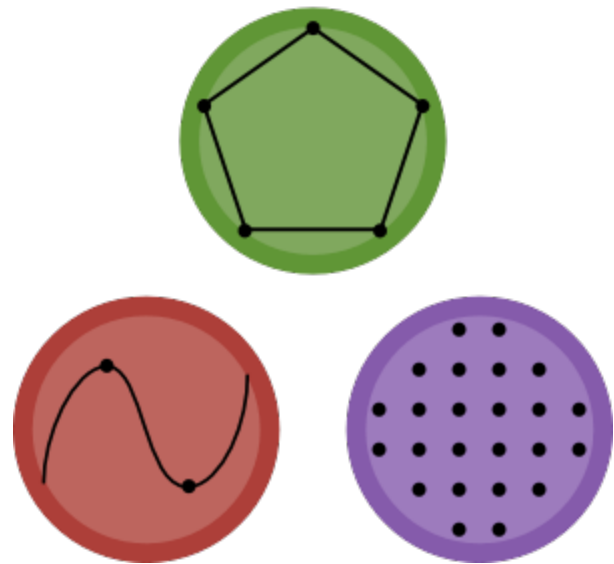

JuliaOpt: Optimization packages for Julia



Iain Dunning and Joey Huchette

Mathematical Optimization

$$\begin{array}{ll} \min_x & f(x) \\ \text{subject to} & g_i(x) \leq 0 \quad \forall i \end{array}$$

- Machine learning, resource allocation, production planning, scheduling, control...
-

Mathematical Optimization

Many flavours with different approaches:

- **unconstrained optimization** (*machine learning, model fitting, ...*) versus **constrained problems** (*transportation, logistics, control systems, ...*)
 - continuous, combinatorial, derivative-free, online, ...
-

Mathematical Optimization

- Need to be able to write code that is a **natural** and **maintainable** translation of the mathematics of our problem.
 - Preferably loosely coupled to the specific solution method and data, but also...
 - Need specialized solvers for linear problems, linear-integer, conic, semidefinite, ...
-

Structure of Talk

Part 1

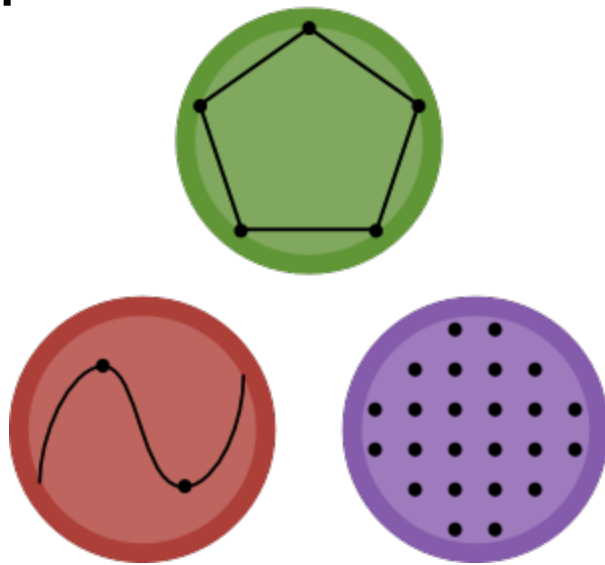
What are JuliaOpt, JuMP, and MathProgBase?

Part 2

How Julia makes cool things possible

What is JuliaOpt?

- A collection of high-quality optimization-related packages
 - Documentation compulsory
 - Tests compulsory
 - Cross-platform binaries
 - BinDeps
- <http://juliaopt.org>
- <http://github.com/JuliaOpt>
- [julia-opt](#) mailing list



JuliaOpt Packages (June 2014)

- JuMP
 - MathProgBase
 - Optim
 - NLOpt
 - LsqFit
 - Mosek
 - Clp/Cbc
 - Gurobi
 - Ipopt
 - GLPK
-

JuliaOpt Packages (June 2014)

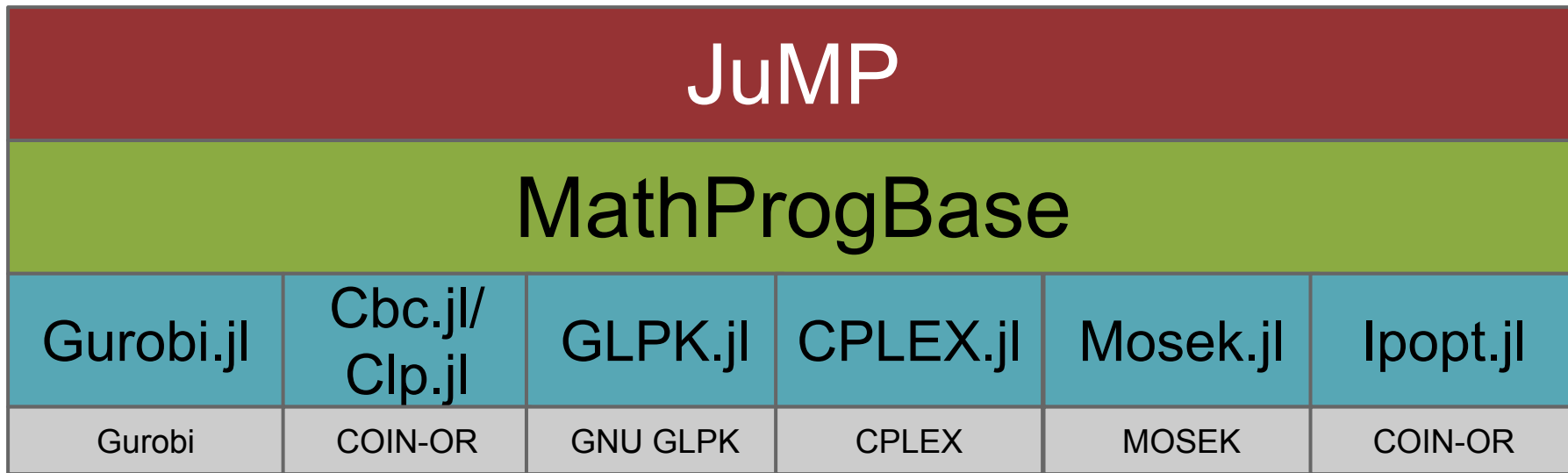
- JuMP
- MathProgBase
- Optim
- NLOpt
- LsqFit

- Mosek
- Clp/Cbc
- Gurobi
- Ipopt
- GLPK

Main focus for today's talk

The JuliaOpt Stack

([Iain Dunning](#), [Miles Lubin](#), [Joey Huchette](#), [Carlo Baldassi](#), [Dahua Lin](#), [Ulf Worsøe](#))



Light-weight **wrappers** around C interfaces of powerful commercial and open-source solvers

LPs, MILPs, SOCPs, nonlinear...

JuMP					
MathProgBase					
Gurobi.jl	Cbc.jl/ Clp.jl	GLPK.jl	CPLEX.jl	Mosek.jl	Ipopt.jl
Gurobi	COIN-OR	GNU GLPK	CPLEX	MOSEK	COIN-OR

An **algebraic modelling language** that lets you express optimization problems naturally and quickly in Julia programs: <http://github.com/JuliaOpt/JuMP>

c.f. [YALMIP](#), [PuLP](#), [AMPL](#), ...

JuMP

MathProgBase

Gurobi.jl

Cbc.jl/
Clp.jl

GLPK.jl

CPLEX.jl

Mosek.jl

Ipopt.jl

Gurobi

COIN-OR

GNU GLPK

CPLEX

MOSEK

COIN-OR

JuMP: Algebraic modeling

- AMLs allow us to translate mathematical statements of optimization problems into code.
 - They efficiently generate the (sparse) data structures that solvers require as input.
 - *Much* easier to reason about (and more generic) than hand-coding problem structure.
-

JuMP Philosophy

- Make modeling quick and painless for practitioners (a la AMPL)...
 - ...while allowing experts to easily use integrate advanced features (usually only available through low-level solver interfaces)
 - Make it as **fast** as possible!
-

Example 1: Sudoku

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

- 1 to 9 in each 3x3 square
- 1 to 9 in each row
- 1 to 9 in each column
- Each cell has 1 to 9 (implied)
- Define
$$x_{ijk} = 1 \text{ iff cell } (i,j) = k$$
- MILP!

Example 1: Sudoku

all numbers appear in each row

$$\sum_{j \in \{1, \dots, 9\}} x_{ijk} = 1$$

$$\forall i \in \{1, \dots, 9\}, k \in \{1, \dots, 9\}$$

all numbers appear in each subgrid

$$\sum_{i,j=1}^3 x_{i+I,j+J,k} = 1$$

$$\forall I, J \in \{0, 3, 6\}, k \in \{1, \dots, 9\}$$

JuMP + Julia

```
@addConstraint(m,  
    row[i=1:9,k=1:9],  
    sum(x[i,:,k]) == 1)
```

Gurobi + C

```
for (v = 0; v < DIM; v++) {  
    for (j = 0; j < DIM; j++) {  
        for (i = 0; i < DIM; i++) {  
            ind[i] = i*DIM*DIM + j*DIM + v;  
            val[i] = 1.0;  
        }  
        error = GRBaddconstr(model, DIM,  
            ind, val, GRB_EQUAL, 1.0, NULL);  
        if (error) goto QUIT;  
    }  
}
```


JuMP + Julia

```
@addConstraint(m,  
    subgrid[i=1:3:7,j=1:3:7,k=1:9],  
    sum(x[i:i+2,j:j+2,k]) == 1)
```

Gurobi + C

```
for (v = 0; v < DIM; v++) {  
    for (ig = 0; ig < SUBDIM; ig++) {  
        for (jg = 0; jg < SUBDIM; jg++) {  
            count = 0;  
            for (i = ig*SUBDIM; i < (ig+1)*SUBDIM; i++) {  
                for (j = jg*SUBDIM; j < (jg+1)*SUBDIM; j++) {  
                    ind[count] = i*DIM*DIM + j*DIM + v;  
                    val[count] = 1.0;  
                    count++;  
                }  
            }  
            error = GRBaddconstr(model, DIM, ind, val,  
GRB_EQUAL, 1.0, NULL);  
            if (error) goto QUIT;  
        }  
    }  
}
```

Example 2: Wind power dispatch

- (Very) recent work with Argonne Lab
 - Front-end to custom parallel stochastic interior point solver
 - JuMP: ~70 LOC for modeling, ~500 for “glue”
 - C++: ~2300 LOC
 - Added bonuses: fast prototyping, *generic*
-

A **common interface** to solvers, as well as allow
MATLAB-style optimization calls, e.g

`linprog(A,b,c...)` <http://github.com/JuliaOpt/MathProgBase>

JuMP					
MathProgBase					
Gurobi.jl	Cbc.jl/ Clp.jl	GLPK.jl	CPLEX.jl	Mosek.jl	Ipopt.jl
Gurobi	COIN-OR	GNU GLPK	CPLEX	MOSEK	COIN-OR

MathProgBase

- Defines an **AbstractMathProgSolver**, documents interface, use multiple dispatch
 - **For users:** Ignore solver interface details, decouple problem from solver
 - **For package developers:** create 1:1 mapping to C interface -> build MPB interface -> instantly available to users
-

MPB-Supported Solver Types

- LP/MILP/MIO
 - Linear and mixed-integer linear optimization
 - QCQP
 - Quadratically-constrained quadratic optimization
 - SDP
 - Semidefinite programming
 - Coming soon: constrained nonlinear, conic
-

Julia makes wrapping C simple

- e.g. Ipopt.jl is a wrapper for the nonlinear interior point solver Ipopt
 - ~370 SLOC including MPB interface
 - Compare with Python interface Pylpopt which ~1300 SLOC
-

Composability of Solvers

- JuliaOpt has solvers for MILP, and **convex** QCQP, but **nonconvex** QCQP is still quite new and quality of solvers is not there yet.
 - JuMP can model them though...
 - Can be solved as sequence of MILPs...
 - MPB enabled creation of solver, Junquo, in ~400 SLOC including MPB interface
-

Other JuliaOpt packages

- `Optim.jl` - pure Julia implementations of the standard algorithms for unconstrained or box constrained problems.
 - `NLopt.jl` - interface to the NLopt constrained solver by Steven G. Johnson
 - `LsqFit.jl` - least-squares fitting functionality, formerly in `Optim.jl`
-

Why Julia?

Generation of Efficient Code

```
@addConstraint(mod, sum{x[i], i=1:9} + s == 10)
# Generates the following code
coefs, vars = Float64[], JuMP.Variable[]
len = length(1:9)
sizehint(coefs, len); sizehint(vars, len)
for i in 1:9
    push!(coefs, 1.0); push!(vars, x[i])
end
push!(coefs, 1.0); push!(vars, s)
addConstraint(mod, coefs, vars, :((==), 1.0)
```

Generation of efficient code

$$x_{i,j} \in \mathbb{Z}^+, \quad i \in \{3, 4, 5\}, \quad j \in \{Red, Blue\}$$

```
@defVar(m, x[3:5, [:red, :blue]] >= 0, Int)
```

- Macro generates custom JuMPDict type with custom Base.getindex etc. to access elements, e.g. getValue(x[4, :blue])
-

Why go to all this trouble?

- Other languages use operator overloading
 - Problem: lots of memory allocation due to all the temporary variables
 - JuMP can JIT-compile-time analyze statements and efficiently generated the sparse data structure
 - As fast as standalone commercial AMLs
-

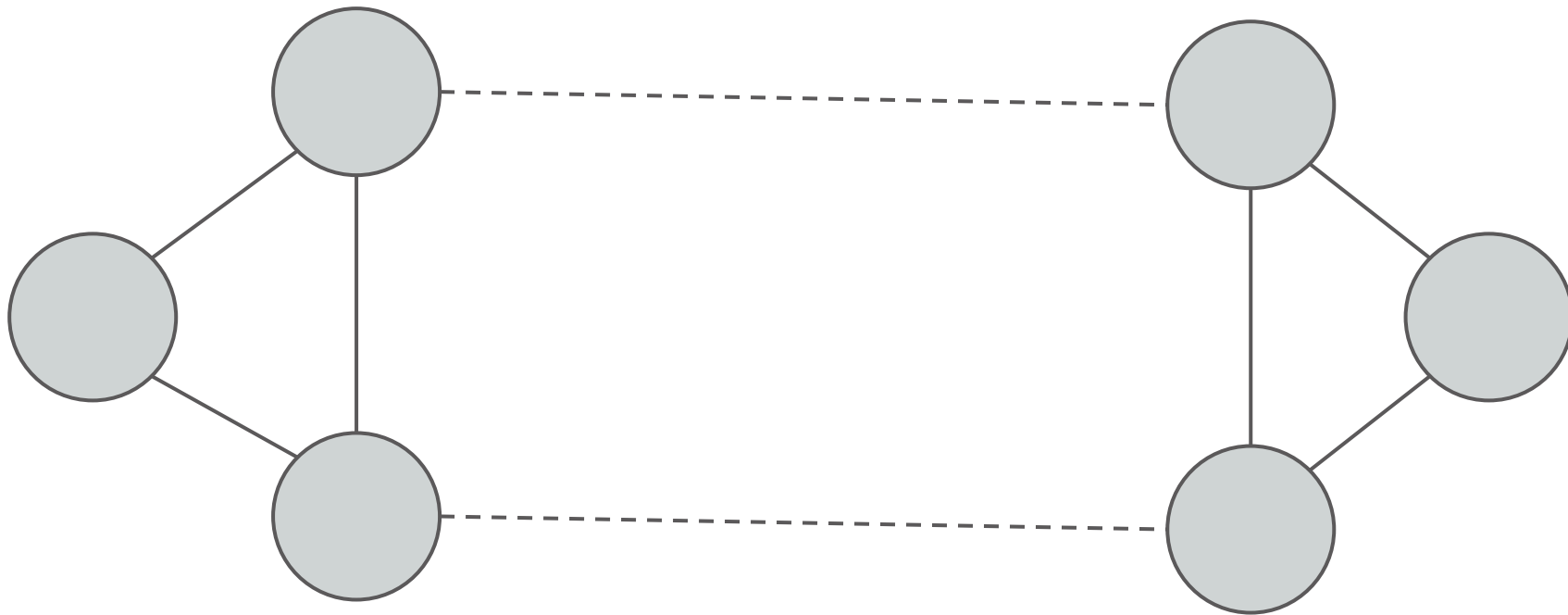
Callbacks

- Solvers allow experts to tinker with internals of optimization process with *callbacks*
 - In e.g. Python, lots of work needs to be done
 - With Julia, it's as simple as `cfunction` and `ccall`
-

Example: Lazy constraints

- Travelling Salesman Problem: find the shortest trip through N cities that visits them all once and ends up where you started.
 - We can write down an optimization problem to solve it, but it has an exponential number of constraints that eliminate subtours.
 - Add as needed: “lazy” constraints
-

Subtours in TSP



Solving TSP with JuMP

- Create JuMP model with only the constraints that ensure each city is visited once
 - Write callback that analyses intermediate solutions, and finds the constraint to eliminate the current incorrect solution
 - Julia advantage: low overhead C interop
 - JuliaOpt/JuMP unique: solver-indep. callbacks
-

Derivatives

- Optimization methods perform better when derivatives available. Two options:

Provide derivatives

- often complex
- error-prone

Estimate derivatives

- inefficient
 - inaccurate
-

- Use type system, multiple dispatch, metaprogramming to get **exact derivatives**
 - `DualNumbers`, `HyperDualNumbers`, `PowerSeries`, ... create new data types that are used in place of e.g. `Float64`
 - `ReverseDiff`[`Sparse`|`Source`|`Overload`] implement automatic differentiation, operate on AST (as does `Calculus`)
-

How are they used?

- Optim and NLSolve use DualNumbers
 - Just set autodiff=true
 - JuMP uses ReverseDiffSparse
 - Compute gradients and sparse Hessians for efficient interior-point solvers
 - MCMC uses ReverseDiffSource
 - Compute gradients of statistical models
-

Who is using JuliaOpt?

- Google Analytics numbers for last 30 days:
 - <http://juliaopt.org> 541
 - JuMP docs 493
 - MathProgBase docs 93
-

What's next for JuliaOpt?

- Constraint programming with [SCIP](#)
 - JuMP still adding features:
 - SDP modeling, presolve, reformulations...
 - DCP via CVX.jl is under development
@Stanford - talk to Madeleine
 - Solvers written in pure Julia, esp. leveraging distributed computing?
-

Problem modification

- Solver-level model persists in memory after optimization
-