The Architecture of Green Energy Systems Summer 2024

Computation Survey

Computational tools play a crucial role in the analysis and computation of green energy system models, enabling accurate simulations and optimizations; however, individuals tend to have specific modeling languages that they are accustomed to using, which can pose a challenge when high-quality solvers do not support their preferred languages. Modeling language transformation offers a potential solution to this problem by allowing models to be translated between different languages, thereby enabling access to a wider range of solvers.

Nevertheless, the modeling language transformation approach has its limitations, particularly when it comes to capturing intricate problem structures, such as those found in stochastic programming models. Recognizing these challenges, we have conducted a survey of existing stochastic programming tools, exploring how models should be formulated using different languages. Our aim is twofold: first, to assist users in selecting the most suitable stochastic programming tools and modeling approaches for their needs; and second, to identify specific structures that could inform the development of more effective modeling language transformation mechanisms.

Additionally, we provide a tutorial on identifying core components in complex models, such as those implemented in SDDP.jl, and demonstrate how these components can be translated into the modeling languages that users prefer. This guidance is intended to help users more effectively manage the complexities of their models while leveraging the tools and languages with which they are most comfortable.

1 Modeling Transformation

In the field of algebraic modeling, there is an interest in the ability to transform models across various algebraic modeling languages. This capability is particularly valuable because it enables access to a broader range of solvers, which may be available only in specific modeling languages as the development of a specific solver link might not have been implemented in certain languages. The motivation behind such transformations is rooted in the fact that different solvers often have unique strengths and are optimized for particular types of problems. By transforming models between languages, users can take advantage of these specialized solvers, thereby enhancing the efficiency and effectiveness of their computational efforts.

Some people are interested in the direct transformation of modeling languages from the point of view of text files. However, the task of directly parsing model files at the text level presents significant challenges. One major obstacle is that users often write logically equivalent models in different ways, using varying syntax, conventions, or structures within a given language. This variability makes it difficult to achieve accurate transformations through simple text-based parsing alone. As a result, this approach is often insufficient for capturing the full complexity and nuances of the models.

To address this issue, a more robust approach involves focusing on the data structures that each modeling language uses to internally represent and store models. By understanding these data structures, it becomes possible to create intermediate functions or packages that facilitate the transformation of models between different languages at a deeper level. This method not only ensures greater accuracy in the transformation

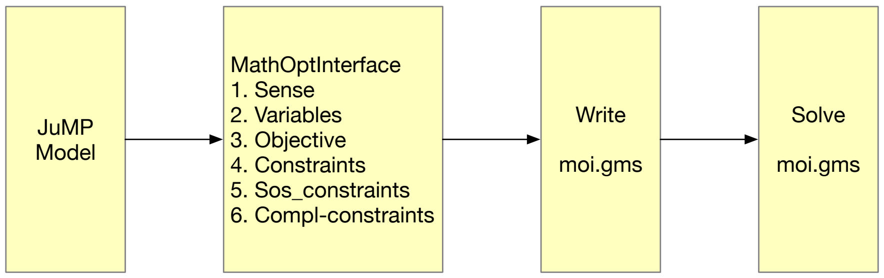

Figure 1: Workflow of GAMS.jl.

process but also provides a more flexible and scalable solution for working with complex models.

For example, GAMS.jl [1] is a tool that enables the transformation of models between Julia and GAMS by interacting with the underlying data structures Julia. Similarly, amplsolver, a solver in GAMS that allows GAMS model to be solved using solvers in AMPL [2], facilitates the transformation between GAMS and AMPL, allowing models to be seamlessly transferred. These examples illustrate how focusing on the data structures and developing intermediate tools can overcome the limitations of text-level parsing, ultimately enabling more effective and reliable model transformations across different algebraic modeling languages.

1.1 GAMS.jl

The GAMS.jl is a Julia package that utilizes the MathOptInterface[2], which is a Jump abstraction layer designed to provide a unified interface to mathematical optimization solvers so that users do not need to understand multiple solver-specific APIs, to implement transformation from a Julia model to a GAMS one. The workflow graph is in figure 1.

1.2 Amplsolver

Similar to GAMS.jl, the Amplsolver is a GAMS solver that allows GAMS model to be solved using the solvers available in AMPL, which is shown in figure 2. The data structure used to store the models in GAMS is the GAMS Modeling Object (GMO) [3].

1.3 Observations and discussion

We can observe that Julia and GAMS have their own data structures, MathOptInterface and GMO, to store models respectively. The existence of such general interfaces is crucial to perform modeling language transformation. The main efforts we need to put into when designing transformation packages lie in the ”write” steps in both figure 1 and 2. These steps involve querying the interfaces and writing out the model correctly in the target languages. This provides us with a more stable and accurate way to perform modeling language transformation.

However, there are still limitations to transformation like these. In figure 1 and 2, we have intermediate files, moi.gms, and xprob.nl, which are later used to be solved in the target languages. These models are scalar

[1] https://github.com/GAMS-dev/gams.jl
[2] https://github.com/coin-or/GAMSlinks
[3] https://www.gams.com/latest/docs/API_gmodesign.html

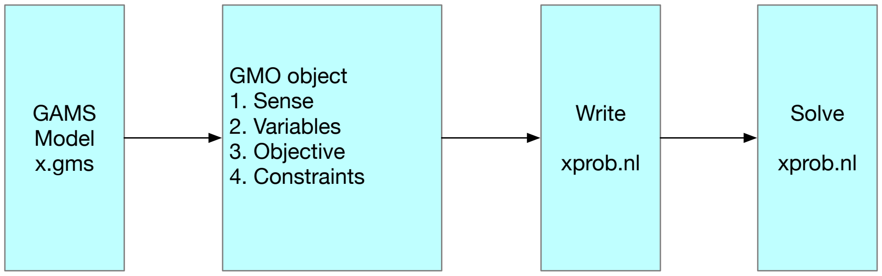

Figure 2: Workflow of Amplsolver.

models in that every variable is regarded as a single variable, and the constraints are all separated even if there might be some structural relationships among them. For example, in many stochastic programming solvers, constraints and variables are separated into different stages, so the solvers can automatically reformulate these problems in deterministic equivalent form or can use more advanced solving techniques such as Bender's decomposition. It might be interesting to look at how more structural information can be incorporated into the model data interfaces to allow more structures to also be transformed into the target language.

2 Stochastic Programming Tools

Stochastic programming plays a pivotal role in ensuring that decision-making processes are robust, as it incorporates uncertainty into the planning and optimization of systems. This is particularly essential in the realm of green energy systems, where variables like weather conditions, which can be unpredictable and fluctuate significantly, directly impact the performance and reliability of energy generation and distribution. As such, decisions in this field must consider these uncertainties to develop resilient and adaptive strategies.

To effectively manage these uncertainties, it is important to explore and utilize a variety of tools specifically designed for stochastic modeling. Different tools offer unique capabilities and approaches, and understanding how each handles the complexity of stochastic problems is key to making informed decisions. In this context, we introduce a simple two-stage stochastic programming example, which serves as a foundational case for demonstrating the application of various tools.

We solve this example model using two different stochastic programming tools, StochasticPrograms.jl, and GAMS EMP. Each of these tools has its own method for structuring and solving stochastic models, and by applying them to the same problem, we can gain valuable insights into their respective strengths and weaknesses. Our analysis focuses on how each tool structures the model, and the ease of use.

We look at the following model:

x	units bought
i	inventory
l	lost sales
s	units sold

$$\begin{aligned} \max \quad & -30x + \mathbb{E}[Q(x,\xi)] \\ Q(x,\xi) := \max \quad & 60s - 10i - 5l \\ st. \quad & d_\xi = s + l \\ & i = x - s \end{aligned}$$

2.1 GAMS EMP implementation

The following is a GAMS implementation [4] of the model using the Extended Mathematical Programming (EMP). The detailed documentation is on GAMS official website [5].

The first component of the stochastic model is the core model, it contains all the equations and the variables that appear in the mathematical model.

```
$ontext
 https://www.gams.com/latest/docs/UG_EMP_SP.html#INDEX_EMP_22_stochastic_21_programming
$offtext

Variable z      "profit";
Positive Variables
         x      "units bought"
         i      "inventory"
         l      "lost sales"
         s      "units sold";

Scalars  c      "purchase costs per unit"      / 30 /
         p      "penalty shortage cost per unit" / 5 /
         h      "holding cost per leftover unit" / 10 /
         v      "revenue per unit sold"        / 60 /
         d      "demand, random parameter"     / 45 /;

Equations profit "profit to be maximized"
         row1   "demand = UnitsSold + LostSales"
         row2   "inventory = UnitsBought - UnitsSold";

profit.. z =e= v*s - c*x - h*i - p*l;
row1..   d =e= s + l;
row2..   i =e= x - s;

Model nv / all /;
```

The next one is the emp.info file. In the file, we will determine the stages a variable or a constraint belongs to and also give the values and the associated probability distribution of the random variable using the keyword ”randvar”.

```
File emp / '%emp.info%' /;
put emp '* problem %gams.i%'/;
$onput
randvar d discrete 0.7 45 0.2 40 0.1 50
stage 2 i l s d
stage 2 Row1 Row2
$offput
putclose emp;
```

Finally, we need to define a scenario dictionary ”dict” and use it when we are solving the stochastic model.

[4]all code are accessible at https://github.com/WalterLu3/computational_group
[5]https://www.gams.com/latest/docs/UG_EMP_SP.html

```
Set scen         "scenarios" / s1*s3 /;
Parameter
     s_d(scen)    "demand realization by scenario"
     s_x(scen)    "units bought by scenario"
     s_s(scen)    "units sold by scenario"
     s_rep(scen,*) "scenario probability" / #scen.prob 0/;

Set dict / scen .scenario.''
          d    .randvar .s_d
          s    .level   .s_s
          x    .level   .s_x
          ''   .opt     .s_rep /;

solve nv max z use EMP scenario dict;
display s_d, s_x, s_s, s_rep;
```

2.2 stochasticProgram.jl

In stochasticProgram.jl, instead of tagging the variables and constraints, we directly formulate the subproblem and declare the random variable in each stage.

```
using GLPK
using HiGHS
using StochasticPrograms

@stochastic_model simple_model begin
    @stage 1 begin
        @decision(simple_model, x >= 0)
        @objective(simple_model, Max, -30*x)
    end
    @stage 2 begin
        @uncertain d
        @recourse(simple_model, 0 <= s)
        @recourse(simple_model, 0 <= i)
        @recourse(simple_model, 0 <= l)
        @objective(simple_model, Max, 60*s - 10*i - 5*l)
        @constraint(simple_model, d == s + l)
        @constraint(simple_model, i == x - s)
    end
end
```

Finally, we assign the probability distribution of random variables and pass them into the model.

```
e1 = @scenario d = 45 probability = 0.7
e2 = @scenario d = 40 probability = 0.2
e3 = @scenario d = 50 probability = 0.1

sp = instantiate(simple_model, [e1, e2, e3], optimizer = GLPK.Optimizer)

optimize!(sp)

obj = objective_value(sp)
```

2.3 Comparison

In the two implementation of stochastic programming model above, we can see that in GAMS EMP, users first need to specify a core model, and they need an extra information file ”emp.info” to tag the assignment of stages of each variable and constraint. Finally, we need to use some specified data dictionary structures to allow the model to be solved using data from different scenarios. The main complexity of this modeling formulation lies in the tagging part. As the number of stages gets larger, it is easy for the model to make mistakes when assigning variables and constraints to the correct stage. As for the Julia implementation, it declares the model in different stages, saving the need to manually tag the variables and constraints. However, as the number of stages gets larger, it is harder to see the structure of the core model, as users need to write each stage model out explicitly. Both methods allow us to use more advanced solution techniques, such as Bender's Decomposition, provided by specific solvers.

2.4 Some other existing stochastic optimization packages.

Here we list some other possible tools of interest for modeling stochastic programming problems. We eventually decided not to include these in our implementation examples because of the lack of maintenance of these tools.

PySP [6] (Pyomo Stochastic Programming) is a package within the Pyomo optimization framework designed for solving stochastic programming problems. It provides tools to define scenario trees that represent different stages of decision-making, where each node corresponds to a decision point and branches represent possible outcomes. By extending deterministic optimization models in Pyomo to stochastic models, PySP allows the incorporation of scenario data, random variables, and decision rules that depend on the realization of uncertainty. Additionally, PySP offers methods to solve these stochastic models using algorithms like Progressive Hedging.

SAMPL [7](Stochastic AMPL) is an extension of the AMPL framework designed to handle stochastic programming problems. It is a development version of AMPL that incorporates additional syntax such as trees and scenario sets to model stochastic programs. Users can extend their deterministic AMPL models to include these scenarios, enabling the formulation of stochastic models. SAMPL provides the necessary syntax and tools to define scenario trees, specify random variables, and implement decision rules that depend on the realization of uncertainty. SAMPL integrates seamlessly with AMPL's existing optimization solvers, allowing for the solution of complex stochastic programs using algorithms designed for such problems.

All of the tools mentioned above are used to model stochastic programming problems in such a way that the stochastic tree structure can be exploited and utilized by advanced decomposition or solution methods.

3 More Specific Stochastic Model - SDDP

Stochastic Dual Dynamic Programming (SDDP) is an advanced optimization algorithm designed to solve multi-stage stochastic programming problems, particularly those involving decision-making over time under uncertainty. It is especially effective for problems with a large number of scenarios, such as those found in energy planning, water resource management, and supply chain optimization. SDDP works by decomposing the problem into a series of subproblems, each corresponding to a stage in the decision process. These subproblems are solved iteratively, using dual information to construct cutting planes that approximate the cost-to-go functions, which represent the future costs based on current decisions. This iterative process

[6]https://pysp.readthedocs.io/en/latest/pysp.html
[7]https://optirisk-systems.com/

allows SDDP to efficiently handle the computational complexity of large-scale problems by focusing on the most critical scenarios and decisions, making it a powerful tool for optimizing systems under uncertainty.

SDDP.jl [1] is a Julia package that implements the Stochastic Dual Dynamic Programming (SDDP) algorithm, providing a flexible and efficient tool for solving complex, multi-stage stochastic optimization problems. The package allows users to model their stochastic problems using straightforward, expressive syntax and supports various features, including parallelization and scenario management, to enhance computational efficiency.

To understand how to implement complex algorithms like Stochastic Dual Dynamic Programming (SDDP) when your native programming language is not Julia, we start by examining a simple example in SDDP.jl. This allows us to explore the structure and logic of the algorithm in a high-performance environment. We then translate and implement the SDDP algorithm in GAMS to solve the same example, providing a practical guide for users working in different programming languages. Throughout this process, we guide the reader in identifying the core components of such models, helping them understand the fundamental building blocks needed to implement SDDP and similar algorithms in their preferred programming environment.

3.1 Example in SDDP.jl

This example is demonstrated in the guide of SDDP.jl[8]. With the following formulation:

v_t	water storage at stage t
h_t	hydro-generation at stage t
s_t	hydro-spill at stage t
g_t	thermal-generation at stage t
C_t	thermal-generation cost at stage t
F_t	water inflow at t

$$\begin{aligned} Q(v_{t-1}) = \min \quad & C_t g_t + \mathbb{E}[Q(v_t)] \\ s.t \quad & v_t = v_{t-1} - h_t - s_t + F_t \\ & h_t + g_t = 150 \end{aligned}$$

SDDP.jl provides a straightforward interface for this problem. In the code example, the first step is to declare the state variable v_t that will be passed among stages and the control variables.

```
# State variables
@variable(subproblem, 0 <= volume <= 200, SDDP.State, initial_value = 200)
# Control variables
@variables(subproblem, begin
    thermal_generation >= 0
    hydro_generation >= 0
    hydro_spill >= 0
end)
```

Then, we need to specify the probability of our random variable

```
sigma = [0.0, 50.0, 100.0]
P = [1 / 3, 1 / 3, 1 / 3]
```

[8]https://sddp.dev/stable/tutorial/first_steps/

```
SDDP.parameterize(subproblem, sigma, P) do w
    return JuMP.fix(inflow, w)
end
```

Finally, we define the constraints and objectives at each stage.

```
# Transition function and constraints
@constraints(
    subproblem,
    begin
        volume.out == volume.in - hydro_generation - hydro_spill + inflow
        demand_constraint, hydro_generation + thermal_generation == 150
    end
)
# Stage-objective
if node == 1
    @stageobjective(subproblem, 50 * thermal_generation)
elseif node == 2
    @stageobjective(subproblem, 100 * thermal_generation)
else
    @assert node == 3
    @stageobjective(subproblem, 150 * thermal_generation)
end
return subproblem
```

The ease of usage is the main advantage of SDDP.jl. Users do not need to worry about the implementation details but just need to focus on how to formulate their models. In order to implement SDDP in other languages, we first need to know how we should treat the core model structure, and how the algorithm of SDDP works.

3.2 GAMS implementation of Water Reservoir example

Here we first provide a brief overview of how SDDP works. The main idea behind SDDP is to approximate the expected value term $\mathbb{E}[Q(v_t)] = \alpha$ with a piecewise linear function constructed by a set of hyperplanes in the form of $\alpha \geq \pi^T v_t + b$, i.e. Bender's cuts. In order to learn the policy, the algorithm has two main components.

1. Forward Pass: choose a single random realization at each stage and calculate the policy on it.
2. Backward pass: Using the policy we got from the forward pass, calculate the single Bender's cut on all scenarios and add them. Start this from the last stage and keep moving to the previous stage.

The following is the implementation of the same example in GAMS. We constructed this example based on the code of SDDP.gms, which is a completely different GAMS model instance solved with SDDP. The implementation here might not be the most efficient one. We will show the most relevant part of the model and skip the description of the implementation for forward and backward passes. More information can be found in the example file.

First, we define all the parameters and variables needed in our model.

```
set t 'stages'  /t1*t3/
```

```
    s 'scenarios' /s1*s3/
    j 'iteration index' / j1*j20 /
    i 'trial index' /i1*i2/;

Set
   tt(t) 'control set for stages'
   jj(j) 'dynamic j'
;

Parameter
    inflow(t)
    cost(t)   /t1 50
               t2 100
               t3 150/
;

Parameter
    sto_inflow(s) /s1 0
                  s2 50
                  s3 100/;

Positive Variable
    volume(t)
    thermalGeneration(t)
    hydroGeneration(t)
    hydroSpill(t)
;

Free    Variable ACOST  'approximation of cost';
Positive Variable ALPHA(t) 'approximation of future cost function (FCF)';

volume.up(t) = 200;
```

Then we define the constraints in the DP model. Note that it is different from SDDP.jl in that we write out the model for each stage using the dynamic set "tt(t)". We do this for the ease of implementing forward and backward passes, as we can just keep the constraints of stage t_1 by doing "tt(t) = no; tt(t_1) = yes;"

```
Equation
    stateTransfer(t) 'State update variable'
    demandSatisfy(t) 'Need to meet demand'
;

stateTransfer(tt(t))..
    volume(t) =e= volume(t-1) + 200$(sameas('t1',t)) - hydroGeneration(t) - hydroSpill(t) +
        inflow(t);

demandSatisfy(tt(t))..
    hydroGeneration(t) + thermalGeneration(t) =e= 150;
```

The next part constructs the approximation of the objective functions and also defines some constraints which will later be added as Bender's cut, which is the standard way for users to add cut in GAMS.

```
parameter
    cont_m(j,i,t) 'dual variables associated with the mass balance constraint'
    delta(j,i,t) 'RHS of the Benders cuts';

* no cuts at the beginning
cont_m(j,i,t) = 0;
delta(j,i,t) = 0;
jj(j)        = no;

Equation
    Obj_Approx
    Cuts(j,i,t);

Obj_Approx..
    ACOST =e= sum(tt(t), cost(t) * thermalGeneration(t))
             + sum(tt(t), ALPHA(t+1));

* no cuts for the leaf node
Cuts(jj,i,t)$(ord(t) < card(t))..
   ALPHA(t+1) - cont_m(jj,i,t+1)*volume(t) =g= delta(jj,i,t);

Model waterSDDP / all /;
```

We will skip the algorithm steps here where forward and backward passes are implemented. The idea is that we will use the dynamic subset $tt(t)$ to turn on and off the constraints, fix stage variables, and solve submodels in order to get the policy and Bender's cut.

In this example, we see that SDDP.jl is a powerful tool because it abstracts many of the algorithm's implementation details, making it easier for modelers to focus on the problem itself. However, as shown here, it is also possible to implement SDDP in other modeling languages. Doing so reveals additional considerations that arise when adapting the algorithm to solve a highly structured problem in a different environment. For instance, the core model is divided into distinct stages, and additional constraints are needed to approximate the expected value in the objective function.

4 Conclusion

In conclusion, while computational tools and modeling language transformations offer valuable solutions for optimizing green energy systems, they come with challenges, especially when dealing with complex problem structures like those in stochastic programming. By surveying existing tools and providing practical guidance on translating key components of complex models into different languages, we aim to empower users to make informed decisions about the tools and approaches best suited to their needs. This survey ultimately looks at the example of SDDP, trying to identify some additional considerations that arise when adapting the algorithm to solve a highly structured problem in a different environment.

References

[1] O. Dowson and L. Kapelevich. SDDP.jl: a Julia package for stochastic dual dynamic programming. *INFORMS Journal on Computing*, 33:27–33, 2021.

[2] B. Legat, O. Dowson, J. D. Garcia, and M. Lubin. MathOptInterface: a data structure for mathematical optimization problems. *INFORMS Journal on Computing*, 2021.