Peter Donahue
ECON 504

Warm-Up Assignment – Commentary

1. Linear Programming

The solution to this question is in the script **1LinearProgramming.jl**. When the script is executed the **main()** function is called. This is a thin diver function, which also serves as a high level description of the process: some data is loaded into memory (**InitData**), a model is initialized with this data (**InitModel**), the optimizer is invoked, and some results are printed to the terminal (**prettyPrintResults**).

```
function main()
#---------------------------------------------------------------------------
# Conductor
#---------------------------------------------------------------------------
    inputs, resources = initData()
    model, products = initModel(inputs, resources)

    optimize!(model)
    @assert is_solved_and_feasible(model)

    prettyPrintResults(inputs, resources, model, products)
end
```

First, some data is loaded into memory. I defined two structures to hold the data from the table. **ProductInputs** holds the resource use and profit profiles of each product and **Resources** holds the total units available of each resource. Instances of these structures are initialized and populated with data in the **initData** function.

```
struct ProductInputs
    fabrication::Float64
    assembly::Float64
    machining::Float64
    wood::Float64
    profit::Float64
end

struct Resources
    fabrication::Float64
    assembly::Float64
    machining::Float64
    wood::Float64
end
```

Second, a model is initialized with this data in the **initModel** function. This is the most logically important piece of the script, here I shall highlight and explain the most important parts.

First, a new JuMP model (**model**) is instantiated with the HiGHS.Optimizer. Next, I create a set of decision variables named **products**. This set has one element for each of the three products under consideration: Chairs, Desks, Tables.

```
@variable(model, products[productNames])
```

Next, I assign the models objective, which is to select the values for the decision variables[1] so as to maximize the total profit of the firm – the sum of profit per unit times units produced.

```
@objective(model, Max, sum(profitPerUnit .*  Array(products)));
```

Lastly, two constraints are added to the model. First, the quantity produced of each product cannot be negative. Second, the total quantity of each resource required to achieve that output cannot exceed the quantity available.

```
# Non-Negativity Constraint
@constraint(model, 0 <= Array(products))

#Resource Constraint
@constraint(
    model,
    [resource in fieldnames(typeof(resources))],
    sum([getfield(inputs[name], resource) for name in productNames] .* Array(products))
    <=  getfield(resources, resource),
)
```

After the model has been initialized the program returns to the **main** function where the model is optimized and the results are passed off to a helper function which displays them to the terminal.

```
----------------------------
          Results
- - - - - - - - - - - - - -
+ Units Produced
  Tables = 220.0
  Desks  = 85.0
  Chairs = -0.0
- - - - - - - - - - - - - - -
+ Resource: Used / Available
  fabrication: 475.0 / 1000.0
  assembly: 2000.0 / 2000.0
  machining: 1390.0 / 1440.0
  wood: 1440.0 / 1440.0
- - - - - - - - - - - - - -
Total Profit = $4355.0
----------------------------
```

[1] In the language of the problem, “to select the values for the decision variables” means to pick the quantity produced of each product

Peter Donahue
ECON 504

2. Nonlinear Programming

The solution to this question is in the script **2NonlinearProgramming.jl**. When the script is executed the **main()** function is called. This is a thin diver function, which also serves as a high level description of the process: a model is initialized with this data (**initModel**), the optimizer is invoked, and some results are printed to the terminal (**prettyPrintResults**). Contrasting my approach to that taken in the first question, I chose to define all relevant data as module level constants rather than instantiating structures with this information.

```
function main()
#---------------------------------------------------------------------------
# Conductor
#---------------------------------------------------------------------------
    model, p1, p2, x1, x2 = initModel()

    # Solve the model
    optimize!(model)
    @assert is_solved_and_feasible(model)

    prettyPrintResults(model, p1, p2, x1, x2)

    return nothing
end
```

As with the first solution, the **initModel** function is really the logical core of this script. After instantiating a new JuMP model (**model**) with the Ipopt.Optimizer, I start by defining the decision variables, which in the model are price. Here I explicitly include a non-negativity constraint, but that is not strictly necessary as they will not end up binding.

```
@variable(model, p1 >= 0)
@variable(model, p2 >= 0)
```

I use the **@expression** function to supply the model with the demand functions defined in the question.

```
@expression(model, x1, _BETA_1 + _ALPHA_11 * p1 + _ALPHA_12 * p2)
@expression(model, x2, _BETA_2 + _ALPHA_21 * p1 + _ALPHA_22 * p2)
```

Next, I define the objective function and specify that the optimizer's goal should be to maximize it. The object function is the profit function where **p1*x1 + p2*x2** is the revenue from selling **x1**,**x2** units at prices **p1**,**p2** respectively, and **(_C_1*x1 + _C_2*x2)** is the associated cost.

```
@objective(model, Max, p1 * x1 + p2 * x2 - (_C_1 * x1 + _C_2 * x2))
```

Lastly, I add the resource constrains, namely that the total amount of fabric type 1 (**_A_11*x1 + _A_12*x2**) and the total amount of fabric type 2 (**_A_21*x1 + _A_22*x2**) cannot exceed **_B_1** and **_B_2**, respectively.

```
@constraint(model, _A_11 * x1 + _A_12 * x2 <= _B_1)
@constraint(model, _A_21 * x1 + _A_22 * x2 <= _B_2)
```

With the model fully defined, the program flow returns to the **main** function, where the optimization is run and the results are passed off to a helper function which displays them to the terminal.

```
------------------------------
          Results
- - - - - - - - - - - - - - - -
+ Optimal Quantity
  Diaper type 1 = 2.7222221823301505
  Diaper type 2 = 0.7777777214987571
- - - - - - - - - - - - - - - -
+ Optimal Prices
  Diaper type 1 = 3.55555560503307
  Diaper type 2 = 2.5555555747264407
- - - - - - - - - - - - - - - -
+ Optimal Resource use
  Fabric 1 in diaper 1 = 2.7222221823301505
  Fabric 1 in diaper 2 = 1.5555554429975142
  Total amount of Fabric 1 used = 4.277777625327665

  Fabric 2 in diaper 1 = 5.444444364660301
  Fabric 2 in diaper 2 = 2.3333331644962714
  Total amount of Fabric 2 used = 7.777777529156572
- - - - - - - - - - - - - - - -
Optimal profit = 5.444444444444444
------------------------------
```

3. Global Optimization

Note: there is a fair amount of code that was used to generate the graph in this file (as well as two additional dependencies), I have commented it out to speed up the runtime and avoid the risk of trying to use a dependency that may not be installed on all machines. I apologize for the resulting sloppy presentation.

The solution to this question is in the script **3GlobalOptimization.jl**. When the script is executed I start by generating the data for **x1**, **x2**, **epsilon**, and **y**. Although I'm sure that is a better way to do it, I found it much easier to just make those global variables.

```
x = rand(MvNormal(_X_DIST_MU, _X_DIST_COV), _N)'
x1 = x[:, 1]
x2 = x[:, 2]
epsilon = [rand(Normal(0, abs(0.3 * x1[i] + 0.8 * x2[i]))) for i in 1:_N]
y = @. (x1 + x2 * _TRUE_BETA + epsilon) >= 0
```

After the data has been generated, the **main()** function is called. With the plot generation portion commented out, the **main** function consists solely of a call to **bboptimize** – the **BlackBoxOptim** optimization solver. It supply's **bboptimize** the **objective_wrapper** as the function to perform the optimization with, as well as some **kwarg** parameters that I added to try and speed up the runtime.

```
function main()
#---------------------------------------------------------------------------
# Conductor
#---------------------------------------------------------------------------
    results = bboptimize(objective_wrapper;
        SearchRange = _SEARCH_RANGE,
        MaxSteps = _MAX_STEPS,
        PopulationSize = _POPULATION_SIZE
    )

# == Plot Generation
#    plotResults(results.archive_output.best_candidate[1])
# ==
    return nothing
end
```

The **objective_wrapper** function intermediates between the **max_score_objective** function and the optimization routine. It retrieves the **beta** from the **beta_vec** parameter passed in by the optimizer, passes it onto the **max_score_objective** function to calculates a fitness score, which is then returned to the optimizer.

```
function max_score_objective(beta)
    correct_predictions = sum((y .== 1) .* ((x1 .+ x2 .* beta) .>= 0) .+ (y .== 0) .* ((x1 .+ x2 .* beta) .< 0))
    return correct_predictions / _N
end

function objective_wrapper(beta_vec)
    beta = beta_vec[1]
    return -max_score_objective(beta)  # Minimizing, so we return the negative
end
```

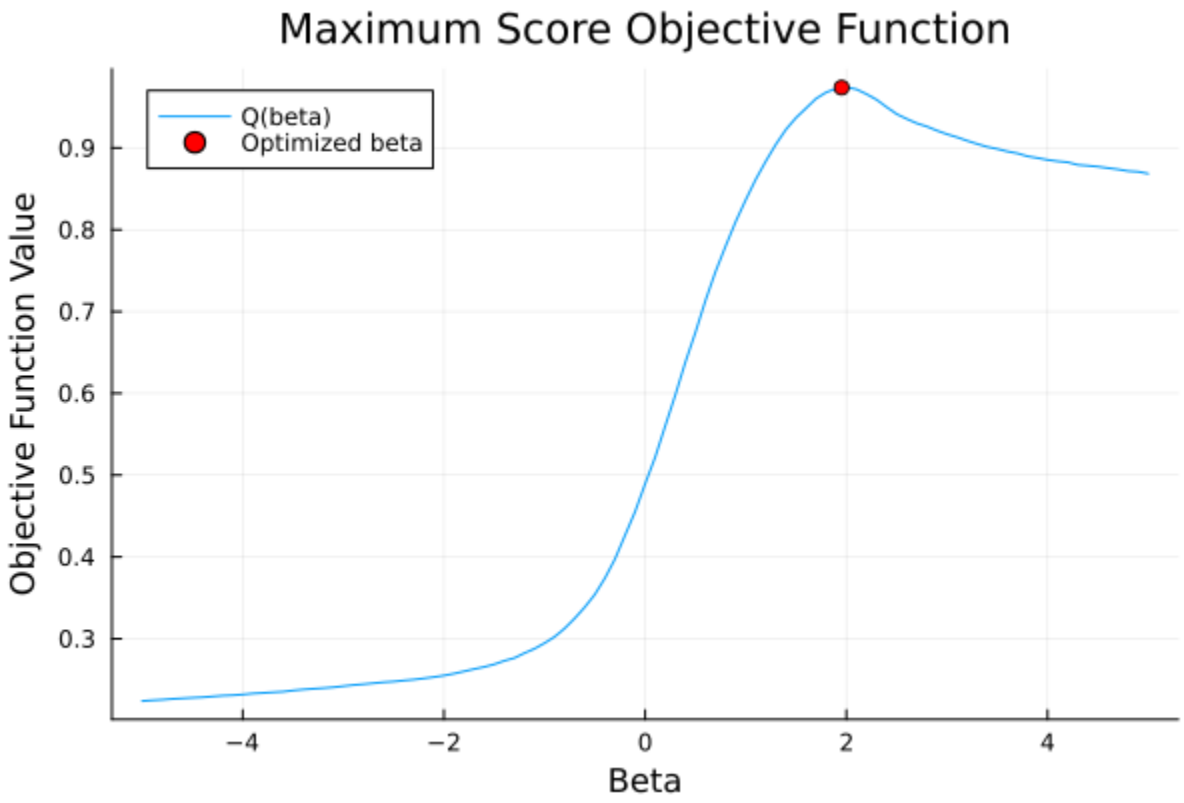
Here is an example of the output from this optimization process. After 11 steps it landed on a best candidate for the Beta value of 2.47251, with a fitness score of 0.9453.

```
Optimization stopped after 11 steps and 346.65 seconds
Termination reason: Max number of steps (10) reached
Steps per second = 0.03
Function evals per second = 0.06
Improvements/step = 0.40000
Total function evaluations = 21


Best candidate found: [2.47251]

Fitness: -0.945300000
```

Peter Donahue
ECON 504

Object Function Plot and Discussion

Note: this plot was produced during a different run from the sample output snippet above, so the numbers aren't exactly the same.



The value of the objective function at the optimized Beta represents the fraction of correct predictions made by the model. This fraction indicates how well the estimated parameter fits the data. A value close to 1 means that the model is correctly predicting most of the outcomes. In the plot above the objective function value at the optimized Beta is around 0.97, meaning that at this Beta about 97% of the predictions made by the objective function are "correct".