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

<table>
<thead>
<tr>
<th>Function</th>
<th>Fortran</th>
<th>Julia</th>
<th>Python</th>
<th>Matlab</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib</td>
<td>0.28</td>
<td>1.97</td>
<td>46.03</td>
<td>1587.03</td>
</tr>
<tr>
<td>parse_int</td>
<td>9.22</td>
<td>1.72</td>
<td>25.29</td>
<td>846.67</td>
</tr>
<tr>
<td>quicksort</td>
<td>1.65</td>
<td>1.37</td>
<td>69.20</td>
<td>133.46</td>
</tr>
<tr>
<td>mandel</td>
<td>0.76</td>
<td>1.45</td>
<td>34.88</td>
<td>74.61</td>
</tr>
<tr>
<td>pi_sum</td>
<td>1.00</td>
<td>1.00</td>
<td>33.64</td>
<td>1.46</td>
</tr>
<tr>
<td>rand_mat_stat</td>
<td>2.23</td>
<td>1.95</td>
<td>29.01</td>
<td>7.71</td>
</tr>
<tr>
<td>rand_mat_mul</td>
<td>1.14</td>
<td>1.00</td>
<td>1.75</td>
<td>1.08</td>
</tr>
</tbody>
</table>

Perform 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?

- 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.
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.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Julia 0.1</th>
<th>C++ GCC</th>
<th>MATLAB R2012b</th>
<th>PyPy 1.9</th>
<th>Python 2.7.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dense $A^T_N x$</td>
<td>1.27</td>
<td>0.79</td>
<td>7.78</td>
<td>4.53</td>
<td>84.69</td>
</tr>
<tr>
<td>Ratio test</td>
<td>1.67</td>
<td>0.86</td>
<td>5.68</td>
<td>4.54</td>
<td>70.95</td>
</tr>
<tr>
<td>$y \leftarrow \alpha x + y$</td>
<td>1.37</td>
<td>0.68</td>
<td>10.88</td>
<td>3.07</td>
<td>83.71</td>
</tr>
<tr>
<td>Sparse $A^T x$</td>
<td>1.25</td>
<td>0.89</td>
<td>5.72</td>
<td>6.56</td>
<td>69.43</td>
</tr>
<tr>
<td>Ratio test</td>
<td>1.65</td>
<td>0.78</td>
<td>4.35</td>
<td>13.62</td>
<td>73.47</td>
</tr>
<tr>
<td>$y \leftarrow \alpha x + y$</td>
<td>1.84</td>
<td>0.68</td>
<td>17.83</td>
<td>8.57</td>
<td>81.48</td>
</tr>
<tr>
<td>Operation</td>
<td>Julia 0.1</td>
<td>Julia 0.2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>-----------</td>
<td>-----------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dense $A_N^T x$</td>
<td>1.27</td>
<td>1.19</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ratio test</td>
<td>1.67</td>
<td>1.43</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$y \leftarrow \alpha x + y$</td>
<td>1.37</td>
<td>1.23</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sparse $A^T x$</td>
<td>1.25</td>
<td>1.13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ratio test</td>
<td>1.65</td>
<td>1.38</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$y \leftarrow \alpha x + y$</td>
<td>1.84</td>
<td>1.43</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 13% speedup since last release in February.
Julia is homoiconic: code represented as a data structure

Consider:

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

Expression is stored like

```plaintext
(+, (*, 2, x), (^, y, x))
```
Macro: a function that operates on code, e.g.

```python
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

\[ (-, (\ast, 2, x), (\wedge, y, x)) \]
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

```julia
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)

type vars = vector<Variable>;
type coeffs = vector<double>;

def vars: = vector<Variable>(N);
def coeffs: = vector<double>(N);

def main()
    for (int i = 1; i <= N; i++) {
        var x[i]:
        coeff weight[i]:
        vars.push_back(x[i]);
        coeffs.push_back(weight[i]);
    }

def s: = vars.push_back(s);
def capacity: = coeffs.push_back(1.0);

model.addConstraint(vars, coeffs, "\leq", 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).

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

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

- 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.

<table>
<thead>
<tr>
<th>Prob.</th>
<th>Build model (s)</th>
<th>Evaluate Jacobian (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AMPL</td>
<td>Julia</td>
</tr>
<tr>
<td>A-5</td>
<td>0.2</td>
<td>0.1</td>
</tr>
<tr>
<td>A-50</td>
<td>1.8</td>
<td>0.3</td>
</tr>
<tr>
<td>A-500</td>
<td>18.3</td>
<td>3.3</td>
</tr>
<tr>
<td>B-2</td>
<td>1.1</td>
<td>0.3</td>
</tr>
<tr>
<td>B-4</td>
<td>4.4</td>
<td>1.4</td>
</tr>
<tr>
<td>B-10</td>
<td>27.6</td>
<td>6.1</td>
</tr>
</tbody>
</table>
Conclusions

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