GRAVITY: A Modeling Language for Mathematical Optimization and Machine Learning

Hassan Hijazi*,1,2, Guanglei Wang[2], Carleton Coffrin[1]

[1] Los Alamos National Laboratory, Los Alamos, New Mexico, USA
* corresponding author, hlh@lanl.gov
[2] The Australian National University, Canberra, ACT, Australia

Abstract. GRAVITY is an open source, scalable, memory efficient modeling language for solving mathematical models in Optimization and Machine Learning. It exploits structure to reduce function evaluation time including Jacobian and Hessian computation. GRAVITY is implemented in c++ with a flexible interface allowing the user to specify the numerical accuracy of variables and parameters. It is also designed to handle iterative model solving, convexity detection, distributed algorithms, and constraint generation approaches. When compared to state-of-the-art modeling languages such as JUMP, GRAVITY is 5 times faster in terms of function evaluation and up to 60 times more memory efficient. It also dominates commercial languages such as AMPL on structured models including quadratically-constrained and polynomial programs. This short paper serves as a quick introduction to the language, presenting its main features along with some preliminary results, an extended version of the work is in progress.

1 Introduction

While modeling languages play a critical role across all scientific areas, they constitute a key stone in Computer Science and Applied Mathematics. Modeling tools are ubiquitous to fields like Constraint Programming (CP), Artificial Intelligence (AI) and Operations Research (OR). A non-exhaustive list of existing modeling languages include AMPL [1], JUMP [2], GAMS [3], AIMMS [4], CASADI [5], PYOMO [6,7], and MINIZINC [8]. With increasingly large data inputs and the desire to model complex nonlinear functions, the efficiency of modeling tools is becoming critical for the implementation of scalable solution algorithms. This is especially true for Machine Learning (ML) applications dealing with large data sets and nonlinear learning functions. Note that tools like TENSORFLOW[9] and THEANO[10] are highly specialized modeling languages in that area. Modern mathematical modeling languages have to comply to a new set of requirements. These include the ability to efficiently solve multiple replica of the same model, to implement lazy constraint generation, to automatically detect convexity, to change numerical precision of variables and parameters, to parallelize subproblem solving, and to handle large data sets, just to name a few. GRAVITY is designed with these requirements in mind.

2 Design Choices

GRAVITY's efficiency in speed and memory can mainly be attributed to its design choices. These choices are based on exploiting structure. Structure is what differentiate a Linear Program (LP) from a Quadratic Program (QP) from a general Nonlinear Program (NLP). Problems solved by scientists and practitioners will inherently have structure, and exploiting it is key. It is not necessary to build an expression tree of your model if the constraints and the objective have a predefined structure, e.g., are linear, quadratic or polynomial.

Template Constraints. Structure can also be exploited by considering the fact that most mathematical formulations have a small set of "template" constraints, i.e., an abstract/symbolic representation of the constraint where only variables and parameters' indices change.

Flexible Numerical Precision. Being able to dynamically select the numerical precision of variables, parameters and functions is another critical design choice made in GRAVITY.

Model Readability. Readability of the models was also a main driver during the design process. Examples presented in subsequent sections will reveal the effort put into making the formulations as user-friendly as possible.

Efficiency. Gravity is implemented in c++, allowing for a flexible memory management and various code optimization under the hood.

3 Building Blocks for Mathematical Models

3.1 Constants, Parameters and Variables

All classes inherit from the primitive constant class. The type and the numerical precision of a constant can be dynamically set by the user. Currently supported types are presented in Code Block 1.

Code Block 1 Declaring Constants in GRAVITY

```
#include <gravity/solver.h>
using namespace gravity;
constant<> cd; /* Default type is double */
constant<bool> cb; /* Binary constant */
constant<short> cs; /* Integer constant with short precision */
constant<int> ci; /* integer constant */
constant<float> cf; /* Real constant with float precision */
constant<double> cd; /* Real constant with double precision */
constant<long double> cld; /* Real constant with long double precision */
```

Parameters are used to represent named constants, i.e., input data that is assigned a name and a set of values. The same types presented in Code Block 1

apply for parameters. Code Block 2 presents examples on declaring different parameters and assigning corresponding values. Finally, Code Block 3 illustrates

Code Block 2 Declaring Parameters in GRAVITY

```
param<int> p("int_p"); /* Integer constant */
p(1) = 0; /* Indexing can also be done using string indices */
p(5) = 1;
p.print();/* Will print:  int_p = [ (1=0) (5=1) ];*/
```

the declaration of variables, which are treated as bounded parameters.

Code Block 3 Declaring Variables in GRAVITY

```
var<> x("x", -0.1, 3.5); /* Default type is double */
x(0).print(); /* Will print: x[0] in  [ -0.1, 3.5]; */

param<int> lb("lb"), ub("ub");
lb("bus1") = -1;
ub("bus1") = 2;
var<int> y("y", lb, ub);
y("bus1").print(); /* Will print: y[bus1] in  [ -1, 2]; */
```

3.2 Functions and Constraints

In order to exploit structure, the building blocks of a function are split into four different parts. A function has a linear part, a quadratic part, a polynomial part, and an expression tree as depicted in equation 1.

$$f(\mathbf{x}) = \mathbf{a}^T\mathbf{x} + \mathbf{x}^T Q\mathbf{x} + P(\mathbf{x}) + E(\mathbf{x}), \tag{1}$$

where a^T denotes the transpose of the coefficient vector appearing in the linear part, Q represents the quadratic coefficients matrix, P denotes the polynomial part and E represents the expression tree gathering the remaining nonlinear terms in f. $E(\mathbf{x})$ can represent either unary or binary expressions, unary operators currently supported in GRAVITY include $exp, log, sqrt, sin$, and cos. Binary operators include multiplication, addition, subtraction, division and power. Depending on the function type, the corresponding part will be populated. A constraint is a function with a right-hand-side and an operator that can take values from $\leq, \geq$ and $=$. Constraint examples are given in Code Block 4.

Code Block 4 Declaring Constraints in GRAVITY

```
Constraint cstr1("cstr1");
cstr1 += power(x,4) - power(y, 2) + p*x*y + 2*x;
cstr1 <= 2;
cstr1.print(); /* Will print: cstr1 : x^4 + (int_p)y*x - y^2 + 2x <= 2; */
Constraint cstr2("cstr2");
cstr2 += cos(p*x) + expo(2*y);
cstr2 = 2;
cstr2.print(); /* Will print: cstr2 : cos((int_p)x) + exp(2y) = 2; */
```

3.3 Convexity Detection

GRAVITY has built-in detection procedures for convexity preserving operations and implements a set of sufficient conditions for checking function's convexity. Code Block 5 illustrates this feature on a quadratic constraint.

Code Block 5 Convexity Detection

```
param<int> a("a");
a(0) = -1;
a(5) = -3;
a.print(); /* Will print a = [ (0=-1) (5=-3) ]; */
auto f = a*power(x,2); /* f is a template constraint with two indices:
0 and 5. */
f.print(); /* Will print: Concave function: f(x) = (a)x^2, Gravity
detects that all instances of "a" are negative. */
f *= a;
f.print(); /* Will print: Convex function: f(x) = (a^2)x^2 */
```

3.4 Abstract Models: Template Constraints

Template constraints can help reduce computational time for operations that only require the symbolic structure of underlying functions, e.g., computing derivatives and detecting convexity. Code Block 6 showcases template constraints and the way GRAVITY handles their indexing.

Graph aware. Note that GRAVITY has an underlying graph implementation and indexing can be done on nodes, edges, and node pairs (pairs of nodes joined by one or more edges) of the graph as shown in Code Block 6. Some useful graph algorithms such as tree decomposition and cycle basis computation are also implemented in GRAVITY.

Code Block 6 Template Constraints in GRAVITY

```
var<> x("x"), y("y"), z("z",pos_); /* x and y are unbounded real variables,
z is a non-negative real variable */
/* Second-order cone constraints */
Constraint SOC("SOC");
SOC =  power(x, 2) + power(y, 2) - z.from()*z.to() ;
SOC.in(node_pairs) <= 0;
SOC.print(); /* Will print: SOC : x.in_index_pair^2 + y.in_index_pair^2
- z.to_index_pair*z.from_index_pair <= 0; */
```

3.5 Models and Solvers

A model is a collection of variables, constraints and an objective function. Populating a model can be done by invoking the function presented in Code Block 7.

Code Block 7 Model Declaration

```
Model SOCP("Second-Order Cone Program");
SOCP.add_var(x^10); /* Adding a variable vector of size 10 */
SOCP.add_constraint(SOC.in(node_pairs) <= 0); /* Adding second-order
constraints indexed by node_pairs (see previous Code Block) */
SOCP.min(x*y - 2*y); /* Declaring the objective function */
```

GRAVITY currently links to a small number of solvers including CPLEX [11], GUROBI [12], IPOPT[13], BONMIN[14] and MOSEK [15]. We plan to support more solvers such as SCIP [16], XPRESSMP [17], COUENNE [18] and SAT solvers such as MINISAT [19] in the near future. The way to invoke a solver is illustrated in Code Block 8.

Code Block 8 Invoking Solvers

```
if (use_cplex) {
    solver SCOPF_CPX(SOCP, cplex);
    SCOPF_CPX.run(output = 0, relax = false, tol = 1e-6);
}
else {
    solver SCOPF(SOCP,ipopt);
    SCOPF.run(output = 0, relax = false, tol = 1e-6, "ma27", mehrotra = "no");
    }
```

4 Symbolic vs. Automatic Differentiation

Automatic Differentiation (AD) has numerous benefits, including the ability to compute derivative for non-mathematical structures, e.g., computer code. There is an extensive literature on the subject, we only provide a few references herein [20,21,22,23,24]. However, the common belief that AD outperforms Symbolic Differentiation (SD) (see [23,25]) on mathematical models is inaccurate. Let us emphasize that AD is currently the norm in all state-of-the-art mathematical modeling tools including ML frameworks such as Theano [10] and TensorFlow [9]. In this work, we oppose the common belief, showing that SD can dominate AD when exploiting structure and using template constraints. By avoiding redundant storage of symbolic nonlinear expressions, our SD outperforms state-of-the-art AD implementations (e.g., [2], which uses graph coloring methods for exploiting sparsity of the Hessian matrix). The former approach is also better from a memory consumption point of view, as will be highlighted in the numerical experiments section.

5 Numerical Experiments

Numerical experiments were conducted on HPE ProLiant XL170r servers featuring two Intel2.10 GHz 16 Core CPUs and 128 GB of memory. Ipopt v3.12 [13] compiled with HSL [26] was used for solving Nonlinear Programs. In the results tables, "mem." indicates that available memory was exceeded for the corresponding instance.

5.1 AC Optimal Power Flow

The Alternating Current Optimal Power Flow (ACOPF) Problem is a fundamental building block in Power Systems Optimization. The problem admits two nonconvex NLP formulations, one known as the polar formulation (featuring trigonometric functions) and one known as the rectangular formulation (quadratically-constrained). A comprehensive description of the two formulations can be found in [27]. Figure 1 is a performance profile illustrating percentage of instances solved as a function of time. The figure compares Gravity, JuMP and Ampl's NL interface (used by Ampl and Pyomo) on all standard instances found in the pglib benchmark library [28]. The recorded time corresponds to the wall-clock time spent inside Ipopt. Figure 1 indicates that Gravity outperforms both JuMP and Ampl's NL interface on all instances, with speed improvements up to 300%. Let us emphasize that, for this problem, half of the computational time is dedicated to function evaluation (including Jacobian and Hessian). While we observe a factor 3 improvement in the total run time, the underlying improvement in function evaluation is a factor of 5. Table 1 reports detailed results on both formulations along with runtimes corresponding to the Second-Order Cone relaxation introduced in [29]. Note that $|N|$ and $|E|$ respectively denote the number of nodes and edges in the underlying power network.

The largest instance has 117,370 variables, 187,371 constraints, 666,023 non zeros in the Jacobian, and 299,384 non zeros in the Hessian matrix.

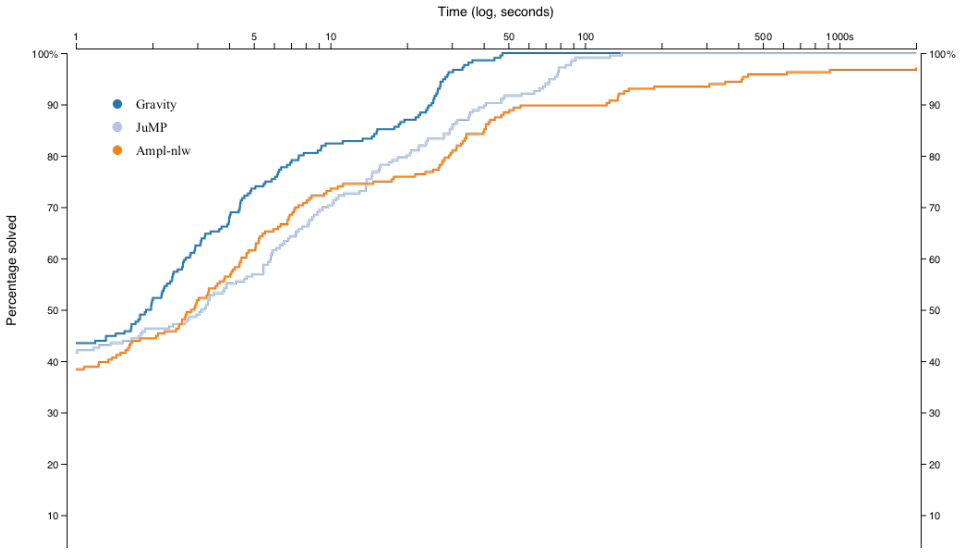

Fig. 1: Performance profile on Polar and Rectangular ACOPF (216 instances).

5.2 Learning of Ising Models

Learning the structure and the parameters of an Ising model is a typical Machine Learning problem where data input quickly becomes problematic (the learning relies on binary samples input). Matrices with 92 million non-zero entries appear in the biggest instances. A comprehensive formulation of the problem can be found in [30] and [31]. Figure 2 shows that both JuMP and Ampl crash after a certain threshold due to memory issues. Gravity is able to scale up on all tested instances. Since the problem has a natural parallelization procedure, a parallel implementation in Gravity was also tested on this problem. Figure 2 also compares Gravity running with 12 threads, showing gains up to one order of magnitude. Table 2 presents a detailed version of the results and includes different thread configurations. Let us emphasize that Gravity is not able to compute the Jacobian and Hessian matrices when parameters are in matrix form. This is a feature currently under development. Nevertheless, Gravity supports user-defined derivatives including matrices, the corresponding source code implementation can be downloaded here.

Table 1: Solver Runtime on AC Optimal Power Flow

			Solver Runtime (seconds)								
			AC Polar			AC Rect.			SOC Relax.		
Test Case	$\|N\|$	$\|E\|$	JuMP	NL	GVT	JuMP	NL	GVT	JuMP	NL	GVT
Typical Operating Conditions (TYP)											
pglib_opf_case3_lmbd	3	3	<1	<1	<1	<1	<1	<1	<1	<1	<1
pglib_opf_case5_pjm	5	6	<1	<1	<1	<1	2	<1	<1	<1	<1
pglib_opf_case14_ieee	14	20	<1	<1	<1	<1	<1	<1	<1	<1	<1
pglib_opf_case24_ieee_rts	24	38	<1	<1	<1	<1	<1	<1	<1	<1	<1
pglib_opf_case30_as	30	41	<1	<1	<1	<1	<1	<1	<1	<1	<1
pglib_opf_case30_fsr	30	41	<1	<1	<1	<1	<1	<1	<1	<1	<1
pglib_opf_case30_ieee	30	41	<1	<1	<1	<1	<1	<1	<1	<1	<1
pglib_opf_case39_epri	39	46	<1	<1	<1	<1	<1	<1	<1	<1	<1
pglib_opf_case57_ieee	57	80	<1	<1	<1	<1	<1	<1	<1	<1	<1
pglib_opf_case73_ieee_rts	73	120	<1	<1	<1	<1	<1	<1	<1	<1	<1
pglib_opf_case89_pegase	89	210	<1	<1	<1	<1	<1	<1	<1	<1	<1
pglib_opf_case118_ieee	118	186	<1	<1	<1	<1	<1	<1	<1	<1	<1
pglib_opf_case162_ieee_dtc	162	284	<1	<1	<1	<1	<1	<1	<1	<1	<1
pglib_opf_case200_pserc	200	245	<1	<1	<1	<1	<1	<1	<1	<1	<1
pglib_opf_case240_pserc	240	448	4	2	3	2	2	2	2	2	2
pglib_opf_case300_ieee	300	411	<1	<1	<1	<1	<1	<1	<1	<1	<1
pglib_opf_case1354_pegase	1354	1991	6	2	2	2	135	2	3	86	3
pglib_opf_case1888_rte	1888	2531	15	5	5	32	42	3	5	6	6
pglib_opf_case1951_rte	1951	2596	18	6	7	7	9	6	6	6	6
pglib_opf_case2383wp_k	2383	2896	9	3	3	4	5	3	6	6	6
pglib_opf_case2736sp_k	2736	3504	8	3	3	3	4	3	4	5	5
pglib_opf_case2737sop_k	2737	3506	6	3	2	3	4	2	4	4	4
pglib_opf_case2746wop_k	2746	3514	7	3	2	3	4	2	4	4	4
pglib_opf_case2746wp_k	2746	3514	8	3	3	3	4	2	5	5	5
pglib_opf_case2848_rte	2848	3776	21	7	8	6	8	6	7	8	8
pglib_opf_case2868_rte	2868	3808	23	7	8	7	11	10	8	9	10
pglib_opf_case2869_pegase	2869	4582	14	6	5	6	413	5	7	224	8
pglib_opf_case3012wp_k	3012	3572	12	5	4	5	7	4	8	8	9
pglib_opf_case3120sp_k	3120	3693	11	4	4	5	6	4	6	6	6
pglib_opf_case3375wp_k	3375	4161	14	mem.	5	16	mem.	7	26	105	7
pglib_opf_case6468_rte	6468	9000	79	30	29	49	122	27	27	30	30
pglib_opf_case6470_rte	6470	9005	48	18	16	19	27	15	23	26	25
pglib_opf_case6495_rte	6495	9019	89	35	30	35	51	26	24	27	27
pglib_opf_case6515_rte	6515	9037	73	28	25	28	40	23	23	26	26
pglib_opf_case9241_pegase	9241	16049	64	24	19	25	1999	19	38	42	38
pglib_opf_case13659_pegase	13659	20467	92	41	34	69	125	37	45	248	50

References

1. Fourer, R., Gay, D.M., Kernighan, B.: Algorithms and model formulations in mathematical programming. Springer-Verlag New York, Inc., New York, NY, USA (1989) 150–151
2. Dunning, I., Huchette, J., Lubin, M.: Jump: A modeling language for mathematical optimization. SIAM Review **59**(2) (2017) 295–320
3. Bussieck, M.R., Meeraus, A.: General algebraic modeling system (gams). Applied Optimization **88** (2004) 137–158
4. Bisschop, J.: AIMMS - Optimization Modeling. Lulu.com (2006)
5. Andersson, J., Åkesson, J., Diehl, M.: CasADI: A symbolic package for automatic differentiation and optimal control. In Forth, S., Hovland, P., Phipps, E., Utke, J., Walther, A., eds.: Recent Advances in Algorithmic Differentiation. Volume 87 of Lecture Notes in Computational Science and Engineering. Springer, Berlin (2012) 297–307

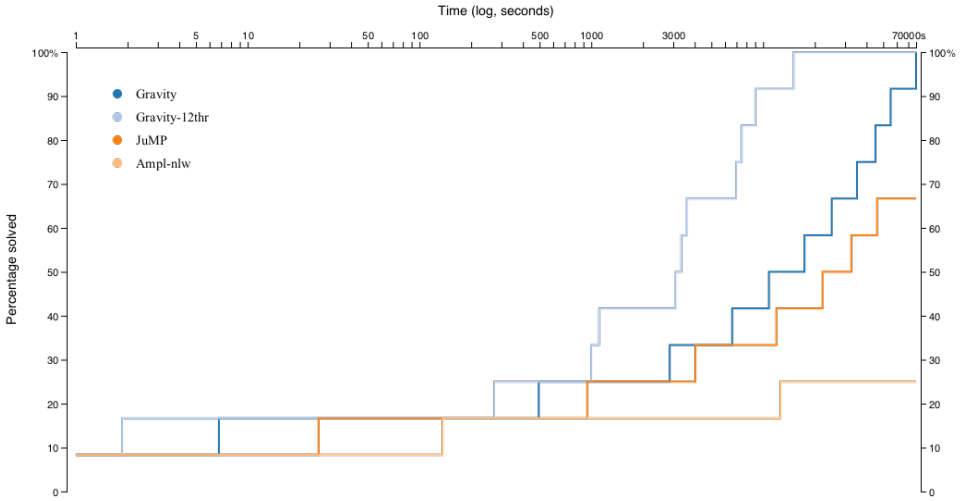

Fig. 2: Performance profile on Inverse Ising problems (12 instances).

Table 2: Runtimes on Inverse Ising Problems

		Total Runtime (seconds)							
$\|V\|$	$\|S\|$	JuMP	NL	GVT	GVT02	GVT04	GVT08	GVT12	GVT16
7	1,000,000	<1	<1	<1	<1	<1	<1	<1	<1
15	1,000,000	26	137	7	4	3	4	2	7
22	1,000,000	947	12609	495	249	183	195	273	175
30	1,000,000	4040	mem.	2862	1474	932	937	1000	1511
38	1,000,000	11945	mem.	6606	3272	2529	1958	1110	1700
46	1,000,000	22168	mem.	10818	5447	3280	2896	3343	4047
54	1,000,000	32624	mem.	17420	8651	5933	4164	3593	3379
61	1,000,000	46079	mem.	25103	13092	7274	5580	3090	6845
69	1,000,000	mem.	mem.	35187	17428	9649	7098	7511	5180
77	1,000,000	mem.	mem.	45057	24623	13783	9387	9011	12216
84	1,000,000	mem.	mem.	55296	34304	17696	11478	6965	7708
92	1,000,000	mem.	mem.	77720	40294	21563	15074	15057	16535

6. Hart, W.E., Watson, J.P., Woodruff, D.L.: Pyomo: modeling and solving mathematical programs in python. Mathematical Programming Computation **3**(3) (2011) 219–260
7. Hart, W.E., Laird, C.D., Watson, J.P., Woodruff, D.L., Hackebeil, G.A., Nicholson, B.L., Siirola, J.D.: Pyomo–optimization modeling in python. Second edn. Volume 67. Springer Science & Business Media (2017)
8. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: Minizinc: Towards a standard cp modelling language. In: Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming. CP'07, Berlin, Heidelberg, Springer-Verlag (2007) 529–543

9. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D.G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., Zheng, X.: Tensorflow: A system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). (2016) 265–283
10. Theano Development Team: Theano: A Python framework for fast computation of mathematical expressions. arXiv e-prints **abs/1605.02688** (May 2016)
11. Ibm: IBM ILOG CPLEX Optimization Studio CPLEX User's Manual. (2017)
12. Gurobi Optimization, I.: Gurobi optimizer reference manual (2017)
13. Wächter, A., Biegler, L.T.: On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming. Mathematical Programming **106**(1) (2006) 25–57
14. Bonami, P., Biegler, L.T., Conn, A.R., Cornuejols, G., Grossmann, I.E., Laird, C.D., Lee, J., Lodi, A., Margot, F., Sawaya, N., Wächter, A.: An algorithmic framework for convex mixed integer nonlinear programs. Discrete Optimization **5**(2) (2008) 186 – 204
15. Mosek ApS: The MOSEK optimization software
16. Achterberg, T.: SCIP: solving constraint integer programs. Mathematical Programming Computation **1**(1) (2009) 1–41
17. Dash Optimization: Xpress-MP (2001)
18. Belotti, P.: Couenne: User manual. Published online at https://projects.coin-or.org/Couenne/ (2009) Accessed: 10/04/2015.
19. Sorensson, N., Een, N.: Minisat v1. 13-a sat solver with conflict-clause minimization. SAT **2005**(53) (2005) 1–2
20. Rall, L.B.: Automatic Differentiation: Techniques and Applications. Volume 120 of Lecture Notes in Computer Science. Springer, Berlin (1981)
21. Griewank, A., Juedes, D., Utke, J.: Algorithm 755: Adol-c: A package for the automatic differentiation of algorithms written in c/c++. ACM Trans. Math. Softw. **22**(2) (June 1996) 131–167
22. Rall, L.B., Corliss, G.F.: An introduction to automatic differentiation. Computational Differentiation: Techniques, Applications, and Tools **89** (1996)
23. Fournier, D.A., Skaug, H.J., Ancheta, J., Ianelli, J., Magnusson, A., Maunder, M.N., Nielsen, A., Sibert, J.: Ad model builder: using automatic differentiation for statistical inference of highly parameterized complex nonlinear models. Optimization Methods and Software **27**(2) (2012) 233–249
24. Bell, B.M.: CppAD: a package for C++ algorithmic differentiation. Computational Infrastructure for Operations Research (2012)
25. Ahn, S., Kaplan, G., Moll, B., Winberry, T., Wolf, C.: When inequality matters for macro and macro matters for inequality. NBER Working Paper No. w23494 https://ssrn.com/abstract=2984671 (June 2017)
26. U.K., R.C.: The hsl mathematical software library Published online at http://www.hsl.rl.ac.uk/.
27. Hijazi, H., Coffrin, C., Van Hentenryck, P.: Convex quadratic relaxations for mixed-integer nonlinear programs in power systems. Mathematical Programming Computation **9**(3) (2017) 321–367
28. The IEEE PES Task Force on Benchmarks for Validation of Emerging Power System Algorithms: PGLib Optimal Power Flow Benchmarks. Published online at https://github.com/power-grid-lib/pglib-opf Accessed: October 4, 2017.
29. Jabr, R.: Radial distribution load flow using conic programming. Power Systems, IEEE Transactions on **21**(3) (Aug 2006) 1458–1459

30. Vuffray, M., Misra, S., Lokhov, A., Chertkov, M.: Interaction screening: Efficient and sample-optimal learning of ising models. In: Advances in Neural Information Processing Systems. (2016) 2595–2603
31. Lokhov, A.Y., Vuffray, M., Misra, S., Chertkov, M.: Optimal structure and parameter learning of ising models. arXiv preprint arXiv:1612.05024 (2016)