COMPUTATIONAL PERFORMANCE OF ALGEBRAIC MODELING LANGUAGES WITH ALTERNATE SOLVER INTERFACES AND ADVANCED MODELING COMPONENTS

B. L. Ammari [a,1], S. Kompalli [a,1], M. Meraklı [b], Y. Qian [a], J. L. Pulsipher [a], M. Bynum [c], K. C. Furman [b], C. D. Laird [a,2]

[a] Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh, PA 15213
[b] ExxonMobil Upstream Research Company, Spring, TX 77389
[c] Center for Computing Research, Sandia National Laboratories, Albuquerque, NM

Abstract

Algebraic modeling languages (AMLs) have seen widespread success, enabling transparent formulation and rigorous solution of engineering, science, and business optimization problems. As we tackle larger, more complex problems, we often require specialized approaches and tailored meta-algorithms for efficient solution. AMLs implemented in high-level programming languages (e.g., Pyomo in Python, and JuMP in Julia) bring significant flexibility and promise rapid development of optimization applications. However, there are often concerns about performance of open-source AMLs. The JuMP team has long demonstrated comparable performance with compiled tools and recent development efforts in Pyomo have sought to shrink the performance gap. In this paper, we evaluate the performance of the open-source languages JuMP and Pyomo, and we demonstrate how to improve performance on large-scale case studies using alternate solver interfaces, advanced modeling components and new capabilities that support efficient "resolves" of problems with similar structure. The case studies selected for comparison include a modified facility location problem, a linear-quadratic control problem, a maritime inventory routing problem, and an unconstrained nonlinear optimization problem. For Pyomo, our computational studies show that a 35-40 % performance improvement is possible for linear problems by switching from the "direct" interface to the LP file-based interface. Furthermore, for repeated solution of problems with similar structure (but different parameter values), over an order of magnitude performance increase is possible with new solver interfaces in Pyomo.

Keywords

Algebraic Modeling Languages, Pyomo, JuMP

Introduction

Algebraic Modeling Languages (AMLs) have emerged as the popular choice of frameworks to represent models in applied mathematical optimization (Fragniere and Gondzio, 2002). With advancements in solution algorithms in the 1950's, the high cost and error-prone nature of directly interfacing with solvers necessitated development of computer programs to automate the process (Fourer, 2013), and the first AMLs were introduced as matrix generators for linear programming problems (Kallrath, 2004). In 1976, discussions on creation of a general algebraic modeling language ultimately led to the development of GAMS by the World Bank (Meeraus, 1976).

Subsequently, AML environments expanded significantly through the introduction of more commercial and open-source AMLs (Fourer, 2012). In addition to GAMS, AMPL was developed in 1985 to handle separate model and data files and included options to run batch operations (Chen et al., 1996; Fourer et al., 2003). Later, AIMMS was developed with a graphical user interface to assist engineers in coding mathematical programs and provide tools for enterprise deployment. In contrast to commercial AMLs, recent developments in high-level programming languages such as Python and Julia have led to several open-source AMLs. For example, PuLP is a Python package tailored for linear programming and mixed integer linear programming problems (Mitchell et al., 2011). GEKKO specializes in dynamic optimization for mixed-integer, nonlinear, and differential algebraic equations (DAE) problems (Beal et al., 2018). For general classes of optimization problems, Pyomo provides a comprehensive suite of capabilities in Python (Bynum et al., 2021). Likewise, JuMP has emerged as the primary optimization package in Julia (Dunning et al., 2017).

Open-source tools like Pyomo and JuMP are implemented in full-featured, high-level programming languages that provide significant flexibility for implementation of ad-

[1] Authors contributed equally to this work.
[2] Corresponding author. Email: claird@andrew.cmu.edu.

vanced modeling and solution approaches with access to a range of external libraries to support rapid development of end-to-end optimization applications. The primary concerns with open-source packages compared with their commercial counterparts is language stability and computational performance. JuMP is known to have performance comparable to compiled languages (Dunning et al., 2017), however, for Pyomo in particular, performance can be a concern, especially for the subset of problems where model generation time exceeds solver time (typically large LPs). While packages developed in C++ like Gravity may provide faster instance generation (Hijazi et al., 2018), the simplicity and syntax of high-level programming languages still makes tools like Pyomo and JuMP favorable alternatives.

Performance comparisons between AMLs do exist in the literature. Dunning et al. (2017) provide a comparison between Pyomo and JuMP, however further development of both packages since 2017 warrants new investigation, and there is a need to investigate performance differences across an array of available solver interfaces. More recently, Jusevičius et al. (2021) conducted computational experiments between AIMMS, AMPL, GAMS, Pyomo, and JuMP by utilizing models from the GAMS model library. They use automated tools to convert GAMS models into explicit, expanded forms of models compatible with the other AMLs - that is, all constraint equations are written explicitly as opposed to utilizing compact set notation. This expanded form may impact the performance of an AML and is not representative of how models are typically formulated in these languages. Furthermore, this comparison is performed with relatively small test problems where language overheads (e.g., package imports) may dominate the computational time.

Our work aims to provide several contributions over the existing literature. We compare the computational performance of Pyomo and JuMP, and show how the performance can be significantly improved through selection of solver interfaces and the use of advanced modeling components. We add additional test cases, including the maritime inventory routing problem and the multivariate Rosenbrock example. Furthermore, we investigate the performance differences when performing multiple optimizations with similarly structured models (showing that over an order of magnitude performance improvement is possible). This use case is very common in industrial applications where:

- multiple optimizations are required on a single model with changes in parameter values
- the application solution requires a meta-algorithm that may include parameter changes and activation or deactivation of constraints
- addition of new constraints is required between solves, usually through a cut-generation approach

In these applications, it is desirable for these "resolves" to be computationally efficient, avoiding the overhead of initial model construction for subsequent solves.

The remainder of this paper is structured as follows. The next section provides a description of the benchmark problems used in this study. We then present the numerical results along with methodology used to perform the computational experiments. We close the paper with a short discussion of the conclusions and directions for future work.

Benchmark Model Formulations

In this section, we describe the benchmark problems that are used in this study. This includes two mixed-integer programming problems, a quadratic programming problem, and a nonlinear optimization problem. We adopted the modified facility location problem and the linear-quadratic control problem formulations described in Dunning et al. (2017) with implementations from GitHub (https://github.com/orfusion/OptimizationModelComparisons, accessed 2022-05), and we included the multivariate Rosenbrock example from Dixon and Mills (1994). Each of these test cases have parameters that allow us to adjust the size of the problem and investigate scalability. Furthermore, we include the maritime inventory routing problem formulation presented by Papageorgiou et al. (2018) using four examples of group 2 instance data from MIRPLib (Papageorgiou et al., 2014). Each of these formulations is discussed in more detail below.

Facility Location Problem (FAC)

This formulation is a modification of the classic facility location problem proposed by Owen and Daskin (1998). Dunning et al. (2017) generate data by placing C customers evenly on a grid of size G x G with F number of facilities. Thus, the sets are defined as $c \in \{1,...,C\}$ and $f \in \{1,...,F\}$. The objective is to minimize the maximum distance between a customer and its nearest facility. The original formulation involves a second-order cone, which can be further translated into a quadratic form using additional constraints and auxiliary variables. Assuming that there is a customer located at each coordinate (i,j) where $i,j \in \{0,\frac{1}{G},...,1\}$ of a 2-dimensional unit square, the problem is described as,

$$
\begin{aligned}
\min_{d,r,s,y,z} \quad & d \\
\text{s.t.} \quad & \sum_{f=1}^{F} z_{i,j,f} = 1 && \forall\, i,j \\
& s_{i,j,f} = d + M(1 - z_{i,j,f}) && \forall\, i,j,f \\
& r_{i,j,f,1} = \frac{i}{G} - y_{f,1} && \forall\, i,j,f \\
& r_{i,j,f,2} = \frac{j}{G} - y_{f,2} && \forall\, i,j,f \\
& r_{i,j,f,1}^2 + r_{i,j,f,2}^2 \leq s_{i,j,f}^2 && \forall\, i,j,f \\
& z_{i,j,f} \in \{0,1\} && \forall\, i,j,f \\
& s_{i,j,f} \geq 0 && \forall\, i,j,f.
\end{aligned}
$$

Here, the big-M parameter is given as $M = \max_{c,c'} ||x_c - x_{c'}||_2$. As in Dunning et al. (2017), we consider the problem sizes of $F \in \{25,50,75,100\}$ with G=F.

Linear-Quadratic Control Problem (LQCP)

The linear quadratic control problem (LQCP) formulation is a simplified version of the formulation presented by Mittleman (2001). Here, m and n are parameters that allow

us to scale the number of variables and constraints. The sets are defined as $I = \{0,...,m\}$, $J = \{0,...,n\}$, $I' \leftarrow I\backslash\{m\}$, and $J' \leftarrow J\backslash\{0,n\}$. The problem formulation from Dunning et al. (2017) is shown below, and we consider the same instance sizes with m=n and $n \in \{500, 1000, 1500, 2000\}$.

$$\min_{u,y} \ \frac{1}{4}\Delta_x((y_{m,0}-y_0^t)^2 + 2\sum_{j=1}^{n-1}(y_{m,j}-y_j^t)^2 + (y_{m,n}-y_n^t)^2) + \frac{1}{4}\alpha\Delta_t(2\sum_{i=1}^{m-1}u_i^2 + u_m^2) \quad (1)$$

$$\text{s.t.} \ \frac{1}{\Delta_t}(y_{i+1,j}-y_{i,j}) = \frac{1}{2h_2}(y_{i,j-1}-2y_{i,j}+y_{i,j+1}) + \frac{1}{2h_2}(y_{i+1,j-1}-2y_{i+1,j}+y_{i+1,j+1}) \quad \forall\, i \in I', j \in J' \quad (2)$$

$$y_{0,j} = 0 \quad \forall\, j \in J \quad (3)$$

$$y_{i,2} - 4y_{i,1} + 3y_{i,0} = 0 \quad \forall\, i \in I \quad (4)$$

$$\frac{1}{2\Delta_x}(y_{i,n-2} - 4y_{i,n-1} + 3y_{i,n}) = u_i - y_{i,n} \quad \forall\, i \in I \quad (5)$$

$$-1 \le u_i \le 1 \quad \forall\, i \in I \quad (6)$$

$$0 \le y_{i,j} \le 1 \quad \forall\, i \in I, j \in J \quad (7)$$

Maritime Inventory Routing Problem (MIRP)

A single product maritime inventory routing problem (MIRP) is formulated using a discrete-time arc-flow MILP model involving T time periods as described in group 2 instances in Papageorgiou et al. (2014). This formulation contains port, vessel, and vessel classes as the main components. Ports can be classified either as loading ports J^P or consumption ports, J^C, and Δ_j is set to $+1$ or -1 respectively. Each port j has a fixed number of berths B_j which restricts the number of vessels that can load or discharge simultaneously and a non-constant per-period production or consumption rate $D_{j,t}$. There is an inventory of capacity $S_{j,t}^{max}$ for every port irrespective of the type of port. A simplified version of spot market has been included to sell excess inventory or buy product whenever necessary. Trading with spot market at port j in time period t incurs a penalty $P_{j,t}$ per unit volume such that $P_{j,t} > P_{j,t+1}$ to ensure that the spot market is used as late as possible.

Each vessel of vessel class vc has a capacity Q^{vc} and it is assumed that only direct deliveries are possible, i.e., vessels load and discharge fully at respective ports. Port inventory capacities are assumed to be always greater than the maximum vessel capacity $S_{j,t}^{max} > \max(Q^{vc} : vc \in VC)\ \forall j,t$. Loading and discharge operations are completed within the same time period that they started for each vessel at any port.

The formulation is based on a time-space network (Song and Furman, 2013), where each port-time period pair corresponds to a node and arcs between nodes represent vessel movements between ports and/or time periods. The network consists of a set of nodes $N_{0,T+1}$ and a set of directed arcs A. The set of nodes $N_{0,T+1}$ consists of N regular nodes (port-time pairs) as well as a source node n_0 and a sink node n_{T+1}. The set of arcs available to vessel class vc is denoted as A^{vc}. In general arcs can be divided into four categories: source arcs, travel arcs, waiting arcs and sink arcs. $A^{vc,inter} \subset A^{vc}$ consists of set of travel and sink arcs associated with a particular vessel class. A set of incoming RS_n^{vc}, outgoing arcs FS_n^{vc} and incoming inter-regional arcs $RS_n^{vc,inter}$ are defined with respect to a node and a vessel class.

Decision variable x_a^{vc} denotes the number of vessels of vessel class vc travelling on arc $a \in A^{vc}$, $s_{j,t}$ represents the inventory at port j at the end of time period t, and $\alpha_{j,t}$ denotes the amount bought from or sold to the spot market at port j in time period t. The formulation for group 2 instances described in Papageorgiou et al. (2014) is

$$\min \sum_{vc\in VC}\sum_{a\in A^{vc}} C_a^{vc}x_a^{vc} + \sum_{j\in J}\sum_{t\in T} P_{j,t}\alpha_{j,t}$$

$$\text{s.t.} \sum_{a\in FS_n^{vc}} x_a^{vc} - \sum_{a\in RS_n^{vc}} x_a^{vc} = \begin{cases} +1 & if\ n = n_0 \\ -1 & if\ n = n_{T+1} \\ 0 & if\ n \in N \end{cases} \quad \forall\, n \in N_{0,T+1}, vc \in VC$$

$$s_{j,t} = s_{j,t-1} + \Delta_j(D_{j,t} - \alpha_{j,t} - \sum_{vc\in VC}\sum_{a\in FS_n^{vc,inter}} Q^{vc}x_a^{vc}) \quad \forall\, n = (j,t) \in N$$

$$\sum_{vc\in VC}\sum_{a\in FS_n^{vc,inter}} x_a^{vc} \le B_j \quad \forall\, n = (j,t) \in N$$

$$\alpha_{j,t} \ge 0 \quad \forall\, n = (j,t) \in N$$

$$s_{j,t} \in [0, S_{j,t}^{max}] \quad \forall\, n = (j,t) \in N$$

$$x_a^{vc} \in \{0,1\} \quad \forall\, vc \in VC, a \in A^{vc,inter}$$

$$x_a^{vc} \in Z_+ \quad \forall\, vc \in VC, a \in A^{vc}\backslash A^{vc,inter}$$

For the benchmark studies, we selected four examples from group 2 instances of MIRPLib (Papageorgiou et al., 2014) (as shown in Table 1). We wrote a parser to read the individual data files and create the necessary sets and parameters for constructing the model in Pyomo and JuMP.

Multivariable Rosenbrock Problem (ROSEN)

The classic two-variable Rosenbrock function has been used in many unconstrained nonlinear optimization examples. To provide a scalable nonlinear example with relatively large expression trees, we make use of the multivariable *extended Rosenbrock function* shown below (Dixon and Mills, 1994).

$$\min_x \sum_{i=1}^{N/2} 100(x_{2i-1}^2 - x_{2i})^2 + (x_{2i-1}-1)^2$$

For the case studies in this paper, we select the sizes N=$\{100000, 500000, 1000000\}$.

Methods and Numerical Results

Using the benchmarks described above, we evaluate the performance of Pyomo and JuMP on several case studies. All the computational studies were performed on a Linux server running Ubuntu with 1TB of RAM and 4 Intel(R) Xeon(R) Gold 6234 CPUs (3.30GHz) with 8 cores each. The key packages were Python 3.9.12, Pyomo 6.4.1, Julia 1.7.3, and

JuMP v1.1.1. All computational case studies were performed in serial.

Single Model Generation and Translation: In the first computational study, we evaluate the performance of Pyomo and JuMP when generating a single model instance and translating it appropriately for the required solver interface. Our focus is on the time spent in the modeling language itself (not the solver). For each run, we record a wall-clock time that includes model generation, translation, and solver time, and then subtract the reported solver time to determine the time spent in the AML only. This is done since the AMLs do not provide an easy way to separate the translation step from the solve step. Also note that we set a timeout on the solver to reduce the overall execution time for the analysis. For Pyomo, we show the timing for three different solver interfaces: **Gurobi (LP)** utilizes an interface that produces an `.lp` file from the Pyomo model and calls Gurobi (Gurobi Optimization, LLC, 2022) as an executable; **Gurobi (Py)** (also known as Gurobi direct) uses a Pyomo interface based on the Gurobi Python API; and **Gurobi (APPSI)** uses a newer set of solver interfaces (called APPSI) recently developed in Pyomo that are focused on efficient resolves of similar models. For JuMP we consider two different solver interfaces. **Default** refers to the default or "natural" solver interface. With the **Direct** interface in JuMP, memory allocation can be significantly reduced by building models initialized via `direct_model` which, similar to APPSI, sets up a specialized solver backend for solvers that support incremental creation/modification (eliminating the use of the more general `CachingOptimizer` backend that enables incremental builds/modifications with any solver) (Legat et al., 2022).

Timing results for model generation and translation across the different AMLs and solver interfaces are shown in Table 1. For JuMP, we excluded any just-in-time (JIT) compilation time by performing the model construction and optimization twice, and timing the second run only. Therefore, the JIT compilation is invoked on the first set of calls and this precompiled code is used for the second call. Julia also provides a mechanism for storing and loading an image to avoid overheads, however, that approach was not used here. Immediately, we notice that JuMP is approximately an order of magnitude faster than Pyomo - an observation that is consistent with other published timing results. However, we also note that we can obtain approximately a 35-40% performance improvement in Pyomo by selecting the Gurobi (LP) interface over the Gurobi (Py) interface. Likewise, performance improvements of approximately 10-40% are possible with JuMP by utilizing the "Direct" interface.

Advanced Modeling Components: For the FAC-LE and the LQCP-LE cases in Table 1 we show performance improvements that are possible with advanced modeling components. In particular, for the facility location and linear quadratic control problems, we identified the constraints that were the most time-consuming to construct. We know that these constraints are linear, and Pyomo supports creation of `LinearExpression` objects directly by specifying lists of variables and coefficients. For the LQCP case study, updating the construction rules for equation 2 with `LinearExpression` objects improved performance significantly (approximately an additional 45%) with the Gurobi (LP) solver interface. It should be noted, however, that this approach requires manual reformulation in matrix notation and removes many of the benefits of an AML. The JuMP models already specify that their constraints are linear and no additional computational experiments were necessary.

End-to-End Computational Performance: Table 2 shows results for end-to-end computational time that includes language overheads and just-in-time (JIT) compilation time. For these studies, a single script was executed for each test problem, and the wall-clock time was reported for complete execution of the script. Then we subtracted the time spent in the solver. Comparing the times in Table 1 with those in Table 2, we can see the difference caused by overhead. For smaller test cases, the JIT overhead for JuMP is substantial, however, this overhead is a small fraction of the overall time for larger test cases. For Pyomo, the overhead for small cases is minimal, however, for larger test cases, the overhead increased, likely due to memory cleanup. Note that the maritime inventory routing problem was not included in this analysis since it involved additional code to parse the data files that could lead to timing differences unrelated to the modeling language.

Nonlinear Rosenbrock Model: We performed timing on the multivariable Rosenbrock example considering both the model generation and translation time, as well as the end-to-end timing that includes overheads. The solver used in this case was IPOPT where Pyomo was using an `.nl` file-based interface and JuMP was using a direct solver interface. The Pyomo time without overheads ranged from 3.7 to 43.8 seconds for the smallest and largest test cases respectively, while the JuMP computational times ranged from 0.1 to 1.2 seconds respectively. To save space, the complete table of results for this nonlinear case study are not shown, but are consistent with the linear case studies discussed above.

Repeated solves with similar models: There are many applications that require repeated solution of problems with similar structure, including nonlinear model predictive control where the structure is the same, but some data changes from run to run. Implementation of meta-algorithms and decomposition strategies can require repeated solution with different parameter values, constraints activated/deactivated, or the addition of cuts at each iteration. Recently, the developers of Pyomo have implemented a new set of solver interfaces called APPSI. These interfaces support efficient in-memory modification and resolve of models. Here, we demonstrate this Pyomo feature on the facility location problem. After completing an initial solve, we change the value of the big-M parameter and solve the problem again. This is accomplished by specifying that the big-M parameter is `mutable` to provide an indicator to Pyomo that the value may change between calls to the solver. The timing results reflect the model update and translation time for the "resolve". Table 3 shows the timing results for the three solver interfaces investigated with Pyomo. With the APPSI interface, significant performance improvements are possible for this use case, with over an order of magnitude reduction in computational time compared with the best result for the single solve case. We omit-

ted JuMP from this study as it does not support mutable parameters in linear/quadratic expressions without an extension packages (e.g., ParametricOptInterface).

Conclusions and Future Work

In this work, we compared the performance of two open-source modeling languages, Pyomo and JuMP, and investigated the performance improvements that are possible using different solver interfaces and advanced modeling components. We also investigated the performance benefits that could be gained with the common use case of repeated solution of similar problems with different data. In general, JuMP was approximately an order of magnitude faster than Pyomo for single model generation and translation. While JuMP has higher overheads on end-to-end timing comparisons due to the just-in-time compilation steps, this overhead was negligible on larger case studies and can be reduced by saving and loading already compiled images. The end-to-end overhead with Pyomo was minimal for smaller test cases, but actually did scale with problem size.

Significant performance improvements were identified, however. For Pyomo, the Gurobi (LP) interface was approximately 30-40% faster than the Gurobi (Py) solver interfaces. An additional improvement of approximately 45% was possible for the linear quadratic control problem using the `LinearExpression` object directly for the large PDE constraint. Furthermore, major performance benefits are possible when resolving models with different data using Pyomo's new APPSI solver interfaces, resulting in over an order of magnitude reduction in computational time for this use case. Future work will focus on expanding the case studies to include modification of decomposition approaches (e.g., deactivation/activation of constraints, and addition of cuts) and further investigation of extension packages in JuMP.

Acknowledgements: Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

References

Beal, L., D. Hill, R. Martin, and J. Hedengren (2018). Gekko optimization suite. *Processes 6*, 106.

Bynum, M. L., G. A. Hackebeil, W. E. Hart, C. D. Laird, B. L. Nicholson, J. D. Siirola, J.-P. Watson, and D. L. Woodruff (2021). *Pyomo — Optimization Modeling in Python* (3 ed.), Volume 67. Springer International Publishing.

Chen, X., K. S. Rao, J. Yu, and R. W. Pike (1996). Comparison of GAMS, AMPL, and MINOS for optimization. *Chemical Engineering Education 30*(3), 220–227.

Dixon, L. and D. Mills (1994). Effect of rounding errors on the variable metric method. *Journal of Optimization Theory and Applications 80*(1), 175–179.

Dunning, I., J. Huchette, and M. Lubin (2017). Jump: A modeling language for mathematical optimization. *SIAM Review 59*, 295–320.

Fourer, R. (2012). On the evolution of optimization modeling systems. *Optimization Stories, Documenta Mathematica, Extra Volume ISMP (2012)*, 377–388.

Fourer, R. (2013). Algebraic modeling languages for optimization. In *Encyclopedia of Operations Research and Management Science*, pp. 43–51. Springer US.

Fourer, R., D. M. Gay, and B. W. Kernighan (2003). *AMPL A Modeling Language for Mathematical Programming* (2 ed.). Thomson.

Fragniere, E. and J. Gondzio (2002). Optimization modeling languages. In *Handbook of Applied Optimization*, pp. 993–1007. Citeseer.

Gurobi Optimization, LLC (2022). Gurobi Optimizer Reference Manual.

Hijazi, H., G. Wang, and C. Coffrin (2018). Gravity: A modeling language for mathematical optimization and machine learning. In *The Thirty-second Annual Conference on Neural Information Processing Systems (NeurIPS)*.

Jusevičius, V., R. Oberdieck, and R. Paulavičius (2021). Experimental analysis of algebraic modelling languages for mathematical optimization. *Informatica (Netherlands) 32*, 283–304.

Kallrath, J. (2004). *Modeling Languages in Mathematical Optimization*, Volume 88. Springer US.

Legat, B., O. Dowson, J. D. Garcia, and M. Lubin (2022). Mathoptinterface: a data structure for mathematical optimization problems. *INFORMS Journal on Computing 34*(2), 672–689.

Meeraus, A. (1976). Toward a general algebraic modelling system. In *IX. International Symposium on Mathematical Programming, Budapest, Hungary, August 23–27*, Volume 185.

Mitchell, S., M. O'Sullivan, and I. Dunning (2011). Pulp: a linear programming toolkit for python. Technical report, The University of Auckland, Auckland, New Zealand.

Mittleman, H. D. (2001). Sufficient optimality for discretized parabolic and elliptic control problems. In K.-H. Hoffmann, R. H. W. Hoppe, and V. Schulz (Eds.), *Fast Solution of Discretized Optimization Problems*, pp. 184–196. Springer International Publishing.

Owen, S. H. and M. S. Daskin (1998). Strategic facility location: A review. *European Journal of Operational Research 111*, 423–447.

Papageorgiou, D. J., M.-S. Cheon, S. Harwood, F. Trespalacios, and G. L. Nemhauser (2018). Recent progress using matheuristics for strategic maritime inventory routing. In C. Konstantopoulos and G. Pantziou (Eds.), *Modeling, Computing and Data Handling Methodologies for Maritime Transportation*, Volume 131, pp. 59–94. Springer International Publishing.

Papageorgiou, D. J., G. L. Nemhauser, J. Sokol, M.-S. Cheon, and A. B. Keha (2014). Mirplib – a library of maritime inventory routing problem instances: Survey, core model, and benchmark results. *European Journal of Operational Research 235*, 350–366.

Song, J.-H. and K. C. Furman (2013). A maritime inventory routing problem: Practical approach. *Computers & Operations Research 40*(3), 657–665.

Table 1: Model generation and translation time (reported in seconds). Timing does not include the time spent in the solver or the overhead associated with launching the script and any just-in-time compilation.

	Pyomo			JuMP[2]	
Model & Size	Gurobi (LP)	Gurobi (Py)	Gurobi (APPSI)	*Default*	Direct
FAC-25	4.2	7.3	7.2	0.5	0.4
FAC-50	33.3	57.5	57.6	4.2	2.7
FAC-75	111.6	196.6	191.8	14.0	10.7
FAC-100	264.2	454.8	459.2	33.9	23.8
LQCP-500	23.4	36.1	35.0	1.6	1.3
LQCP-1000	94.6	144.6	140.8	8.4	5.3
LQCP-1500	214.0	326.9	309.7	18.5	11.8
LQCP-2000	378.9	598.2	561.6	33.0	21.3
MIRP-LR1_DR08_VC05_V40a_360	3.8	5.5	5.4	1.3	1.0
MIRP-LR1_DR08_VC10_V40b_360	8.0	10.6	10.4	2.7	1.8
MIRP-LR1_DR12_VC05_V70a_360	5.9	8.6	7.6	1.9	1.5
MIRP-LR1_DR12_VC10_V70a_360	11.7	15.6	15.1	3.3	3.3
FAC-LE-25	3.9	6.7	6.7	–	–
FAC-LE-50	28.8	55.3	51.0	–	–
FAC-LE-75	105.1	183.6	167.3	–	–
FAC-LE-100	246.7	441.8	403.9	–	–
LQCP-LE-500	13.1	21.8	25.6	–	–
LQCP-LE-1000	49.7	86.5	103.0	–	–
LQCP-LE-1500	119.6	202.1	237.2	–	–
LQCP-LE-2000	215.1	348.3	419.6	–	–

[2] For JuMP, the model is solved once to invoke any just-in-time compilation, and then the reported time is the time for for the second call to build and solve the model.

Table 2: End-to-end wall-clock time (reported in seconds). Values include the overhead associated with launching the script from the command line and any just-in-time compilation, but not the time spent in the solver.

	Pyomo			JuMP	
Model & Size	Gurobi (LP)	Gurobi (Py)	Gurobi (APPSI)	*Default*	Direct
FAC-25	4.9	8.0	8.0	19.3	12.4
FAC-50	36.0	60.3	60.7	21.8	14.8
FAC-75	119.8	204.8	202.0	32.0	21.0
FAC-100	282.9	476.4	482.7	51.3	34.8
LQCP-500	25.7	38.6	37.5	18.4	14.4
LQCP-1000	102.6	153.5	149.9	25.4	18.0
LQCP-1500	231.7	344.5	329.8	35.7	24.6
LQCP-2000	409.8	629.8	595.0	51.7	32.4
FAC-LE-25	4.5	7.3	7.7	–	–
FAC-LE-50	31.2	57.5	54.3	–	–
FAC-LE-75	113.3	191.5	176.4	–	–
FAC-LE-100	262.8	459.7	425.2	–	–
LQCP-LE-500	14.1	22.7	27.5	–	–
LQCP-LE-1000	52.4	89.5	110.1	–	–
LQCP-LE-1500	125.2	208.5	252.7	–	–
LQCP-LE-2000	225.2	358.2	445.2	–	–

Table 3: Computational time for "resolve" with different parameter values (reported in seconds)

	Pyomo		
Model & Size	Gurobi (LP)	Gurobi (Py)	Gurobi (APPSI)
FAC-25	2.6	5.5	**0.3**
FAC-50	20.0	45.0	**2.0**
FAC-75	68.4	147.6	**6.8**
FAC-100	163.6	351.5	**16.6**