Introduction. Julia is a language meant for working with data. In that regard, we would like to be able to be able to perform analyses like linear regression and hypothesis testing. We would also like to perform more complex operations including building neural networks, decision trees, and more modern methods in general. Thankfully, there are packages for this.

In this presentation, I will be focusing on three different packages; MLJ.jl, Flux.jl, and Graphs.jl.

MLJ.jl and Flux.jl are both machine learning frameworks, while Graphs.jl is meant for network analysis.

MLJ is a machine learning package that is similar to Python's sci-kitlearn, due to its breadth and flexibility in performing different kinds of analyses.

```
using Pkg
Pkg.add(["MLJ", "MLJLinearModels", "DataFrames", "StatsBase"])
```

```
   Resolving package versions...
  No Changes to `C:\Users\joshu\.julia\environments\v1.11\Project.toml`
  No Changes to `C:\Users\joshu\.julia\environments\v1.11\Manifest.toml`
```

There are numerous models we can configure and implement for many different machine learning problems, as shown by listing out the possible models.

```
using MLJ
models()
```

```
234-element Vector{NamedTuple{(:name, :package_name, :is_supervised, :abstract_type, :constru
 (name = ABODDetector, package_name = OutlierDetectionNeighbors, ... )
 (name = ABODDetector, package_name = OutlierDetectionPython, ... )
 (name = ARDRegressor, package_name = MLJScikitLearnInterface, ... )
 (name = AdaBoostClassifier, package_name = MLJScikitLearnInterface, ... )
 (name = AdaBoostRegressor, package_name = MLJScikitLearnInterface, ... )
 (name = AdaBoostStumpClassifier, package_name = DecisionTree, ... )
 (name = AffinityPropagation, package_name = Clustering, ... )
 (name = AffinityPropagation, package_name = MLJScikitLearnInterface, ... )
 (name = AgglomerativeClustering, package_name = MLJScikitLearnInterface, ... )
 (name = AutoEncoder, package_name = BetaML, ... )
 (name = BM25Transformer, package_name = MLJText, ... )
 (name = BaggingClassifier, package_name = MLJScikitLearnInterface, ... )
 (name = BaggingRegressor, package_name = MLJScikitLearnInterface, ... )

 (name = TSVDTransformer, package_name = TSVD, ... )
```

```
 (name = TfidfTransformer, package_name = MLJText, ... )
 (name = TheilSenRegressor, package_name = MLJScikitLearnInterface, ... )
 (name = TomekUndersampler, package_name = Imbalance, ... )
 (name = UnivariateBoxCoxTransformer, package_name = MLJModels, ... )
 (name = UnivariateDiscretizer, package_name = MLJModels, ... )
 (name = UnivariateFillImputer, package_name = MLJModels, ... )
 (name = UnivariateStandardizer, package_name = MLJModels, ... )
 (name = UnivariateTimeTypeToContinuous, package_name = MLJModels, ... )
 (name = XGBoostClassifier, package_name = XGBoost, ... )
 (name = XGBoostCount, package_name = XGBoost, ... )
 (name = XGBoostRegressor, package_name = XGBoost, ... )
```

One example that we can look at is implementing linear regression on the Boston housing dataset. We would like to model the prices of properties as functions of the other variables.

We start by loading the dataset and then dividing the data into training and testing data. This can be done without any external tools.

```
using MLJ, DataFrames, MLJLinearModels, StatsBase

# Load dataset
X, y = @load_boston

# Convert X to a DataFrame
X = DataFrame(X)

# Split data into training (80%) and test (20%) sets
train, test = partition(eachindex(y), 0.8, shuffle=true)
X_train, X_test = X[train, :], X[test, :]
y_train, y_test = y[train], y[test]
```

```
([20.6, 20.6, 14.6, 19.5, 19.8, 29.8, 19.3, 23.0, 26.7, 18.4  …  27.9, 7.0, 30.8, 21.7, 11.9
```

Now, we load an MLJ model. Specifically, we will use LinearRegressor, which is a simple linear regression model. MLJ offers other alternatives, including RidgeRegressor and LassoRegressor, as well as linear model classifiers like LogisticClassifier.

```
# Load Linear Regression model
@load LinearRegressor pkg=MLJLinearModels
model = LinearRegressor()

# Create a machine (model + data)
```

```
mach = machine(model, X_train, y_train)

# Train the model
fit!(mach)
```

```
[ Info: For silent loading, specify `verbosity=0`.

import MLJLinearModels

[ Info: Training machine(LinearRegressor(fit_intercept = true, …), …).
 Info: Solver: Analytical
   iterative: Bool false
   max_inner: Int64 200

trained Machine; caches model-specific representations of data
  model: LinearRegressor(fit_intercept = true, …)
  args:
    1:  Source @171   Table{AbstractVector{Continuous}}
    2:  Source @505   AbstractVector{Continuous}
```

After fitting the model, we can now make predictions to see how accurate our model is! We can use the StatsBase package to calculate metrics like R-squared and MAE.

```
# Make predictions
y_pred = MLJ.predict(mach, X_test)

# Convert predictions to raw values
y_pred_numeric = mode.(y_pred)

# Compute R² and MAE
r2_score = cor(y_pred_numeric, y_test)^2  # R² is squared correlation
mae_score = mean(abs.(y_pred_numeric - y_test))  # Mean Absolute Error

# Print results
println("R² Score: ", r2_score)
println("MAE: ", mae_score)
```

```
R² Score: 0.7724035562296963
MAE: 3.2788422179621817
```

On the other hand, if you are interested in solving problems with deep learning techniques, Flux.jl is a popular option for that. Flux is a deep learning framework that has a lot of the same functionalities as MLJ, but is more focused on deep learning and allowing the user to control those areas of computation.

We can take a look at Flux by building a neural network to learn classification on the MNIST dataset.

```
Pkg.add(["Flux", "MLDatasets", "Statistics"])
```

```
   Resolving package versions...
  No Changes to `C:\Users\joshu\.julia\environments\v1.11\Project.toml`
  No Changes to `C:\Users\joshu\.julia\environments\v1.11\Manifest.toml`
```

```
using Flux
using Flux: onehotbatch, onecold, crossentropy, params, DataLoader
using MLDatasets: MNIST
using Statistics
```

First we need to load the MNIST dataset.

```
# Load MNIST dataset
function get_data()
    train_x, train_y = MNIST.traindata()
    test_x, test_y = MNIST.testdata()

    # Reshape images to 28x28 and normalize pixel values to [0,1]
    train_x = reshape(Float32.(train_x) ./ 255, 28*28, :)
    test_x = reshape(Float32.(test_x) ./ 255, 28*28, :)

    # Convert labels to one-hot encoding
    train_y = onehotbatch(train_y, 0:9)
    test_y = onehotbatch(test_y, 0:9)

    return train_x, train_y, test_x, test_y
end

train_x, train_y, test_x, test_y = get_data()
```

```
  Warning: MNIST.traindata() is deprecated, use `MNIST(split=:train)[:]` instead.
  @ MLDatasets C:\Users\joshu\.julia\packages\MLDatasets\0MkOE\src\datasets\vision\mnist.jl:1
  Warning: MNIST.testdata() is deprecated, use `MNIST(split=:test)[:]` instead.
  @ MLDatasets C:\Users\joshu\.julia\packages\MLDatasets\0MkOE\src\datasets\vision\mnist.jl:1
```

```
(Float32[0.0 0.0 … 0.0 0.0; 0.0 0.0 … 0.0 0.0; … ; 0.0 0.0 … 0.0 0.0; 0.0 0.0 … 0.0 0.0], Bo
```

Defining the actual neural network structure is very simple.

```
# Define the neural network
model = Chain(
    Dense(28*28, 128, relu),
    Dense(128, 64, relu),
    Dense(64, 10),
    softmax
)
```

```
Chain(
  Dense(784 => 128, relu),              # 100_480 parameters
  Dense(128 => 64, relu),               # 8_256 parameters
  Dense(64 => 10),                      # 650 parameters
  NNlib.softmax,
)                   # Total: 6 arrays, 109_386 parameters, 427.594 KiB.
```

We next define the loss function and the optimization function.

```
# Loss function
loss(x, y) = crossentropy(model(x), y)

# Define optimizer with new Flux API
optim_state = Flux.setup(ADAM(0.001), model)
```

```
(layers = ((weight = Leaf(Adam(eta=0.001, beta=(0.9, 0.999), epsilon=1.0e-8), (Float32[0.0 0
```

Then we train the model.

```
# Training loop
function train_model(; epochs=10, batch_size=64)
    train_loader = DataLoader((train_x, train_y), batchsize=batch_size, shuffle=true)

    for epoch in 1:epochs
        loss_total = 0.0
        for (x, y) in train_loader
            # Ensure x and y are Float32 for Flux computations
            x, y = Float32.(x), Float32.(y)
```

```
            # Compute loss and gradients correctly
            loss_value, grads = Flux.withgradient(model) do m
                crossentropy(m(x), y)
            end

            # Update model parameters with the optimizer state
            Flux.update!(optim_state, model, grads[1])  # No need for `grads[1]`

            loss_total += loss_value
        end
        println("Epoch $epoch, Loss: $(loss_total/length(train_loader))")
    end
end

# Train the model
train_model(epochs=10, batch_size=64)
```

```
Epoch 1, Loss: 1.0302363859056665
Epoch 2, Loss: 0.437749562963748
Epoch 3, Loss: 0.3590324820517731
Epoch 4, Loss: 0.3246071812297617
Epoch 5, Loss: 0.30028443229096785
Epoch 6, Loss: 0.27842386137606745
Epoch 7, Loss: 0.2577773577678623
Epoch 8, Loss: 0.23745311343117054
Epoch 9, Loss: 0.21835283804406871
Epoch 10, Loss: 0.20103652156524057
```

Now we can see how our basic neural network performs on the test set!

```
# Evaluate accuracy
function accuracy(x, y)
    predictions = onecold(model(x), 0:9)
    labels = onecold(y, 0:9)
    return mean(predictions .== labels)
end

println("Test Accuracy: ", accuracy(test_x, test_y))
```

```
Test Accuracy: 0.9426
```

The accuracy is almost 95%, which is not bad. Maybe we think that we can do better if we change the neural network structure. Flux allows us the ability to create our own layers for our model! We can try that to see if we get higher accuracy.

```
# Custom Fully Connected Layer
struct MyDense
    W::Matrix{Float32}  # Weights
    b::Vector{Float32}  # Bias
    activation::Function  # Activation function
end

# Initialize MyDense layer
function MyDense(in_dim::Int, out_dim::Int, activation=relu)
    W = randn(Float32, out_dim, in_dim) * 0.1  # Random weights
    b = zeros(Float32, out_dim)  # Zero biases
    return MyDense(W, b, activation)
end

# Define forward pass for MyDense
function (layer::MyDense)(x)
    return layer.activation.(layer.W * x .+ layer.b)
end
```

```
# Define Neural Network using Custom Layers
model = Chain(
    MyDense(28*28, 128, relu),  # Custom fully connected layer
    MyDense(128, 64, relu),     # Another custom dense layer
    MyDense(64, 10, identity),  # Output layer
    softmax                     # Apply softmax for classification
)
```

```
Chain(
  MyDense(Float32[-0.12807706 -0.09489949 … -0.040312473 -0.12375561; -0.2802467 0.2009633 …
  MyDense(Float32[-0.009126644 0.03726612 … -0.14118257 0.23216943; -0.21537426 0.020078344 .
  MyDense(Float32[-0.036260467 0.07575139 … 0.09142425 0.059532307; 0.039816283 0.19488889 …
  NNlib.softmax,
)                   # Total: 6 arrays, 109_386 parameters, 427.617 KiB.
```

```
# Loss function
loss(x, y) = crossentropy(model(x), y)

# Define optimizer with new Flux API
```

```julia
optim_state = Flux.setup(ADAM(0.001), model)

# Training loop
function train_model(; epochs=10, batch_size=64)
    train_loader = DataLoader((train_x, train_y), batchsize=batch_size, shuffle=true)

    for epoch in 1:epochs
        loss_total = 0.0
        for (x, y) in train_loader
            # Ensure x and y are Float32 for Flux computations
            x, y = Float32.(x), Float32.(y)

            # Compute loss and gradients correctly
            loss_value, grads = Flux.withgradient(model) do m
                crossentropy(m(x), y)
            end

            # Update model parameters with the optimizer state
            Flux.update!(optim_state, model, grads[1])  # No need for `grads[1]`

            loss_total += loss_value
        end
        println("Epoch $epoch, Loss: $(loss_total/length(train_loader))")
    end
end


# Train the model
train_model(epochs=10, batch_size=64)
```

```
Epoch 1, Loss: 1.0133030599813218
Epoch 2, Loss: 0.45638008480832015
Epoch 3, Loss: 0.3722419646630155
Epoch 4, Loss: 0.3343864954404358
Epoch 5, Loss: 0.30873031227954667
Epoch 6, Loss: 0.28864312967829614
Epoch 7, Loss: 0.26895526032457984
Epoch 8, Loss: 0.2517742814301555
Epoch 9, Loss: 0.2351082132569254
Epoch 10, Loss: 0.21774770665381635
```

```julia
# Evaluate accuracy
function accuracy(x, y)
    predictions = onecold(model(x), 0:9)
    labels = onecold(y, 0:9)
    return mean(predictions .== labels)
end

println("Test Accuracy: ", accuracy(test_x, test_y))
```

```
Test Accuracy: 0.9384
```

Unfortunately, we did not improve our accuracy, but this feature showcases Flux's flexibility in model construction.

It is also worth mentioning that Flux supports GPU acceleration with CUDA. These many features mean that Flux is a very versatile package that is useful for deep learning while also allowing the user to maintain control over their research/development. As a relatively easy-to-use package, Flux is one of the top choices for deep learning in Julia.

MLJ, on the other hand, is more suited towards high-level implementations. It supports diverse approaches to problems, and allows users to pull together and integrate multiple models, without requiring low-level mastery. However, it does not inherently support GPU acceleration, which makes it slower when running large tasks.

The last two packages I want to go over are Graphs.jl and (INSERT SOME LINEAR PROGRAMMING PACKAGE). While MLJ and Flux are meant for machine learning, there may be times when a user is interested in implementing optimization algorithms, like Dijkstra's or Simplex. These algorithms are of a different flavor as they are not necessarily meant to solve prediction or inference problems, but are many times useful for finding optimal paths or augmentations for a specific task.

We can see how Graphs.jl can be useful through an implementation of Dijkstra's Algorithm.

Below is a raw implementation of the algorithm on a simple graph, with no external packages being used.

```julia
function dijkstra(graph::Dict{Int, Dict{Int, Int}}, start::Int)
    # Initialize distances to infinity, except for the start node
    dist = Dict(k => Inf for k in keys(graph))
    dist[start] = 0

    # Track visited nodes
    visited = Dict(k => false for k in keys(graph))
```

```
    # Track shortest paths
    prev = Dict{Int, Union{Nothing, Int}}(k => nothing for k in keys(graph))

    while true
        # Select the unvisited node with the smallest distance
        u = nothing
        for node in keys(graph)
            if !visited[node] && (u === nothing || dist[node] < dist[u])
                u = node
            end
        end

        # Stop if there are no reachable unvisited nodes
        if u === nothing || dist[u] == Inf
            break
        end

        # Mark node as visited
        visited[u] = true

        # Update distances to neighbors
        for (v, weight) in graph[u]
            alt = dist[u] + weight
            if alt < dist[v]
                dist[v] = alt
                prev[v] = u
            end
        end
    end

    return dist, prev
end

# Function to reconstruct the shortest path
function shortest_path(prev::Dict{Int, Union{Nothing, Int}}, target::Int)
    path = []
    while target !== nothing
        pushfirst!(path, target)
        target = prev[target]
    end
    return path
end
```

```julia
# Define a weighted graph using a properly typed dictionary
graph = Dict{Int, Dict{Int, Int}}(
    1 => Dict(2 => 4, 3 => 1),
    2 => Dict(4 => 1),
    3 => Dict(2 => 2, 4 => 5),
    4 => Dict(5 => 3),
    5 => Dict()
)

# Run Dijkstra's algorithm from node 1
distances, predecessors = dijkstra(graph, 1)

# Print shortest distance from node 1 to 5
println("Shortest distance from 1 to 5: ", distances[5])

# Get the shortest path from node 1 to 5
path = shortest_path(predecessors, 5)
println("Shortest path from 1 to 5: ", path)
```

```
Shortest distance from 1 to 5: 7.0
Shortest path from 1 to 5: Any[1, 3, 2, 4, 5]
```

The code is quite bulky and requires a lot of work to develop. By using the Graphs package, we can implement a shorter, more elegant solution to the same problem.

```julia
# Install required packages
using Pkg
Pkg.add("Graphs")
# Load the package
using Graphs
```

```
   Resolving package versions...
  No Changes to `C:\Users\joshu\.julia\environments\v1.11\Project.toml`
  No Changes to `C:\Users\joshu\.julia\environments\v1.11\Manifest.toml`
```

```julia
# Create a directed graph with 5 nodes
g = SimpleDiGraph(5)

# Define weighted edges
edges = [
```

```
    (1, 2, 4), (1, 3, 1),
    (3, 2, 2), (3, 4, 5),
    (2, 4, 1), (4, 5, 3)
]

# Add edges to the graph
for (u, v, _) in edges
    add_edge!(g, u, v)  # Add edge
end

# Create a weight matrix (initialize with Inf)
weights = fill(Inf, nv(g), nv(g))

# Assign weights to the adjacency matrix
for (u, v, w) in edges
    weights[u, v] = w
end

# Run Dijkstra's Algorithm from node 1
result = dijkstra_shortest_paths(g, 1, weights)

# Get the shortest path distance to node 5
println("Shortest distance from node 1 to 5: ", result.dists[5])

# Retrieve the shortest path to node 5
function get_path(result, target)
    path = []
    while target != 0
        pushfirst!(path, target)
        target = result.parents[target]
    end
    return path
end

best_path = get_path(result, 5)
println("Shortest path from node 1 to 5: ", best_path)
```

```
Shortest distance from node 1 to 5: 7.0
Shortest path from node 1 to 5: Any[1, 3, 2, 4, 5]
```

We can see that the Graphs package is quite useful for the representation of non-numeric

objects. Networks are the foundation of many important problems, and combining them with the numerical computing power of Julia can help to solve many problems.

Finally, we will go over a mathematical optimization framework called JuMP.jl, as well as a mathematical solver package called Gurobi. Mathematical optimization is at the core of many different problems like portfolio optimization, control theory, and other various fields. Being able to solve optimization problems as fast as possible carries a lot of importance in fields like supply chain logistics, asset allocation, etc.

JuMP works within Julia but acts like its own language. By defining objects like objective functions and constraints, JuMP becomes a versatile tool for constructing and solving problems like Linear Programs, Quadratic Programs, and Mixed Integer Optimization problems.

On the flip side, solvers like Gurobi are the workhorses for solving these problems. Gurobi in particular is the world's fastest mathematical solver, due to its ability to adjust its solving strategies to each problem as well as the extreme level of care put into optimizing each process through parallel computing and fast code.

Gurobi is also free for academics!

```
Pkg.add("JuMP")    # Install JuMP
Pkg.add("Gurobi")  # Install Gurobi wrapper
```

```
   Resolving package versions...
  No Changes to `C:\Users\joshu\.julia\environments\v1.11\Project.toml`
  No Changes to `C:\Users\joshu\.julia\environments\v1.11\Manifest.toml`
   Resolving package versions...
  No Changes to `C:\Users\joshu\.julia\environments\v1.11\Project.toml`
  No Changes to `C:\Users\joshu\.julia\environments\v1.11\Manifest.toml`
```

```
using JuMP, Gurobi
env = Gurobi.Env()
println("Gurobi version: ", env)
```

```
Set parameter Username
Set parameter LicenseID to value 2630108
Academic license - for non-commercial use only - expires 2026-03-02
Gurobi version:

WARNING: using JuMP.mode in module Main conflicts with an existing identifier.

Gurobi.Env(Ptr{Nothing} @0x000002af5e129dd0, false, 0)
```

```
# Define a JuMP model with Gurobi as the solver
model = JuMP.Model(Gurobi.Optimizer)
```

```
Set parameter Username
Set parameter LicenseID to value 2630108
Academic license - for non-commercial use only - expires 2026-03-02

A JuMP Model
  solver: Gurobi
  objective_sense: FEASIBILITY_SENSE
  num_variables: 0
  num_constraints: 0
  Names registered in the model: none
```

Let's say that we have a problem of the form:

$$\max_{x_1, x_2} 5x_1 + 4x_2$$

With the constraints:

$$6x_1 + 4x_2 \leq 24$$
$$x_1 + 2x_2 \leq 6$$
$$x_1, x_2 \geq 0$$

We can define this problem with JuMP and subsequently solve!

```
# Define variables (x1, x2 >= 0)
@variable(model, x1 >= 0)
@variable(model, x2 >= 0)

# Objective function: Maximize 5x1 + 4x2
@objective(model, Max, 5x1 + 4x2)

# Constraints
@constraint(model, 6x1 + 4x2 <= 24)
@constraint(model, x1 + 2x2 <= 6)

# Solve the optimization problem
optimize!(model)
```

```
# Display results
println("Optimal x1 = ", value(x1))
println("Optimal x2 = ", value(x2))
println("Optimal objective value = ", objective_value(model))
```

```
Gurobi Optimizer version 12.0.1 build v12.0.1rc0 (win64 - Windows 11.0 (26100.2))

CPU model: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz, instruction set [SSE2|AVX|AVX2|AV
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 2 rows, 2 columns and 4 nonzeros
Model fingerprint: 0x9f631cdb
Coefficient statistics:
  Matrix range     [1e+00, 6e+00]
  Objective range  [4e+00, 5e+00]
  Bounds range     [0e+00, 0e+00]
  RHS range        [6e+00, 2e+01]
Presolve time: 0.01s
Presolved: 2 rows, 2 columns, 4 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    9.0000000e+30   2.750000e+30   9.000000e+00      0s
       2    2.1000000e+01   0.000000e+00   0.000000e+00      0s

Solved in 2 iterations and 0.01 seconds (0.00 work units)
Optimal objective  2.100000000e+01

User-callback calls 47, time in user-callback 0.01 sec
Optimal x1 = 3.0
Optimal x2 = 1.5
Optimal objective value = 21.0
```

JuMP can also be used to solve other genres of optimization problems. For example, let us consider an MIO problem of the form:

$$max_{x_1,x_2} 2x_1 + 3x_2$$

$$4x_1 + 3x_2 \leq 12$$

$$2x_1 + x_2 \leq 6$$

$$x_1, x_2 \geq 0$$

$$x_1, x_2 \in \mathbb{Z}$$

```julia
using JuMP, Gurobi

model = JuMP.Model(Gurobi.Optimizer)

@variable(model, x1 >= 0, Int) # Must specify as Int
@variable(model, x2 >= 0, Int) # Must specify as Int

@objective(model, Max, 2x1 + 3x2)

@constraint(model, 4x1 + 3x2 <= 12)
@constraint(model, 2x1 + x2 <= 6)

optimize!(model)

println("Optimal x1 = ", value(x1))
println("Optimal x2 = ", value(x2))
println("Optimal objective value = ", objective_value(model))
```

```
Set parameter Username
Set parameter LicenseID to value 2630108
Academic license - for non-commercial use only - expires 2026-03-02
Gurobi Optimizer version 12.0.1 build v12.0.1rc0 (win64 - Windows 11.0 (26100.2))

CPU model: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz, instruction set [SSE2|AVX|AVX2|AV
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 2 rows, 2 columns and 4 nonzeros
Model fingerprint: 0xb17432a7
Variable types: 0 continuous, 2 integer (0 binary)
Coefficient statistics:
  Matrix range     [1e+00, 4e+00]
  Objective range  [2e+00, 3e+00]
  Bounds range     [0e+00, 0e+00]
```

```
  RHS range        [6e+00, 1e+01]
Found heuristic solution: objective 6.0000000
Presolve removed 2 rows and 2 columns
Presolve time: 0.01s
Presolve: All rows and columns removed

Explored 0 nodes (0 simplex iterations) in 0.05 seconds (0.00 work units)
Thread count was 1 (of 8 available processors)

Solution count 2: 12 6

Optimal solution found (tolerance 1.00e-04)
Best objective 1.200000000000e+01, best bound 1.200000000000e+01, gap 0.0000%

User-callback calls 198, time in user-callback 0.00 sec
-0.0mal x1 =
Optimal x2 = 4.0
Optimal objective value = 12.0
```

We can try a large problem. MIO is an especially difficult problem to solve, and when we have a lot of variables and constraints to fulfill, it can take more and more time to solve. JuMP and Gurobi offer functionalities useful in working around potentially nasty problems; we can adjust the number of the threads being used, and we can tell the solver that we are willing to accept certain levels of optimality gaps, in order to speed up solving.

```
Pkg.add("MathOptInterface")
```

```
   Resolving package versions...
  No Changes to `C:\Users\joshu\.julia\environments\v1.11\Project.toml`
  No Changes to `C:\Users\joshu\.julia\environments\v1.11\Manifest.toml`
```

```
using JuMP, Gurobi, Random

# Use MOI directly from MathOptInterface (no need to redefine it)
Random.seed!(42)

# Create the model using Gurobi
model = JuMP.Model(Gurobi.Optimizer)

# Problem size parameters
num_int_vars = 300  # Integer variables
```

```
num_bin_vars = 150  # Binary variables
num_constraints = 500  # Total constraints

# Generate random coefficients ensuring feasibility & boundedness
c = rand(10:100, num_int_vars)      # Objective coefficients for x
d = rand(5:50, num_bin_vars)        # Objective coefficients for y
e = rand(1:5, num_int_vars)         # Small penalty to prevent unbounded solutions
a = rand(1:5, num_constraints, num_int_vars)  # Constraint coefficients for x (smaller values
b = rand(1:3, num_constraints, num_bin_vars)  # Constraint coefficients for y (smaller values
k = sum(a, dims=2)[:, 1] * 50 + sum(b, dims=2)[:, 1] * 1  # Ensures feasibility

# Define integer variables (1   x_i   100, x_i   )
@variable(model, 1 <= x[1:num_int_vars] <= 100, Int)

# Define binary variables (y_j   {0,1})
@variable(model, y[1:num_bin_vars], Bin)

# Define the objective function (maximize Z)
@objective(model, Max,
    sum(c[i] * x[i] for i in 1:num_int_vars) +
    sum(d[j] * y[j] for j in 1:num_bin_vars) -
    sum(e[i] * x[i]^2 for i in 1:num_int_vars)  # Small quadratic penalty to ensure boundedne
)

# Add constraints ensuring feasibility
for j in 1:num_constraints
    @constraint(model,
        sum(a[j, i] * x[i] for i in 1:num_int_vars) +
        sum(b[j, j] * y[j] for j in 1:num_bin_vars) <= k[j]
    )
end

# Increase solver difficulty while keeping it feasible
set_optimizer_attribute(model, "TimeLimit", 120)  # Give Gurobi 2 minutes
set_optimizer_attribute(model, "MIPGap", 0.02)  # Allow 2% optimality gap
# set_optimizer_attribute(model, "Threads", 1)  # Single-threaded (slows down solving)
# set_optimizer_attribute(model, "Presolve", 2)  # Enable presolve for better performance
# set_optimizer_attribute(model, "Cuts", 1)  # Allow cuts to help find solutions

# Solve the problem and track time
@time optimize!(model)
```

```julia
println("Optimal Objective Value: ", objective_value(model))
println("First 10 integer variable values: ", [value(x[i]) for i in 1:10])
println("First 10 binary variable values: ", [value(y[j]) for j in 1:10])
```

```
Set parameter Username
Set parameter LicenseID to value 2630108
Academic license - for non-commercial use only - expires 2026-03-02
Set parameter TimeLimit to value 120
Set parameter MIPGap to value 0.02
Set parameter MIPGap to value 0.02
Set parameter TimeLimit to value 120
Gurobi Optimizer version 12.0.1 build v12.0.1rc0 (win64 - Windows 11.0 (26100.2))

CPU model: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz, instruction set [SSE2|AVX|AVX2|AV
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Non-default parameters:
TimeLimit  120
MIPGap  0.02

Optimize a model with 500 rows, 450 columns and 225000 nonzeros
Model fingerprint: 0xe3ee7d39
Model has 300 quadratic objective terms
Variable types: 0 continuous, 450 integer (150 binary)
Coefficient statistics:
  Matrix range     [1e+00, 5e+00]
  Objective range  [5e+00, 1e+02]
  QObjective range [2e+00, 1e+01]
  Bounds range     [1e+00, 1e+02]
  RHS range        [4e+04, 5e+04]
Found heuristic solution: objective 15690.000000
Presolve added 1 rows and 0 columns
Presolve removed 0 rows and 58 columns
Presolve time: 0.58s
Presolved: 501 rows, 392 columns, 150592 nonzeros
Presolved model has 300 quadratic objective terms
Variable types: 0 continuous, 392 integer (50 binary)
Found heuristic solution: objective 127175.00000

Root relaxation: objective 1.314432e+05, 95 iterations, 0.04 seconds (0.01 work units)

    Nodes    |    Current Node    |     Objective Bounds      |     Work
```

```
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0 131443.208    0  229 127175.000 131361.000  3.29%     -    0s
H    0     0                     131192.00000 131361.000  0.13%     -    0s

Explored 1 nodes (95 simplex iterations) in 0.88 seconds (0.16 work units)
Thread count was 8 (of 8 available processors)

Solution count 3: 131192 127175 15690

Optimal solution found (tolerance 2.00e-02)
Best objective 1.311920000000e+05, best bound 1.313610000000e+05, gap 0.1288%

User-callback calls 244, time in user-callback 0.00 sec
  1.750054 seconds (185.28 k allocations: 18.838 MiB, 4.48% gc time, 42.51% compilation time
Optimal Objective Value: 131192.0
[12.0, 13.0, 14.0, 13.0, 36.0, 7.0, 33.0, 7.0, 26.0, 5.0]
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
```