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

O. Dowson*[a], L. Kapelevich[b]
[a] *Department of Engineering Science, University of Auckland, New Zealand.*
[b] *Operations Research Center, Massachusetts Institute of Technology, Cambridge, MA.*
*odow003@aucklanduni.ac.nz

Abstract

In this paper we present SDDP.jl, an open-source library for solving multi-stage stochastic optimization problems using the Stochastic Dual Dynamic Programming algorithm. 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 low-level languages. We benchmark the performance of SDDP.jl against a C++ implementation of SDDP for the New Zealand Hydro-Thermal Scheduling Problem. On the benchmark problem, SDDP.jl is approximately 30% slower than the C++ implementation. However, this performance penalty is small when viewed in context of the generic nature of the SDDP.jl library compared to the single purpose C++ implementation.

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 (Murtagh 1981) to modern algebraic modelling languages embedded in high-level languages such as JuMP (Dunning, Huchette, and Lubin 2015), or the 73-fold speed-up in solving difficult mixed-integer linear programs in seven years by Gurobi (Gurobi Optimization 2017)).

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 about how to best formulate a stochastic program (Powell 2016; Powell 2014). Moreover, when it comes to communicating the problem to the computer, various formats have been proposed (Birge et al. 1987; Gassmann and Kristjansson 2007), 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.

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 Pereira and Pinto (1991). 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 (Guan 2008), C++ (Philpott and Matos 2012; Helseth and Braaten 2015), GAMS (Ourani, Baslis, and Bakirtzis 2012; Bussieck, Ferris, and Lohmann 2012), JAVA (Asamov and Powell 2015) and MATLAB (Parpas et al. 2015), as well as in commercial products such as the seminal SDDP (PSR 2016) and DOASA (Philpott and Guan 2008). 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 there is a large up-front cost to development. As such, many researchers develop and test new algorithmic improvements without being able to easily compare their ideas against other implementations, and to the current state-of-the-art.

This paper presents `SDDP.jl` – a Julia package for solving multistage stochastic optimization problems using SDDP. The paper is split into two main sections. First, in Section 2, we introduce `SDDP.jl` and explain many of its features through a simple example. Second, in Section 3, we benchmark `SDDP.jl` against a C++ implementation of the SDDP algorithm for the New Zealand Hydro-Thermal Scheduling problem.

It is not the intention of the authors 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 Example: The Air-Conditioning Problem

To illustrate the many features of `SDDP.jl`, we consider the problem of producing air-conditioners over a period of three months. During standard working hours, the factory can produce 200 units per month at a cost of $100/unit. Unlimited overtime can be scheduled, however the cost increases to $300 per unit. In the first month, there is a known demand of 100 units. However, in months two and three, there is an equally likely demand of 100 or 300 units. Air-conditioners can be stored between months at a cost of $50/unit, and all demand must be met.

There are four steps to solving the air-conditioning problem. First, we must formulate it mathematically. Second, we need to communicate the formulation to the solver. Third, we need to solve the problem efficiently. Finally, we need to understand the solution. We now step through each of these steps, beginning with the problem formulation.

2.1 Formulating the problem

We can formulate the air-conditioning example as a discrete time, stochastic optimal control problem in a hazard-decision setting. To do so, it is necessary to define some important terms that will be used in the remainder of this paper.

A *stage* is a discrete unit in time when uncertainty is revealed (i.e. the demand), and decisions are made (i.e. how many units to produce). In the air-conditioning example, there

[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`.

are three stages (i.e. months).

A *state variable* is a piece of information that flows between stages. There is one state variable in the model: x_t, the quantity of air-conditioners in storage at the end of stage (i.e. month) t. x_t can also be thought of as the quantity of air-conditioners in storage at the start of stage $t+1$.

A *control variable* is an action or decision taken by the agent during a stage. There are three control variables in the model. p_t is the number of air-conditioners produced during standard working hours during month t, o_t is the number of air-conditioners produced during overtime working hours during month t, and s_t is the number of air-conditioners sold during month t.

A *noise* is a random variable that gets observed at the start of each stage. There is one random variable in the model: ω_t, the demand for air-conditioners in month t. We say that the model is a Hazard-Decision (also called Wait-and-See) model as the noise is realized at the start of the stage before the controls are chosen.

The *value-to-go* in a stage t is the expected future value that can be obtained from stages $t+1$ until the end of the time horizon, assuming that the agent takes the optimal control in each stage. For the air-conditioning problem, the value-to-go in stage t, V_t, given x_t units in storage and an observed demand of ω_t units can be expressed as the optimal objective value of the optimization problem:

$$\begin{aligned} V_t(x_t, \omega_t) = \max \quad & 100p_t + 300o_t + 50x_t + \mathbb{E}[V_{t+1}(x_t, \omega_{t+1})] \\ \text{s.t.} \quad & x_t + p_t + o_t - s_t = x_{t+1} \\ & s_t = \omega_t \\ & 0 \le p_t \le 200 \\ & x_t, x_{t+1}, o_t, s_t \ge 0. \end{aligned} \tag{1}$$

In each stage, ω is sampled from a finite discrete distribution so that $\omega_1 = 100$ with probability 1, and both ω_2 and $\omega_3 = 100$ with probability 0.5, and 300 with probability 0.5. In addition, $V_4(\cdot, \cdot) = 0$. We seek the solution to the first stage problem $\mathbb{E}[V_1(0, \omega_1)]$.

2.2 Communicating the problem to the solver

Before we introduce `SDDP.jl`, we must first introduce JuMP (Dunning, Huchette, and Lubin 2015), an algebraic modelling language for the Julia programming language (Bezanson et al. 2017) that we use as the basis for creating and manipulating the subproblems in the SDDP algorithm. 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 how to communicate the formulation of the air-conditioning problem (Eq. 1) to `SDDP.jl`. We give the complete code as an appendix item in Listing 1. The file can be run by saving the file to the disk, opening a Julia REPL, and then running:

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

However before we can run the example, we need to install `SDDP.jl` (this requires Julia 0.5 or later):

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

We now walk through the file and explain points of interest.

Loading packages In Line 1, we load the relevant packages needed by the example: SDDP, JuMP, and Clp. Users are free to use any of the other solvers in the JuliaOpt ecosystem that support MathProgBase instead of Clp (see Dunning, Huchette, and Lubin 2015 for more details).

The SDDPModel object The core type in `SDDP.jl` is the `SDDPModel` object. This is analogous to the JuMP `Model` object. The constructor for the `SDDPModel` object has the following form:

```
m = SDDPModel(keyword arguments...) do sp, t
    ... Eq. 1 definition ...
end
```

There are two key features to discuss. First, we describe the `keyword arguments...`. There are many optional keyword arguments which we shall not discuss as they are not necessary for our simple example. However, the four given in Listing 1 are required. They are `stages`, the number of stages in the model; `objective_bound`, a known lower bound (if minimising) on the first stage problem; `sense`, `:Min` or `:Max`; and `solver`, any MathProgBase compatible solver.

Second, the unusual `do sp, t ... end` syntax. This syntax is a Julia construct that is equivalent to writing:

```
function foo(sp::JuMP.Model, t::Int)
    ... Eq. 1 definition ...
end
m = SDDPModel(foo, keyword arguments...)
```

The arguments *sp* and *t* can be given arbitrary names, however the first argument will be an empty JuMP model that we use to build each subproblem (i.e. Eq. 1), and the second argument is an index that runs from 1 to the number of stages (in this case, three).

We now describe how to construct the subproblem *sp* using a mix of JuMP functionality, and `SDDP.jl` specific functions.

State Variables `SDDP.jl` provides a new macro called `@state` that can be used to add state variables to the subproblem *sp*. It takes three arguments: the first is the JuMP model *sp*, the second is an expression for the outgoing state variable (i.e. x_{t+1} in Eq. 1), and the third is one for the incoming state variable (i.e. x_t). The second argument can be any valid JuMP syntax for the `@variable` macro. The third argument must be an arbitrary name for the outgoing variable, followed by `==`, and then the value of the state variable in the first stage (i.e. x_1). For the air-conditioning example, we can create the state variable x_t as:

```
@state(sp, xt_1 >= 0, xt == 0)
```

The reader should note that the syntax of the third argument is just a convenient way of specifying the initial condition for the state variable. `SDDP.jl` does not enforce the constraint in every stage. In constraints and in the objective (detailed below), state variables (i.e. `xt` and `xt_1`) behave just like any other JuMP variable.

Control Variables Recall that *sp* is a standard JuMP model. Therefore, the user is free to add any JuMP variables to the model via the `@variable` and `@variables` macros. In the air-conditioning example, we add the number of units to produce during standard production hours (`pt`), the number of units to produce during overtime production (`ot`), and the number of air-conditioners to sell (`st`):

```
@variables(sp, begin
    0 <= pt <= 200
         ot >= 0
         st >= 0
end)
```

All controls are non-negative, and the standard production capacity (`pt`) has an upper bound of 200 units.

Dynamics The user is also free to add any arbitrary JuMP constraints via the `@constraint` macro. In this case, we add the balance constraint that the number of air-conditioners in storage at the end of a month (x_{t+1}) is the number in storage at the start of the month (x_t), plus any production (p_t) and overtime production (o_t), less any sales (s_t):

```
@constraint(sp, xt + pt + ot - st == xt_1)
```

Uncertainty `SDDP.jl` supports random variables in the right-hand-side of constraints. Instead of using `@constraint` to add a constraint with a random variable, `SDDP.jl` provides the macro `@rhsnoise`. This macro takes three arguments. The first is the JuMP model *sp*. The second is a keyword argument of the form `name = realizations`, where `name` is an arbitrary name for the random variable, and `realizations` is a vector containing the finite discrete number of realizations that the random variable can take. The third argument is any valid JuMP `@constraint` syntax that contains `name` as part of the right-hand-side term (i.e. not a variable coefficient). For the air-conditioning problem, the agent must sell exactly the quantity of units demanded. Therefore, we add the `@rhsnoise` constraint:

```
D = [ [100], [100, 300], [100, 300] ]
@rhsnoise(sp, wt=D[t], st == wt)
```

where $D[t]$ is the list of possible demand realizations in stage t. We can set the probability of the demand realizations in stage t using the `setnoiseprobability!` function:

```
P = [ [1.0], [0.5, 0.5], [0.5, 0.5] ]
setnoiseprobability!(sp, P[t])
```

Readers should note that `SDDP.jl` does *not* support uncertainty in the constraint coefficients. This a known limitation and will hopefully be resolved in a future release.

The stage objective All that remains is to define the objective of the stage problem. `SDDP.jl` handles the $\mathbb{E}[V_{t+1}(x_t, \omega_{t+1})]$ term behind the scenes, so the user only has to provide the immediate cost via the `@stageobjective` macro:

```
@stageobjective(sp, 100 * pt + 300 * ot + 50 * xt)
```

We now describe how to solve the model and visualize the solution.

2.3 Solving the problem

As we have previously mentioned, one area in which stochastic programming has made significant progress is in the efficient computational solution of the problem. `SDDP.jl` includes many of the state-of-the art features that have appeared in the literature. In addition, we heavily utilize Julia's type-system and ability to overload generic methods to provide an interface that is extensible by the user without having to modify the package's source-code. Due to space constraints, we are precluded from providing a detailed discussion of each individual feature. However, three of the most important are:

1. User-defined cut selection routines: without modifying the `SDDP.jl` source-code, users are able to define new cut-selection heuristics (See Matos, Philpott, and Finardi (2015).) to reduce the size of the linear subproblems.

2. User-defined risk measures: without modifying the `SDDP.jl` source-code, users are able to define new risk measures that seamlessly integrate with the entire `SDDP.jl` library. This functionality has been used by Philpott, de Matos, and Kapelevich (2017) to develop a distributionally-robust SDDP algorithm without having to implement the full SDDP algorithm.

3. Parallel solution process: `SDDP.jl` leverages Julia's built-in parallel functionality to perform SDDP iterations in parallel. `SDDP.jl` has been successfully scaled from the single core of a laptop to tens of cores in high-performance computing environments without the user needing to modify a single line of code.

In Line 23 of Listing 1, we solve the air-conditioning problem using the SDDP algorithm. There are many different options that can be passed to the solve command, so in the interests of conciseness, we omit them here. However, the argument `max_iterations` causes the algorithm to terminate after the provided number of iterations have been conducted. Once the solution process has terminated, we can query the bound on the solution using the function `getbound(m)`. This returns the optimal solution (which can be verified by solving the deterministic equivalent) of \$62,500.

2.4 Understanding the solution

Unlike deterministic mathematical programming, SDDP does not provide an explicit optimal value for every variable. Instead, it constructs a *policy* in the form of a linear program that approximates Eq. 1. To obtain the optimal control for a stage, the user sets the value of the incoming state variable x_t, and the realization of the random variable ω_t, and solves the approximated linear program. It can be difficult to understand the policy due to the uncertainty, and the large dimensionality of the state and control spaces. Therefore, one commonly used approach is to use Monte Carlo simulation. In `SDDP.jl`, this can be done using the `simulate` function:

```
sims = simulate(m, 40, [:xt_1, :pt, :ot, :st])
```

The first argument is the `SDDPModel` `m`. The second argument is the number of independent scenarios to simulate, and the third argument is a list of variable values to record at each stage. For the air-conditioning problem, we record `xt_1`, the number of air-conditioners in storage at the end of stage t, as well as the three control variables (standard production `pt`, overtime production `ot`, and sales `st`). `sims` is a vector of dictionaries (one element for each simulation), and can be manipulated or saved to a file for later analysis. For example, the user can query

the number of units produced during standard production hours during the second month in the tenth Monte Carlo simulation by:

```
julia> sims[10][:pt][2]
```

In Figure 1, we plot four of the Monte Carlo simulations (chosen as they sample different demand realizations) for the four variables recorded by `simulate`. In all scenarios, 200 units are produced during normal production hours during the first stage (Figure 1b). This is despite the fact that the demand of 100 units is known ahead of time. Therefore, 100 units are in storage at the end of the first stage (Figure 1c). If the demand (Figure 1a) is high (i.e. 300 units) in the second stage, then production remains at 200 units per stage. In addition, 100 units are sold from storage to meet the demand of 300 units. If demand is low (i.e. 100 units) in the second stage, then standard production drops to 100 units and no units are sold from storage. In all scenarios, there is no overtime during the first two stages (Figure 1d).

At the beginning of the third stage, the system can be in one of two states: with 100 units in storage if the previous stage's demand was low, or with zero units in storage if the previous stage's demand was high. If there are zero units in storage, and the demand in the third stage is high, then the optimal solution is to produce 200 units during standard production hours, and 100 units during overtime hours. In all other cases, it is possible to meet the demand using a combination of the units in storage and standard production.

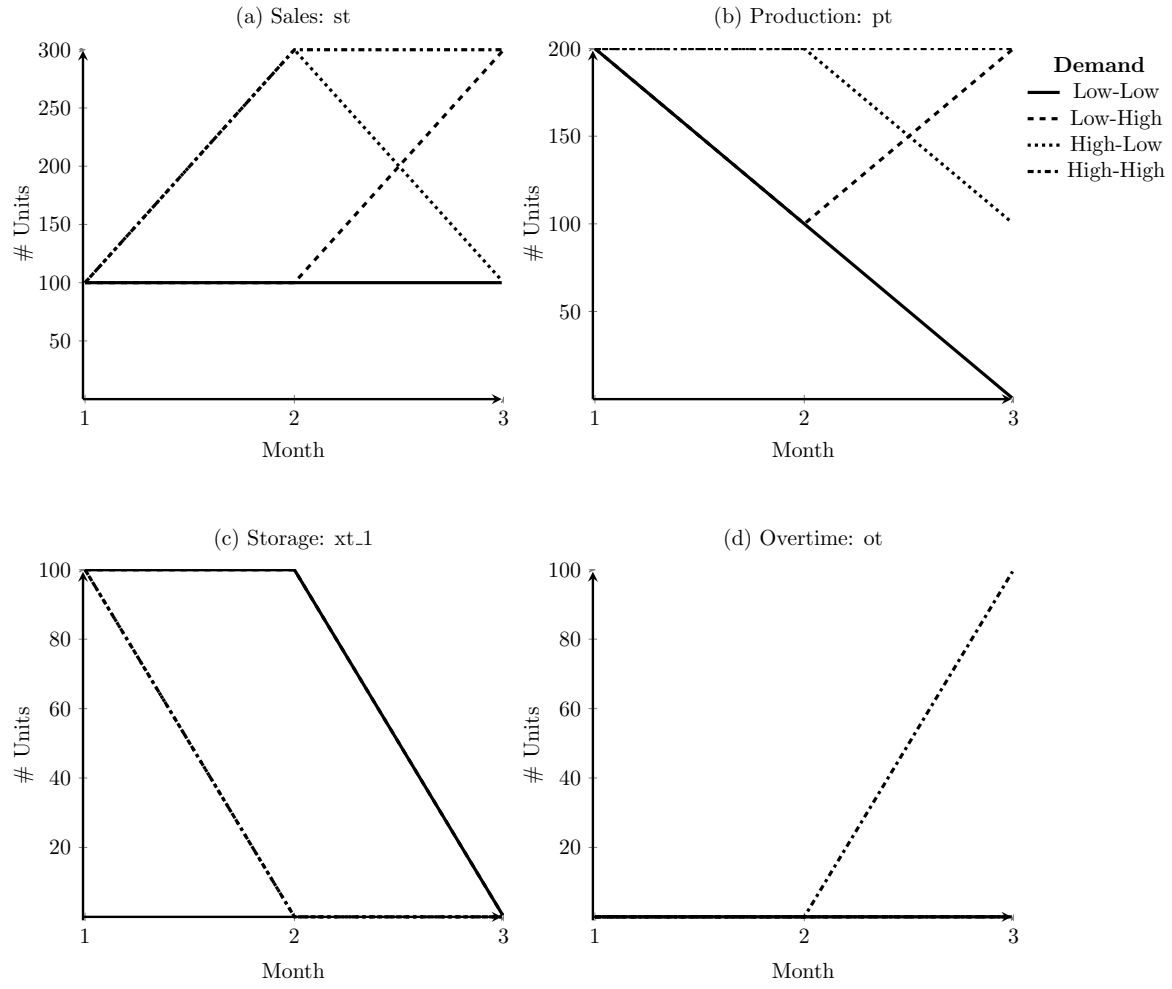

Figure 1: Four simulations (each sampling a different demand scenario) of the air-conditioning problem using the optimal policy.

In addition to the Monte Carlo simulation functionality, `SDDP.jl` provides a number of Javascript plotting tools to help the user understand, and interact with, the solution. One of those tools automates the plotting of the Monte Carlo simulation results like those shown in Figure 1. However, for brevity, we direct the reader to the online documentation for more information,

3 Benchmark: Hydro-Thermal Scheduling

In the previous section, we showed how to implement a simple multi-stage stochastic program using `SDDP.jl`. Now, in this section, we use a more complicated model to benchmark the performance of `SDDP.jl` against an existing C++ implementation of the SDDP algorithm.

The most common application of the SDDP algorithm (dating back to the original paper of Pereira and Pinto (1991)) is the Hydro-Thermal Scheduling Problem (HTSP). In this problem, an operator owns a number of hydro-reservoirs that are connected by a series of rivers and hydro-electric generators. Water can be released from the reservoirs and turbined to produce electricity. However, future inflows into the reservoirs from rainfall or ice-melt are uncertain. Any un-met electricity demand is met by thermal generation. Therefore objective of the operator is to find a strategy for controlling the release of water over a planning horizon that minimizes the total cost of thermal generation.

Software for solving this problem using the SDDP algorithm has been successfully commercialized by the Brazilian company PSR (PSR 2016).[2] However for this paper, we used `DOASA`, a C++ implementation of the SDDP algorithm for the New Zealand HTSP.[3] In contrast to the generic `SDDP.jl`, `DOASA` is hard-coded to solve the New Zealand Hydro-Thermal scheduling problem. This allows it to implement advanced performance optimisations that are not possible in `SDDP.jl` with the current version of JuMP.[4] In particular, `DOASA` implements a form of cut-selection (see Matos, Philpott, and Finardi 2015) which, instead of periodically rebuilding the constraint matrix with a subset of the discovered cuts (as `SDDP.jl` does), modifies the constraint coefficients of a cut that is scheduled to be removed with the most recent cut coefficients. This minimizes the number of constraints added to the model, avoids having to periodically rebuild the constraint matrix, and enables a more efficient hot-start of the subsequent solve (compared to the solve after the matrix rebuild which has no basis information). Our copy of `DOASA` was provided by the Electric Power Optimization Centre at the University of Auckland, New Zealand.

To implement the `SDDP.jl`[5] version of the New Zealand HTSP, we referred only to the description of the model given in Philpott and Pritchard (2013). At no point did we refer to any of the C++ source code. We now test the correctness and performance of `SDDP.jl` by comparing it to `DOASA`.

3.1 Correctness

To test the correctness of `SDDP.jl` and HTSP model, five deterministic instances of the New Zealand HTSP were created using historic inflows, demand, and pricing for the years 2005 – 2009. These problems were solved for 100 SDDP iterations using `DOASA` and `SDDP.jl`. In all years, both implementations converged to identical lower bounds (Table 1). This experiment

[2] Somewhat confusingly, the software is named SDDP

[3] To add to the naming confusion, the term DOASA was also used to refer to a class of algorithms related to SDDP (the algorithm) by Philpott and Guan (2008). We shall refer to `DOASA` the software by stylizing it in typewriter font.

[4] This may change with JuMP v0.19.

[5] Further adding to the naming confusion

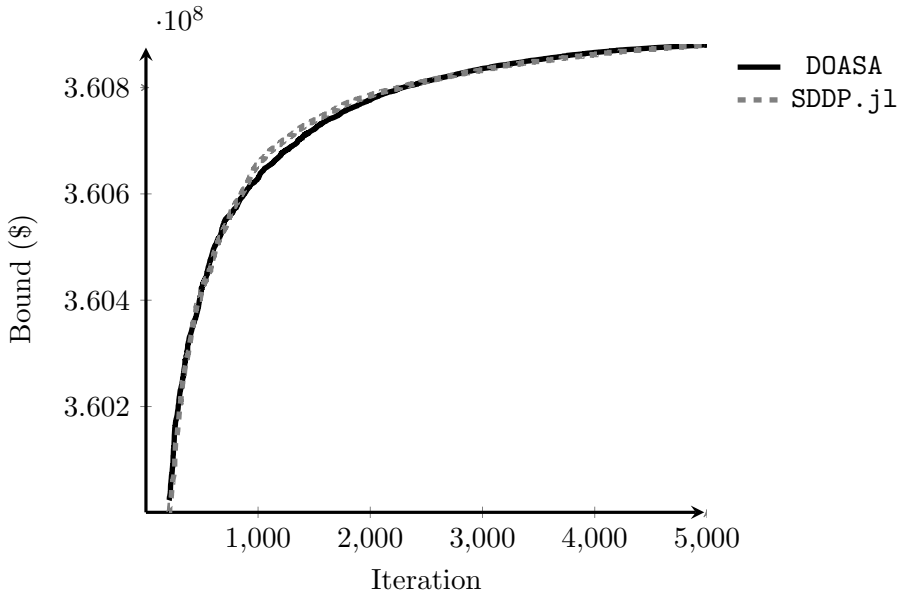

Figure 2: Lower bound convergence of `DOASA` and `SDDP.jl` on stochastic instance of the New Zealand HTSP.

strongly suggests that the two implementations of the model and deterministic SDDP algorithm are identical.

	Year				
	2005	2006	2007	2008	2009
`DOASA`	493,125,281	423,420,729	575,859,349	446,507,222	340,096,459
`SDDP.jl`	493,125,281	423,420,729	575,859,349	446,507,222	340,096,459

Table 1: Lower bound ($) after 100 SDDP iterations.

To test the correctness of `SDDP.jl` and `DOASA` on a stochastic problem, an instance of the New Zealand HTSP was created using historic inflows from 1970–2007 and demand and pricing data from 2008. This problem was solved for 5000 SDDP iterations using both implementations. In Figure 2, we plot the lower bound against the number of iterations for both implementations. Due to the different random number generators used by `DOASA` and `SDDP.jl`, different random inflows are sampled. This can lead to a difference in the bound at any particular iteration. However, we see clear evidence that both implementations converge towards an identical bound at approximately the same rate. When combined with the deterministic experiments, there is very strong evidence that the SDDP algorithms in `DOASA` and `SDDP.jl` are correct, and the implementations of the New Zealand HTSP are identical.

To the best of our knowledge, this is the first time that the correctness of a SDDP implementation in a real-world setting has been demonstrated in the literature.

3.2 Performance

To compare the performance of `SDDP.jl` and `DOASA`, an instance of the New Zealand HTSP was created using 38 years of historic inflows (1970–2007), with data from 2008 for demand and thermal pricing. Four different solver configurations were setup: `DOASA` using Gurobi version 6.5.0 (Gurobi 2017a), `SDDP.jl` using Gurobi version 6.5.0, `SDDP.jl` using Gurobi version 7.0.0 (Gurobi 2017b), and `SDDP.jl` using CPLEX version 12.6.1 (IBM 2017). The problem was solved 20 times for each of the configurations. The SDDP algorithm terminated after 500 cuts had

been generated for the first stage problem. All experiments were conducted on a Windows 7 machine with an Intel i7-4770 CPU and 16GB of memory. All solvers used the default parameter settings.

In Table 2, we summarize the results of these experiments. There are four columns of interest: *LP Optimize*, *JuMP*, *SDDP*, and *Total*. In the *LP Optimize* column, we report the time spent in the external solver libraries solving the LP subproblems. In the *JuMP* column, we report the time that `SDDP.jl` spends in calls to the JuMP method *solve*, excluding the call to the LP solver. This measures the overhead of solving the problem via the generic JuMP interface rather than directly through the solver API. In the *SDDP* column we measure the time that is spent performing tasks related to the SDDP algorithm except the actual solving of the subproblems. These include sampling the random variables, calculating cut coefficients, and writing the solutions to file. We have aggregated the *JuMP* and *SDDP* columns for the `DOASA` configuration as it does not use JuMP. Finally, in the *Total* column, we report the total time spent solving the 500 iterations.

		Time (s)			
	Solver	LP Optimize	JuMP	SDDP	Total
`DOASA`	Gurobi v6.5.0	458.1 (1.1)	60.5 (0.8)		518.6 (1.4)
`SDDP.jl`	Gurobi v6.5.0	559.0 (1.0)	68.7 (0.7)	41.5 (0.5)	669.2 (1.5)
`SDDP.jl`	Gurobi v7.0.0	580.2 (1.1)	69.8 (0.7)	41.6 (0.6)	691.6 (1.4)
`SDDP.jl`	CPLEX v12.6.1	319.6 (3.6)	75.9 (1.4)	41.3 (0.9)	436.8 (5.4)

Table 2: Solution time after 500 iterations. All values are reported as the mean (standard deviation) of twenty repetitions.

After 500 iterations, the lower bound of all configurations was similar (∼$360.4 million). However, solution times varied between `DOASA` and `SDDP.jl`, Gurobi and CPLEX, and even between different Gurobi versions. Of the four configurations, `SDDP.jl` with CPLEX v12.6.1 performed the fastest (mean of 436.8 seconds to solve 500 iterations), followed by `DOASA` (518.6 seconds), `SDDP.jl` with Gurobi v6.5.0 (669.2 seconds), and `SDDP.jl` with Gurobi v7.0.0 (691.6 seconds).

The `SDDP.jl` configuration with Gurobi v6.5.0 spent 22% longer in the LP optimizer than `DOASA` with Gurobi v6.5.0 despite solving an identical number of linear programs. This is due to the advanced cut selection routines implemented in `DOASA` which aggressively minimize the size of the subproblems. Interestingly, Gurobi v7.0.0 is 3.7% slower than Gurobi v6.5.0. This may be a sign that Gurobi is focused on improving the performance of difficult problems that take a long time to solve at the expense of small LP's (On average, the subproblems in the New Zealand HTSP take on the order of $50\mu s$ to solve).

Despite solving larger subproblems, `SDDP.jl` with CPLEX v12.6.1 spent 28% less time in the LP optimizer compared to `DOASA` with Gurobi v6.5.0. If the same cut selection routines were implemented in `SDDP.jl`, this suggests that the total solution time could be improved further. It may also be possible to improve the performance of both solvers by using non-default settings.

The use of JuMP adds 10–20% to the total solution time. However, this allows the user to specify the subproblems using the simple input syntax described in the previous section. In contrast, `DOASA` builds the constraint matrix directly using the Gurobi C++ API. This significantly increases the development time needed to modify and debug the stage problems (for example, adding a new constraint).

Finally, less than 10% of the total solution time is spent performing tasks in the `SDDP.jl` library. This highlights the efficiency of the Julia implementation and demonstrates that further performance gains are more likely to be found by reducing the solve time of the individual

subproblems, instead of improving the code in the `SDDP.jl` library.

4 Conclusion

This paper introduced `SDDP.jl`, a Julia package for Stochastic Dual Dynamic Programming. In addition to describing the convenient user-interface, we showed that for the New Zealand HTSP, the overhead of using JuMP and the generic library `SDDP.jl` compared to a hard-coded C++ implementation is less than the variance between two commercial LP solvers. We also gave a strong guarantee that the implementation is correct by showing that two independently developed implementations give identical solutions.

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.

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, and Vitor de Matos. Credit also goes to Vincent Leclère, François Pacaud, Tristan Rigaut, Henri Gerard, and Benoît Legat for their work in creating other implementations of the SDDP algorithm in Julia.

References

Asamov, Tsvetan and Warren B. Powell (2015). "Regularized Decomposition of High-Dimensional Multistage Stochastic Programs with Markov Uncertainty". In: *arXiv preprint arXiv:1505.02227*. URL: `http://arxiv.org/abs/1505.02227` (visited on 05/13/2016).

Bezanson, Jeff et al. (2017). "Julia: A Fresh Approach to Numerical Computing". In: *SIAM Review* 59.1, pp. 65–98. URL: `http://epubs.siam.org/doi/10.1137/141000671`.

Birge, John R. et al. (1987). *A standard input format for multiperiod stochastic linear programs*. IIASA Working Paper. IIASA, Laxenburg, Austria: WP-87-118.

Bussieck, Michael R., Michael C. Ferris, and Timo Lohmann (2012). "GUSS: Solving collections of data related models within GAMS". In: *Algebraic Modeling Systems*. Springer, pp. 35–56.

Dunning, Iain, Joey Huchette, and Miles Lubin (2015). "JuMP: A modeling language for mathematical optimization". In: *arXiv:1508.01982 [math.OC]*. URL: `http://arxiv.org/abs/1508.01982`.

Gassmann, H. I. and B. Kristjansson (2007). "The SMPS format explained". In: *IMA Journal of Management Mathematics* 19.4, pp. 347–377.

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

Gurobi (2017a). *Gurobi 6.5 Reference Manual*. [Online; accessed 2017-11-02]. URL: `http://www.gurobi.com/documentation/6.5/refman/index.html`.

— (2017b). *Gurobi 7.0 Reference Manual*. [Online; accessed 2017-11-02]. URL: `http://www.gurobi.com/documentation/7.0/refman/index.html`.

Gurobi Optimization (2017). *Gurobi 7.5 Performance Benchmarks*. Tech. rep. Gurobi Optimization, Inc. URL: `https://gurobi.com/pdfs/benchmarks.pdf`.

Helseth, Arild and Hallvard Braaten (2015). "Efficient Parallelization of the Stochastic Dual Dynamic Programming Algorithm Applied to Hydropower Scheduling". In: *Energies* 8.12, pp. 14287–14297.

IBM (2017). *CPLEX Users Manual*. [Online; accessed 2017-11-02]. URL: `https://www.ibm.com/support/knowledgecenter/SSSA5P_12.6.1/ilog.odms.studio.help/pdf/usrcplex.pdf`.

Matos, Vitor L. de, Andy B. Philpott, and Erlon C. Finardi (2015). "Improving the performance of Stochastic Dual Dynamic Programming". In: *Journal of Computational and Applied Mathematics* 290, pp. 196–208.

Murtagh, Bruce A (1981). *Advanced linear programming: computation and practice*. McGraw-Hill International Book Co.

Ourani, Konstantina I., Costas G. Baslis, and Anastasios G. Bakirtzis (2012). "A Stochastic Dual Dynamic Programming model for medium-term hydrothermal scheduling in Greece". In: *Universities Power Engineering Conference (UPEC), 2012 47th International*. IEEE, pp. 1–6.

Parpas, Panos et al. (2015). "Importance sampling in stochastic programming: A Markov chain Monte Carlo approach". In: *INFORMS Journal on Computing* 27.2, pp. 358–377.

Pereira, M.V.F. and L.M.V.G. Pinto (1991). "Multi-stage stochastic optimization applied to energy planning". In: *Mathematical Programming* 52, pp. 359–375.

Philpott, A. B. and Z. Guan (2008). "On the convergence of sampling-based methods for multi-stage stochastic linear programs". In: *Oper Res Lett* 36, pp. 450–455.

Philpott, Andrew B. and Vitor L. de Matos (2012). "Dynamic sampling algorithms for multi-stage stochastic programs with risk aversion". In: *European Journal of Operational Research* 218.2, pp. 470–483.

Philpott, Andy, Vitor de Matos, and Lea Kapelevich (2017). *Distributionally Robust SDDP*. Tech. rep. Auckland, New Zealand: Electric Power Optimization Centre. URL: http://www.epoc.org.nz/papers/DROPaperv52.pdf.

Philpott, Andy and Geoffrey Pritchard (2013). *EMI-DOASA*. Tech. rep. Stochastic Optimization Limited. URL: http://www.emi.ea.govt.nz/Content/Tools/Doasa/DOASA\%20paper\%20by\%20SOL.pdf (visited on 05/09/2016).

Powell, Warren B. (2014). "Clearing the Jungle of Stochastic Optimization". In: *Bridging Data and Decisions*. Ed. by Alexandra M. Newman et al. INFORMS, pp. 109–137. URL: http://pubsonline.informs.org/doi/abs/10.1287/educ.2014.0128.

— (2016). "A Unified Framework for Optimization under Uncertainty". In: *Optimization Challenges in Complex, Networked and Risky Systems*. Ed. by Aparna Gupta and Agostino Capponi. TutORials in Operations Research. INFORMS, pp. 45–83. URL: http://castlelab.princeton.edu/Papers/Powell-UnifiedFrameworkforStochasticOptimization_April162016.pdf.

PSR (2016). *Software | PSR*. [Online; accessed 2017-11-02]. URL: http://www.psr-inc.com/softwares-en/.

A Code

Listing 1: Air-Conditioning Example

```
using JuMP, SDDP, Clp

m = SDDPModel(
                stages = 3,
    objective_bound = 0.0,
                sense = :Min,
                solver = ClpSolver()
                                ) do sp, t
    @state(sp, xt_1 >= 0, xt == 0)
    @variables(sp, begin
        0 <= pt <= 200
                ot >= 0
                st >= 0
    end)
    @constraint(sp, xt  + pt + ot - st == xt_1)
    D = [ [100], [100, 300], [100, 300] ]
    @rhsnoise(sp, wt=D[t], st == wt)
    P = [ [1.0], [0.5, 0.5], [0.5, 0.5] ]
    setnoiseprobability!(sp, P[t])
    @stageobjective(sp, 100 * pt + 300 * ot + 50 * xt)
end

status = solve(m, max_iterations=10)
getbound(m) # should be 62,500

sims = simulate(m, 100, [:xt_1, :pt, :ot, :st])
```