# StAMPL: A Filtration-Oriented Modeling Tool for Stochastic Programming

Robert Fourer Leo Lopes

15th December 2003

**Abstract**

This research investigates how to create a modeling tool specifically for stochastic programming problems with recourse. By taking advantage of the special structure these problems have, we produce a modeling language whose syntax is less redundant, more modular, and more expressive than the notation commonly associated with stochastic programming. We then implement a system that can convert models written using this syntax to instances that can be solved using standard mechanisms. With this approach, we are able to represent models in a very clean, simple, and scalable format.

## 1. Introduction

Over the last decades, progress in solving stochastic optimization problems has been driven by considerable scientific development and by substantial increases in inexpensive computing power. Applications in areas involving very substantial amounts of capital, like finance and energy, have used stochastic programming extensively. The fundamental idea of stochastic programming — modeling decisions available over time in response to uncertain parameters — is, of course, far more applicable. A factor limiting the development of more stochastic optimization applications is that modeling stochastic programs is challenging. Many researchers

believe that the realm of applications for which stochastic programming is economically viable can be greatly expanded by discovering ways to make the modeling process simpler.

Most solution approaches for the class of stochastic programs studied in this research involve solving a much larger deterministic equivalent linear program. As a result, these stochastic programs have traditionally been viewed as extensions of multistage linear programs. However, from the modeler's perspective, conforming stochastic programs to the linear programming framework creates two problems: ambiguity in the treatment of the concepts of time and stage; and the use of extensive indexing to capture stage and scenario relationships fundamental to the stochastic programming framework.

Our solution to these problems is based on modeling stochastic programs as sequences of single-stage problems organized according to how new information impacts the decisions available. A (discrete time) *filtration process* is a stochastic process that describes this information flow. Thus, our system can be viewed as filtration-oriented.

By devising a simple new syntax to describe the relationships between the different linear programs, we eliminate the need for using stage and scenario indexes, enable more modularization, and resolve the ambiguity between the definition of time and stage.

The rest of this paper is organized as follows. In section 2 we introduce mathematical formulations for the multistage stochastic linear programming problem with recourse. We also outline the previous and concurrent work being developed in the modeling of stochastic programming, and contrast our work with it. Section 3 introduces two examples, shows how they are handled in StAMPL, and explains our design choices and their alternatives. Section 4 provides a general view of the architecture of the system, including implementation decisions, their alternatives, and their justifications. Finally, section 5 concludes the paper with some possible extensions, implications, and generalizations of this research.

# 2. Background

From a systems analyst's point of view, stochastic programs can be seen as descriptions of decisions available over time, subject to uncertainty, as in figure 2.1. A framework based on

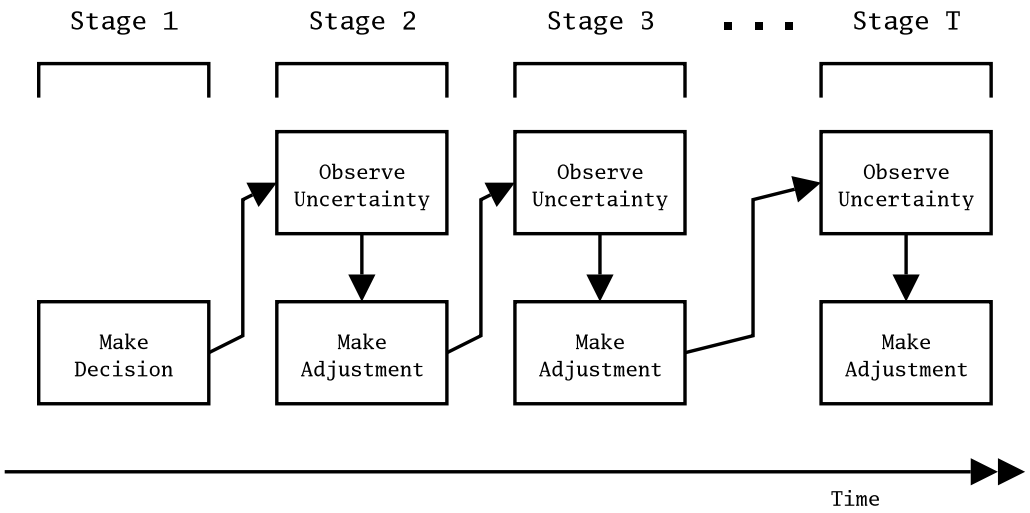


Figure 2.1: A Systems Analyst's view of stochastic programs

this representation is useful in numerous application areas.

Finding solutions to stochastic programs involves the use of sophisticated numerical techniques, which often come from disparate fields of knowledge. Each of these fields has its own view of how a stochastic program should be described, and along with each of those views comes a different notation. Since good modeling environments are available in those fields to cover problems that in some ways are similar to stochastic programs, it is only natural that researchers have attempted to extend existing modeling paradigms to stochastic programming.

Unfortunately, the extension approach can often lead to syntactic quirks or unnecessary complications. Simplicity was the fundamental design principle behind the development of StAMPL. It was clear from studying existing systems and from the literature that progress was possible focusing on three issues: excessive indexing; ambiguity in the representation of time;

and conditional structures. In addition, we wanted each expression written by the modeler to be shorter and involve fewer components; and wanted to find a way to divide models into sequences of simpler models. To accomplish these goals, we investigated alternative ways of organizing the components of a stochastic programming model.

In contrast to viewing stochastic programs as extensions of linear programs, this research proposes an approach to modeling where the objects needed to precisely define the stochastic program are specified in a manner based on the view in figure 2.1, leaving the computer system in charge of the conversion to a representation more useful to a solution method.

In the remainder of this section, we specify the properties of the multistage stochastic linear program independently of any mathematical notation, and then relate its mathematical programming and dynamic programming formulations to those properties. Then, we examine the previous and contemporary work on modeling stochastic programs.

## 2.1 Dynamic Stochastic Programming

We are concerned specifically with **Dynamic (or Multistage) Scenario-based Linear Stochastic Programs with Recours**e **(MSP)**. The goal of these models is to provide decisions that minimize some immediate, known cost plus the expected future cost of the decisions. We start this section by describing some properties of stochastic programming as a modeling framework. We then introduce the mathematical programming and dynamic programming representations of the stochastic program. The order of exposition illustrates that the properties discussed truly belong to the *model*, independently of which solution technique might be used to obtain results.

### 2.1.1 Characteristics of MSP Models

In the **MSP**, time plays a fundamental conceptual role. The modeling framework includes the assumption that there will be discrete, known points in the future where new information will become available. The manner in which this information becomes available is described

by a discrete time filtration process.

The space of time in between the points where new information becomes available is referred to as a **stage**. Decisions are made immediately at the beginning of each stage. Decisions made after the first stage are referred to as **recourse** actions. A stage may or may not coincide with a time period. For example, in a multi-period production problem with uncertain demand, we might need to decide on weekly production schedules, but estimates of new orders may be updated only monthly.

The mechanism most commonly used in stochastic programming to represent filtration processes is the scenario tree. Scenario trees are the only mechanism currently in the StAMPL implementation, but the syntax does not preclude any other representation from being used.

In a scenario framework, all uncertain data are represented by discrete random variables. Any continuous random variables must be discretized, so that a finite number of outcomes is possible. Each possible combination of outcomes from all the random variables at all stages defines a **scenario**. The relationships between different scenarios can be described by means of a **scenario tree**. A three-stage scenario tree with five scenarios can be seen in figure 2.2. Each node of the tree at depth $t$ represents a collection of scenarios which can not be

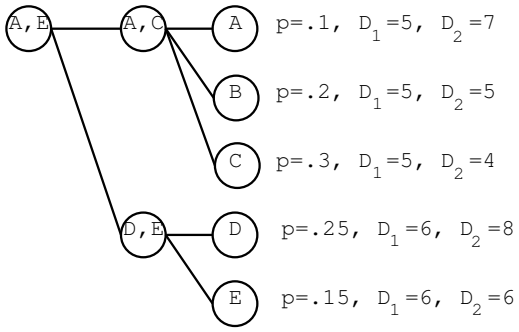


Figure 2.2: A Scenario Tree with three stages and five scenarios. $p$ is the probability of the scenario, and $D_t$ is a random variable representing demand during stage $t$

distinguished from each other at time $t$.

Any decisions made at time $t$ (or earlier) must be the same for all the scenarios represented

by the same node at time $t$. This property is referred to as **Non-anticipativity**. It is this property that makes MSP models so realistic, and at the same time, so difficult to solve.

Within any given stage, it is assumed that decisions can be modeled by a linear program. The entire MSP is *not* a linear program, because of the expected value of future cost term. (The expected value can be shown to be a piecewise linear function of the first stage variables [3].) The MSP *can* and often is rewritten into an equivalent much larger linear program, called a **deterministic equivalent**. Most solution methods actually operate on the deterministic equivalent, although they don't necessarily process the entire problem at once.

### 2.1.2 Mathematical Programming Representation

The MSP may be expressed as a mathematical program as follows:

$$
\begin{array}{lllllll}
\min & c^T x_{s_1} & + & \sum_{s_2 \in \mathcal{S}_{s_1}} p_{s_2}(q_{s_2}^T x_{s_2} & + & \sum_{s_3 \in \mathcal{S}_{s_2}} p_{s_3}(q_{s_3}^T x_{s_3} & + \ldots)) \\
& A_{s_1} x_{s_1} & & & & & & = b_{s_1} \\
& T_{s_2} x_{s_1} & + & W_{s_2} x_{s_2} & & & & = h_{s_2}, \forall s_2 \in \mathcal{S}_{s_1} \\
& & & A_{s_2} x_{s_2} & & & & = b_{s_2}, \forall s_2 \in \mathcal{S}_{s_1} \\
& & & T_{s_3} x_{s_2} & + & W_{s_3} x_{s_3} & = h_{s_3}, \forall s_3 \in \mathcal{S}_{s_2}, & \\
& & & & & & \forall s_2 \in \mathcal{S}_{s_1} & \\
& & & & & A_{s_3} x_{s_3} & = b_{s_3}, \forall s_3 \in \mathcal{S}_{s_2}, & \\
& & & & & & \forall s_2 \in \mathcal{S}_{s_1} & \\
& & & & & \ldots & \ldots & \\
& x_{s_1} \geq 0 & & x_{s_2} \geq 0, \forall s_2 \in \mathcal{S}_{s_1} & & x_{s_3} \geq 0, \forall s_3 \in \mathcal{S}_{s_2}, & & \\
& & & & & \forall s_2 \in \mathcal{S}_{s_1} & &
\end{array}
\tag{2.1}
$$

$\mathcal{S}$ is the scenario set, $s_1$ is the root node, $s_t$ is a node at stage $t$, representing a bundle of scenarios that can not be distinguished up to stage $t$, and $\mathcal{S}_{s_t}$ is the set of children of node $s_t$. $p_{s_t}$ is the *conditional* probability of bundle (or partial scenario) $s_t$. For each $s_t \subset \mathcal{S}$, the $A_{s_t}$ matrix represents intra-stage constraints on the decisions. The $T_{s_t}$ matrix is referred to as the

**technology matrix**. The $W_{s_t}$ matrix is referred to as the **recourse matrix**. Together, the $T_{s_t}$ and $W_{s_t}$ matrices represent the means of communicating information between different stages.

Because there is only one $x_{s_{t-1}}$ variable being connected to $|\mathcal{S}_{s_{t-1}}|$ children by using the $T_{s_t}$ and $W_{s_t}$ matrices, the non-anticipativity property holds. The formulation above is often referred to as the *implicit* formulation, because there are no explicit non-anticipativity constraints.

Another formulation is obtained by writing a separate linear program for each scenario, then adding explicit constraints forcing corresponding variables in related scenarios to take the same values. This formulation allows any special structure present across stages to be preserved, at the cost of more variables and constraints. It is referred to as the *explicit* or *split-variable* formulation:

$$\begin{aligned}
\min \quad & \sum_{s_t \in \mathcal{S}_{t-1}} p^u_{s_t}(c^T x_{1,s_t} \quad + \quad \sum_{t=2}^{T}(q^T_{t,s_t} x_{t,s_t})) & \\
& A_{t,s_t} x_{t,s_t} & = b_t, \forall s_t \in \mathcal{S}_{t-1}, \forall t \\
& T_{t+1,s_t} x_{t,s_t} \quad + \quad W_{t+1,s_t} x_{t+1,s_t} & = h_{t+1,s_t}, \forall s_t \in \mathcal{S}_{t-1}, \forall t < T \\
& x_{t,s_{t_1}} \quad - \quad x_{t,s_{t_2}} & = 0, \forall s_{t_1}, s_{t_2} \in \mathcal{S}_{t-1}, \forall t > 1 \\
& x_{t,s_t} \geq 0 & \forall s_t \in \mathcal{S}_{t-1}
\end{aligned} \tag{2.2}$$

In this formulation, variables are created for each scenario from the first to the last period. $p^u_{s_t}$ is the *unconditional* joint probability of all the outcomes in a scenario taking place. The third constraint set explicitly forces non-anticipativity to hold for related scenarios.

### 2.1.3 Dynamic Programming Representation

Yet another formulation of the **MSP** might be appealing to those with a dynamic programming background. In this context, the function $C(t, s, x)$ returns the expected future cost of the objective function from time $t$ to the last stage $T$ for scenario $s$ given a value of the

decision value $x$. Rewriting the problem description in [4] with the notation in this paper yields:

$$\begin{aligned} C(t, s_t, x_{t-1}) = \quad & min \begin{pmatrix} c_{s_t}^T x_{s_t} + \sum_{s \in \mathcal{S}_{s_t}} C(t+1, s, x_{s_t}) & : t < T \\ c_{s_t}^T x_{s_t} & : t = T \end{pmatrix} \\ s.t. \quad & A_{s_t} x_{s_t} = b_{s_t} \\ & T_{s_t} x_{t-1} + W_{s_t} x_{s_t} = h_{s_t} \\ & x_{s_t} \geq 0 \end{aligned} \tag{2.3}$$

All the above formulations are equivalent. It is possible to generate one formulation from any of the others, as well as from other representations. We set out to find a representation which takes aspects from all of the above formulations, but has advantages over all of them in terms of clarity and intuitiveness, and then invent whatever was needed to leave the computer — not the analyst — in charge of the conversion.

## 2.2 Modeling Tools for Stochastic Programming

Just since the year 2000, at least six distinct research groups have released new or improved versions of their modeling language-based systems, or published research related to them. When examining any of the systems proposed since 1998, when research in modeling languages for stochastic programming accelerated, elements of one or more of the three formulations above are apparent.

Language extensions whose syntax are mostly based on the explicit formulation 2.2 include: SPiNE [26][25], which has extensions for AMPL and GAMS; STOCHASTICS™ [8], which has an AMPL extension; DECIS© [18]; and SISP [5].

One reason for using an approach derived from the explicit formulation is that given a scenario tree — usually furnished separately — the explicit non-anticipativity constraints can be generated where necessary. When examined without the non-anticipativity constraints, formulation 2.2 becomes essentially a multistage linear program. Many of the current ap-

plications of stochastic programming are extensions of multistage linear programs to the stochastic case.

GAMS/OSLSE [13] has a syntax resembling the implicit formulation 2.1. In that system, all variables are indexed over nodes of the scenario tree, and then the relationship between different nodes is provided separately.

One advantage of using an approach derived from the implicit formulation is that once a problem description is available, it is complete. There is no need to add any non-anticipativity constraints, and the formulation can be sent directly to a linear programming solver.

sMAGIC [4], from Buchanan et al, has a syntax much closer to the dynamic formulation 2.3. In that system, a single function is defined as the cost for the decisions in a stage, and all the constraints are encapsulated into the definition of that function. Then, a model is described by calling the first stage model function. The stage $t$ model function then calls the stage $t + 1$ model function (possibly itself, in recursive fashion) with the appropriate parameters.

There are many advantages to using formulation 2.3 as a basis for a modeling language definition. The first is simplicity. This should be apparent from a comparison of the general mathematical formulations 2.1, 2.2, and 2.3 themselves. Less indexing is necessary to distinguish variables in different stages and under different scenarios. In addition, this formulation encourages good systems analysis practices like encapsulation and reuse, especially in the case where different decision models are needed for different periods. Encapsulation is encouraged by allowing smaller, simpler, mostly independent expressions to be written for each stage. Reuse is encouraged because a decision model that reoccurs at some non-consecutive stage needs to be defined only once.

As of the writing of this paper, all of the above systems except DECIS support multistage problems. SPiNE and sMAGIC propose extensions to modeling languages. SPiNE and SISP use an arbitrary index associated with each model component to mark the stage the component belongs to. DECIS and STOCHASTICS use suffixes to store the stage information. sMAGIC

has a completely different approach, in some ways similar to ours. In sMAGIC, stages are specified as distinct problems, and a sequence of stages is specified by referring (sometimes in a recursive fashion) to one problem from the definition of another.

There has also been quite a bit of recent work on modeling languages for stochastic programming which has not yet been implemented in software. Entriken [10], Fourer and Gay[11], Dempster, Medova and Scott[9], and Gassmann and Ireland[16][17], have all contributed ideas and examined the consequences of different design choices.

Extensions to modeling languages are not the only approaches that have been investigated for modeling stochastic programming. In SLP-IOR[19], the modeler works quite closely with the actual mathematical objects that describe an instance of a stochastic program. The environment takes care of managing and interacting with the individual matrices and writing and reading files, while the modeler must only fill in the appropriate entries in each data structure. That system has an approach geared toward the teaching of stochastic programming, which makes some of the design decisions somewhat divergent from those of other systems. In STOCHGEN, which precedes but is now part of the STOCHASTICS system, no extension to a modeling language is proposed. Instead, a representation of the scenario tree, along with an instantiation of one scenario, is used to generate a stochastic program. In [8], there is also a demonstration of one way that pure AMPL can be used to model a stochastic program. Another way to create stochastic programs is by using libraries such as the SP/OSL[20] library.

# 3. Using StAMPL

Since modeling is an essentially human activity, comparing one modeling system to another involves a subjective component of preference. There are, however, aspects of system design for which certain options offer unambiguous advantages over others. A few examples include: a representation that is more terse is preferable to one that is more verbose, provided they are equally clear; providing a simple interface while keeping the implementation of a solution

private (encapsulation); and dividing complex problems into a collection of simpler problems (modularization) are universally accepted characteristics of good modeling.

From an operations research-specific perspective, it is desirable that a modeling systems be able to preserve special structure in problems when it exists, preferably without burdening the modeler.

The above principles, along with two others: addressing the ambiguity between the definition of *time* and *stage* in stochastic programming; and reducing the number of new concepts added to those defined by AMPL, guided the design of StAMPL.

In the remainder of this section, we introduce a couple of well known models, and use them to illustrate some aspects of the design choices in StAMPL. In section 3.1 we introduce a textbook stochastic programming model and some basic notation aspects of our system. In section 3.2 we explore in more detail our stage structure definitions. In section 3.3 we explore the scenario structure definitions. Finally, in section 3.4 we introduce another model with a different structure and explore some issues that arise under those conditions.

## 3.1 A Financial Planning and Control Model

In this problem (from [3, page 20]), an amount of initial wealth is available for investment. Two investment instruments are available, Stocks and Bonds, and their returns are stochastic. The objective is to maximize the expected utility over all scenarios, which is defined in terms of an investment goal at the final stage. Reaching the goal is more important than exceeding it, in typical risk-averse fashion. At each stage, the investor is given the opportunity to re-balance the portfolio.

In the following sections, we will refer to the following small StAMPL code excerpt, which defines the model:

```
1 definestage 1;
2
3 set INSTR;
```

```
4  var Buy{INSTR} >= 0;
5
6  param initial_wealth;
7
8  subject to InvestAll:
9      sum{i in INSTR} Buy[i] = initial_wealth;
10
11 ##########################################################################
12 definestage 2..(stages()-1);
13
14 set INSTR;
15 var Buy{INSTR} >= 0;
16
17 param return{INSTR};
18
19 subject to ReinvestAll:
20     sum{i in INSTR} parent().Buy[i]*return[i] =
21     sum{i in INSTR} Buy[i];
22
23 ##########################################################################
24 definestage stages();
25
26 set INSTR;
27 var Shortage >= 0;
28 var Overage >= 0;
29
30 param shortage_penalty;
```

```
31 param overage_reward;
32     check: shortage_penalty > overage_reward;
33 param return{INSTR};
34 param goal;
35
36 maximize Final_Wealth:
37     overage_reward*Overage - shortage_penalty*Shortage;
38
39 subject to ReinvestAll:
40     sum{i in INSTR} parent().Buy[i]*return[i] +
41     Shortage - Overage = goal;
```

When compared to the same model in [3, page 27], written in AMPL alone, the difference is quite stark. In the StAMPL notation, there are no stage indexes or scenario indexes. Eliminating the indexes for stage and scenario makes the problem significantly more readable and manageable, and separates the algebraic model from the representation of the filtration.

On line 12, the stage declaration command *definestage* is used to indicate which of the boxes of figure 2.1 is being specified. The definition of each stage is encapsulated by the *definestage* declaration.

Also on line 12, the use of the *stages()* function makes it unnecessary to change the model to solve problems with more or fewer stages (For this particular model, there must be at least 3 stages).

Notice the *parent()* function on line 20. This is the mechanism for connecting different stages. Its use makes it easy (for both the modeler and the system) to identify components of the problem, such as recourse and technology matrices. Using the *parent()* function, any element from a previous stage can be accessed.

## 3.2 The Stage Structure

The fundamental new construct that has been added is a *parent()* function relating problems from different stages. This allows us to treat the problems in each state as individual objects, instead of the previous approaches of designating an index or suffix to denote time, or of introducing a recursive syntax. Taking the ***ReinvestAll*** constraint defined above as an example, the alternatives would be as follows: with a syntax based on the implicit formulation 2.1:

```
1 #Implicit Formulation-based Syntax
2 subject to ReinvestAll{s in SCENS diff ROOT diff LEAVES}:
3 sum {i in INSTR} Buy[i,parent(s)]*return[i,s] =
4 sum {i in INSTR} Buy[i,s];
```

The design above has several disadvantages compared to the StAMPL syntax. There are more indices in every variable and parameter definition than in the equivalent StAMPL syntax. Some identifier has to be designated as a special "scenario" indexer, and treated differently from the others. This syntax precludes the use of anything but a scenario tree to represent the filtration process. Finally, it forces the use of scenarios even in the case (as is usual in parts of the development phase) where only one scenario exists.

With the syntax above, there can be no encapsulation of each stage. Therefore, even in this simple example, **if...then...else** constructs are required. They are not visible in the snippet above because they are embedded in the definition of the **ROOT** and **LEAVES** sets. At the cost of creating new keywords, the **ROOT** and **LEAVES** sets can be made automatic. This provision, however, would not eliminate the need to use conditional constructs. More sophisticated models, where decisions may be different at various intermediary stages, would still require them.

With a syntax based on the explicit formulation 2.2, the ***ReinvestAll*** constraint might be written as follows:

```
#Explicit Formulation-based Syntax
subject to ReinvestAll{t in 2..(STAGES-1)}:
sum {i in INSTR} Buy[i,previous(t)]*return[i,t] =
sum {i in INSTR} Buy[i,t];
```

Under this design, the representation of the filtration is no longer limited to scenario trees. But other issues remain. There is still extra indexing. Encapsulation is still difficult to accomplish. The discussion above on conditional constructs still applies. And the need to designate an identifier as a special "time" index to be treated differently from the others is still present. In addition, in this formulation, there could be (in more complex models) confusion between the *stage* index and a *time* index.

With a syntax based on the dynamic programming formulation 2.3, the ***ReinvestAll*** constraint might be written as:

```
#Dynamic Programming Formulation-based Syntax
maximize Utility(ParentBuy):
if stage = STAGES-1 then FinalUtility(Buy);
else Utility(Buy);

subject to ReinvestAll:
sum {i in INSTR} ParentBuy[i]*return[i] =
sum {i in INSTR} Buy[i];
```

Under this design, modularization and encapsulation are enhanced, and extra indexing problems go away. In this example, **if...then...else** constructs are necessary. However, in the case where decisions models differ over different intermediary stages the problem can be more elegantly handled in this framework than in the previous two, by defining different models. The problem of identifying which variables belong to which stage remains, although it can be reduced by following a sane convention, like prepending "Parent" to the name of such vari-

ables, as in the example above. A new problem appears because it is now necessary to define an objective (even if its definition is only $\sum_{s \in \mathcal{S}_{s_t}} C(t+1, s, x_{s_t})$) even when the objective contribution at a given stage is 0. In addition, there is an AMPL-specific problem: Adding parameterized models involves many changes to the AMPL syntax. We would like to avoid making any more changes than necessary.

Our syntax is closer to the dynamic programming formulation than to the other two. It can be thought of as similar to a transformation of the recursion in the dynamic programming formulation to an iteration:

```
1 #StAMPL Syntax
2 subject to ReinvestAll:
3 sum {i in INSTR} Parent().Buy[i]*return[i] =
4 sum {i in INSTR} Buy[i]
```

The *parent()* function is inspired by the "named problems" feature of AMPL. It returns a problem object, corresponding to the linear program defined at the previous stage. Using the '.' operator, any element belonging to the problem referred to by *parent()* can be accessed, with some mild restrictions.

In addition to the *parent()* function, the following functions have been defined:

- *stages()* – returns the total number of stages.
- *setstages(integer)* – normally used in a data file, defines the number of stages.
- *stage()* – returns the current stage, a number in $\{1, \ldots, stages()\}$.

The models for each stage are defined almost independently, whether or not they are in the same file. The *parent()* function is the only mechanism necessary to couple them. This allows completely different problems to be specified for different stages without difficulty, and encourages modular design.

The *definestage integer | interval* command has been introduced to aid in the definition of stages. Typically, the first stage and the last stage in a problem tend to be different from

the intermediate stages and from each other. So a StAMPL model definition will often have three *definestage* declarations, as in the first example in section 3.1.

A minor inconvenience of this approach is that some objectives, constraints, and other parameters may need to be specified more than once. For example, the **INSTR** set is defined once for every problem in the example in this section. There are two alternative ways of handling this issue. The first would be to use a syntax similar to the AMPL named problems feature. In AMPL, objects — say variables like Buy and Sell or constraints like Reinvest1 and Reinvest2 — can be associated with named problems with the following syntax:

```
problem p1: Buy, Sell, Reinvest1;
problem p2: Buy, Sell, Reinvest2;
```

The problem with using this syntax in our context is that all the variables and constraints in AMPL are global objects. We need all of our objects to be stage-specific, and reusing the *problem* statement would redefine the semantics of the statement.

The second alternative would be to create some additional syntax that would allow for certain modules to be defined separately and imported into the definition of certain objects. In this example, if such syntax were available, we could create a module called **"Instruments"** containing the set **INSTR**, the variable **Buy**, and the parameter **return**. This module could then be included into each single-stage model using a statement like **include**. We decided against this, however, because enabling this feature would again have created ambiguities related to the AMPL *problem* statement and its scope model.

## 3.3 The Scenario Structure

As in AMPL, the data for a model is usually specified in a different file. Here is a small excerpt from the accompanying data for this model:

```
9  definestage 2..(stages()-1);
10
```

```
11 set INSTR := STOCKS BONDS;
12 #param return:= STOCKS 1.25 BONDS 1.14;
```

The data file is more interesting because of what it *doesn't* contain than because of what it does. There is no attempt to insert into the algebraic description any stochastic information, or to extend the Modeling Language with stochastic constructs designed to produce a scenario tree.

In previous research, variations on three basic approaches to dealing with scenarios have been attempted: inventing special syntax for the definition of common scenario structures, as implemented in SPiNE [27], and suggested by [17], [11], and others; using existing language data structures indexed over scenarios, as in sMAGIC [4]; or moving the scenario representation outside the modeling language, as in STOCHASTICS [8].

Our approach is more similar to that of STOCHASTICS than to the other two in the sense that the scenario representation is completely separate from the modeling language. At the same time, there are important differences.

STOCHASTICS has a module called *Procgen* used to generate scenario trees. *Procgen* takes two inputs: the topology of a tree; and a function that describes how outcomes already observed affect the distribution of random variables yet to be observed. The output is a scenario tree. The developers of STOCHASTICS chose this approach because they found the *procgen* approach to be less error prone than building a generic tree node by node.

In our system, a C++ library to handle the scenario tree has been designed from scratch. The library is designed for simplicity, covering only trees and random variables. Thus we *do* require the tree to be built node by node. We chose this approach because devising a simple library makes it easy to interface the library with any general-purpose language or (with a little bit more work) with specialized systems like Mathematica, S-plus, and others, which are often used to generate trees and include additional functionality.

The scenario tree is written to a file, and this file is then used by StAMPL to generate an instance. The file can also be made available to any solver which can take advantage of it. It

could also be made available to procedures for stage and scenario manipulation like some of those in [21].

The decision to remove the specification of the scenario tree from the language extension was made in light of earlier research and from the following observations:

- Generating scenario trees is problem-specific. For example, problems in financial engineering need to be concerned with creating trees that are arbitrage-free. Different applications sample from a multitude of different distributions, and sometimes perform complicated calculations for which appropriate libraries in other environments are already available.
- The varied topologies of scenario trees in different applications make any single specification syntax convoluted for other applications. If all different syntaxes are to be added, conflicts may occur. For example, the syntax necessary for conveniently expressing a general balanced tree is different from that needed to conveniently express a tree originating from a trinomial lattice.
- Trees are not usually manipulated in similar ways to algebraic objects. Constructs for manipulating trees thus are more naturally handled in general-purpose languages, which have a richer (and more complex) set of object manipulation mechanisms. Alternatively, user interfaces or domain specific libraries which employ the general purpose constructs can be used.
- Information in the tree may be useful to solution methods, and it is valuable to make it available as a streamable object (typically a file) outside the modeling environment.
- Leaving the scenario specification separate allows extensions to other schemes of representing filtrations to be developed more easily.

We will illustrate the use of the library for the simple asset-liability management model defined earlier. The scenario tree we are representing in this example is the one in [3], which can be seen graphically in figure 3.1.

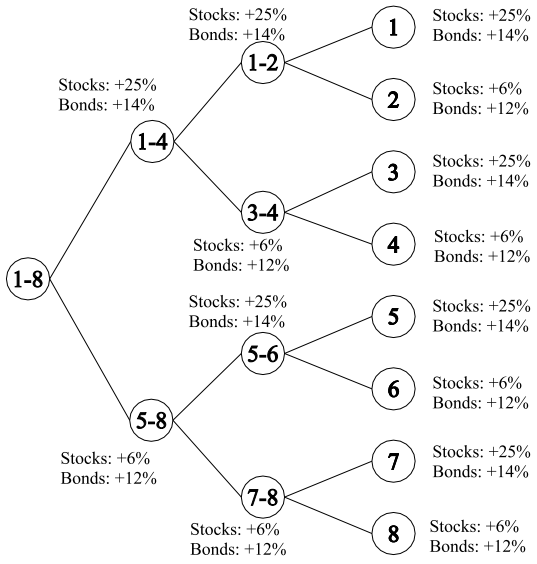


Figure 3.1: Scenario Tree for the Asset-Liability Management Example

In this example, we are constructing a scenario tree from the leaves to the node, one stage at a time. We will highlight here the most relevant sections. The entire listing can be found in appendix A.

The *RandomElem* class is one of the two main classes in the library. *RandomElem* contains all the information necessary to store one stochastic element instantiation, including a label, indexes, and the new value. One can start by declaring some *RandomElem*s, and initializing them with appropriate values:

```
11    RandomElem retStocksGood, retStocksBad;
12    RandomElem retBondsGood, retBondsBad;
13    retStocksGood.param = retStocksBad.param = "return";
14    retBondsGood.param = retBondsBad.param = "return";
15    //push_back adds an element to the end of a list.
16    retStocksGood.index.push_back("STOCKS");
17    retStocksBad.index.push_back("STOCKS");
```

```
18      retBondsGood.index.push_back("BONDS");
19      retBondsBad.index.push_back("BONDS");
20      retStocksGood.val = 1.25; retStocksBad.val = 1.06;
21      retBondsGood.val = 1.14; retBondsBad.val = 1.12;
```

The other main class in the library is *RVNode*. It contains characteristics of scenarios and of nodes of trees. Like scenarios, it has data on path probability, stage, and a list of data elements (RandomElem objects). Like nodes of a tree, it has a list of children, data on conditional probability, and tree-oriented functions like *depth()*. In this example, each node in every non-leaf node is the parent of two other nodes. We declare handles to manipulate the parent and the two children:

```
29      RVNode *parent, *goodChild, *badChild;
```

For each stage except the last, we obtain a list of nodes. For each of those nodes, we initialize a *good* child node and a *bad* child node and push them to the back of each parents' list of children:

```
37              goodChild->stage = badChild->stage = stage+1;
38              goodChild->pathprob = badChild->pathprob = pathprob;
39              goodChild->elems.push_back(retStocksGood);
40              goodChild->elems.push_back(retBondsGood);
41              badChild->elems.push_back(retStocksBad);
42              badChild->elems.push_back(retBondsBad);
43              parent->children.push_back(*goodChild);
44              parent->children.push_back(*badChild);
```

Since we initialize the tree from the leaves to the root, the last parent node handled is the root node. We initialize it, give default labels to each node, and check the probabilities for consistency:

```
52      parent->stage = 1;
53      parent->pathprob = pathprob;
54      parent->NumberScens();
55      parent->ReconcileProbabilities();
```

The completely initialized tree can then be written to a file. Another example, which is more complex and illustrates the advantages of moving the scenario definition outside of the modeling language and into a specialized library, can be found in appendix A.

## 3.4 A Currency Hedging Model with Non-stairstep structure

This model is from the collection in [1], and was originally published in [22]. In this problem, a multinational corporation expects to have significant revenue in a foreign currency, and wishes to protect itself against variations in currency exchange rates. The corporation has the opportunity to purchase options at several points in time that guarantee an exchange rate at the end of the period. The objective is to minimize the dollar amount spent on the options. The constraints are: to guarantee for every possible final real exchange rate scenario, that the corporation will be able to cover its transactions at an acceptable exchange rate; and to not engage in speculative behavior. The real exchange rates are stochastic, and since the prices are functions of those, they are also stochastic. The complete listing can again be found in appendix A.

This model illustrates another use of the *stage()* and *stages()* function calls. They are employed as part of the discounting calculation:

```
13  param toend := stages()-stage();
```

```
21  param price{j in OPTIONS} :=
22      (1-c1[j])*(exp(-us_r*toend)/j) -
23      (1-c2[j])*(exp(-fc_r*toend)/xchange);
```

Another interesting feature of this model is that there are constraints which relate variables belonging to more than two stages. This happens because all the purchases made during the planning period need to be considered in this problem. To accommodate this situation, the *parent()* function can be called with a stage parameter in this case. It is a logic error to call *parent(value)* if *value* is greater or equal to *stage()*.

Two constraints occurring at the last stage use elements from stages earlier than the previous stage. The first constraint forces an acceptable final exchange rate to be guaranteed under all scenarios:

```
66 subject to OptionCost:
67     xchange +
68     sum{t in 1..(stages()-1)}(
69         sum{j in OPTIONS}(
70             parent(t).Buy[j]*(
71                 max(j-xchange,0) -
72                 (1+us_r)^(stages()-t)*parent(t).price[j]
73             )
74         )
75     ) >= acceptable_xchange;
```

The second constraints limits the total amount of options purchased throughout the planning period by the corporation to its liability, normalized to 1. This guarantees that the corporation will not engage in speculative behavior:

```
77 subject to DoNotSpeculate:
78     sum{t in 1..(stages()-1)}
79         sum{j in OPTIONS} parent(t).Buy[j]<=1;
```

Allowing *parent()* to access elements of stages other than the immediate predecessor of a stage introduces significant data management issues. The main complication is that to cor-

rectly support non-stairstep constructs, technology matrices $T_{s,t-k}, \forall s \in \mathcal{S}_{s_{t-1}}, \forall k < t, \forall t \leq T$ have to be maintained and updated. As the number of branches increases, the number of matrices to maintain can become very large.

The relatively complex calculations necessary to compute the option prices illustrate another interesting feature of this model. The full power of AMPL's algebraic capabilities is maintained, including the ability to add new functions, such as the *CumNormal(Reals)* function (lines 47 and 49):

```
37 param xchange;
38 param condexpxchange;
39 param volatility;
40 param us_r;
41 param fc_r := (xchange*(1 + us_r))/condexpxchange - 1;
42 param toend := stages()-stage();
43 param temp1 := us_r - fc_r + (toend)*((volatility^2)/2);
44 param temp2 := us_r - fc_r - (toend)*((volatility^2)/2);
45 param temp3 := volatility*sqrt(toend);
46 param c1{j in OPTIONS} :=
47     CumNormal( (log(j/xchange) + temp1)/temp3 );
48 param c2{j in OPTIONS} :=
49     CumNormal( (log(j/xchange) + temp2)/temp3 );
50 param price{j in OPTIONS} :=
51     (1-c1[j])*(exp(-us_r*toend)/j) -
52     (1-c2[j])*(exp(-fc_r*toend)/xchange);
```

The scenario structure used for this problem in [22] has a trinomial lattice topology like the one in figure 3.2. Before this problem can be written in SMPS format, this lattice needs to be converted into an equivalent scenario tree. An elegant procedure to accomplish this is a relatively simple exercise in a general programming language, but might look convoluted

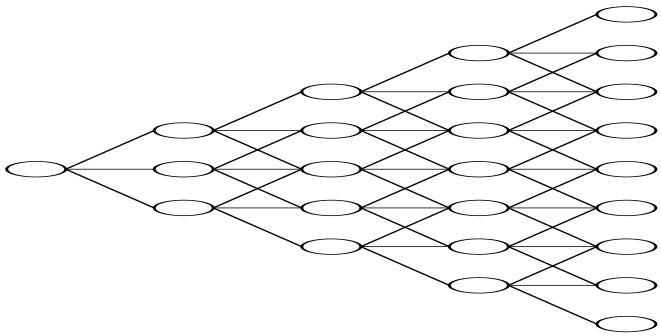

Figure 3.2: A ternary lattice

in an algebraic modeling language, unless the language were to have features for (at least) binary and ternary lattices embedded into it. This provides one more argument to leave the tree specification outside the model. The C++ code for generating the scenario tree for this problem can be found in appendix A.

# 4. Technical Description

Systems for modeling, solving and communicating stochastic programming problems are rapidly evolving. Several approaches for different tasks in modeling of stochastic programming are being experimented with, each with its advantages and disadvantages. In addition, there is research currently being done that could be symbiotically involved with progress made in this research, like our system [12] for adapting the structure of stochastic programs to different solution methodologies. In order to make this system, as well as components of this system, usable with complementary technology, a strong commitment to modularity in design been made from the start.

The system was implemented in a mixture of Python and C++. It uses the asl library[7] for .nl files (AMPL's mechanism for communicating instances to solvers), plus another small library for communicating with AMPL. StAMPL's general architecture can be seen in figure 4.1.

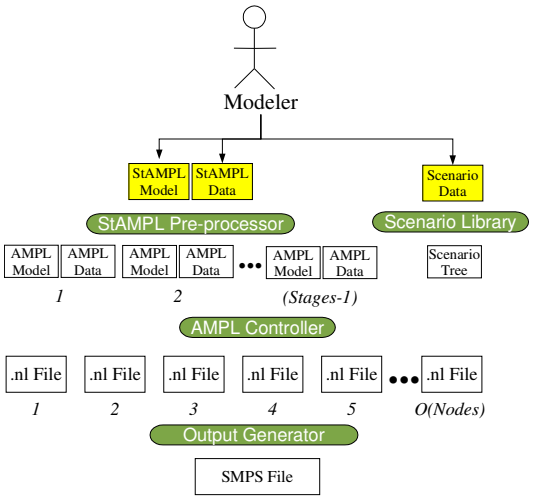


Figure 4.1: The StAMPL system architecture.

The modeler writes StAMPL model and data files like the ones in section 3.1, and provides a scenario tree specification like in section 3.3. A Pre-processor module then divides the model into a collection of AMPL models, one for each pair of stages, with StAMPL's syntax additions reformulated as AMPL constructs. Next, an AMPL controller module takes the generated AMPL models and the scenario data, and instructs the AMPL processor to generate a set of .nl files. Finally, an instance builder takes that collection of .nl files and generates an SMPS file.

In practice, because of efficiency concerns, the implementation is not so clearly defined. For example, we don't need to keep all the .nl files generated. If a tree is balanced with depth $h$ and degree $d$, and each node requires $k$ kilobytes of storage, then keeping all the .nl files would require $kd^h$ kilobytes. Once an .nl file is read, the data contained in it are stored in memory in a far more compact and efficient format and the .nl file can be discarded. Implementing this mechanism reduces the system's modularity, but is necessary as a concession to efficiency.

## 4.1 Generating AMPL models from StAMPL

The StAMPL pre-processor works in a relatively simple way. It operates in two phases.

The first phase pre-processor extracts the number of stages from the data file. Next, it generates one StAMPL problem for each stage, according to the instructions in the original StAMPL model written by the user.

The second phase pre-processor then operates on two single-stage problems written by the first phase preprocessor. It checks that all the references to the previous stage are legitimate, then copies the necessary references from the previous-stage problem into the current-stage problem. It changes any references to *parent()* into a specific label, and prepends that label into each variable from a previous stage. Then it writes a pure AMPL model combining the two stages.

The AMPL language is relatively easy to interpret, at least as far as the additions we have added are concerned. At no point do we need to do any serious syntactic analysis on the input. We implemented a reasonably complete AMPL lexical analyzer. The only syntactic analysis necessary is to make sure that the production immediately around the *parent()* function is correct. For example, we do not wish to allow (at least at this point) for functions to be associated with stages. So a construct like *parent().expense()* or any other reference to a function belonging to the problem returned by *parent()* is illegal.

## 4.2 Using AMPL to generate a stochastic problem

At the end of the preprocessing stage, we end up with a one-scenario description of the problem (albeit in several different files) and a scenario tree. This input is not fundamentally different than that of many other systems. Thus, this translation step is relatively well understood, and is probably not fundamentally different in our work than in other systems. For a nice illustration of the process, see [5]. Still, a couple of important details are worth explaining: the options for how the scenario tree should be transversed; and the mechanisms for storing similarities between scenarios efficiently.

### 4.2.1 Tree traversal

AMPL sees our system as just another solver, and communicates with it through files. One file is written by AMPL for each node of the scenario tree. Because of the amount of file reading and writing, we strongly suggest that a ramdisk (a virtual hard disk which actually resides in memory) be used.

There are two usual ways to traverse a tree: depth-wise, and breadth-wise. They are fundamentally different when it comes to managing trees in which the nodes contain as much data as the ones in this application.

In the depth-wise case, only information about $T-1$ nodes (one for each stage except the last) needs to be maintained at any time. Unfortunately, in this case, once all the children of a given node are processed, it is necessary to reprocess the model for the node's parent in order to process the node's siblings. Thus, in a balanced tree of degree $d$, the depth-wise approach requires $O(d)$ times more computation than the breadth-wise traversal for some operations, since each node of height $> 1$ will be reprocessed $d$ times. Furthermore, this computation implies quite a bit of file reading and writing (by AMPL), in addition to the time necessary for AMPL to rebuild its data structures. Thus, the impact on performance could be considerable.

In the breadth-wise case, each problem needs to be loaded only once per node, and data written only once per node. The disadvantage is that intermediary data for each node in a given stage, plus one reference node, needs to be stored at any time. For the next to last stage, the number of such nodes is $d^{T-1}$.

The comparison of $O(d)$ with $d^{T-1}$ should ordinarily favor the former. But in our case, we chose the latter approach (breadth-wise traversal). The typical numbers associated with problems in the literature, the amount of data that needs to be stored, and the availability of computing resources allowed us this luxury, since the difference in performance is significant. There will be likely be cases where this choice will prove inappropriate, but because of the modular design of the system, replacing the breadth-wise traversal algorithm with a depth-wise traversal algorithm should not be a major undertaking.

### 4.2.2 Finding similarities between scenarios

The internal data structures we chose to implement keep the $T$ and $W$ matrices distinct. This could be useful were a solver to be connected directly to the system in place of the "Output Generator" in figure 4.1, as has been done by [8] and [5].

We have implemented sparse matrix objects with a difference operator and a composition operator. We use these operators to compare corresponding matrices in different scenarios, so that we can store only the elements that differ between matrices. Given matrices $M$, $M'$, and $M''$ the composition operator can be defined as:

$$M \equiv M' \circ M'' \Leftrightarrow m_{ij} = \begin{cases} 0, & m'_{ij} = m''_{ij} = 0 \\ m''_{ij}, & m''_{ij} \neq 0 \\ m'_{ij}, & m'_{ij} \neq 0, m''_{ij} = 0 \end{cases}$$

The difference operator can be defined as:

$$M \equiv M' \setminus M'' \Leftrightarrow m_{ij} = \begin{cases} 0, & m'_{ij} = m''_{ij} \\ m'_{ij}, & m'_{ij} \neq m''_{ij} \end{cases}$$

The operators have the following useful property:

$$M = M' \setminus M'' \Leftrightarrow M' = M'' \circ M$$

Since many of the matrices tend to be identical (for example, often all the $W$ matrices at the same stage will be the same), we have implemented their storage data structures as lists of pointers and implemented smart pointers with reference counting (a simple technique for memory management) to simplify the creation and destruction of the matrices and reduce the memory requirements.

### 4.3 Generating solver-readable output

The output of the previous part of the system is a tree essentially containing the same information as an SMPS file. Typically, for efficiency reasons, this tree is never written to disk, but instead the object containing it is passed in memory to the algorithms in charge of generating output in a form amenable to current solvers. We used MSLiP [14] on NEOS [6] for testing. MSLiP currently does not accept the new version of SMPS files [15] produced by our system, so another small script needed to be written converting the new style SMPS files to the older specification [2].

The details of generating SMPS are relatively straightforward, so we will omit that discussion. The important point about our design when it comes to solver communication is that generating different specifications is relatively easy, since that task is clearly separated from the rest of the computation.

## 5. Conclusions and future Steps

The central conclusion offered by this research is that it is not always the case that modeling environments should try to mirror as closely as possible the mathematical representation of an object. When a simpler representation exists, the onus should be on the modeling environment — instead of on the analyst — to convert the modeler's representation of a problem to a mathematical representation appropriate for existing solution tools. In the process, sometimes special structure that would otherwise be lost can be preserved.

Specifically, putting the stochastic programming-specific concept of a Filtration Process at the center of the modeling syntax has proven to make code more readable, clarified awkward ambiguities with respect to stages and times, and eliminated the need to mix model definitions with special considerations often present in initial and final stage models.

In future research, it would be interesting to try to produce a literate programming [23] tool for optimization modeling. Such a tool would be used not only to produce mathematical

programs, but also to document the business processes associated with the applications.

A further advance to this system which would be very useful would be to extend the syntax a little more to allow different models to be associated with different outcomes at the same stage. This is a case that has largely been ignored by researchers so far, and has the potential to increase the scope of applications amenable to stochastic programming. For example, because of some regulation that only comes into effect under certain conditions dependent on random variables, there could be a completely different model under some set of scenarios $\mathcal{S}_{s_t}$ than under some other set of scenarios $\mathcal{S}_{s'_t}$ at stage $t$.

Another conclusion reached by this research is that the value of an object-oriented view of optimization modeling can be beneficial, confirming the suggestion in [4].

Once a mechanism for encapsulating models is available, in becomes interesting to examine the possibility of developing models or parts of models as objects, creating libraries of such models, and allowing those objects to be combined with each other. Another possibility would be to create a graphical system containing a library of one-stage problems which can be connected graphically, similarly in some aspects to [24]. A system containing some of these features is available from River Logic (`www.riverlogic.com`), but only for a special class of (deterministic) linear programming problems.

Valuable extensions to StAMPL include: a declarative language for generating scenario trees that could be used in place of the C++ library described in section 3.3; a graphical environment in which aggregation and navigation of scenario tree values and other data analysis could be performed; or a port of the system to other modeling languages, such as has been done with [26].

One particularly useful extension is integrating this system with a system for generating different types of decompositions[12]. The availability of a system that provides access to different types of solution methods enhances the value of a good modeling tool.

On a more general note, this research invites the observation that much can still be done to simplify activity of modeling. The simpler we can make modeling, the more success we

will have in increasing the role of optimization technology in decision making in modern organizations.

# A. Examples

## A.1 A Simple Asset-Liability Management model from [3]

### A.1.1 Model File

```
definestage 1;

set INSTR;
var Buy{INSTR} >= 0;

param initial_wealth;

subject to InvestAll:
    sum{i in INSTR} Buy[i] = initial_wealth;

#############################################################
definestage 2..(stages()-1);

set INSTR;
var Buy{INSTR} >= 0;

param return{INSTR};

subject to ReinvestAll:
    sum{i in INSTR} parent().Buy[i]*return[i] =
    sum{i in INSTR} Buy[i];

#############################################################
definestage stages();

set INSTR;
var Shortage >= 0;
var Overage >= 0;

param shortage_penalty;
param overage_reward;
    check: shortage_penalty > overage_reward;
```

```
33 param return{INSTR};
34 param goal;
35
36 maximize Final_Wealth:
37     overage_reward*Overage - shortage_penalty*Shortage;
38
39 subject to ReinvestAll:
40     sum{i in INSTR} parent().Buy[i]*return[i] +
41     Shortage - Overage = goal;
```

### A.1.2 Data File

```
1  setstages(4);
2
3  definestage 1;
4
5  set INSTR := STOCKS BONDS;
6  param initial_wealth := 55;
7
8  ######################################
9  definestage 2..(stages()-1);
10
11 set INSTR := STOCKS BONDS;
12 #param return:= STOCKS 1.25 BONDS 1.14;
13
14
15 ######################################
16 definestage stages();
17
18 set INSTR := STOCKS BONDS;
19 #param return := STOCKS 1.25 BONDS 1.14;
20 param goal := 80;
21 param shortage_penalty := 4;
22 param overage_reward := 1;
```

### A.1.3 Scenario Generation Code

```
1 #include <fstream>
2 #include <iostream>
3 #include <math.h>
4 #include <rvnode/rvnode.H>
```

```
#include <utils/leostl.H>

int main(void){

    const int nStages = 4;

    RandomElem retStocksGood, retStocksBad;
    RandomElem retBondsGood, retBondsBad;
    retStocksGood.param = retStocksBad.param = "return";
    retBondsGood.param = retBondsBad.param = "return";
    //push_back adds an element to the end of a list.
    retStocksGood.index.push_back("STOCKS");
    retStocksBad.index.push_back("STOCKS");
    retBondsGood.index.push_back("BONDS");
    retBondsBad.index.push_back("BONDS");
    retStocksGood.val = 1.25; retStocksBad.val = 1.06;
    retBondsGood.val = 1.14; retBondsBad.val = 1.12;

    int nScens = (int)pow(2,(nStages-1));
    const double pathprob = 1./nScens;

    int listSize = nScens / 2;
    RVNode *children = new RVNode[nScens];
    RVNode *parents;
    RVNode *parent, *goodChild, *badChild;

    for(int stage=nStages-1; stage > 0; stage--){
        parents = new RVNode[listSize];
        for(int i=0;i<listSize;i++){
            parent = parents+i;
            goodChild = children+2*i;
            badChild = goodChild+1;
            goodChild->stage = badChild->stage = stage+1;
            goodChild->pathprob = badChild->pathprob = pathprob;
            goodChild->elems.push_back(retStocksGood);
            goodChild->elems.push_back(retBondsGood);
            badChild->elems.push_back(retStocksBad);
            badChild->elems.push_back(retBondsBad);
            parent->children.push_back(*goodChild);
            parent->children.push_back(*badChild);
        }
        listSize /= 2;
        delete [] children;
        children = parents;
    }

```

```
51      parent = parents;
52      parent->stage = 1;
53      parent->pathprob = pathprob;
54      parent->NumberScens();
55      parent->ReconcileProbabilities();
56      cout << *parent << endl;
57      delete [] parents;
58
59  }
```

## A.2 A Foreign Currency Exchange model from [22]

### A.2.1 Model File

```
1  ###################################################################
2  definestage 1;
3
4  set OPTIONS;
5
6  var Buy{OPTIONS} >= 0;
7
8  param xchange;
9  param condexpxchange;
10 param volatility;
11 param us_r;
12 param fc_r := (xchange*(1 + us_r))/condexpxchange - 1;
13 param toend := stages()-stage();
14 param temp1 := us_r - fc_r + (toend)*((volatility^2)/2);
15 param temp2 := us_r - fc_r - (toend)*((volatility^2)/2);
16 param temp3 := volatility*sqrt(toend);
17 param c1{j in OPTIONS} :=
18     CumNormal( (log(j/xchange) + temp1)/temp3 );
19 param c2{j in OPTIONS} :=
20     CumNormal( (log(j/xchange) + temp2)/temp3 );
21 param price{j in OPTIONS} :=
22     (1-c1[j])*(exp(-us_r*toend)/j) -
23     (1-c2[j])*(exp(-fc_r*toend)/xchange);
24
25 minimize Cost:  sum{j in OPTIONS} Buy[j]*price[j];
26
27 subject to DoNotSpeculate:
28     sum{j in OPTIONS} Buy[j] <= 1;
29
```

```
30 ##################################################################
31 definestage 2..(stages()-1);
32
33 set OPTIONS;
34
35 var Buy{OPTIONS} >= 0;
36
37 param xchange;
38 param condexpxchange;
39 param volatility;
40 param us_r;
41 param fc_r := (xchange*(1 + us_r))/condexpxchange - 1;
42 param toend := stages()-stage();
43 param temp1 := us_r - fc_r + (toend)*((volatility^2)/2);
44 param temp2 := us_r - fc_r - (toend)*((volatility^2)/2);
45 param temp3 := volatility*sqrt(toend);
46 param c1{j in OPTIONS} :=
47     CumNormal( (log(j/xchange) + temp1)/temp3 );
48 param c2{j in OPTIONS} :=
49     CumNormal( (log(j/xchange) + temp2)/temp3 );
50 param price{j in OPTIONS} :=
51     (1-c1[j])*(exp(-us_r*toend)/j) -
52     (1-c2[j])*(exp(-fc_r*toend)/xchange);
53
54 minimize Cost:  (1/1+us_r)^(stage()-1) *
55     sum{j in OPTIONS} Buy[j]*price[j];
56
57 ##################################################################
58 definestage stages();
59
60 set OPTIONS;
61
62 param xchange;
63 param acceptable_xchange;
64 param us_r;
65
66 subject to OptionCost:
67     xchange +
68     sum{t in 1..(stages()-1)}(
69         sum{j in OPTIONS}(
70             parent(t).Buy[j]*(
71                 max(j-xchange,0) -
72                 (1+us_r)^(stages()-t)*parent(t).price[j]
73             )
74         )
75     ) >= acceptable_xchange;
```

```
76
77 subject to DoNotSpeculate:
78     sum{t in 1..(stages()-1)}
79         sum{j in OPTIONS} parent(t).Buy[j]<=1;
```

### A.2.2 Data File

```
1  setstages(5);
2
3  definestage 1..(stages()-1);
4
5  set OPTIONS := 0.44 0.50 0.57 0.63 0.70 0.76 0.83 0.89 0.96 1.02;
6
7  param xchange := .56;
8  param condexpxchange := .52;
9  param volatility := .11;
10 param us_r := .1;
11
12 definestage stages();
13
14 set OPTIONS := 0.44 0.50 0.57 0.63 0.70 0.76 0.83 0.89 0.96 1.02;
15
16 param xchange := .37;
17 param acceptable_xchange := .407;
18 param us_r := .1;
```

### A.2.3 Example Scenario Tree generation code

```
1  #include <fstream>
2  #include <iostream>
3  #include <math.h>
4  #include <rvnode/rvnode.H>
5  #include <utils/leostl.H>
6
7  int Pascal(int idx,int depth){
8      if (depth == idx == 1) return 1;
9      else if (idx < 1 || idx > 2*(depth-1)+1) return 0;
10     else return Pascal(idx-2,depth-1)+
11         Pascal(idx-1,depth-1)+ Pascal(idx,depth-1);
12 }
13
```

```
14 int main(void){
15     const int stages = 5;
16     const int arity = 3;
17     const double accxchg[9] =
18         {.407,.416,.423,.429,.444,.466,.494,.527,.564};
19     const double xchg[25] = {
20         .56,
21         .47,.52,.57,
22         .43,.46,.49,.55,.61,
23         .39,.42,.44,.49,.52,.58,.65,
24         .37,.39,.41,.43,.46,.50,.55,.61,.68
25     };
26     RandomElem rexchg;
27     rexchg.param = "xchange";
28     RandomElem reaccxchg;
29     reaccxchg.param = "acceptable_xchange";
30
31     RVNode *children = new RVNode[(stages-1)*2+1];
32     RVNode *rn;
33     const double pathprob = 1./pow(arity,stages-1);
34     for(int stage=stages-1;stage>0;stage--){
35         const int card = stage*2+1, zeroidx = stage*stage;
36         rn = children;
37         const double condprob = 1./pow(arity,stage);
38         for(int i=0;i<card;i++){
39             rexchg.val = xchg[zeroidx+i];
40             rn->prob = condprob;
41             rn->pathprob = pathprob;
42             rn->stage = stage+1;
43             rn->elems.push_back(rexchg);
44             if(stage == stages-1){
45                 reaccxchg.val = accxchg[i];
46                 rn->elems.push_back(reaccxchg);
47             }
48             rn++;
49         }
50         RVNode *parents = new RVNode[card-2];
51         rn = parents;
52         RandomElem cex;
53         cex.param = "condexpxchange";
54         for(int i=0;i<card-2;i++){
55             cex.val = 0;
56             for(int j=0;j<arity;j++){
57                 cex.val += xchg[zeroidx+i+j]/arity;
58                 rn->children.push_back(children[i+j]);
59             }
```

```
60              rn->elems.push_back(cex);
61              rn++;
62          }
63          delete [] children;
64          children = parents;
65      }
66
67      rn--;
68      rn->stage = 1;
69      rn->prob = 1;
70      rn->pathprob = 1./81;
71      rn->NumberScens();
72
73      cout << *rn << endl;
74      delete [] rn;
75
76  }
```

# References


[1] K. A. Ariyawansa and A. J. Felt. On a new collection of stochastic linear programming test problems. Technical Report 4, Department of Mathematics, Washington State University, Pullman, WA 99164, Apr. 2001.

[2] J. Birge, M. Dempster, H. Gassmann, E. Gunn, A. King, and S. Wallace. A standard input format for multiperiod stochastic linear programs. *COAL Newsletter*, 17:1–19, 1987.

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

[4] C. S. Buchanan, K. I. M. McKinnon, and G. K. Skondras. The Recursive Definition of Stochastic Linear Programming Problems within an Algebraic Modeling Language. *Annals of Operations Research*, 104:15–32, Apr. 2001.

[5] C. Condevaux-Lanloy, E. Fragniere, and A. King. SISP, Simplified Interface for Stochastic Programming: Establishing a hard link between mathematical programming modeling

languages and SMPS codes. *Optimization Methods and Software*, 17(3):423 – 443, 2002. Special Issue on Stochastic Programming. Guest Editors: A. Prekopa and A. Ruszczynski.

[6] J. Czyzyk, M. P. Mesnier, and J. J. Moré. The NEOS Server. *IEEE Computational Science and Engineering*, 5(3):68–75, July 1998.

[7] David M. Gay. Hooking your solver to AMPL. Technical report, Bell Laboratories, Murray Hill, NJ, 18 Apr. 1997.

[8] M. Dempster, J. Scott, and G. Thompson. Stochastic Modelling and Optimization using Stochastics. Technical report, Judge Institute of Management Studies, 2002.

[9] M. A. H. Dempster, E. A. Medova, and J. E. Scott. A modelling system for stochastic programming. In *OR41 Edinburgh*, September 1999.

[10] R. Entriken. Language constructs for modeling stochastic linear programs. Technical report, Systems Optimization Laboratory - Stanford University, 1997.

[11] R. Fourer and D. Gay. Stochastic programming in the ampl modeling language. In *Intl. Symposium on Mathematical Programming*, August 1997.

[12] R. Fourer and L. Lopes. A management system for decompositions in stochastic programming. Under Review.

[13] GAMS optimization. *OSL Stochastic Extensions*. http://www.gams.com/solvers/oslse.pdf.

[14] H. Gassmann. MSLiP: A computer code for the multistage stochastic linear programming problem. *Mathematical Programming*, 47:407–423, 1990.

[15] H. Gassmann and E. Schweitzer. A comprehensive input format for stochastic linear programs. Working paper, School of Business Administration, Dalhousie Uinversity, Halifax, Canada, 1996.

[16] H. I. Gassmann and A. M. Ireland. Scenario formulation in an algebraic modelling language. *Annals of Operations Research*, 59:45–75, 1995.

[17] H. I. Gassmann and A. M. Ireland. On the formulation of stochastic linear programs using algebraic modelling languages. *Annals of Operations Research*, 64:83–112, 1996.

[18] G. Infanger. GAMS/DECIS User's Guide. http://www1.gams.com/docs/solver/decis.pdf, 1999.

[19] P. Kall and J. Mayer. Slp-ior: An interactive model management system for stochastic linear programs. *Mathematical Programming*, 75:221–240, 1996.

[20] A. King. *SP/OSL V1.0, Stochastic Programming Interface Library, User's Guide.*

[21] P. Klaassen. Financial asset-pricing theory and stochastic programming models for asset/liability management: A synthesis. *Management Science*, 44:31–48, 1998.

[22] P. Klaassen, J. F. Shaprio, and D. E. Spitz. Sequential decision models for selecting currency options. Technical Report 133-90, MIT, International Financial Services Research Center, July 1990.

[23] D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[24] P. Ma, F. H. Murphy, and E. A. Stohr. An Implementation of LPFORM. *INFORMS Journal on Computing*, Sept. 1996.

[25] E. Messina and G. Mitra. Modelling and analysis of multistage stochastic programming problems: A software environment. *European Journal of Operational Research*, 101:343–359, 1997.

[26] P. Valente, G. Mitra, and E. F. D. Ellison. Extending algebraic modelling languages for stochastic programming. working paper, Brunel University 9/2000, September 2000.

[27] P. Valente, G. Mitra, C. Poojari, and T. Kyriakis. Software Tools for Stochastic Programming: A Stochastic Programming Integrated Environment (SPInE). Technical report, Brunel University, Uxbridge, UK UB8 3PH, Aug. 2001.