
# MathOptAI.jl

**Oscar Dowson**

Robert Parker, Nicole LoGiudice, Manuel Garcia, Kaarthik Sundar, Russell Bent

SIAM OP 2026

# MathOptAI.jl

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

## Mission

Embed machine learning predictors into a JuMP model.

## Similar to

- OMLT
- gurobi-machinelearning
- PySCIPOpt-ML
- GAMSPy
- …

## Problem class

$$\begin{aligned} \min\ & f_0(x, y) \\ & f_i(x, y) \in S_i \quad \forall i \\ & y = F(x) \end{aligned}$$

where $F$ is a neural network, decision tree, logistic regression,…

# Application Example

## Stability Constrained Optimal Power Flow

$$\begin{aligned} &\min f_0(x) && \text{[total cost]} \\ &\quad f_i(x) \in S_i \quad \forall i && \text{[physics: AC-OPF]} \\ &\quad y = F(x) && \text{[NN: is network stable?]} \\ &\quad y \geq 0.95 && \text{[high likelihood]} \end{aligned}$$

# Code Example

## Embed a NN from Pytorch in JuMP

```
#!/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")
```

# Code Example

## Embed a NN from Pytorch in JuMP

```
#!/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")

#!/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)
@constraint(model, y .>= 0.9)
```

# Code Example

## Embed a NN from Pytorch in JuMP

```
#!/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")

#!/usr/bin/julia
using JuMP, HiGHS, MathOptAI, PythonCall
model = Model(HiGHS.Optimizer)
@variable(model, 0 <= x[1:10] <= 1)
predictor = MathOptAI.PytorchModel("model.pt")
config = Dict(:ReLU => MathOptAI.ReLUSOS1())
y, formulation = MathOptAI.add_predictor(model, predictor, x; config)
@constraint(model, y .>= 0.9)
```

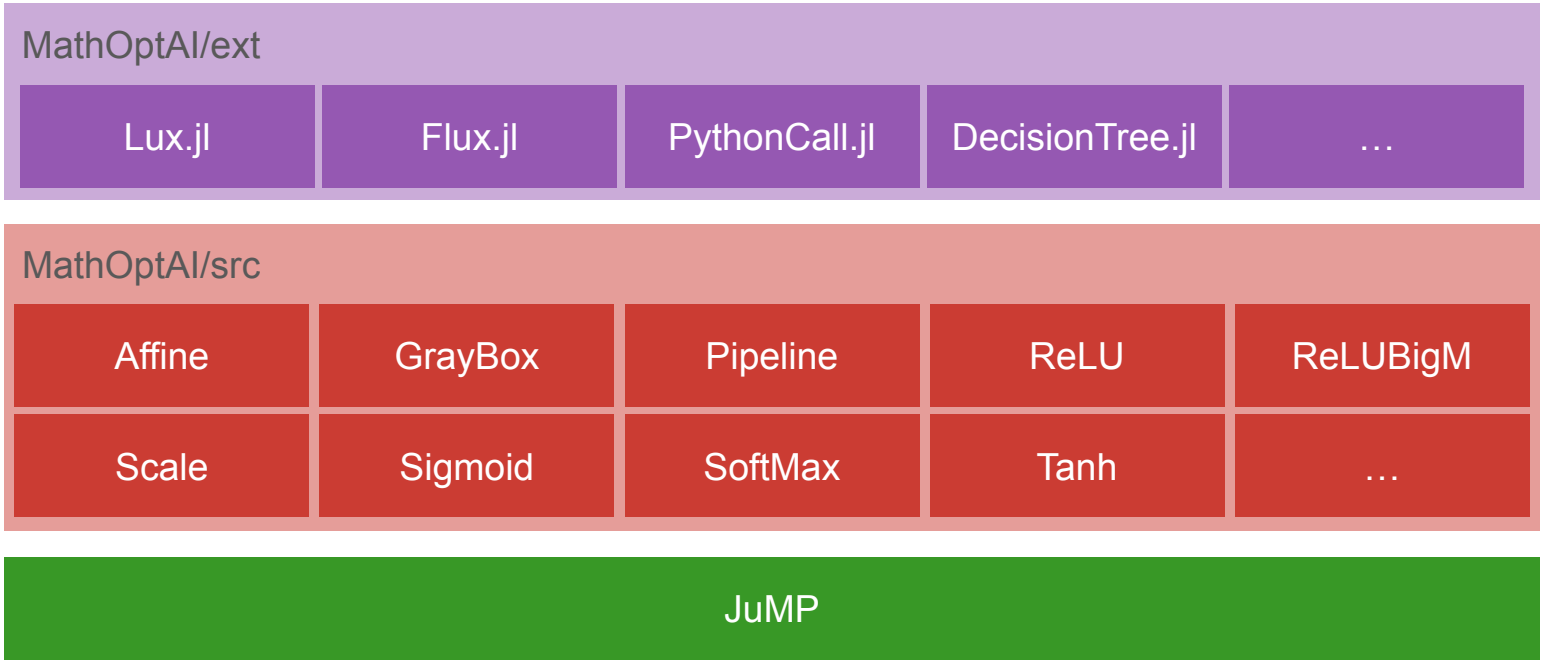
# Code Example

## Embed a NN from Flux.jl in JuMP

```
#!/usr/bin/julia
using JuMP, MathOptAI, Flux

predictor = Flux.Chain(
   Flux.Dense(28^2 => 32, Flux.sigmoid),
   Flux.Dense(32 => 10),
   Flux.softmax,
)

model = Model()
@variable(model, 0 <= x[1:28^2] <= 1)
y, formulation = MathOptAI.add_predictor(model, predictor, x)
```

# MathOptAI sits on top of JuMP

## We implement many package extensions



# AbstractPredictors and package extensions

## The Affine predictor

```julia
# src/predictors/Affine.jl
struct Affine{T} <: AbstractPredictor
    A::Matrix{T}
    b::Vector{T}
end

function add_predictor(model::JuMP.AbstractModel, predictor::Affine, x::Vector)
    m = size(predictor.A, 1)
    y = JuMP.@variable(model, [1:m], base_name = "moai_Affine")
    cons = JuMP.@constraint(model, predictor.A * x .+ predictor.b .== y)
    return y, Formulation(predictor, y, cons)
end
```

# AbstractPredictors and package extensions

## The GLM package extension

```julia
# ext/MathOptAIGLMExt.jl
function MathOptAI.build_predictor(predictor::GLM.LinearModel)
    return MathOptAI.Affine(GLM.coef(predictor))
end

# src/MathOptAI.jl
function add_predictor(
    model::JuMP.AbstractModel,
    predictor::Any,
    x::Vector;
    kwargs...,
)
    inner_predictor = build_predictor(predictor; kwargs...)
    return add_predictor(model, inner_predictor, x)
end
```

# Three-ways to formulate a problem

## Each with a different trade-off

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

# Full-space

## Add intermediate variables and constraints

```julia
using JuMP, MathOptAI
# y = ReLU(x) = max.(0, A * x + b)
predictor = MathOptAI.Pipeline(
    MathOptAI.Affine(A, b),
    MathOptAI.ReLU(),
)
model = Model()
@variable(model, x[1:n])
y, _ = MathOptAI.add_predictor(
    model,
    predictor,
    x,
)
```

```julia
using JuMP
model = Model()
@variables(model, begin
    x[1:n]
    tmp[1:m]
    y[1:m]
end)
@constraints(model, begin
    tmp == A * x + b
    y .== max.(0, tmp)
end)
```

# Three-ways to formulate a problem

## Each with a different trade-off

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

# Reduced-space

## Use nested expressions

```julia
using JuMP, MathOptAI
# y = ReLU(x) = max.(0, A * x + b)
predictor = MathOptAI.Pipeline(
    MathOptAI.Affine(A, b),
    MathOptAI.ReLU(),
)
model = Model()
@variable(model, x[1:n])
y, _ = MathOptAI.add_predictor(
    model,
    predictor,
    x;
    reduced_space = true,
)
```

```julia
using JuMP
model = Model()
@variables(model, begin
    x[1:n]
end)
@expressions(model, begin
    tmp, A * x + b
    y, max.(0, tmp)
end)
```

# Three-ways to formulate a problem

## Each with a different trade-off

| | Full-space | Reduced-space | Gray-box |
|---|---|---|---|
| Pros | Sparsity<br><br>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
using JuMP, MathOptAI
# y = ReLU(x) = max.(0, A * x + b)
predictor = MathOptAI.Pipeline(
    MathOptAI.Affine(A, b),
    MathOptAI.ReLU(),
)
model = Model()
@variable(model, x[1:n])
y, _ = MathOptAI.add_predictor(
    model,
    predictor,
    x;
    gray_box = true,
)
```

```julia
using JuMP
model = Model()
@variables(model, begin
    x[1:n]
    y[1:m]
end)
set = MOI.VectorNonlinearOracle(
    # g(x) := F(x) - y
    # evaluate g(x), ∇g(x)
)
@constraints(model, begin
    [x, y] in set
end)
```

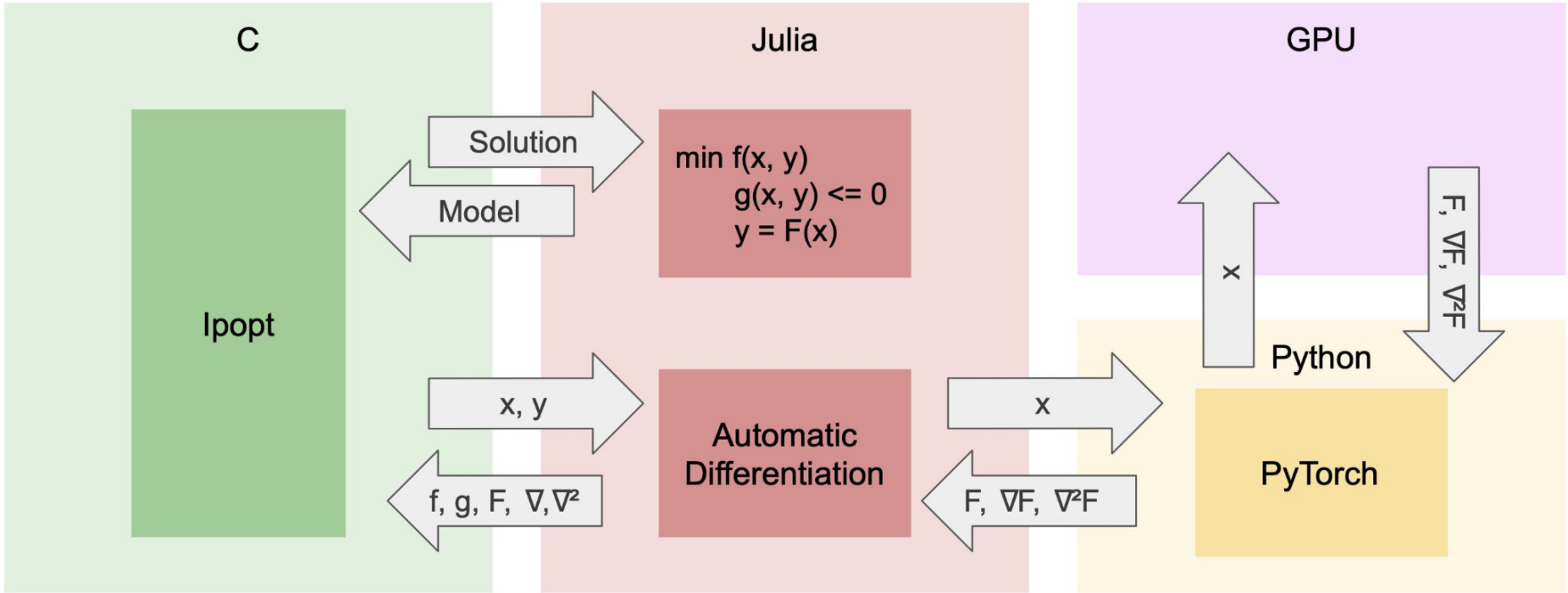
# Gray-box oracles are evaluated in Pytorch

## MathOptAI automatically sets up the Julia-Python intercommunication

```
#!/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

| | Full-space | Reduced-space | Gray-box |
|---|---|---|---|
| Pros | Sparsity<br><br>Solvers can exploit linearity | Fewer variables and constraints | Can use external evaluation for oracles.<br><br>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 |

# Three-ways to formulate a problem

## Solve time [s] for a model with varying # parameters in the neural network

| Parameters | Full-space | Reduced-space | Gray-box (CPU) | Gray-box (GPU) |
|---|---|---|---|---|
| 7k | 2 | 7 | 8 | 7 |
| 25k | 5 | 1125 | 9 | 7 |
| 578k | 699 | - | 11 | 8 |
| 7M | - | - | 22 | 8 |
| 68M | - | - | 140 | 9 |
| 592M | - | - | 748 | 15 |

**Takeaway:** embedding very large NN's into a model is tractable with gray-box

# Reasons to use MathOptAI

## Here are four

### Great syntax

```
y, formulation =
    MathOptAI.add_predictor(
        model,
        predictor,
        x,
    )
```

### Gray-box support

- Supports arbitrary size models
- Supports layers that cannot easily be modeled algebraically

### Wide range of formulations

JuMP is great for this

- LP
- MIP
- Global NLP
- Oracle NLP

### Easy to add new predictors

- The literature is open and growing
- If you have an idea, you can implement it in MathOptAI.jl

# Learn more

## https://github.com/lanl-ansi/MathOptAI.jl

- Dowson, Parker, and Bent (2026). MathOptAI.jl: Embed Trained Machine-Learning Predictors into JuMP Models. *INFORMS Journal on Computing*. https://doi.org/10.1287/ijoc.2025.1446

- Parker et al. (2025). Nonlinear Optimization with GPU-accelerated neural network constraints. https://arxiv.org/pdf/2509.22462

- Parker, Garcia, and Bent (2026). Exploiting block triangular submatrices in KKT systems. https://arxiv.org/pdf/2602.17968

- Parker (2026). Generating adversarial inputs for a graph neural network model of AC power flow. https://arxiv.org/pdf/2602.17975