E F F I C I E N T S T O C H A S T I C P R O G R A M M I N G I N J U L I A

A P R E P R I N T

Martin Biel  
Division of Decision and Control Systems  
School of EECS, KTH Royal Institute of Technology  
SE-100 44 Stockholm, Sweden  
mbiel@kth.se

Mikael Johansson  
Division of Decision and Control Systems  
School of EECS, KTH Royal Institute of Technology  
SE-100 44 Stockholm, Sweden  
mikael@kth.se

September 24, 2019

A B S T R A C T

We present StochasticPrograms.jl, an open-source framework for stochastic programming written in the Julia language. The framework includes both modeling tools and structure-exploiting optimization algorithms. We show how stochastic programming models can be efficiently formulated using expressive syntax. The framework was implemented to scale seamlessly to distributed environments. As a result, stochastic programs are efficiently memory-distributed on supercomputers or cloud architectures and solved using parallel optimization algorithms. These structured solvers are based on variations of the classical L-shaped and progressive-hedging methods. We provide concise mathematical backgrounds for the various tools and constructs available in the framework, along with code listings exemplifying their usage. Both software innovations related to the implementation of the framework and algorithmic innovations related to the structured solvers are highlighted. We conclude by performing numerical benchmarks of the distributed algorithms in a multi-node setup. The large-scale benchmark problems involve finding optimal order strategies on the Nordic day-ahead electricity market. We showcase strong scaling properties of the solvers and outline techniques for further speedups.

K e y w o r d s  S t o c h a s t i c p r o g r a m m i n g · D i s t r i b u t e d c o m p u t a t i o n s · L - s h a p e d · P r o g r e s s i v e - h e d g i n g · S w e e p c h s o f t w a r e f r a m e w o r k · J u l i a l a n g u a g e

1 I n t r o d u c t i o n

Stochastic programming is an effective mathematical framework for modeling decision problems that involve uncertainty [3]. It has been used to model complex real-world problems in diverse fields such as power systems [8, 20], finance [13, 30], and transportation [21, 22]. The classical setting is linear stochastic programs where an actor takes decisions in two stages:

\[
\text{decision } x \rightarrow \text{observation } \omega \rightarrow \text{recourse } y(x, \xi(\omega))
\]

The actor takes the first-stage decision \(x\) based on initial information before the realization of a random event \(\omega\). After later observing \(\omega\), the actor takes a recourse decision \(y\) with respect to the initial choice \(x\) and the output of some random variable \(\xi\). The notion of optimal decisions is captured by letting \(x\) and \(y\) be optimization variables in linear programs, where \(\xi(\omega)\) parameterizes the second-stage problem for each outcome \(\omega\). A linear objective function quantifies the quality of the decisions. Linear two-stage stochastic programming readily extends to multi-stage settings with nonlinear or mixed-integer models [3].

In the applied setting, a given stochastic program models some real-world decision problem and there is often some statistical model of \(\xi\) available. We then compute approximations of optimal decision policies by solving approximated instances of the stochastic program. In brief, this involves computing a first-stage decision \(\hat{x}\) that is optimal in expectation over a set of second-stage scenarios \(\xi(\omega_i)\) sampled from the model of \(\xi\). This technique is known as sampled average approximation (SAA). In the linear setting, one can in principle solve SAA instances by formulating
the extensive form that considers all available scenarios at once. This mathematical program can be solved using standard linear programming solvers, including both open-source solvers such as GLPK [17] and commercial solvers such as Gurobi [10]. However, the size of the extensive form grows linearly in the number of scenarios, and industry-scale applications typically involve 10,000+ scenarios. For example, a 24-hour unit commitment problem with 16,384 scenarios amounted to about 4 billion variables and constraints in the extensive form [20]. Hence, solving the extensive form of the SAA in industrial applications may easily become intractable in practice. Moreover, the memory requirement for storing the SAA will eventually exceed the capacity of a single machine. This clarifies the need for a distributed-memory approach when modeling large-scale stochastic programs. Structure-exploiting decomposition methods [24, 27] that operate in parallel on distributed data become essential to solve large-scale SAA instances.

1.1 Contribution

In this work, we present an open-source software framework for efficiently modeling and solving stochastic programs in a distributed-memory setting. We implemented the framework in the Julia [11] programming language. Henceforth, we refer to the framework as SPjl. The main functionality and tools are available in the registered Julia package StochasticPrograms.jl. Specialized solvers are available in the separate modules LShapedSolvers.jl and ProgressiveHedgingSolvers.jl.

A fundamental principle in the SPjl framework is the separation between model design and data design. This design principle has resulted in two key software innovations: deferred model instantiation and data injection. Optimization models in the first and second stages are formulated using straightforward syntax while simultaneously specifying the data dependencies between the stages. The data structures related to future scenarios, and their statistical properties, are defined separately. An essential consequence of this design is that we can efficiently distribute stochastic program instances in memory. Many computations involving distributed stochastic programs can then natively run in parallel. Moreover, when the sample space is infinite, it becomes possible to adequately distinguish between the abstract representation of a stochastic program and finite SAA instances. The design also enables swift implementation of various constructs from classical stochastic programming theory. In short, SPjl is a powerful, versatile, and extensible framework for stochastic programs. Domain-experts can develop and test complex models locally on a laptop and then seamlessly run large-scale instances of the same code in production, on a supercomputer or a cloud cluster.

We developed SPjl in Julia, which has several distinct benefits. Through just-in-time compilation and type inference, Julia can achieve C-like performance while being as expressive as similar dynamic languages such as Python or Matlab. Using the high-level capabilities of Julia, it is possible to create domain-specific tools with expressive syntax and high performance. Many software implementations in SPjl were created using MacroTools.jl, which is a metaprogramming tool that can be used to introduce new domain-specific syntax to the Julia language. Another benefit is access to Julia’s large and rapidly expanding ecosystem of libraries, many of which play a central role in SPjl. For example, the parallel capabilities of SPjl are implemented using the standard module for distributed computing, while optimization models in SPjl are formulated using the JuMP [5] ecosystem. JuMP is an algebraic modeling language implemented in Julia using similar metaprogramming tools. It has been shown to achieve similar performance to AMPL [5], with syntax that is both readable and expressive. Also, it is possible to mutate model instances at runtime, which we utilize in the structure-exploiting algorithms in SPjl. JuMP implements interfaces to many third-party optimization solvers, both open-source and commercial. These can be hooked in to solve extensive forms of stochastic programs or subproblems that arise in decomposition methods.

1.2 Related work

We give a short survey of similar software packages and highlight distinguishing features of SPjl. The most similar approach is the PySP framework [28], implemented in the Python language. Optimization models in PySP are created using Pyomo [11]; an algebraic modeling language also implemented in Python. In contrast, SPjl is written in the Julia language and formulates optimization models in JuMP, which has been shown to outperform Pyomo in various benchmarks [5]. In PySP, stochastic programs are composed of multiple .dat files and .py files, and the models are solved by running different solver scripts. In SPjl, all models are described in pure Julia and can be created, analyzed and solved in a single interactive session. Moreover, all operations are natively distributed in memory and run in parallel if multiple Julia processes are available. The parallel capabilities of PySP extend to running parallelized versions of the solver scripts. The primary function of PySP is to formulate and solve stochastic programs, while SPjl also provides a large set of stochastic programming constructs and analysis tools. The expressiveness of the modeling syntax can be compared by observing how the well-known farmer problem [23] is modeled using PySP [28] and how it is modeled using SPjl, as shown in Section 5.1. A more extensive list of similar software approaches is provided in [28], along with comparisons to PySP. This allows for a transitive comparison to SPjl.
The StructJuMP package [12] is a similar Julia implementation, which provides a simple interface to create block structured JuMP models. The primary reason for developing StructJuMP was to facilitate a parallel modeling interface to existing structured solvers [15, 20] that operate in computer clusters. These parallel solvers are implemented in C++ and are parallelized using MPI. This led to StructJuMP also making use of MPI to distribute stochastic programs in blocks. Apart from formulating distributed stochastic programs in a cluster, StructJuMP does not offer any modeling tools nor any way to generate the extensive form of a stochastic program. In comparison, SPjl provides numerous analysis tools as well as a compatible suite of structured solvers. In addition, SPjl natively distributes and solves stochastic programs using Julia, without relying on external software such as MPI or having to install external solvers.

2 Stochastic programming

This section introduces the implemented stochastic programming concepts in SPjl. We restrict the discussion to linear two-stage stochastic programming. The modeling capabilities of SPjl extend to nonlinear and mixed-integer problems as well, and there is support for solving such problems through third-party solvers applied to the extensive forms. However, the specialized solvers in SPjl are currently only implemented for linear stochastic programs. When appropriate, we accompany the introduced concepts by code listings that exemplify the usage of SPjl. We highlight functionality where SPjl implements parallelized strategies.

2.1 Definitions and notation

A linear two-stage recourse model enables a simple but powerful framework for making decisions under uncertainty. To formalize, consider some probability space \((\Omega, \mathcal{F}, \pi)\) where \(\Omega\) is a discrete (possibly infinite, but countable) sample space, \(\mathcal{F}\) is a \(\sigma\)-algebra over \(\Omega\) and \(\pi: \mathcal{F} \to [0, 1]\) is a probability measure. Let \(\xi(\omega): \Omega \to \mathbb{R}^N\) be some discrete random variable on \(\Omega\) with finite second moments, i.e. 
\[
\mathbb{E}_\omega[\xi(\omega)^2] = \sum_{\omega \in \Omega} \pi(\omega)\xi(\omega)^2 < \infty
\]

The first-stage decision is denoted by \(x \in \mathbb{R}^n\). We associate this decision with a linear cost function \(c^T x\) that the actor pays after making the decision. Moreover, \(x\) is constrained to the standard polyhedron in linear programming, i.e. 
\[
\{x \in \mathbb{R} \mid Ax = b, \ x \geq 0\}
\]

Let \(\omega \in \Omega\) denote a scenario observed after making the decision \(x\). The recourse decisions are represented by \(y \in \mathbb{R}^m\). Further, the random variable \(\xi(\omega) = (q_\omega, T_\omega, h_\omega)^T\) parameterizes the second-stage model in the following way
\[
Q(x, \xi(\omega)) = \min_{y \in \mathbb{R}^m} q_\omega^T y \\
\text{s.t.} \ T_\omega x + W y = h_\omega \\
y \geq 0
\]

Now, we formulate the two-stage recourse problem as follows.
\[
\begin{align*}
\text{minimize} & \quad c^T x + \mathbb{E}_\omega[Q(x, \xi(\omega))] \\
\text{s.t.} & \quad Ax = b \\
& \quad x \geq 0
\end{align*}
\] (1)

This mathematical representation of a recourse problem is well-defined under the stated assumptions on \(\Omega\) and \(\xi\). Specifically, if \(\Omega\) is finite, say with \(n\) scenarios of probability \(\pi_s\) respectively, then we can represent (1) compactly as
\[
\begin{align*}
\text{minimize} & \quad c^T x + \sum_{s=1}^n \pi_s q_s^T y_s \\
\text{s.t.} & \quad Ax = b \\
& \quad T_s x + W y_s = h_s, \quad s = 1, \ldots, n \\
& \quad x \geq 0, \ y_s \geq 0, \quad s = 1, \ldots, n
\end{align*}
\] (2)

In the general case, with no assumptions on \(\Omega\), the formulation (1) provides an abstraction of a present-time decision that is optimal in expectation. This representation is not necessarily computable. However, it is possible to formulate computationally tractable approximations of (1) of the form (2). The most common approximation technique is the sampled average approximation (SAA) [3], described in Section 2.3. We say that (2) is a sampled model that approximates the base model (1). The general purpose of the SPjl framework is to model stochastic programs (1) and store and solve instances (2) of the stochastic programs efficiently, on a computer or a compute cluster.
2.2 Simple example

A simple instance of (1) is given by:

\[
\begin{align*}
\text{minimize} \quad & 100x_1 + 150x_2 + E_{\omega}[Q(x_1, x_2, \xi(\omega))] \\
\text{s.t.} \quad & x_1 + x_2 \leq 120 \\
& x_1 \geq 40 \\
& x_2 \geq 20 \\
\end{align*}
\]

where

\[
Q(x_1, x_2, \xi(\omega)) = \min_{y_1, y_2 \in \mathbb{R}} q_1(\omega)y_1 + q_2(\omega)y_2
\]

\[
\begin{align*}
\text{s.t.} \quad & 6y_1 + 10y_2 \leq 60x_1 \\
& 8y_1 + 5y_2 \leq 80x_2 \\
& 0 \leq y_1 \leq d_1(\omega) \\
& 0 \leq y_2 \leq d_2(\omega)
\end{align*}
\]

and the stochastic variable

\[
\xi(\omega) = (q_1(\omega) \quad q_2(\omega) \quad d_1(\omega) \quad d_2(\omega))^T
\]

parameterizes the second-stage model.

In SPjl, we define the model (3) in two steps. First, we identify the scenario-specific data to define it as shown in Listing 1. This defines a Julia structure SimpleScenario with predefined functionality. For example, we obtain the discrete probability of a given scenario through probability. Also, expectation over a set of scenarios can be obtained using expected. The defined scenario structure automatically becomes available on remote processes for distributed processing. Next, we formulate the optimization models as shown in Listing 2. This creates a stochastic model where the two stages are given by the mathematical programs (3) and (4), using JuMP syntax. The line @decision x1 x2 internally defines the data dependencies between the first and second stage. Moreover, the @uncertain line annotates the random parameters and defines a point of data injection. The code specifies how the optimization models should be defined, but the actual model instantiation is deferred until we add scenario data. We will consider two different distributions of \( \xi \) and use the same model object simple_model from Listing 2 to instantiate stochastic programs.
First, let $\xi$ be a discrete distribution, taking on the value

$$\xi_1 = (500 \ 100 \ -24 \ -28)^T$$

with probability 0.4 and

$$\xi_2 = (300 \ 300 \ -28 \ -32)^T$$

with probability 0.6. In Listing 3 an instance of the stochastic program (3) is created for this distribution. This code uses the model recipe created in Listing 2 to create second-stage models for each of the supplied scenarios. Listing 4 shows a printout of the resulting stochastic program.

For smaller problem instances it is tractable to consider (2) as one large optimization problem. We generally refer to this optimization problem as the deterministic equivalent problem (DEP), or the extensive form, as it considers all scenarios at once. The result of generating the extensive form of (3) when $\xi$ has a small discrete distribution is shown in Listing 5. We can easily verify the correctness of the result since the problem is small.

### 2.3 Sampled average approximation

In practice, $\Omega$ is typically infinite and $\xi$ is continuous. The finite instance (2) is still of great value as we can approximate (1) through sampling. Assume that we sample $n$ scenarios $\omega_s$, $s = 1, \ldots, n$ from $\Omega$ with equal probability. These scenarios now constitute a finite sample space $\hat{\Omega}$ with a simple measure

$$\hat{\pi}(\omega_i) = \frac{1}{n}, \quad i = 1, \ldots, n$$
Listing 5: Extensive form (deterministic equivalent) of (3) in SPjl.

```julia
julia> dep = DEP(sp);

julia> print(dep)
Min 100 $x_1 + 150 x_2 - 9.6 y_1 - 11.2 y_2 - 16.8 y_3 - 19.2 y_4$
Subject to
$x_1 + x_2 \leq 120$
$6 y_1 + 10 y_21 - 60 x_1 \leq 0$
$8 y_1 + 5 y_21 - 80 x_2 \leq 0$
$6 y_3 + 10 y_22 - 60 x_1 \leq 0$
$8 y_3 + 5 y_22 - 80 x_2 \leq 0$
$x_1 \geq 40$
$x_2 \geq 20$
$0 \leq y_1 \leq 500$
$0 \leq y_21 \leq 100$
$0 \leq y_22 \leq 300$
```

We form an SAA model as a finite model (2) using the sampled scenarios. An optimal solution to this SAA model approximates the optimal solution to (1) in the sense that the empirical average second-stage cost

$$
\frac{1}{n} \sum_{s=1}^{n} q_s^T y_s
$$

where

$$
\hat{y}_s = \arg \min_{y \in \mathbb{R}^m} \{ Q(x, \xi(\omega_s)) \}
$$

converges pointwise with probability 1 to

$$
E_\omega[Q(x, \xi(\omega))]
$$

as $n \to \infty$.

Consider again the simple example introduced in Section 2.2 but let instead $\xi \sim \mathcal{N}(\mu, \Sigma)$. In other words, $\xi$ has a multivariate normal distribution. In general, there is no closed form solution of (1) when $\xi$ has a continuous distribution. However, by the law of large numbers, a viable discrete approximation can be obtained by sampling scenarios from the continuous distribution. In SPjl, we achieve this by creating a sampler object associated with the defined scenario structure. In Listing 6, a sampler object for a multivariate distribution with

$$
\mu = \begin{pmatrix} -28 \\ -32 \\ 300 \end{pmatrix}, \quad \Sigma = \begin{pmatrix} 2 & 0.5 & 0 & 0 \\ 0.5 & 1 & 0 & 0 \\ 0 & 0 & 50 & 20 \\ 0 & 0 & 20 & 30 \end{pmatrix}
$$

is created and used to generate an instance of (3) with 100 sampled scenarios. The same stochastic model object defined in Listing 2 is used in Listing 6 to generate the SAA instance. This is a key feature in SPjl. Any instance of a stochastic program knows the underlying stochastic model (1). Regardless of the distribution of $\xi$, a stochastic program instance in SPjl is always a finite program of the form (2). If multiple Julia processes are available, either locally or remotely, then the code in Listing 6 would automatically distribute the stochastic program in memory on the available nodes. Although not practically required for this small example, this leads to significant performance gains for large-scale industrial models. See for example the scaling results for large day-ahead problems presented in Section 5.2.

### 2.4 Decision evaluation

Decision evaluation is an important concept in stochastic programming. It concerns how to evaluate the performance of a given candidate decision $\tilde{x}$ to (1). The straightforward approach is to quantify the performance by the expected result of making the decision $\tilde{x}$, i.e.

$$
V(\tilde{x}) = c^T \tilde{x} + E_\omega[Q(\tilde{x}, \xi(\omega))]
$$

Computing a candidate decision is in general more involved than estimating (5). Consequently, we can compute a more accurate estimate of the stochastic performance by evaluating (5) with larger sampled models.
using Distributions

@sampler SimpleSampler = begin
    N::MvNormal
        SimpleSampler(µ, Σ) = new(MvNormal(µ, Σ))
    end
end

@sample SimpleScenario begin
    x = rand(sampler.N)
    (q₁, q₂, d₁, d₂) = x
    return SimpleScenario(q₁, q₂, d₁, d₂, probability = pdf(sampler.N, x))
end

µ = [-28, -32, 300, 300]
Σ = [2 0.5 0 0
     0.5 1 0 0
     0 0 50 20
     0 0 20 30]
saa = SAA(simple_model, SimpleSampler(µ, Σ), 100)

Stochastic program with:
* 100 scenarios of type SimpleScenario
* 2 decision variables
* 2 recourse variables
Solver is default solver

Listing 7: Evaluation of the candidate decision \( \tilde{x} \), defined in (6), to problem (3) when \( \xi \) follows a discrete distribution.

```
julia> x = [50, 50];

julia> evaluate_decision(sp, x)
356.0
```

2.4.1 Finite sample space

If we consider a discrete model (2), the expectation in (5) is well-defined and can be calculated by solving \( n \) optimization problems and forming the average objective. In SPjl, the result of evaluating the candidate decision

\[
\tilde{x} = (50 \quad 50)^T
\]

(6)

to the problem (3) is shown in Listing 7. The subproblems that have to be solved are independent. Hence, we automatically parallelize finite decision evaluation if multiple Julia processes are available.

2.4.2 Infinite sample space

In the general case, the best we can achieve is to evaluate (5) statistically. We follow the theory developed in [16]. First, consider an SAA instance of (5)

\[
V_N(\tilde{x}) = c^T \tilde{x} + \frac{1}{N} \sum_{s=1}^{N} Q(\tilde{x}, \xi_s)
\]

(7)

Evaluation of the subproblems is relatively cheap, so we can use large values of \( N \). Now, we solve \( T \) sampled batches \( \xi^T_i \) where \( \xi^T_i = \{ \xi^T_i \}_{s=1}^{N} \) of the above problem, and construct the unbiased estimator

\[
U_T = \frac{1}{T} \sum_{i=1}^{T} V_N^i = \frac{1}{T} \sum_{i=1}^{T} \left( c^T \tilde{x} + \frac{1}{N} \sum_{s=1}^{N} Q(\tilde{x}, \xi^T_i) \right)
\]

which estimates \( V(\tilde{x}) \). In addition, it can be used to upper bound an estimate of the gap between \( V(\tilde{x}) \) and the true optimum of (1). Here, \( N \) can be a large number since the above problem simply amounts to solving \( N \) independent
smaller optimization problems. Using the $\alpha$-critical value of the $t$-distribution with $T-1$ degrees of freedom, a $1-\alpha$ confidence interval around $U_T$ is given by

$$[U_T - \frac{t_{\alpha/2,T-1}\sigma_T^2}{\sqrt{T}}, U_T + \frac{t_{\alpha/2,T-1}\sigma_T^2}{\sqrt{T}}]$$

where the sample variance is given by

$$\sigma_T^2 = \frac{1}{T-1} \sum_{i=1}^{T} (V_N^i(\tilde{x}) - U_T)^2$$

A lower bound on the gap between $V(\tilde{x})$ and the true optimum of (1) is computed by first solving $n$-sized SAA instances of the form

$$\hat{V}_n^i = \min_{x \in \mathbb{R}^n} c^T x + \frac{1}{n} \sum_{s=1}^{n} Q(x, \xi^i_s)$$

s.t. $Ax = b$

$$x \geq 0$$

(8)

for $M$ sampled batches $\{\xi^i\}_{i=1}^{M}$. Now,

$$c^T x + \mathbb{E}_{\omega}[Q(x, \xi(\omega))] = \mathbb{E}_{\omega_n} \left[ c^T x + \frac{1}{n} \sum_{s=1}^{n} Q(x, \xi_s(\omega_n)) \right] \geq \mathbb{E}_{\omega_n} [\hat{V}_n(\omega_n)]$$

Consequently,

$$\hat{V} \geq \mathbb{E}_{\omega_k} [\hat{V}_n(\omega_n)]$$

and an estimate of this lower bound can be computed by

$$\hat{L}_{n,M} = \frac{1}{M} \sum_{i=1}^{M} \hat{V}_n^i$$

Since this involves solving stochastic programs, the values of $M$ or $n$ cannot be too large for the procedure to be computationally tractable on a single node. In a distributed environment, we can employ parallel solver strategies since the procedure trivially parallelizes over $M$. An approximate $(1 - \alpha)$ confidence interval around the lower bound is then given by

$$\left[ L_n,M - \frac{t_{\alpha/2,M-1}\sigma_{n,M}^2}{\sqrt{M}}, L_n,M + \frac{t_{\alpha/2,M-1}\sigma_{n,M}^2}{\sqrt{M}} \right]$$

where the sample variance is given by

$$\sigma_{n,M}^2 = \frac{1}{M-1} \sum_{i=1}^{M} (\hat{V}_n^i - L_{n,M})^2$$

The two bounds are now combined to form a $1 - 2\alpha$ confidence interval around the gap between $V(\tilde{x})$ and the true optimum of (1):

$$\left[ 0, U_T - L_{n,M} + \frac{t_{\alpha/2,T-1}\sigma_T^2}{\sqrt{T}} + \frac{t_{\alpha/2,M-1}\sigma_{n,M}^2}{\sqrt{M}} \right]$$

For large enough $n$, if we acquire a candidate decision by solving a single SSA instance then we can use the above procedure to calculate a confidence interval around the true optimum of (1).

We have implemented the above procedures in SPjl. We evaluate the decision candidate (6) on the problem where $\xi$ follows a normal distribution by supplying the sampler object defined in Listing 6 to `evaluate_decision`. The results are shown in Listing 8. A confidence interval around the true optimum can be directly obtained as shown in Listing 9.
2.5 Optimal decisions

We determine optimal decision candidates to the recourse problem (1) by solving optimization problems associated with (1). There are several common ways to formulate the associated optimization problems, many of which are supported by SPjl. Again, note that the resulting decision candidates are only optimal with respect to some sampled model (2), which depends on the given scenario data. A possible workflow is to determine optimal decisions to a sampled model with carefully selected scenarios and then statistically evaluate the decision as described in the previous section.

2.5.1 Here-and-now decisions

The primary method for finding optimal decision candidates consists of approximately solving the recourse problem (1) by formulating and solving some sampled model (2). This here-and-now approach gives the best candidate decision with respect to all conjectured scenarios included in the sampled model. This problem can be hard to solve since the problem size rapidly grows when adding scenarios. However, the solution quality, in relation to the base model (1), can be high if we choose the included scenarios carefully.

As (2) is a proper linear program, we can solve it using standard linear programming methods. In SPjl, we do this by calling `optimize!` with a supplied third-party solver. In Listing 10, we solve the simple example (3) through the extensive form using the GLPK solver [17]. Afterwards, we can query the optimal decision using `optimal_decision`.

```julia
julia> optimize!(sp, solver=GLPKSolverLP())
:Optimal
julia> optimal_decision(sp)  # Optimal first-stage decision
2-element Array{Float64,1}:
 46.6667
 36.25
julia> optimal_value(sp)
-855.83
julia> optimal_decision(sp, 1)  # Optimal recourse-decision in scenario 1
2-element Array{Float64,1}:
 300.0
 100.0
julia> VRP(sp, solver=GLPKSolverLP())  # Value of recourse problem directly
-855.83
```

Listing 8: Statistical evaluation of the candidate decision $\tilde{x}$, defined in (6), to problem (3) when $\xi$ follows a normal distribution. Both the expected result and the gap to the true optimum is calculated.

```julia
julia> x̃ = [50, 50];
julia> CI = evaluate_decision(simple_model, x̃, sampler, confidence = 0.95)
Confidence interval (p = 90%): [228.25 - 279.17]
julia> CI = gap(simple_model, x̃, sampler, confidence = 0.9)
Confidence interval (p = 90%): [0.00 - 2858.11]
```

Listing 9: Confidence interval at 95% around the true optimum of problem (3) when $\xi$ follows a normal distribution.

```julia
julia> confidence_interval(simple_model, sampler, confidence = 0.95)
Confidence interval (p = 95%): [-2544.05 - -2524.30]
```
Listing 11: Solving (3) using the L-shaped algorithm in SPjl.

```
julia> using LShapedSolvers
julia> optimize!(sp, solver=LShapedSolver(GLPKSolverLP()))
L-Shaped Gap Time: 0:00:01 (4 iterations)
  Objective:  -855.8333333333358
  Gap:       2.1229209144670507e-15
  Number of cuts:   6
:Optimal

julia> optimal_decision(sp)
2-element Array{Float64,1}:
  46.6667
  36.25
```

Also, the subproblems associated with the scenarios are filled with the optimal recourse actions corresponding to the optimal first-stage decision. We can query these decisions by supplying the subproblem index to `optimal_decision`. For convenience, the command `VRP(value of the recourse problem)` can be used to solve `sp` and directly return the optimal value. For large problems, solving extensive forms can be time-consuming, and it is advised to make use of decomposition approaches. SPjl includes a solver suite based on variations of the classical L-shaped algorithm [27] and the progressive-hedging algorithm [24]. L-shaped and progressive-hedging algorithm variants, as well as their implementation in SPjl, are described more in-depth in Section 3.1 and Section 3.2. Each implemented algorithm has an associated solver object that can be supplied to `optimize!` in place of standard solvers.

In Listing 11 the simple example (3) is solved using the classical L-shaped algorithm, using the GLPK solver [17] to solve emerging subproblems. If Julia is started with multiple processes, supplying `distributed = true` to `LShapedSolver` would instead run a parallel variant of the L-shaped algorithm on distributed data.

### 2.5.2 Wait-and-see decisions

Another approach is to solve the following problem

$$EW_S = \mathbb{E}_\omega \left[ \min_{x \in \mathbb{R}^n} c^T x + Q(x, \xi(\omega)) \right] \quad \text{s.t.} \quad Ax = b, \quad x \geq 0 \tag{9}$$

In words, this is the expected outcome if the scenarios are known already when the actor makes the first-stage decision. For a given finite model with \( n \) scenarios, approximately solving (9) corresponds to solving \( n \) problems of the form

$$\min_{x \in \mathbb{R}^n, y_s \in \mathbb{R}^m} c^T x + q_s^T y_s \quad \text{s.t.} \quad Ax = b, \quad T_s x + W y = h_s, \quad x \geq 0, \quad y \geq 0 \tag{10}$$

and forming the expectation of the objective values. The expected wait-and-see solution, \( EW_S \), is the average outcome of having perfect information in each possible scenario. Since each candidate solution \( \hat{x}_s \) adapts to a specific scenario, it is expected to perform better than the here-and-now solution if scenario \( s \) is the actual outcome. Wait-and-see problems are available in SPjl through the `WS` command. The command is supplied with a scenario structure and yields the wait-and-see model associated with that scenario. See Listing 12 for an example.

As shown in Listing 13 the command `EWS` solves wait-and-see problems for all available scenarios and returns the expected result. Similar to decision evaluation, the wait-and-see subproblems are independent. Therefore, if we start Julia with multiple processes, the `EWS` call will automatically be run in parallel on distributed data.

### 2.5.3 Expected value decisions

The last alternative is to first form the expected scenario out of the available scenarios

$$\bar{\xi} = \mathbb{E}_\omega [\xi(\omega)] \tag{11}$$
and then solve the expected value problem

\[
\begin{align*}
\text{minimize} & \quad c^T x + Q(x, \bar{\xi}) \\
\text{subject to} & \quad Ax = b \\
& \quad x \geq 0
\end{align*}
\]

(12)

Note, that this corresponds to solving a wait-and-see problem for the expected scenario. Among the alternatives, the EV problem is the smallest and therefore easiest to solve. However, it yields a naive solution since it does not consider the second-stage uncertainty. Hence, the optimal value of the EV problem will typically promise a better result than what we would expectedly achieve if the actor makes the suggested decision. In SPjl, the EV problem is available through the \texttt{EVP} command, as shown in Listing [14].

The optimal value of the expected value problem is available directly through the \texttt{EV} command. Moreover, the result of evaluating the expected value decision, often denoted as the expected result of using the EV solution or \texttt{EEV} = \mathbb{E}_{\omega}[c^T \bar{x}(\bar{\xi}) + Q(\bar{x}(\bar{\xi}), \xi(\omega))]

(13)
is available directly through the \texttt{EEV} command in SPjl. Note that although the optimal value of the expected value problem is typically low, the solution tends to be suboptimal to the sampled problem. Consequently, EEV is often higher than the optimal cost of the sampled problem.

### 2.6 Stochastic performance

The solution concepts introduced in the previous section give rise to two fundamental measures of stochastic performance. The expected value of perfect information (EVPI) measures the expected loss of not knowing the exact outcome beforehand. For a given recourse model, \textit{EVPI} is given by

\[
EVPI = VRP - EWSS
\]

(14)

With respect to some sampled model, \textit{VRP} and \textit{EWSS} are calculated as shown in Section [2.5.1] and Section [2.5.2] respectively. In SPjl, \textit{EVPI} is available through the \texttt{EVPI} command, as shown in Listing [15]. The resulting \textit{EVPI} quantifies the value of having access to an accurate forecast.
Listing 14: Expected value solution in SPjl. After solving the expected value problem, the optimal value is compared to the evaluated performance on the sampled model.

```julia
julia> evp = EVP(sp);
julia> print(evp)
Min 100 x₁ + 150 x₂ - 26.4 y₁ - 30.4 y₂
Subject to
  x₁ + x₂ ≤ 120
  6 y₁ + 10 y₂ - 60 x₁ ≤ 0
  8 y₁ + 5 y₂ - 80 x₂ ≤ 0
  x₁ ≥ 40
  x₂ ≥ 20
  0 ≤ y₁ ≤ 380
  0 ≤ y₂ ≤ 220
julia> x = EVP_decision(sp)
2-element Array{Float64,1}:
  71.4583
  48.5417
julia> EV(sp)
-1445.92
julia> evaluate_decision(sp, x)
-568.92
julia> EEV(sp, x)
-568.92
```

Listing 15: Evaluation of \( EVPI \) for the simple problem (3) in SPjl.

```julia
julia> EVPI(sp)
662.92
```

The second concept is the value of the stochastic solution \( (VSS) \), which measures the expected loss of ignoring the uncertainty in the problem. For a given recourse model, \( VSS \) is given by

\[
VSS = EEV - VRP
\]  

With respect to some sampled model, \( EEV \) and \( VRP \) are calculated as shown in Section 2.5.3 and Section 2.5.1 respectively. In SPjl, \( VSS \) is available through the \( \text{VSS} \) command, as shown in Listing 16. The resulting value indicates if the second stage is sensitive to the stochastic data. It is advised to evaluate \( VSS \) on a reasonably sized sampled model. If the \( VSS \) is low, then there is little benefit in solving a large-scale instance.

### 3 Structured algorithms

SPjl includes a solver suite of structure-exploiting algorithms. The various algorithms are based on different decompositions of finite sampled instances \( \mathcal{S} \), and we specialize each considered algorithm to distributed-memory environments through different parallelization strategies.

Listing 16: Evaluation of \( VSS \) for the simple problem (3) in SPjl.

```julia
julia> VSS(sp)
286.92
```
3.1 L-shaped algorithms

The solver family \texttt{LShapedSolvers.jl} in SPjl is based on the L-shaped algorithm [27]. In brief, the L-shaped method is a cutting plane algorithm that decomposes (2) into a master problem and \( n \) subproblems. The master problem is given by

\[
\begin{align*}
\text{minimize} & \quad c^T x + \sum_{s=1}^{n} \theta_s \\
\text{subject to} & \quad Ax = b \\
& \quad Fx \geq f \\
& \quad \partial Q_s x + \theta_s e \geq q_s, \quad s = 1, \ldots, n \\
& \quad x \geq 0
\end{align*}
\]

(16)

which progressively approximates (2) through the addition of cutting plane constraints. These constraints are determined by evaluating candidate decisions \( x_k \) on the \( n \) scenario problems, given by

\[
\begin{align*}
\text{minimize} & \quad Q^k_s = q^T_s y_s \\
\text{subject to} & \quad Wy_s = h_s - T_s x_k \\
& \quad y_s \geq 0 \quad (17)
\end{align*}
\]

We determine candidate decisions by solving the master problem for the current set of cutting planes. If the current candidate is not second-stage feasible, we add feasibility cuts to the master problem (16), through the constraint set \( F_k = Fx \geq f \), according to Algorithm 1.

\begin{algorithm}
\textbf{function} \text{CHECKFEASIBILITY}(x_k, F_k, \ldots)
\begin{algorithmic}
\State for all scenarios \( s = 1, \ldots, n \) do
\State \text{minimize} \quad w_s = e^T v_s^+ + e^T v_s^-
\State \quad P_{f,s}(x_k) \leftarrow \text{solve}(S_{f,s}(x_k))
\State \quad Wy_s + Iv_s^+ - Iv_s^- = h_s - T_s x_k
\State \quad y_s \geq 0, \quad v_s^+ \geq 0, \quad v_s^- \geq 0
\State \quad w_s \leftarrow \text{solve}(S_{f,s}(x_k))
\If{\( w_s > 0 \)}
\State \( \sigma_s \leftarrow \text{GETDUAL}(S_{f,s}(x_k)) \)
\State \( F_k \leftarrow \text{ADDFEASIBILITYCUT}(F_k, \sigma_s^T T_s x_k \geq \sigma_s^T h_s) \)
\EndIf
\EndFor
\end{algorithmic}
\textbf{end function}
\end{algorithm}

The authors of [27] show that a finite number of feasibility cuts are required to represent \( \mathcal{X}_r \) in the master problem. Therefore, candidate decisions will eventually become second-stage feasible in the L-shaped procedure if \( \mathcal{X}_r \neq \emptyset \). If a feasible candidate is not second-stage optimal, optimality cuts are added to the master problem (16), through the constraint sets \( O_k = \partial Q_s x + \theta_s e \geq q_s, \quad s = 1, \ldots, n \). An optimality cut related to scenario \( s \) is given by

\[
\pi_s \lambda_s^T T_s x + \theta_s \geq \pi_s \lambda_s^T h_s \quad (18)
\]

where \( \lambda_s \) is the optimal dual variables for the constraints in (17). Cutting planes of this type form supports for the second-stage objective \( Q(x) = \sum_{s=1}^{n} \pi_s Q_s(x) \) [27]. It follows that the master problem objectives

\[
\Theta_k = c^T x_k + \sum_{s=1}^{n} \theta^k_s
\]

(19)

forms a lower bound for the optimal value of (2). Moreover, the subproblem objective

\[
Q_k = c^T x_k + \sum_{s=1}^{n} Q^k_s
\]

(20)

forms an upper bound for the optimal value of (2). Thus, a termination criterion for the L-shaped procedure is given by

\[
|Q_k - \Theta_k| \leq \tau (\epsilon + |Q_k|) \quad (21)
\]

where \( \tau \) is some desired relative tolerance and \( \epsilon \) is a small number to avoid division by zero.
In total, SPjl includes four variants of the L-shaped algorithm, each with a distributed version that runs in parallel. The algorithms are implemented in a separate Julia module and are made available through using LShapedSolvers. A generic description of the L-shaped algorithm implemented in SPjl is given in Algorithm \[\text{1.2}\] We implement each algorithm as a specialization of Algorithm \[\text{2}\] with particular implementations of the FORMULATE_MASTER and TAKE_STEP functions.

**Algorithm 2** Generic L-Shaped Implementation

\[
\begin{align*}
\text{input:} & \quad \min_{x \in \mathbb{R}^n, y_k \in \mathbb{R}^m} c^T x + \sum_{s=1}^n \pi_s q_{ks}^T y_s \\
& \quad \text{s.t.} \quad A x = b, \quad T_s x + W y = h_s, \quad s = 1, \ldots, n \\
& \quad x \geq 0, \quad y_s \geq 0, \quad s = 1, \ldots, n \\
\text{output:} & \quad \hat{x}, \text{optimal to } P, \text{with optimal value } Q^*.
\end{align*}
\]

\textbf{procedure L-SHAPED}

\[
\begin{align*}
& k \leftarrow 0 \\
& x_0 \leftarrow \text{CRASH}(P) \\
& Q_0 \leftarrow \infty \\
& \Theta_0 \leftarrow -\infty \\
& \text{for } k = 0, 1, 2, \ldots \\
& \quad \text{end for}
\end{align*}
\]

\[
\begin{align*}
& \text{if } T_{F_{k-1}} = F_k \text{ then} \\
& \quad \text{for all scenarios } s = 1, \ldots, n \text{ do} \\
& \quad \quad \text{minimize } q_{ks}^T y_s \\
& \quad \quad \text{s.t.} \quad W y_s = h_s - T_s x_k \\
& \quad \quad y_s \geq 0 \\
& \quad \quad Q_k^s \leftarrow \text{SOLVE}(P_s(x_k)) \\
& \quad \quad \lambda_s \leftarrow \text{GETCONSTRAINTDUAL}(P_s) \\
& \quad \quad \Theta_k \leftarrow \text{ADDOPTIMALCUT}(\Theta_{k-1}, \pi_s \lambda_s^T T_s x + \theta_s \geq \pi_s \lambda_s^T h_s) \\
& \quad \text{end for} \\
& \quad Q_k \leftarrow \sum_{s=1}^n Q_k^s \\
& \quad \text{if } |Q_k - \Theta_{k-1}| \leq \tau (1 + |Q_k|) \text{ then} \\
& \quad \quad \hat{x} \leftarrow x_k \\
& \quad \quad Q^* \leftarrow Q_k \\
& \quad \text{return } \hat{x}, Q^* \text{ end if} \\
& \text{end if} \\
& \text{end if} \\
& M_{x_k} \leftarrow \text{FORMULATE_MASTER}(x_k, P_k, Q_k) \\
& \hat{x}_k, \Theta_k \leftarrow \text{SOLVE}(M_{x_k}) \\
& x_{k+1} \leftarrow \text{TAKE_STEP}(x_k, \hat{x}_k, Q_k, \Theta_k) \text{ end for}
\end{align*}
\]

Each algorithm is represented by a Julia object in the LShapedSolver module. For convenience, there is a factory function, LShapedSolver, that interfaces with stochastic programs created in SPjl. The function signature of the factory function is shown in Listing \[\text{17}\]

The \texttt{variant} symbol specifies which L-shaped algorithm variant the factory should return. A third-party solver, capable of solving the corresponding master problem, must be supplied as the \texttt{lp solver} argument. Each L-shaped algorithm in SPjl is associated with a set of parameters that may require tuning in specific problems. These parameters are specified through keyword arguments in the factory function. If a problem is known to have relatively complete recourse, we can entirely avoid the feasibility checking step as there is no need for feasibility cuts. For simplicity, this is the default setting. If the user encounters a problem that becomes second-stage infeasible as the algorithm progresses, then checkfeas =\texttt{true} can be supplied to the factory method to enable feasibility cuts.

### 3.1.1 Nominal L-shaped

The nominal L-shaped implementation in SPjl is based on the multicut L-shaped algorithm [4]. The master problem is given by \[\text{16}\], so that the \texttt{FORMULATE_MASTER} function is implemented as shown in Algorithm \[\text{5}\].
The optimal solution to the master problem gives a new candidate decision. Hence, the TakeStep function is a no-operation, as shown in Algorithm 4.

3.1.2 Regularizations of the L-shaped method

Different regularization procedures [14, 25, 7] can improve the performance of the L-shaped algorithm. Common to these procedures is that the candidate search is limited to a neighborhood of the current best iterate in the master problem. The result is more effective cutting planes and thereby faster convergence. Moreover, regularization enables the use of initial decisions for warm-starting the L-shaped procedure. We implement the regularized variants through specializations of the FormulateMaster and TakeStep functions. A thorough review, as well as numerical tests, of the regularization procedures implemented in LShapedSolvers.jl is given in [2].

3.1.3 Cut aggregation

The authors of [4] show that that the multi-cut variant of L-shaped, in general, converges in fewer iterations than the classical L-shaped method [27]. However, there is computational merit to aggregating cuts, especially in the distributed setting where a multi-cut approach infers a lot of data passing. Consider $m$ scenario aggregates of type

$$
A_i \subset \{1, \ldots, n\}, \quad i = 1, \ldots, m
$$

where the idea is that cuts generated from subproblems of index $j \in A_i$ are aggregated together into a single cut. If

$$
\bigcup_{i=1}^{m} A_i = \{1, \ldots, n\}
$$

then the results in [27] naturally extend so that

$$
\sum_{s \in A_i} \pi_s \lambda^T_s T_s x + \theta_i \geq \sum_{s \in B_i} \pi_s \lambda^T_s h_s
$$

form supports for the second-stage objective and

$$
\Theta_k = c^T x_k + \sum_{i=1}^{m} \theta^k_i
$$

forms a lower bound for the optimal value of (2). An effect of using aggregated cuts is that the size of the master problem does not grow as rapidly during the L-shaped procedure. Moreover, cut bundling could significantly reduce communication overhead in a distributed setting, since the bundling effort operates locally. The caveat is that the master has less information available each iteration, which in general leads to an increased number of iterations required to converge.

LShapedSolvers includes an automatic procedure for cut bundling. The user sets a uniform size $|A|$ of the aggregates, and SPjl constructs the master problem appropriately. The aggregation procedure naturally extends to the distributed setting, where extra aggregates might be added to ensure coverage if scenarios or not evenly distributed. The codebase can easily be extended to support more advanced features such as irregular aggregate sizes and advanced selection of which subproblems to aggregate.
3.1.4 Distributed L-shaped

Each of the L-shaped algorithms in SPj can operate in parallel on distributed data. A master node solves the master problem [16], and then communicates the resulting solution candidate to worker nodes. With respect to the current master solution, each worker solves every subproblem [17] that it administers, generates cutting planes and sends them back to the master node. An asynchronicity parameter, $\kappa$, determines the fraction of cutting planes the master must receive to generate a new iterate. When $\kappa < 1$, the master and worker operations can overlap at the cost of having less cut information in the master iterations. A simplified description of the distributed L-shaped algorithm is given in Algorithm 5 and a description of the worker tasks is given in Algorithm 6. A more in-depth description of the software implementation is given in Section 4.3.2.

Algorithm 5 Generic Distributed L-Shaped Implementation

```plaintext
procedure L-SHAPED
    $k \leftarrow 0$
    $t_0 \leftarrow 0$
    $x_0 \leftarrow \text{CRASH}(P)$
    $D \leftarrow \text{ADDDecision}(x_0)$
    $C \leftarrow \text{INITIALIZECutQueue}$
    $Q_k \leftarrow \infty$
    $\Theta_k \leftarrow -\infty$
    $W \leftarrow \text{INITIALIZEWorkers}(D, C)$
    $\text{QUEUEWork}(W, 0)$
repeat
    if $\text{CUTSReady}$ then
        for all $k, Q_k, C_k$ in $C$ do
            $Q_k \leftarrow Q_k$
            $t_k \leftarrow t_k + 1$
            $\Theta_k \leftarrow \text{ADDCut}(C_k)$
        end for
    end if
    if $t_k = \kappa n$ then
        $Q_k \leftarrow \sum_{i=1}^{n} Q_{k,i}$
        if $(Q_k - \Theta_{k-1}) \leq \tau(|\Theta| + |Q_k|)$ then
            $\hat{x} \leftarrow x_k$
            $Q^* \leftarrow Q_k$
            return $\hat{x}, Q^*$
        end if
    end if
    if $t_k \geq \kappa n$ then
        $M_k \leftarrow \text{FORMULATEMASTER}(x_k, C_k)$
        $\hat{x}_k, \Theta_k \leftarrow \text{SOLVE}(M_k)$
        $x_{k+1} \leftarrow \text{TAKESTEP}(x_k, \hat{x}_k, Q_k, \Theta_k)$
        $D \leftarrow \text{ADDDecision}(x_{k+1})$
        $\text{QUEUEWork}(W, k + 1)$
        $k \leftarrow k + 1$
    end if
until done
end procedure
```

▷ Workers can start working on the initial decision
▷ Workers have queued new cuts
▷ No more work on timestamp $k$
▷ Enough work done to proceed
▷ $\Theta_k = \sum_{i=1}^{n} \Theta_{k,i}$
▷ Send new decision vector to the workers

Algorithm 6 Simplified L-shaped Worker Task

```plaintext
function DoWork($W, D, C$)
repeat
    if WORKReady then
        $k \leftarrow \text{TAKEWORK}$
        $x_k \leftarrow \text{GETDecision}(D, k)$
        for all local scenarios $s = 1, \ldots, \hat{n}$ in $W$ do
            minimize $q^T y_s$
            subject to $W y_s = h_s - T_s x_k$
            $Q^T y_s \leftarrow \text{SOLVE}(P_s(x_k))$
            $\lambda_s \leftarrow \text{GETConstraintDual}(P_s)$
            $C \leftarrow \pi_s \lambda_s^T T_s x + \Theta_s \geq \pi_s \lambda_s^T h_s$
            $\text{QUEUECut}(C, k, Q^T y_s, C)$
        end for
    end if
until done
end function
```

▷ Master has generated a new iterate
3.2 Progressive Hedging

The second solver family in SPjl, `ProgressiveHedgingSolvers.jl`, is based on the progressive-hedging algorithm [24]. In contrast to the L-shaped algorithm, applying progressive-hedging to solve (2) yields a complete decomposition over the $n$ scenarios. The method is a specialization of the proximal-point algorithm [23], and convergence in the linear case (2) is derived in this setting in [24]. Here, we convey the main idea in terms of the established notation.

First, consider

$$\begin{align*}
& \text{minimize} & & \sum_{s=1}^{n} \pi_s c^T x_s + \pi_s q_s^T y_s \\
& \text{subject to} & & x_s = \xi, & s = 1, \ldots, n \\
& & & A x_s = b, & s = 1, \ldots, n \\
& & & T_s x_s + W y_s = h_s, & s = 1, \ldots, n \\
& & & x_s \geq 0, & s = 1, \ldots, n \\
& & & y_s \geq 0, & s = 1, \ldots, n
\end{align*}$$

(25)

The constraints $x_s = \xi$, $s = 1, \ldots, n$ are called non-anticipative and enforce the fact that the first-stage decision is given when the second-stage uncertainty is realized. Separability across the $n$ scenarios is achieved by introducing the following regularized relaxation of each subproblem:

$$\begin{align*}
& \text{minimize} & & c^T x_s + q_s^T y_s + \rho_s (x_s - \xi) + \frac{r}{2} \|x_s - \xi\|_2^2 \\
& \text{subject to} & & A x_s = b \\
& & & T_s x_s + W y_s = h_s \\
& & & x_s \geq 0, & s = 1, \ldots, n \\
& & & y_s \geq 0
\end{align*}$$

(26)

The algorithm now proceeds by iteratively alternating between generating new admissible solutions $x^k_s$, $s = 1, \ldots, n$ and an implementable solution $\xi_k$. In the two-stage setting, an admissible solution is feasible in every scenario, and an implementable solution is consistent in the sense that $x_s = \xi$ for all $s$. We obtain the implementable solution through aggregation

$$\xi_k = \sum_{s=1}^{n} \pi_s x^k_s$$

and the Lagrange multipliers are updated scenario-wise through

$$\rho^{k+1}_s = \rho^k_s + r(x^k_s - \xi_k)$$

Hence, the non-anticipative constraints are enforced as the dual variables converge. Numerical convergence is measured using

$$\delta_k = \sqrt{\|\xi_k - \xi_{k-1}\|_2^2 + \sum_{s=1}^{n} \pi_s \|x^k_s - \xi_k\|_2^2}$$

(27)

which takes both primal and dual convergence into account. Thus, a termination criterion for the progressive-hedging procedure can be formulated as

$$\delta_k \leq \tau (\epsilon + \|\xi_k\|_2)$$

(28)

where $\tau$ is some desired relative tolerance and $\epsilon$ is a small number to avoid division by zero.

SPjl includes two versions of the progressive-hedging method; One with a fixed penalty and one with an adaptive penalty. Distributed variants are also available. These algorithms are, like the L-shaped methods, implemented in a separate Julia module and they are made available through `using ProgressiveHedgingSolvers`. A generic description of the progressive-hedging algorithm implemented in SPjl is given in Algorithm 7.

### 3.2.1 Adaptive penalty parameter

The numerical performance of the progressive-hedging algorithm is in practice governed by choice of the penalty parameter $r$. A low/high value of $r$ is expected to encourage primal/dual convergence and a balanced choice is required to achieve good primal and dual convergence. Several suggestions for dynamic update procedures for $r$ have been proposed. A survey of such approaches, as well as a suggested adaptive update procedure, is provided in [29]. The adaptive update procedure is heuristic, but improved convergence is observed in several numerical examples. Listing 8 outlines a similar adaptive update procedure implemented in SPjl.
Algorithm 7 Progressive Hedging Implementation

```plaintext
input:
\[ \text{minimize}_{x \in \mathbb{R}^n, y_s \in \mathbb{R}^m} c^T x + \sum_{s=1}^{n} \pi_s q_s^T y_s \]
\[ P \leftarrow \text{s.t.} A x = b \]
\[ T_s x + W y_s = h_s, \quad s = 1, \ldots, n \]
\[ x \geq 0, y_s \geq 0, \quad s = 1, \ldots, n \]

output: \( \hat{x} \), optimal to \( P \), with optimal value \( Q^* \).

procedure PROGRESSIVE-HEDGING

\( k \leftarrow 0 \)
\( \xi_0 \leftarrow \text{CRASH}(P) \)
\( Q_0 \leftarrow \infty \)
for \( k = 0, 1, 2, \ldots \) do

for all scenarios \( s = 1, \ldots, n \) do

\[ P_s(\xi_k) \leftarrow \text{SOLVE}(P_s(\xi_k)) \]
\[ \pi_s x_s^k \leftarrow \text{ARGMIN}(P_s(\xi_k)) \]
end for
\( Q_k \leftarrow \sum_{s=1}^{n} \pi_s Q_s^k \)
\( \xi_k \leftarrow \sum_{s=1}^{n} \pi_s x_s^k \)

for all scenarios \( s = 1, \ldots, n \) do

\( \rho_{s}^{k+1} \leftarrow \rho_{s}^{k} + \tau(x^k - \xi_k) \)
\( \delta^k_s \leftarrow \|x^k_s - \xi_k\|_2 \)
end for
\( \delta_k \leftarrow \sqrt{\|\xi_k - \xi_{k-1}\|_2^2 + \sum_{s=1}^{n} \pi_s \delta^k_s} \)
if \( \delta_k \leq \tau(\epsilon + \|\xi_k\|) \) then

\( \hat{x} \leftarrow \xi_k \)
\( Q^* \leftarrow Q_k \)

return \( \hat{x}, Q^* \)
end if
\( r_{k+1} \leftarrow \text{ADAPT-PENALTY}(r_k) \)
end for
end procedure
```

Algorithm 8 Adaptive Penalty Update

```plaintext
function ADAPT-PENALTY(r_k)
\( \gamma_1, \gamma_2, \gamma_3 \in (0, 1), \quad \alpha, \nu, \epsilon \in (0, 1), \quad 1 < \theta < \beta < \eta \)
\( \delta_1 \leftarrow \|\xi_k - \xi_{k-1}\|_2 \)
\( \delta_2 \leftarrow \sum_{s=1}^{n} \pi_s \|x_s^k - \xi_k\|_2 \)
\( \delta_3 \leftarrow \sum_{s=1}^{n} \pi_s \|x_s^{k-1} - \xi_{k-1}\|_2 \)
if \( \delta_1 \geq \gamma_1(\epsilon + \|\xi_k\|) \) then

\( \mu \leftarrow \alpha \)
else if \( \delta_2 > \gamma_2 \max\{1, \delta_1\} \) then

\( \mu \leftarrow \alpha \)
else

\( \mu \leftarrow 1 \)
else if \( \delta_3 > \delta_1 \) then

\( \mu \leftarrow \beta \)
else if \( \delta_2 > \delta_3 \) then

\( \mu \leftarrow \beta \)
else

\( \mu \leftarrow \eta \)
end if
\( r_{k+1} \leftarrow \mu r_k \)
end function
```

Efficient Stochastic Programming in Julia
3.2.2 Distributed progressive-hedging

The progressive-hedging algorithm is embarrassingly parallel in the sense that the subproblems are independent. However, the algorithm includes two blocking reduction operations that require synchronization. These operations are the aggregation of the primal implementable solution $\xi_k$ and the calculation of the dual gaps $\delta_k^s$, which also depends on $\xi_k$. The reductions are carried out each iteration $k$. Consequently, a synchronous version of progressive-hedging, that operates in parallel on distributed data is straightforward to implement. In each iteration, worker nodes solve the progressive-hedging subproblems (26) in parallel. A master node synchronizes when all workers have finished, administers the necessary reduction operations and possibly adapts the penalty parameter, and then sends a new implementable decision candidate to the workers.

SPjl also includes a preliminary design of a distributed progressive-hedging algorithm that can be run asynchronously, using a scheme similar to the one described in Algorithm 5 for the distributed L-shaped method. The idea is that workers keep running averages of local primal aggregates and dual gaps. The master node is allowed to initiate the reduction operations with some admissible solutions $x_s$ possibly being out of date. This could allow slow workers to finish their tasks while synchronization is taking place, leading to useful overlapping computations. Such a procedure can be similarly tuned using an asynchronicity parameter, $\kappa$, that governs how many subproblems must be solved at an iteration for the master to initiate the reductions. This is ongoing work, but an implementation of this scheme has been applied successively on small test problems. A similar but more involved scheme is presented in a working paper [6] along with convergence guarantees based on previously presented theory. Future work could involve showing correspondence with the simple asynchronous scheme in SPjl as a transitive proof of convergence.

4 Implementation details

In this section, we provide a summary of the main software innovations in SPjl. The inner workings of SPjl are primarily based on two ideas: deferred model instantiation and data injection. In brief, a model definition in SPjl is a recipe for how to use data structures when building optimization models. This concept is implemented using the powerful metaprogramming tools available in the Julia language. When a specific model is instantiated, scenario data is injected where it is required to construct second-stage models. The main effect of this approach is that the stochastic model formulation is separated from the design of stochastic data parameters, which makes the SPjl framework versatile and flexible to use. Both concepts play a large role in distributing stochastic program instances in memory. Moreover, they allow the user to properly distinguish between the base model (1) and finite models (2). We also discuss the implementation of the distributed capabilities of SPjl.

4.1 Deferred model instantiation

In contrast to standard JuMP models, SPjl models defined through the @stage macros are not necessarily instantiated immediately. Instead, the user-defined Julia code that constructs the optimization problems is stored in lambda functions as model recipes. In other words, instead of creating and storing a JuMP object, the lines of code required to create the JuMP object is stored. This is achievable since Julia code is itself a data structure defined within the Julia language. Apart from having a smaller memory footprint, deferred model instantiation enables the creation of many useful structures that use the first- and second-stage recipes as building blocks. This technique is also a premise for implementing data injection.

One significant syntax restriction is that the user must annotate the first-stage variables that show up in the second stage with @decision at the top of the second @stage block, as exemplified in Listing 2. This is used internally in SPjl to form a bridge between the first and second stage. All references to first-stage variables are replaced by extraction calls to an internal parent model. This parent model originates from the first @stage definition. This technique, combined with the defined generator functions, is employed in SPjl to construct various auxiliary model objects. These helper models are then used to produce the different constructs introduced in Section 2.

4.1.1 Decoupled models

By default, an SPjl model is stored in parts. An instance of the first-stage problem

\[
\begin{align*}
\text{minimize} \quad & c^T x \\
\text{s.t.} \quad & Ax = b \\
& x \geq 0
\end{align*}
\]


19
is created using the corresponding generator and is always stored and available on the master node. The second-stage subproblem

\[
\begin{align*}
\text{minimize} & \quad q_s^T y_s \\
\text{s.t.} & \quad W y_s = h_s - T_s x_k \\
& \quad y_s \geq 0
\end{align*}
\]

is created for each possible scenario using the second-stage generator. The subproblems are either stored in vector format on the master node or distributed on remote nodes as described in Section 4.3.1. If new scenarios are added to the model through `add_scenario!`, `add_scenarios!`, or `sample!`, then new subproblems are automatically generated and stored appropriately. We distribute new scenarios as evenly as possible on remote nodes to achieve load balance. This storage format is suitable for structured solver approaches, such as the L-shaped algorithm or progressive hedging.

### 4.1.2 Extensive form

We construct the extensive form of a finite model in steps using the stored model recipes. First, we generate the first-stage model in full using the corresponding generator. Next, we process all available scenarios iteratively. For each scenario, we run the second-stage generator with the first stage as the parent model and append the resulting subproblem the DEP model. Before generating the subsequent scenario problem, we internally annotate the variables and constraints from the current scenario \( s \) with \( s \) to keep them distinct. This labeling is visible in the printout shown in Listing 5. The DEP model is generated and cached if the user asks for it using the `DEP` command. If new scenarios are added, the cached DEP model is invalidated and is only regenerated when the user queries for it again. Calling `optimize!` with a supplied standard third-party solver will internally generate, if necessary, and solve a DEP model.

### 4.1.3 Outcome models

We perform decision evaluation by constructing an outcome model. Outcome models are associated with some decision vector and a scenario. First, we generate copies of the variable definitions from the first stage of the outcome model. Next, we fix these variables to the corresponding values in the supplied decision vector. We generate a copy of the second-stage problem corresponding to the given scenario in the outcome model. In this case, the outcome model itself acts as the parent model with fixed first-stage variables. Now, solving the outcome model for a decision \( \hat{x} \) and scenario \( \tilde{\xi} \) will consequently yield

\[ Q(\hat{x}, \tilde{\xi}) \]

Hence, formulating and solving outcome models for all available scenarios, as well as evaluating \( c^T \hat{x} \), gives exactly 5. Thus, the outcome model approach can be employed for both decision evaluation and the EEV calculation (13).

### 4.1.4 Wait-and-see models

The process for generating a wait-and-see model (10) \((WS)\) is equivalent to one iteration of the DEP model generation. In short, the first-stage generator is run, followed by the second-stage generator run on some supplied scenario data. The second-stage generator is run with the wait-and-see model as the parent model so that the first and second-stage variables are connected into one problem. Note that generation of the EV P model (12) is implemented by generating a WS model on the expected scenario of all available scenarios.

### 4.2 Data injection

Data injection is the second software pattern used to separate model and data design in SPjl. When an SPjl model is formulated, several annotations can be made to specify points of data injection. Consider the simple second-stage formulation in Listing 18. There are several points of data injection in the above formulation. The `@parameters` annotation specifies stage independent data, i.e., data parameters that are the same across all scenarios. The `@uncertain` annotation specifies the scenario-specific data, which is always some structure of type AbstractScenario. Finally, the `model` keyword is a placeholder for a JuMP object that will later contain the actual optimization model. The `@stage` macro transforms the given code block into an anonymous lambda function that has the reserved keyword names as its arguments. Internally, when the user wants to instantiate the defined SPjl model, the required data is passed to the stored lambda function. Specifically, using the simple definition above and a given scenario, the generator function will be passed an empty JuMP object, a value of \( d \), and the specific value of \( \xi \). The generator then returns a JuMP object that is a specific instance of the defined second-stage model.
The use of data injection results in multiple benefits. First, it enables efficient and flexible model instantiations. For instance, one can test small instances of a model locally to ensure that it is properly defined. The same model can then be run in production on a large set of scenarios, in a distributed environment, without changing the code and in the same interactive Julia session. A defined stochastic model object can be used to both compute a first-stage decision candidate and to instantiate outcome models to evaluate the decision. Second, it adds versatility to the framework. The user is only restricted to use the `model` keyword in the JuMP macro calls. Otherwise, all JuMP features are supported in the stage blocks. Also, there is no restriction on the scenario data types. Hence, instead of a simple structure with fields, it is possible to define a more complex data type that for example performs calculations at runtime to determine optimization parameters. Any Julia methods defined on the scenario type become available in the stage blocks. Finally, as the model definition is decoupled from the data, it is possible to send the model recipe to a remote process where scenario data resides. This forms the foundation of the distributed implementation described next.

### 4.3 Distributed computations

SPjl has distributed capabilities for both modeling, analysis and optimization. All implementations rely on the `Distributed` module in Julia. This has allowed us to develop SPjl using high-level abstractions and utilize the efficient low-level communication protocols in Julia. In this way, the same codebase can be used to distribute computations locally, using shared-memory, and remotely, in a cluster or in the cloud. We briefly describe these abstractions before discussing their usage in SPjl.

Distributed computing in Julia is centered around remote references and remote calls. Remote references are used to administer which node particular data resides on. Also, they provide the remaining processes access to the remote data. Remote calls are used to schedule tasks on the nodes. Any process can `wait` on a remote reference, which blocks until data can be fetched, and then `fetch` the result when it is ready. The `RemoteChannel` objects are special remote references where processes can also `put!` data. Besides, specialized channel objects can be designed for specific data types. This feature is used frequently in the implementation of the distributed structured solvers.

#### 4.3.1 Distributed stochastic programs

If multiple Julia processes are available, then any instantiated stochastic program in SPjl is automatically distributed in memory. A master node administers the first-stage problem and schedules tasks and data transfers. Every active worker node administers a `ScenarioProblems` channel, which is a specialized data type in SPjl that describes second-stage subproblems. A simplified definition of this type is shown in Listing 19. The `stage` field contains any scenario-independent data, the `scenarios` field stores the individual scenario structures, and the `problems` field stores instantiated JuMP models. The `parent` field stores the auxiliary parent model described in Section 4.1. The second stages of a stochastic program can in single memory be fully represented by one instance of `ScenarioProblems`. In principle, an instantiated program distributes in memory by partitioning and passing the internal data structures on every available worker node. However, a far more efficient approach is to let the workers generate the necessary scenario...
data and the optimization models themselves, with minimal data passing. This is possible since SPjl also allows to send a sampler object capable of randomly generating scenario data, such as the one in defined in Listing 6 and the model recipes created by the @stage blocks. Scenario data and subproblems can then be generated in parallel on the worker nodes. The master keeps track of the scenario distribution and ensures that new scenarios and subproblems are generated in a load-balanced fashion on available workers. This design aims to minimize the amount of data passing.

Aside from distributing the models in memory, SPjl parallelizes as many computations as possible. In many cases, speedups stem from subtasks being embarrassingly parallel over the independent subproblems. For example, this occurs during decision evaluation and calculation of EVPI and VSS. In these instances, the master schedules the same computation tasks on all workers using remote calls and then initiates any necessary reductions after the workers have finished. The more involved parallelization strategies in SPjl relate mostly to the structure-exploiting solvers, which we describe next.

4.3.2 Distributed structured optimization algorithms

The implementations of the distributed structured solvers are also centered around remote calls and channels. Remote calls are used to initiate running tasks on every worker node, and the algorithm logic is driven by having the master and worker tasks wait on and write/fetch to/from specialized queue channels.

In the case of the L-shaped method, whenever the master re-solves the master problem (16), it writes the new decision vector to a specialized Decision channel. It then sends a corresponding index to a Work channel on every remote node. Every worker continuously fetches tasks from its Work channel and uses the acquired index to fetch the latest decision vector from the master. Every new decision candidate infers a batch of subproblems to solve for each worker. After a worker has solved a subproblem (17), it sends the computed cutting planes to a CutQueue channel on the master. The master continuously fetches cuts from the CutQueue and appends them to the master problem. As soon as the master has received \( κn \) cuts, where \( n \) is the total number of subproblems, the master problem is re-solved, and the procedure continues. Timestamps are communicated throughout to keep track of the algorithm history and allow synchronized convergence checks. All subproblems are solved to completion each iteration regardless of the value of \( κ \), to be able to check convergence properly. When the master has received all cuts corresponding to a specific iteration, it performs a convergence check and terminates if appropriate. For clarity, the procedure is illustrated in Fig. 1.

A similar scheme is employed to implement asynchronous progressive-hedging. There is no need for a cut queue, but the workers maintain a specialized RunningAverage channel to update their local information. The master can fetch from these channels during reductions, update the Decision channel and send a new task index to the workers. Synchronous progressive-hedging does not require any queue objects. Instead, the algorithm is driven by the master node initiating and waiting for worker tasks through remote calls. As an alternative, setting \( κ \) to 1 also results in synchronous operations, possibly with some overhead from the extra channel objects.

5 Numerical examples

5.1 The farmer problem

To exemplify functional correctness, and allow for comparisons with similar tools, we consider the instructive farmer problem by Birge and Louveaux [3]. Listing 20 shows a suggested code excerpt for how the farmer problem can be defined in SPjl. A print output of the farmer problem is shown in Listing 21. Now, we solve the farmer problem and evaluate the stochastic performance of the model. The results are given in Listing 22. The correctness of the numerical values can be verified in [3]. Finally, we solve the farmer problem using an L-shaped solver and a progressive-hedging solver, as shown in Listing 23. Both solvers eventually converge to the correct result. The convergences of the L-shaped and progressive hedging algorithms are illustrated in Fig. 2.

5.2 Day-ahead problems

We evaluate distributed functionality and performance on a large-scale day-ahead planning problem. We used the same type of problem to benchmark SPjl’s L-shaped solvers in [2]. Here, we present updated benchmarks as well as performance tests of recently added SPjl features.

We repeat a brief description of the problem for self-containment. A day-ahead planning problem involves specifying optimal order volumes in a deregulated electricity market. In such markets, producers place orders which specify the amount of electricity that they are willing to produce at different price levels for the next day. The next-day market price determines which orders are accepted each hour. The market price is not known when the producer places the
(a) Master sends task to workers. Workers fetch latest decision vector.

(b) Workers solve subproblems and send cuts to master. Master sends new task to workers when new decision vector is ready.

(c) Convergence check when all cuts have been received. Ready workers fetch latest decision. Procedure continues.

Figure 1: Asynchronous L-shaped procedure
using StochasticPrograms
farmer_model = @stochastic_model begin
  @stage 1 begin
    @parameters begin
      Crops = [:wheat, :corn, :beets]
      Cost = Dict(:wheat => 150, :corn => 230, :beets => 260)
      Budget = 500
    end
    @variable(model, x[c=Crops] >= 0)
    @objective(model, Min, sum(Cost[c]*x[c] for c in Crops))
    @constraint(model, sum(x[c] for c in Crops) <= Budget)
  end
  @stage 2 begin
    @decision x
    @parameters begin
      Purchased = [:wheat, :corn]
      Sold = [:wheat, :corn, :bquota, :bextra]
      Required = Dict(:wheat => 200, :corn => 240, :beets => 0)
      PurchasePrice = Dict(:wheat => 238, :corn => 210)
      SellPrice = Dict(:wheat => 170, :corn => 150, :bquota => 36, :bextra => 10)
    end
    @uncertain ξ::YieldScenario = begin
      wheat::Float64
      corn::Float64
      beets::Float64
    end
    @variable(model, y[p=Purchased] >= 0)
    @variable(model, w[s=Sold] >= 0)
    @objective(model, Min, sum(PurchasePrice[p] * y[p] for p in Purchased) - sum(SellPrice[s] * w[s] for s in Sold))
    @constraint(model, const_minreq[p=Purchased], ξ[p] * x[p] + y[p] - w[p] >= Required[p])
    @constraint(model, const_minreq_beets, ξ[:beets] * x[:beets] - w[:bquota] - w[:bextra] >= Required[:beets])
    @constraint(model, const_aux, w[:bquota] <= 600)
  end
end
ξ₁ = YieldScenario(3.0, 3.6, 24.0, probability = 1/3)
ξ₂ = YieldScenario(2.5, 3.0, 20.0, probability = 1/3)
ξ₃ = YieldScenario(2.0, 2.4, 16.0, probability = 1/3)
farmer_problem = instantiate(farmer_model, [ξ₁, ξ₂, ξ₃])

Figure 2: Convergence progress for L-shaped and progressive-hedging.
Efficient Stochastic Programming in Julia

Listing 21: Print output of the farmer problem in SPjl.

```julia
julia> print(farmer_problem)
First-stage
==========
Min 150 x[wheat] + 230 x[corn] + 260 x[beets]
Subject to
  x[wheat] + x[corn] + x[beets] ≤ 500
  x[c] ≥ 0 ∀ c ∈ \{wheat, corn, beets\}
Second-stage
==========
Subproblem 1 (p = 0.33):
Min 238 y[wheat] + 210 y[corn] - 170 w[wheat] - 150 w[corn] - 36 w[beets_quota] - 10 w[beets_extra]
Subject to
  3 x[wheat] + y[wheat] - w[wheat] ≥ 200
  3.6 x[corn] + y[corn] - w[corn] ≥ 240
  24 x[beets] - w[beets_quota] - w[beets_extra] ≥ 0
  w[beets_quota] ≤ 6000
  y[p] ≥ 0 ∀ p ∈ \{wheat, corn\}
  w[s] ≥ 0 ∀ s ∈ \{wheat, corn, beets_quota, beets_extra\}
Subproblem 2 (p = 0.33):
Min 238 y[wheat] + 210 y[corn] - 170 w[wheat] - 150 w[corn] - 36 w[beets_quota] - 10 w[beets_extra]
Subject to
  y[wheat] - w[wheat] + 2.5 x[wheat] ≥ 200
  y[corn] - w[corn] + 3 x[corn] ≥ 240
  -w[beets_quota] - w[beets_extra] + 20 x[beets] ≥ 0
  w[beets_quota] ≤ 6000
  y[p] ≥ 0 ∀ p ∈ \{wheat, corn\}
  w[s] ≥ 0 ∀ s ∈ \{wheat, corn, beets_quota, beets_extra\}
Subproblem 3 (p = 0.33):
Min 238 y[wheat] + 210 y[corn] - 170 w[wheat] - 150 w[corn] - 36 w[beets_quota] - 10 w[beets_extra]
Subject to
  2 x[wheat] + y[wheat] - w[wheat] ≥ 200
  2.4 x[corn] + y[corn] - w[corn] ≥ 240
  16 x[beets] - w[beets_quota] - w[beets_extra] ≥ 0
  w[beets_quota] ≤ 6000
  y[p] ≥ 0 ∀ p ∈ \{wheat, corn\}
  w[s] ≥ 0 ∀ s ∈ \{wheat, corn, beets_quota, beets_extra\}
```

Listing 22: Solving and analyzing the farmer problem using Gurobi [10]

```julia
julia> using Gurobi
# Optimize stochastic program (through extensive form)
julia> optimize!(farmer_problem, solver = GurobiSolver())
:Optimal
# Inspect optimal decision
julia> # = optimal_decision(farmer_problem)
3-element Array{Float64,1}:
  170.0
  80.0
  250.0
# Inspect optimal value
julia> optimal_value(farmer_problem)
-108390.0
# Calculate expected value of perfect information
julia> EVPI(farmer_problem, solver = GurobiSolver())
7015.56
# Calculate value of the stochastic solution
julia> VSS(farmer_problem, solver = GurobiSolver())
1150.0
```
Listing 23: Solving the farmer problem using L-shaped and progressive-hedging, using Gurobi \cite{10} to solve emerging subproblems.

```julia
julia> using Gurobi, LShapedSolvers, ProgressiveHedgingSolvers
# Create subproblem solver
julia> gurobi = GurobiSolver()
# Solve with L-shaped
julia> optimize!(farmer_problem, solver = LShapedSolver(gurobi))
L-Shaped Gap Time: 00:00:00 (6 iterations)
Objective: -108389.99999999997
Gap: 5.370205822812747e-16
Number of cuts: 14
:Optimal
# Solve with progressive-hedging
julia> optimize!(farmer_problem, solver = ProgressiveHedgingSolver(gurobi))
Progressive Hedging Time: 00:00:00 (112 iterations)
Objective: -108390.02598823253
\( \delta \): 9.269354032255935e-7
:Optimal
```

Figure 3: Confidence intervals around optimal value of the day-ahead problem as a function of SAA sample size.

Therefore, a stochastic program can be formulated and solved to generate optimal order strategies. The first-stage decisions are hourly volume orders for the upcoming day placed on the day-ahead market. In each second stage scenario, the market price is known, and electricity production is optimized with respect to profits to satisfy the settled orders. In addition, the producer can take recourse decisions by trading surplus or shortage in an intraday electricity market. A more thorough introduction to day-ahead markets and the order strategy problem is given in \cite{8}.

We formulate a day-ahead planning problem in SPjI from the perspective of a fictional hydropower producer that owns all 15 power stations in the Swedish river Skellefteåd’ven. Physical specifications for these power stations is available in \cite{26}. A first-stage model is formulated to place bids, per regulations, on the Nordic market NordPool \cite{18}. Random price curves, defined using \texttt{@scenario}, are generated by sampling from a multivariate normally distributed model fitted to historical market prices from the NordPool market \cite{19}. The fitted model is stored in a lightweight sampler object, defined through \texttt{@sampler}, that we pass to every worker. We calculate confidence intervals around the optimum, using the method described in Section 2.4.2, for increasing sample sizes of the SAA models. The result is shown in Fig. 3.
Efficient Stochastic Programming in Julia

5.2.1 Distributed benchmarks

We evaluate the structured solvers by solving distributed day-ahead problems with 1000 scenarios. Benchmarks are performed using the Julia package BenchmarkTools.jl, which schedules multiple solve procedures and reports median computation times. Every solver runs until convergence criteria are reached with a relative tolerance of $10^{-6}$. The master node is a laptop computer with a 2.6 GHz Intel Core i7 processor and 16 GB of RAM. We spawn workers on a remote multi-core machine with two 3.1 GHz Intel Xeon processors (total 32 cores) and 128 GB of RAM. The two machines are connected to the same local network, so data passing latency is expected to be low. Throughout, the Gurobi solver \cite{Gurobi} is used to solve emerging subproblems.

First, we evaluate the L-shaped procedures. We benchmarked all regularization variants in \cite{2}. Here, we only benchmark the classical L-shaped method and the most efficient regularization procedure that makes use of trust-regions. The results of a strong scaling experiment are shown in Fig. 4.

We have improved the code since \cite{2} and the results now indicate better scalability and an overall decrease in computation time. Weaker scaling at higher worker counts is attributed to load imbalance between the master and the workers as the L-shaped procedure progresses. The master grows by 1000 cuts each iteration, and the time required to re-solve the problem is not reduced by adding more cores. To alleviate this issue, we make use of the cut bundling procedure. The experiment is repeated on 8 worker cores using only the trust-region variant. The results are shown in Fig. 5.

The reduction in master size and the reduced amount of data passed leads to speedups when the cuts are aggregated in groups of 10. Afterwards, there are apparent diminishing returns as the aggregation level increases, since the number of major iterations required to converge increase.

As in \cite{2}, the best performance is achieved when $\kappa = 1$, which corresponds to synchronous iterations. The main reason appears to be load imbalance between the master and workers again. Even when using bundling, the size of the master eventually exceeds the size of the subproblems. As a result, apart from the first iterations, the workers have time to solve all subproblems while the master problem is re-solved. This makes values of $\kappa < 1$ superfluous. Moreover, the subproblems appear equally difficult to solve, as there are no stalling workers. This also removes the benefit of allowing the master to asynchronously continue before all workers have finished.

Next, we evaluate the performance of the progressive-hedging solvers. Using the nominal method, we did not observe convergence even after waiting a longer time than required for solving the extensive form. Using the adaptive penalty variant described in Section 3.2.1 with the parameter values shown in Table \ref{table:parameters}, eventually yields convergence. With
Figure 5: Median computation time required for the L-shaped method with trust-region to solve a day-ahead problem with 1000 scenarios, as a function of number of cuts in each cut aggregate. The experiment was performed on 8 worker cores.

Table 1: Parameter settings for the progressive-hedging method with adaptive penalty.

| Parameter | Setting |
|-----------|---------|
| $\zeta$   | 0.01    |
| $\gamma_1$ | $10^{-5}$ |
| $\gamma_2$ | 0.01    |
| $\gamma_3$ | 0.25    |
| $\sigma$  | $10^{-5}$ |
| $\alpha$  | 0.75    |
| $\theta$  | 1.1     |
| $\nu$     | 0.1     |
| $\beta$   | 1.1     |
| $\eta$    | 1.25    |
| $\tau$    | $5 \cdot 10^{-6}$ |

these settings on the adaptive penalty strategy, we performed a strong scaling test of the synchronous distributed progressive-hedging method. The results are shown in Fig. 6.

Although at much worse time-to-solution than the L-shaped methods, the distributed progressive-hedging method displays great scaling, with 96% parallel efficiency at 16 worker cores. The efficiency probably stems from the problem being load-balanced across the workers. Again, the subproblems appear equally difficult as there were no stalling workers. Consequently, we did not observe any speedups from running the asynchronous variant.

The L-shaped methods outperform progressive-hedging methods on the day-ahead problems. This is not indicative of a general trend since the day-ahead problems can be solved without feasibility cuts, which is favourable for the L-shaped performance. Progressive-hedging has the benefit of implicitly handling second-stage feasibility since it decomposes completely over the scenarios, and could perform better on problems without relatively complete recourse. However, the time to convergence is still notably large, which implies the need for algorithmic improvements to the progressive-hedging methods. Further testing is also required for the asynchronous algorithms.

6 Conclusion

In this work, we have presented an open-source framework, SPjl, for large-scale stochastic programming. It is written entirely in Julia and includes both modeling tools and solver algorithms. The framework is designed for distributed computations and naturally scales to high-performance clusters or the cloud. By using the extensive form, which is efficiently generated using metaprogramming techniques, stochastic program instances can be solved using open-
source or commercial solvers. Through deferred model instantiation and data injection, the framework becomes flexible and can operate in a distributed architecture with minimal data passing. In addition, several analysis tools and stochastic programming constructs are included with efficient implementations, many of which can run in parallel.

The framework also includes a solver suite of scalable algorithms that exploit the structure of the stochastic programs. The structured solvers are shown to perform well on large-scale day-ahead planning problems. High parallel efficiency is achieved for distributed L-shaped methods using cut bundling techniques and regularizations. Also, progressive-hedging methods successfully converge using an adaptive penalty procedure. Like the L-shaped algorithms, distributed progressive-hedging methods scale well, with almost full parallel efficiency on 16 cores.

There are several directions for future additions to the framework. First, SPjl does not yet fully support multi-stage problems. We have finished an infrastructure for representing multi-stage problems in a way that leverages the two-stage design. Ongoing work involves designing a suitable Julian syntax for encoding transitive probabilities in a multi-stage scenario tree. Second, we will consider further algorithmic improvements to the existing L-shaped and progressive-hedging solvers. Moreover, extensions are required to be able to solve problems with integer or binary variables.

The framework is well-tested through continuous integration and is freely available on Github\(^1\). The solver packages, LShapedSolvers\(^2\) and ProgressiveHedgingSolvers\(^3\) are also freely available on Github. A comprehensive documentation is included\(^4\). The modeling framework, StochasticPrograms.jl, exists as a registered Julia package, which can be installed and included in any interactive Julia session.

References

[1] Bezanson, J., Edelman, A., Karpinski, S., Shah, V.B.: Julia: A fresh approach to numerical computing. SIAM Review 59(1), 65–98 (2017). DOI 10.1137/141000671

[2] Biel, M., Johansson, M.: Distributed l-shaped algorithms in julia. In: 2018 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM) (2018). DOI 10.1109/PAW-ATM.2018.00011

[3] Birge, J.R., Louveaux, F.: Introduction to Stochastic Programming. Springer New York (2011). DOI 10.1007/978-1-4614-0237-4

\(^1\)https://github.com/martinbiel/StochasticPrograms.jl
\(^2\)https://github.com/martinbiel/LShapedSolvers.jl
\(^3\)https://github.com/martinbiel/ProgressiveHedgingSolvers.jl
\(^4\)https://martinbiel.github.io/StochasticPrograms.jl/dev/
[4] Birge, J.R., Louveaux, F.V.: A multicut algorithm for two-stage stochastic linear programs. European Journal of Operational Research 34(3), 384–392 (1988). DOI 10.1016/0377-2217(88)90159-2

[5] Dunning, I., Huchette, J., Lubin, M.: JuMP: A modeling language for mathematical optimization. SIAM Review 59(2), 295–320 (2017). DOI 10.1137/15m1020575

[6] Eckstein, J., Watson, J.P., Woodruff, D.L.: Asynchronous projective hedging for stochastic programming (2018). http://www.optimization-online.org/DB_HTML/2018/10/6895.html

[7] Fábián, C.I., Szöke, Z.: Solving two-stage stochastic programming problems with level decomposition. Computational Management Science 4(4), 313–353 (2006). DOI 10.1007/s10287-006-0026-8

[8] Fleten, S.E., Kristoffersen, T.K.: Stochastic programming for optimizing bidding strategies of a nordic hydropower producer. European Journal of Operational Research 181(2), 916–928 (2007). DOI 10.1016/j.ejor.2006.08.023

[9] GrĂłwe-Kuska, N., RĂłmsisch, W.: Stochastic unit commitment in hydrothermal power production planning. In: Applications of Stochastic Programming, pp. 633–653. Society for Industrial and Applied Mathematics (2005). DOI 10.1137/1.9780898718799.ch30

[10] Gurobi Optimization, L.: Gurobi optimizer reference manual. http://www.gurobi.com

[11] Hart, W.E., Laird, C.D., Watson, J.P., Woodruff, D.L., Hackebeil, G.A., Nicholson, B.L., Sirola, J.D.: Pyomo — Optimization Modeling in Python. Springer International Publishing (2017). DOI 10.1007/978-3-319-58821-6

[12] Huchette, J., Lubin, M., Petra, C.: Parallel algebraic modeling for stochastic optimization. In: 2014 First Workshop for High Performance Technical Computing in Dynamic Languages. IEEE (2014). DOI 10.1109/hptcdl.2014.6

[13] Krokhmal, P., Uryasev, S., Zrazhevsky, G.: Numerical comparison of conditional value-at-risk and conditional drawdown-at-risk approaches: Application to hedge funds. In: Applications of Stochastic Programming, pp. 609–631. Society for Industrial and Applied Mathematics (2005). DOI 10.1137/1.9780898718799.ch29

[14] Linderoth, J., Wright, S.: Decomposition Algorithms for Stochastic Programming on a Computational Grid. Computational Optimization and Applications 24(2-3), 207–250 (2003). DOI 10.1023/A:1021858008222. https://link.springer.com/article/10.1023/A:1021858008222

[15] Lubin, M., Hall, J.A.J., Petra, C.G., Anitescu, M.: Parallel distributed-memory simplex for large-scale stochastic LP problems. Computational Optimization and Applications 55(3), 571–596 (2013). DOI 10.1007/s10589-013-9542-y. https://link.springer.com/article/10.1007/s10589-013-9542-y

[16] Mak, W.K., Morton, D.P., Wood, R.: Monte carlo bounding techniques for determining solution quality in stochastic programs. Operations Research Letters 24(1), 47 – 56 (1999). DOI https://doi.org/10.1016/S0167-6377(98)00054-6. URL http://www.sciencedirect.com/science/article/pii/S0167637798000546

[17] Makhorin, A.: Gnu linear programming kit. https://www.gnu.org/software/glpk/

[18] NordPool: https://www.nordpoolgroup.com/ (2018)

[19] NordPool: Hourly Elspot prices 2017 in EUR. https://www.nordpoolgroup.com/globalassets/marketdata-excel-files/elspot-prices_2017_hourly_eur.xls (2018)

[20] Petra, C.G., Schenck, O., Anitescu, M.: Real-Time Stochastic Optimization of Complex Energy Systems on High-Performance Computers. Computing in Science Engineering 16(5), 32–42 (2014). DOI 10.1109/MCSE.2014.53

[21] Powell, W.B.: An operational planning model for the dynamic vehicle allocation problem with uncertain demands. Transportation Research Part B: Methodological 21(3), 217–232 (1987). DOI 10.1016/0191-2615(87)90005-1

[22] Powell, W.B., Topaloglu, H.: Fleet management. In: Applications of Stochastic Programming, pp. 185–215. Society for Industrial and Applied Mathematics (2005). DOI 10.1137/1.9780898718799.ch12

[23] Rockafellar, R.T.: Monotone operators and the proximal point algorithm. SIAM Journal on Control and Optimization 14(5), 877–898 (1976). DOI 10.1137/0314056

[24] Rockafellar, R.T., Wets, R.J.B.: Scenarios and policy aggregation in optimization under uncertainty. Mathematics of Operations Research 16(1), 119–147 (1991). DOI 10.1287/moor.16.1.119

[25] RuszczyÅński, A.: A regularized decomposition method for minimizing a sum of polyhedral functions. Mathematical Programming 35(3), 309–333 (1986). DOI 10.1007/BF01580883. https://link.springer.com/article/10.1007/BF01580883
[26] Sag, J.J.: Simulation of hydro power expansion in skellefteälven. Master’s thesis, KTH, Optimization and Systems Theory (2018)

[27] Van Slyke, R., Wets, R.: L-Shaped Linear Programs with Applications to Optimal Control and Stochastic Programming. SIAM Journal on Applied Mathematics 17(4), 638–663 (1969). DOI 10.1137/0117061. http://epubs.siam.org/doi/abs/10.1137/0117061

[28] Watson, J.P., Woodruff, D.L., Hart, W.E.: PySP: modeling and solving stochastic programs in python. Mathematical Programming Computation 4(2), 109–149 (2012). DOI 10.1007/s12532-012-0036-1

[29] Zehtabian, S., Bastin, F.: Penalty parameter update strategies in progressive hedging algorithm. CIRRELT (2016)

[30] Zenios, S.A.: Optimization models for structuring index funds. In: Applications of Stochastic Programming, pp. 471–501. Society for Industrial and Applied Mathematics (2005). DOI 10.1137/1.9780898718799.ch24