`SDDP.jl`: a Julia package for Stochastic Dual Dynamic Programming

Oscar Dowson[a,*]

[a]*Department of Engineering Science at The University of Auckland, 70 Symonds Street, Auckland 1010, New Zealand*

Abstract

`SDDP.jl` is an open-source library for solving computational stochastic optimization problems using Stochastic Dual Dynamic Programming. `SDDP.jl` is built upon JuMP, an algebraic modelling language in Julia. This enables a high-level interface for the user while simultaneously providing performance that is similar to implementations in lower-level languages. In this work we present `SDDP.jl`, justify implementation decisions, and demonstrate the capabilities through an example from the literature.

Keywords: SDDP, multistage, computational stochastic optimization, stochastic programming, library

1. Introduction

Solving any mathematical optimization problem requires four steps: the formulation of the problem by the user; the communication of the problem to the computer; the efficient computational solution of the problem; and the communication of the computational solution back to the user. Over time, considerable effort has been made to improve each of these four steps for a variety of problem classes such linear, quadratic, mixed-integer, conic, and non-linear (consider the evolution from early file-formats such as MPS [1] to modern algebraic modelling languages embedded in high-level languages such as JuMP [2], or the 73 fold speed-up in solving difficult Mixed-Integer Linear Programs in seven years by Gurobi [3]).

However, the same cannot be said for stochastic problem classes. This is particularly true of convex, multistage, stochastic optimization problems (which are to be the focus of this paper). There is even considerable debate [4, 5] about how best to even formulate a stochastic program. Moreover, when it comes to communicating the problem to the computer, some formats have been proposed [6, 7], but owing to the lack of agreement about the problem formulation, acceptance of these is not widespread. Instead, what happens is the development of an *ad-hoc*, problem-specific format that is often tightly coupled to individual implementations on a case-by-case basis. The visualization of stochastic policies is also difficult due to the high-dimensionality of the state and action spaces, and the inherent uncertainty. As such, policy visualization is also problem-specific, and in our opinion, an often neglected part of the solution process.

*Corresponding author
Email address: `odow003@aucklanduni.ac.nz` (Oscar Dowson)

Where progress has been made however, is in the development of efficient computational solution algorithms. The state-of-the-art solution technique for convex multistage stochastic optimization problems, Stochastic Dual Dynamic Programming (SDDP), was introduced in the seminal work of [8]. Since that time, SDDP (and its variants) have been widely used to solve a number of problems in both academia and industry. However, until recently, no open-source, flexible implementations of the algorithm existed in the public domain[1]. Instead, practitioners were forced to code their own implementations in a variety of languages and styles. Research implementations have been reported in a variety of languages including AMPL [9], C++ [10, 11], GAMS [12, 13], JAVA [14] and MATLAB [15], as well as in commercial products such as the seminal SDDP [16] and DOASA [17]. In our opinion, this "re-invention of the wheel" has limited the adoption of the SDDP algorithm in areas outside of the electricity industry (which is the focus of most researchers) as it poses a large up-front cost to development. Furthermore, not all implementations are state-of-the-art as there is a high cost of maintenance. As such, many researchers develop and test new algorithmic improvements without being able to easily compare their ideas to the current state-of-the-art.

In developing `SDDP.jl`, it was our intention to provide a modern solution to all four steps of the mathematical optimization process for multistage convex stochastic optimization problems. In Section 2 we describe the standardized form for describing multistage stochastic optimization problems of the type that can be solved by SDDP. In Section 3, we describe the user-interface for `SDDP.jl` that was designed to closely match the mathematical description of the problem by building on-top-of `JuMP.jl` [18]. In Section 4, we discuss some of the improvements that have been made to the SDDP algorithm since its inception that have been incorporated into `SDDP.jl`, and how it has been designed to be extensible in order to allow new improvements to be added to the codebase with minimal effort. Lastly, we describe in Section 5, the interactive JavaScript visualization tool in `SDDP.jl` that allows the user to quickly and easily visualize high-dimensional solutions to stochastic optimization problems. As a central thread running through the paper, we use a variant on the Asset Management Problem from [19] to illustrate the features of `SDDP.jl` we describe in each section.

It is not the intention of the author to make this paper a comprehensive tutorial for Julia or SDDP. In places, familiarity is assumed of both the SDDP algorithm, Julia, and JuMP. Readers are directed to the project website at `github.com/odow/SDDP.jl` for more detailed documentation, examples, and source code.

2. Formulating the Problem

This paper is concerned with multistage, convex, stochastic optimization problems in a *Hazard-Decision* (also called *Wait-and-See*) framework. We shall use the notation of stochastic optimal control, where x_t represents state variables and u_t represents control variables in stage t. The goal of the optimization is to find a policy π_t that maps an incoming state variable $\bar{x}_t$, along with the observation of the random variable $\omega_t \sim \Omega_t$ to a feasible control u_{t,ω_t} for each stage t such that it minimizes the immediate cost, plus the risk-adjusted expected future cost. This can be expressed as the following optimization problem:

[1]There are now at least four: `https://github.com/JuliaOpt/StochDynamicProgramming.jl`, `https://github.com/blegat/StructDualDynProg.jl`, `https://web.stanford.edu/~lcambier/fast/`, and `https://github.com/odow/SDDP.jl`.

$$\min_{\pi} \underset{i \in 0^+;\ \omega_i \in \Omega_i}{\mathbb{F}_0} \left[V_i^{\pi}(x_0, \omega_i) \mid x_0 \right], \tag{1}$$

where

$$V_i^{\pi}(\bar{x}, \omega) = C_i(\bar{x}, u_{i,\omega}, \omega) + \underset{j \in i^+,\ \omega_j \in \Omega_j}{\mathbb{F}_{\mathrm{i}}} \left[V_j^{\pi}(x_{i,\omega}, \omega_j) \right], \tag{2}$$

$$x_{i,\omega} = T_i(\bar{x}, u_{i,\omega}, \omega), \tag{3}$$

and

$$u_{i,\omega} = \pi_i(\bar{x}, \omega). \tag{4}$$

The policy $\pi_i(x, \omega)$ is a function that maps the incoming state of the system $\bar{x}$, and the observation of the noise ω into a control $u_{i,\omega}$. The policy respects any constraints on the domain of the control so that $u_{i,\omega} \in \mathcal{U}_i(x, \omega)$ and $x_{i\omega} \in \mathcal{X}_i(x, \omega)$. Stages form the vertices in a directed graph $\mathcal{G}$. We say that stage j is a child of stage i if there is a directed edge from stage i to stage j. We define the set of child nodes of stage i as i^+. There is a known, finite probability of transitioning from stage i to every stage in i^+. $\mathbb{F}$ is a risk measure over the transition probabilities to each child stage j, and observations of the noise ω_j within each child stage. We also assume relatively complete recourse of $\bar{x}$. That is, for all $\omega \in \Omega_i$, there is a feasible control $u_{i,\omega}$, with a bounded objective value $V_i^{\pi}(\bar{x}, \omega)$.

In the stochastic optimization community, policies typically take the form of an optimization problem. Therefore, $V_i^{\pi}(\bar{x}, \omega)$ is the objective of the optimization problem:

$$\begin{array}{rll}
\mathbf{P}_i(\bar{x}, \omega): & \min\limits_{u_{i,\omega}} & C_i(\bar{x}, u_{i,\omega}, \omega) + \underset{j \in i^+,\ \omega_j \in \Omega_j}{\mathbb{F}_{\mathrm{i}}} \left[V_j^{\pi}(x_{i,\omega}, \omega_j) \right] \\
& s.t. & x_{i,\omega} = T_i(\bar{x}, u_{i,\omega}, \omega) \\
& & u_{i,\omega} \in \mathcal{U}_i(x, \omega) \\
& & x_{i,\omega} \in \mathcal{X}_i(x, \omega),
\end{array} \tag{5}$$

where the policy $\pi_i(\bar{x}, \omega) \in \underset{u_{i,\omega}}{\arg\min}\, \mathbf{P}_i(\bar{x}, \omega)$.

`SDDP.jl` is able to solve the sub-set of these problems that satisfy the following requirements:

1. $\mathcal{X}$ and $\mathcal{U}$ are polyhedral sets;
2. C is linear for a given ω;
3. T is a linear functions of its arguments;
4. $\mathbb{F}_0$ is the Expectation risk measure;
5. $\mathbb{F}_{\mathrm{i}}$ is a convex combination of the expectation, and Average Value @ Risk (AV@R) risk measures;
6. Ω is a discrete distribution with a finite number of outcomes;
7. The stage graph forms an acyclic Markov chain (in the vein of [20]) with a finite number of Markov states in each stage.

Requirements (1), (2), and (3) are another way of saying that the problem $\mathbf{P}_i(\bar{x}, \omega)$ can be written as a linear program with varying objective and right-hand-side coefficients, but not constraint coefficients. We note that these requirements are more restrictive than necessary, and that variations of the SDDP algorithm have been proven to converge for more general classes of problems (for example, with convex non-linear programs [21], or when $\mathcal{X} = \{0, 1\}$ [22]). However, these requirements permit the modelling of most real-world applications.

We now formulate a variation on the Asset Management Problem from [19] in order to illustrate these requirements.

2.1. Asset Management Example Formulation

The goal of the asset management problem is to choose an investment portfolio that is composed of stocks and bonds in order to meet a target wealth goal at the end of the time horizon. After five, and ten years, the agent observes the portfolio and is able to re-balance their wealth between the two asset classes. As an extension to the original problem, we introduce two new random variables. The first that represents a source of additional wealth in years 5 and 10. The second is an immediate reward that the agent incurs for holding stocks at the end of years 5 and 10. This can be though of as a dividend that cannot be reinvested.

Stage Graph. There are four sequential stages: `Today`; `Year 5`; `Year 10`; and `Horizon`. `Today` is a deterministic Decision-Only stage. The next three stages are Hazard-Decision problems. In Figure 1, we provide a graphical representation of this staging. Each square node represents a stage. Straight arrows are connections between stages. Wavy lines indicated exogenous random information is revealed at the start of the stage. The shaded circle is the root node at which we evaluate the risk-adjusted expectation of Eq. 1.

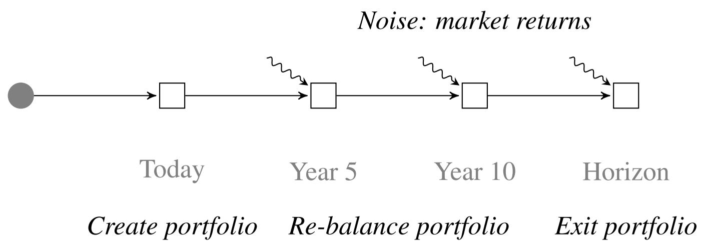

Figure 1: Asset Management Problem: stage graph

We now give the stage problem formulations (Eq. 5) for each of the stages using the notation of linear programming. We exclude the future cost component of the objective as this is handled separately by `SDDP.jl`.

Today (Decision-Only).

$$\begin{aligned} \mathrm{P}_{\text{today}}\left([\bar{x}^s, \bar{x}^b]\right): \quad & \min \quad 0 \\ & \text{s.t.} \quad x^s + x^b = 55 + \bar{x}^s + \bar{x}^b \\ & \qquad\ \ x^s, x^b \geq 0, \end{aligned} \tag{6}$$

where $\bar{x}^s$ is the value of stocks (in dollars) held at the start of the stage, x^s is the value of stocks (in dollars) held at the end of the stage, $\bar{x}^b$ is the value of bonds (in dollars) held at the start of the stage, and x^b is the value of bonds (in dollars) held at the end of the stage. The control variables (value of stocks and bonds to buy) are implicit in this formulation.

The first stage has the initial solution $(\bar{x}^s, \bar{x}^b) = (0, 0)$.

Year 5 and Year 10 (Hazard-Decision).

$$\text{P}_{\text{re-balance}}\left([\bar{x}^s, \bar{x}^b], [\omega^s, \omega^b, \varphi, \psi]\right): \quad \begin{array}{ll} \min & -\psi \times x^s \\ \text{s.t.} & \omega^s \bar{x}^s + \omega^b \bar{x}^b + \varphi = x^s + x^b \\ & x^s, x^b \geq 0, \end{array} \tag{7}$$

where ω^s is the increase in the value of the stocks during the stage, and ω^b is the increase in the value of the bonds during the stage. The control variable (value of stocks to exchange for bonds) is implicit in this formulation. φ is an additional injection of cash (in dollars) that may represent the sale of other assets not modelled in the optimization problem. ψ is some fraction of the stocks value that is paid out as a dividend and is not able to be reinvested.

The two random variables φ and ψ are perfectly correlated[2] so that:

$$(\varphi, \psi) = \begin{cases} (5, 0) & w.p.\ 0.4, \\ (-1, 0.02) & w.p.\ 0.6. \end{cases} \tag{8}$$

Horizon (Hazard-Decision).

$$\text{P}_{\text{horizon}}\left([\bar{x}^s, \bar{x}^b], [\omega^s, \omega^b]\right): \quad \begin{array}{ll} \min & 4u - v \\ \text{s.t.} & \omega^s \bar{x}^s + \omega^b \bar{x}^b + u - v = 80 \\ & u, v \geq 0, \end{array} \tag{9}$$

where u is the quantity by which the portfolio is less than the target (\$80), and v is the quantity by which the portfolio exceeds the target.

Exogenous Information. Stages Year 5, Year 10, and Horizon have the following probability set for ω of:

$$(\omega^s, \omega^b) = \begin{cases} (1.25, 1.14) & w.p.\ 0.5 \quad \text{(High Return)}, \\ (1.06, 1.12) & w.p.\ 0.5 \quad \text{(Low Return)}. \end{cases} \tag{10}$$

Reformulating to meet `SDDP.jl` criteria. In the formulation above, the random variable ω occurs in the coefficients of the constraints. However, one requirement of `SDDP.jl` is that the transition function is linear in its arguments. Therefore, we model the observations of ω by the Markov chain in Figure 2.

In this formulation, ω^s and ω^b can be replaced by constants in the subproblems, with the uncertainty now lying on the arcs exiting the stages in the stage graph. For each stage, we need to provide the transition probabilities of arriving from the Markov states in the previous stage. For simplicity, we shall now refer to the stages `Today`, `Year 5`, `Year 10`, and `Horizon` as stages 1,2,3, and 4 respectively. In addition, we shall refer to the `Low Return` and `High Return` Markov states as 1 and 2 respectively. By convention, we say that the root node, and any subproblem in a stage with one Markov state is Markov state 1. Therefore, for stage 1 (`Today`), the probability transition matrix of transitioning from the root node (Markov state 1) to the only Markov state in the stage is

[2]We note that this is not a limiting assumption as it is possible to construct the Cartesian product of the two random variables.

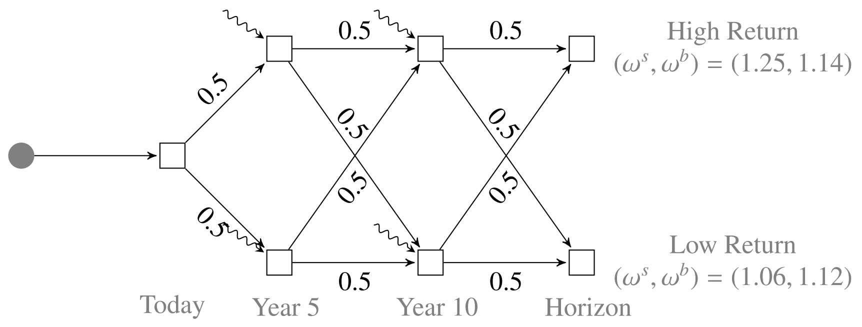

Figure 2: Asset Management Problem: revised stage graph

$$P_1 = \begin{bmatrix} 1.0 \end{bmatrix}. \tag{11}$$

In the second stage (`Year 5`), the probability of transitioning from Markov state 1 in the previous stage, to each Markov state is 0.5. Therefore the transition matrix is:

$$P_2 = \begin{bmatrix} 0.5 & 0.5 \end{bmatrix}. \tag{12}$$

In the final two stages, it is possible to arrive from either Markov state 1 or 2. Therefore, the transition matrix is:

$$P_t = \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix}, \quad t = 3, 4. \tag{13}$$

Risk Measures. Another piece of information we need to describe is the risk measure ($\mathbb{F}_i$) for each stage. We shall use a convex combination of the Expectation, and Average Value at Risk measures as proposed by [23] and [20]:

$$\mathbb{F}[X] = \lambda\,\mathbb{E}[X] + (1-\lambda)\text{AVaR}_{1-\beta}[X]. \tag{14}$$

As the values of λ and β increase, the measure becomes less risk-averse[3].

To introduce risk into the Asset Management problem, we choose to make the risk measure associated with stages `today`, `Year 5`, and `Horizon` the Expectation operator. However, in `Year 10`, we use Eq. (14) with the values $\lambda = 0.5$ and $\beta = 0.5$.

Lower bound. Lastly, we need to give a bound on the value of Eq. 1. Although not necessary from a theoretical point of view, providing such a bound simplifies the practical implementation of the algorithm. If the user is unsure of a bound, we recommend choosing some large negative (if minimizing) value and running the SDDP algorithm for a number of iterations. If the problem converges to a value close to the bound set by the user, we suggest increasing the initial bound by an order of magnitude and resolving. The best-case outcome of the asset management problem would be to end assets worth much greater than \$80. To be safe, we assume a bound of -\$1000 which is well in excess of the potential profit over the time horizon.

We have now described all the information necessary to implement the model in `SDDP.jl`. In the next section, we introduce the user interface of `SDDP.jl`, and implement the Asset Management example.

[3]This differs from other authors (i.e. [20]) who prefer the $(1-\lambda)$ on the expectation operator.

3. Communicating the Problem to the Solver

In the previous section, we formulated the Asset Management example as a multistage linear stochastic optimization problem. This was done by decomposing the problem into a collection of sub-problems that could be described as linear programs, along with some meta information (i.e.e the transition matrices) about how the stages were inter-related. This design decision greatly simplifies the challenge of communicating the stochastic optimization problem to the solver as we can leverage other work on communicating linear subproblems. For that, we turn to JuMP.

JuMP [2] is a algebraic modelling language for mathematical optimization in the Julia programming language [24]. JuMP supports a wide range of problem classes including linear, mixed-integer, quadratic, conic-quadratic and non-linear. In particular, a large effort has been placed on abstracting multiple steps in the typical optimization modelling process in a way that is open to extension by third parties. This has allowed us to create a SDDP modelling library that builds on the functionality of both JuMP and Julia. The expressiveness of JuMP's modelling syntax is available to the user with minimal implementation effort, while Julia's multiple dispatch, parallelism and macro-programming features enable a performant yet flexible implementation of the SDDP algorithm.

We now describe many features of the user interface of `SDDP.jl` by walking through the Asset Management example. As a note to the reader, lines of code are given as verbatim text. Commands to be entered into the Julia REPL are prefixed with `julia>`. Lines of code for the Asset Management example are enclosed in a framed box. The entire text file can be run by calling:

```
julia> include("path/to/text/file")
```

To begin, we need to install the latest version of `SDDP.jl` (this requires Julia 0.5 or later):

```
julia> Pkg.clone("https://github.com/odow/SDDP.jl")
```

If not already installed, this will also install JuMP and the COIN-OR solver Clp. At the top of our model file, we load the relevant packages[4]:

```
using JuMP, SDDP, Clp
```

3.1. Initializing the model

The most important constructor in the `SDDP.jl` library is the following:

```
m = SDDPModel(
    # ... keyword arguments will go here ...
                ) do sp, t, i
    # ... sub-problem definition will go here ...
end
```

[4]Users are free to use any of the other solvers in the JuliaOpt ecosystem that support MathProgBase instead of Clp.

This command constructs a new `SDDPModel` object and attaches it to the variable `m`. Of note to the reader, we highlight the `SDDPModel() do ... end` syntax with three arguments after the `do` keyword. The first is a pointer to a new JuMP model for each subproblem. This can be named anything by the user, but we recommend standardizing upon `sp`. The second is an integer counter that will take the values 1, 2, ... T, where T is the number of stages in the model. The third is an integer counter that will take the values 1, 2, ..., M, where M is the number of Markov states in stage t. These three arguments can be arbitrarily named by the user, by we prefer `sp`, `t`, and `i` by convention.

Inside the `SDDPModel()` constructor, we must provide a number of keyword arguments. These are:

1. `sense`: must be `:Max` or `:Min` (defaults to `:Min`);
2. `stages`: an `Int` that describes the number of stages;
3. `solver`: a MathProgBase compatible solver (i.e. `ClpSolver()`);
4. `objective_bound`: a bound on the optimal objective of the first stage problem;
5. `markov_transition`: a vector of Markov transition matrices with one entry for each stage. If this is not specified, it is assumed that there is only one Markov state in each stage;
6. `risk_measure`: an `AbstractRiskMeasure` defined by the `SDDP.jl` library. If this is not specified it defaults to the expectation operator (`Expectation()`).

Risk Measures. Two risk measures are currently implemented in the `SDDP.jl` library: Expectation, and a convex combination of Expectation and Average Value @ Risk (Ex. 14) using the "change of probabilities" approach of [25]. The Average Value @ Risk risk measure can be constructed with the constructor:

```
julia> NestedAVaR(;lambda=1.0, beta=1.0)
```

The expectation operator is simply `Expectation()`. Risk measures for a stage are set using the `risk_measure` keyword in the `SDDPModel` constructor. If a single risk measure is given (i.e. `risk_measure = Expectation()`), then this measure will be applied to every stage in the problem. Another option is to provide a vector of risk measures. There must be one element for every stage. For example:

```
risk_measure = [ NestedAVaR(lambda=0.5, beta=0.25), Expectation() ]
```

This will apply the i^{th} element of `risk_measure` to every Markov state in the i^{th} stage. The last option is to provide a vector (one element for each stage) of vectors of risk measures (one for each Markov state in the stage). For example:

```
risk_measure = [
 # Stage 1 Markov 1 # Stage 1 Markov 2 #
    [ Expectation(), Expectation() ],
    # ------- Stage 2 Markov 1 ------- ## ------- Stage 2 Markov 2 ------- #
    [ NestedAVaR(lambda=0.5, beta=0.25), NestedAVaR(lambda=0.25, beta=0.3) ]
]
```

Note that even though the last stage does not have a future cost function associated with it (as it has no children), we still have to specify a risk measure. This is necessary to simplify the implementation of the algorithm.

We can now update our model file with the data for the asset management example:

```
m = SDDPModel(
              sense = :Min,
             stages = 4,
             solver = ClpSolver(),
    objective_bound = -1000,
  markov_transition = [
                [1.0],
                [0.5 0.5],
                [0.5 0.5; 0.5 0.5],
                [0.5 0.5; 0.5 0.5]
            ],
       risk_measure = [
            Expectation(),
            Expectation(),
            NestedAVaR(lambda=0.5, beta=0.5),
            Expectation()
        ],

                    ) do sp, t, i
    # ... sub-problem definition will go here ...
end
```

Let us now turn out attention to the definition of the sub-problems. As we mentioned earlier, we seek to describe each subproblem (Eq. 5) as a JuMP model (the `sp` in the constructor). Where possible, we have utilized the JuMP tooling directly. However, some modifications were necessary (for instance, to create state variables). In the next few sub-sections, we step through the process of representing the stage problem by a JuMP model. However, before we begin, we define some data that will be used later. These vectors correspond to the returns ω_s and ω_b in the `Low Return` and `High Return` Markov states:

```
    ws = [1.06, 1.25]
    wb = [1.12, 1.14]
```

We now describe how to create state variables in `SDDP.jl`.

3.2. State Variables

State variables are created using the `@state` macro. It takes three arguments. The first is the variable pointing to the JuMP model in the `do sp, t, i` part of the model constructor (i.e. `sp`). The second is a definition of the outgoing state variable (i.e. x_t), and the third is the definition of the incoming state variable (i.e. $\bar{x}_t$). The outgoing state variable permits any valid JuMP syntax that could be used to construct a variable via `@variable`. For example:

```
@state(sp, x >= 0, ... still to describe ...)
@state(sp, i <= x[i=1:3] <= 2i, ... still to describe ...)
```

The third argument contains syntax specific to `SDDP.jl`. It must be of the form `[variablename] == [initial value in the root node]`. This will create a new JuMP variable with the same index sets as the outgoing state variable. In addition, the initial value can use the elements in the index set in its definition. For example:

```
@state(sp, x >= 0, xbar == 0.0)
@state(sp, i <= x[i=1:3] <= 2i, xbar == i - 0.5)
```

We now construct the state variables of the Asset Management example:

```
    @state(sp, xs >= 0, xsbar == 0.0)
    @state(sp, xb >= 0, xbbar == 0.0)
```

Just as JuMP has a plural form of many of its macros (i.e. `@variables`), there is an `@states` macro. Its syntax will be familiar to users of JuMP:

```
@states(sp, begin
    x >= 0, xbar == 0.0
    i <= y[i=1:3] <= 2i, ybar == i - 0.5
end)
```

3.3. Control Variables

Control variables can be added as normal JuMP variables (using the `@variable` macro). In the formulation of the Asset Management example, there are two control variable (u and v) in the final stage. These should not be added to the first three stages. This can be accomplished by using an `if` statement on the stage counter t (from the `do sp, t, i` line). For example:

```
if t == 4
    @variable(sp, u >= 0)
    @variable(sp, v >= 0)
end
```

3.4. Transition Function and Feasibility Sets

The transition constraints ($\mathcal{T}_i$) and feasibility sets of the controls ($\mathcal{U}_i$) and state ($\mathcal{X}_i$) can be implemented as normal JuMP constraints. However, if they depend on the realization of the stage-wise noise, we need to use the `@rhsnoise` macro instead the `@constraint` macro.

Important: the stage-wise independent noise must occur in the RHS of the constraint. It cannot be used in the constraint matrix (i.e. as a coefficient to a variable).

The `@rhsnoise` macro takes three arguments. The first is the subproblem variable (i.e. `sp`). The second has the form `keyword = [ list of values]`. The third is any valid JuMP constraint that incorporates the `keyword` in the definition. For example:

```
w = [0.5, 1.0]
@rhsnoise(sp, i=w, x <= i)
@rhsnoise(sp, i=1:2, x <= w[i])
```

The probability of each scenario can be set with

```
setnoiseprobability!(sp, [ ... probabilities ... ])
```

The remainder of the transition function and feasibility sets that do not rely on the stage-wise independent noise can be set using the `@constraint` macro. We now give the definitions for the Asset Management example:

```
if t == 1
    @constraint(sp, xs + xb == 55 + xsbar + xbbar)
elseif t == 2 || t == 3
    @rhsnoise(sp, phi=[-1, 5], ws[i] * xsbar +
        wb[i] * xbbar + phi == xs + xb)
    setnoiseprobability!(sp, [0.6, 0.4])
else # t == 4
    @constraint(sp, ws[i] * xsbar + wb[i] * xbbar + u - v == 80)
end
```

Within a stage problem, there can be multiple constraints that depend on the realization of the stage-wise independent variable. These can be created by multiple `@rhsnoise` lines, or by the `@rhsnoises` macro (described below). If multiple `@rhsnoise` macros are used, the user must ensure that the number of elements in the vector of the second argument is the same for each call. For example, the following is valid:

```
@rhsnoise(sp, w=[1,2], x >= w)
@rhsnoise(sp, w=[3,4], x <= w)
```

but the following is invalid:

```
@rhsnoise(sp, w=[1,2], x >= w)
@rhsnoise(sp, w=[3,4,5], x <= w)
```

The use of the `@rhsnoises` macro is very similar to other plural versions of the JuMP macros (such as `@constraints`). In contrast to the `@rhsnoise` macro, the third argument is a `begin ...` end block with one constraint per line.

```
@rhsnoises(sp, i=[1,2], begin
    x >= i
    x <= i + 2
end)
```

Performance Tip: order the realizations of the random variable from most-probable to least-probable. (i.e. `setnoiseprobability!(sp, [0.6, 0.4])` instead of `setnoiseprobability!(sp, [0.4, 0.6])`.)

3.5. Objective Function

All that remains is to specify the stage-objective function (i.e. C_i) for each stage. In contrast to JuMP, this is done with the `@stageobjective!` macro. There are two versions of the `@stageobjective!` macro. The first should be used when the stage-objective function does not depend on the realization of the random variable. It takes two arguments. The first is the

JuMP subproblem (sp). The second is any valid JuMP expression that could be used in the third argument of the JuMP macro @objective (i.e. the objective expression).

The other version of the @stageobjective! macro takes three arguments and should be used when the stage-objective function *does* depend on the realization of the random variable. This functions very similarly to the @rhsnoise macro. The second argument should be of the form keyword = [list of values]. If the @rhsnoise macro is also used in the subproblem, then there should be the same number of realizations in the objective as there are realizations in the right-hand-side random variable. The two noises will be sampled in unison, so that when the first element of the RHS noise is sampled, so to will the first element of the objective noise. If the two random variables should be sampled independently, the user should form the Cartesian product of the two random variables. The noises are sampled using the probability distribution set by the setnoiseprobability! function. The objective functions for the Asset Management problem are:

```
if t == 1
    @stageobjective!(sp, 0.0)
elseif t == 2 || t == 3
    @stageobjective!(sp, psi = [0.0, 0.02], -psi * xs)
else # t == 4
    @stageobjective!(sp, 4u - v)
end
```

We have now finished communicating the model to SDDP.jl. The full code for the model is given in Appendix A.

4. Efficient Computational Solution of the Problem

Of the four basic steps in the mathematical optimization process, the efficient computational solution of the problem has received the most attention from the academic community. In this section, we highlight some of the features from the literature that have been included in SDDP.jl. It does not represent a complete state-of-the-art solution. One reason for this is the quantity and complexity of ideas that have been proposed presents a formidable challenge for a single author to implement. However, we hope to slowly build out the capability of SDDP.jl over time to incorporate as many of the state-of-the-art features as possible.

The basic call to solve the SDDP model m is

```
status = solve(m; kwargs...)
```

The status describes how the algorithm was terminated. There are many keyword arguments that can be used to modify the solution process. In the following sections, we describe some of the important ones.

4.1. Termination Criteria

One question that is unresolved in the literature is the best way to terminate the SDDP algorithm. In SDDP.jl, we provide four different methods.

1. Iteration Limit: the algorithm terminates after a set number of iterations. This can be set with the iteration_limit::Int keyword in the solve command;

2. Time Limit: the algorithm terminates after a set number of seconds. This can be set with the `time_limit::Float64` keyword in the `solve` command;
3. Statistical convergence check: the algorithm terminates when the lower bound is within a confidence interval estimate for the upper bound (if minimizing). In particular, we implement the sequential sampling test of [26]. Briefly, the algorithm works as follows: after a number of iterations, a small number of Monte-Carlo simulations are conducted. With these values, a confidence interval for the estimate of the statistical bound is created. If there is evidence of convergence, more simulations are conducted. If there is no evidence of convergence, performing more simulations will only reduce the variance and further reduce the evidence of convergence. Therefore, we halt our test for convergence and resume the cutting phase of the SDDP algorithm. We control the behaviour of the policy simulation phase of the algorithm using the `simulation = MonteCarloSimulation(;kwargs...)` constructor. This just groups a series of related keyword arguments. The keywords are

   - `frequency::Int` the frequency (by iteration) with which to run the policy simulation phase of the algorithm in order to construct a statistical bound for the policy. Defaults 0 (never run).
   - `min::Int` the minimum number of simulations to conduct before constructing a confidence interval for the bound. Defaults to 20.
   - `step::Int` the number of additional simulations to conduct before constructing a new confidence interval for the bound. Defaults to 1.
   - `max::Int` the maximum number of simulations to conduct in the policy simulation phase. Defaults to `min`.
   - `confidence::Float64` Confidence level of the confidence interval. Defaults to '0.95' (95% CI).
   - `termination::Bool` Whether to terminate the solution algorithm with the status `:converged` if the deterministic bound is with in the statistical bound after `max` simulations. Defaults to false.

4. Bound Stalling: terminate the algorithm when the deterministic bound fails to improve after a number of iterations (as suggested by [27]). This can be controlled by the `bound_convergence = BoundConvergence(;kwargs...)` keyword. `BoundConvergence` has the following keywords:
   (a) `iterations::Int` terminate if the maximum deviation in the deterministic bound from the mean over the last `iterations` number of iterations is less than `rtol` (in relative terms) or `atol` (in absolute terms).
   (b) `rtol::Float64` maximum allowed relative deviation from the mean. Defaults to '0.0'
   (c) `atol::Float64` maximum allowed absolute deviation from the mean. Defaults to '0.0'

Performance Tip: we prefer to set a maximum number of iterations and a time limit, rather than some other test of convergence.

4.2. Cut Oracles

As the SDDP algorithm progresses, many cuts (typically thousands) are added to each subproblem. This increases the computational effort required to solve each subproblem (a real-world model may begin with subproblems with tens of variables and constraints, only to add 5000 constraints!). In addition, many of the cuts created early in the solution process may be redundant once additional cuts are added. This issue has spurred a vein of research (see [28, 29, 30, 31]) into heuristics for choosing cuts to keep or remove (henceforth called "cut selection"). To facilitate the development of such heuristics, `SDDP.jl` features the concept of a *cut oracle* associated with each subproblem. A cut oracle has two jobs: it should store the complete list of cuts created for that subproblem; and when asked, it should provide a subset of those cuts to retain in the subproblem. This can be done by defining a new sub-type of the abstract type `AbstractCutOracle` (defined in `SDDP.jl`), and then overloading two methods: `storecut!`, and `validcuts`. To illustrate this feature, we now give the code that implements the default behaviour of `SDDP.jl` (no cut selection):

```
immutable DefaultCutOracle <: AbstractCutOracle
    cuts::Vector{Cut}
end
DefaultCutOracle() = DefaultCutOracle(Cut[])

# add the cut to the oracle
function storecut!(o::DefaultCutOracle, m::SDDPModel, sp::JuMP.Model, c::Cut)
    push!(o.cuts, c)
end

# return all cuts to the user
function validcuts(o::DefaultCutOracle)
    return o.cuts
end
```

By creating a new type that is a sub-type of `AbstractCutOracle`, users can leverage Julia's multiple dispatch mechanisms to easily integrate new functionality into the library without having to dive into the internals. At present, only the default (described above), and Level-One [28] cut selection methods have been implemented. The Level-One cut selection oracle can be created as follows:

```
julia> cut_oracle = DematosCutOracle()
```

Cut oracles can be added to the model with the `cut_oracle` keyword in the `SDDPModel` constructor (see Section 3.1).[5] For example:

```
m = SDDPModel(
    # ... other keyword arguments ...
        cut_oracle = DematosCutOracle()
```

[5]Readers may argue that a cut oracle belongs to the solution process rather than the model object. However this design is necessary to satisfy some of Julia's quirks regarding type stability.

```
                    ) do sp, t, i
    # ... sub-problem definition will go here ...
end
```

Due to the design of JuMP, we are unable to delete cuts from the model[6]. Therefore, selecting a subset of cuts involves rebuilding the subproblems from scratch. The user can control the frequency by which the cuts are selected and the subproblems rebuilt with the `cut_selection_frequency::Int` keyword argument to `solve`. Frequent cut selection (i.e. when `cut_selection_frequency` is small) reduces the size of the subproblems that are solved, but incurs the overhead of rebuilding the subproblems. However, infrequent cut selection (i.e. when `cut_selection_frequency` is large) allows the subproblems to grow large (by adding many constraints) leading to an increase in the solve time of individual subproblems. Ultimately, this will be a model specific trade off.

Performance Tip: As a rule of thumb, simpler (i.e. few variables and constraints) models benefit from more frequent cut selection compared to complicated (i.e. many variables and constraints) models.

4.3. Risk Measures

In the last five years, significant efforts have been made to incorporate risk in to the SDDP algorithm [23, 20, 32, 25, 33, 34]. In `SDDP.jl`, we use the "change-of-probability" approach of [25]. They use the result of [35] to show how, for random variable Z with a finite number of realizations $\{Z(\omega) : \omega \sim \Omega\}$ where Ω is a finite discrete distribution with probability $p_\omega = \mathbb{P}(\Omega = \omega)$, a coherent risk measure $\mathbb{F}$ can be expressed as

$$\underset{\omega\in\Omega}{\mathbb{F}}[Z] = \sum_{\omega\in\Omega} p_\omega \times \xi_\omega^{\mathbb{F}} \times Z(\omega), \tag{15}$$

where $\xi_\omega^{\mathbb{F}} \in \mathbb{R}^+$ and meets some technical assumptions. If we replace $p_\omega \times \xi_\omega^{\mathbb{F}}$ with $\hat{p}$, we see this corresponds to taking the expectation of the same realizations of the random variable with respect to a "changed" probability distribution. Therefore, a "change-of-probabilty" risk measure is a function that takes the realizations of the random variable Z and the original probabilities p_ω, and returns the risk-adjusted distribution $\hat{p}$. For example, the worst-case risk measure is

$$\hat{p}_\omega = \begin{cases} 1, & \omega = \underset{\omega\in\Omega}{\arg\min}\,\{Z(\omega)\} \\ 0, & \textit{otherwise.} \end{cases} \tag{16}$$

As we described in Section 3.1, two risk measures have been implemented in the `SDDP.jl` library: Expectation, and a convex combination of AV@R and Expectation. However, similarly to the *cut oracle* design, `SDDP.jl` allows a core function (`modifyprobability!`) to be overloaded to allow the development and testing of new risk measures (for a real-world example of this, see [36]). `modfiyprobability!` takes a number of arguments. The first is an instance of the risk measure. The second is a vector of `Float64` corresponding to the risk-adjusted probability (according to the "change-of-probability" approach of [25]) of each scenario. This should be modified in-place by the function. The third argument is a vector of the un-adjusted probabilities (i.e. the original probability distribution). The fourth argument is a vector of objective

[6]This may be rectified in future re-writes of JuMP.

values (one for each scenario i.e. for all $j \in i^+; \omega_j \in \Omega_j$). The fifth and sixth arguments are the `SDDPModel` and JuMP subproblems to allow for other risk-measures that require additional information.

To illustrate this feature, we give the code necessary to create the worst-case risk measure[7]. The worst-case risk measure places all of the probability on the scenario with the greatest objective value (if minimizing), or the smallest objective value (if maximizing):

```
immutable WorstCase <: AbstractRiskMeasure end
function modifyprobability!(::WorstCase,
        riskadjusted_distribution,
        original_distribution::Vector{Float64},
        observations::Vector{Float64},
        m::SDDPModel,
        sp::JuMP.Model
    )
    if getsense(sp) == :Min
        idx = indmax(observations)
    else
        idx = indmin(observations)
    end
    riskadjusted_distribution .= 0.0
    riskadjusted_distribution[idx] = 1.0
end
```

After defining this code, the user could use `risk_measure = WorstCase()` in their definition of the `SDDPModel` object *as if it were a risk measure defined in `SDDP.jl`.* This highlights one of the core benefits of building a SDDP framework in Julia: that the underlying library can easily be extended by the user without modifying the library itself.

4.4. Parallelization

The SDDP algorithm is highly parallelizable. This observation dates back to [8] who noted that the process could be conducted asyncronously, with performing iterations independently, and sharing cuts periodically. Different authors [37, 38] have proposed different methods for efficiently parallelizing the algorithm. However, we choose an approach that minimizes inter-process communication.

In our implementation, one processor is designated the "master" process, and the remainder are designated as "slaves". Each slave receives a full copy of the SDDP model and is set to work performing cutting iterations. At the end of each iteration, the slave passes the master the cuts it discovered during the iteration, and receives any new cuts discovered by other slaves. The slave also queries the master as to whether it should terminate, perform another cutting iteration, or perform a simulation. If the master requests a simulation (for example, in order to calculate a confidence interval in order to test for convergence), the slave returns the objective value of the simulation rather than a new set of cuts.

[7]This risk measure is not yet implemented in `SDDP.jl`. However, it would make a great start for a reader who would like to begin contributing to the project at `https://github.com/odow/SDDP.jl`.

This functionality can be controlled by the `solve_type` keyword. For example, `solve_type=Serial()` will solve the problem utilizing one processor. However, setting `solve_type=Asyncronous()` will utilize all of the processors available to Julia.

Performance Tip: typically, you should start Julia with the same number of processors as you have cores. For example, on a typical 8-core workstation with 16Gb of RAM, we find starting Julia with `julia -p 8` works the best.

4.5. Logging

During the solution process, `SDDP.jl` outputs some logging information (an example of this is given in Appendix Appendix B). We briefly describe the columns:

- `Objective Simulation`: If this consists of one value, it is the objective value of the forward simulation. If it is a pair, it correspond to the confidence interval estimate as a result of a simulation phase;
- `Objective Bound`: is the deterministic bound of the problem (lower if minimizing, upper if maximizing);
- `Objective % Gap`: the relative percentage gap between the closest edge of the confidence interval and the deterministic bound (i.e. if minimizing: $([CIlower]-lowerbound)/lowerbound)$;
- `Cut Passes #`: the number of iterations (one forward pass, one backward pass) conducted;
- `Cut Passes Time`: the number seconds spent iterating to discover cuts;
- `Simulations #`: the number of forward passes conducted in order to estimate the upper (if minimizing, otherwise lower) bound;
- `Simulations Time`: the number of seconds spent conducting forward passes in order to estimate the upper (if minimizing, otherwise lower) bound;
- `Total Time`: total time (in seconds) of the solution process (including initialization).

Logging can be turned off by setting `print_level` to 0. It can also be written to the file specified by the `log_file` keyword.

4.6. Other extensions

One core goal of Dunning et al. [18] in designing JuMP was to make it extensible [18]. Therefore, JuMP contains functionality for injecting user-code into many of the points in the solution process. By building on-top-of JuMP and Julia, we have leveraged these features to allow other authors to implement features on-top-of `SDDP.jl` that seamlessly integrate into the existing infrastructure. The first such example of this is the Julia Package `SDDiP.jl` [39] which implements the method of [22] to solve stochastic integer programs. We believe this modularity and extensibility of the `SDDP.jl` framework is a rich starting point for future researchers to build upon.

5. Communicating the Solution to the User

The last step in solving a mathematical optimization problem is communicating the solution back to the user. In deterministic problems, this means some comment about the solver termination status (did it succeed or fail? If it failed, why?), the primal solution (feasibility etc., and the value of each of the variables, and maybe constraints in the solution), and if applicable, the dual solution. This can be compactly communicated to the user (the primal solution just requires a vector with one element for every variable).

In stochastic problems, the solution to the optimization problem isn't one solution vector, but a *policy*. That policy is a function that, for every stage, maps an incoming state variable and observation of the stage-wise independent noise to a feasible action. As such, communicating the policy to the user is much more difficult.

Status. The call to `solve(m, ...)` returns a symbol describing the termination status of the algorithm. It is one of:

1. `:solving`: if the solution process has not finished;
2. `:interrupted`: if the user halts the solution process by throwing an `InterruptException` (i.e. by calling `CRTL+C`);
3. `:converged`: if the deterministic bound is within the confidence interval of the statistical bound;
4. `:max_iterations`: if the maximum number of iterations has been exceeded;
5. `:bound_convergence`: if the deterministic bound has stalled;
6. `:time_limit`: if the time spent solving exceeds the limit.

Objective Bound. The user can query the objective bound (i.e. the best known bound for Eq. 1) of a solved `SDDPModel` with the `getbound` function. For example:

```
julia> getbound(m)
```

Policy Simulation: Monte-Carlo. One way of evaluating the policy is to simulate it's performance via Monte-Carlo simulation. This can be done using the `simulate` function. `simulate` takes three arguments: the first is the `SDDPModel` `m`; the second is the number of independent Monte-Carlo simulations to perform; the third is a vector of symbols that correspond to variables in the JuMP subproblems. For example:

```
julia> results=simulate(m::SDDPPModel, N::Int, variables::Vector{Symbol})
```

`results` is a vector containing one dictionary for each simulation (indexed as `results[simulation index]`). In addition to the `variables` specified in the function call, other special keys are:

1. `:stageobjective` : costs incurred during the stage (not future);
2. `:obj` : objective of the stage including future cost;
3. `:markov` : index of Markov state visited;
4. `:noise` : index of noise visited;
5. `:objective` : total objective of simulation.

All values can be accessed as follows:

```
julia> results[simulation index][key][stage]
```

with the exception of ':objective' which is just:

```
julia> results[simulation index][:objective]
```

For example, the following code will perform 100 Monte-Carlo simulations, and store the value of the JuMP variables `x` and `u` in every stage:

```
julia> results = simulate(m, 10, [:x, :u])
```

We can access the value of the variable `x` in the third stage of the second simulation by:

```
julia> results[2][:x][3]
```

We could also calculate the mean objective of the policy across the simulations by:

```
julia> mean(r[:objective] for r in results)
```

Policy Simulation: Historic. Another option is to simulate the policy along one particular trajectory. This can be conducted using a variation on the `simulate` function with two keyword arguments: `noises` and `markovstates`. `noises` is a vector of integers with one element for each stage giving the index of the (in-sample) stage-wise independent noise to sample in each stage. `markovstates` is a vector of integers with one element for each stage giving the index of the (in-sample) Markov state to sample in each stage. For example:

```
julia> results = simulate(m, [:x, :u],
                     noises=[1,2,2], markovstates=[1,1,2]
                 )
```

To simulate out-of-sample outcomes (i.e. realizations of the stage-wise independent random variables that were not used to build the policy) the user has two options:

1. include the out-of-sample outcomes in the initialization of the model but use `setnoiseprobability!` to set their probability to 0.0;
2. save the cuts to file, build a new `SDDPModel` with the new realizations, load the cuts from file, then simulate.

Although the first option incurs some overhead during the solution process, this can be minimized by ensuring that the out-of-sample outcomes are collected at the end of the noise vectors.

For the Asset Management example, we perform 100 Monte-Carlo simulations of the policy:

```
results = simulate(m, 100, [:xs, :xb])
```

Simulation Visualization. Now that we have simulated the policy, we need to interpret the results. This can be done by plotting the trajectories of each state and action over time. To begin, we initialize a new plotting object:

```
julia> p = SDDP.newplot()
```

Next, we can add plots to this object using the `SDDP.addplot!` method:

```
SDDP.addplot!(p::SimulationPlot, I::AbstractVector{Int},
    T::AbstractVector{Int}, f::Function; kwargs...)
```

This will add a new figure to the SimulationPlot `p`, where the y-value is given by `f(i,t)` for all `i` in `I` (one for each series) and `t` in `T` (one for each stage). There are a number of keyword arguments that can be used to modify the plots. They are:

1. `xlabel`: set the xaxis label;
2. `ylabel`: set the yaxis label;
3. `title`: set the title of the plot;
4. `ymin`: set the minimum y value;
5. `ymax`: set the maximum y value;
6. `cumulative`: plot the additive accumulation of the value across the stages;
7. `interpolate`: interpolation method for lines between stages. Defaults to `"linear"`. See the d3 documentation at `https://github.com/d3/d3-3.x-api-reference/blob/master/SVG-Shapes.md#line_interpolate` for all options.

Finally, we can open a browser window and render the final interactive Javascript plot with:

```
julia> show(p)
```

Therefore, the code to visualize the policy of the Asset Management example is:

```
p = SDDP.newplot()
SDDP.addplot!(p, 1:1000, 1:4, (i, t)->results[i][:stageobjective][t],
    title="Objective", ylabel="Profit (\$)", cumulative=true)
SDDP.addplot!(p, 1:1000, 1:4, (i, t)->results[i][:xs][t],
    title="Stocks")
SDDP.addplot!(p, 1:1000, 1:4, (i, t)->results[i][:xb][t],
    title="Bonds")
show(p)
```

Value Function Visualization. Plot the value function of stage `stage` and Markov state `markovstate` in the SDDPModel `m` at the points in the discretized state space given by `states`. If the value in `states` is a real number, the state is evaluated at that point. If the value is a vector, the state is evaluated at all the points in the vector. At most two states can be vectors.

```
SDDP.plotvaluefunction(m, 3, 1, 0.0:2.0:100, 0.0:2.0:100,
    label1 = "Stocks",
    label2="Bonds"
)
```

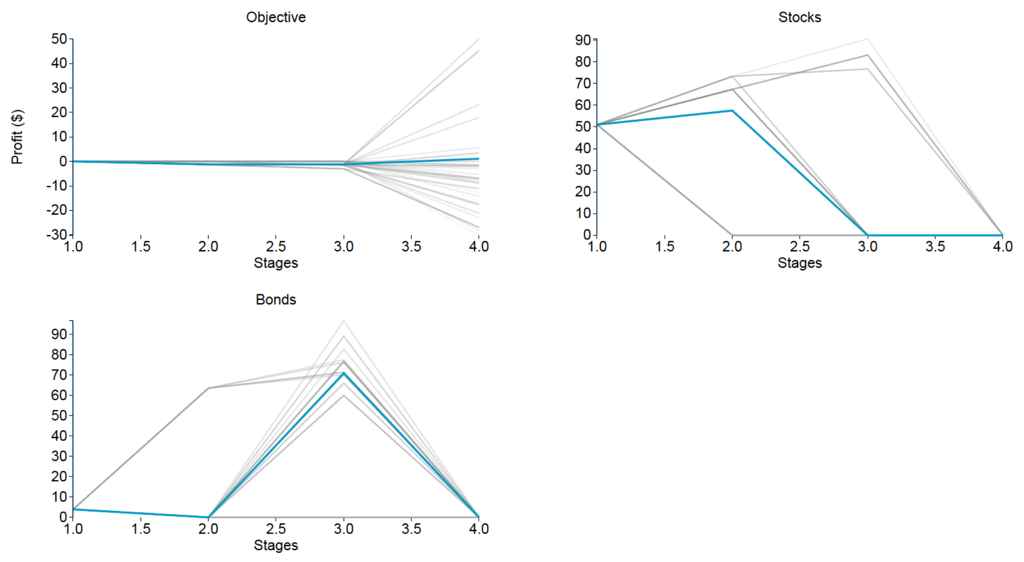

Figure 3: Policy Simulation of the Asset Management Example

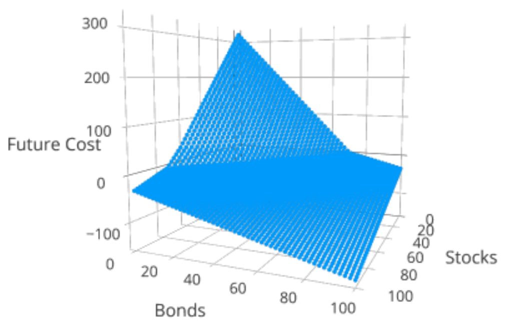

Figure 4: First Stage Value Function of the Asset Management Example

6. Conclusion

This paper describes `SDDP.jl`, a Julia package for Stochastic Dual Dynamic Programming. We believe the unique features of `SDDP.jl` (that it is written entirely in a high-level language and built upon the state-of-the-art mathematical optimization library JuMP) provide an excellent platform upon which to build and test, new improvements and extensions, to the SDDP algorithm.

We hope the community will see the value in collaborating on open-source implementations of stochastic programming codes to avoid the duplication of work that has hampered progress in our field.

Acknowledgements

`SDDP.jl` is the culmination of a decade of work at the Electric Power Optimization Center (EPOC) at the University of Auckland. Many people at EPOC have contributed to the development of SDDP, and related codes over the years. These include Andy Philpott, Ziming Guan, Geoff Pritchard, Anthony Downward, Faisal Wahid, Vitor de Matos, and Lea Kapelevich. Credit also goes to Vincent Leclère, François Pacaud, Tristan Rigaut, Henri Gerard, and Benoît Legat.

References

[1] B. A. Murtagh, Advanced linear programming: computation and practice, McGraw-Hill International Book Co., 1981.

[2] I. Dunning, J. Huchette, M. Lubin, JuMP: A modeling language for mathematical optimization, arXiv:1508.01982 [math.OC] URL `http://arxiv.org/abs/1508.01982`.

[3] Gurobi Optimization, Gurobi 7.5 Performance Benchmarks, Tech. Rep., Gurobi Optimization, Inc, URL `https://gurobi.com/pdfs/benchmarks.pdf`, 2017.

[4] W. B. Powell, A Unified Framework for Optimization under Uncertainty, in: A. Gupta, A. Capponi (Eds.), Optimization Challenges in Complex, Networked and Risky Systems, TutORials in Operations Research, INFORMS, 45–83, URL `http://castlelab.princeton.edu/Papers/Powell-UnifiedFrameworkforStochasticOptimization_April162016.pdf`, 2016.

[5] W. B. Powell, Clearing the Jungle of Stochastic Optimization, in: A. M. Newman, J. Leung, J. C. Smith, H. J. Greenberg (Eds.), Bridging Data and Decisions, INFORMS, ISBN 978-0-9843378-5-9, 109–137, URL `http://pubsonline.informs.org/doi/abs/10.1287/educ.2014.0128`, 2014.

[6] J. R. Birge, M. A. Dempster, H. I. Gassmann, E. Gunn, A. J. King, S. W. Wallace, A standard input format for multiperiod stochastic linear programs, IIASA Working Paper, WP-87-118, IIASA, Laxenburg, Austria, URL `http://pure.iiasa.ac.at/2934`, 1987.

[7] H. I. Gassmann, B. Kristjansson, The SMPS format explained, IMA Journal of Management Mathematics 19 (4) (2007) 347–377, ISSN 1471-678X, 1471-6798, doi:\bibinfo{doi}{10.1093/imaman/dpm007}, URL `https://academic.oup.com/imaman/article-lookup/doi/10.1093/imaman/dpm007`.

[8] M. Pereira, L. Pinto, Multi-stage stochastic optimization applied to energy planning, Mathematical Programming 52 (1991) 359–375.

[9] Z. Guan, Strategic Inventory Models for International Dairy Commodity Markets, PhD Thesis, University of Auckland, Auckland, New Zealand, 2008.

[10] A. B. Philpott, V. L. de Matos, Dynamic sampling algorithms for multi-stage stochastic programs with risk aversion, European Journal of Operational Research 218 (2) (2012) 470–483, URL `http://www.sciencedirect.com/science/article/pii/S0377221711010332`.

[11] A. Helseth, H. Braaten, Efficient Parallelization of the Stochastic Dual Dynamic Programming Algorithm Applied to Hydropower Scheduling, Energies 8 (12) (2015) 14287–14297, ISSN 1996-1073, doi:\bibinfo{doi}{10.3390/en81212431}, URL `http://www.mdpi.com/1996-1073/8/12/12431`.

[12] K. I. Ourani, C. G. Baslis, A. G. Bakirtzis, A Stochastic Dual Dynamic Programming model for medium-term hydrothermal scheduling in Greece, in: Universities Power Engineering Conference (UPEC), 2012 47th International, IEEE, 1–6, URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6398566`, 2012.

[13] M. R. Bussieck, M. C. Ferris, T. Lohmann, GUSS: Solving collections of data related models within GAMS, in: Algebraic Modeling Systems, Springer, 35–56, URL `http://link.springer.com/10.1007%2F978-3-642-23592-4_3`, 2012.

[14] T. Asamov, W. B. Powell, Regularized Decomposition of High-Dimensional Multistage Stochastic Programs with Markov Uncertainty, arXiv preprint arXiv:1505.02227 URL `http://arxiv.org/abs/1505.02227`.

[15] P. Parpas, B. Ustun, M. Webster, Q. K. Tran, Importance sampling in stochastic programming: A Markov chain Monte Carlo approach, INFORMS Journal on Computing 27 (2) (2015) 358–377, URL `http://pubsonline.informs.org/doi/abs/10.1287/ijoc.2014.0630`.

[16] PSR, Software | PSR, URL `http://www.psr-inc.com/softwares-en/`, 2016.

[17] A. Philpott, G. Pritchard, EMI-DOASA, Tech. Rep., Stochastic Optimization Limited, URL `http://www.emi.ea.govt.nz/Content/Tools/Doasa/DOASA\%20paper\%20by\%20SOL.pdf`, 2013.

[18] I. Dunning, J. Huchette, M. Lubin, JuMP: A modeling language for mathematical optimization, arXiv:1508.01982 [math.OC] URL `http://arxiv.org/abs/1508.01982`.

[19] J. R. Birge, F. Louveaux, Introduction to Stochastic Programming, Springer Series in Operations Research and Financial Engineering, Springer New York, New York, NY, ISBN 978-1-4614-0236-7 978-1-4614-0237-4, URL `http://link.springer.com/10.1007/978-1-4614-0237-4`, dOI: 10.1007/978-1-4614-0237-4, 2011.

[20] A. B. Philpott, V. L. de Matos, Dynamic sampling algorithms for multi-stage stochastic programs with risk aversion, European Journal of Operational Research 218 (2) (2012) 470–483, URL `http://www.sciencedirect.com/science/article/pii/S0377221711010332`.

[21] P. Girardeau, V. Leclere, A. B. Philpott, On the Convergence of Decomposition Methods for Multistage Stochastic Convex Programs, Mathematics of Operations Research 40 (1) (2015) 130–145, ISSN 0364-765X, 1526-5471, doi:\bibinfo{doi}{10.1287/moor.2014.0664}, URL `http://pubsonline.informs.org/doi/10.1287/moor.2014.0664`.

[22] J. Zou, S. Ahmed, X. A. Sun, Nested Decomposition of Multistage Stochastic Integer Programs with Binary State Variables, Optimization Online URL `http://www.optimization-online.org/DB_HTML/2016/05/5436.html`.

[23] A. Shapiro, Analysis of stochastic dual dynamic programming method, European Journal of Operational Research 209 (1) (2011) 63–72, ISSN 03772217, doi:\bibinfo{doi}{10.1016/j.ejor.2010.08.007}, URL `http://linkinghub.elsevier.com/retrieve/pii/S0377221710005448`.

[24] J. Bezanson, A. Edelman, S. Karpinski, V. B. Shah, Julia: A Fresh Approach to Numerical Computing, SIAM Review 59 (1) (2017) 65–98, ISSN 0036-1445, 1095-7200, doi:\bibinfo{doi}{10.1137/141000671}, URL `http://epubs.siam.org/doi/10.1137/141000671`.

[25] A. Philpott, V. de Matos, E. Finardi, On Solving Multistage Stochastic Programs with Coherent Risk Measures, Operations Research 61 (4) (2013) 957–970, ISSN 0030-364X, 1526-5463, doi:\bibinfo{doi}{10.1287/opre.2013.1175}, URL `http://pubsonline.informs.org/doi/abs/10.1287/opre.2013.1175`.

[26] G. Bayraksan, D. P. Morton, A Sequential Sampling Procedure for Stochastic Programming, Operations Research 59 (4) (2011) 898–913, ISSN 0030-364X, 1526-5463, doi:\bibinfo{doi}{10.1287/opre.1110.0926}, URL `http://pubsonline.informs.org/doi/abs/10.1287/opre.1110.0926`.

[27] M. Hindsberger, A. B. Philpott, Stopping criteria in sampling strategies for multistage SLP-problems, presented at the conference "Applied Mathematical Programming and Modelling", Varenna, Italy, 2002.

[28] V. L. de Matos, A. B. Philpott, E. C. Finardi, Improving the performance of Stochastic Dual Dynamic Programming, Journal of Computational and Applied Mathematics 290 (2015) 196–208, ISSN 03770427, doi:\bibinfo{doi}{10.1016/j.cam.2015.04.048}, URL `http://linkinghub.elsevier.com/retrieve/pii/S0377042715002794`.

[29] L. Pfeiffer, R. Apparigliato, S. Auchapt, Two methods of pruning Benders' cuts and their application to the management of a gas portfolio, Optmization Online URL `http://www.optimization-online.org/DB_FILE/2012/11/3683.pdf`.

[30] G. Nannicini, E. Traversi, R. W. Calvo, A Benders squared (B2) framework for infinite-horizon stochastic linear programs, Optimization Online.http://www.optimization-online.org/DB_HTML/2017/06/6101.html .

[31] M. Bandarra, V. Guigues, Multicut decomposition methods with cut selection for multistage stochastic programs, arXiv preprint arXiv:1705.08977 URL `https://arxiv.org/abs/1705.08977`.

[32] A. Shapiro, W. Tekaya, J. P. da Costa, M. P. Soares, Risk neutral and risk averse Stochastic Dual Dynamic Programming method, European Journal of Operational Research 224 (2) (2013) 375–391, ISSN 03772217, doi:\bibinfo{doi}{10.1016/j.ejor.2012.08.022}, URL `http://linkinghub.elsevier.com/retrieve/pii/S0377221712006455`.

[33] V. Kozmk, D. Morton, Risk-averse stochastic dual dynamic programming, Optimization Online. http://www.optimization-online. org/DB_HTML/2013/02/3794. html URL `http://www.optimization-online.org/DB_FILE/2013/02/3794.pdf`.

[34] V. Kozmk, D. P. Morton, Evaluating policies in risk-averse multi-stage stochastic programming, Mathematical Programming 152 (1-2) (2015) 275–300, ISSN 0025-5610, 1436-4646, doi:\bibinfo{doi}{10.1007/s10107-014-0787-8}, URL `http://link.springer.com/10.1007/s10107-014-0787-8`.

[35] A. Shapiro, D. Dentcheva, A. Ruszczyński, Lectures on Stochastic Programming: Modelling and Theory, Society for Induatrial and Applied Mathematics, Philadelphia, 2009.

[36] A. Philpott, V. de Matos, L. Kapelevich, Distributionally Robust SDDP, Tech. Rep., Electric Power Optimization Centre, Auckland, New Zealand, URL `http://www.epoc.org.nz/papers/DROPaperv52.pdf`, 2017.

[37] R. J. Pinto, C. T. Borges, M. E. P. Maceira, An Efficient Parallel Algorithm for Large Scale Hydrothermal System Operation Planning, IEEE Transactions on Power Systems 28 (4) (2013) 4888–4896, ISSN 0885-8950, 1558-0679, doi:\bibinfo{doi}{10.1109/TPWRS.2012.2236654}, URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6493512`.

[38] A. Helseth, H. Braaten, Efficient Parallelization of the Stochastic Dual Dynamic Programming Algorithm Applied to Hydropower Scheduling, Energies 8 (12) (2015) 14287–14297, ISSN 1996-1073, doi:\bibinfo{doi}{10.3390/en81212431}, URL `http://www.mdpi.com/1996-1073/8/12/12431`.

[39] L. Kapelevich, SDDiP.jl: SDDP extension for integer local or state variables, URL `https://github.com/`

lkapelevich/SDDiP.jl, [Online; accessed 2017-09-04], 2017.

Appendix A. Code for Asset Management

```
using SDDP, JuMP, Clp

ws = [1.25, 1.06]
wb = [1.14, 1.12]

m = SDDPModel(
    sense = :Min,
    stages = 4,
    objective_bound = -1000.0,
    solver = ClpSolver(),
    markov_transition = Array{Float64, 2}[
        [1.0]',
        [0.5 0.5],
        [0.5 0.5; 0.5 0.5],
        [0.5 0.5; 0.5 0.5]
    ],
    risk_measure = [
        Expectation(), Expectation(),
        NestedAVaR(lambda = 0.5, beta=0.5), Expectation()
    ]
) do sp, t, i

    @state(sp, xs >= 0, xsbar==0)
    @state(sp, xb >= 0, xbbar==0)
    if t == 1
        @constraint(sp, xs + xb == 55 + xsbar + xbbar)
        @stageobjective(sp, 0)
    elseif t == 2 || t == 3
        @rhsnoise(sp, phi=[-1, 5], ws[i] * xsbar +
            wb[i] * xbbar + phi == xs + xb)
        @stageobjective(sp, psi = [0.02, 0.0], -psi * xs)
        setnoiseprobability!(sp, [0.6, 0.4])
    else
        @variable(sp, u  >= 0)
        @variable(sp, v >= 0)
        @constraint(sp, ws[i] * xsbar + wb[i] * xbbar +
            u - v == 80)
        @stageobjective(sp, 4u - v)
    end
end

status = solve(m,
    max_iterations = 30,
    simulation = MonteCarloSimulation(
        frequency = 5,
        min  = 100,
        step = 100,
        max  = 500,
        termination = false
    ),
    bound_convergence = BoundConvergence(
        iterations = 5,
        rtol       = 0.0,
        atol       = 0.001
    ),
    log_file="asset.log"
)

status == :bound_convergence # true

isapprox(getbound(m), -1.278, atol=1e-3) # true

results = simulate(m, 100, [:xs, :xb])

p = SDDP.newplot()
SDDP.addplot!(p, 1:100, 1:4, (i, t)->results[i][:stageobjective][t],
    ylabel="Profit (\$)", cumulative=true)
SDDP.addplot!(p, 1:100, 1:4, (i, t)->results[i][:xs][t],
    ylabel="Stocks")
SDDP.addplot!(p, 1:100, 1:4, (i, t)->results[i][:xb][t],
    ylabel="Bonds")
SDDP.show(p)
```

Appendix B. Output from `SDDP.jl`

```
-------------------------------------------------------------------------------
                      SDDP Solver.  Oscar Dowson, 2017.
-------------------------------------------------------------------------------
    Solver:
        Serial solver
    Model:
        Stages:         4
        States:         2
        Subproblems:    7
        Value Function: Default
-------------------------------------------------------------------------------
              Objective              |  Cut  Passes    Simulations   Total
    Simulation       Bound   % Gap   |   #     Time     #    Time    Time
-------------------------------------------------------------------------------
```

```
     -41.484         -9.722         |     1    0.0      0    0.0    0.0
      -2.172         -7.848         |     2    0.0      0    0.0    0.0
       4.284         -7.550         |     3    0.0      0    0.0    0.0
     -10.271         -6.398         |     4    0.0      0    0.0    0.0
 -7.782    -4.760    -6.346  -22.6  |     5    0.0    500    0.4    0.4

                       ... some lines omitted ...

     -30.756         -5.570         |    29    0.1    1.9K   1.6    1.8
 -7.716    -4.601    -5.570  -38.5  |    30    0.1    2.4K   2.0    2.2
 --------------------------------------------------------------------------------
   Statistics:
       Iterations:         30
       Termination Status: max_iterations
---------------------------------------------------------------------------------
```