, extracting individual scenarios, and creating a scenario tree via Monte Carlo sampling. The focus of SUTIL is similar to that of SMI. In contrast to SUTIL, PySP provides for expression of the base deterministic model and associated scenario data in a full-fledged AML (as opposed to SMPS), and assumes that the user has performed scenario sampling exogenously.

OSiL/SE OSiL [19] is an effort to develop a modern inter-change language for mathematical programs, to replace the antiquated MPS format. OSiL leverages a modern, extensible file format, specifically XML. OSiL/SE [19] is an extension of OSiL for

supporting the specification of stochastic programs, with the objective of modernizing and eventually replacing SMPS. The objectives of OSiL/SE and PySP differ with respect to modeling. Specifically, OSiL/SE is an interchange format for specifying concrete problem instances, in contrast to the symbolic models specified via modern AMLs (including Pyomo). OSiL/SE is intended to be used to communicate with solvers, and is a possible future format to be considered by PySP (in addition to the presently supported LP and NL formats). However, very few solvers currently support OSiL/SE—or the core OSiL format in the case of deterministic math programs.

6 Progressive hedging: a generic decomposition strategy

We now transition from modeling stochastic programs and solving them directly via the extensive form to decomposition-based solution strategies, which are in practice typically required to efficiently solve large-scale instances with large numbers of scenarios, non-linearities, discrete variables, or decision stages. There are two broad classes of decomposition-based strategies in stochastic programming: horizontal and vertical. *Vertical* strategies decompose a stochastic program by stages; the L-shaped method of Van Slyke and Wets [54] is the primary method in this class, targeted to two-stage problems. Nested decomposition schemes [6,22] extend the basic L-shaped method to the multi-stage case. In contrast, *horizontal* strategies decompose a stochastic program by scenario; Rockafellar and Wets' [46] Progressive Hedging (PH) algorithm and Caroe and Schultz's [9] Dual Decomposition (DD) algorithm are two notable methods in this class.

Currently, there is not a large body of literature to provide an understanding of practical, computational aspects of stochastic programming solvers, particularly in the non-linear and mixed integer cases. For any given problem class, there are few heuristics to guide selection of the algorithm likely to be most effective. Similarly, while stochastic programming solvers are typically parameterized and/or configurable, there is little guidance available regarding how to select particular parameter values or configurations for a specific problem. Lacking such knowledge, the interface to solver libraries must provide facilities to allow for easily selecting parameters and configurations.

Beyond the need for highly configurable solvers, solvers should also be generic, i.e., independent of any particular AML description. Decomposition strategies are non-trivial to implement, requiring significant development time—especially when more advanced features are considered. The lack of generic decomposition solvers is a known impediment to the broader adoption of stochastic programming. Thénié et al. [51] concisely summarize the challenge as follows: "Devising efficient solution methods is still an open field. It is thus important to give the user the opportunity to experiment with solution methods of his choice." By introducing both customizable and generic solvers, our goal is to promote the broader use of and experimentation with stochastic programming by significantly reducing the barrier to entry.

In this section, we discuss the interface to and implementation of a generic implementation of Progressive Hedging. Our selection of this particular decomposition

algorithm is based largely on our successful experience with PH in solving difficult, multi-stage mixed-integer stochastic programs. However, the method is equally applicable to multi-stage linear and non-linear stochastic programs, as we discuss. In Sect. 6.1 we introduce the Progressive Hedging algorithm, and discuss its use in both linear and mixed-integer stochastic programming contexts. The interface to the PySP script for executing PH given an arbitrary PySP model is described in Sect. 6.2. Finally, we present an overview of the generic implementation in Sect. 6.3.

6.1 The Progressive Hedging algorithm

Progressive Hedging (PH) is a horizontal or scenario-based decomposition technique for solving stochastic programs, which possesses theoretical convergence properties when all decision variables are continuous. In particular, the algorithm converges in linear time given a convex reference scenario optimization model. PH was initially introduced as a decomposition strategy for solving large-scale stochastic linear programs; Rockafellar and Wets [46] further note the potential application of PH to solving nonlinear stochastic programs.

Despite its introduction in the context of stochastic linear and non-linear programs, PH has proved to be a very effective heuristic for solving stochastic mixed-integer programs [13,14,31,39,40]. PH is particularly effective in this context when there exist computationally efficient techniques for solving the deterministic single-scenario optimization problems. A key advantage of PH in the mixed-integer case is the absence of requirements concerning the number of stages or the type of variables allowed in each stage—as is common for many proposed stochastic mixed-integer algorithms. A disadvantage is the current lack of provable convergence and optimality results. For large, real-world stochastic mixed-integer programs, the determination of optimal solutions is generally not computationally tractable.

The basic idea of PH for the linear case is as follows:

1. For each scenario s, solutions are obtained for the problem of minimizing, subject to the problem constraints, the deterministic f_s (Formulation P_s).
2. The variable values for an implementable—but likely not admissible—solution are obtained by averaging over all scenarios at a scenario tree node.
3. For each scenario s, solutions are obtained for the problem of minimizing, subject to the problem constraints, the deterministic f_s (Formulation P_s) plus terms that penalize the lack of implementability using a sub-gradient estimator for the non-anticipativity constraints and a squared proximal term.
4. If the solutions have not converged sufficiently and the allocated compute time is not exceeded, goto Step 2.
5. Post-process, if needed, to produce a fully admissible and implementable solution.

To begin the PH implementation for solving formulation (P), we first organize the scenarios and decision stages into a tree. The leaves correspond to scenario realizations, such that each leaf is connected to exactly one node at stage $t \in \mathcal{T}$ and each of these nodes represents a unique realization up to stage t. The leaf nodes are connected to nodes at stage $t - 1$, such that each scenario associated with a node at stage $t - 1$ has the same realization up to stage $t - 1$. This process is iterated back to stage 1

(i.e., "now"). Two scenarios whose leaves are both connected to the same node at stage t have the same realization up to stage t. Consequently, in order for a solution to be implementable it must be true that if two scenarios are connected to the same node at some stage t, then the values of $x_i(t')$ must be the same under both scenarios for all i and for $t' \leq t$.

Progressive Hedging is a technique to iteratively and gradually enforce implementability, while maintaining admissibility at each step in the process. For each scenario s, approximate solutions are obtained for the problem of minimizing, subject to the constraints, the deterministic f_s plus terms that penalize the lack of implementability. These terms strongly resemble those found when the method of augmented Lagrangians is used [5]. The method makes use of a system of row vectors, w, that have the same dimension as the column vector system X, so we use the same shorthand notation. For example, $w(s)$ denotes $(w(s, 1), \ldots, w(s, |\mathcal{T}|))$ in the multiplier system.

To provide an algorithm statement of PH, we first develop notation for some of the scenario tree concepts. We use $\Pr(\mathcal{A})$ to denote the sum of $\Pr(s)$ over all scenarios s emanating from node $\mathcal{A}$ (i.e., those s that are the leaves of the sub-tree having $\mathcal{A}$ as a sub-tree root, also referred to as $s \in \mathcal{A}$). We use $t(\mathcal{A})$ to indicate the stage index for node $\mathcal{A}$ (i.e., node $\mathcal{A}$ contains scenario indices that correspond to scenarios with data that is the same up to stage t). We use $\overline{X}(t; \mathcal{A})$ on the left hand side of a assignment statement to indicate assignment to the vectors $(\overline{x}_1(s, t), \ldots, \overline{x}_{N(t)}(s, t))$ for each $s \in \mathcal{A}$, where $N(t)$ is the number of decision vector elements corresponding to stage t. The notation on the right-hand side of that assignment is similar: We refer to vectors $X(t; s)$ to indicate $(x_1(s, t), \ldots, x_{N(t)}(s, t))$. This notation enables us to succinctly express the computation and assignment of the average of decision elements for a node in the scenario tree.

The vectors at each iteration of PH are identified using a superscript; e.g., $w^{(0)}(s)$ is the multiplier vector for scenario s at PH iteration zero. The PH iteration counter is k. If we briefly defer the discussion of termination criteria, a formal version of the algorithm (with step numbering that matches in the informal statement just given) can be stated as follows, taking $\rho > 0$ as a parameter.

$k \longleftarrow 0$

1. For all scenario indices, $s \in \mathcal{S}$:

$$X^{(0)}(s) \longleftarrow \text{argmin}[f_s(X(s)) : X(s) \in \Omega_s] \tag{1}$$

and

$$w^{(0)}(s) \longleftarrow 0$$

$k \longleftarrow k + 1$

2. For each node, $\mathcal{A}$, in the scenario tree, and for $t = t(\mathcal{A})$:

$$\overline{X}^{(k-1)}(\mathcal{A}) \longleftarrow \sum_{s \in A} \Pr(s) X(t; s)^{(k-1)} / \Pr(\mathcal{A})$$

3. For all scenario indices, $s \in \mathcal{S}$:

$$w^{(k)}(s) \longleftarrow w^{(k-1)}(s) + (\rho)\left(X^{(k-1)}(s) - \overline{X}^{(k-1)}\right)$$

and

$$X^k(s) \longleftarrow \text{argmin}[f_s(X(s)) + w^{(k)}(s)X(s) + \rho/2 \left\| X(s) - \overline{X}^{k-1} \right\|^2 : X(s) \in \Omega_s]. \quad (2)$$

4. If the termination criteria are not met (e.g., solution discrepancies quantified via a metric $g^{(k)}$), then goto Step 3.

The termination criteria are based mainly on convergence, but we must also allow for the use of time-based termination because non-convergence is a possibility. Iterations are continued until k reaches some pre-determined limit or the algorithm has *converged*—which we take to indicate that the set of scenario solutions is sufficiently homogeneous. One possible definition is to require the inter-solution distance (e.g., Euclidean) to be less than some parameter.

The value of the perturbation vector ρ strongly influences the actual convergence rate of PH: if ρ is small, the penalty coefficients will vary little between consecutive iterations. To achieve tractable PH run-times, significant tuning and problem-dependent strategies for computing ρ are often required; mechanisms to support such tuning are described in Sect. 6.2.

We note that the generic PH implementation described subsequently has been applied by the authors and their collaborators to stochastic linear programs (hydro-thermal generator scheduling), stochastic non-linear programs (parameter estimation for infectious disease models) [58], and various stochastic mixed-integer linear programs (e.g., network design, sensor placement, forest harvesting, and generation expansion). We have also explored the use of PH in the context of stochastic mixed-integer non-linear programs, but this area of application is effectively an open research issue.

6.2 The **runph** script

Analogous to the **runef** script for generating and solving the extensive form, PySP provides a script—**runph**—to solve and post-process stochastic programs via PH. We now briefly describe the general usage of this script, followed by a discussion of some generally effective options to customize the execution of PH. As is the case with the **runef** script, all options begin with a double dash prefix. A number of key options are shared with the **runef** script: `--verbose`, `--model-directory`, `--instance-directory`, and `--solver`. In particular, the `--model-directory` and `--instance-directory` options are used to specify the PySP problem instance, while the `--solver` option is used to specify the solver applied to individual scenario sub-problems. The most general PH-specific options are:

`--max-iterations=MAX_ITERATIONS`
The maximal number of PH iterations. Defaults to 100.

`--default-rho=DEFAULT_RHO`
The default (global) ρ scalar parameter value for all variables with the exception of those appearing in the final stage. Defaults to 1.
`--termdiff-threshold=TERMDIFF_THRESHOLD`
The convergence threshold used to terminate PH (Step 6 of the pseudocode). Convergence is by default quantified as the difference between variable values and the mean scaled by the average and normalized by the number of variables to be blended. Defaults to 0.0001. This quantity is known as the *termdiff*.

In general, the default values for the maximum allowable iteration count, ρ, and the convergence threshold are likely to yield slow convergence of PH; for any real application, experimentation and analysis should be applied to obtain a more computationally effective configuration.

To illustrate the execution **runph** on a stochastic linear program, we again consider Birge and Louveaux's farmer problem. To solve the farmer problem with PySP, a user simply executes the following:

```
runph --model-directory=models \
      --instance-directory=scenariodata \
      --solver=cplex
```

which will result in eventual convergence to an optimal, admissible, and implementable solution—subject to the numerical tolerance issues. For the sake of brevity, we do not illustrate the output here; the final solution is reported in a format identical to that illustrated in Sect. 4. The quantity of information generated by PH can be significant, e.g., including the penalty weights and solutions for each scenario problem $s \in \mathcal{S}$ at each iteration. However, this information is not generated by default. Rather, simple summary information, including the value of $g^{(k)}$ at each PH iteration k, is output. As is theoretically promised in the case of stochastic linear programs, **runph** does converge given a linear PySP input model. The exact number of iterations depends in part on the precise solver used; on our test platform, for example, convergence is achieved in 48 iterations using CPLEX 11.2.1. It should be noted that for many stochastic linear—and even small, mixed-integer—programs (including the farmer example), any implementation of PH may solve significantly *slower* than the extensive form. This behavior is primarily due to the overhead associated with communicating with solvers for each scenario, for each PH iteration. However, this overhead is not significant with larger and/or more difficult scenario problems.

Finally, we critically note that by default the solver supplied to the **runph** script must be capable of handling the quadratic proximal terms introduced by PH to augment the original objective function. Examples of such solvers supported by PySP (through the Coopr optimization interface library) include CPLEX, Gurobi, and Ipopt. If such solvers are not available, the **runph** script supports automatic linearization of these quadratic terms, as described in detail below. If **runph** is run without such linearization using a solver that does not support quadratic terms, then the script will fail with an indication that the associated scenario sub-problems could not be solved.

Setting variable-specific ρ. In many applications, no single value of ρ for all variables yields a computationally efficient PH configuration. Consider the situation in which the objective is to minimize expected investment costs in a spare parts supply chain, e.g., for maintaining an aircraft fleet. The acquisition cost for spare parts is highly variable, ranging from very expensive (engines) to very cheap (gaskets). If ρ values are too small, e.g., on the order of the price of a tire, PH will require large iteration counts to achieve changes—let alone convergence—in the decision variables associated with engine procurement counts. If ρ values are too high, e.g., on the order of the price of an engine, then the PH weights w associated with gasket procurement counts may converge too quickly, yielding sub-optimal variable values. Alternatively, we have observed in many examples that PH sub-problem solves may "over-shoot" the optimal variable value, resulting in oscillation. Various strategies for computing variable-specific ρ are discussed in Watson and Woodruff [56].

To support the implementation of variable-specific ρ strategies in PySP, we define the following command-line option to **runph**:

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

The rho configuration file is a piece of executable Python code that computes the desired ρ. This allows for the expression of arbitrarily complex formulas and procedures. An example of such a configuration file, used in conjunction with the PySP SIZES example [32], is as follows:

```
m = self._model_instance # syntactic sugar

for i in m.ProductSizes:
 self.setRhoAllScenarios(m.ProduceSizeFirstStage[i], \
                         m.SetupCosts[i] * 0.001)
 self.setRhoAllScenarios(m.NumProducedFirstStage[i], \
                         m.UnitProductionCosts[i] * 0.001)
 for j in m.ProductSizes:
  if j <= i:
    self.setRhoAllScenarios(m.NumUnitsCutFirstStage[i,j], \
                            m.UnitReductionCost * 0.001)
```

The **self** object in the script refers to the PH object itself, which in turn possesses an attribute **_model_instance**. The **_model_instance** attribute represents the deterministic reference model instance, from which the full set of problem variables can be accessed. The example script implements a simple cost-proportional ρ strategy, in which ρ is specified as a function of a variable's objective function cost coefficient. Once the appropriate ρ value is computed, the script invokes the **setRhoAllScenarios** method of the PH object, which distributes the computed ρ value to the corresponding parameter of each of the scenario problem instances. It is also possible to set the ρ values on a per-variable, per-scenario basis; however, there are currently no reported strategies that effectively use this mechanism.

The customization strategy underlying the PySP variable-specific ρ mechanism is a limited form of callback in which the core PH code temporarily hands control back

to a user script to set specific model parameters. While the code is necessarily executable Python, the constrained scope is such that very limited knowledge of the Python language is required to write such an extension.

Linearization of the proximal penalty terms. At each iteration $k \geq 1$ of PH, scenario sub-problem solves involve an augmented form of the original optimization objective, with both linear and quadratic penalty terms. The presence of the quadratic terms can cause significant practical computational difficulties. At present, no open-source linear or mixed-integer solvers currently support quadratic objective terms in an integrated, robust manner. While most commercial solvers can handle problems with quadratic linear and mixed-integer objectives, solver efficiency is often dramatically worse relative to the linear case: we have consistently observed quadratic scenario sub-problems requiring an order of magnitude or more of run-time for solution than their linearized counterparts. The presence of quadratic terms is not a significant factor for non-linear solvers (e.g., Ipopt and Bonmin), where instead sub-problem size and achieving global optimality are the primary concerns.

To address this issue, the **runph** script provides for automatic linearization of quadratic penalty terms in PH. We first observe that a linear expression results from the expansion of any quadratic penalty term involving binary variables. Consequently, the default behavior is to linearize these terms for binary variables. To linearize penalty terms involving continuous and general integer variables (via simple linear interpolation between sampled points of the quadratic term), the **runph** script allows specification of the following options:

`--linearize-nonbinary-penalty-terms=BPTS`
Approximate the PH quadratic term for non-binary variables with a piece-wise linear function. The argument BPTS gives the number of breakpoints in the linear approximation. Defaults to 0, indicating linearization is disabled.

`--breakpoint-strategy=BREAKPOINT_STRATEGY`
Specify the strategy to distribute breakpoints on the [lb, ub] interval of each variable when linearizing. Defaults to 1, indicating a uniform distribution of BPTS breakpoints between lb and ub. A value of 2 distributes breakpoints uniformly between current minimum and maximum values observed for the variable at the corresponding node in the scenario tree; segments between the node min/max values and the variable lower and upper bounds are also automatically generated. A value of 3 places half of the BPTS breakpoints on either side of the observed variable average at the corresponding node in the scenario tree, with exponentially increasing distance from the mean.

To linearize a proximal term, **runph** requires that both lower and upper bounds (respectively denoted *lb* and *ub*) be specified for each variable in each scenario instance. This is most straightforwardly accomplished by specifying bounds or rules for computing bounds in each of the variable declarations appearing in the base deterministic scenario model. In reality, lower and upper bounds can be specified for all variables, even if trivially. If for some reason bounds are not easily specified in the deterministic scenario model, the option `--bounds-cfgfile` option is available, which functions in a fashion similar to the mechanism for setting variable-specific ρ described

above. Note that if a breakpoint would be very close to a variable bound, then the breakpoint is omitted. In other words, the BPTS parameter serves as an upper bound on the number of actual breakpoints.

By introducing automatic linearization of the proximal penalty term, PySP enables both a much broader base of solvers to be used in conjunction with PH and more efficient utilization of those solvers. In particular, it facilitates the use of open-source solvers—which can be critical in parallel environments in which it may be infeasible to procure large numbers of commercial solver licenses for concurrent use (see Sect. 8).

Other command-line options. While not discussed here, the **runph** script also provides options to control the type and extent of output at each iteration (weights and/or solutions), specify solver options, report exhaustive timing information, and track intermediary solver files. In general, these are provided for more advanced users; more information can be obtained by supplying the `--help` option to **runph**.

6.3 Implementation details

We now discuss high-level aspects of the implementation of the **runph** script, emphasizing the mechanisms linking the PH implementation with a generic Pyomo specification of the stochastic program. In doing so, our objective is to illustrate the power of embedding an algebraic modeling language within a high-level programming language, and specifically one that enables object introspection.

The PySP PH initialization process is similar to that for the EF writer/solver: the scenario tree, reference Pyomo instance, and scenario Pyomo instances are all created and initialized from user-supplied data. Without loss of generality, we assume two-stage problems in the following discussion. Following this general initialization, for each first-stage variable PH must create the corresponding: (1) ρ parameter, (2) node average vector $\overline{x}$, and (3) weight vector w. This is accomplished by accessing the information in the **StageVariables** set. In the farmer example, the **StageVariables** set contains the singleton string "DevotedAcreage[*]". The "*" in this example indicates that non-anticipativity must be enforced at the root node for all indices of the variable **DevotedAcreage**: `DevotedAcreage[CORN]`, `DevotedAcreage[SUGAR_BEETS]`, and `DevotedAcreage[WHEAT]`.

Using Python introspection (via the `getattr` built-in function to query object attributes by name), PySP accesses the corresponding variable objects in the reference model instance. From the variable object, the index set (also a first-class Python object) is extracted and cloned, eliminating all indices (none, in the case of a template equal to "*") not matching the specified template.

PySP uses the newly constructed index set to create new parameter objects representing the ρ, weight w, and node average $\overline{x}$ corresponding to the identified variable; the index set is the first argument to the parameter class constructor. Using the Python **setattr** method, the ρ and w parameters are attached to the appropriate scenario instance (the process is repeated for each scenario), while the node average $\overline{x}$ is attached to the root node object in the scenario tree. The ability to create object attributes on-the-fly is directly supported in dynamic languages such

as Python or Java, as opposed to C++ or other static and compiled strongly typed languages.

Following initialization, PH solves the original scenario sub-problems and loads the resulting solutions into the corresponding Pyomo instances. Using the same dynamic object query mechanism, PySP computes the first-stage variable averages and stores the result in the newly created parameters in the scenario tree. An analogous process is then used to compute and store the current w parameter values for each scenario. Before executing PH iterations $k \geq 1$, PySP must augment the original objective expressions with the linear and quadratic penalty terms discussed in Sect. 6.1. Because the Pyomo scenario instances and their attributes (e.g., parameters, variables, constraints, and objectives) are first-class Python objects, their contents can be programmatically modified at run-time. Consequently, it is straightforward to—for each first-stage variable—identify the corresponding variable, weight parameter, and average parameter objects, create objects representing the penalty terms, and augment the original optimization objective.

In summary, the processes described above rely on three capabilities explicitly facilitated through the use of Python. First, user-specified strings (e.g., first stage variables names) can be manipulated to dynamically identify attributes of objects (e.g., variables of scenario instances). Second, all elements of the Pyomo algebraic modeling language (e.g., parameters, variables, constraints, and objectives) are first-class Python objects, and as a consequence can be programmatically queried, cloned, and—most importantly—modified. Third, Python allows for the dynamic addition of attributes to objects (e.g., weight and ρ parameters to scenario instances). None of these enabling features of Python are particularly advanced, and are in general easy to use. Rather, these are key properties of a dynamic high-level programming language, which can be effectively leveraged to construct generic solvers for stochastic programming.

7 Progressive hedging extensions: advanced configuration

The basic PySP PH implementation is by design customizable to a rather limited degree: mechanisms are provided to allow for specification of ρ values and linearization of the PH objective. In either case, core PH functionality is not perturbed. We now describe more extensive and intrusive customization of the PySP PH behavior. In Sect. 7.1, we describe the interface to a PH extension providing functionality that is often critical to achieving good performance on stochastic mixed-integer programs. Some components are additionally useful in the case of stochastic linear programs. We then discuss in Sects. 7.2 and 7.3 command-line options that enable functionality commonly used in PH practice. Finally, we discuss in Sect. 7.4 the programmatic facilities that PySP provides to users (typically programmers) that want to develop their own extensions.

7.1 Convergence accelerators and mixed-integer heuristics

The basic PH algorithm can converge slowly, even if appropriate values of ρ have been computed. Further, in the mixed-integer case, PH can exhibit cyclic behavior,

preventing convergence. Consequently, PH implementations in practice are augmented with methods to both accelerate convergence and prevent cycling. Many of these extensions are either described or introduced in Watson and Woodruff [56].

The PySP implementation of PH provides these extensions in the form of a *plugin*, i.e., a piece of code that extends the core functionality of the underlying algorithm, at well-defined points during execution. This "Watson–Woodruff" (WW) plugin generalizes the accelerator and cycle-avoidance mechanisms described in Watson and Woodruff [56]. The Python module implementing this plugin is named `wwextension.py`; general users do not need to understand the contents of this module.

The **runph** script provides three command-line options to control the execution of the Watson–Woodruff extensions plugin:

`--enable-ww-extensions`
Enable the Watson–Woodruff PH extensions plugin. Defaults to False.
`--ww-extension-cfgfile=WW_EXTENSION_CFGFILE`
The name of a configuration file for the Watson–Woodruff PH extensions plugin. Defaults to "wwph.cfg".
`--ww-extension-suffixfile=WW_EXTENSION_SUFFIXFILE`
The name of a variable suffix file for the Watson–Woodruff PH extensions plugin. Defaults to "wwph.suffixes".

As discussed in Sect. 7.4, user-defined extensions can co-exist with the Watson–Woodruff extension.

Before discussing the configuration of this extension (which necessarily relies on problem-specific knowledge), we provide more motivation and algorithmic detail underlying the extension:

- *Convergence detection*. A detailed analysis of PH behavior on a variety of problems indicates that individual decision variables frequently converge to specific, fixed values across all scenarios in early PH iterations. Further, despite interactions among the variables, this 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 converged to an identical value across all scenarios for some number of iterations, fix it to that value. However, the strategy must be used carefully. In particular, for problems where the problem constraints impose both upper and lower bounds on variables x, these methods may result in PH encountering infeasible scenario sub-problems even though the problem is ultimately feasible.
- *Cycle detection*. In the presence of integer variables, PH occasionally exhibits cycling behavior. 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 weight vectors w, heuristically implemented using a simple hashing scheme [57] 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 (using problem-specific, user-supplied knowledge) across scenarios in order to break the cycle.
- *Convergence-based sub-problem optimality thresholds*. A number of researchers have noted that it is unnecessary to solve scenario sub-problems to optimality

in early PH iterations [29]. In these early iterations, the primary objective is to quickly obtain coarse estimates of the PH weight vectors, which (at least empirically) does not require optimal solutions to scenario sub-problems. Once coarse weight estimates are obtained, optimal solutions can then be pursued to tune the weight vectors in the effort to achieve convergence. Given a measure of scenario solution homogeneity (e.g., the convergence threshold $g^{(k)}$), a commonly used strategy is to set the solver *mipgap*—a termination threshold based on the difference in current lower and upper bounds—in proportion to this measure.

7.1.1 Mipgap control and cycle detection parameters

The WW extension defines and exposes a number of key user-controllable parameters, each of which can be optionally specified in the WW PH configuration file (named `wwph.cfg` by default). The full range of parameters available is documented in the PySP user's manual, installed with the software. Informally, parameters are available for controlling mipgaps and cycle detection logic.

Users specify values for these parameters in the WW PH configuration file, which is loaded by specifying the `--ww-extension-cfgfile` **runph** command-line option. To simplify implementation, the parameters are set directly using Python syntax, e.g., as follows:

```
self.Iteration0MipGap = 0.1
self.InitialMipGap = 0.05
self.FinalMipGap = 0.001
```

The contents of the configuration file are read by the WW extension following initialization. The `self` identifier refers to the WW extension object itself; the file contents are directly executed by the WW extension via the Python `execfile` command. While powerful and simplistic, this approach to initialization is potentially dangerous, as any attribute of the WW extension object is subject to manipulation.

7.1.2 General variable fixing and slamming parameters

Variable fixing is often an empirically effective heuristic for accelerating PH convergence. Fixing strategies implicitly rely on strong correlation between the converged value of a variable across all scenario sub-problems in an intermediate PH iteration and the value of the variable in the final solution should no fixing be imposed. Variable fixing reduces scenario sub-problem size, accelerating solve times. However, depending on problem structure, the strategy can lead to either sub-optimal solutions (due to premature declarations of convergence) or the failure of PH to converge (due to interactions among the constraints). Consequently, careful and problem-dependent tuning is typically required to achieve an effective fixing strategy. To facilitate such tuning, the WW PH extension allows for specification of various global

parameters to control fixing, e.g., the conditions under which discrete and continuous variables can be fixed as a function of the number of PH iterations over which convergence is observed. The full set of parameters is documented in the PySP user's manual.

Fixing strategies at iteration 0 are typically distinct from those in subsequent iterations, e.g., iteration 0 agreement of acquisition quantities in a resource allocation problem to a value of 0 may (depending on the problem structure) indicate that no such resources are likely to be required. In general, longer lag times for PH iterations $k \geq 1$ yield better solutions, albeit at the expense of longer run-times; this trade-off is numerically illustrated in Watson and Woodruff [56]. Differentiation between fixing behaviors at lower bounds, upper bounds, or intermediate values are typically necessary due to variable problem structure (e.g., variables being constrained from lower or upper bounds).

For many mixed-integer problems, PH can spend a disproportionately large number of iterations "fine-tuning" the values of a small number of variables in order to achieve convergence. Consequently, it is often desirable to force early agreement of these variables, even at the expense of sub-optimal final solutions. This mechanism is referred to as *slamming* in Watson and Woodruff [56]. Slamming is also used to break cycles detected through the mechanisms described above. The WW PH extension supports a number of configuration options to control variable slamming, e.g., the number of iterations allowed to proceed before slamming is enabled, or the values to which a variable can be slammed. The full set of variable slamming options is documented in the PySP user's manual.

Slamming to the minimum and maximum scenario tree node values is often useful in resource allocation problems. For example, it is frequently safe with respect to feasibility to slam a variable value to the scenario maximum in the case of one-sided "diet" problems (i.e., problems in which there are no constraints that imply upper bounds on the resources needed, which means that additional resources strictly inflate solution cost but cannot result in infeasibility; see Watson and Woodruff [56] for additional details). In the event that multiple slamming options are available, the priority order is given as: lower bound, minimum, upper bound, maximum, and anywhere.

7.1.3 Variable-specific fixing and slamming parameters

Global controls for variable fixing and slamming are generally useful, but for many problems more fine-grained control is required. For example, in one-sided diet problems, feasibility can be maintained during slamming by fixing a variable value at the maximal level observed across scenarios (assuming a minimization objective) [56]. Similarly, it is often desirable in a multi-stage stochastic program to fix variables appearing in early stages before those appearing in later stages, or to fix binary variables for siting decisions in facility location prior to discrete allocation variables associated with those sites.

The WW PH extension provides fine-grained, variable-specific control of both fixing and slamming using the concept of *suffixes*, similar to the mechanism employed by AMPL [3]. Global defaults are established using the mechanisms described in

Sect. 7.1.2, while optional variable-specific over-rides are specified via the suffix mechanism we now introduce.

The specific suffixes (fully documented in the PySP user's manual) recognized by the WW PH extension include analogous, variable-specific functionality to that provided by the parameters described in Sect. 7.1.2. In addition, we introduce the suffix `SlammingPriority`, which allows for prioritization of variables slammed during convergence acceleration; larger values indicate higher priority. The latter are particularly useful, for example, in the context of resource allocation problems in which early slamming of lower-cost items tends to yield lower-cost final solutions.

Variable-specific suffixes are supplied to the WW PH extension in a file, the name of which is communicated to the **runph** script through the `--ww-extension-suffixfile` option. An example of a suffix file (notionally implementing the motivational examples described above) is as follows:

```
resource_quantity[*,Stage1] FixWhenItersConvergedAtUB 10
resource_quantity[*,Stage2] FixWhenItersConvergedAtUB 50

resource_quantity[TIRES, Stage1] SlammingPriority 50
resource_quantity[ENGINES, Stage1] SlammingPriority 10

resource_quantity[TIRES, *] CanSlamToMax True
resource_quantity[ENGINES, *] CanSlamToMax True
```

In general, suffixes are specified via (`VARSPEC`, `SUFFIX`, `VALUE`) triples, where `VARSPEC` indicates a variable slice (i.e., a template that matches one or more indices of a variable; if the variable is not indexed, only the variable name is specified), `SUFFIX` indicates the name of a suffix recognized by the WW PH extension, and `VALUE` indicates the quantity associated with the specified suffix (and is expected to be consistent with the type of value expected by the suffix). If no suffix is associated with a given variable, then the global default parameter values are accessed.

In terms of implementation, suffixes are easily processed via Python's dynamic object attribute functionality. For each `VARSPEC` encountered, the index template (if it exists) is expanded and all matching variable value objects are identified. Then, for each variable value, a call to `setattr` is performed to attach the corresponding attribute/value pair to the object. The advantage of this approach is simplicity and generality: any suffix can be applied to any variable. The disadvantage is the lack of error-checking, in that suffixes unknown to the WW PH extension can inadvertently be specified, e.g., a capitalized `SLAMMINGPRIORITY` suffix. However, specification of unknown suffixes is benign, in the sense that they will be stored but never queried by the system.

7.2 Solving a constrained extensive form

A common practice in using PH as a mixed-integer stochastic programming heuristic involves running PH for a limited number of iterations (e.g., via the `--max-iterations` option), fixing the values of discrete variables that appear to have converged, and then solving the significantly smaller extensive form that results

[40]. The resulting compressed extensive form is generally far smaller and easier to solve than the original extensive form. This technique directly avoids issues related to the empirically long number of PH iterations required to resolve relatively small remaining discrepancies in scenario sub-problem solutions. Any disadvantage stems from variable fixing itself, i.e., premature fixing of variables can lead to sub-optimal extensive form solutions.

To write and solve the extensive form following PH termination, we provide the following options in the **runph** script:

`--write-ef`
Upon termination, write the extensive form of the model. Disabled by default.
`--solve-ef`
Following write of the extensive form model, solve the extensive form and display the resulting solution. Disabled by default.
`--ef-output-file=EF_OUTPUT_FILE`
The name of the extensive form output file. Defaults to "efout.lp".

When writing the extensive form, all variables whose value is currently fixed in any scenario sub-problem are automatically (via Pyomo) preprocessed into constant terms in any referencing constraints or the objective. Solver selection is controlled with the `--solver` keyword, and is identical to that used for solving scenario sub-problems. The **runph** script additionally provides mechanisms for specifying solver options (including mipgap) specific to the extensive form solve.

7.3 Alternative convergence criteria

The PySP PH implementation supports a variety of alternative convergence metrics, enabled via the following **runph** command-line options:

`--enable-termdiff-convergence`
Terminate PH based on the termdiff convergence metric, which is defined as the unscaled sum of differences between variable values and the mean. Defaults to False.
`--enable-normalized-termdiff-convergence`
Terminate PH based on the normalized termdiff convergence metric. Each term in the termdiff sum is scaled by the average variable value and the overall sum is normalized by the number of variables to blend (i.e. number of variables for which non-anticipativity must be enforced). Defaults to True.
`--enable-free-discrete-count-convergence`
Terminate PH based on the free discrete variable count convergence metric, which is a function of the current number of non-fixed, non-anticipative discrete variables in the scenario tree (e.g., all but those in the final stage). Defaults to False.
`--free-discrete-count-threshold=FREE_DISCRETE_COUNT_THRESHOLD`
The convergence threshold associated with the free discrete variable count convergence metric. PH will terminate once the number of free discrete variables (see definition immediately above) drops below this threshold.

Only a single termination criterion can be activated for any given run; the software will warn and exit if multiple criteria are enabled. The default value for the **–enable-termdiff-convergence** ensures that at least one criterion is always active. The termination criterion associated with the free discrete variable count is particularly useful when deployed in in conjunction with the capability to solve restricted extensive forms described in Sect. 7.2.

7.4 User-defined extensions

The Watson–Woodruff PH extensions described in Sect. 7.1 are built upon a simple, general callback framework in PySP for developing user-defined extensions to the core PH algorithm. While most modelers and typical PySP users would not make use of this feature, programmers and algorithm developers can easily leverage the capability. The interface for user-defined PH extensions is defined in a PySP read-only file called `phextension.py`. The file contents are supplied as follows, to identify the points at which **runph** temporarily transfers control to the user-defined extension:

```
class IPHExtension(Interface):

    def post_ph_initialization(self, ph):
        """ Called after PH initialization."""
        pass

    def post_iteration_0_solves(self, ph):
        """ Called after the iteration 0 solves."""
        pass

    def post_iteration_0(self, ph):
        """ Called after the iteration 0 solves, averages \
            computation, and weight update."""
        pass

    def post_iteration_k_solves(self, ph):
        """ Called after the iteration k solves."""
        pass

    def post_iteration_k(self, ph):
        """ Called after the iteration k solves, averages \
            computation, and weight update."""
        pass

    def post_ph_execution(self, ph):
        """ Called after PH has terminated."""
        pass
```

To create a user-defined extension, one simply needs to define a Python class that implements the PH extension interface shown above, e.g., via the following code fragment:

```
from pyutilib.component.core import *
from coopr.pysp import phextension

class examplephextension(SingletonPlugin):

    implements (phextension.IPHExtension)

    def post_instance_creation(self, ph):
      print ''Done creating PH scenario sub-problem instances!''

    # over-ride for each defined callback in phextension.py

    ...
```

The full example PH extension is supplied with PySP, in the form of the Python file `testphextension.py`. All Coopr user plugins are derived from a `SingletonPlugin` base class (indicating that there cannot be multiple instances of each type of user-defined extension), which can for present purposes be viewed simply as a necessary step to integrate the user-defined into the Coopr framework in which PySP is embedded. We defer to Hart and Siirola [27] for an in-depth discussion of the Coopr plugin framework leveraged by PySP.

Each transfer point (i.e., callback) in the user-defined extension is supplied the PH object, which includes the current state of the scenario tree, reference instance, all scenario instances, PH weights, etc. User code can then be developed to modify the state of PH (e.g., current solver options) or variable attributes (e.g., fixing as in the case of the Watson–Woodruff extension).

To use a customized extension with **runph**, the user invokes the command-line option `--user-defined-extension=EXTENSIONFILE`. Here, `EXTENSIONFILE` is the Python module name, which is assumed to be either in the current directory or in some directory specified via the `PYTHONPATH` environment variable. Finally, we observe that both a user-defined extension and the Watson–Woodruff PH extension can co-exist. However, the Watson–Woodruff extension will be invoked prior to any user-defined extension.

8 Solving PH scenario sub-problems in parallel

One immediate benefit to embedding PySP and Pyomo in a high-level language such as Python is the ability to leverage both native and third-party functionality for distributed computation. Specifically, Python's **pickle** module provides facilities for object serialization, which involves encoding complex Python objects—including Pyomo instances—in a form (e.g., a byte stream) suitable for transmission and/or storage. The third-party, open-source Pyro (Python Remote Objects) package [43] provides capabilities for distributed computing, building on Python's **pickle** serialization functionality.

PySP currently supports distributed solves, accessible from both the **runef** and **runph** scripts. At present, only a simple client-server paradigm is supported in the publicly available distribution. The general distributed solver capabilities provided

in the Coopr library are discussed in Hart et al. [28], including the mechanisms and scripts by which name servers (used to locate distributed objects) and solver servers (daemons capable of solving MIPs, for example) are initialized and interact. Here, we simply describe the use of a distributed set of solver servers in the context of PySP.

Both the **runef** and **runph** scripts are implemented such that all requests for the solution of scenario sub-problems are mediated by a "solver manager". The default solver manager in both scripts is a serial solver manager, which executes all solves locally. Alternatively, a user can invoke a remote solver manager by specifying the command-line option `--solver-manager=pyro`. The remote (pyro) solver manager identifies available remote solver daemons, serializes the relevant Pyomo model instance for communication, and initiates a solve request with the daemon. After the daemon has solved the instance, the solution is returned to the remote solver manager, which then transfers the solution to the invoking script.

The accessibility of remote solvers within the PySP PH implementation immediately confers the benefit of trivial parallelization of scenario sub-problem solves. In the case of commercial solvers, all available licenses can be leveraged. In the open-source case, cluster solutions can be deployed in a straightforward manner. Parallelism in PySP most strongly benefits stochastic mixed-integer and non-linear program solves, in which the difficulty of scenario sub-problems masks the overhead associated with object serialization and client-server communication. At the same time, parallel efficiency necessarily *decreases* as the number of scenarios increases, due to high variability in mixed-integer and non-linear solve times and the presence of barrier synchronization points in PH (after Step 2 in the pseudocode introduced in Sect. 6.1). However, parallel efficiency is of increasingly diminishing concern relative to the need to support high-throughput computing.

For solving the extensive form, remote solves serve a different purpose: to facilitate access to a central computing resources, with associated solver licenses. For example, our primary workstation for developing PySP is a 16-core workstation with a large amount of RAM (96GB). All commercial solver licenses are localized to this workstation, which in turn exposes parallelism enabled by now-common multi-threaded solver implementations.

Given the growing availability of cluster-based computing resources and the increasing accessibility of solver licenses (as is particularly the case for academics, with access to free licenses for most commercial solvers), the use of distributed computation by PySP users is expected to continue to grow.

9 Conclusions

Despite the potential of stochastic programming to solve real-world decision problems involving uncertainty, the use of this tool in industry is far from widespread. Historical impediments include the lack of stochastic programming support in algebraic modeling tools and the inability to experiment with and customize stochastic programming solvers, which are immature relative to their deterministic counterparts.

We have described PySP, an open-source software tool designed to address both impediments simultaneously. PySP users express stochastic programs using Pyomo,

an open-source algebraic modeling language co-developed by the authors. In addition to exhibiting the benefits of open-source software, Pyomo is based on the Python high-level programming language, allowing generic access and manipulation of model elements. PySP is also embedded within Coopr, which provides a wide range of solver interfaces (both commercial and open-source), problem writers, solution readers, and distributed solvers.

PySP leverages the features of Python, Pyomo, and Coopr to develop model-independent algorithms for generating and solving the extensive form directly, and solving the extensive form via scenario-based decomposition (PH). The resulting PH implementation is highly configurable, broadly extensible, and trivially parallelizable. PySP serves as a novel case study in the design of generic, configurable, and extensible stochastic programming solvers, and illustrates the benefits of integrating the core algebraic modeling language within a high-level programming language.

PySP has been used to develop and solve a number of difficult stochastic linear, non-linear, and mixed-integer linear multi-stage programs, and is under active use and development by the authors and their collaborators. PySP ships with a number of academic examples that have been used during the development effort, including examples from Birge and Louveaux's text, a network flow problem [56], and a production planning problem [32]. Real-world applications that have either been completed in PySP or are under active investigation include biofuel network design [30], forest harvesting [4], wind farm network design, sensor placement, and electrical grid generation expansion.

In-progress and future PySP development efforts include new solvers (e.g., the L-shaped method), scenario bundling techniques, and support for large-scale parallelism. In addition, we have recently integrated capabilities for confidence interval estimation on solution quality.

Both PySP and the Pyomo algebraic modeling language upon which PySP is based are actively developed and maintained by Sandia National Laboratories. Both are packages distributed with the Coopr open-source Python project for optimization, which is now part of the COIN-OR open-source initiative [11].

Acknowledgments Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the US Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. This research was funded in part by the Department of Energy's Office of Advanced Scientific Computing Research, as part of the Complex Interconnected Distributed Systems program. The authors would like to acknowledge several early users of PySP and Pyomo, whose experience dramatically improved the quality and design of the software: Yueyue Fan (UC Davis), Yongxi Huang (UC Davis), Chien-Wei Chen (UC Davis), and Fernando Badilla Veliz (University of Chile). The authors would also like to acknowledge the anonymous referees, whose comments led to a significantly improved manuscript.

Appendix: Getting started

Both PySP and the underlying Pyomo modeling language are distributed with the Coopr software package. All documentation, information, and source code related to Coopr is available from: http://www.software.sandia.gov/trac/coopr. Installation instructions are found by clicking on the *Download* tab found on the Coopr main

page, or by navigating directly to following web page: http://www.software.sandia.gov/trac/coopr/wiki/GettingStarted. A variety of installation options are available, including directly from the project SVN repositories or via release snapshots found on PyPi (http://www.pypi.python.org/pypi). Two Google group e-mail lists are associated with Coopr and all contained sub-projects, including PySP. The *coopr-forum* group is the main source for user assistance and release announcements. The *coopr-developers* group is the main source for discussions regarding code issues, including bug fixes and enhancements. Much of the information found at http://www.software.sandia.gov/trac/coopr is mirrored on the COIN-OR web site (http://www.projects.coin-or.org/Coopr); similarly, a mirror of the Sandia SVN repository is maintained by COIN-OR.

References

1. AIMMS: Optimization software for operations research applications. http://www.aimms.com/operations-research/mathematical-programming/stochastic-programming, July (2010)
2. Alonso-Ayuso, A., Escudero, L.F., Ortuño, M.T.: BFC, a branch-and-fix coordination algorithmic framework for solving some types of stochastic pure and mixed 0-1 programs. Eur. J. Oper. Res. **151**(3), 503–519 (2003)
3. AMPL: A modeling language for mathematical programming. http://www.ampl.com, July (2010)
4. Badilla, F.: Problema de Planificación Forestal Estocástico Resuelto a Traves del Algoritmo Progressive Hedging. PhD thesis, Facultad de Ciencias Físicas y Matemáticas, Universidad de Chile, Santiago, Chile (2010)
5. Bertsekas, D.P.: Constrained Optimization and Lagrange Multiplier Methods. Athena Scientific, Massachusetts (1996)
6. Birge, J.R.: Decomposition and partitioning methods for multistage stochastic linear programs. Oper. Res. **33**, 989–1007 (1985)
7. Birge, J.R., Dempster, M.A., Gassmann, H.I., Gunn, E.A., King, A.J., Wallace, S.W.: A standard input format for multiperiod stochastic linear program. COAL (Math. Prog. Soc. Commun. Algorithms) Newsletter **17**, 1–19 (1987)
8. Birge, J.R., Louveaux, F.: Introduction to Stochastic Programming. Springer, Berlin (1997)
9. Carøe, C.C., Schultz, R.: Dual decomposition in stochastic integer programming. Oper. Res. Lett. **24**(1–2), 37–45 (1999)
10. Chen, D.-S., Batson, R.G., Dang, Y.: Applied Integer Programming. Wiley, New York (2010)
11. COIN-OR: COmputational INfrastructure for Operations Research. http://www.coin-or.org, July (2010)
12. CPLEX: http://www.cplex.com, July (2010)
13. Crainic, T.G., Fu, X., Gendreau, M., Rei, W., Wallace, S.W.: Progressive hedging-based meta-heuristics for stochastic network design. Technical report CIRRELT-2009-03, University of Montreal CIRRELT, January (2009)
14. Fan, Y., Liu, C.: Solving stochastic transportation network protection problems using the progressive hedging-based method. Netw. Spatial Econ. **10**(2), 193–208 (2010)
15. FLOPCPP: Flopc++: Formulation of linear optimization problems in C++. http://www.projects.coin-or.org/FlopC++, August (2010)
16. Fourer, R., Gay, D.M., Kernighan, B.W.: AMPL: a mathematical programming language. Manage. Sci. **36**, 519–554 (1990)
17. Fourer, R., Lopes, L.: A management system for decompositions in stochastic programming. Ann. Oper. Res. **142**, 99–118 (2006)
18. Fourer, R., Lopes, L.: StAMPL: a filtration-oriented modeling tool for multistage recourse problems. INFORMS J. Comput. **21**(2), 242–256 (2009)
19. Fourer, R., Ma, J., Martin, K.: OSiL: an instance language for optimization. Comput. Optim. Appl. **45**(1), 181–203 (2010)
20. FrontLine: Frontline solvers: developers of the Excel solver. http://www.solver.com, July (2011)

21. GAMS: The General Algebraic Modeling System. http://www.gams.com, July (2010)
22. Gassmann, H.I.: MSLiP: a computer code for the multistage stochastic linear programming problem. Math. Program. **47**, 407–423 (1990)
23. Gassmann, H.I., Ireland, A.M.: On the formulation of stochastic linear programs using algebraic modeling languages. Ann. Oper. Res. **64**, 83–112 (1996)
24. Gassmann, H.I., Schweitzer, E.: A comprehensive input format for stochastic linear programs. Ann. Oper. Res. **104**, 89–125 (2001)
25. GUROBI: Gurobi optimization. http://www.gurobi.com, July (2010)
26. Hart, W.E., Laird, C.D., Watson, J.P., Woodruff, D.L.: Pyomo: Optimization Modeling in Python. Springer, Berlin (2012)
27. Hart, W.E., Siirola, J.D.: The PyUtilib component architecture. Technical report, Sandia National Laboratories (2010)
28. Hart, W.E., Watson, J.P., Woodruff, D.L.: Python optimization modeling objects (Pyomo). Math. Program. Comput. **3**, 219–260 (2011)
29. Helgason, T., Wallace, S.W.: Approximate scenario solutions in the progressive hedging algorithm: a numerical study. Ann. Oper. Res. **31**(1–4), 425–444 (1991)
30. Huang, Y.: Sustainable Infrastructure System Modeling under Uncertainties and Dynamics. PhD thesis, Department of Civil and Environmental Engineering, University of California, Davis (2010)
31. Hvattum, L.M., Løkketangen, A.: Using scenario trees and progressive hedging for stochastic inventory routing problems. J. Heurist. **15**(6), 527–557 (2009)
32. Jorjani, S., Scott, C.H., Woodruff, D.L.: Selection of an optimal subset of sizes. Int. J. Prod. Res. **37**(16), 3697–3710 (1999)
33. Kall, P., Mayer, J.: Building and solving stochastic linear programming models with SLP-IOR. In: Wallace, S.W., Ziemba, W.T. (eds.) Applications of Stochastic Programming, pp. 79–93. MPS-SIAM (2005)
34. Kall, P., Mayer, J.: Stochastic Linear Programming: Models, Theory, and Computation. Springer, Berlin (2005)
35. Karabuk, S.: An open source algebraic modeling and programming software. Technical report, University of Oklahoma, School of Industrial Engineering, Norman (2005)
36. Karabuk, S.: Extending algebraic modeling languages to support algorithm development for solving stochastic programming models. IMA J. Manage. Math. **19**, 325–345 (2008)
37. Karabuk, S., Grant, F.H.: A common medium for programming operations-research models. IEEE Softw. **24**(5), 39–47 (2007)
38. LINDO: LINDO systems, August (2010)
39. Listes, O., Dekker, R.: A scenario aggregation based approach for determining a robust airline fleet composition. Transport. Sci. **39**, 367–382 (2005)
40. Løkketangen, A., Woodruff, D.L.: Progressive hedging and tabu search applied to mixed integer (0,1) multistage stochastic programming. J. Heurist. **2**, 111–128 (1996)
41. Maximal Software: http://www.maximal-usa.com/maximal/news/stochastic.html, July (2010)
42. Parija, G.R., Ahmed, S., King, A.J.: On bridging the gap between stochastic integer programming and mip solver technologies. INFORMS J. Comput. **16**, 73–83 (2004)
43. PYRO: Python remote objects. http://pyro.sourceforge.net, July (2009)
44. Python: Python programming language—official website. http://python.org, July (2010)
45. Dive Into Python: http://diveintopython.org/power_of_introspection/index.html, July (2010)
46. Rockafellar, R.T., Wets, R.J.-B.: Scenarios and policy aggregation in optimization under uncertainty. Math. Oper. Res. **16**(1), 119–147 (1991)
47. Schultz, R., Tiedemann, S.: Conditional value-at-risk in stochastic programs with mixed-integer recourse. Math. Program. **105**(2–3), 365–386 (2005)
48. Shapiro, A., Dentcheva, D., Ruszczynski, A.: Lectures on stochastic programming: modeling and theory. Society for Industrial and Applied Mathematics (SIAM) (2009)
49. SMI: SMI. http://www.projects.coin-org.org/Smi, August (2010)
50. SUTIL: SUTIL—a stochastic programming utility library. http://www.coral.ie.lehigh.edu/~sutil, July (2011)
51. Thénié, J., van Delft, Ch., Vial, J.-Ph.: Automatic formulation of stochastic programs via an algebraic modeling language. Comput. Manage. Sci. **4**(1), 17–40 (2007)
52. Valente, C., Mitra, G., Sadki, M., Fourer, R.: Extending algebraic modelling languages for stochastic programming. INFORMS Journal On Computing **21**(1), 107–122 (2009)

53. Valente, P., Mitra, G., Poojari, C.A.: A stochastic programming integrated environment. In: Wallace, S.W., Ziemba, W.T. (eds.) Applications of Stochastic Programming, pp. 115–136. MPS-SIAM (2005)
54. Van Slyke, R.M., Wets, R.J.-B.: L-shaped linear programs with applications to optimal control and stochastic programming. SIAM J. Appl. Math. **17**, 638–663 (1969)
55. Wallace, S.W., Ziemba, W.T. (eds.): Applications of Stochastic Programming. Society for Industrial and Applied Mathematics (SIAM) and the Mathematical Programming Society (MPS) (2005)
56. Watson, J.P., Woodruff, D.L.: Progressive hedging innovations for a class of stochastic mixed-integer resource allocation problems. Comput. Manage. Sci. **8**(4), 355–370 (2011)
57. Woodruff, D.L., Zemel, E.: Hashing vectors for tabu search. Ann. Oper. Res. **41**(2), 123–137 (1993)
58. Word, D.P., Burke, D.A., Iamsirithaworn, D.S., Laird, C.D.: A nonlinear programming approach for estimation of transmission parameters in childhood infectious disease using a continuous time model. J. R. Soc. Interface (Under Review)
59. Xpress-Mosel. http://www.dashopt.com/home/products/products_sp.html, July (2010, to appear)
60. XpressMP: FICO express optimization suite. http://www.fico.com/en/products/DMTools/pages/FICO-Xpress-Optimization-Suite.aspx, July (2010)