Computing in Operations Research using Julia

Miles Lubin, Iain Dunning
MIT Operations Research Center, 77 Massachusetts Avenue, Cambridge, MA USA
mlubin@mit.edu, idunning@mit.edu

The state of numerical computing is currently characterized by a divide between highly efficient yet typically cumbersome low-level languages such as C, C++, and Fortran and highly expressive yet typically slow high-level languages such as Python and MATLAB. This paper explores how Julia, a modern programming language for numerical computing which claims to bridge this divide by incorporating recent advances in language and compiler design (such as just-in-time compilation), can be used for implementing software and algorithms fundamental to the field of operations research, with a focus on mathematical optimization. In particular, we demonstrate algebraic modeling for linear and nonlinear optimization and a partial implementation of a practical simplex code. Extensive cross-language benchmarks suggest that Julia is capable of obtaining state-of-the-art performance.

Key words: algebraic modeling; scientific computing; programming languages; metaprogramming; domain-specific languages

1. Introduction

Operations research and digital computing have grown hand-in-hand over the last 60 years, with historically large amounts of available computing power being dedicated to the solution of linear programs (Bixby 2002). Linear programming is one of the key tools in the operations research toolbox and concerns the problem of selecting variable values to maximize a linear function subject to a set of linear constraints. This foundational problem, the algorithms to solve it, and its extensions form a large part of operations research-related computation. The purpose of this paper is to explore modern advances in programming languages that will affect how algorithms for operations research computation are implemented, and we will use linear and nonlinear programming as motivating cases.

The primary languages of high-performance computing have been Fortran, C, and C++ for a multitude of reasons, including their interoperability, their ability to compile to highly efficient machine code, and their sufficient level of abstraction over programming in an assembly language. These languages are compiled offline and have strict variable typing, allowing advanced optimizations of the code to be made by the compiler.

A second class of more modern languages has arisen that is also popular for scientific computing. These languages are typically interpreted languages that are highly expressive but do not match the speed of lower-level languages in most tasks. They make up for this by focusing on "glue code" that links together, or provides wrappers around, high-performance code written in C and Fortran. Examples of languages of this type would be Python (especially with the Numpy (van der Walt et al. 2011) package), R, and MATLAB. Besides being interpreted rather than statically compiled, these languages are slower for a variety of additional reasons, including the lack of strict variable typing.

Just-in-time (JIT) compilation has emerged as a way to have the expressiveness of modern scripting languages and the performance of lower-level languages such as C. JIT compilers attempt to compile at run-time by inferring information not explicitly stated by the programmer and use these inferences to optimize the machine code that is produced. Attempts to retrofit this functionality to the languages mentioned above has had mixed success due to issues with language design conflicting with the ability of the JIT compiler to make these inferences and problems with the compatibility of the JIT functionality with the wider package ecosystems.

Julia (Bezanson et al. 2012) is a new programming language that is designed to address these issues. The language is designed from the ground-up to be both expressive and to enable the LLVM-based JIT compiler (Lattner and Adve 2004) to generate efficient code. In benchmarks reported by its authors, Julia performed within a factor of two of C on a set of common basic tasks. The contributions of this paper are two-fold: firstly, we develop publicly available codes to demonstrate the technical features of Julia which greatly facilitate the implementation of optimization-related tools. Secondly, we will confirm that the aforementioned performance results hold for realistic problems of interest to the field of operations research.

This paper is not a tutorial. We encourage interested readers to view the language documentation at `julialang.org`. An introduction to Julia's syntax will not be provided, although the examples of code presented should be comprehensible to readers with a background in programming. The source code for all of the experiments in the paper is available in the online supplement[1]. *JuMP*, a library developed by the authors for mixed-integer

[1] `http://www.mit.edu/~mlubin/juliasupplement.tar.gz`

algebraic modeling, is available directly through the Julia package manager, together with community-developed low-level interfaces to both Gurobi and the COIN-OR solvers Cbc and Clp for mixed-integer and linear optimization, respectively.

The rest of the paper is organized as follows. In Section 2, we present the package JuMP. In Section 3, we explore nonlinear extensions. In Section 4, we evaluate the suitability of Julia for low-level implementation of numerical optimization algorithms by examining its performance on a realistic partial implementation of the simplex algorithm for linear programming.

2. JuMP

Algebraic Modeling Languages (AMLs) are an essential component in any operations researcher's toolbox. AMLs enable researchers and programmers to describe optimization models in a natural way, by which we mean that the description of the model in code resembles the mathematical statement of the model. AMLs are particular examples of domain-specific languages (DSLs) which are used throughout the fields of science and engineering.

One of the most well-known AMLs is AMPL (Fourer et al. 1993), a commercial tool that is both fast and expressive. This speed comes at a cost: AMPL is not a fully-fledged modern programming language, which makes it a less than ideal choice for manipulating data to create the model, for working with the results of an optimization, and for linking optimization into a larger project.

Interpreted languages such as Python and MATLAB have become popular with researchers and practitioners alike due to their expressiveness, package ecosystems, and acceptable speeds. Packages for these languages that add AML functionality such as YALMIP (Lofberg 2004) for MATLAB and PuLP (Mitchell et al. 2011) and Pyomo (Hart et al. 2011) for Python address the general-purpose-computing issues of AMPL but sacrifice speed. These AMLs take a nontrivial amount of time to build the sparse representation of the model in memory, which is especially noticeable if models are being rebuilt a large number of times, which arises in the development of models and in practice, e.g. in simulations of decision processes. They achieve a similar "look" to AMPL by utilizing the *operator overloading* functionality in their respective languages, which introduces significant overhead and inefficient memory usage. Interfaces in C++ based on operator overloading, such

as those provided by the commercial solvers Gurobi and CPLEX, are often significantly faster than AMLs in interpreted languages, although they sacrifice ease of use and solver independence.

We propose a new AML, *JuMP* (Julia for Mathematical Programming), implemented and released as a Julia package, that combines the speed of commercial products with the benefits of remaining within a fully-functional high-level modern language. We achieve this by using Julia's *metaprogramming* features to turn natural mathematical expressions into sparse internal representations of the model without using operator overloading. In this way we achieve performance comparable to AMPL and an order-of-magnitude faster than other embedded AMLs.

2.1. Metaprogramming with macros

Julia is a *homoiconic* language: it can represent its own code as a data structure of the language itself. This feature is also found in languages such as Lisp. To make the concept of metaprogramming more clear, consider the following Julia code snippet:

```
1 macro m(ex)
2     ex.args[1] = :(-)  # Replace operation with subtraction
3     return esc(ex)     # Escape expression (see below)
4 end
5 x = 2; y = 5  # Initialize variables
6 2x + y^x      # Prints 29
7 @m(2x + y^x)  # Prints -21
```

On lines 1-4 we define the *macro* `m`. Macros are compile-time source transformation functions, similar in concept to the preprocessing features of C but operating at the syntactic level instead of performing textual substitution. When the macro is invoked on line 7 with the expression $2x + y^x$, the value of `ex` is a Julia object which contains a representation of the expression as a tree, which we can compactly express in Polish (prefix) notation as:

$$(+, (*, 2, x), (^\wedge, y, x))$$

Line 2 replaces the $+$ in the above expression with $-$, where `:(-)` is Julia's syntax for the symbol $-$. Line 3 returns the *escaped* output, indicating that the expression refers to variables in the surrounding scope. Hence, the output of the macro is the expression $2x - y^x$, which is subsequently compiled and finally evaluated to the value -21.

Macros provide powerful functionality to efficiently perform arbitrary transformation of expressions. The complete language features of Julia are available within macros, unlike the

limited syntax for macros in C. Additionally, macros are evaluated only once, at compile time, and so have no runtime overhead, unlike `eval` functions in MATLAB and Python. (Note: with JIT compilation, "compile time" in fact occurs during the program's execution, e.g., the first time a function is called.) We will use macros as a basis for both linear and nonlinear modeling.

2.2. Language Design

While we did not set out to design a full modeling language with the wide variety of options as AMPL, we have sufficient functionality to model any linear optimization problem with a very similar number of lines of code. Consider the following simple AMPL model of a "knapsack" problem (we will assume the data are provided before the following lines):

```
var x{j in 1..N} >= 0.0, <= 1.0;

maximize Obj:
  sum {j in 1..N} profit[j] * x[j];

subject to CapacityCon:
  sum {j in 1..N} weight[j] * x[j] <= capacity;
```

The previous model would be written in Julia using JuMP with the following code:

```
m = Model(:Max)

@defVar(m, 0 <= x[1:N] <= 1)

@setObjective(m, sum{ profit[j] * x[j], j = 1:N })

@addConstraint(m, sum{ weight[j] * x[j], j = 1:N } <= capacity)
```

The syntax is mostly self-explanatory and is not the focus of this paper, but we draw attention to the similarities between the syntax of our Julia AML and existing AMLs. In particular, macros permit us to define new syntax such as `sum{}`, which is not part of the Julia language.

2.3. Building Expressions

The model is stored internally as a set of rows until it is completely specified. Each row is defined by two arrays: the first array is the indices of the columns that appear in this row and the second contains the corresponding coefficients. This representation is essentially the best possible while the model is being built and can be converted to a sparse column-wise format with relative efficiency. The challenge then is to convert the user's statement of

the problem into this sparse row representation as quickly as possible, while not requiring the user to express rows in a way that loses the readability that is expected from AMLs.

AMLs like PuLP achieve this with operator overloading. By defining new types to represent variables, new definitions are provided for the basic mathematical operators when one or both the operands is a variable. The expression is then built by combining subexpressions together until the full expression is obtained. This typically leads to an excessive number of intermediate memory allocations. One of the advantages of AMPL is that, as a purpose-built tool, it has the ability to statically analyze the expression to determine the storage required for its final representation. One way it may achieve this is by doing an initial pass to determine the size of the arrays to allocate, and then a second pass to store the correct coefficients. Our goal with Julia was to use the metaprogramming features to achieve a similar effect and bypass the need for operator overloading.

2.4. Metaprogramming implementation

Our solution is similar to what is possible with AMPL and does not rely on operator overloading at all. Consider the knapsack constraint provided in the example above. We will change the constraint into an equality constraint by adding a slack variable to make the expression more complex than a single sum. The `addConstraint` macro converts the expression

```
@addConstraint(m, sum{weight[j]*x[j], j=1:N} + s == capacity)
```

into the following code, transparently to the user:

```
aff = AffExpr()

sumlen = length(1:N)
sizehint!(aff.vars, sumlen)
sizehint!(aff.coeffs, sumlen)
for i = 1:N
    addToExpression(aff, 1.0*weight[i], x[i])
end

addToExpression(aff, 1.0, s)

addToExpression(aff, -1.0, capacity)

addConstraint(m, Constraint(aff,"=="))
```

The macro breaks the expression into parts and then stitches them back together as in our desired data structure. `AffExpr` represents the custom type that contains the variable

indices (`vars`) and coefficients (`coeffs`). In the first segment of code the macro pulls the indexing scheme from out of the sum and determines how long an array is required. Sufficient space to store the sum is reserved in one pass using the built-in function `sizehint!` before `addToExpression` (defined elsewhere) is used to fill it out. We use *multiple dispatch* to let Julia decide what type of object `x[i]` is, either a constant or a variable placeholder, using its efficient built-in type inference mechanism. After the sum is handled, the single slack variable s is appended and finally the right-hand-side of the constraint is set. Note the invocation of `addToExpression` with different argument types in the last usage - this time two constants instead of a constant and a variable. The last step is to construct the `Constraint` object that is essentially a wrapper around the expression and the sense. The function `addConstraint` is defined separately from the macro with the same name. We note that our implementation is not as efficient as AMPL's can be; space for the coefficients of single varibles like s is not preallocated, and so additional memory allocations are required; however, we still avoid the creation of many small temporary objects that would be produced with operator overloading.

2.5. Benchmarks

Different languages produce the final internal representation of the problem at different stages, making pure in-memory "model construction" time difficult to isolate. Our approach was to force all the AMLs to output the resulting model in the LP and/or MPS file formats and record the total time from executing the script until the file is output. We evaluated the performance of Julia relative to other AMLs by implementing two models whose size can be controlled by varying a parameter. Experiments were performed on a Linux system with an Intel Xeon E5-2650 processor.

1. P-median: this model was used by Hart et al. (2011) to compare Pyomo with AMPL. The model determines the location of M facilities over L possible locations to minimize the distance between each of N customers and the closest facility. C_i is a vector of customer locations that we generate randomly. In our benchmarks we fixed $M = 100$ and $N = 100$, and varied L. The results are in Table 1, and show that JuMP is safely within a factor of two of the speed AMPL, comparable in speed to, if not occasionally faster than, Gurobi's C++ modeling interface, and an order of magnitude faster than the Python-based modeling languages. Note that JuMP does not need to transpose the naturally row-wise data when outputting in LP format, which explains the observed difference in execution times. In

JuMP and AMPL, model construction was observed to consume 20% to 50% of the total time for this model.

$$\begin{aligned}
&\min \sum_{i=1}^{N}\sum_{j=1}^{L} |C_i - j|\, x_{ij} \\
&\text{s.t. } x_{ij} \le y_j \quad i = 1,\ldots,N, \quad ,j = 1,\ldots,L \\
&\qquad \sum_{j=1}^{L} x_{ij} = 1 \quad i = 1,\ldots,N \\
&\qquad \sum_{j=1}^{L} y_j = M
\end{aligned}$$

2. Linear-Quadratic control problem (*cont5_2_1*): this quadratic programming model is part of the collection maintained by Hans Mittleman (Mittelmann 2013). Not all the compared AMLs support quadratic objectives, and the quadratic objective sections of the file format specifications are ill-defined, so the objective was dropped and set to zero. The results in Table 2 mirror the general pattern of results observed in the p-median model.

$$\begin{aligned}
&\min_{y_{i,j},u_i} \ldots \\
&\text{s.t. } \frac{y_{i+1,j} - y_{i,j}}{\Delta t} = \frac{1}{2\left(\Delta x\right)^2}\left(y_{i,j-1} - 2y_{i,j} + y_{i,j+1} + y_{i+1,j-1} - 2y_{i+1,j} + y_{i+1,j+1}\right) \\
&\qquad\qquad i = 0,\ldots M-1,\ j = 1,\ldots,N-1 \\
&\quad y_{i,2} - 4y_{i,1} + 3y_{i,0} = 0 \qquad i = 1,\ldots,M \\
&\quad y_{i,n-2} - 4y_{i,n-1} + 3y_{i,n} = (2\Delta x)(u_i - y_{i,n}) \qquad i = 1,\ldots,M \\
&\quad -1 \le u_i \le 1 \qquad i = 1,\ldots,M \\
&\quad y_{0,j} = 0 \qquad j = 0,\ldots,N \\
&\quad 0 \le y_{i,j} \le 1 \qquad i = 1,\ldots,m,\ j = 0,\ldots,N \\
&\text{where } g_j = \frac{1}{2}\left(1 - (j\Delta x)^2\right)
\end{aligned}$$

2.6. Availability

JuMP (`https://github.com/IainNZ/JuMP.jl`) has been released with documentation as a Julia package. It remains under active development. We do not presently recommend its use in production environments. It currently interfaces with both Gurobi and the COIN-OR solvers Cbc and Clp for mixed-integer and linear optimization. Linux, OS X, and Windows platforms are supported.

Table 1 **P-median benchmark results. L is the number of locations. Total time (in seconds) to process the model definition and produce the output file in LP and MPS formats (as available).**

	JuMP/Julia		AMPL	Gurobi/C++		Pulp/PyPy		Pyomo
L	LP	MPS	MPS	LP	MPS	LP	MPS	LP
1,000	0.5	1.0	0.7	0.8	0.8	5.5	4.8	10.7
5,000	2.3	4.2	3.3	3.6	3.9	26.4	23.2	54.6
10,000	5.0	8.9	6.7	7.3	8.3	53.5	50.6	110.0
50,000	27.9	48.3	35.0	37.2	39.3	224.1	225.6	583.7

Table 2 **Linear-quadratic control benchmark results. N=M is the grid size. Total time (in seconds) to process the model definition and produce the output file in LP and MPS formats (as available).**

	JuMP/Julia		AMPL	Gurobi/C++		Pulp/PyPy		Pyomo
N	LP	MPS	MPS	LP	MPS	LP	MPS	LP
250	0.5	0.9	0.8	1.2	1.1	8.3	7.2	13.3
500	2.0	3.6	3.0	4.5	4.4	27.6	24.4	53.4
750	5.0	8.4	6.7	10.2	10.1	61.0	54.5	121.0
1,000	9.2	15.5	11.6	17.6	17.3	108.2	97.5	214.7

3. Nonlinear Modeling

Commercial AMLs such as AMPL and GAMS (Brooke et al. 1999) are widely used for specifying large-scale nonlinear optimization problems, that is, problems of the form

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{subject to} \quad & g_i(x) \leq 0 \quad i = 1, \ldots, m, \end{aligned} \tag{1}$$

where f and g_i are given by closed-form expressions.

Similar to the case of modeling linear optimization problems, open-source AMLs exist and provide comparable, if not superior, functionality; however, they may be significantly slower to build the model, even impractically slower on some large-scale problems. The user guide of CVX, an award-winning open-source AML for convex optimization built on

MATLAB, states that "CVX is *not* meant for very large problems" (Grant and Boyd 2013). This statement refers to two cases which are important to distinguish:

- An appropriate solver is available for the problem, but the time to build the model in memory and pass it to the solver is a bottleneck. In this case, users are directed to use a low-level interface to the solver in place of the AML.
- The problem specified is simply too difficult for available solvers, whether in terms of memory use or computation time. In this case, users are directed to consider reformulating the problem or implementing specialized algorithms for it.

Our focus in this section is on the *first* case, which is somehow typical of the large-scale performance of open-source AMLs implemented in high-level languages, as will be demonstrated in the numerical experiments in this section.

This performance gap between commercial and open-source AMLs can be partially explained by considering the possibly very different motivations of their respective authors; however, we posit that there is a more technical reason. In languages such as MATLAB and Python there is no programmatic access to the language's highly optimized expression parser. Instead, to handle nonlinear expressions such as `y*sin(x)`, one must either overload both the multiplication operator and the `sin` function, which leads to the expensive creation of many temporary objects as previously discussed, or manually parse the expression as a string, which itself may be slow and breaks the connection with the surrounding language. YALMIP and Pyomo implement the operator overloading approach, while CVX implements the string-parsing approach.

Julia, on the other hand, provides first-class access to its expression parser through its previously discussed metaprogramming features, which facilitates the generation of resulting code with performance comparable to that of commercial AMLs. In Section 3.1 we describe our proof-of-concept implementation, followed by computational results in Section 3.2.

3.1. Implementation in Julia

Whereas linear expressions are represented as sparse vectors of nonzero values, nonlinear expressions are represented as algebraic *expression graphs*, as will be later illustrated. Expression graphs, when available, are integral to the solution of nonlinear optimization problems. The AML is responsible for using these graphs to evaluate function values and

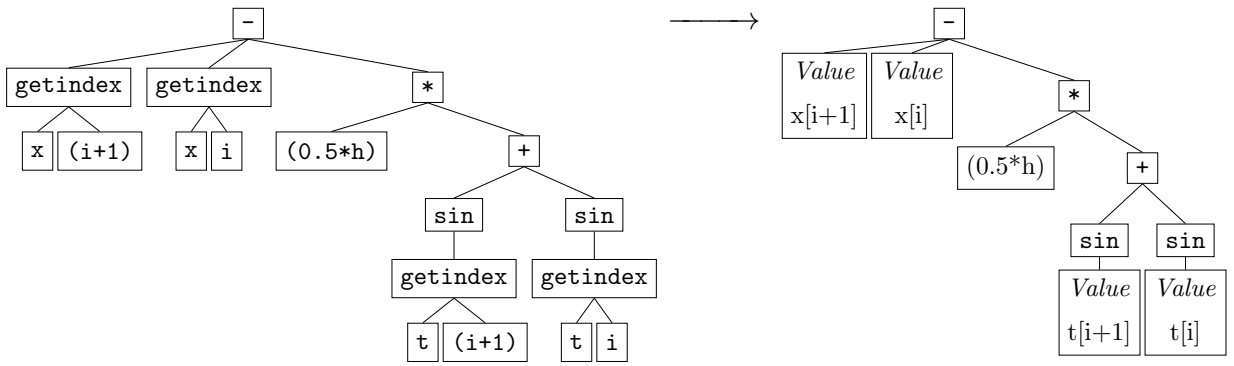

Figure 1 **A macro called with the text expression `x[i+1] - x[i] - (0.5h)*(sin(t[i+1])+sin(t[i]))` is given as input as the expression tree on the left. Parentheses indicate subtrees combined for brevity. At this stage, `symbols` are abstract and not resolved. To prepare the nonlinear expression, the macro produces code that generates the expression tree on the right with variable placeholders spliced in from the runtime context.**

first and second derivatives as requested by the solver (typically through *callbacks*). Additionally, they may be used by the AML to infer problem structure in order to decide which solution methods are appropriate (Fourer and Orban 2010) or by the solver itself to perform important problem reductions in the case of mixed-integer nonlinear programming (Belotti et al. 2009).

Analogously to the linear case, where macros are used to generate code which forms sparse vector representations, a macro was implemented which generates code to form nonlinear expression trees. Macros, when called, are provided an expression tree of the input; however, symbols are not resolved to values. Indeed, values do not exist at compile time when macros are evaluated. The task of the macro, therefore, is to generate code which replicates the input expression with runtime values (both numeric constants and variable placeholder objects) spliced in, as illustrated in Figure 1. This splicing of values is by construction and does not require expensive runtime calls such as MATLAB's `eval` function.

The implementation is compact, approximately 20 lines of code including support for the `sum{}` syntax presented in Section 2.2. While a nontrivial understanding of Julia's metaprogramming syntax is required to implement such a macro, the effort should be compared with what would be necessary to obtain the equivalent output and performance from a low-level language; in particular, one would need to write a custom expression parser.

Given expression trees for the constraints (1), we consider computing the Jacobian matrix

$$J(x) = \begin{bmatrix} \nabla g_1(x) \\ \nabla g_2(x) \\ \vdots \\ \nabla g_m(x) \end{bmatrix},$$

where $\nabla g_i(x)$ is a row-oriented gradient vector. Unlike the typical approach of using automatic differentiation for computing derivatives in AMLs (Gay 1996), a simpler method based on symbolic differentiation can be equally as efficient in Julia. In particular, we derive the form of the *sparse* Jacobian matrix by applying the chain rule symbolically and then, using JIT compilation, *compile a function which evaluates the Jacobian* for any given input vector. This process is accelerated by identifying equivalent expression trees (those which are symbolically identical and for which there exists a one-to-one correspondence between the variables present) and only performing symbolic differentiation once per equivalent expression.

The implementation of the Jacobian computation spans approximately 250 lines of code, including the basic logic for the chain rule. In the following section it is demonstrated that evaluating the Jacobian using the JIT compiled function is as fast as using AMPL through the low-level amplsolver library (Gay 1997), presently a de-facto standard for evaluating derivatives in nonlinear models. Interestingly, there is an executable accompanying the amplsolver library (`nlc`) which generates and compiles C code to evaluate derivatives for a specific model, although it is seldom used in practice because of the cost of compilation and the marginal gains in performance. However, in a language such as Julia with JIT compilation, compiling functions generated at runtime can be a technique to both simplify an implementation and obtain performance comparable to that of low-level languages.

3.2. Computational tests

We test our implementation on two nonlinear optimization problems obtained from Hans Mittelmann's AMPL-NLP benchmark set (`http://plato.asu.edu/ftp/ampl-nlp.html`). Experiments were performed on a Linux system with an Intel Xeon E5-2650 processor. Note that we have not developed a complete nonlinear AML; the implementation is intended to serve as a proof of concept only. The operations considered are solely the construction of

the model and the evaluation of the Jacobian of the constraints. Hence, objective functions and right-hand side expressions are omitted or simplified below.

The first instance is *clnlbeam*:

$$\begin{aligned}
\min_{t,x,u\in\mathbb{R}^{n+1}} \quad & \ldots \\
\text{subject to} \quad & x_{i+1} - x_i - \frac{1}{2n}(\sin(t_{i+1}) + \sin(t_i)) = 0 \quad i = 1,\ldots,n \\
& t_{i+1} - t_i - \frac{1}{2n}u_{i+1} - \frac{1}{2n}u_i = 0 \quad i = 1,\ldots,n \\
& -1 \le t_i \le 1, \quad -0.05 \le x_i \le 0.05 \quad i = 1,\ldots,n+1
\end{aligned}$$

We take $n =$ 5,000, 50,000, and 500,000. The following code builds the corresponding model in Julia using our proof-of-concept implementation:

```
m = Model(:Min)
h = 1/n
@defVar(m, -1 <= t[1:(n+1)] <= 1)
@defVar(m, -0.05 <= x[1:(n+1)] <= 0.05)
@defVar(m, u[1:(n+1)])

for i in 1:n
        @addNLConstr(m, x[i+1] - x[i] -
                (0.5h)*(sin(t[i+1])+sin(t[i])) == 0)
end
for i in 1:n
        @addNLConstr(m, t[i+1] - t[i] -
                (0.5h)*u[i+1] - (0.5h)*u[i] == 0)
end
```

The second instance is *cont5_1*:

$$\begin{aligned}
\min_{y\in\mathbb{R}^{(n+1)\times(n+1)},u\in\mathbb{R}^n} \quad & \ldots \\
\text{subject to} \quad & n(y_{i+1,j+1} - y_{i,j+1}) - a(y_{i,j} - 2y_{i,j} + y_{i,j+1} + y_{i+1,j} - 2y_{i+1,j+1} + y_{i+1,j+2}) = 0 \\
& \quad i = 1,\ldots,n,\ j = 1,\ldots,n-1 \\
& y_{i+1,3} - 4y_{i+1,2} + 3y_{i+1,1} = 0 \quad i = 1,\ldots,n \\
& c(y_{i+1,n-1} - 4y_{i+1,n} + 3y_{i+1,n+1}) + y_{i+1,n+1} - u_i + y_{i+1,n+1}((y_{i+1,n+1})^2)^{\frac{3}{2}} = 0 \\
& \quad i = 1,\ldots,n,
\end{aligned}$$

where $a = \frac{8n^2}{\pi^2}$ and $c = \frac{2n}{\pi}$. We take $n =$ 200, 400, and 1,000.

The dimensions of these instances and the number of nonzero elements in the corresponding Jacobian matrices are listed in Table 3. In Table 4 we present a benchmark of

our implementation compared with AMPL, YALMIP (MATLAB), and Pyomo (Python). JIT compilation of the Jacobian function is included in the "Build model" phase for Julia. Observe that Julia performs as fast as AMPL, if not faster. Julia's advantage over AMPL is partly explained by AMPL's need to write the model to an intermediate `nl` file before evaluating Jacobians; this I/O time is included. AMPL's preprocessing features are disabled. YALMIP performs well on the mostly linear cont5_1 instance but is unable to process the largest clnlbeam instance in under an hour. Pyomo's performance is more consistent but over 50x slower than Julia on the largest instances. Pyomo is run under pure Python; it does not support JIT accelerators such as PyPy.

Table 3 Nonlinear test instance dimensions. Nz = Nonzero elements in Jacobian matrix.

Instance	# Vars.	# Constr.	# Nz
clnlbeam-5	15,003	10,000	40,000
clnlbeam-50	150,003	100,000	400,000
clnlbeam-500	1,500,003	1,000,000	4,000,000
cont5_1-2	40,601	40,200	240,200
cont5_1-4	161,201	160,400	960,400
cont5_1-10	1,003,001	1,001,000	6,001,000

4. Implementing Optimization Algorithms

In this section we evaluate the performance of Julia for implementation of the simplex method for linear programming, arguably one of the most important algorithms in the field of operations research. Our aim is not to develop a complete implementation but instead to compare the performance of Julia to that of other popular languages, both high- and low-level, on a benchmark of core operations.

Although high-level languages can achieve good performance when performing *vectorized* operations (that is, block operations on dense vectors and matrices), state-of-the-art implementations of the simplex method are characterized by their effective exploitation of sparsity (the presence of many zeros) in all operations, and hence, they use sparse linear algebra. Opportunities for vectorized operations are small in scale and do not represent

Table 4 **Nonlinear benchmark results. "Build model" includes writing and reading model files, if required, and precomputing the structure of the Jacobian. Pyomo uses AMPL for Jacobian evaluations.**

	Build model (s)				Evaluate Jacobian (ms)		
Instance	AMPL	Julia	YALMIP	Pyomo	AMPL	Julia	YALMIP
clnlbeam-5	0.2	0.1	36.0	2.3	0.4	0.3	8.3
clnlbeam-50	1.8	0.3	1344.8	23.7	7.3	4.2	96.4
clnlbeam-500	18.3	3.3	>3600	233.9	74.1	74.6	*
cont5_1-2	1.1	0.3	2.0	12.2	1.1	0.8	9.3
cont5_1-4	4.4	1.4	1.9	49.4	5.4	3.0	37.4
cont5_1-10	27.6	6.1	13.5	310.4	33.7	39.4	260.0

a majority of the execution time; see Hall (2010). Furthermore, the sparse linear algebra operations used, such as Suhl and Suhl (1990)'s LU factorization, are specialized and not provided by standard libraries.

The simplex method is therefore an example of an algorithm that requires a low-level coding style, in particular, manually-coded loops, which are known to have poor performance in languages such as Matlab or Python (see, e.g., van der Walt et al. (2011)). To achieve performance in such cases, one would be required to code time-consuming loops in another language and link to these separate routines from the high-level language, using, for example, Matlab's MEX interface. Our benchmarks will demonstrate, however, that within Julia, the native performance of this style of computation can nearly achieve that of low-level languages.

4.1. Benchmark Operations

A presentation of the simplex algorithm and a discussion of its computational components are beyond the scope of this paper. We refer the reader to Maros (2003) and Koberstein (2005) for a comprehensive treatment of modern implementations, which include significant advances over versions presented in most textbooks. We instead present three selected operations from the *revised dual simplex* method in a mostly self-contained manner. Knowledge of the simplex algorithm is helpful but not required. The descriptions are realistic and reflect the essence of the routines as they might be implemented in an efficient implementation.

The first operation considered is a matrix-transpose-vector product (Mat-Vec). In the revised simplex method, this operation is required in order to form a row of the *tableau*. A nonstandard aspect of this Mat-Vec is that we would like to consider the matrix formed by a constantly changing subset the columns (those corresponding to the non-basic variables). Another important aspect is the treatment of sparsity of the *vector* itself, in addition to that of the matrix (Hall and McKinnon 2005). This is achieved algorithmically by using the nonzero elements of the vector to form a linear combination of the rows of the matrix, instead of the more common approach of computing dot-products with the columns, as illustrated in (2). This follows from viewing the matrix A equivalently as either a collection of column vectors A_i and or as row vectors a_i^T.

$$A = \begin{bmatrix} A_1 & A_2 & \cdots & A_n \end{bmatrix} = \begin{bmatrix} a_1^T \\ a_2^T \\ \vdots \\ a_m^T \end{bmatrix} \longrightarrow A^T x = \begin{bmatrix} A_1^T x \\ A_2^T x \\ \vdots \\ A_n^T x \end{bmatrix} = \sum_{\substack{i=1 \\ x_i \neq 0}}^{m} a_i x_i \tag{2}$$

Algorithm 1 Restricted sparse matrix transpose-dense vector product

Input: Sparse column-oriented $m \times n$ matrix A, dense vector $x \in \mathbb{R}^m$, and
flag vector $\mathcal{N} \in \{0,1\}^n$ (with $n - m$ nonzero elements)
Output: $y := A_{\mathcal{N}}^T x$ as a dense vector, where $\mathcal{N}$ selects columns of A
for i **in** $\{1, \ldots, n\}$ **do**
  if $\mathcal{N}_i = 1$ **then**
    $s \leftarrow 0$
    For each nonzero element q (in row j) of ith column of A **do**
      $s \leftarrow s + q * x_j$ ▷ Compute dot-product of x with column i
    end for
    $y_i \leftarrow s$
  end if
end for

The Mat-Vec operation is illustrated for dense vectors in Algorithm 1 and for sparse vectors in Algorithm 2. Sparse matrices are provided in either compressed sparse column (CSC) or compressed sparse row (CSR) format as appropriate (Duff et al. 1989). Note that in Algorithm 1 we use a flag vector to indicate the selected columns of the matrix A. This corresponds to skipping particular dot products. The result vector has a memory layout

Algorithm 2 Sparse matrix transpose-sparse vector product

Input: Sparse row-oriented $m \times n$ matrix A and sparse vector $x \in \mathbb{R}^m$
Output: Sparse representation of $A^T x$.
For each nonzero element p (in index j) in x **do**
  For each nonzero element q (in column i) of jth row of A **do**
    Add $p * q$ to index i of output. ▷ Compute linear combination of rows of A
  end for
end for

with n, not $n - m$ entries. This form could be desired in some cases for subsequent operations and is illustrative of the common practice in simplex implementations of designing data structures with a global view of the operations in which they will be used (Maros 2003, Chap. 5). In Algorithm 2 we omit what would be a costly flag check for each nonzero element of the row-wise matrix; the gains of exploiting sparsity often outweigh the extra floating-point operations.

Algorithm 3 Two-pass stabilized minimum ratio test (dual simplex)

Input: Vectors $d, \alpha \in \mathbb{R}^n$, state vector $s \in \{\text{“lower”}, \text{“basic”}\}^n$, parameters $\epsilon_{\mathrm{P}}, \epsilon_{\mathrm{D}} > 0$
Output: Solution index *result*.
$\Theta_{\max} \leftarrow \infty$
for i **in** $\{1, \ldots, n\}$ **do**
  if $s_i =$ “lower” and $\alpha_i > \epsilon_{\mathrm{P}}$ **then**
    Add index i to list of candidates
    $\Theta_{\max} \leftarrow \min(\frac{d_i + \epsilon_{\mathrm{D}}}{\alpha_i}, \Theta_{\max})$
  end if
end for
$\alpha_{\max} \leftarrow 0$, $result \leftarrow 0$
for i **in** list of candidates **do**
  if $d_i/\alpha_i \le \Theta_{\max}$ and $\alpha_i > \alpha_{\max}$ **then**
    $\alpha_{\max} \leftarrow \alpha_i$
    $result \leftarrow i$
  end if
end for

The second operation is the minimum ratio test, which determines both the step size of the next iteration and the constraint that prevents further progress. Mathematically this may be expressed as

$$\min_{\alpha_i > 0} \frac{d_i}{\alpha_i},$$

for given vectors d and α. While seemingly simple, this operation is one of the more complex parts of an implementation, as John Forrest mentions in a comment in the source code of the open-source Clp solver. We implement a relatively simple two-pass variant (Algorithm 3) due to Harris (1973) and described more recently in (Koberstein 2005, Sect.

6.2.2.2), whose aim is to avoid numerical instability caused by small values of α_i. In the process, small infeasibilities up to a numerical tolerance ϵ_{D} may be created. Note that our implementation handles both upper and lower bounds; Algorithm 3 is simplified in this respect for brevity. A sparse variant is easily obtained by looping over the nonzero elements of α in the first pass.

The third operation is a modified form of the vector update $y \leftarrow \alpha x + y$ (Axpy). In the variant used in the simplex algorithm, the value of each updated component is tested for membership in an interval. For example, given a tolerance ϵ, a component belonging to the interval $(-\infty, -\epsilon)$ may indicate loss of numerical feasibility, in which case a certain corrective action, such as local perturbation of problem data, may be triggered. This procedure is more naturally expressed using an explicit loop over elements of x instead of performing operations on vectors.

The three operations discussed represent a nontrivial proportion of execution time of the simplex method, between 20% and 50% depending on the problem instance (Hall and McKinnon 2005). Most of the remaining execution time is spent in factorizing and solving linear systems using specialized procedures, which we do not implement because of their complexity.

4.2. Results

The benchmark operations described in the previous section were implemented in Julia, C++, MATLAB, and Python. Examples of code have been omitted for brevity. The style of the code in Julia is qualitatively similar to that of the other high-level languages. Readers are encouraged to view the implementations available in the online supplement. To measure the overhead of bounds-checking, a validity check performed on array indices in high-level languages, we implemented a variant in C++ with explicit bounds checking. We also consider executing the Python code under the PyPy engine (Bolz et al. 2009), a JIT-compiled implementation of Python. We have not used the popular NumPy library in Python because it would not alleviate the need for manually coded loops and so would provide little speed benefit. No special runtime parameters are used, and the C++ code is compiled with `-O2`.

Realistic input data were generated by running a modified implementation of the dual simplex algorithm on a small set of standard LP problems and recording the required input data for each operation from iterations sampled uniformly over the course of the

algorithm. At least 200 iterations are recorded from each instance. Using such data from real instances is important because execution times depend significantly on the sparsity patterns of the input. The instances we consider are greenbea, stocfor3, and ken-13 from the NETLIB repository (Gay 1985) and the fome12 instance from Hans Mittelmann's benchmark set (Mittelmann 2013). These instances represent a range of problem sizes and sparsity structures.

Experiments were performed under the Linux operating system on a laptop with an Intel i5-3320M processor. See Table 5 for a summary of results. Julia consistently performs within a factor of 2 of the implementation in C++ with bounds checking, while MATLAB and PyPy are within a factor of 4 to 18. Pure Python is far from competitive, being at least 70x slower than C++.

Figure 2 displays the absolute execution times broken down by instance. We observe the consistent performance of Julia, while that of MATLAB and PyPy are subject to more variability. In all cases except the smaller greenbea instance, use of the vector-sparse routines significantly decreases execution time, although PyPy's performance is relatively poorer on these routines.

Table 5 Execution time of each language (version listed below) relative to C++ with bounds checking. Lower values are better. Figures are geometric means of average execution times over iterations over 4 standard LP problems. Recorded value is fastest time of three repetitions. Dense/sparse distinction refers to the vector x; all matrices are sparse.

	Operation	Julia 0.1	C++ GCC 4.7.2	MATLAB R2012b	PyPy 1.9	Python 2.7.3
Dense	Mat-Vec ($A_{\mathcal{N}}^T x$)	1.27	0.79	7.78	4.53	84.69
	Min. ratio test	1.67	0.86	5.68	4.54	70.95
	Axpy ($y \leftarrow \alpha x + y$)	1.37	0.68	10.88	3.07	83.71
Sparse	Mat-Vec ($A^T x$)	1.25	0.89	5.72	6.56	69.43
	Min. ratio test	1.65	0.78	4.35	13.62	73.47
	Axpy ($y \leftarrow \alpha x + y$)	1.84	0.68	17.83	8.57	81.48

Our results are qualitatively similar to those reported by Bezanson et al. (2012) on a set of unrelated general language benchmarks and thus serve as an independent corroboration

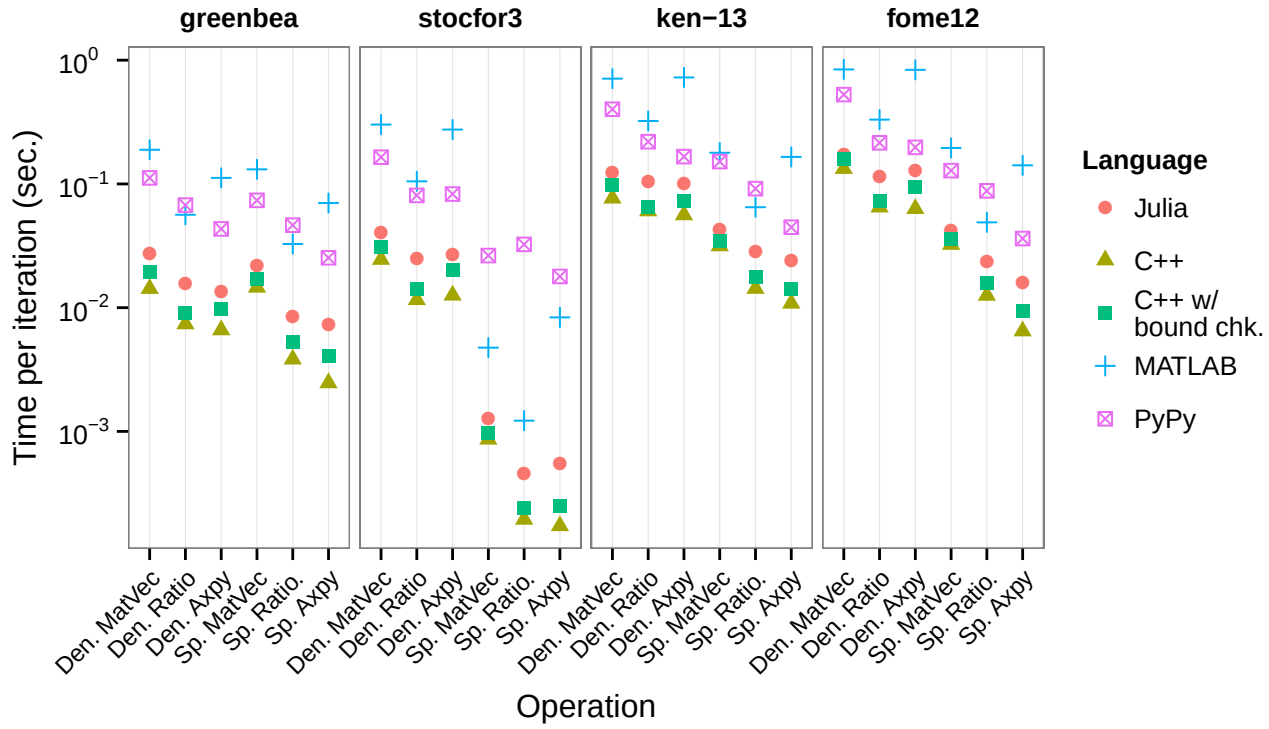

Figure 2 **Average execution time for each operation and language, by instance. Compared with MATLAB and PyPy, the execution time of Julia is significantly closer to that of C++.**

of their findings that Julia's performance is within a factor of 2 of equivalent low-level compiled code.

Acknowledgments

This work would not be possible without the effort of the Julia team, Jeff Bezanson, Stefan Karpinski, Viral Shah, and Alan Edelman, as well as that of the larger community of Julia contributors. We acknowledge, in particular, Carlo Baldassi and Dahua Lin for significant contributions to the development of interfaces for linear programming solvers. We thank Juan Pablo Vielma for his comments on this manuscript which substantially improved its presentation. M. Lubin was supported by the DOE Computational Science Graduate Fellowship, which is provided under grant number DE-FG02-97ER25308.

References

Belotti, Pietro, Jon Lee, Leo Liberti, Franois Margot, Andreas Wächter. 2009. Branching and bounds tightening techniques for non-convex MINLP. *Optimization Methods and Software* **24** 597–634.

Bezanson, Jeff, Stefan Karpinski, Viral B. Shah, Alan Edelman. 2012. Julia: A fast dynamic language for technical computing. *CoRR* **abs/1209.5145**.

Bixby, Robert E. 2002. Solving real-world linear programs: A decade and more of progress. *Operations research* **50** 3–15.

Bolz, Carl Friedrich, Antonio Cuni, Maciej Fijalkowski, Armin Rigo. 2009. Tracing the meta-level: PyPy's tracing JIT compiler. *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. ICOOOLPS '09, ACM, New York, 18–25.

Brooke, A., D. Kendrick, A. Meeraus, R. Raman. 1999. *GAMS: A User's Guide*. Scientific Press.

Duff, I. S., Roger G. Grimes, John G. Lewis. 1989. Sparse matrix test problems. *ACM Trans. Math. Softw.* **15** 1–14.

Fourer, R., Dominique Orban. 2010. DrAmpl: a meta solver for optimization problem analysis. *Computational Management Science* **7** 437–463.

Fourer, Robert, David M Gay, Brian W Kernighan. 1993. *AMPL*. Scientific Press.

Gay, David M. 1985. Electronic mail distribution of linear programming test problems. *Mathematical Programming Society COAL Newsletter* **13** 10–12.

Gay, David M. 1996. More AD of nonlinear AMPL models: Computing hessian information and exploiting partial separability. *in Computational Differentiation: Applications, Techniques, and Tools*. SIAM, 173–184.

Gay, David M. 1997. Hooking your solver to AMPL. Tech. rep., Bell Laboratories, Murray Hill, NJ.

Grant, Michael C., Stephen P. Boyd. 2013. The CVX users' guide (release 2.0). URL `http://cvxr.com/cvx/doc/CVX.pdf`.

Hall, J. 2010. Towards a practical parallelisation of the simplex method. *Computational Management Science* **7** 139–170.

Hall, J., K. McKinnon. 2005. Hyper-sparsity in the revised simplex method and how to exploit it. *Computational Optimization and Applications* **32** 259–283.

Harris, Paula M. J. 1973. Pivot selection methods of the DEVEX LP code. *Mathematical Programming* **5** 1–28.

Hart, William E, Jean-Paul Watson, David L Woodruff. 2011. Pyomo: modeling and solving mathematical programs in Python. *Mathematical Programming Computation* **3** 219–260.

Koberstein, Achim. 2005. The dual simplex method, techniques for a fast and stable implementation. Ph.D. thesis, Universität Paderborn, Paderborn, Germany.

Lattner, Chris, Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. *Code Generation and Optimization, 2004. International Symposium on*. IEEE, 75–86.

Lofberg, John. 2004. YALMIP: A toolbox for modeling and optimization in MATLAB. *Computer Aided Control Systems Design, 2004 IEEE International Symposium on*. IEEE, 284–289.

Maros, István. 2003. *Computational Techniques of the Simplex Method*. Kluwer Academic Publishers, Norwell, MA.

Mitchell, Stuart, Michael O'Sullivan, Iain Dunning. 2011. Pulp: A linear programming toolkit for python URL `https://code.google.com/p/pulp-or/`. Unpublished manuscript.

Mittelmann, Hans. 2013. Benchmarks for optimization software. URL `http://plato.la.asu.edu/bench.html`. Accessed April 28, 2013.

Suhl, Uwe H., Leena M. Suhl. 1990. Computing sparse LU factorizations for large-scale linear programming bases. *ORSA Journal on Computing* **2** 325.

van der Walt, S., S.C. Colbert, G. Varoquaux. 2011. The NumPy array: A structure for efficient numerical computation. *Computing in Science Engineering* **13** 22 –30.