

Why a SDDP framework is a big deal?

Alternatives

- FAST (Finally An SDDP Toolbox)
- StochDynamicProgramming.jl
- StructDualDynProg.jl

Why SDDP.jl (Oscar Dowson)

- Easy to use
- Easy to extend
- Many features

[The original Oscar Downson's presentation \(https://github.com/odow/talks/blob/master/sddp_jl.ipynb\)](https://github.com/odow/talks/blob/master/sddp_jl.ipynb)

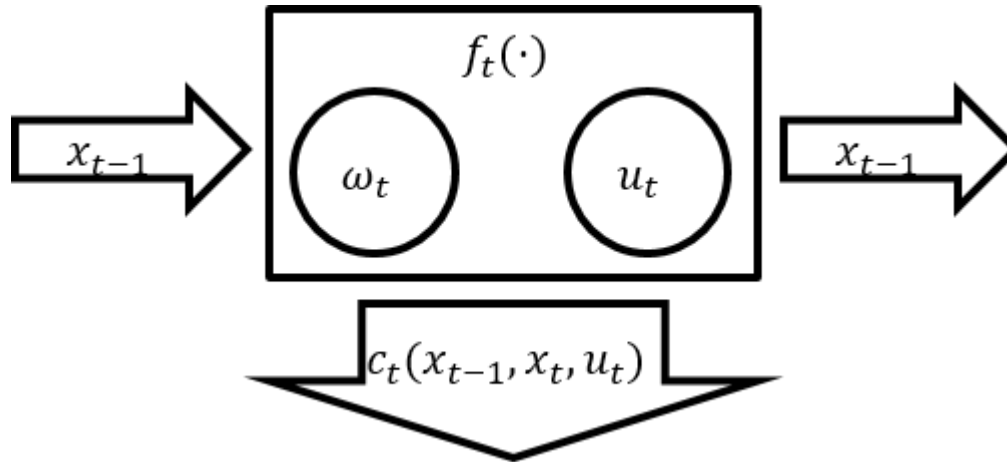
SDDP.jl - A Flexible SDDP Library

What we can do with it:

- Multistage stochastic linear program in discrete time
- RHS uncertainty (scenarios)
- Markov uncertainty
- Risk neutral or risk averse

What are we talking about

A stage has six things



1. An incoming state x_{t-1}
2. An outgoing state x_t
3. Uncertainty that is realised at the beginning of the state ω_t
4. An action that is taken u_t
5. Some dynamics $x_t = f_t(x_{t-1}, u_t, \omega_t)$
6. A reward that is earned $c_t(x_{t-1}, x_t, u_t)$

$$\begin{aligned}
SP_t(\bar{x}_{t-1}, \omega_t) : \quad & \min_{u_t} \quad c_t(x_{t-1}, x_t, u_t) + \theta_{t+1} \\
\text{s.t.} \quad & x_{t-1} = \bar{x}_{t-1} & [\pi_t(\omega_t)] \\
& x_t = f_t(x_{t-1}, u_t) \\
& u_t \in U_t(x_{t-1}, \omega_t)
\end{aligned}$$

SP_t is a user defined JuMP model.

Where this might differ

- If I record 6 different states (initial, + five more), there are five stages, not six;
- Wait-and-See in a stage. You take an action today after realising the uncertainty(hazard-decision);
- Each stage is set-up as a linear programme.

We call the linear programme that defines a stage a *subproblem*.

```
In [1]: # To get started we need to clone SDDP.jl
        Pkg.clone("https://github.com/odow/SDDP.jl")

        # load some packages
        using SDDP, JuMP, Clp
```

The stock example

Links to [StochasticDynamicProgramming.jl](https://github.com/JuliaOpt/StochasticDynamicProgramming.jl/blob/master/examples/stock-example.jl) (<https://github.com/JuliaOpt/StochasticDynamicProgramming.jl/blob/master/examples/stock-example.jl>) and [SDDP.jl](https://github.com/odow/SDDP.jl/blob/master/examples/StochasticDynamicProgramming.jl/stock-example.jl) (<https://github.com/odow/SDDP.jl/blob/master/examples/StochasticDynamicProgramming.jl/stock-example.jl>) versions.

1. Sense: Minimising
2. Stages: 5 stages ($t = 1, 2, 3, 4, 5$)
3. States: 1 State $x_t \in [0, 1]$ (initial state $x_0 = 0.5$)
4. Controls: 1 control $u_t \in [0, 0.5]$
5. Noises: 10 stagewise independent noises:
 $\omega_t \in [0, 0.0333..., 0.0666..., \dots, 0.3]$
6. Dynamics: linear dynamics $x_t == x_{t-1} + u_t - \omega_t$
7. Stage Objective: linear objective $(\sin(3t) - 1) \cdot u_t$

$$\min_{u_t} \quad (\sin(3t) - 1)u_t + \theta_{t+1}$$

$$s. t. \quad x_t = x_{t-1} + u_t - \omega_t$$

$$x_t \in [0, 1]$$

$$u_t \in [0, 0.5]$$

$$x_0 = 0.5$$

Syntax for creating a new SDDPModel

We define 1. and 2. in the constructor using keyword arguments.

```
m = SDDPModel(  
    sense = :Min,          # :Max or :Min?  
    stages = 5,            # Number of stages  
    solver = ClpSolver(),  
    objective_bound = -2# Valid lower bound  
) do sp, t  
    # ) do subproblem_jump_model, stage_index  
    # the first is a new JuMP Model for the subproblem, the second is an index from  
    1,2,...,5  
  
    # ... subproblem definition goes here ...  
  
end
```

Defining the subproblem

We still need to define the last five things:

- 3. States
- 4. Controls
- 5. Noises
- 6. Dynamics
- 7. Objective

We're going to use both s_p and t from above.

3. Defining a state

A stage has an incoming, and an outgoing state variable. Behind the scenes we'll take care of matching them up between stages.

To define a new state variable use the `@state` macro.

```
@state(sp, lb <= outgoing <= ub, incoming == initial value)
```

First argument is the subproblem variable from the constructor, second argument is the outgoing variable (any feasible JuMP variable definition), third argument is the incoming variable (`symbol == initial value`).

From above, we have one state $x_t \in [0, 1]$, $x_0 = 0.5$

```
@state(sp, 0 <= x <= 1, x0 == 0.5)
```

The x_0 is the incoming variable in each stage. It will only be forced to 0.5 in the first stage. The syntax is just for convinence.

We could also create three state variables

$$x_t^i \in [0, \infty), \quad x_0^i = i, \quad i = \{1, 2, 3\} \quad t = \{1, 2, \dots, T\}$$

```
@state(sp, x[i=1:3] >= 0, x0==i)
```

4. Defining a control

Controls are just JuMP variables. Nothing special.

From above $u_t \in [0, 0.5]$

```
@variable(sp, 0 <= control <= 0.5)
```

5. Defining a Noise

A noise has three things:

1. A constraint
2. A set of RHS values
3. A probability distribution

Julia code is

```
@noise(sp, name = RHS Values, constraint)
```

```
setnoiseprobability!(sp, probability distribution)
```

From above we have

5 - Noises

- 10 stagewise independent noises: $\omega_t \in [0, 0.0333..., 0.0666..., \dots, 0.3]$

6 - Dynamics

- linear dynamics $x_t == x_{t-1} + u_t - \omega_t$

```
@noise(sp, omega = linspace(0, 0.3, 10), x == x0 + u - omega)
```

```
# set uniform probability (but its the default so you don't have to  
setnoiseprobability!(sp, fill(0.1, 10))
```


6. Defining dynamics

These can just be any JuMP constraints

```
@constraint(sp, x + u <= 1.5)
```

7. Defining the Stage Objective

We only care about defining the stage objective. The future costs get handled automatically.

```
stageobjective!(sp, AffExpr of Objective)
```

We can use the index t to change coefficients between subproblems so our objective is

```
stageobjective!(sp, (sin(3 * t) - 1) * u)
```


Compare the Julia code to the mathematical subproblem

$$\min_{u_t} \quad (\sin(3t) - 1)u_t + \theta_{t+1}$$

$$s. t. \quad x_t = x_{t-1} + u_t - \omega_t$$

$$x_t \in [0, 1]$$

$$u_t \in [0, 0.5]$$

$$x_0 = 0.5$$

```

m = SDDPModel(
    sense = :Min,
    stages = 5,
    solver = ClpSolver(),
    objective_bound = -2
) do sp, t
    @state(sp, 0 <= x <= 1, x0 == 0.5)
    @variable(sp, 0 <= u <= 0.5)
    @noise(sp, ω = linspace(0, 0.3, 10),
           x == x0 + u - ω)
    stageobjective!(sp, (sin(3t) - 1)*u )
end

```

Solve options

For a full list run `julia>? SDDP.solve`

```
status = solve(m,  
    max_iterations = 10,  
    time_limit     = 600,  
    simulation      = MonteCarloSimulation(  
        frequency = 5,  
        min       = 10,  
        step      = 10,  
        max       = 100,  
        terminate = false  
    )  
)
```

```

In [3]: srand(1111)
status = solve(m,
    max_iterations = 20,
    time_limit     = 600,
    simulation      = MonteCarloSimulation(
        frequency = 5,  # Number of forwards to construct the statistical bound
        min        = 10, # Min number of forwards to evaluate confidence interval for the bound
        step       = 10,
        max        = 100,
        confidence = 0.95,
        termination = false
    ),
    print_level=0
)

# MonteCarloSimulation(frequency,steps,confidence,termination)
# MonteCarloSimulation(frequency,collect(min:step:max),confidence,termination)

# Check bound is correct
println("Final bound is $(SDDP.getbound(m)) (Expected -1.471).")

```

WARNING: Solver does not appear to support providing initial feasible solutions.

Final bound is -1.471483864147188 (Expected -1.471).

SDDP Solver. © Oscar Dowson, 2017.

Solver:

Serial solver

Model:

Stages: 5

States: 1

Subproblems: 5

Value Function: Default

Objective				Cut	Passes	Simulations		Total
Expected	Bound	% Gap		#	Time	#	Time	Time
-1.591	-1.471			1	0.0	0	0.0	0.0
-1.365	-1.471			2	0.0	0	0.0	0.0
-1.518	-1.471			3	0.0	0	0.0	0.0
-1.624	-1.471			4	0.0	0	0.0	0.0
-1.569	-1.479	-1.471	-6.7	5	0.0	20	0.0	0.1
-1.537	-1.471			6	0.0	20	0.0	0.1

```
In [ ]: simulation = simulate(m, 1000, [:x, :u])
println("Mean of simulation objectives is $(mean(r[:objective] for r in simulation)
)")
```

```
In [ ]: @visualise(simulation, i, t, begin
           simulation[i][:x][t], (title="State")
           simulation[i][:u][t], (title="Control")
           simulation[i][:scenario][t], (title="Scenario")
           simulation[i][:stageobjective][t], (title="Objective", cumulative=true)
       end)
```

Open Visualisation (<https://odow.github.io/talks/assets/stock-example-visualisation.html>)

Example: Simplified Hydrothermal Dispatch

- Assume two thermoelectrics plants and one hydroelectric plant with reservoir and unit productivity coefficient.
- The first thermoelectric with cost 100 and the second with 1000 (R\$/ MWh) and capacities equal to 50 MW each.
- The hydroelectric plant has a reservoir with a capacity equivalent to 150 MWh that starts with a power of 150 MW.
- We want to minimize the cost of generating the next 3 hours.
- Demand is constant and equal to 150 MWh in all hours.

Notation

- $g_{i,t}$ - thermoelectric generation
- u_t - turbine
- v_t - reservoir volume
- a_t - affluence
- s_t - spillway

Subproblem

$$FCF(v_{t-1}) = \begin{array}{l} \min \limits_{g,s,u,s \geq 0} 100 g_{1,t} + 1000 g_{2,t} \text{ s.t.} \\ g_{1,t} + g_{2,t} + u_t = 150 \text{ \& } v_t + u_t + st = v_{t-1} + a_t \text{ \& } 0 \leq v_t \leq 200 \text{ \& } \\ 0 \leq u_t \leq 150 \text{ \& } 0 \leq g_{1,t} \leq 50 \text{ \& } 0 \leq g_{2,t} \leq 50 \end{array}$$

```
In [ ]: # load some packages
        using SDDP, JuMP, Clp

        m = SDDPModel(
            sense = ,
            stages = ,
            solver = ClpSolver(),
            objective_bound = 0
        ) do sp, t

            # State
            @state(sp,)

            # Variables
            @variable(sp,)

            # Noise
            @noise(sp, , )

            # Constraints
            @constraint(sp, )

            # Objective function
            stageobjective!(sp, )
        end
```



```
In [ ]: srand(1111)
status = solve(m,
    max_iterations = ,
    time_limit     = ,
    simulation      = MonteCarloSimulation(
        frequency = ,
        min        = ,
        step       = ,
        max        = ,
        termination = false
    ),
    print_level=0
)
```

Average Value at Risk

`risk_measure = NestedAVaR(lambda = 0.5, beta = 0.5)`

A risk measure that is a convex combination of Expectation and Average Value @ Risk (also called Conditional Value @ Risk).

$$\text{lambda} * E[x] + (1 - \text{lambda}) * \text{AV@R}(1-\text{beta})[x]$$

Keyword Arguments

- `lambda` - Convex weight on the expectation ((1 - `lambda`) weight is put on the AV@R component. Increasing values of `lambda` are less risk averse (more weight on expectation)
- `beta` - The quantile at which to calculate the Average Value @ Risk. Increasing values of `beta` are less risk averse. If `beta=0`, then the AV@R component is the worst case risk measure.

```

In [ ]: m_risk = SDDPModel(
            sense = :Min,
            stages = 5,
            solver = ClpSolver(),
            # risk_measure = Expectation(),
            risk_measure = NestedAVaR(lambda=0.5, beta=0.5),
            objective_bound = -2
        ) do sp, t

    # the state
    @state(sp, 0 <= x <= 1, x0 == 0.5)

    # the control
    @variable(sp, 0 <= u <= 0.5)

    # the noise (and dynamics)
    @noise(sp, omega = linspace(0, 0.3, 10), x == x0 + u - omega)

    # the objective
    stageobjective!(sp, (sin(3 * t) - 1) * u)

end
println(typeof(m_risk))

```

```
In [ ]: srand(1111)
status = solve(m_risk,
    max_iterations = 20,
    time_limit     = 600,
    simulation      = MonteCarloSimulation(
        frequency = 5,
        min        = 10,
        step       = 10,
        max        = 100,
        termination = false
    ),
    print_level=0
)

# Check bound is correct
println("Final bound is $(SDDP.getbound(m_risk)) (Expectation bound was -1.471).")
```

De Matos (Level One) Cut Selection

```
m_risk = SDDPModel(  
    sense = :Min,  
    stages = 5,  
    solver = ClpSolver(),  
    risk_measure = Expectation(),  
    objective_bound = -2,  
    cut_oracle = DematosCutOracle()  
    ) do sp, t
```

Asynchronous Solver

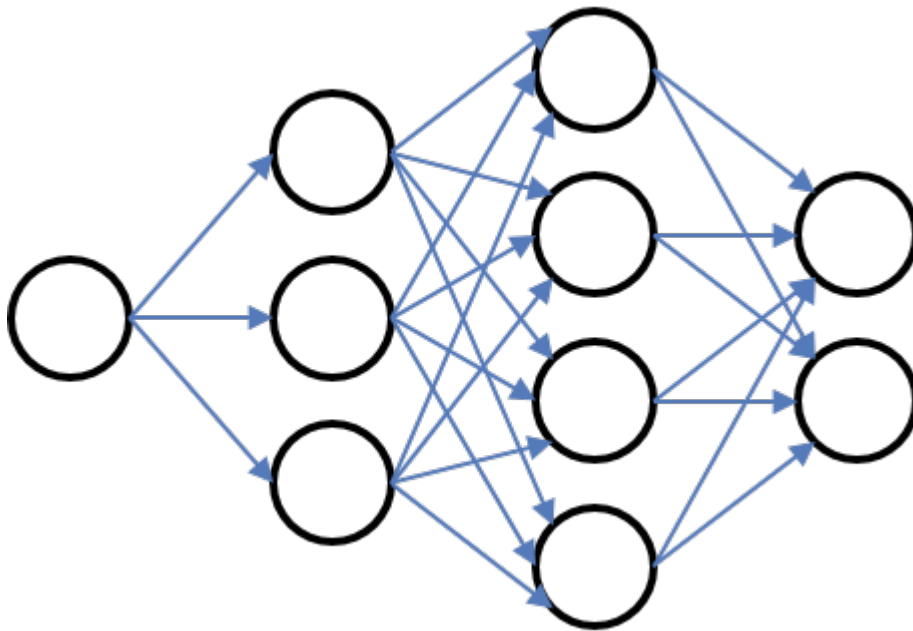
We parallelise by farming out a new instance of the SDDPModel to all slave processors.

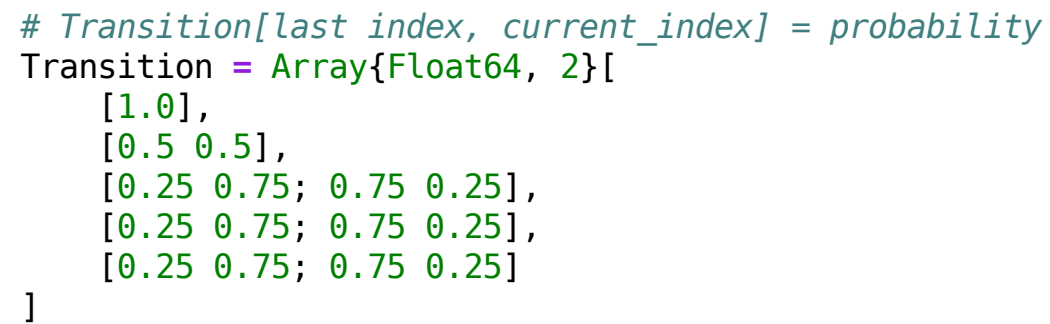
Slaves perform iterations independently, and asynchronously share cuts between themselves.

```
solve(m,  
      solve_type = Serial()  
      # or  
      solve_type = Asynchronous()  
)
```

Markov Uncertainty

More like a feed-forward graph with discrete stages but arbitrary number of nodes and transitions





```
# Transition[last_index, current_index] = probability
Transition = Array{Float64, 2}[
    [1.0],
    [0.5 0.5],
    [0.25 0.75; 0.75 0.25],
    [0.25 0.75; 0.75 0.25],
    [0.25 0.75; 0.75 0.25]
]
```



```

In [4]: Transition = Array{Float64, 2}[
    [1.0]',
    [0.5 0.5],
    [0.25 0.75; 0.75 0.25],
    [0.25 0.75; 0.75 0.25],
    [0.25 0.75; 0.75 0.25]
]

m_markov = SDDPModel(
    sense = :Min,
    stages = 5,
    solver = ClpSolver(),
    objective_bound = -10,
    # A vector of transition matrices. One for each stage
    markov_transition = Transition
    # markov_state will go from 1, 2, ..., S
    ) do sp, t, markov_state

    @state(sp, 0 <= x <= 1, x0 == 0.5)
    @variable(sp, 0 <= u <= 0.5)
    @noise(sp, omega = linspace(0, 0.3, 10), x == x0 + u - omega)

    # the objective
    stageobjective!(sp, (sin(3 * t) - 0.75 * markov_state) * u)

end
println(typeof(m_markov))

```

```
SDDP.SDDPModel{SDDP.DefaultValueFunction{SDDP.DefaultCutOracle}}
```

```
In [5]: status = solve(m_markov,  
    max_iterations = 10,  
    print_level=0  
    )  
  
    # Check bound is correct  
    println("Final bound is $(SDDP.getbound(m_markov)).")
```

Final bound is -1.634417972667261.

More information

Examples, parameters and code github.com/odow/SDDP.jl
(<https://github.com/odow/SDDP.jl>)

Original Oscar Dowson presentation github.com/odow/talks/blob/master/sddp_jl.ipynb
(https://github.com/odow/talks/blob/master/sddp_jl.ipynb)

This talk thuener.github.io/talks/ (<https://thuener.github.io/talks/>)