Computing in Operations Research using Julia

Miles Lubin and Iain Dunning

MIT Operations Research Center

INFORMS 2013 – October 7, 2013

- High-level, high-performance, open-source dynamic language for technical computing.
- Keep productivity of dynamic languages without giving up speed.
- Familiar syntax
- Python+PyPy+SciPy+NumPy integrated completely.
- Latest concepts in programming languages.

- Claim: "close-to-C" speeds
  - **Within a factor of 2**

	Fortran	Julia	Python	Matlab
fib	0.28	1.97	46.03	1587.03
parse_int	9.22	1.72	25.29	846.67
quicksort	1.65	1.37	69.20	133.46
mandel	0.76	1.45	34.88	74.61
pi_sum	1.00	1.00	33.64	1.46
rand_mat_stat	2.23	1.95	29.01	7.71
rand_mat_mul	1.14	1.00	1.75	1.08

- Performs well on microbenchmarks, but how about real computational problems in OR? Can we stop writing solvers in C++?

Technical advancements in Julia:

- Fast code generation (JIT via LLVM).
- Excellent connections to C libraries - BLAS/LAPACK/...
- Metaprogramming.
- Optional typing, multiple dispatch.

- Write generic code, compile efficient type-specific code

C: (fast)

```c
int f() {
    int x = 1, y = 2;
    return x+y;
}
```

Julia: (No type annotations)

```julia
function f()
    x = 1; y = 2
    return x + y
end
```

Python: (slow)

```python
def f():
    x = 1; y = 2
    return x+y
```

- Requires *type inference* by compiler
- Difficult to add onto exiting languages
    - Available in MATLAB – limited scope
    - PyPy for Python – incompatible with many libraries
- Julia designed from the ground up to support type inference efficiently

Simplex algorithm

- "Bread and butter" of operations research
- Computationally very challenging to implement efficiently[1]
- Matlab implementations too slow to be used in practice
  - High-quality open-source codes exist in C/C++
- Can Julia compete?

[1] Bixby, Robert E. "Solving Real-World Linear Programs: A Decade and More of Progress", *Operations Research*, Vol. 50, pp. 3–15, 2002.

- Implemented benchmark operations in Julia, C++, MATLAB, Python.
- Run on real iteration data from 4 diverse instances from NETLIB
- `https://github.com/mlubin/SimplexBenchmarks`

- Sparse matrix-vector product
  - Used to compute row $(A^T(B^{-T}x))$ of tableau
- Ratio test
  - Realistic two-pass test for numerical stability
- Vector update $(y \leftarrow \alpha x + y)$
  - For updating primal and dual iterates.
- Both vector-dense and vector-sparse variants.
- Combined 20%–50% of total execution time on typical instances.

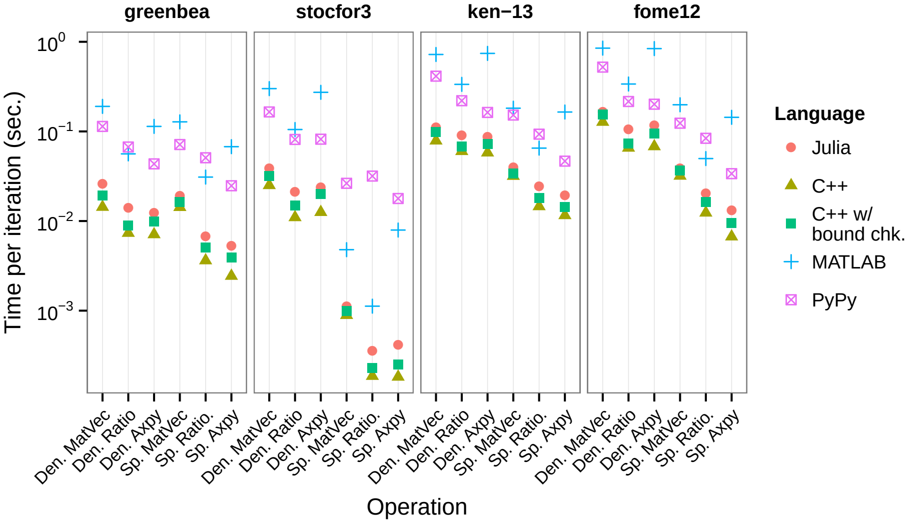
greenbea
stocfor3
ken−13
fome12
Time per iteration (sec.)
10^0
10^{-1}
10^{-2}
10^{-3}
Den. MatVec
Den. Ratio
Den. Axpy
Sp. MatVec
Sp. Ratio.
Sp. Axpy
Operation
Language
Julia
C++
C++ w/ bound chk.
MATLAB
PyPy

Table: Execution time of each language (version listed below) relative to C++ with bounds checking. Lower values are better. Dense/sparse distinction refers to the *vector* x; all matrices are sparse.

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

	Operation	Julia 0.1	Julia 0.2
Dense	$A_{\mathcal{N}}^T x$	1.27	1.19
	Ratio test	1.67	1.43
	$y \leftarrow \alpha x + y$	1.37	1.23
Sparse	$A^T x$	1.25	1.13
	Ratio test	1.65	1.38
	$y \leftarrow \alpha x + y$	1.84	1.43

- 13% speedup since last release in February.

Macros and metaprogramming

- Julia is homoiconic: code represented as a data structure
- Consider:

```
x = 2; y = 5  # Initialize variables
2x + y^x      # Prints 29 on terminal
```

- Expression is stored like

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

- Macro: a function that operates on code, e.g.

```
macro m(ex)
  ex.args[1] = :(-)  # Replace operation with subtraction
  return esc(ex)     # Escape expression
end
@m(2x + y^x)         # Prints 2*2 - 5^2 = -21
```

- Transform existing code, and generate new code

$$(-, (*, 2, x), (\wedge, y, x))$$

Algebraic Modeling Languages

- Dedicated/commercial - e.g. AMPL, GAMS
  - Fast and expressive, not general purpose
  - AMPL:

```
var pick {i in 1..N} >= 0;
maximize Obj:
    sum {i in 1..N} profit[i] * pick[i];
```

- Embedded/open-source - e.g. PuLP, Pyomo, CVX, YALMIP
  - Domain-specific language embedded in Python/MATLAB/...
  - Work via "operator overloading" - slow
  - PuLP (Python):

```
prob = LpProblem("knapsack", LpMaximize)
pick = LpVariable.dicts("Pick",[i in range(N)], 0)
prob += sum(profit[i] * pick[i] for i in range(N)), "Obj"
```

- JuMP is AML in Julia that supports MILP, MIQCQP
- Use macros to avoid issues with operator overloading

```
m = Model(:Max)

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

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

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

- Goal: sparse representation of rows as pairs (number,variable)
- AMPL: could determine storage in first pass, evaluate in second
- Operator overloading: multiple allocations, final size unknown
- Julia macro: macro analyzes constraint, preallocates space, evaluates at run-time.
    - Generates code you would've written by hand.

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

```
vector<Variable> vars;
vector<double> coeffs;

vars.reserve( N );
coeffs.reserve( N );
for (int i = 1; i <= N; i++) {
    vars.push_back( x[i] );
    coeffs.push_back( weight[i] );
}

vars.push_back( s );
coeffs.push_back( 1.0 );

model.addConstraint(vars, coeffs, "<=", capacity);
```

Table: 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

Availability

`http://github.com/IainNZ/JuMP.jl`

- Available via Julia package manager
- **Completely documented!**
- GPL license
- Solver independent (COIN Clp, COIN Cbc, GLPK, Gurobi)
- Works on Linux, OS X, and Windows

Nonlinear modeling

$$\begin{aligned} &\min\ f(x) \\ &\text{s.t.}\ \ g(x) \leq 0 \end{aligned}$$

- AMLs need to provide derivatives of expressions $f(x)$ and $g(x)$ to solvers
- Traditional technique: automatic differentiation
  - Outputs representation of derivative, e.g. .nl file
  - Complex implementation
- Julia
  - Apply chain rule directly to symbolic expression
  - JIT compile a function which evaluates the derivative

```
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
```

```
    @addNLConstr(m, x[i+1] - x[i] -
            (0.5h)*(sin(t[i+1])+sin(t[i])) == 0)
```

```
void eval_jac(double *x, int *iRow, int *jCol, double *values)
{
    int vindex1[] = {...};
    ...

    for (int i = 0; i < n; i++)
        values[vindex1[i]] = 1;
    for (int i = 0; i < n; i++)
        values[vindex2[i]] = -1;
    for (int i = 0; i < n; i++)
        values[vindex3[i]] = -0.5*h*cos(x[xindex3[i]]);
    for (int i = 0; i < n; i++)
        values[vindex4[i]] = -0.5*h*cos(x[xindex4[i]]);

}
```

- In-place update of values of sparse matrix
- `vindex` and `xindex` precomputed, matching sparse indices

Table: 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)		
Prob.	AMPL	Julia	YALMIP	Pyomo	AMPL	Julia	YALMIP
A-5	0.2	0.1	36.0	2.3	0.4	0.3	8.3
A-50	1.8	0.3	1344.8	23.7	7.3	4.2	96.4
A-500	18.3	3.3	>3600	233.9	74.1	74.6	*
B-2	1.1	0.3	2.0	12.2	1.1	0.8	9.3
B-4	4.4	1.4	1.9	49.4	5.4	3.0	37.4
B-10	27.6	6.1	13.5	310.4	33.7	39.4	260.0

Conclusions

- Julia delivers on promise of C-like performance.
- Alegebraic modeling via metaprogramming
- JuMP released for use
- New features under active development