INFORMS JOURNAL ON COMPUTING
Vol. 00, No. 0, Xxxxx 0000, pp. 000–000
ISSN 0899-1499 | EISSN 1526-5528 | 00 | 0000 | 0001

INFORMS
DOI 10.1287/xxxx.0000.0000

`SDDP.jl`: a Julia package for stochastic dual dynamic programming

Oscar Dowson
Department of Industrial Engineering and Management Sciences at Northwestern University, Evanston, IL, USA, oscar.dowson@northwestern.edu

Lea Kapelevich
Operations Research Center at the Massachusetts Institute of Technology, Cambridge, MA, USA, lkap@mit.edu

We present `SDDP.jl`, an open-source library for solving multistage stochastic programming problems using the stochastic dual dynamic programming algorithm. `SDDP.jl` is built upon JuMP, an algebraic modeling language in Julia. JuMP provides `SDDP.jl` with a solver-agnostic, user-friendly interface. In addition, we leverage unique features of Julia, such as multiple-dispatch, to provide an extensible framework for practitioners to build upon our work. `SDDP.jl` is well-tested, and accessible documentation is available at `https://github.com/odow/SDDP.jl`.

Key words: julia; jump; stochastic dual dynamic programming

1. Introduction

Multistage stochastic programming is a structured way of modeling and solving sequential decision problems under uncertainty. Due to the large number of outcomes that are possible as a random process evolves over time, multistage stochastic programs are difficult to solve. One state-of-the-art solution technique for *convex* multistage stochastic programming problems is stochastic dual dynamic programming (SDDP) (Pereira and Pinto 1991).

Since the seminal work of Pereira and Pinto (1991), SDDP has been widely used to solve problems in both academia and industry, and the original paper has been cited more than 1000 times. Most famously, SDDP is used in production to set power prices in Brazil (Maceira et al. 2008, 2018).

However, until recently, no open-source, generic implementations of the algorithm existed in the public domain. (We discuss recent implementations in Section 6.) Instead, practitioners were forced to code their own implementations in a variety of languages and styles. Closed-source research implementations include those in AMPL (Guan 2008), C++ (Helseth and Braaten 2015), GAMS (Ourani et al. 2012), Java (Asamov and Powell 2018), and MATLAB (Parpas et al. 2015), as well as in commercial products in Fortran and Julia (PSR 2019) and Java (Quantego 2019).

In our opinion, this "re-invention of the wheel" and the large up-front cost to development has limited the adoption of the SDDP algorithm in areas outside the electricity industry (which is the focus of most researchers). Moreover, many researchers develop and test new algorithmic improvements without being able to easily compare their ideas against other implementations, or to the current state-of-the-art.

This paper presents `SDDP.jl`—a state-of-the-art implementation of the SDDP algorithm in Julia (Bezanson et al. 2017). The key feature of `SDDP.jl` is that it decouples the solution algorithm from the modeling framework. We achieve this decoupling by building `SDDP.jl` on-top-of JuMP (Dunning et al. 2017), an algebraic modeling language in Julia. JuMP enables `SDDP.jl` to provide a high-level interface for the user, while simultaneously providing performance that is similar to implementations in low-level languages.

By providing a free, open-source implementation of the SDDP algorithm, `SDDP.jl` has lowered the barriers to entry for practitioners looking to solve multistage stochastic programming problems. This view is best summarized in the following quote from Reus et al. (2019), who used `SDDP.jl` to solve a mine planning problem:

> SDDP is a complex algorithm that requires computational skill and experience to implement reliably and efficiently. The effort of implementing SDDP from scratch has probably precluded its wider adoption in other areas such as transportation, forestry, oil and gas and agriculture. The new library `SDDP.jl` ... bridges this gap by offering an efficient in-built implementation of the algorithm, inviting the user to focus exclusively on the modeling aspects of the problem.

The rest of this paper is structured as follows. In Section 2, we discuss how to model a multistage stochastic program to make it amenable to solution by `SDDP.jl`. Then, in Section 3, we briefly sketch the SDDP algorithm. In Section 4, we provide a worked example of modeling and solving a simple hydro-thermal scheduling problem using `SDDP.jl`. In

Section 5, we describe some of `SDDP.jl`'s state-of-the-art features. We conclude with a comparison of `SDDP.jl` to alternative libraries in Section 6.

This paper is not intended to be a comprehensive tutorial for `SDDP.jl`. Instead, readers are directed to `https://github.com/odow/SDDP.jl` for source-code, examples, tutorials, and documentation.

2. Modeling multistage stochastic programs

Before we can begin to solve multistage stochastic programming problems using `SDDP.jl`, we must define what they are. `SDDP.jl` uses the *policy graph* formulation of a multistage stochastic program introduced by Dowson (2020).

A policy graph, $\mathcal{G} = (R, \mathcal{N}, \mathcal{E}, \Phi)$, contains a root node R, a further set of nodes $\mathcal{N}$, and a set of directed edges, $\mathcal{E}$, connecting the nodes. For each node $i \in \mathcal{N}$, we seek a *decision rule*, $\pi_i(x, \omega)$, that maps the *incoming state variable* x and realization of a nodewise-independent *noise* ω to a feasible *control* u. The set of feasible controls is denoted by $U_i(x, \omega)$, and depends on the state variable and noise realization. The noise ω is a random variable drawn from a sample space Ω_i with probability mass function $\mathbb{P}(\omega \in \Omega_i)$. *Nodewise-independent* means that realizations of ω are independent of x and of the realizations of ω at all other nodes. As a result of choosing a control u, the incoming state x transitions to the *outgoing state variable* x', according to a *transition function* $x' = T_i(x, u, \omega)$, and the agent incurs a cost $C_i(x, u, \omega)$. Figure 1 shows a visualization of the components in each node $i \in \mathcal{N}$.

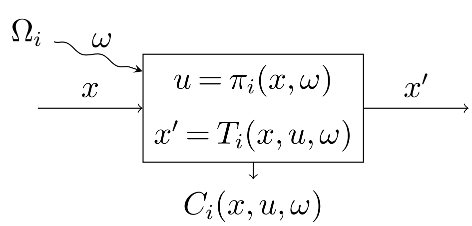

Figure 1 Schematic of a node in a policy graph.

In the policy graph, an $|\mathcal{N}| + 1$ by $|\mathcal{N}|$ matrix Φ specifies transition probabilities between nodes, with entries ϕ_{ij}. Given a node $i \in \mathcal{N} \cup \{R\}$, for each edge $(i, j) \in \mathcal{E}$ we transition to node $j \in \mathcal{N}$ with probability $\phi_{ij} > 0$. If no edge exists, then $\phi_{ij} = 0$. Since the edges represent probabilities, $\sum_{j:(i,j)\in\mathcal{E}} \phi_{ij} \leq 1$. This inequality accounts for discount factors; see Dowson (2020) for details. We say that the *children* of node i are the elements in the set $i^+ = \{j : \phi_{ij} > 0\}$, and i is a *leaf node* if $i^+ = \varnothing$.

Using this notation, a *multistage stochastic program* seeks an optimal decision rule, $\pi_i(x,\omega)$, for each node $i \in \mathcal{N}$, that minimizes the expected cost at the root node over all node transitions $R \to i \in R^+$ and realizations of the noise ω from Ω_i:

$$\min_{\pi} \mathop{\mathbb{E}}_{i \in R^+;\ \omega \in \Omega_i} [V_i(x_R, \omega)], \tag{1}$$

where x_R is the initial state vector, and:

$$V_i(x,\omega) = C_i(x,u,\omega) + \mathop{\mathbb{F}_{\mathrm{i}}}_{j \in i^+;\ \varphi \in \Omega_j} [V_j(x', \varphi)],$$

where $u = \pi_i(x,\omega) \in U_i(x,\omega)$, $x' = T_i(x,u,\omega)$, and $\mathbb{F}_{\mathrm{i}}$ is a coherent risk measure (Artzner et al. 1999). The collection of decision rules $\pi = \{\pi_1, \pi_2, \ldots, \pi_{|\mathcal{N}|}\}$ is referred to as a *policy*.

There are a few special-case policy graphs. First, if there are T nodes indexed from $t = 1, \ldots, T$, and each node has at most one child such that $\phi_{t,t+1} = 1$, then we say that the policy graph is *linear*. This corresponds to a traditional T-stage stochastic program in the literature. Second, if the nodes and transition matrix form a Markov chain, we say that the policy graph is *Markovian*. Finally, we differentiate between graphs that are cyclic and those that are acyclic. Cyclic policy graphs are infinite horizon stochastic programs, while acyclic policy graphs are finite horizon stochastic programs.

It is important to note that the nodes in a policy graph to not correspond to the nodes in a scenario tree. A stagewise-independent scenario tree with T stages and K realizations in each stage (i.e., K^T total nodes) corresponds to a linear policy graph with T nodes and $|\Omega_i| = K$. See Dowson (2020) for a larger discussion on the correspondence between policy graphs and scenario trees.

3. Solving multistage stochastic programs

Multistage stochastic programs are difficult to solve. One algorithm that has proven efficient at solving large scale problems is the stochastic dual dynamic programming (SDDP) method of Pereira and Pinto (1991). SDDP is a sampling-based version of the nested decomposition algorithm (Birge 1985). It can also be viewed as a value-function approximation algorithm (Powell 2016) in the spirit of value iteration (Howard 1960), approximate linear programming (de Farias and van Roy 2003), and approximate dynamic programming (Powell 2011), in which we exploit the structure of the problem through linear programming duality to find provably optimal basis functions. Since SDDP is well-studied in the literature, we will not give a full description of the algorithm in this paper; instead, we will provide a brief sketch of the main ideas. Readers are directed to Dowson (2020) for a full description of the algorithm as implemented in `SDDP.jl`.

First, using Bellman's *principle of optimality* (Bellman 1957), the decision rule π_i can be formulated as the arg min of the following optimization problem:

$$\begin{aligned} V_i(x,\omega) = & \\ & \min_{u,\bar{x},x'} \; C_i(\bar{x},u,\omega) + \underset{j\in i^+;\; \varphi\in\Omega_j}{\mathbb{F}_{\text{i}}} [V_j(x',\varphi)] \\ & \;\;\text{s.t.}\; \bar{x} = x \\ & \qquad x' = T_i(\bar{x},u,\omega) \\ & \qquad u \in U_i(\bar{x},\omega), \end{aligned} \tag{2}$$

where decision rule $\pi_i(x,\omega)$ takes the value of u in an optimal solution.

However, as formulated, (2) is intractable. Therefore, SDDP forms an approximation of the subproblem (2) for every node i in the policy graph. In this approximation, the expectation term is replaced by a variable θ and approximated from below by a set of linear basis functions called *cuts*. SDDP is an algorithm that iteratively refines the approximation. After K iterations the approximated version of model (2) is as follows:

$$\begin{aligned} V_i^K(x,\omega) = & \\ & \min_{u,\bar{x},x',\theta} \; C_i(\bar{x},u,\omega) + \theta \\ & \;\;\text{s.t.}\; \bar{x} = x \qquad\qquad [\lambda] \\ & \qquad x' = T_i(\bar{x},u,\omega) \\ & \qquad u \in U_i(\bar{x},\omega) \\ & \qquad \theta \geq \alpha_k^{(i)} + \beta_k^{(i)\top} x', \quad k = 1,\ldots,K \\ & \qquad \theta \geq -M, \end{aligned}$$

where M is a sufficiently large real number. Note that we add the so-called *fishing* constraint $\bar{x} = x$ to simplify the computation of the dual variable λ.

Each iteration of SDDP consists of two phases: (i) a forward pass, which samples a sequence of nodes and realizations of the node-independent noise terms and then generates a sequence of values for the state variables at which new cuts are computed; and (ii) a backward pass, which constructs a new cut at each value of the state variables from the forward pass. Under mild sampling assumptions (notably, independence of the forward passes), this algorithm has been shown to converge to an optimal policy almost surely in a finite number of iterations (Philpott and Guan 2008, Shapiro 2011, Guigues 2016).

The approximation formed by SDDP is only valid if $V_i(x,\omega)$ is convex with respect to x. Therefore, in order to guarantee convergence, a number of assumptions are needed. Given a policy graph $\mathcal{G} = (R, \mathcal{N}, \mathcal{E}, \Phi)$, the main assumptions are:

(A1) The number of nodes in $\mathcal{N}$ is finite.

(A2) The sample space Ω_i of random noise outcomes is finite at each node $i \in \mathcal{N}$.

(A3) Given fixed ω and x, and excluding the risk-adjusted expectation term $\mathbb{F}_i[\cdot]$, subproblem (2) associated with each node $i \in \mathcal{N}$ can be formulated as a convex programming problem.

(A4) For every node $i \in \mathcal{N}$, there exists a bounded and feasible optimal control u for every achieved incoming state x and realization of the noise ω.

(A5) For every node $i \in \mathcal{N}$, the sub-graph rooted at node i has a positive probability of reaching a leaf node.

Assumption (A3) requires some additional technical assumptions, such as an appropriate constraint qualification; see Girardeau et al. (2015) for details. Assumption (A3) can be weakened further to include problems with binary state variables. This results in the so-called *SDDiP* method of Zou et al. (2019). The SDDiP method is implemented in `SDDP.jl`, but we omit a description in the interest of brevity.

Notably, Assumption (A5) allows the modeling and solution of discounted-cost infinite-horizon stochastic programs; see Dowson (2020) for details.

4. An example using `SDDP.jl`

In this section, we provide the reader with a brief demonstration of `SDDP.jl`'s ease-of-use by means of a simplified hydro-thermal scheduling example.

In a hydro-thermal problem, the agent controls a hydro-electric generator and reservoir. At each time period, they need to choose a generation quantity from thermal g_t, and hydro g_h, in order to meed demand ω_d, which is a stagewise-independent random variable. The state variable, x, is the quantity of water in the reservoir at the start of each time period, and it has a minimum level of 5 units and a maximum level of 15 units. We assume that there are 10 units of water in the reservoir at the start of time, so that $x_R = 10$. The state-variable is connected through time by the water balance constraint:

$$x' = x - g_h - s + \omega_i,$$

where x' is the quantity of water at the end of the time period, x is the quantity of water at the start of the time period, s is the quantity of water spilled from the reservoir, and ω_i is a stagewise-independent random variable that represents the inflow into the reservoir during the time period.

We assume that there are three stages, $t = 1, 2, 3$, and that we are solving this problem in an infinite-horizon setting with a discount factor of 0.95. The structure of the policy graph is visualized in Figure 2, where the numbers on the arcs are the transition probabilities ϕ_{ij}.

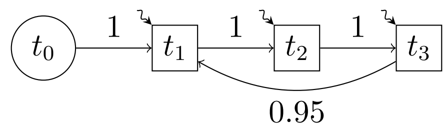

Figure 2 **Policy graph of the hydro-thermal scheduling example.**

In each stage, the agent incurs the cost of spillage, plus the cost of thermal generation. We assume that the cost of thermal generation is dependent on the stage $t = 1, 2, 3$, and that in each stage, $\omega = (\omega_i, \omega_d)$ is drawn from:

$$\Omega = \{(0, 7.5), (3, 5), (10, 2.5)\}$$

with equal probability.

Putting everything together, and ignoring the expectation term, the subproblem at each stage t can be formulated as follows:

$$\begin{aligned} V_t(x, \omega) = \min_{\bar{x}, x', s, g} \;& s + t \times g_t \\ \text{s.t. } & \bar{x} = x \\ & x' - \bar{x} + g_h + s = \omega_i \\ & g_h + g_t = \omega_d \\ & 5 \leq x' \leq 15 \\ & g_h, g_t, s \geq 0, \end{aligned} \tag{3}$$

and the problem faced by the agent is $\min \mathbb{E}[V_1(10, \omega)]$.

The driving principle behind the design of `SDDP.jl` is to represent model (3) as a JuMP model parameterized by the incoming state variable x and the realization of the stagewise independent random variable ω. This is achieved by extending some of the features of JuMP to introduce state variables, and to allow the parameterization of the JuMP model by ω. Where new methods and macros have been added (for example, adding state variables), we have tried to stay close to the syntactic feel of JuMP. These modifications typically produce standard JuMP constructs that are visible to the user (for example, a state variable is just a JuMP variable), along with some additional SDDP specific information that is hidden from the user (for example, how to link state variables between stages). Behind the scenes, `SDDP.jl` handles the expectation term in the objective, adds the *fishing* constraint $\bar{x} = x$, manages the realizations of the random variable ω, and passes the values of the state variables between nodes. Complete code to implement example (3) in `SDDP.jl` is given in Figure 3.

The first part of Figure 3 declares a linear policy graph with three nodes (with $\phi_{0,1} = \phi_{1,2} = \phi 2, 3 = 1$), and then adds an additional arc from node 3 to node 1 with probability 0.95, creating a cyclic policy graph. Much of the macro code (i.e., lines starting with `@`) in the first part of Figure 3 should be familiar to users of JuMP. Inside the `do-end` block, `sp` is a standard JuMP model, and `t` is an index for the state variable that will be called with `t = 1, 2, 3`. The state variable `x`, constructed by passing the `SDDP.State` tag to `@variable`, is actually a Julia `struct` with two fields: `x.in` and `x.out`, corresponding to the incoming and outgoing state variables respectively. Both `x.in` and `x.out` are standard JuMP variables. The `initial_value` keyword provides the value of the state variable in the root node (i.e., x_R). Compared to a JuMP model, one key difference is that we use `@stageobjective` instead of `@objective`. The `parameterize` function takes a list of supports for ω and parameterizes the JuMP model `sp` by setting the right-hand sides of the appropriate constraints (note how the constraints initially have a right-hand side of 0). By default, it is assumed that the realizations have uniform probability, but a probability mass vector can also be provided.

Once a model has been constructed, the next step is to train the policy. This can be achieved using the `SDDP.train` method. There are many options that can be passed, but `iteration_limit` terminates the training after the prescribed number of SDDP iterations. After training, we can simulate the policy. This is achieved using the `SDDP.simulate` function. Finally, we can use the `SDDP.ValueFunction` and `SDDP.evaluate` functions to obtain and evaluate the value function at different points in the state-space. We find that on average, 2.37 units of thermal are generated in the first stage, and the marginal value of water at the end of the first stage is \$0.66/unit.

```
using SDDP, GLPK, Statistics
graph = SDDP.LinearGraph(3)
SDDP.add_edge(graph, 3 => 1, 0.95)
model = SDDP.PolicyGraph(
  graph, sense = :Min, lower_bound = 0.0, optimizer = GLPK.Optimizer
) do sp, t
  @variable(sp, 5 <= x <= 15, SDDP.State, initial_value = 10)
  @variable(sp, g_t >= 0)
  @variable(sp, g_h >= 0)
  @variable(sp, s >= 0)
  @constraint(sp, balance, x.out - x.in + g_h + s == 0)
  @constraint(sp, demand, g_h + g_t == 0)
  @stageobjective(sp, s + t * g_t)
  SDDP.parameterize(sp, [[0, 7.5], [3, 5], [10, 2.5]]) do w
    set_normalized_rhs(balance, w[1])
    set_normalized_rhs(demand, w[2])
  end
end
SDDP.train(model, iteration_limit = 100)
sims = SDDP.simulate(model, 100, [:g_t])
mu = round(mean([s[1][:g_t] for s in sims]), digits = 2)
println("On average, $(mu) units of thermal are used in the first stage.")
# output >>> On average, 2.37 units of thermal are used in the first stage.
V = SDDP.ValueFunction(model[1])
cost, price = SDDP.evaluate(V, x = 10)
# output >>> (233.53828121398493, Dict(:x=>-0.660014))
```

Figure 3 Solving the hydro-thermal example using `SDDP.jl`**.**

5. Advancing the state-of-the-art

`SDDP.jl` includes many features that are not present in alternative software. These features include the ability to model and solve models formulated as arbitrarily complicated policy graphs using the algorithm of Dowson (2020), the objective-state interpolation method of Downward et al. (2020) that allows practitioners to model some classes of interstage-dependent price processes, and the distributionally robust risk measures of Philpott et al. (2018). Perhaps most successfully, the distributionally robust risk measure of Philpott et al. (2018) was implemented in less than 50 lines of code, demonstrating the ease with which practitioners are able to build upon `SDDP.jl` to advance the state-of-the-art.

More recently, `SDDP.jl` was extended to support *partially observable* multistage stochastic programs, using the algorithm of Dowson et al. (2019). `SDDP.jl` also supports multiple ways of solving multistage stochastic integer programs, including the SDDiP algorithm of Zou et al. (2019). Notably, these features are modular, and it is possible to combine them. Therefore, it is possible to solve a distributionally robust, partially observable, infinite-horizon, multistage stochastic integer program with a stagewise-dependent price process.

Other notable features of `SDDP.jl` that we have not discussed include a number of plotting utilities to help users visualize the policy and the value function, tools to help discretize stochastic processes into a policy graph, extensible schemes for modifying the sampling behavior on the forward and backward passes (enabling, for example, an out-of-sample simulation of the policy), and a function to convert a policy graph into the monolithic deterministic equivalent. `SDDP.jl` also utilizes Julia's distributed computing functionality to efficiently parallelize the training process.

Moreover, although rarely mentioned in the literature, SDDP is highly susceptible to numerical issues. (As one example, some solvers treat tolerances in the presolve and main simplex routines differently, leading to node $j \in i^+$ declaring that an "optimal" solution for x' in node i is infeasible when x' is set as the right-hand side in node j.) Therefore, `SDDP.jl` contains a number of routines that attempt to detect and fix some numerical issues, and `SDDP.jl` warns users who provide models with poor numerical properties.

6. Comparison with other libraries

`SDDP.jl` is not the only library for solving multistage stochastic programming problems using SDDP. We briefly describe seven alternatives and contrast their abilities to `SDDP.jl` A summary can be found in the Appendix. Five of the alternatives are open-source: FAST (Cambier and Scieur 2019), msppy (Ding et al. 2019), StOpt (Gevret et al. 2016), StochDynamicProgramming.jl (Leclère et al. 2019), and StructDualDynProg.jl (Legat 2019). The remaining two are commercial software: QUASAR (Quantego 2019), and the seminal SDDP by PSR (PSR 2019). (PSR-SDDP is a single-purpose implementation for hydro-thermal scheduling; we include it for comparison because it arose from the original implementation of Pereira and Pinto (1991)).

Perhaps the most interesting observation is that half of the implementations are in Julia (`SDDP.jl`, StochDynamicProgramming, StochDualDynProg.jl, and PSR's SDDP). An even more interesting observation is that each of the authors began to develop an SDDP library in Julia independently, and all within a few months of each other. This supports our belief that the combination of Julia's meta-programming and multiple dispatch abilities, as well as the JuMP (Dunning et al. 2017) package and wider JuliaOpt ecosystem, are the ideal foundation upon which to build an SDDP library.

When analysing features across different software, it is readily apparent that `SDDP.jl` is the most generic of the software, none the least because it is the only software to support arbitrary cyclic policy graphs. The software with the closest feature support to `SDDP.jl` is msppy, which can solve cyclic Markovian policy graphs and also implements the SDDiP method. However, msppy only supports the Gurobi optimizer (and therefore is restricted to quadratic, rather than general convex programs), and it cannot model non-Markovian policy graphs.

7. Conclusions

This paper introduced `SDDP.jl`, a state-of-the-art open-source solver for multistage stochastic programs. We believe the generality and ease-of-use of our library make it an ideal tool for multistage stochastic programming practitioners. Readers are directed to `https://github.com/odow/SDDP.jl` for source-code, examples, tutorials, and detailed documentation.

The roadmap for future development of `SDDP.jl` includes improvements to the numerical stability of the algorithm, and improvements to the way in which we handle general multistage stochastic mixed-integer programs.

Acknowledgments

`SDDP.jl` began development at the Electric Power Optimization Centre at the University of Auckland, and we especially thank Andy Philpott and Anthony Downward for their input. We also thank Joaquim Garcia, Vincent Leclère, Benoît Legat, Nils Löhndorf, and François Pacaud for discussing their implementations of SDDP. The first author was supported, in part, by Northwestern University's Center for Optimization & Statistical Learning (OSL).

References

Artzner P, Delbaen F, Eber JM, Heath D (1999) Coherent measures of risk. *Mathematical Finance* 9(3):203–228.

Asamov T, Powell WB (2018) Regularized decomposition of high-dimensional multistage stochastic programs with markov uncertainty. *SIAM Journal on Optimization* 28(1):575–595.

Bellman R (1957) *Dynamic Programming* (Princeton: Princeton University Press).

Bezanson J, Edelman A, Karpinski S, Shah VB (2017) Julia: A Fresh Approach to Numerical Computing. *SIAM Review* 59(1):65–98.

Birge JR (1985) Decomposition and Partitioning Methods for Multistage Stochastic Linear Programs. *Operations Research* 33(5):989–1007.

Cambier L, Scieur D (2019) FAST. https://github.com/leopoldcambier/FAST, [Online; accessed 2019-12-18].

de Farias D, van Roy B (2003) The linear programming approach to approximate dynamic programming. *Operations Research* 51(6):850–865.

Ding L, Ahmed S, Shapiro A (2019) A Python package for multi-stage stochastic programming. *Optimization Online* .

Downward A, Dowson O, Baucke R (2020) Stochastic dual dynamic programming with stagewise-dependent objective uncertainty. *Operations Research Letters* 48:33–39.

Dowson O (2020) The policy graph decomposition of multistage stochastic programming problems. *Networks* 1–21, URL `http://dx.doi.org/10.1002/net.21932`.

Dowson O, Morton DP, Pagnoncelli B (2019) Partially observable multistage stochastic programming. *Optimization Online* Http://www.optimization-online.org/DB_HTML/2019/03/7141.html.

Dunning I, Huchette J, Lubin M (2017) JuMP: A Modeling Language for Mathematical Optimization. *SIAM Review* 59(2):295–320.

Gevret H, Langrené N, Lelong J, Warin X (2016) STochastic OPTimization library in C++. Technical report, EDF.

Girardeau P, Leclère V, Philpott AB (2015) On the Convergence of Decomposition Methods for Multistage Stochastic Convex Programs. *Mathematics of Operations Research* 40(1):130–145.

Guan Z (2008) *Strategic Inventory Models for International Dairy Commodity Markets*. PhD Thesis, University of Auckland, Auckland, New Zealand.

Guigues V (2016) Convergence Analysis of Sampling-Based Decomposition Methods for Risk-Averse Multistage Stochastic Convex Programs. *SIAM Journal on Optimization* 26(4):2468–2494.

Helseth A, Braaten H (2015) Efficient Parallelization of the Stochastic Dual Dynamic Programming Algorithm Applied to Hydropower Scheduling. *Energies* 8(12):14287–14297.

Howard R (1960) *Dynamic Programming and Markov Processes* (Cambridge, MA.: MIT Press).

Leclère V, Pacaud F, Rigaut T, Gerard H (2019) StochDynamicProgramming.jl. https://github.com/JuliaOpt/StochDynamicProgramming.jl, [Online; accessed 2019-12-18].

Legat B (2019) StructDualDynProg.jl. https://github.com/blegat/StructDualDynProg.jl, [Online; accessed 2019-12-18].

Maceira M, Penna D, Diniz A, Pinto R, Melo A, Vasconcellos C, Cruz C (2018) Twenty years of application of stochastic dual dynamic programming in official and agent studies in brazil-main features and improvements on the newave model. *2018 Power Systems Computation Conference (PSCC)*, 1–7 (IEEE).

Maceira MEP, Duarte V, Penna D, Moraes L, Melo A (2008) Ten years of application of stochastic dual dynamic programming in official and agent studies in brazil-description of the newave program. *16th PSCC, Glasgow, Scotland* 14–18.

Ourani KI, Baslis CG, Bakirtzis AG (2012) A Stochastic Dual Dynamic Programming model for medium-term hydrothermal scheduling in Greece. *Universities Power Engineering Conference (UPEC), 2012 47th International*, 1–6 (IEEE).

Parpas P, Ustun B, Webster M, Tran QK (2015) Importance sampling in stochastic programming: A Markov chain Monte Carlo approach. *INFORMS Journal on Computing* 27(2):358–377.

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

Philpott AB, de Matos V, Kapelevich L (2018) Distributionally Robust SDDP. *Computational Management Science* 15(3-4):431–454.

Philpott AB, Guan Z (2008) On the convergence of sampling-based methods for multi-stage stochastic linear programs. *Operations Research Letters* 36:450–455.

Powell WB (2011) *Approximate Dynamic Programming: Solving the Curses of Dimensionality*. Wiley Series in Probability and Statistics (Hoboken, N.J: Wiley), 2nd ed edition.

Powell WB (2016) A Unified Framework for Optimization under Uncertainty. Gupta A, Capponi A, Smith JC, eds., *Optimization Challenges in Complex, Networked and Risky Systems*, 45–83, TutORials in Operations Research (INFORMS).

PSR (2019) Software — PSR. http://www.psr-inc.com/softwares-en/, [Online; accessed 2019-12-18].

Quantego (2019) QUASAR. http://quantego.com/, [Online; accessed 2019-12-18].

Reus L, Pagnoncelli B, Armstrong M (2019) Better management of production incidents in mining using multistage stochastic optimization. *Resources Policy* 63:101404.

Shapiro A (2011) Analysis of stochastic dual dynamic programming method. *European Journal of Operational Research* 209(1):63–72.

Zou J, Ahmed S, Sun XA (2019) Stochastic dual dynamic integer programming. *Mathematical Programming* 175(1-2):461–502.

Appendix

In this appendix, we provide a feature comparison between `SDDP.jl` and seven alternative libraries.

The comparison criteria are defined as follows:

1. License: What is the legal license under which the code is distributed?
2. Language: What is the main programming language in which the library is implemented?
3. Solver agnostic: Can the library be used with multiple solvers?
4. Subproblems: Can the subproblems be linear, quadratic, or arbitrary convex programs?
5. Policy Graph: What type of policy graph can the library solve? Options are linear, Markovian, and general.
6. Cyclic: Does the library support cyclic policy graphs?
7. Noise: Does the library support stagewise-independent noise terms? If no, then $|\Omega_i| = 1$. If "Any," then these terms can appear in the objective or in the constraints. If "RHS," then the noise terms can only appear in the right-hand side of the constraints.
8. Risk: Does the library support nested risk-measures? If "Custom," these can be user-defined.
9. Parallel: Does the library use parallel computing in its solution algorithm?
10. SDDiP: Does the library implement the algorithm of Zou et al. (2019)?

Name	SDDP.jl	FAST	msppy	StochDynamicProgram.jl
License	MPL-2.0	GPLv3	BSD-3	MPL-2.0
Language	Julia	MATLAB	Python	Julia
Solver agnostic	Yes	Yes	No	Yes
Subproblems	Convex	Linear	Quadratic	Linear
Policy Graph	General	Markovian	Markovian	Linear
Cyclic	Yes	No	Yes	No
Noise	Any	No	Any	Any
Risk	Custom	No	Yes	Custom
Parallel	Yes	No	Yes	No
SDDiP	Yes	No	Yes	No

Name	StOpt	StructDualDynProg.jl	QUASAR	PSR-SDDP
License	LGPLv3	MIT	Commercial	Commercial
Language	Julia	C++	Java	Julia
Solver agnostic	Yes	Yes	Yes	No
Subproblems	Linear	Convex	Quadratic	Quadratic
Policy Graph	Markovian	General	Markovian	Markovian
Cyclic	No	Yes†	No	No
Noise	RHS	No	Any	Any
Risk	No	No	Yes	Yes
Parallel	Yes	No	Yes	Yes
SDDiP	No	No	No	No

Table 1 SDDP library comparison. † StructDualDynProg.jl can support cyclic policy graphs if the problem has no objective function, i.e., it is pure feasibility.