

# Parallel algebraic modeling for stochastic optimization

Joey Huchette  
Operations Research Center  
Massachusetts Institute of  
Technology  
77 Massachusetts Avenue  
Cambridge, MA 02139  
huchette@mit.edu

Miles Lubin  
Operations Research Center  
Massachusetts Institute of  
Technology  
77 Massachusetts Avenue  
Cambridge, MA 02139  
mlubin@mit.edu

Cosmin Petra  
Mathematics and Computer  
Science Division  
Argonne National Laboratory  
9700 S. Cass Avenue  
Argonne, IL 60439  
petra@mcs.anl.gov

## ABSTRACT

We present scalable algebraic modeling software, StochJuMP, for stochastic optimization as applied to power grid economic dispatch. It enables the user to express the problem in a high-level algebraic format with minimal boilerplate. StochJuMP allows efficient parallel model instantiation across nodes and efficient data localization. Computational results are presented showing that the model construction is efficient, requiring roughly one percent of solve time. StochJuMP is configured with the parallel interior-point solver PIPS-IPM but is sufficiently generic to allow straight forward adaptation to other solvers.

## Keywords

optimization, parallel programming, high performance computing, mathematical model, Power system modeling, scalability

## 1. INTRODUCTION

Algebraic modeling languages (AMLs) for optimization are widely used by both academics and practitioners for specifying optimization problems in a human-readable, mathematical form, which is then automatically translated by the AML to the low-level formats required by efficient implementations of optimization algorithms, known as *solvers*. For example, a classical family of optimization problems is linear programming (LP), that is, minimization of a linear function of decision variables subject to linear equality and inequality constraints. As input, LP solvers expect vectors  $c$ ,  $l$ ,  $u$ ,  $L$ , and  $U$  and a sparse constraint matrix  $A$  defining the LP problem,

$$\begin{aligned} & \underset{x}{\text{minimize}} \quad c^T x \\ & \text{subject to} \quad L \leq Ax \leq U \\ & \quad \quad \quad l \leq x \leq u, \end{aligned}$$

where inequalities are componentwise.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

HPTCDL November 16-21, 2014, New Orleans, Louisiana, USA  
Copyright 2014 ACM 978-1-4799-5500-8/14 ...\$15.00.

In practice, it is tedious and error-prone to explicitly form a sparse constraint matrix and corresponding input vectors; this process often involves concatenating different classes of decision variables, each indexed over multidimensional symbolic sets or numeric ranges, into a single  $x$  decision vector. Instead, AMLs, which may be considered *domain-specific languages* in the vocabulary of computer science, handle these transformations automatically.

Notable AMLs include AMPL [1], GAMS [2], YALMIP [3], Pyomo [4], and CVX [5]. While commercial AMLs (AMPL and GAMS) offer standalone environments, open-source AMLs are typically embedded in high-level dynamic languages; CVX and YALMIP are both toolboxes for MATLAB, and Pyomo is a package for Python. Indeed, dynamic languages provide convenient platforms for AMLs, and domain-specific languages in general, because they make development easier (no need for a customized parser), and promote the ease of use by being accessible from a language that is already familiar to users and has its own tools for processing and formatting data.

Historically, speed has been a trade-off for using AMLs embedded in high-level dynamic languages, primarily because of their extensive use of operator overloading within an interpreted language. These AMLs may be orders of magnitude slower at *generating the optimization model* before passing it to a solver; in some reasonable use cases, this model generation time may perversely exceed the time spent in the solver.

The Julia programming language [6] presented an opportunity to address this performance gap. Lubin and Dunning developed JuMP [7], an AML embedded in Julia. By exploiting Julia's metaprogramming and just-in-time compilation functionality, they achieved performance competitive with that of commercial AMLs (AMPL and Gurobi/C++) and an improvement of one to two orders of magnitude over open-source AMLs (Pyomo and PuLP [8]) in model generation time for a benchmark of LP problems.

In this paper, we present an extension to JuMP, called StochJuMP, for modeling a class of stochastic optimization problems, namely, two-stage stochastic optimization with recourse, a popular paradigm for optimization under uncertainty [9]. These problems are computationally challenging and at the largest scale require the use of specialized solvers employing distributed-memory parallel computing. These solvers require structural information not provided by standard AMLs, and because of memory limits, it may not be feasible to build up the entire instance in serial and then distribute the pieces. Instead, StochJuMP generates the model



nical perspective, efficiently handling the highly structured problem in a parallelized setting requires care.

The SML project [16] is an extension built on top of AMPL for conveying multistage structure to solvers by using a new `block` keyword. SML is implemented as a pre- and post-processor to AMPL itself; as a result, the subproblems must be constructed as intermediary AMPL problem files on disk. The more recent work on PSMG [17] parses SML files and provides parallel model instantiation, avoiding the memory bottleneck that can arise when attempting to construct massive problem instances on a single node. While PSMG aims to address the same technical challenge as `StochJuMP`, we emphasize that PSMG is a large C++ project that required considerable development effort, based on our discussions with the authors. The development of `StochJuMP`, on the other hand, was greatly accelerated by being written in a high-level dynamic language at no significant penalty in total solution time.

We will describe details of the implementation and the use of `StochJuMP`. Throughout, we will refer to the code in Listing 1, which is the entirety of the Illinois model used in the computational results section. For brevity, we omit the somewhat tedious code that reads the problem data from file into memory.

`JuMP` represents all data required to describe an optimization problem in a `Model` data type in Julia; `StochJuMP` extends this by attaching new data to the `Model`: `parent`, a reference to the parent block, and `children`, a vector of references to children blocks. Since each block is a *bona fide* `Model` object, all methods to describe variables and constraints apply to blocks in a `StochJuMP` model. Additionally, using Julia’s scoping rules, it is possible to structure the model specification such that a child block can include variables from parent blocks in constraints. This allows the description of arbitrary nested block structure in the model, all within a very lightweight extension to the existing `JuMP` infrastructure. Note that no alteration to the `JuMP` source code is made to accommodate this; an “extension dictionary” is attached to the `Model` by default, allowing third-party packages to attach arbitrary structured data for external use.

As a brief primer on `JuMP` and `StochJuMP` syntax, we discuss the functions and macros that appear in the example code in Listing 1. The `StochasticModel(NS)` method defines a stochastic model container object with `NS` scenarios. Similarly, the `StochasticBlock(m)` method is a thin wrapper around a `Model` constructor, specifying that `m` is the parent model.

The macro `@defVar` defines a variable or dictionary of variables, attached to a particular model, for use in describing the problem constraints and objective. The first argument is the model object; the second argument is a description of the variable name, the index set used for indexing into the dictionary of variables, and appropriate variable bounds. For instance,

```
1 @defVar(m, 0 <= Pgen_f[i=GENTHE] <= np_capThe[i])
```

constructs a dictionary named `Pgen_f`, attached to the model `m`, for which indexing is defined by the iterable object `GENTHE`. The lower bound is zero for every component, and the  $i$ th entry has an upper bound from the  $i$ th entry in the array `np_capThe`.

The `@addConstraint` macro adds a set of constraints to a specified model. For instance,

```
1 @addConstraint(bl, t_w_con2[g=GENWIN],
2 tw[g] >= gen_cost_win[g]*PgenWin_f[
g])
```

adds a set of constraints to the model `bl`. The second argument specifies a constraint identifier that is indexed into in the same way as in `@defVar`. The third argument takes an algebraic description of constraints to be added. Similarly, the `@setObjective` macro takes an algebraic description of the objective function, as well as an associated `Model` object and a value specifying whether the model is a minimization or maximization problem.

Again, we stress that minimal additions to `JuMP` have been made for this feature set: no new types have been introduced. A series of `Model` objects has been constructed with corresponding constraints (containing variables owned by other `Models`), and the hierarchical structure of the model is disentangled immediately prior to passing the problem data to PIPS-IPM. Furthermore, the nested structure can be arbitrarily more complex than that of the two-stage example illustrated here.

The `@second_stage` macro abstracts data localization in the model specification, allowing concise, rank-agnostic code to be distributed across the cores. The first argument `m` specifies the `Model` object to which to add the second stage structure. The second argument `node` is a global index for the particular scenario assigned to a particular node. More specifically, the macro expands to construct the global indices assigned to a particular compute node, and wraps the body of `@second_stage` (denoted with the `begin` and `end` delimiters) in a loop over these particular values. In this example, the loop index is `node`.

Since the blocks in a `StochJuMP` model are specified with `JuMP` models themselves, one can easily adapt the existing `JuMP` functions to construct the problem data needed by a particular solver. The repository for `StochJuMP` is available at <https://github.com/joehuchette/StochJuMP.jl>. All told, the code to specify nested model structure takes less than 100 lines of Julia code, whereas the interface to PIPS-IPM requires roughly 300 lines.

## 5. COMPUTATIONAL RESULTS

We present here computational results and analyze the efficiency of the construction of a model arising in the optimization of power grid.

### 5.1 Model

To test `StochJuMP`, we consider the economic wind dispatch model presented in [18], which we briefly describe here. The model attempts to capture the impact that wind supply correlation information has on economic dispatch, a problem solved in real-time by power grid operators in order to set market prices. The model uses data describing the power transmission grid in the state of Illinois. The model problem is as follows.

**Listing 1: An implementation in StochJuMP of the stochastic economic dispatch model described in Section 5.1**

```

1  m = StochasticModel(NS)
2
3  # Stage 0
4  @defVar(m, 0 <= Pgen_f[i=GENTHE] <= np_capThe[i])
5  @defVar(m, 0 <= PgenWin_f[i=GENWIN] <= np_capWin[i])
6  @defVar(m, -lineCutoff*Pmax[i] <= P_f[i=LIN] <= lineCutoff*Pmax[i])
7
8  # (forward) power flow equations
9  @addConstraint(m, pfeq_f[j=BUS],
10     +sum{P_f[i], i=LIN; j==rec_bus[i]}
11     -sum{P_f[i], i=LIN; j==snd_bus[i]}
12     +sum{Pgen_f[i], i=GENTHE; j==bus_genThe[i]}
13     +sum{PgenWin_f[i], i=GENWIN; j==bus_genWin[i]}
14     -sum{loads[i], i=LOAD; j==bus_load[i]} >= 0)
15
16 @second_stage m node begin
17     b1 = StochasticBlock(m)
18     # variables
19     @defVar(b1, 0 <= Pgen[i=GENTHE] <= np_capThe[i])
20     @defVar(b1, 0 <= PgenWin[i=GENWIN] <= windPower[node,i])
21     @defVar(b1, -lineCutoff*Pmax[i] <= P[i=LIN] <= lineCutoff*Pmax[i])
22     @addConstraint(b1, rampUpDown[g=GENTHE],
23         -0.1np_capThe[g] <= Pgen[g] - Pgen_f[g] <= 0.1np_capThe[g])
24     # (spot) power flow equations
25     @addConstraint(b1, pfeq[j=BUS],
26         +sum{P[i]-P_f[i], i=LIN; j==rec_bus[i]}
27         -sum{P[i]-P_f[i], i=LIN; j==snd_bus[i]}
28         +sum{Pgen[i]-Pgen_f[i], i=GENTHE; j==bus_genThe[i]}
29         +sum{PgenWin[i]-PgenWin_f[i], i=GENWIN; j==bus_genWin[i]} >= 0)
30     @defVar(b1, t[GENTHE] >= 0)
31     @addConstraint(b1, t_con1[g=GENTHE],
32         t[g] >= gen_cost_the[g]*Pgen_f[g] +
33         1.2gen_cost_the[g]*(Pgen[g]-Pgen_f[g]))
34     @addConstraint(b1, t_con2[g=GENTHE],
35         t[g] >= gen_cost_the[g]*Pgen_f[g])
36     @defVar(b1, tw[GENWIN] >= 0)
37     @addConstraint(b1, t_w_con1[g=GENWIN],
38         tw[g] >= gen_cost_win[g]*PgenWin_f[g] +
39         1.2gen_cost_win[g]*(PgenWin[g]-PgenWin_f[g]))
40     @addConstraint(b1, t_w_con2[g=GENWIN],
41         tw[g] >= gen_cost_win[g]*PgenWin_f[g])
42
43     @setObjective(b1, Min, sum{t[g], g=GENTHE} + sum{tw[g], g=GENWIN})
44 end

```

$$\begin{aligned}
& \min_{x, X_i(\omega)} \sum_{i \in G} (p_i x_i \mathbb{E}_\omega [p_i^+ (X_i(\omega) - x_i)_+ - p_i^- (X_i(\omega) - x_i)_-]) \\
& \text{s.t. } \tau_n(f) + \sum_{i \in G(n)} x_i = d_n, \quad \forall n \in \mathcal{N} \\
& \tau_n(F(\omega)) - \tau_n(f) + \sum_{i \in G(n)} (X_i(\omega) - x_i) = 0, \\
& \quad \forall n \in \mathcal{N}, \omega \in \Omega \\
& f, F(\omega) \in \mathcal{U}, \omega \in \Omega \\
& (x_i, X_i(\omega)) \in \mathcal{C}_i(\omega), i \in G, \omega \in \Omega
\end{aligned}$$

For this model,  $N$  is the number of nodes or buses,  $L$  is the set of transmission lines,  $\mathcal{C}$  is the set of buses, and  $\mathcal{G}$  is the set of all energy suppliers. We use the subset  $\mathcal{G}(n)$  to denote all those providers connected to a given node  $n$ . The forward dispatch values are  $x_i$  for each provider, and their spot quantity for a given scenario  $\omega$  is  $X_i(\omega)$ . Forward power flow through a line  $\ell$  is  $f_\ell$ . The demand for each node is  $d_n$ , which is assumed to be deterministic and inelastic. The function  $\tau_n$  maps the flow vector to a node  $n$ . We use  $v_1(n)$  and  $v_0(n)$  to denote the inflow and outflow lines, respectively, to node  $n$ . Bid prices are denoted by  $p_i$ , and  $p_i^+$  and  $p_i^-$  denote bid prices for real-time corrections. More explicitly, supplier  $i$  is capable of selling additional power at price  $p_i^+ > p_i$  or of buying power at  $p_i^- < p_i$ . The random scenarios  $\omega$  represent the randomness in the model and live in a probability space  $(\Omega, \mathcal{F}, \mathcal{P})$ . Using standard notation,  $(z)_+ := \max z, 0$  and  $(z)_- := \min -z, 0$ . Here,  $\mathcal{U}$  is a polyhedral set that constrains the maximum flow constraints on individual lines.

## 5.2 Weak-scaling experiment

We present a weak-scaling study on Blues, a cluster at Argonne National Laboratory with 310 compute nodes, each with two Sandy Bridge 2.6 GHz Pentium Xeon, for a total of 4,960 cores. The PIPS-IPM interior-point solver is used, with MA57 [19] as the underlying sparse linear algebra library.

The weak scaling study runs from 4 to 2,048 scenarios. The data for the first scenario are from empirical measurement, and the remaining scenarios are artificially generated from normally distributed perturbations of the first. We compare the model load time, modeling time, and solver computation time in Table 1; explanations of the interpretations of these timing blocks are detailed below. Note that we use MPI barrier commands at the end of each timing block to ensure representative results.

Because of current limitations in Julia, external packages loaded in a script (e.g., StochJuMP) must be compiled before execution of a program, and this is recorded in the module load time field. As development of Julia continues, precompilation of packages will likely be introduced into the core language, removing this additional cost.

In order to force compilation of the appropriate methods, a common technique in Julia is the ‘‘JIT warm-up,’’ where a much smaller instance of a trial is performed first in order to ensure that all appropriate methods have been compiled by the just-in-time (JIT) compiler for a fair timing. Specifically, we perform a JIT warm-up run with a single scenario and abort just before calling PIPS-IPM to solve the problem. Note that this warm-up run includes reading and broadcast-

N	Load Module	Generate Model	Solve
4	12.161	4.857	344.716
8	13.661	5.234	369.796
16	7.780	2.728	233.337
32	7.570	2.732	297.940
64	7.866	2.737	274.795
128	12.486	2.770	360.695
256	10.348	2.856	394.954
512	13.163	3.157	458.392
1024	13.056	3.414	888.038
2048	-	3.221	644.705

**Table 1: Weak scaling results (in seconds) for the Illinois power generation model.**

N	Load Module	Generate Model
4	3.528	1.409
8	3.694	1.415
16	3.334	1.169
32	2.541	0.917
64	2.863	0.996
128	3.462	0.768
256	2.620	0.723
512	2.871	0.689
1024	1.470	0.384
2048	-	0.500

**Table 2: Ratio of StochJuMP timings over solve time ( $\times 100$ ).**

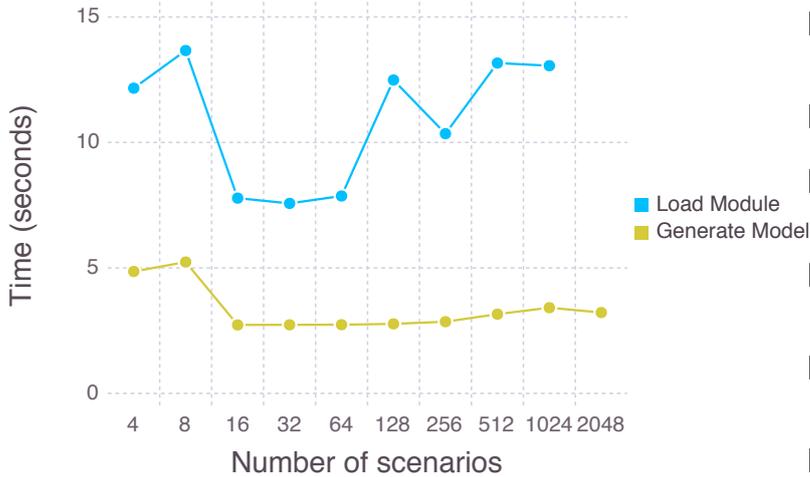
ing data, building the model, and constructing the appropriate callbacks to pass to the solver. For reference, the JIT warm-up runs took roughly twice as long as the model generation time.

Model generation time records how long it takes for the internal representation to be built up in memory, as well as the transformations necessary to construct the data required by PIPS-IPM for the solve process. This is performed in parallel across the cluster, independently for each scenario.

Table 2 shows that model generation time takes at worst 1.5% of the solve time for all experiments in the weak scaling trial. Additionally, the model generation displays favorable scalable properties compared to the PIPS-IPM underlying solver. This timing includes the time it takes to read the problem data from file on a single core and broadcast it to all worker cores.

There are two regimes in the data presented: the small trials with 4 and 8 scenarios take roughly twice as long as larger instances across the timing metrics. We believe this result is attributable to the saturation of the 16 cores on each node. All experiments assigned one MPI process per core. We use the MPI.jl Julia package to interface with MPI in Julia code. MPI is used for all internode communication, both in StochJuMP to distribute data during model generation and to synchronize nodes via MPI\_Barrier, and internally in PIPS-IPM. Figure 1 shows this change in performance clearly, as well as the nice scaling behavior for StochJuMP.

Finally, we emphasize again that the infrastructure in StochJuMP is designed with solver independence in mind. The problem data is stored abstractly as a structured set of JuMP Models, and it is only at solve time (not model



**Figure 1: Timing results for the weak-scaling trials.**

construction time) that the data is converted to a format specific to PIPS-IPM. In the future we envision a straightforward adaptation for support for other stochastic optimization solvers, as well as add a framework for solving such problems using traditional LP and MIP solvers.

## 6. CONCLUSION

In this paper we presented StochJuMP, algebraic modeling software for multi-stage stochastic optimization. StochJuMP is built as an extension to JuMP, a fully-featured and performant algebraic modeling language embedded in the Julia programming language. It is implemented as a front-end for the PIPS-IPM solver and supports model instantiation in parallel. In a weak-scaling experiment we observe extremely good scaling and modeling times which are a very small fraction of solve times ( $\approx 1.5\%$ ). In addition to performance, we emphasize the fast prototyping and development that made StochJuMP possible, due to Julia and the JuMP framework.

## Acknowledgments

We gratefully acknowledge the use of computing resources provided on the Blues high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory. This material is based upon work and resources supported by the U.S. Department of Energy Office of Science under Contract No. DE-AC02-06CH11357. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1122374. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. M. Lubin was supported by the DOE Computational Science Graduate Fellowship, which is provided under grant number DE-FG02-97ER25308.

## References

- [1] R. Fourer, D. M. Gay, and B. W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*, 2nd edition. 2002.
- [2] A. Brooke, D. Kendrick, A. Meeraus, and R. Raman, *GAMS: A User's Guide*. Scientific Press, 1999.
- [3] J. Löfberg, "YALMIP: a toolbox for modeling and optimization in MATLAB," in *Proceedings of the CACSD Conference*, 2004.
- [4] W. Hart, J.-P. Watson, and D. Woodruff, "Pyomo: modeling and solving mathematical programs in Python," *Math. Prog. Comp.*, vol. 3, pp. 219–260, 2011.
- [5] M. Grant and S. Boyd, *CVX: MATLAB software for disciplined convex programming, version 2.1*, <http://cvxr.com/cvx>, 2014.
- [6] J. Bezanson, S. Karpinski, V. Shah, and A. Edelman, "Julia: a fast dynamic language for technical computing," *CoRR*, vol. abs/1209.5145, 2012.
- [7] M. Lubin and I. Dunning, "Computing in operations research using Julia," *ArXiv e-prints*, vol. abs/1312.1431, 2013.
- [8] S. Mitchell, M. O'Sullivan, and I. Dunning, "PuLP: a linear programming toolkit for python," 2011, unpublished manuscript. [Online]. Available: <https://code.google.com/p/pulp-or/>.
- [9] J. R. Birge and F. Louveaux, *Introduction to stochastic programming*. Springer-Verlag, 1997.
- [10] S. Mehrotra and M. G. Ozevin, "Decomposition based interior point methods for two-stage stochastic convex quadratic programs with recourse," *Oper. Res.*, vol. 57, no. 4, pp. 964–974, 2009.
- [11] A. Shapiro, D. Dentcheva, and A. Ruszczyński, *Lectures on Stochastic Programming: Modeling and Theory*. Philadelphia, PA: MPS/SIAM Series on Optimization 9, 2009.
- [12] S. Mehrotra, "On the implementation of a primal-dual interior point method," *SIAM Journal on Optimization*, vol. 2, no. 4, pp. 575–601, 1992.
- [13] M. Lubin, C. G. Petra, M. Anitescu, and V. Zavala, "Scalable stochastic optimization of complex energy systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, Seattle, WA: ACM, 2011, pp. 64:1–64:64.
- [14] C. G. Petra, O. Schenk, M. Lubin, and K. Gärtner, "An augmented incomplete factorization approach for computing the Schur complement in stochastic optimization," *SIAM Journal on Scientific Computing*, vol. 36, no. 2, pp. C139–C162, 2014.
- [15] C. G. Petra, O. Schenk, and M. Anitescu, "Real-time stochastic optimization of complex energy systems on high performance computers," *Computing in Science and Engineering*, vol. 99, no. PrePrints, pp. 1–9, 2014.
- [16] M. Colombo, A. Grothey, J. Hogg, K. Woodsend, and J. Gondzio, "A structure-conveying modelling language for mathematical and stochastic programming," *Math. Prog. Comp.*, vol. 1, pp. 223–247, 2009.

- [17] A. Grothey and F. Qiang, “PSMG: A parallel problem generator for structure conveying modelling language for mathematical programming,” *presentation at IC-COPT 2013*, 2009.
- [18] C. G. Petra, V. Zavala, E. Nino-Ruiz, and M. Anitescu, “Economic impacts of wind covariance estimation on power grid operations,” *Preprint ANL/MCS-P5M8-0614*, 2014.
- [19] I. S. Duff, “MA57—a code for the solution of sparse symmetric definite and indefinite systems,” *ACM Trans. Math. Softw.*, vol. 30, no. 2, pp. 118–144, Jun. 2004.