The life and times of SDDP.jl

https://sddp.dev

Oscar Dowson

JuMP-dev 2025

The purpose of this talk

Why is this talk needed:

- SDDP.jl is a somewhat popular (if niche) JuMP extension
- It has been in continuous development for 10 years
- It has a lot of ideas that might be useful for other extensions

I've never given a talk about it at JuMP-dev (or on YouTube)

By the end of this talk you will:

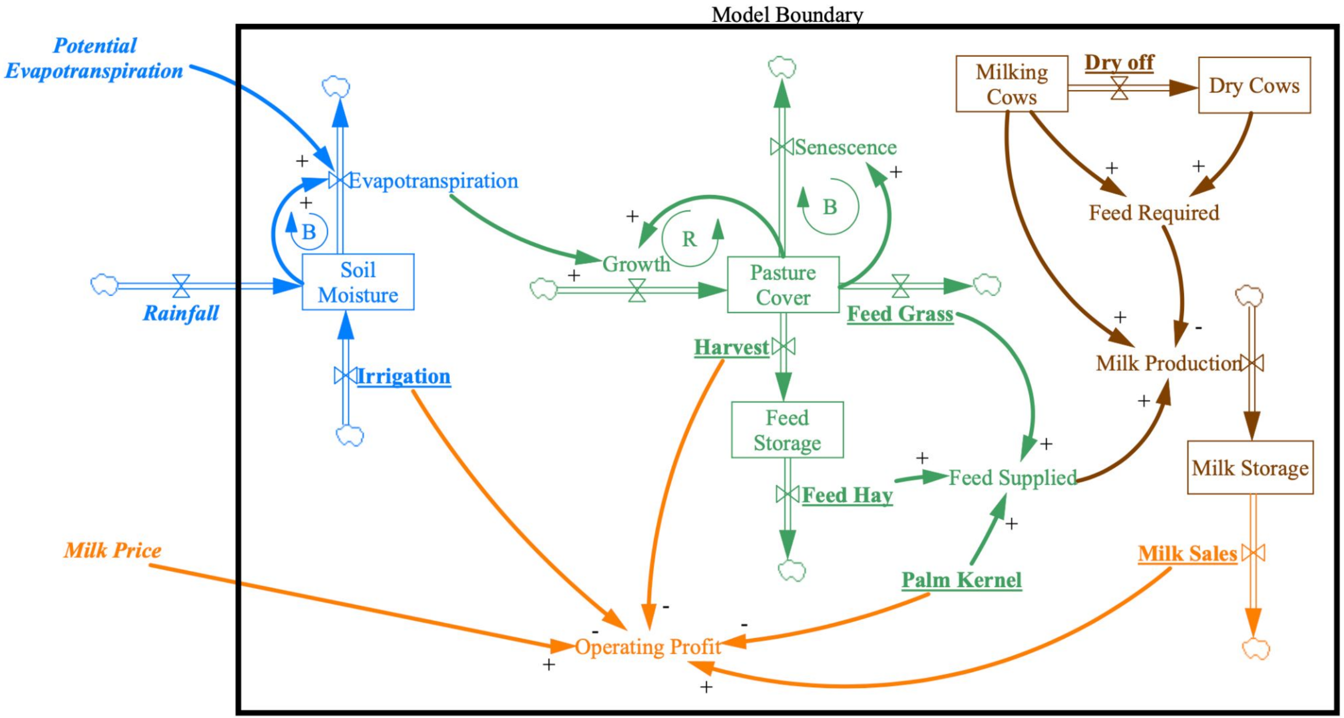
1. Have some knowledge of the New Zealand dairy and electricity industries
2. Understand the policy graph decomposition for modeling sequential decision problems
3. Know how SDDP.jl implements a JuMP extension, uses multiple dispatch, and supports multithreading

By the end of this talk you will not:

- Know the details of the SDDP algorithm

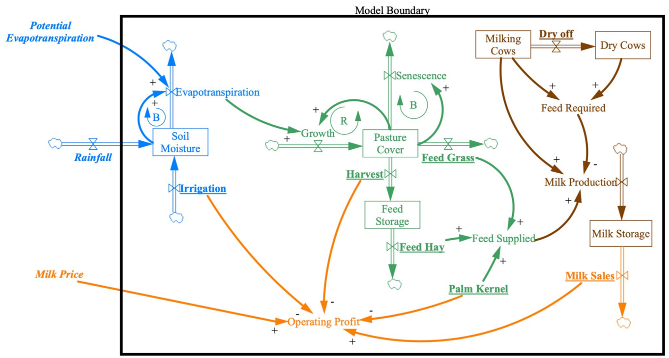
The New Zealand dairy industry

It's a stochastic optimal control problem



The New Zealand dairy industry

It’s a stochastic optimal control problem



ω_i

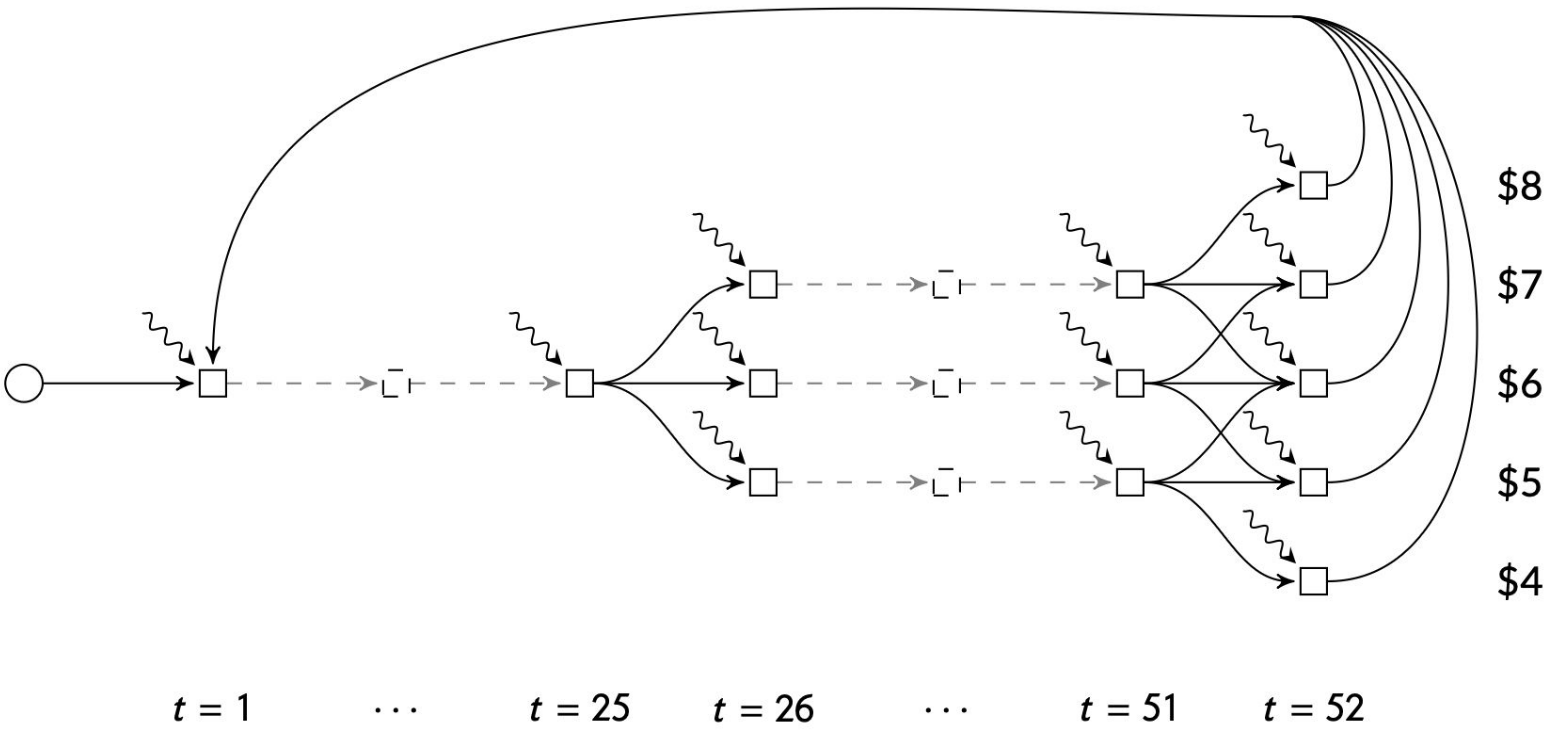
x

$$\begin{aligned} \min_{\bar{x},u,x'} \quad & C_i(\bar{x}, u, x', \omega_i) \\ \text{s.t.} \quad & (u, x') \in \mathcal{X}_i(\bar{x}, \omega_i) \\ & \bar{x} = x \end{aligned}$$

x'

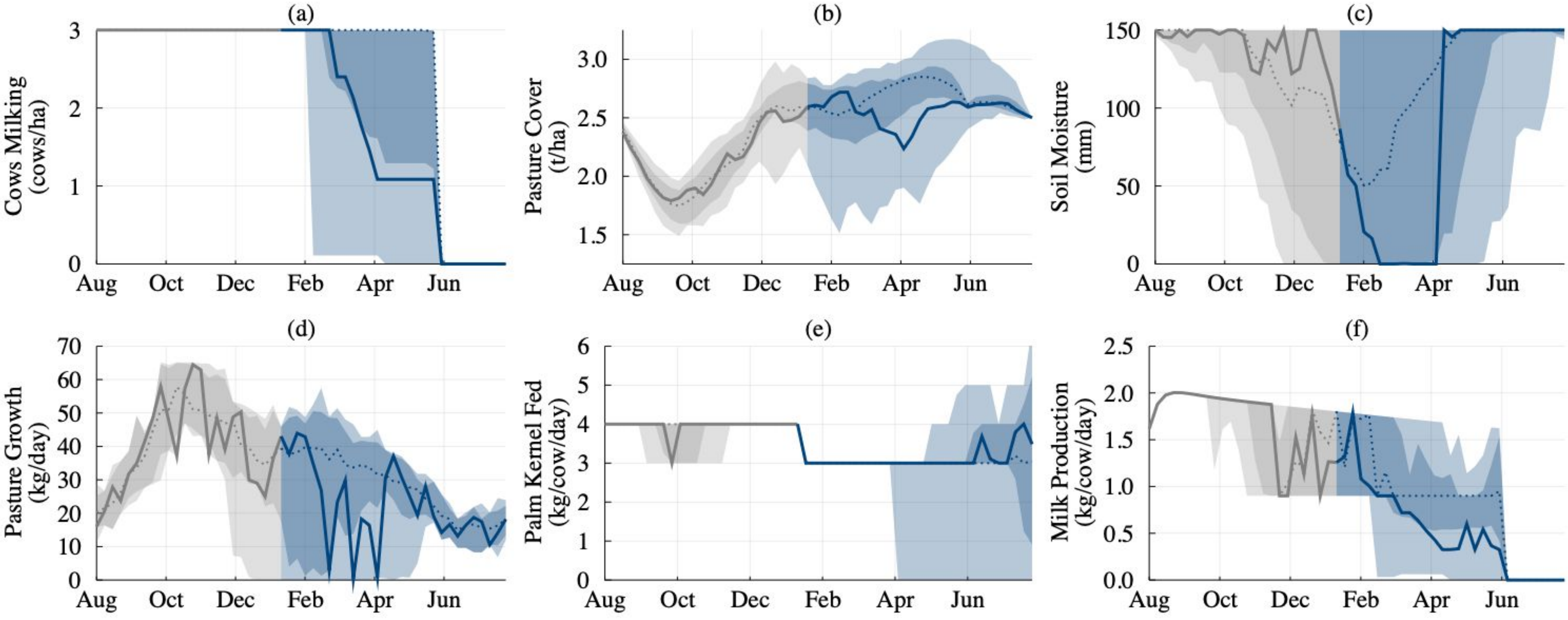
The New Zealand dairy industry

It’s a stochastic optimal control problem



The New Zealand dairy industry

It's a stochastic optimal control problem



The New Zealand energy system

It’s a stochastic optimal control problem

State variables

- Volume of water in each reservoir

Control variables

- Hydro generation
- Thermal generation

Random variables

- Inflow

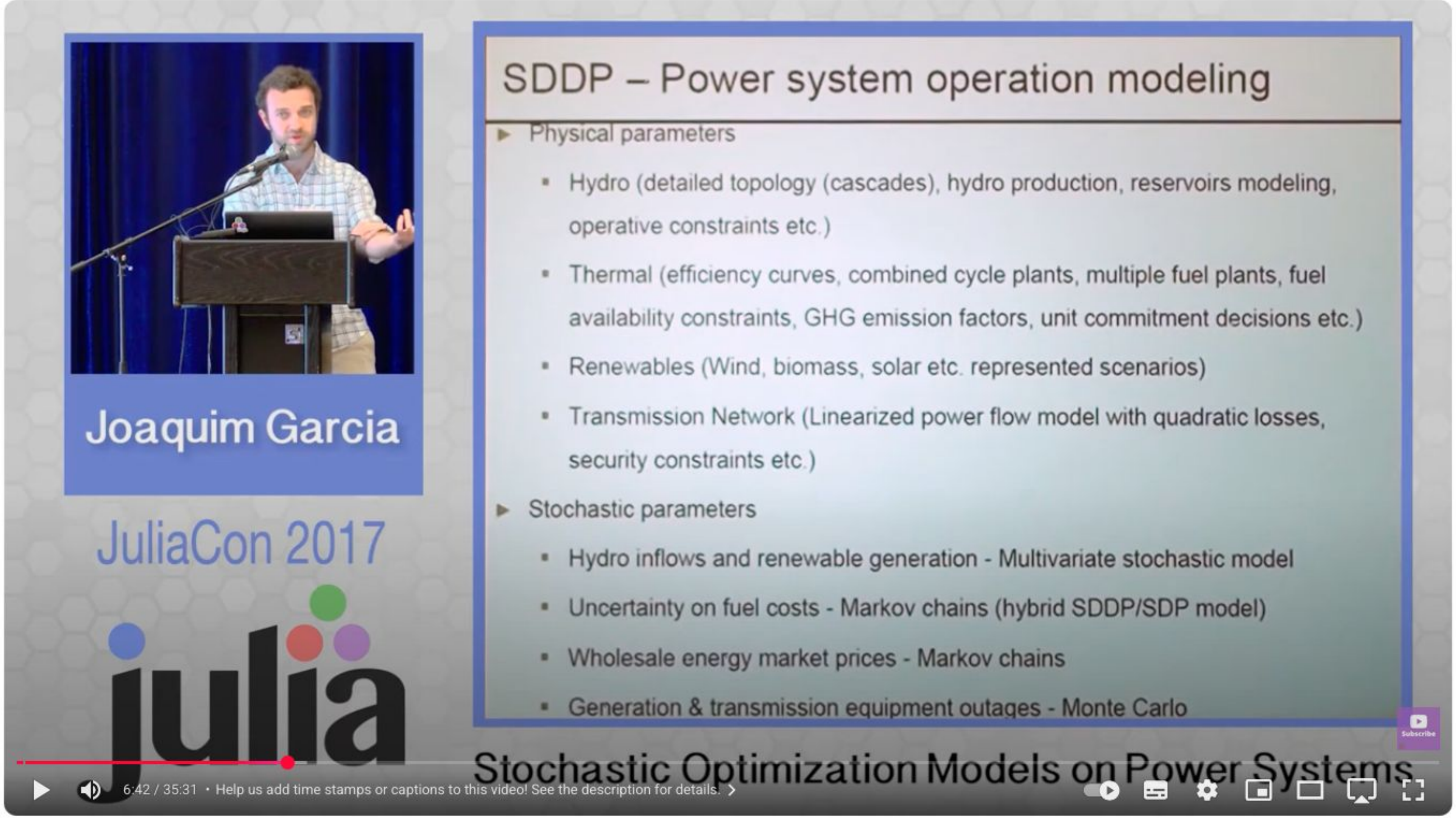
Constraints

- water in = water out
- supply = demand

Objective function

- Minimize cost

Cows, Lakes, and a JuMP extension... | Oscar Dowson | JuliaCon 2017



Stochastic Optimization Models on Power Systems | Camila Metello and Joaquim Garcia | JuliaCon 2017

JADE: Just Another DOASA Environment

Contributions from the Electric Power Optimization Centre

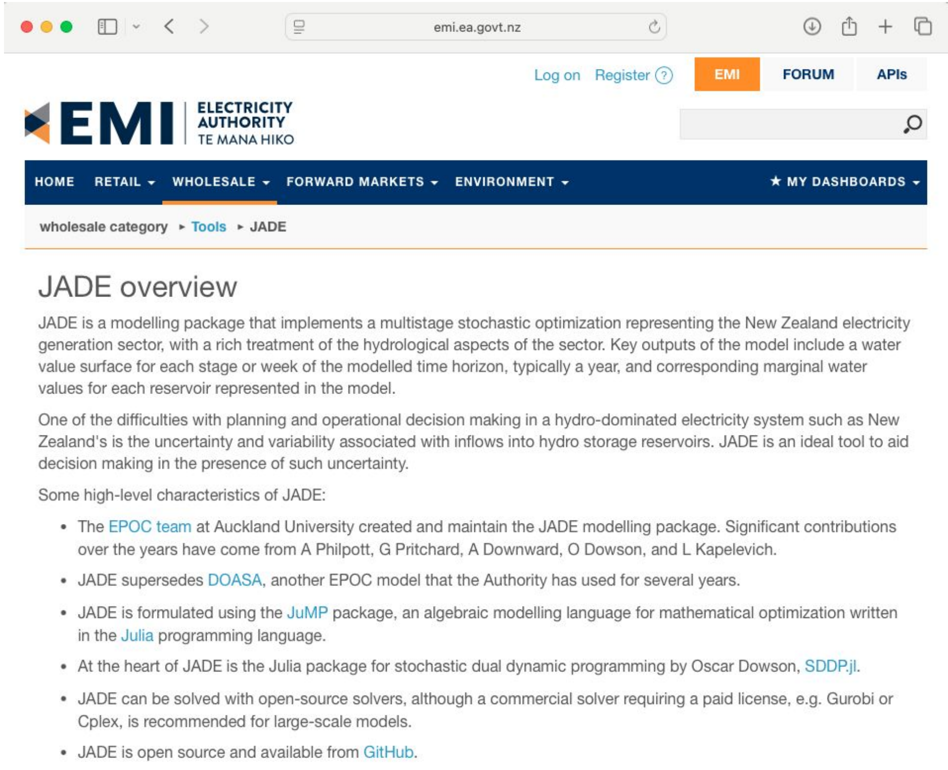
1991: SDDP introduced by Pereira and Pinto

2008: Philpott & Guan wrote the first AMPL version of NZ model called DOASA

2012-16: Philpott and de Matos wrote a C++ version of DOASA

2016-18: I wrote SDDP.jl (for cows), Lea Kapelevich wrote a NZ energy model called JADE

2022: JADE was adopted by New Zealand electricity authority



emi.ea.govt.nz

Log on Register ? EMI FORUM APIs

EMI ELECTRICITY AUTHORITY TE MANA HIKO

HOME RETAIL ▾ WHOLESALE ▾ FORWARD MARKETS ▾ ENVIRONMENT ▾ ★ MY DASHBOARDS ▾

wholesale category ▸ Tools ▸ JADE

JADE overview

JADE is a modelling package that implements a multistage stochastic optimization representing the New Zealand electricity generation sector, with a rich treatment of the hydrological aspects of the sector. Key outputs of the model include a water value surface for each stage or week of the modelled time horizon, typically a year, and corresponding marginal water values for each reservoir represented in the model.

One of the difficulties with planning and operational decision making in a hydro-dominated electricity system such as New Zealand's is the uncertainty and variability associated with inflows into hydro storage reservoirs. JADE is an ideal tool to aid decision making in the presence of such uncertainty.

Some high-level characteristics of JADE:

- The EPOC team at Auckland University created and maintain the JADE modelling package. Significant contributions over the years have come from A Philpott, G Pritchard, A Downward, O Dowson, and L Kapelevich.
- JADE supersedes DOASA, another EPOC model that the Authority has used for several years.
- JADE is formulated using the JuMP package, an algebraic modelling language for mathematical optimization written in the Julia programming language.
- At the heart of JADE is the Julia package for stochastic dual dynamic programming by Oscar Dowson, SDDP.jl.
- JADE can be solved with open-source solvers, although a commercial solver requiring a paid license, e.g. Gurobi or Cplex, is recommended for large-scale models.
- JADE is open source and available from GitHub.

Modelling

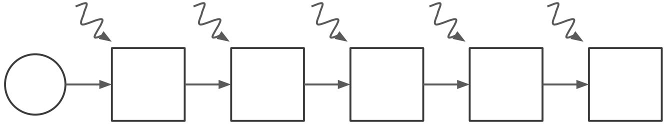
The ingredients

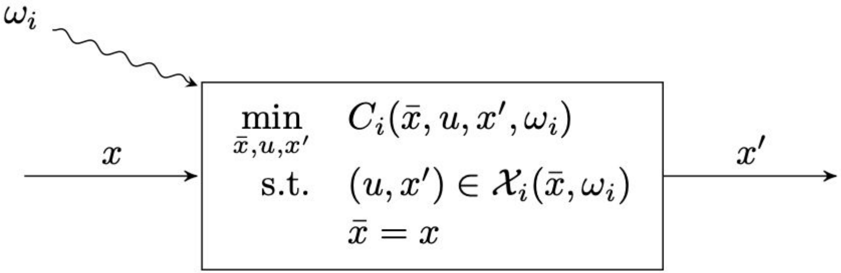
To model a policy graph we need

- A description of the graph
- A subproblem for each node

Each subproblem needs

1. Incoming and outgoing state variables
2. Control variables
3. Constraints
4. An objective function
5. A random variable





Modelling

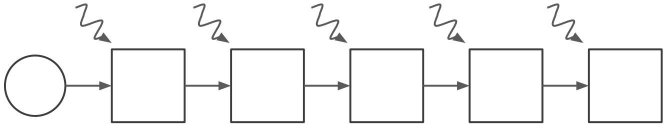
An example

```
model = SDDP.LinearPolicyGraph(;
    stages = 5, lower_bound = 0,
) do sp::JuMP.Model, t::Int
    @variable(sp, 0 <= x_reservoir <= 10, SDDP.State, initial_value = 5)
    @variable(sp, u_thermal >= 0)
    @variable(sp, u_hydro >= 0)
    @constraint(sp, c_water, x_reservoir.out - x_reservoir.in + u_hydro <= 0)
    @constraint(sp, c_demand, u_thermal + u_hydro == 1)
    SDDP.@stageobjective(sp, t * u_thermal)
    Ω, P = [0, 1, 2], [0.3, 0.5, 0.2]
    SDDP.parameterize(sp, Ω, P) do ω
        set_normalized_rhs(c_water, ω)
    end
end
SDDP.train(model)
simulations = SDDP.simulate(model, 100)
```

Modelling

A description of the graph

```julia
model = SDDP.LinearPolicyGraph(;
    stages = 5, lower_bound = 0,
) do sp::JuMP.Model, t::Int
    @variable(sp, 0 <= x_reservoir <= 10, SDDP.State, initial_value = 5)
    @variable(sp, u_thermal >= 0)
    @variable(sp, u_hydro >= 0)
    @constraint(sp, c_water, x_reservoir.out - x_reservoir.in + u_hydro <= 0)
    @constraint(sp, c_demand, u_thermal + u_hydro == 1)
    SDDP.@stageobjective(sp, t * u_thermal)
    Ω, P = [0, 1, 2], [0.3, 0.5, 0.2]
    SDDP.parameterize(sp, Ω, P) do ω
        set_normalized_rhs(c_water, ω)
    end
end
SDDP.train(model)
simulations = SDDP.simulate(model, 100)
```



Modelling

State variables

```
model = SDDP.LinearPolicyGraph(;
    stages = 5, lower_bound = 0,
) do sp::JuMP.Model, t::Int
    @variable(sp, 0 <= x_reservoir <= 10, SDDP.State, initial_value = 5)
    @variable(sp, u_thermal >= 0)
    @variable(sp, u_hydro >= 0)
    @constraint(sp, c_water, x_reservoir.out - x_reservoir.in + u_hydro <= 0)
    @constraint(sp, c_demand, u_thermal + u_hydro == 1)
    SDDP.@stageobjective(sp, t * u_thermal)
    Ω, P = [0, 1, 2], [0.3, 0.5, 0.2]
    SDDP.parameterize(sp, Ω, P) do ω
        set_normalized_rhs(c_water, ω)
    end
end
SDDP.train(model)
simulations = SDDP.simulate(model, 100)
```

Modelling

Control variables

```
model = SDDP.LinearPolicyGraph(;
    stages = 5, lower_bound = 0,
) do sp::JuMP.Model, t::Int
    @variable(sp, 0 <= x_reservoir <= 10, SDDP.State, initial_value = 5)
    @variable(sp, u_thermal >= 0)
    @variable(sp, u_hydro >= 0)
    @constraint(sp, c_water, x_reservoir.out - x_reservoir.in + u_hydro <= 0)
    @constraint(sp, c_demand, u_thermal + u_hydro == 1)
    SDDP.@stageobjective(sp, t * u_thermal)
    Ω, P = [0, 1, 2], [0.3, 0.5, 0.2]
    SDDP.parameterize(sp, Ω, P) do ω
        set_normalized_rhs(c_water, ω)
    end
end
SDDP.train(model)
simulations = SDDP.simulate(model, 100)
```

Modelling

Constraints

```
model = SDDP.LinearPolicyGraph(;
    stages = 5, lower_bound = 0,
) do sp::JuMP.Model, t::Int
    @variable(sp, 0 <= x_reservoir <= 10, SDDP.State, initial_value = 5)
    @variable(sp, u_thermal >= 0)
    @variable(sp, u_hydro >= 0)
    @constraint(sp, c_water, x_reservoir.out - x_reservoir.in + u_hydro <= 0)
    @constraint(sp, c_demand, u_thermal + u_hydro == 1)
    SDDP.@stageobjective(sp, t * u_thermal)
    Ω, P = [0, 1, 2], [0.3, 0.5, 0.2]
    SDDP.parameterize(sp, Ω, P) do ω
        set_normalized_rhs(c_water, ω)
    end
end
SDDP.train(model)
simulations = SDDP.simulate(model, 100)
```

Modelling

Objective function

```
model = SDDP.LinearPolicyGraph(;
    stages = 5, lower_bound = 0,
) do sp::JuMP.Model, t::Int
    @variable(sp, 0 <= x_reservoir <= 10, SDDP.State, initial_value = 5)
    @variable(sp, u_thermal >= 0)
    @variable(sp, u_hydro >= 0)
    @constraint(sp, c_water, x_reservoir.out - x_reservoir.in + u_hydro <= 0)
    @constraint(sp, c_demand, u_thermal + u_hydro == 1)
    SDDP.@stageobjective(sp, t * u_thermal)
    Ω, P = [0, 1, 2], [0.3, 0.5, 0.2]
    SDDP.parameterize(sp, Ω, P) do ω
        set_normalized_rhs(c_water, ω)
    end
end
SDDP.train(model)
simulations = SDDP.simulate(model, 100)
```

Modelling

Random variables

```julia
model = SDDP.LinearPolicyGraph(;
    stages = 5, lower_bound = 0,
) do sp::JuMP.Model, t::Int
    @variable(sp, 0 <= x_reservoir <= 10, SDDP.State, initial_value = 5)
    @variable(sp, u_thermal >= 0)
    @variable(sp, u_hydro >= 0)
    @constraint(sp, c_water, x_reservoir.out - x_reservoir.in + u_hydro <= 0)
    @constraint(sp, c_demand, u_thermal + u_hydro == 1)
    SDDP.@stageobjective(sp, t * u_thermal)
    Ω, P = [0, 1, 2], [0.3, 0.5, 0.2]
    SDDP.parameterize(sp, Ω, P) do ω
        set_normalized_rhs(c_water, ω)
    end
end
SDDP.train(model)
simulations = SDDP.simulate(model, 100)
```

Modelling

All of it together

```
model = SDDP.LinearPolicyGraph(;
    stages = 5, lower_bound = 0,
) do sp::JuMP.Model, t::Int
    @variable(sp, 0 <= x_reservoir <= 10, SDDP.State, initial_value = 5)
    @variable(sp, u_thermal >= 0)
    @variable(sp, u_hydro >= 0)
    @constraint(sp, c_water, x_reservoir.out - x_reservoir.in + u_hydro <= 0)
    @constraint(sp, c_demand, u_thermal + u_hydro == 1)
    SDDP.@stageobjective(sp, t * u_thermal)
    Ω, P = [0, 1, 2], [0.3, 0.5, 0.2]
    SDDP.parameterize(sp, Ω, P) do ω
        set_normalized_rhs(c_water, ω)
    end
end
SDDP.train(model)
simulations = SDDP.simulate(model, 100)
```

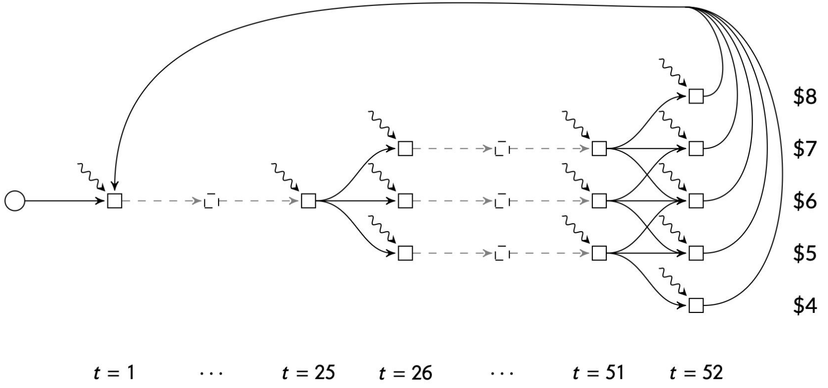
Solution algorithm

SDDP a.k.a each node is Benders

```
julia> SDDP.train(model; time_limit = 240)
-------------------------------------------------------------------
        SDDP.jl (c) Oscar Dowson and contributors, 2017-25
-------------------------------------------------------------------
problem
 nodes           : 108
 state variables : 5
 scenarios       : Inf
options
 solver          : SDDP.Threaded()
 risk measure    : SDDP.Expectation()
-------------------------------------------------------------------
 iteration    simulation      bound        time (s)     solves  pid
-------------------------------------------------------------------
         1   -1.415148e+04  1.000000e+06  3.980785e+00     33013   3
         2   -1.893675e+04  1.000000e+06  4.179685e+00     40286   1
         3   -1.033561e+04  1.000000e+06  4.842130e+00     66687   1
†        4   -2.486776e+04  7.881972e+05  7.643921e+00    175091   1
†        5   -2.831493e+05  7.588107e+05  1.909121e+01    514620   4
         6   -3.021992e+05  7.588107e+05  2.195494e+01    554152   2
         7   -1.556580e+05  7.308679e+05  2.195530e+01    554152   3
         8   -1.025802e+05  7.308679e+05  2.195624e+01    554155   1

†       21   -6.585407e+04  2.692613e+05  6.075782e+01   1568680   3

        83    3.115513e+05  1.272015e+05  2.112643e+02   4539440
```



We modify and solve 4.5e6 LPs in 200 seconds.

Multithreading

Non-deterministic concurrent

Each thread iterates independently

There is a lock at each node

Works great if # nodes >> # threads

Problems

Gurobi environments are not thread-safe. Need a separate license for each node (not each thread)

Limited tooling to detect race conditions

Trivial to implement

```
# Serial
while iteration(model, options)
end

# Parallel
interrupt = Threads.Atomic{Bool}(false)
@sync for _ in 1:Threads.nthreads()
    Threads.@spawn try
        while !interrupt[] && iteration(model, options)
        end
    finally
        interrupt[] = true
    end
end
```

Multiple dispatch

We use it. Perhaps too much

There are many “plug-ins” in SDDP.jl

Type	*Controls…*
`AbstractRiskMeasure`	How random variables are aggregated into a scalar
`AbstractDualityHandler`	How we compute the reduced of a fixed variable
`AbstractSamplingScheme`	How we sample trajectories in the graph
`AbstractForwardPass`	The forward pass
`AbstractBackwardPass`	The backward pass

Multiple dispatch

SDDP.AbstractRiskMeasure

```julia
struct Expectation <: SDDP.AbstractRiskMeasure
end

function SDDP.adjust_probability(
    measure::Expectation, q, p, ω, X, is_min,
)
    q .= p
    return 0.0
end

SDDP.train(
    model;
    risk_measure = SDDP.Expectation(),
)
```

```julia
struct Entropic <: SDDP.AbstractRiskMeasure
    γ::Float64
end

function SDDP.adjust_probability(
    measure::Entropic, q, p, ω, X, is_min,
)
    γ = is_min ? measure.γ : -measure.γ
    y = p .* exp.(big.(γ .* X))
    q .= y / sum(y)
    return -q' * log.(q ./ p) / γ
end

SDDP.train(
    model;
    risk_measure = SDDP.Entropic(10.0),
)
```

Multiple dispatch

SDDP.AbstractDualityHandler

```julia
struct ContinuousConicDuality <:
       SDDP.AbstractDualityHandler
end
function SDDP.get_dual_solution(
    node::SDDP.Node, ::ContinuousConicDuality,
)
    undo =
        relax_integrality(node.subproblem)
    JuMP.optimize!(node.subproblem)
    ret = Dict(
        name => JuMP.dual(JuMP.FixRef(x.in))
        for (name, x) in node.states
    )
    undo()
    return ret
end

duality_handler = SDDP.ContinuousConicDuality()
SDDP.train(model; duality_handler)
```

```julia
struct FixedDiscreteDuality <:
       SDDP.AbstractDualityHandler
end
function SDDP.get_dual_solution(
    node::SDDP.Node, ::FixedDiscreteDuality,
)
    undo =
        fix_discrete_variables(node.subproblem)
    JuMP.optimize!(node.subproblem)
    ret = Dict(
        name => JuMP.dual(JuMP.FixRef(x.in))
        for (name, x) in node.states
    )
    undo()
    return ret
end

duality_handler = SDDP.FixedDisreteDuality()
SDDP.train(model; duality_handler)
```

Takeaways

If you remember nothing else, go to https://sddp.dev

For modelers

- Sequential decision making under uncertainty is pervasive
- One approach is to model them as a policy graph
- We have general purpose software for solving policy graphs

For JuMP developers

- JuMP extensions allow custom syntax for users
- Writing multithreaded algorithms is "easy"
- Multiple dispatch makes it trivial to provide plugins that change the algorithm

restricted modeling + high quality general purpose software

>>>

generic modeling that requires custom algorithms