Introduction to JuMP

Przemysław Szufel
https://szufel.pl/

Linear optimization

```julia
using JuMP, GLPKMathProgInterface
m =Model(solver = GLPKSolverLP());
@variable(m, x₁ >= 0)
@variable(m, x₂ >= 0)
@objective(m, Min, 50x₁ + 70x₂)
@constraint(m, 200x₁ + 2000x₂ >= 9000)
@constraint(m, 100x₁ +   30x₂ >=  300)
@constraint(m, 9x₁   +   11x₂ >=   60)
solve(m)
JuMP.getvalue.([x₁,x₂])
```

Note – how to type indexes in Julia

- julia> x
- julia> x_
- julia> x_1
- julia> x_1***<TAB>***
- julia> x_1

… and Integer programming

```
using JuMP, GLPKMathProgInterface
m =Model(solver = GLPKSolverMIP());
@variable(m, x₁ >= 0, Int)
@variable(m, x₂ >= 0)
@objective(m, Min, 50x₁ + 70x₂)
@constraint(m, 200x₁ + 2000x₂ >= 9000)
@constraint(m, 100x₁ +   30x₂ >=  300)
@constraint(m, 9x₁   +   11x₂ >=   60)
solve(m)
```

How it works - metaprogramming

```
julia> code = Meta.parse("x=5")
:(x = 5)

julia> dump(code)
Expr
  head: Symbol =
  args: Array{Any}((2,))
    1: Symbol x
    2: Int64 5

julia> eval(code)
5

julia> x
5
```

Macros – hello world...

```
macro sayhello(name)
	return :( println("Hello, ", $name) )
end

julia> macroexpand(Main,:(@sayhello("aa")))
:((Main.println)("Hello, ", "aa"))

julia> @sayhello "world!"
Hello, world!
```

Macro @variable

```
julia> @macroexpand @variable(m, x₁ >= 0)
quote
    (JuMP.validmodel)(m, :m)
    begin
        #1###361 = begin
                let
                    #1###361 = (JuMP.constructvariable!)(m, getfield(JuMP, Symbol("#_error#107")){Tuple{Symbol,Expr}}((:m, :(x₁ >= 0))), 0, Inf, :Default, (JuMP.string)(:x₁), NaN)
                    #1###361
                end
            end
        (JuMP.registervar)(m, :x₁, #1###361)
        x₁ = #1###361
    end
end
```

JuMP Solvers ...

Solver	Julia Package	License	LP	SOCP	MILP	NLP	MINLP	SDP
Artelys Knitro	KNITRO.jl	Comm.				X	X	
BARON	BARON.jl	Comm.				X	X	
Bonmin	AmplNLWriter.jl CoinOptServices.jl	EPL	X		X	X	X	
Cbc	Cbc.jl	EPL			X			
Clp	Clp.jl	EPL	X					
Couenne	AmplNLWriter.jl CoinOptServices.jl	EPL	X		X	X	X	
CPLEX	CPLEX.jl	Comm.	X	X	X			
ECOS	ECOS.jl	GPL	X	X				
FICO Xpress	Xpress.jl	Comm.	X	X	X			
GLPK	GLPKMathProgInterface	GPL	X		X			
Gurobi	Gurobi.jl	Comm.	X	X	X			
Ipopt	Ipopt.jl	EPL	X			X		
MOSEK	Mosek.jl	Comm.	X	X	X	X		X
NLopt	NLopt.jl	LGPL				X		
SCS	SCS.jl	MIT	X	X				X

Why it is fast
Mathematical and symbolic computing

JuliaDiff

Differentiation tools in Julia. JuliaDiff on GitHub.

Stop approximating derivatives!

Derivatives are required at the core of many numerical algorithms. Unfortunately, they are usually computed *inefficiently* and *approximately* by some variant of the finite difference approach

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}, h \text{ small} .$$

This method is *inefficient* because it requires $\Omega(n)$ evaluations of $f : \mathbb{R}^n \to \mathbb{R}$ to compute the gradient $\nabla f(x) = \left(\frac{\partial f}{\partial x_1}(x), \cdots, \frac{\partial f}{\partial x_n}(x)\right)$, for example. It is *approximate* because we have to choose some finite, small value of the step length h, balancing floating-point precision with mathematical approximation error.

What can we do instead?

One option is to explicitly write down a function which computes the exact derivatives by using the rules that we know from Calculus. However, this quickly becomes an error-prone and tedious exercise. **There is another way!** The field of automatic differentiation provides methods for automatically computing *exact* derivatives (up to floating-point error) given only the function f itself. Some methods use many fewer evaluations of f than would be required when using finite differences. In the best case, **the exact gradient of f can be evaluated for the cost of $O(1)$ evaluations of f itself**. The caveat is that f cannot be considered a black box; instead, we require either access to the source code of f or a way to plug in a special type of

Why `JuMP` is fast?

`Calculus.jl` – symbolic differantion

```
julia> using Calculus

julia> differentiate(:(sin(x)))
:(1 * cos(x))

julia> expr = differentiate(:(sin(x) + x*x+5x))
:(1 * cos(x) + (1x + x * 1) + (0x + 5 * 1))

julia> x = 0; eval(expr)
```