StochDynamicProgramming Documentation

Release 0.1

leclerev

May 04, 2016

Contents

This package implements the Stochastic Dual Dynamic Programming (SDDP) algorithm with Julia. It relies upon MathProgBase and JuMP.

A complete overview of this algorithm could be found here.

At the moment the plan is to create a type such that :

- you can fix linear time t cost function (then convex piecewise linear)
- you can fix linear dynamic (with a physical state x and a control u)
- the scenarios are created by assuming that the noise ξ_t is independent in time, each given by a table (value, probability)

Then it is standard SDDP :

- fixed number of forward passes
- iterative backward passes
- no clearing of cuts
- stopping after a given number of iterations / computing time

Once solved the SDDP model should be able to :

- give the lower bound on the cost
- simulate trajectories to evaluate expected cost
- give the optimal control given current time state and noise

We have a lot of ideas to implement further :

- spatial construction (i.e : defining stock one by one and then their interactions)
- noise as AR (eventually with fitting on historic datas)
- convex solvers
- refined stopping rules
- cut pruning
- paralellization

CHAPTER 1

Contents:

1.1 Installation guide

1.1.1 StochDynamicProgramming installation

To install StochDynamicProgramming:

```
julia> Pkg.clone("https://github.com/leclere/StochDynamicProgramming.jl")
```

Once the package is installed, you can import it in the REPL:

```
julia> using StochDynamicProgramming
```

1.1.2 Install a linear programming solver

SDDP need a linear programming solver to run. Clp is installed by default with StochDynamicProgramming.jl.

Refer to the documentation of JuMP to get another solver and interface it with SDDP.

The following solvers have been tested:

Solver	Is working?
Clp	Yes
CPLEX	Yes
Gurobi	Yes
GLPK	**No**

1.1.3 Run Unit-Tests

To run unit-tests:

```
$ julia test/runtests.jl
```

1.2 Step-by-step example

This page gives a short introduction to the interface of this package. It explains the resolution with SDDP of a classical example: the management of a dam over one year with random inflow.

1.2.1 Use case

In the following, x_t will denote the state and u_t the control at time t. We will consider a dam, whose dynamic is:

$$x_{t+1} = x_t - u_t + w_t$$

At time t, we have a random inflow w_t and we choose to turbine a quantity u_t of water.

The turbined water is used to produce electricity, which is being sold at a price c_t. At time t we gain:

$$C(x_t, u_t, w_t) = c_t \times u_t$$

We want to minimize the following criterion:

$$J = \min_{x,u} \sum_{t=0}^{T-1} C(x_t, u_t, w_t)$$

We will assume that both states and controls are bounded:

$$x_t \in [0, 100], \qquad u_t \in [0, 7]$$

1.2.2 Problem definition in Julia

We will consider 52 time steps as we want to find optimal value functions for one year:

```
N_STAGES = 52
```

and we consider the following initial position:

```
X0 = [50]
```

Note that X0 is a vector.

Dynamic

We write the dynamic (which return a vector):

```
function dynamic(t, x, u, xi)
    return [x[1] + u[1] - xi[1]]
end
```

Cost

we store evolution of costs c_t in an array *COSTS*, and we define the cost function (which return a float):

```
function cost_t(t, x, u, w)
    return COSTS[t] * u[1]
end
```

Noises

Noises are defined in an array of Noiselaw. This type defines a discrete probability.

For instance, if we want to define a uniform probability with size $N = 10$, such that:

$$\mathbb{P}(X_i = i) = \frac{1}{N} \qquad \forall i \in 1..N$$

we write:

```
N = 10
proba = 1/N*ones(N) # uniform probabilities
xi_support = collect(linspace(1,N,N))
xi_law = NoiseLaw(xi_support, proba)
```

Thus, we could define a different probability laws for each time t. Here, we suppose that the probability is constant over time, so we could build the following vector:

```
xi_laws = NoiseLaw[xi_law for t in 1:N_STAGES-1]
```

Bounds

We add bounds over the state and the control:

```
s_bounds = [(0, 100)]
u_bounds = [(0, 7)]
```

Problem definition

As our problem is purely linear, we instantiate:

```
spmodel = LinearDynamicLinearCostSPmodel(N_STAGES,u_bounds,X0,cost_t,dynamic,xi_laws)
```

Solver

We define a SDDP solver for our problem.

First, we need to use a LP solver:

```
using Clp
SOLVER = ClpSolver()
```

Clp is automatically installed during package installation. To install different solvers on your machine, refer to the JuMP documentation.

Once the solver installed, we define SDDP algorithm parameters:

```
forwardpassnumber = 2 # number of forward pass
sensibility = 0. # admissible gap between upper and lower bound
max_iter = 20  # maximum number of iterations

paramSDDP = SDDPparameters(SOLVER, forwardpassnumber, sensibility, max_iter)
```

Now, we solve the problem by computing Bellman values:

```
V, pbs = solve_SDDP(spmodel, paramSDDP, 10) # display information every 10 iterations
```

`V` is an array storing the value functions, and `pbs` a vector of JuMP.Model storing each value functions as a linear problem.

We have an exact lower bound given by `V` with the function:

```
lb_sddp = StochDynamicProgramming.get_lower_bound(spmodel, paramSDDP, V)
```

1.2.3 Find optimal controls

Once Bellman functions are computed, we can control our system over assessments scenarios, without assuming knowledge of the future.

We build 1000 scenarios according to the laws stored in `xi_laws`:

```
scenarios = StochDynamicProgramming.simulate_scenarios(xi_laws,1000)
```

We compute 1000 simulations of the system over these scenarios:

```
costsddp, stocks = forward_simulations(spmodel, paramSDDP, V, pbs, scenarios)
```

`costsddp` returns the costs for each scenario, and `stocks` the evolution of each stock along time, for each scenario.

1.3 Advanced functions

This page gives an overview of the functions implemented in the package.

In the following, `model` will design a `SPModel` storing the definition of a stochastic problem, and `param` a SDDP-parameters instance which stores the parameters specified for SDDP. See quickstart for more information about these two objects.

1.3.1 Work with PolyhedralFunction

Get Bellman value at a given point

To estimate the Bellman value at a given position `xt` with a `PolyhedralFunction` `Vt`

```
vx = get_bellman_value(model, param, t, Vt, xt)
```

This is a lower bound of the true value.

Get optimal control at a given point

To get optimal control at a given point `xt` and for a given alea `xi`:

```
get_control(model, param, lpproblem, t, xt, xi)
```

where `lpproblem` is the linear problem storing the evaluation of Bellman function at time t.

1.3.2 Save and load pre-computed cuts

Assume that we have computed Bellman functions with SDDP. These functions are stored in a vector of `PolyhedralFunctions` denoted `V`

These functions can be stored in a text file with the command:

```
StochDynamicProgramming.dump_polyhedral_functions("yourfile.dat", V)
```

And then be loaded with the function:

```
Vdump = StochDynamicProgramming.read_polyhedral_functions("yourfile.dat")
```

1.3.3 Build LP Models with PolyhedralFunctions

We can build a vector of `JuMP.Model` with a vector of `PolyhedralFunction` to perform simulation. For this, use the function:

```
problems = StochDynamicProgramming.hotstart_SDDP(model, param, V)
```

1.3.4 SDDP hotstart

If cuts are already available, we can hotstart SDDP while overloading the function `solve_SDDP`:

```
V, pbs = solve_SDDP(model, params, 0, V)
```

1.3.5 Cuts pruning

The more SDDP run, the more cuts you need to store. It is sometimes useful to delete cuts which are useless for the computation of the approximated Bellman functions.

To clean a single `PolyhedralFunction` `Vt`:

```
Vt = exact_prune_cuts(model, params, Vt)
```

To clean a vector of `PolyhedralFunction` `V`:

```
prune_cuts!(model, params, V)
```