Optimization Online manuscript No.
(will be inserted by the editor)

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

Oscar Dowson · Lea Kapelevich

Received: date / Accepted: date

Abstract In this paper we present SDDP.jl, an open-source library for solving multistage 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.

Keywords julia · stochastic dual dynamic programming

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. For example, consider the evolution from early file-formats such as MPS [25] to modern algebraic modelling languages embedded in high-level languages such as JuMP [11], or the 73-fold speed-up in solving difficult mixed-integer linear programs in seven years by Gurobi [16].

O. Dowson
Department of Engineering Science at The University of Auckland, Auckland, New Zealand.
E-mail: o.dowson@auckland.ac.nz

L. Kapelevich
Operations Research Center at the Massachusetts Institute of Technology, Cambridge, MA, USA.

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 [40, 39]. Moreover, when it comes to communicating the problem to the computer, various formats have been proposed [6, 12], 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 solver 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 [31].

SDDP is a dynamic programming-inspired algorithm. It decomposes the multistage stochastic optimization problem in time into a series of sequential subproblems. Each subproblem (i.e. a discrete interval in time) is an optimization problem that chooses an action in order to minimize the a cost associated with the current decision, plus the *cost-to-go* of the remaining stages given the action is taken. Traditional dynamic programming [4] estimates the cost-to-go function (also called the Bellman function) at a set of discretized points. However, because of this discretization, the method is limited so low-dimensional problems (dynamic programming's "curse of dimensionality"). Instead of evaluating the function at a set of discretized points, SDDP approximates the Bellman function with a set of piecewise linear functions called "cuts." When the problem instance has a specific form (stagewise independence of the random variable, convexity of the Bellman function, and continuous state variables), the SDDP algorithm can efficiently find an optimal policy (to within some bound). In this paper, we assume that the reader is familiar with the method. We direct readers that are unfamiliar with the method to the following works: [37, 43, 44]. We can also recommend the accessible introductions to SDDP provided by [27, Ch. 4] and [14, Ch. 5].

Since the paper of [31], 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, generic implementations of the algorithm existed in the public domain (we discuss other implementations in Section 5). 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 [14], C++ [34, 18], GAMS [28, 7], Java [1], and MATLAB [30], as well as in commercial products such as the seminal SDDP [41], QUASAR [42], and DOASA [37]. In our opinion, this "re-invention of the wheel" has limited the adoption of the SDDP algorithm in areas outside 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, or 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 laid out as follows. First, in Section 2, we introduce `SDDP.jl` and explain many of its basic features through a

simple example. Then, in Section 4.2, we detail some of the more advanced features of `SDDP.jl`. In Section 4, we benchmark `SDDP.jl` against a C++ implementation of the SDDP algorithm for the New Zealand Hydro-Thermal Scheduling problem. Finally, in Section 5, we compare `SDDP.jl` against other libraries that implement the SDDP algorithm.

It is not the intention of the authors to make this paper a comprehensive tutorial for Julia or SDDP. In places, familiarity is assumed with 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 air-conditioning problem proposed by [29]. In this problem, the manager of a factory seeks a production plan for 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 during those hours. In the first month, there is a known demand of 100 units. However, in each of months two and three, there is an equally likely demand of either 100 or 300 units. Air-conditioners can be stored between months at a cost of $50/unit, and all demand must be met.

We now step through each of the four steps in the solution process, 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 an interval in time when uncertainty is revealed (i.e. the demand becomes known), and decisions are made (i.e. how many units to produce). In the air-conditioning example, there are three stages (i.e. three 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 start of stage (i.e. month) t. x_t can also be thought of as the quantity of air-conditioners in storage at the end of stage $t-1$. In the remainder of this paper, we denominate x_t as the incoming state variable to stage t, and x_{t+1} as the outgoing state variable of stage t.

A *noise* is a stagewise independent random variable that is observed by the agent at the start of each stage. There is one noise 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.

A *control variable* is an action or decision taken by the agent during the 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.

The *value-to-go* (cost-to-go if minimizing) in 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 (Bellman's principle of optimality [3]). For the air-conditioning problem, the value-to-go at the start of 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:

$$\begin{aligned}\mathbf{SP}_t: \quad V_t(x_t,\omega_t) = \max_{p_t,o_t,s_t} \;& 100p_t + 300o_t + 50x_t + \mathbb{E}_{\omega_{t+1}}[V_{t+1}(x_{t+1},\omega_{t+1})] \\ \text{s.t. } & x_t + p_t + o_t - s_t = x_{t+1} \\ & s_t = \omega_t \\ & 0 \leq p_t \leq 200 \\ & x_t, x_{t+1}, o_t, s_t \geq 0,\end{aligned}$$

where, $V_4(\cdot,\cdot) = 0$. The agent chooses the controls p_t, o_t, and s_t to maximize the value of V_t. Although air-conditioners are discrete, to maintain convexity for SDDP, we solve the LP relaxation. However, as we shall later see, the solution is naturally integer. In each stage, ω_t is independently sampled from a finite discrete distribution so that:

$$\omega_1 = \begin{cases} 100, & \text{with probability } 1, \end{cases}$$

$$\omega_t = \begin{cases} 100, & \text{with probability } 0.5 \\ 300, & \text{with probability } 0.5 \end{cases} \qquad \text{for } t = 2, 3.$$

Therefore, the full optimization problem faced by the factory manager is:

$$\max \quad \mathbb{E}_{\omega_1}\left[V_1(0,\omega_1)\right]. \tag{1}$$

Henceforth, we shall refer to the optimization problem $\mathbf{SP}_t$ parameterized by x_t and ω_t as a *subproblem*. In the air-conditioning problem, there are three subproblems: $\mathbf{SP}_1$, $\mathbf{SP}_2$, and $\mathbf{SP}_3$.

To solve a problem using `SDDP.jl`, the user must express their problem in the form above. In addition to the typical requirements of the SDDP algorithm (convexity of the value-to-go function w.r.t. x_t, a finite discrete distribution of ω_t that is stagewise independent, etc.), `SDDP.jl` introduces two additional requirements. First, the noise ω_t cannot appear as a constraint coefficient. However, ω_t can appear anywhere in the objective function expression, or as the right-hand-side term in a constraint. Second, the uncertainty is revealed in a Hazard-Decision setting and not a Decision-Hazard (although a Decision-Hazard problem can be reformulated as a Hazard-Decision problem with the introduction of additional state variables).

In the next section, we describe how to communicate the formulation of the subproblems $\mathbf{SP}_t$ to `SDDP.jl`.

2.2 Communicating the problem to the solver

In order to introduce `SDDP.jl`, we must first introduce JuMP [11], an algebraic modelling language for the Julia programming language [5] 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 an 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 fast, yet generic, implementation of the SDDP algorithm.

Overview Before we dive into the specifics, we provide the reader with a brief overview of the design behind the user-interface of `SDDP.jl`. The driving principle is to represent each subproblem $\mathbf{SP}_t$ as a JuMP model parameterized by the incoming state variable x_t and the realization of the stagewise independent random variable ω_t, staying as close as possible to the mathematical form of $\mathbf{SP}_t$ described above. This is achieved by extending some of the features of JuMP to introduce state variables, and allow random variables in the right-hand side of constraints and in the objective. 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 expected future cost term in the objective, manages the realizations of the random variables, and passes the values of the state variables between stages.

We now describe how to communicate the formulation of the air-conditioning subproblems to `SDDP.jl`. We give the complete code as an appendix item in Listing 1. 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")
```

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")
```

In the remainder of this section, we walk through the file and explain points of interest in Listing 1.

Loading packages First, we load the relevant packages needed by the example: SDDP, JuMP, and Clp.

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

Users are free to use any of the other solvers in the JuliaOpt ecosystem that support MathProgBase instead of Clp (see [11] for more details).

The SDDPModel object The core type in `SDDP.jl` is the `SDDPModel` object. This stores a single JuMP model for each subproblem, as well as the information needed to parameterize each model based on the incoming state variable x_t and realization of the random variable ω_t. It also stores other information including the linkages between subproblems. The constructor for the `SDDPModel` object has the following form:

```
m = SDDPModel(keyword arguments...) do sp, t
    ... subproblem 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 minimizing, otherwise upper) on the first stage problem; `sense`, the optimization sense `:Min` or `:Max`; and `solver`, any MathProgBase compatible solver.

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

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

where `foo(sp, t)` is a function that takes an empty JuMP model `sp` and the stage `t`, and builds the subproblem $\mathbf{SP}_t$ using a mix of JuMP functionality, and `SDDP.jl` specific functions. We now describe how to construct the function `foo`.

Control Variables *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 these control variables are non-negative, and the standard production capacity (`pt`) has an upper bound of 200 units.

State Variables `SDDP.jl` provides a new macro called `@state` that can be used to add state variables to the subproblem *sp*. This macro is a variant of the JuMP `@variable` macro with a few differences. 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}), 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 second argument of 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, xtp1 >= 0, xt == 0)
```

This will add two standard JuMP variables (`xtp1` and `xt`) to the model `sp` that are visible to the user, as well as some additional information that is hidden from the user describing how to link the value of `xtp1` in stage t to `xt` in stage $t + 1$. The reader should note that the syntax of the third argument is how we specify the value of the state variable at the start of stage one. The following is also valid syntax and demonstrates how the index i is shared between `ytp1` and `yt`:

```
y1 = [0.0, 1.0] # initial value
@state(sp, 0 <= ytp1[i=1:2] <= 1, yt == y1[i])
```

This code will add four variables to the JuMP model `sp`: `ytp1[1]`, `ytp1[2]`, `yt[1]`, and `yt[2]`. In constraints and in the objective (detailed below), state variables (i.e. `xt` and `xtp1`) behave just like any other JuMP variable.

Constraints The user is also free to add any arbitrary JuMP constraints via the `@constraint` macro. For this example, 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 == xtp1)
```

Right-Hand Side Uncertainty `SDDP.jl` supports random variables in the right-hand side of constraints.[1] 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 list of realizations that the random variable can take. In our example, these realizations are just 100 in stage $t = 1$, and either 100 or 300 in stages $t = 2$ and $t = 3$. 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)
```

[1] As noted before, `SDDP.jl` does *not* support uncertainty in the constraint coefficients. This a known limitation and will hopefully be resolved in a future release.

where `D[t]` is the list of possible demand realizations in stage `t`. (Note that we use `t` which was defined in the *SDDPModel Object* section.) 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])
```

In this example, we used a list of lists to store the realization and probability data in a compact manner. However, the following is also valid:

```
if t == 1
    @rhsnoise(sp, wt=[100], st == wt)
else
    @rhsnoise(sp, wt=[100, 300], st == wt)
end
```

Readers should note that this use of of an `if` statement can be used more generally to conditionally add stage specific features to the subproblems.

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

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

`SDDP.jl` also allows the stagewise independent random variable ω_t to appear anywhere in the objective function expression. However, as this functionality is not needed in this example, readers are directed to the online documentation for more details. 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. We briefly summarize three of the most important solution features of `SDDP.jl`:

1. User-defined cut-selection routines: without modifying the `SDDP.jl` source-code, users are able to define new cut-selection heuristics (see [9]) 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 (see [36] and [44]) that seamlessly integrate with the entire `SDDP.jl` library. This functionality has been used by [35] 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.

We elaborate upon these three points in the next section. Returning to the air-conditioning example, we solve the problem using the SDDP algorithm via the `solve` method:

```
status = solve(m, max_iterations=10)
```

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 SDDP 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 objective bound (which can be verified by solving the deterministic equivalent) of $62,500. Status will return information about why the algorithm terminated. In this example, it will be `:max_iterations`.

2.4 Understanding the solution

Unlike deterministic mathematical programming, SDDP does not provide a solution containing an explicit optimal value for every variable. Instead, it constructs a *policy* in the form of a linear program. 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, [:xtp1, :pt, :ot, :st])
```

The first argument is the `SDDPModel m`. The second argument is the number of independent scenarios to simulate (in this case, 40), and the third argument is a list of variable values to record at each stage. For the air-conditioning problem, we record `xtp1`, 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 end of stage two), the system can be in one of two states: 100 units in storage if the previous stage's demand was 100 units, or zero units in storage if the previous stage's demand was 300 units. 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.

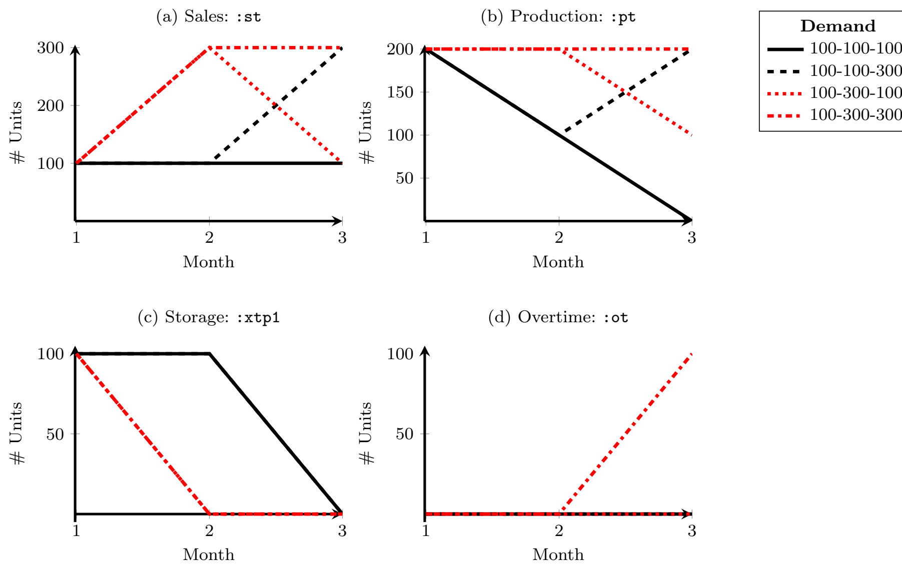

Fig. 1 Four simulations (each sampling a different demand scenario) of the air-conditioning problem using the optimal policy. Red lines sample high demand in stage two ($\omega_2 = 300$ units) .

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 Unique Design Features

In this section, we provide a deeper discussion on some of the unique features of `SDDP.jl`.

3.1 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 [9, 32, 26, 2]) 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 to implement a new cut oracle that is not implemented in `SDDP.jl`. This oracle only stores the N most recently discovered cuts (called the *last-cuts* strategy in [9]).[2] First, we define a new Julia type that is a sub-type of the abstract type `AbstractCutOracle` defined by `SDDP.jl`:

```
struct LastCutOracle{N} <: SDDP.AbstractCutOracle
    cuts::Vector{SDDP.Cut}
end
LastCutOracle(N::Int) = LastCutOracle{N}(SDDP.Cut[])
```

`LastCutOracle` has the type parameter `N` to store the maximum number of most-recent cuts to return. The type has the field `cuts` to store a vector of discovered cuts. More elaborate cut-selection heuristics may need additional fields to store other information. Next, we overload the `SDDP.storecut!` method. This method should store the cut `c` in the oracle `o` so that it can be queried later. In our example, we append the cut to the list of discovered cuts inside the oracle:[3]

```
function SDDP.storecut!{N}(o::LastCutOracle{N},
        m::SDDPModel, sp::JuMP.Model, c::SDDP.Cut)
    push!(o.cuts, c)
end
```

Lastly, we overload the `SDDP.validcuts` method. In our example, the strategy is to return the N most recent cuts. Therefore:

```
function SDDP.validcuts{N}(o::LastCutOracle{N})
    return o.cuts[max(1,end-N+1):end]
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

[2] Readers should note that this is not necessarily an oracle one would want to use.

[3] Note that this implementation stores all of the cuts discovered, rather than just the last N. We present this naïve implementation for brevity.

default behaviour of no cut selection, and the Level-One [9] cut selection method have been implemented. The Level-One cut selection oracle can be created as follows:

```
julia> cut_oracle = LevelOneCutOracle()
```

Cut oracles can be added to the model with the `cut_oracle` keyword in the `SDDPModel` constructor.[4] For example:

```
m = SDDPModel(
    # ... other keyword arguments ...
        cut_oracle = LastCutOracle(500)
                    ) do sp, t
    # ... sub-problem definition will go here ...
end
```

Due to the design of JuMP, we are unable to delete cuts from the model[5]. 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.

3.2 Risk Measures

Recently, significant efforts have been made to incorporate risk in to the SDDP algorithm [43,36,44,33,21,22]. In `SDDP.jl`, we use the "change-of-probability-dist -ribution" approach of [33]. For each risk measure, we create a function that takes the K realizations of a random variable $\{z_k\}_1^K$ with corresponding probabilities $\{p_k\}_1^K$, and returns the risk-adjusted probability distribution $\{\xi_k\}_1^K$. For example, the worst-case risk measure is"

$$\xi_k = \begin{cases} 1, \ k = \underset{k \in \{1,2,\ldots,Z\}}{\arg\min} \ \{z_k\} \\ 0, \ otherwise. \end{cases} \tag{2}$$

In a manner similar to the *cut oracle* design, `SDDP.jl` allows a core function (`SDDP.modifyprobability!`) to be overloaded to allow the development and testing of new risk measures. The `modfiyprobability!` function 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 of each scenario. This

[4] 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.

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

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 values (one for each realization of the noise ω). 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.[6] 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):

```
struct WorstCase <: SDDP.AbstractRiskMeasure end

function SDDP.modifyprobability!(::WorstCase,
        riskadjusted_distribution,
        original_distribution::Vector{Float64},
        observations::Vector{Float64},
        m::SDDPModel,
        sp::JuMP.Model
    )
    # reset the distribution
    riskadjusted_distribution .= 0.0
    if getsense(sp) == :Min
        # indmax returns the index of the maximum element
        idx = indmax(observations)
    else
        # indmax returns the index of the minimum element
        idx = indmin(observations)
    end
    # place all the weight on the worst outcome
    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 an SDDP framework in Julia: that the underlying library can easily be extended by the user without modifying the library itself.

3.3 Parallelization

The SDDP algorithm is highly parallelizable. This observation dates back to [31] who noted that the process could be conducted asyncronously, with performing iterations independently, and sharing cuts periodically. Different authors [38, 18] have proposed different methods for efficiently parallelizing the algorithm. However, we choose an approach that minimizes inter-process communication.

[6] This assumes that the worst outcome in `observations` has a positive probability of occurring. A proper implementation would account for this edge-case, but is too verbose for this simple example.

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.

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

3.4 Non-linear subproblems

Since `SDDP.jl` is built upon JuMP, it supports a wide variety of subproblem classes. For example, the problem of finding the least-squares estimator for a univariate distribution can be formulated as a (trivial) two-stage stochastic programming problem with a quadratic objective in the second stage and no constraints. The fist-stage subproblem is:

$$\mathbf{SP}_1: \quad V_1(x_1) = \min_{x_2} \mathbb{E}_{\omega_2}[V_2(x_2, \omega_2)], \tag{3}$$

and the second-stage subproblem is:

$$\mathbf{SP}_2: \quad V_2(x_2, \omega_2) = \min\ (x_2 - \omega_2)^2.$$

This can be written in `SDDP.jl` as:

```
using SDDP, JuMP, Gurobi
data = rand(50)
m = SDDPModel(
    stages          = 2,
    sense           = :Min,
    objective_bound = 0.0,
    risk_measure    = Expectation(),
    solver          = GurobiSolver(OutputFlag=0)
                                                    ) do sp, t
    @state(sp, xprime, x==0)
    if t == 1
        @stageobjective(sp, 0.0)
    else
        @stageobjective(sp, w=data, (x - w)^2)
    end
end
```

If we solve this problem, we find that the estimator x_2 converges to the sample mean of `data`. However, the 50 elements of `data` are only a sample of the true

distribution which is uniform on $[0, 1]$. The out-of-sample estimator that minimizes (3) is $x_2 = 0.5$. Therefore, if we repeatedly sampled 50 points and computed the least-squares estimator, the mean of those estimates would converge towards 0.5, but with high variance.

To reduce the out-of-sample variance, we can use a distributionally robust risk measure [35]. This risk measure computes the expectation over some worst-case probability distribution in some family of distributions. The robust approach avoids over-fitting the solution to the distribution of the sampled data. For this example, it is sufficient for readers to understand that the DRO risk measure is controlled by a single parameter, the radius, which can vary between 0 and 1. (Readers are directed to [35] for a full explanation of the method.) As the radius increases, the measure becomes more "robust." When the radius is 0.0, the DRO risk measure is equivalent to the expectation operator. When the radius is 1.0, the DRO risk measure is equivalent to the worst-case risk measure. The DRO risk measure can be initialized in SDDP.jl as follows:

```
risk_measure = DRO(radius::Float64)
```

To illustrate the distributionally robust risk measure, we solved the fitting problem 100 times (i.e. using 100 different samples of 50 points from the $U(0, 1)$ distribution) for five different radii. In Figure 2, we plot boxplots showing the distribution of the estimator for each of the five radii (Figure 2a), as well as the variance of the estimator (Figure 2b). As the radius of the DRO risk measure increases, the variance of the estimator decreases.

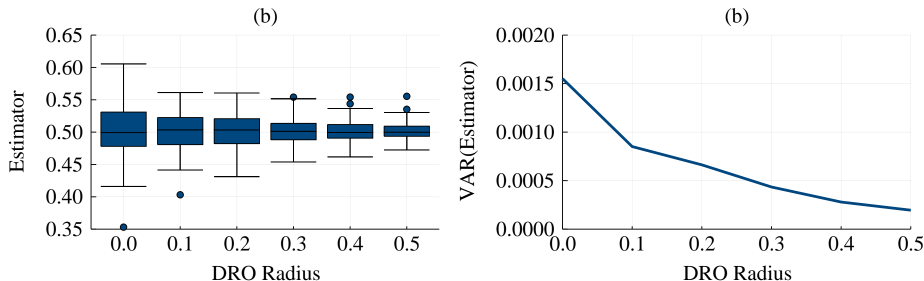

Fig. 2 Conic example using DRO risk measure: (a) boxplots showing the distribution of the estimator (i.e. x_2) over the 100 samples for each of the five radii; and (b) variance of the estimator over the 100 samples for each of the five radii.

This example demonstrates the ability of the SDDP.jl to solve non-linear subproblems, and use independently developed risk measures.

3.5 Other extensions

One core goal of [10] in designing JuMP was to make it extensible. 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` [20] which implements the method of [45] 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.

4 Benchmark: Hydro-Thermal Scheduling

In Section 2, we showed how to implement a simple multistage stochastic program using `SDDP.jl`. 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 [31]) is the Hydro-Thermal Scheduling Problem (HTSP). In the HTSP, 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 directed through hydro-electric turbines to produce electricity. However, future inflows into the reservoirs from rainfall or ice-melt are uncertain. Any unmet electricity demand is met by thermal generation. Therefore the 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 [41].[7] However for this paper, we used `DOASA`, a C++ implementation of the SDDP algorithm for the New Zealand HTSP.[8] In contrast to the generic `SDDP.jl`[9], `DOASA` is hard-coded to solve the New Zealand Hydro-Thermal scheduling problem. This allows it to gain some efficiency over `SDDP.jl`. Our copy of `DOASA` was provided by the Electric Power Optimization Centre at the University of Auckland, New Zealand.

To implement the New Zealand HTSP in `SDDP.jl`, we referred only to the description of the model given in [34]. 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`. In the following, all experiments were conducted on a Windows 7 machine with an Intel i7-4770 CPU and 16GB of memory, and all solvers used the default parameter settings.

4.1 Correctness

To test the correctness of our `SDDP.jl` implementation of the HTSP model, five deterministic instances of the New Zealand HTSP were created using historical 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 strongly

[7] Somewhat confusingly, the software is named SDDP.

[8] 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 [37]. We shall refer to `DOASA` the software by stylizing it in typewriter font.

[9] Further adding to the naming confusion.

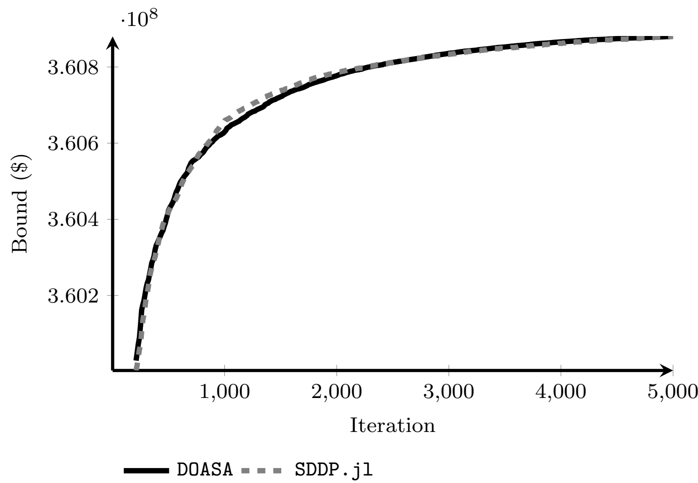

Fig. 3 Lower bound convergence of DOASA and SDDP.jl on stochastic instance of the New Zealand HTSP.

suggests that the two implementations of the model and deterministic SDDP algorithm are functionally equivalent.

	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 historical inflows from 1970–2007, and demand and pricing data from 2008. This problem was solved for 5000 SDDP iterations using both implementations. In Figure 3, 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 equivalent.

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

4.2 Performance

To compare the performance of `SDDP.jl` and `DOASA`, an instance of the New Zealand HTSP was created using 38 years of historical 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 [15], `SDDP.jl` using Gurobi version 6.5.0, `SDDP.jl` using Gurobi version 7.0.0 [17], and `SDDP.jl` using CPLEX version 12.6.1 [19]. The problem was solved 20 times for each of the configurations. The SDDP algorithm was terminated after 500 cuts had been generated for the first subproblem.

In Table 2, we summarize the results of these experiments. There are four columns of interest: *External*, *JuMP*, *SDDP*, and *Total*. In the *External* column, we report the time spent in the external solver libraries (i.e. CPLEX and Gurobi) solving the 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 external 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, excluding the solve time. These include sampling the random variables, calculating cut coefficients, and writing information 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.

	Solver	Time (s) External	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 over the 20 repetitions 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), with the result that `SDDP.jl` was faster than `DOASA`.

However, the `SDDP.jl` configuration with Gurobi v6.5.0 spent 22% longer in the external solver than `DOASA` with Gurobi v6.5.0 (and 29% longer overall) despite solving an identical number of linear programs. This is because `DOASA` is able to use more efficient cut-selection heuristics than `SDDP.jl` due to the tight coupling between `DOASA` and Gurobi. If the same cut-selection routines were implemented in `SDDP.jl`, this suggests that the `SDDP.jl` solution time could be improved further.

In contrast to the tight coupling between `DOASA` and Gurobi, the combination of `SDDP.jl` and JuMP allows the solver to be changed by the user with a single line of code. Thus, it was easy to show that despite solving larger subproblems, `SDDP.jl`

with CPLEX v12.6.1 was 15% faster DOASA with Gurobi v6.5.0. This suggests that in this application, CPLEX is faster than Gurobi and that this speedup outweighs the overhead of going through the JuMP abstraction layer. This overhead is approximately 10–20% of the total solution time. However, using JuMP 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 subproblems (for example, adding a new constraint).

Interestingly, SDDP.jl with Gurobi v7.0.0 is 3.7% slower than SDDP.jl with Gurobi v6.5.0. This could indicate that the improvements between Gurobi version 6.5.0 and Gurobi version 7.0.0 benefited 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.)

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 than by improving the code in the SDDP.jl library.

5 Comparison with Other Libraries

SDDP.jl is not the only library for solving multistage stochastic optimization problems using SDDP. We briefly describe six alternatives and contrast their abilities to SDDP.jl. A summary can be found in Table 3.

Four of the alternatives are open-source: StochDualDynamicProgram.jl [23], StructDualDynProg.jl [24], FAST (Finally, An SDDP Toolbox) [8], and StOpt [13]. The remaining two are commercial software: QUASAR [42], and the seminal SDDP by PSR [41].

Perhaps the most interesting observation is that (including the PSR re-write), four of the seven implementations are in Julia. Even more interesting, is that each of the authors chose 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 metaprogramming and multiple dispatch abilities, as well as the JuMP [11] package and wider JuliaOpt ecosystem are the ideal foundation upon which to build an SDDP library.

Since all of the Julia libraries are based on the JuliaOpt ecosystem, they share many of the same features. StochDynamicProgram.jl and StructDualDynProg.jl even share the same (user-extensible) cut-selection routines. Moreover, since SDDP.jl and StructDualDynProg.jl expose the full functionality of JuMP, they support arbitrary convex subproblems. StochDynamicProgram.jl does not expose the full functionality of JuMP. Instead it implements a different modelling layer. This allows it to support noise terms in the constraint coefficients (something notably missing from SDDP.jl). However, this comes at the expense of a less user-friendly modelling interface. For example, the following is an implementation of the air-conditioning example in StochDynamicProgrmaming.jl:

```
m = LinearSPModel(
    3,                    # stages
```

Name	SDDP.jl	StochDynamicProgram.jl	StructDualDynProg.jl	FAST	StOpt	QUASAR	PSR-SDDP[3]
Availability	Free	Free	Free	Free	Free	Commercial	Commercial
License	MPL-2.0	MPL-2.0	MIT	GPLv3	LGPLv3		
Language	Julia	Julia	Julia	MATLAB	C++[1]	Java[2]	Fortran
General?	Yes	Yes	Yes	Yes	Yes	Yes	No
Markov Chain	Yes	No	Yes	Yes	Yes	Yes	Yes
Subproblems	Convex	Linear	Convex	Linear	Linear	Quadratic	Linear
Noise	RHS/Obj.	Any	No	No	RHS	Any	Any
Risk	Custom	Custom	No	No	No	Yes	Yes
Cut Selection	Custom	Custom	Custom	No	No	Yes	Yes
Solvers	Many	Many	Many	Many	User	Many	Xpress

Table 3 SDDP library comparison. “Language” refers to the host language. “General” refers to whether the library can be used for a variety of problems. “Subproblems” refers to whether the subproblems must be linear, quadratic, or arbitrary convex programs. “Noise” refers to whether the library supports stagewise-independent noise terms. “Risk” refers to whether the library supports nested risk-measures. [1]StOpt also has a Python interface. [2]QUASAR also has MATLAB and Python interfaces. [3]PSR is currently re-writing their implementation in Julia and it will support convex subproblems and many different solvers (J. Dias Garcia, personal communication, April, 2018).

```
    [(0, 200), (0, Inf)], # control bounds
    [0.0],                # initial state
    # cost function
    (t, x, u, w) -> 100u[1] + 300u[2] + 50(x[1] + u[1] + u[2] - w[1]),
    # dynamics function
    (t, x, u, w) -> [ x[1] + u[1] + u[2] - w[1] ],
    [ # stagewise independent noise terms
        NoiseLaw([100], [1.0]),
        NoiseLaw([100, 300], [0.5, 0.5]),
        NoiseLaw([100, 300], [0.5, 0.5])
    ]
)
set_state_bounds(m, [(0, Inf)])
solve_SDDP(m,
    SDDPparameters(ClpSolver(), max_iterations = 10)
)
```

The `SDDP.jl` equivalent is given in Listing 1 at the end of this paper.

One reason for the design choice of StochDynamicProgramming.jl is so that the same `LinearSPModel` definition can be solved a variety of ways including stochastic dynamic programming and by an LP as the deterministic equivalent.

Unlike the other libraries, FAST and StructDualDynProg.jl do not support stagewise-independent noise terms. Therefore, the user is forced to model the stagewise independent noise terms using Markov chain. This is a large limitation since many more subproblems leads to a corresponding increase in memory requirements. Moreover, computation time may also increase.

Compared to the other open-source libraries, StOpt is implemented in a low-level language (C++). Moreover, it provides no modelling capability. Instead, the user codes their own subproblems, including the interface with any solver. This significantly adds to the development of new models. StOpt also lacks many of the features supported by the other libraries such as risk aversion and cut selection.

Finally, the two commercial libraries (QUASAR and PSR's SDDP) contain many of the advanced features in SDDP such as risk aversion, cut-selection, and parallel implementations. However, since these libraries are focused on the end-user, they offer less customization options by the user. PSR's SDDP [41] is also unique as the only non-general library we compare; it is exclusively for hydro-thermal scheduling.

From this analysis, we conclude that `SDDP.jl` is a valuable contribution to the suite of libraries for SDDP. Its biggest strengths are the first-class support for risk measures and cut-selection techniques, as well as the user-friendly interface. The biggest weakness of `SDDP.jl` is the lack of support for noise terms in the constraint coefficients.

6 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 difference between two commercial LP solvers. We also provided strong evidence 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` and `DOASA` are 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. Andrew Mason made numerous suggestions which greatly improved the clarity of this manuscript.

References

1. Asamov, T., Powell, W.B.: Regularized Decomposition of High-Dimensional Multistage Stochastic Programs with Markov Uncertainty. arXiv preprint arXiv:1505.02227 p. 33 (2015). URL `http://arxiv.org/abs/1505.02227`
2. Bandarra, M., Guigues, V.: Multicut decomposition methods with cut selection for multistage stochastic programs. arXiv preprint arXiv:1705.08977 (2017). URL `https://arxiv.org/abs/1705.08977`
3. Bellman, R.: The Theory of Dynamic Programming. Bulletin of the American Mathematical Society **60**(6), 503–515 (1954)
4. Bellman, R.: Dynamic Programming. Princeton University Press, Princeton (1957)
5. Bezanson, J., Edelman, A., Karpinski, S., Shah, V.B.: Julia: A Fresh Approach to Numerical Computing. SIAM Review **59**(1), 65–98 (2017)
6. Birge, J.R., Dempster, M.A., Gassmann, H.I., Gunn, E., King, A.J., Wallace, S.W.: A standard input format for multiperiod stochastic linear programs. IIASA Working Paper, WP-87-118, IIASA, Laxenburg, Austria (1987)
7. Bussieck, M.R., Ferris, M.C., Lohmann, T.: GUSS: Solving collections of data related models within GAMS. In: Algebraic Modeling Systems, pp. 35–56. Springer (2012)
8. Cambier, L., Scieur, D.: FAST (2018). URL `https://web.stanford.edu/~lcambier/fast/`. [Online; accessed 2018-04-05]
9. de Matos, V.L., Philpott, A.B., Finardi, E.C.: Improving the performance of Stochastic Dual Dynamic Programming. Journal of Computational and Applied Mathematics **290**, 196–208 (2015)
10. Dunning, I., Huchette, J., Lubin, M.: JuMP: A modeling language for mathematical optimization. arXiv:1508.01982 [math.OC] (2015). URL `http://arxiv.org/abs/1508.01982`
11. Dunning, I., Huchette, J., Lubin, M.: JuMP: A Modeling Language for Mathematical Optimization. SIAM Review **59**(2), 295–320 (2017)
12. Gassmann, H.I., Kristjansson, B.: The SMPS format explained. IMA Journal of Management Mathematics **19**(4), 347–377 (2007)
13. Gevret, H., Langrené, N., Lelong, J., Warin, X.: STochastic OPTimization library in C++. Tech. rep., EDF (2016). URL `https://hal.archives-ouvertes.fr/hal-01361291v5/document`
14. Guan, Z.: Strategic Inventory Models for International Dairy Commodity Markets. PhD Thesis, University of Auckland, Auckland, New Zealand (2008)
15. Gurobi Optimization: Gurobi 6.5 Reference Manual. Tech. rep., Gurobi Optimization (2016). URL `http://www.gurobi.com/documentation/6.5/refman/index.html`

16. Gurobi Optimization: Gurobi 7.0 Reference Manual. Tech. rep., Gurobi Optimization (2017). URL http://www.gurobi.com/documentation/7.0/refman/index.html
17. Gurobi Optimization: Gurobi 7.5 Performance Benchmarks. Tech. rep., Gurobi Optimization (2017). URL https://gurobi.com/pdfs/benchmarks.pdf
18. Helseth, A., Braaten, H.: Efficient Parallelization of the Stochastic Dual Dynamic Programming Algorithm Applied to Hydropower Scheduling. Energies **8**(12), 14,287–14,297 (2015)
19. IBM Corp.: IBM ILOG CPLEX Optimization Studio CPLEX User's Manual Version 12 Release 6 (2014)
20. Kapelevich, L.: SDDiP.jl: SDDP extension for integer local or state variables (2018). URL https://github.com/lkapelevich/SDDiP.jl. [Online; accessed 2018-01-19]
21. Kozmık, V., Morton, D.: Risk-averse stochastic dual dynamic programming. Optimization Online (2013). URL http://www.optimization-online.org/DB_FILE/2013/02/3794.pdf
22. Kozmík, V., Morton, D.P.: Evaluating policies in risk-averse multi-stage stochastic programming. Mathematical Programming **152**(1-2), 275–300 (2015)
23. Leclere, V., Pacaud, F., Rigaut, T., Gerard, H.: StochDynamicProgramming.jl (2018). URL https://github.com/JuliaOpt/StochDynamicProgramming.jl. [Online; accessed 2018-04-05]
24. Legat, B.: StructDualDynProg.jl (2018). URL https://github.com/blegat/StructDualDynProg.jl. [Online; accessed 2018-04-05]
25. Murtagh, B.: Advanced Linear Programming: Computation and Practice. McGraw-Hill International Book Co., New York (1981)
26. Nannicini, G., Traversi, E., Calvo, R.W.: A Benders squared (B2) framework for infinite-horizon stochastic linear programs. Optimization Online (2017). URL http://www.optimization-online.org/DB_HTML/2017/06/6101.html
27. Newham, N.: Power System Investment Planning using Stochastic Dual Dynamic Programming. Ph.D. thesis, University of Canterbury, Christchurch, New Zealand (2008)
28. Ourani, K.I., Baslis, C.G., Bakirtzis, A.G.: A Stochastic Dual Dynamic Programming model for medium-term hydrothermal scheduling in Greece. In: Universities Power Engineering Conference (UPEC), 2012 47th International, pp. 1–6. IEEE (2012)
29. Papavasiliou, A.: Stochastic Dual Dynamic Programming (2017). URL https://perso.uclouvain.be/anthony.papavasiliou/public_html/SDDP.pdf. [Online; accessed 2018-04-03]
30. Parpas, P., Ustun, B., Webster, M., Tran, Q.K.: Importance sampling in stochastic programming: A Markov chain Monte Carlo approach. INFORMS Journal on Computing **27**(2), 358–377 (2015)
31. Pereira, M., Pinto, L.: Multi-stage stochastic optimization applied to energy planning. Mathematical Programming **52**, 359–375 (1991)
32. Pfeiffer, L., Apparigliato, R., Auchapt, S.: Two methods of pruning Benders' cuts and their application to the management of a gas portfolio. Optimization Online (2012). URL http://www.optimization-online.org/DB_FILE/2012/11/3683.pdf
33. Philpott, A., de Matos, V., Finardi, E.: On Solving Multistage Stochastic Programs with Coherent Risk Measures. Operations Research **61**(4), 957–970 (2013)
34. Philpott, A., Pritchard, G.: EMI-DOASA. Tech. rep., Electric Power Optimization Centre (2013). URL http://www.emi.ea.govt.nz/Content/Tools/Doasa/DOASA\\%20paper\\%20by\\%20SOL.pdf
35. Philpott, A.B., de Matos, V., Kapelevich, L.: Distributionally Robust SDDP. Tech. rep., Electric Power Optimization Centre, Auckland, New Zealand (2017). URL http://www.epoc.org.nz/papers/DROPaperv52.pdf
36. Philpott, A.B., de Matos, V.L.: Dynamic sampling algorithms for multi-stage stochastic programs with risk aversion. European Journal of Operational Research **218**(2), 470–483 (2012)
37. Philpott, A.B., Guan, Z.: On the convergence of sampling-based methods for multi-stage stochastic linear programs. Operations Research Letters **36**, 450–455 (2008)
38. Pinto, R.J., Borges, C.T., Maceira, M.E.P.: An Efficient Parallel Algorithm for Large Scale Hydrothermal System Operation Planning. IEEE Transactions on Power Systems **28**(4), 4888–4896 (2013)
39. Powell, W.B.: Clearing the Jungle of Stochastic Optimization. In: A.M. Newman, J. Leung, J.C. Smith (eds.) Bridging Data and Decisions, TutORials in Operations Research, pp. 109–137. INFORMS (2014)

40. Powell, W.B.: A Unified Framework for Optimization under Uncertainty. In: A. Gupta, A. Capponi, J.C. Smith (eds.) Optimization Challenges in Complex, Networked and Risky Systems, TutORials in Operations Research, pp. 45–83. INFORMS (2016)
41. PSR: Software — PSR (2016). URL http://www.psr-inc.com/softwares-en/. [Online; accessed 2018-04-03]
42. Quantego: QUASAR (2018). URL http://quantego.com/. [Online; accessed 2018-04-05]
43. Shapiro, A.: Analysis of stochastic dual dynamic programming method. European Journal of Operational Research **209**(1), 63–72 (2011)
44. Shapiro, A., Tekaya, W., da Costa, J.P., Soares, M.P.: Risk neutral and risk averse Stochastic Dual Dynamic Programming method. European Journal of Operational Research **224**(2), 375–391 (2013)
45. Zou, J., Ahmed, S., Sun, X.A.: Stochastic Dual Dynamic Integer Programming. Optimization Online (2016). URL http://www.optimization-online.org/DB_HTML/2016/05/5436.pdf

A Code

Listing 1 Air-Conditioning Example

```
1  using JuMP, SDDP, Clp
2
3  m = SDDPModel(
4                   stages = 3,
5       objective_bound = 0.0,
6                    sense = :Min,
7                   solver = ClpSolver()
8                                 ) do sp, t
9       @variables(sp, begin
10          0 <= pt <= 200
11                  ot >= 0
12                  st >= 0
13      end)
14      @state(sp, xtp1 >= 0, xt == 0)
15      @constraint(sp, xt  + pt + ot - st == xtp1)
16      D = [ [100], [100, 300], [100, 300] ]
17      @rhsnoise(sp, wt=D[t], st == wt)
18      P = [ [1.0], [0.5, 0.5], [0.5, 0.5] ]
19      setnoiseprobability!(sp, P[t])
20      @stageobjective(sp, 100 * pt + 300 * ot + 50 * xt)
21 end
22
23 status = solve(m, max_iterations=10)
24 getbound(m) # should be 62,500
25
26 sims = simulate(m, 100, [:xtp1, :pt, :ot, :st])
```