Developments on SDDP.jl package to produce infinite-horizon SDDP functionality

The developments are listed in the general order that they are used when formulating and solving a multi-stage stochastic program with SDDP.jl using infinite-horizon methods.

# type\_definitions

[1] The additional property of the state the cut was sampled at is added to the *Cut* struct. This property has the type Vector{Float64}.

[2] The *SubproblemExt*struct has the additional property of a flag *is\_infinite*for indicating if the model is using infinite-horizon SDDP. This flag has the type Bool.

[3] The function *Subproblem*has the new function input *is\_infinite*defaulting to*false*

# SDDPModel

[1] Boolean flag *is\_infinite* is now an optional function input to SDDPModel to tell method to build the SDDPModel object with infinite-horizon functionality

[2] If the *is\_infinite*flagis true, the following entries are added to the SDDP model objects extension dictionary

* *is\_infinite* (boolean): true if the *is\_infinite* flag of *SDDPModel* is true
* *completed\_iter1* (boolean): true is iteration 1 of SDDP is completed in a given “inner loop”
* *fp\_start\_state* (vector): the starting state in stage 1 before the forward pass is applied
* *fp\_end\_state* (vector): the end state in stage T after forward pass has been applied
* *lb\_state* (vector): lower bound of the state in stage 1
* *ub\_state* (vector): upper bound of the state in stage 1

[3] The boolean entry, *is\_infinite*, is added to the extension dictionary of the *Subproblem* object

# forwardpass!

[1] If using infinite-horizon SDDP set the starting state (of stage 1) in the forward pass to be equal to the end state of stage T, after the forward pass has been applied, of the previous forward pass. This can only occur after the first iteration of SDDP has been applied in a given “inner loop”. This is done because we want the iterations of SDDP to “wrap-around”, and the starting state of a given iteration of SDDP starts when the previous iteration of SDDP finished.

[2] If using infinite-horizon SDDP and the first iteration of SDDP of a given “inner loop” has been completed, and we are in the first stage set **fp\_start\_state** to be equal to the starting state of the forward pass.

[3] If using infinite-horizon SDDP and we are in the final stage, set *fp\_end\_state*to be equal to the state after the forward pass has been applied to this final stage.

# backwardpass!

[1] If using infinite-horizon SDDP apply backward pass to stage 1 as well as we need to generate cuts from stage 1 to be later added to stage T.

[2] Before the subproblem is solved for stage 1 in the backwards pass, set the state of stage 1 to be the starting state in stage 1 before forward passes was applied (held in *fp\_start\_state*). This is because for other stages the states is set to the states of the previous stage, after the forward pass had been applied, however as we are in stage 1 there is no previous stage.

[3] If applying a backward pass to stage 1, call *modifyvaluefunction!*with the subproblems of stage T instead of stage t-1. This contributed to the “wrap-around” idea of infinite-horizon SDDP.

[4] Set the state of stage 1 back to what it was before item [2] of the backwards pass was applied. This is because for computing the initial bound via *compute\_initial\_bound*did not expect the state of stage 1 to be changed as done in [2].

# modifyvaluefunction!

[1] If in final stage (which only occurs in infinite-horizon SDDP due to *backwardpass! item [3]*), use transition probabilities from stage 1 instead of from stage as . This is due to the “wrap around” of infinite-horizon SDDP. However, as only 1 Markov state in stage 1 this doesn’t make sense, perhaps we should use the transition probabilities from stage T?

[2] If in final stage, cuts are not added to value function, they are just written with *writecut!* as we add them later after shifting them down by .

# constructcut!

[1] The state the cut was sampled at must be specified when calling the cut constructor due to *type\_definitions [1].*

[2] Specify the sample state for the cut when constructing cuts at Stage T. The sample state is the final state of the forward pass in stage T in the previous iteration of SDDP. This state is held in *fp\_start\_state.*

# addasynccut!

[1] We now don’t call *addcut!* (which adds cuts to the value function as in *modifyvaluefunction! [2]* if cut is from the final stage T. In standard SDDP this would have never occurred as no cuts are produced for the final stage T so we must add a conditional statement. In infinite-horizon SDDP stage T cuts are added later after shifting them down by and then calling *loadcuts!*.

# writecut!

[1] As cut objects now have the property of the state they were sampled at (due to *type\_definitions [1]*) we now write this sampled state property of the cut when writing the cut.

# setstates!

[1] The setstates! was extended to have the option to specify the state the we want to set in the given subproblem

# loadcuts!

[1] We must now specify the state the cut was sampled at when calling the cut constructor (as per *type\_definitions.jl [1]*).

[2] An alternative loadcuts! method was developed that allows a matrix (each row contains cut information) to be passed to the function instead of a filename.

# setstageobjective!

[1] If using infinite horizon SDDP, stage T now has a future expected cost-to-go, . Previously, the objective of stage T just included the stage objective but if using infinite-horizon SDDP we now add into the objective in stage T as well.

# getstageobjective!

[1] If using infinite horizon SDDP, we must subtract future expected cost-to-go, from the (subproblem) objective for stage T. This is because is part of the objective of the JuMP subproblem but we just want the cost/value accrued in this stage.

# Solve

[1] Two additional inputs can be passed to the *solve* function when solving with infinite horizon SDDP

* *update\_limit* (Int): The maximum number of “inner loops” to apply
* *temp\_dir* (String): The directory for temporary files used in infinite-horizon SDDP to be written/read from/to

[2] If the *SDDPModel* passed to *solve* was created with infinite-horizon SDDP, we create some extra variables

[3] The body of the original *solve* function is now inside a loop. If using standard SDDP this loop is run once however if using infinite-horizon SDDP this loop is run up to *update\_limit* times.

[4] Cuts cannot be currently deleted from JuMP problems. In infinite-horizon SDDP many cuts become redundant as the terminal cost-to-go is built up (from nothing). At the start of each “inner loop” the dominating cuts are added to a blank (i.e. no cuts) SDDPModel of the given SDP.

(There is an existing cut selection method in SDDP.jl however due to running into some errors when trying to use it with infinite-horizon SDDP I made my own cut selection methods that took a different approach.

The method works by having all cuts written to file. After the start of each “inner loop”, the dominating set of cuts are added to a blank (i.e. no cuts) SDDPModel of the given SDP. Hence cuts are cached for *update\_limit* iterations then the dominating set of cuts is determined. By caching cuts then determining the dominating set of cuts, we can take advantage of matrix algebra which is computationally fast.

The existing cut selection method in SDDP.jl uses for loops (slow!). I initially developed my cut selection methods with loops and found >25% of the runtime was due to cut selection. Alternately with matrix algebra, cut selection took <1% of the runtime.)

[5] If using infinite-horizon SDDP, the stage T cuts are shifted down by a . These cuts (along with all other cuts generated) are saved. ( is the value of for inner loop which is a lower bound of the expected cost accrued of operating the policy for the time-horizon of the SDP).

[6] The lower bound, , and the standard deviation of produced from each “inner loop” are then written to the console. Convergence these values are a good proxy for convergence of the algorithm. ( is the maximum distance between the cut of inner loop and the dominating cut surface of “inner loop” , also ).

To do: Implement Dr Tony Downard’s future paper on a computationally efficient way of determining the numerical integral of an N-dimensional convex function. This method could be used to determine the convergence of the terminal future cost-to-go .