Recent improvements to JuMP

Oscar Dowson

EURO 2025

What is JuMP?

An algebraic modeling language in Julia

```julia
using JuMP, Ipopt
function solve_constrained_least_squares_regression(A::Matrix, b::Vector)
    m, n = size(A)
    model = Model(Ipopt.Optimizer)
    @variable(model, -10 <= x[1:n] <= 10)
    @variable(model, residuals[1:m])
    @constraint(model, residuals == A * x - b)
    @constraint(model, sum(x) == 1)
    @objective(model, Min, sum(residuals[i]^2 for i in 1:m))
    optimize!(model)
    return value.(x)
end
x = solve_constrained_least_squares_regression(rand(10, 3), rand(10))
```

What is JuMP?

An algebraic modeling language in Julia

- Under open-source development since 2013
- Supports all major problem types, including MIP, NLP, SDP
- > 50 connected solvers
- > 60 repositories in github.com/jump-dev

Over the last year

- > 10,000 downloads/month
- > 1,000 pull requests
- > 300 issues opened
- > 50 contributors

Improving nonlinear programming support in JuMP

https://jump.dev/JuMP.jl/stable/manual/nonlinear/

```julia
using JuMP, Gurobi
model = Model(Gurobi.Optimizer)
# OR: model = Model(Xpress.Optimizer)
@variable(model, -5 <= x[1:2] <= 5, Int)
@objective(model, Min, x[2]^3 * sin(x[1])^2)
my_func(y) = 2^y[1] + log(sum(exp.(y)))
@constraint(model, 2 * my_func(x) <= 100)
@constraint(model, sqrt(x’ * x) <= 1)
```

Improving nonlinear programming support in JuMP

https://jump.dev/JuMP.jl/stable/manual/nonlinear/

```
using JuMP, Ipopt
model = Model(Ipopt.Optimizer)
backend = MOI.Nonlinear.SparseReverseMode()
# OR: backend = MOI.Nonlinear.SymbolicMode()
set_attribute(model, MOI.AutomaticDifferentiationBackend(), backend)
@variable(model, x[1:2])
@objective(model, Min, (1 - x[1])^2 + 100 * (x[2] - x[1]^2)^2)
optimize!(model)
```

Complex number support

https://jump.dev/JuMP.jl/stable/manual/complex/

```
using JuMP
model = Model()
@variable(model, x in ComplexPlane())
#    real(x) + imag(x) im

@variable(model, y[1:2, 1:2] in HermitianPSDCone())
#    2×2 LinearAlgebra.Hermitian{...}:
#     real(y[1,1])                    real(y[1,2])+imag(y[1,2])*im
#     real(y[1,2])-imag(y[1,2])*im    real(y[2,2])
```

Benoît Legat
blegat

Generic number support

https://jump.dev/JuMP.jl/stable/tutorials/conic/arbitrary_precision/

```
using JuMP, Clarabel
model = GenericModel{BigFloat}(Clarabel.Optimizer{BigFloat})
@variable(model, x[1:2, 1:2] in PSDCone())
@variable(model, t)
y = rand(2, 2)
@constraint(model, [t; vec(x .- y)] in SecondOrderCone())
@objective(model, Min, t)
optimize!(model)
value.(x)  # Returns Vector{BigFloat}
```

Paul Goulart
goulart-paul

Multi-objective support

https://jump.dev/JuMP.jl/stable/tutorials/linear/multi_objective_examples/

```julia
using JuMP, HiGHS
import MultiObjectiveAlgorithms as MOA
model = Model(() -> MOA.Optimizer(HiGHS.Optimizer))
set_attribute(model, MOA.Algorithm(), MOA.Dichotomy())
@variable(model, 0 <= x[1:2] <= 3)
@objective(model, Min, [3x[1] + x[2], -x[1] - 2x[2]])
@constraint(model, 3x[1] - x[2] <= 6)
optimize!(model)
X = [value.(x; result = i) for i in 1:result_count(model)]
```

Gökhan Kof
kofgokhan

XavierG
xgandibleux

MathOptAI.jl

https://lanl-ansi.github.io/MathOptAI.jl/stable/

Robert Parker
Robbybp

Embed machine learning predictors into a JuMP model. Similar to

- OMLT
- gurobi-machinelearning
- PySCIPOpt-ML
- GAMSPy (talk on Tuesday)
- …

```
min f(x, y)
    g(x, y) <= 0
    y = F(x)
```

where F is a neural network/decision tree/logistic regression/…

MathOptAI.jl

https://lanl-ansi.github.io/MathOptAI.jl/stable/

```python
#!/usr/bin/python3
import torch
from torch import nn
model = nn.Sequential(nn.Linear(10, 16), nn.ReLU(), nn.Linear(16, 2))
torch.save(model, "model.pt")
```

MathOptAI.jl

https://lanl-ansi.github.io/MathOptAI.jl/stable/

```julia
#!/usr/bin/julia
using JuMP, Ipopt, MathOptAI, PythonCall
model = Model(Ipopt.Optimizer)
@variable(model, 0 <= x[1:10] <= 1)
predictor = MathOptAI.PytorchModel("model.pt")
y, formulation = MathOptAI.add_predictor(model, predictor, x)
```

Three-ways to formulate a problem

Each with a different trade-off

Parker et al. (2025). Formulations and scalability of neural network surrogates in nonlinear optimization problems

	Full-space	Reduced-space	Gray-box
Pros			
Cons			
Bottleneck			

Full-space

Add intermediate variables and constraints

```
y, _ = MathOptAI.add_predictor(model, predictor, x)
# y = ReLU(x) = max(0, a * x + b)
```

```
model = Model()
@variable(model, x)
@variable(model, tmp[1:2])
@constraint(model, tmp[1] == a * x + b)
@constraint(model, tmp[2] == max(0, tmp[1]))
y = tmp[2]
```

Three-ways to formulate a problem

Each with a different trade-off

Parker et al. (2025). Formulations and scalability of neural network surrogates in nonlinear optimization problems

	Full-space	Reduced-space	Gray-box
Pros	Sparsity Solvers can exploit linearity		
Cons	Many extra variables and constraints		
Bottleneck	Computing linear system because of problem size		

Reduced-space

Represent as nested expressions

```
y, _ = MathOptAI.add_predictor(model, predictor, x; reduced_space = true)
# y = ReLU(x) = max(0, a * x + b)

model = Model()
@variable(model, x)
y = @expression(model, max(0, a * x + b))
```

Three-ways to formulate a problem

Each with a different trade-off

Parker et al. (2025). Formulations and scalability of neural network surrogates in nonlinear optimization problems

	Full-space	Reduced-space	Gray-box
Pros	Sparsity Solvers can exploit linearity	Fewer variables and constraints	
Cons	Many extra variables and constraints	Complicated dense expressions	
Bottleneck	Computing linear system because of problem size	Computing derivatives (JuMP's AD does not do well at dense problems)	

Gray-box

Use external function evaluation

```julia
y, _ = MathOptAI.add_predictor(model, predictor, x; gray_box = true)
# y = ReLU(x) = max(0, a * x + b)
```

```julia
fn(x) = max(0, a * x + b)
fn_dx(x) = ifelse(a * x + b >= 0, a, 0)
model = Model()
@variable(model, x)
@operator(model, op_gray_box, 1, fn, fn_dx)
y = @expression(model, op_gray_box(x))
```

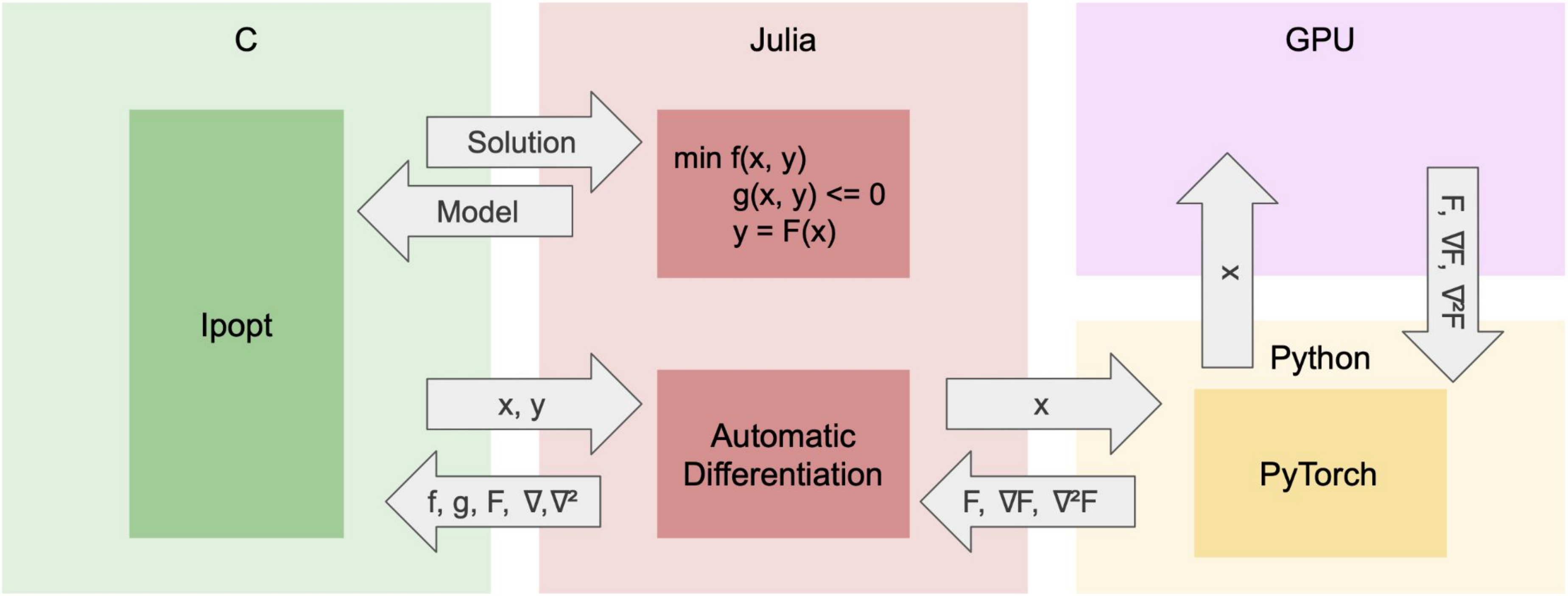
Gray-box: Julia, C, Python, working together

JuMP problems call Ipopt in C, which calls back to Julia for oracles, which calls Python and PyTorch

```julia
#!/usr/bin/julia
using JuMP, Ipopt, MathOptAI, PythonCall
model = Model(Ipopt.Optimizer)
@variable(model, 0 <= x[1:10] <= 1)
predictor = MathOptAI.PytorchModel("model.pt")
y, _ = MathOptAI.add_predictor(model, predictor, x; gray_box = true,
                                  device = "cuda", hessian = true)
```

Gray-box: Julia, C, Python, working together

JuMP problems call Ipopt in C, which calls back to Julia for oracles, which calls Python and PyTorch



Three-ways to formulate a problem

Each with a different trade-off

Parker et al. (2025). Formulations and scalability of neural network surrogates in nonlinear optimization problems

	Full-space	Reduced-space	Gray-box
Pros	Sparsity Solvers can exploit linearity	Fewer variables and constraints	Can use external evaluation for oracles. Scales with input/output dimension, not intermediate dimension
Cons	Many extra variables and constraints	Complicated dense expressions	Requires oracle-based NLP. Cannot be used by global MINLP solvers
Bottleneck	Computing linear system because of problem size	Computing derivatives (JuMP's AD does not do well at dense problems)	Moving data between Julia/Python/GPU

JuMP-dev 2025

November 17–20

Auckland, New Zealand

Three-ways to formulate a problem

Time/iteration for Ipopt

Parker et al. (2025). Formulations and scalability of neural network surrogates in nonlinear optimization problems

# Parameters	Full-space	Reduced-space	Gray-box
7 thousand	4 ms	100 ms	7 ms
25 thousand	8 ms	27,000 ms	7 ms
578 thousand	900 ms	-	8 ms
592 million	-	-	23 ms

Gray-box: Julia, C, Python, working together

JuMP problems call Ipopt in C, which calls back to Julia for oracles, which calls Python and PyTorch