# Homework #1 (4 points)

Due June 27 @ 11:59pm

## Submission requirements

Upload a **single PDF or HTML file** of your IJulia notebook for this entire assigment. Clearly denote which question each section of your file corresponds to.

## Problem 1 - Getting Started with Julia/JuMP

Model the following problem in JuMP:

Solve this problem using Clp, ECOS, and SCS solvers. the following problem in JuMP:

$$\begin{aligned} \min\ & -x_1 + x_2 - 2x_4 \\ \text{s.t. } & 3x_1 - x_3 \geq 5 \\ & x_2 + 2x_3 = 2 \\ & x_3 \geq -x_4 \\ & x_1 - x_2 \leq 2 \\ & x_1 \geq 0, -1 \leq x_2 \leq 1, x_4 \leq 0 \end{aligned}$$

- Which solver is most accurate (gets closest to the "right" answer)?
- Which is fastest (use the @time macro)? NOTE: you should notice it takes longer to solve the problem the first time you run your code, and goes much faster if you re-sovle subsequent times. The first time you run the code, the solver takes extra time to compile/"load" the model (it doesn't repeat the compilation step in future re-solves).
  - Can you speculate as to why?
- If there is no clear difference between the solvers, can you think of some factors that might contribute to solver speed differences?

In [1]:

```julia
using JuMP, Clp, SCS, ECOS
m = Model()
@variable(m,x1 >= 0)
@variable(m, -1 <= x2 <= 1)
@variable(m, x3)
@variable(m, x4 <= 0)
@objective(m, Min, -x1 + x2 - 2x4)

@constraint(m, 3x1 - x3 >= 5)
@constraint(m, x2 + 2x2 == 2)
@constraint(m, x3 >= -x4)
@constraint(m, x1-x2 <= 2)

println("### CLP SOLVER ###")

set_optimizer(m, Clp.Optimizer)
set_silent(m)
optimize!(m)

for i in 1:5
    set_optimizer(m, Clp.Optimizer) # we have to do this every time for SCS (known bug), so we also do it for
    set_silent(m)
    @time(optimize!(m))
    println()
    println("objective: ", objective_value(m))
    println("x1: ", value(x1))
    println("x2: ", value(x2))
    println("x3: ", value(x3))
    println("x4: ", value(x4))
    println()
end

println("### ECOS SOLVER ###")
set_optimizer(m, ECOS.Optimizer)
set_silent(m)
optimize!(m)

for i in 1:5
    set_optimizer(m, ECOS.Optimizer)
    set_silent(m)
    @time(optimize!(m))
    println()
    println("objective: ", objective_value(m))
    println("x1: ", value(x1))
    println("x2: ", value(x2))
    println("x3: ", value(x3))
    println("x4: ", value(x4))
    println()
end

println("### SCS SOLVER ###")

set_optimizer(m, SCS.Optimizer)
set_silent(m)
optimize!(m)

for i in 1:5
    set_optimizer(m, SCS.Optimizer)
    set_silent(m)
    @time(optimize!(m))
    println()
    println("objective: ", objective_value(m))
    println("x1: ", value(x1))
    println("x2: ", value(x2))
    println("x3: ", value(x3))
    println("x4: ", value(x4))
    println()
end
```

```
### CLP SOLVER ###
  0.000412 seconds (541 allocations: 34.203 KiB)

objective: -2.0
x1: 2.6666666666666665
x2: 0.6666666666666666
x3: 3.0
x4: 0.0

  0.000520 seconds (541 allocations: 34.203 KiB)

objective: -2.0
x1: 2.6666666666666665
x2: 0.6666666666666666
x3: 3.0
x4: 0.0

  0.000927 seconds (541 allocations: 34.203 KiB)

objective: -2.0
x1: 2.6666666666666665
x2: 0.6666666666666666
x3: 3.0
x4: 0.0

  0.000963 seconds (541 allocations: 34.203 KiB)

objective: -2.0
x1: 2.6666666666666665
x2: 0.6666666666666666
x3: 3.0
x4: 0.0

  0.001615 seconds (541 allocations: 34.203 KiB)

objective: -2.0
x1: 2.6666666666666665
x2: 0.6666666666666666
x3: 3.0
x4: 0.0

### ECOS SOLVER ###
  0.000836 seconds (1.56 k allocations: 90.500 KiB)

objective: -2.0000000000773084
x1: 2.6666666667096113
x2: 0.6666666666666742
x3: 0.7858212323588336
x4: 1.7185586534956287e-11

  0.000819 seconds (1.56 k allocations: 90.500 KiB)

objective: -2.0000000000773084
x1: 2.6666666667096113
x2: 0.6666666666666742
x3: 0.7858212323588336
x4: 1.7185586534956287e-11

  0.000813 seconds (1.56 k allocations: 90.500 KiB)

objective: -2.0000000000773084
x1: 2.6666666667096113
x2: 0.6666666666666742
x3: 0.7858212323588336
x4: 1.7185586534956287e-11

  0.000815 seconds (1.56 k allocations: 90.500 KiB)

objective: -2.0000000000773084
x1: 2.6666666667096113
x2: 0.6666666666666742
x3: 0.7858212323588336
x4: 1.7185586534956287e-11

  0.000552 seconds (1.56 k allocations: 90.500 KiB)

objective: -2.0000000000773084
x1: 2.6666666667096113
x2: 0.6666666666666742
x3: 0.7858212323588336
x4: 1.7185586534956287e-11

### SCS SOLVER ###
  0.001125 seconds (1.71 k allocations: 96.297 KiB)

objective: -2.0000215886290587
x1: 2.666684854216829
x2: 0.6666666799477369
x3: 0.8696850694571289
x4: 1.707179983342114e-6

  0.001231 seconds (1.71 k allocations: 96.297 KiB)

objective: -2.0000215886290587
x1: 2.666684854216829
x2: 0.6666666799477369
x3: 0.8696850694571289
x4: 1.707179983342114e-6

  0.001587 seconds (1.71 k allocations: 96.297 KiB)

objective: -2.0000215886290587
x1: 2.666684854216829
x2: 0.6666666799477369
x3: 0.8696850694571289
x4: 1.707179983342114e-6

  0.002439 seconds (1.71 k allocations: 96.297 KiB)

objective: -2.0000215886290587
x1: 2.666684854216829
x2: 0.6666666799477369
x3: 0.8696850694571289
x4: 1.707179983342114e-6

  0.002865 seconds (1.71 k allocations: 96.297 KiB)

objective: -2.0000215886290587
x1: 2.666684854216829
x2: 0.6666666799477369
x3: 0.8696850694571289
x4: 1.707179983342114e-6
```

## Problem 2 - Making Chemicals

Spider-man is experimenting with new ways to produce his web fluid. He is trying to figure out the optimal way to generate three necessary chemicals that he uses in the fluid: chemicals A, B, and C.

The chemicals can be produced via two different processes, which we will just call process 1 and process 2. Running process 1 for an hour costs Spider-man $4 and yields 3 vials of A, 1 vial of B, and 1 vial of C. Running process 2 for an hour costs Spider-man \$1 and produces 1 vial of A and 1 vial of B.

To make enough web fluid, he needs at least 10 vials of A, 5 vials of B, and 3 vials of C.

(a) Formulate a linear program to help Spider-man figure out how many hours of each process to run to minize his cost while making enough of each chemical. State the math model, then code and solve the model using Julia. (Hint: you should have two decision variables, one for each chemical production process).

(b) Code the same model once again, this time separating the parameters from the solution as we did in class (see Top Brass examples). Confirm that you obtain the same solution as in part (a).

(c) Solve the problem graphically by plotting the feasible set and at least two isocost lines for the objective function. Confirm that you obtain the same solution as in the previous parts.

(a) Linear Program

$$\begin{aligned} \min\ & 4x_1 + x_2 \\ \text{s.t}\quad & 3x_1 + x_2 \geq 10 \\ & x_1 + x_2 \geq 5 \\ & x_1 \geq 3 \\ & x_1, x_2 \geq 0 \end{aligned}$$

In [20]:

```julia
# Solution to Problem 2a
println("Problem 2a:")

using JuMP, Clp # Clp stands for Coin-or Linear Programming

# ~~ Build the model ~~
m = Model()

# Create a variable for each process
@variable(m, process_1_count >= 0)
@variable(m, process_2_count >= 0)

# Create an objective function
@objective(m, Min, 4*process_1_count + 1*process_2_count)

# Create constraints
@constraint(m, A_count, 3*process_1_count + 1*process_2_count >= 10)
@constraint(m, B_count, 1*process_1_count + 1*process_2_count >= 5)
@constraint(m, C_count, 1*process_1_count >= 3)

# ~~ Solve the model ~~
# Set optimizer to Clp
set_optimizer(m, Clp.Optimizer)

optimize!(m)

# Print the objective value
println("Objective value = " ,objective_value(m))

# Print the values of the variables
println("Run Process #1 " ,value(process_1_count), " times.")
println("Run Process #2 ",value(process_2_count), " times.\n" )
```

```
Problem 2a:
Objective value = 14.0
Run Process #1 3.0 times.
Run Process #2 2.0 times.

Coin0506I Presolve 2 (-1) rows, 2 (0) columns and 4 (-1) elements
Clp0006I 0  Obj 12 Primal inf 2.3333313 (2)
Clp0006I 1  Obj 14
Clp0000I Optimal - objective value 14
Coin0511I After Postsolve, objective 14, infeasibilities - dual 0 (0), primal 0 (0)
Clp0032I Optimal objective 14 - 1 iterations time 0.002, Presolve 0.00
```

In [21]:

```julia
# Solution to Problem 2b
println("Problem 2b:")

# ~~ Assemble Problem Data ~~

# A dictionary containing potential processes
process_types = [:proc_1, :proc_2]

# Dictionaries mapping processes to their A, B, and C yields
a_yield = Dict(:proc_1=>3, :proc_2=>1)
b_yield = Dict(:proc_1=>1, :proc_2=>1)
c_yield = Dict(:proc_1=>1, :proc_2=>0)

# A dictionary mapping the process to its cost/hour
proc_cost = Dict(:proc_1=>4, :proc_2=>1)

# The minimum count of each type of vial
a_count = 10
b_count = 5
c_count = 3

using JuMP, Clp

# ~~ Build the model ~~

m = Model()

# Processes variable is now a dictionary indexed over process types (proc_1, proc_2)
@variable(m, processes[process_types] >= 0)

# Create an objective function
@objective(m, Min, sum(proc_cost[i] * processes[i] for i in process_types))

# Create constraints
@constraint(m, sum(a_yield[i] * processes[i] for i in process_types) >= 10) # Constraint on A vials
@constraint(m, sum(b_yield[i] * processes[i] for i in process_types) >= 5)  # Constraint on B vials
@constraint(m, sum(c_yield[i] * processes[i] for i in process_types) >= 3)  # Constraint on C vials

# ~~ Solve the model ~~
# Set optimizer to Clp
set_optimizer(m, Clp.Optimizer)

optimize!(m)

# Print the objective value
println("Objective value = " ,objective_value(m))

# Print the values of the variables
println("Run Process #1 " ,value(processes[:proc_1]), " times.")
println("Run Process #2 " ,value(processes[:proc_2]), " times.\n" )
```

```
Problem 2b:
Objective value = 14.0
Run Process #1 3.0 times.
Run Process #2 2.0 times.

Coin0506I Presolve 2 (-1) rows, 2 (0) columns and 4 (-1) elements
Clp0006I 0  Obj 12 Primal inf 2.3333313 (2)
Clp0006I 1  Obj 14
Clp0000I Optimal - objective value 14
Coin0511I After Postsolve, objective 14, infeasibilities - dual 0 (0), primal 0 (0)
Clp0032I Optimal objective 14 - 1 iterations time 0.002, Presolve 0.00
```

Solution to Problem 2c SUBMITTED SEPARATELY

## Problem 3 - Craft Beer

A local microbrewery is planning operations for the summer season. They are brewing 10 different rotational beers this season in 4 different locations. During the summer, the brewery has 35,000 hours of tank fermentation time available at each location. Because of the slight differences in equipment at each brewing location, the cost of producing a gallon of beer differs at each location. The time and cost for each brewery are given in the .csv file "brewery.csv." A sample of the data is given in the table below. There is a minimum number of kegs required for each type of beer, also given in the .csv file.

The data is provided in "brewery.csv" on Canvas. You can use the code snippet provided below to read the .csv files and load them into Julia, or you can read/load the data in any way you choose.

| Beer | Time (hrs) | Min required | Cost/gall ($) at loc 1 | Cost/gall ($) at loc 2 | Cost/keg ($) at loc 3 | Cost/keg ($) at loc 4 |
|---|---|---|---|---|---|---|
| 1 | 20 | 500 | 10 | 12 | 9 | 13 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋱ |

(a) Formulate a linear program to help the microbrewery minimize cost of meeting demand for each type of beer. Give a general form (parameters only -- no numbers) of the math model.

(b) Implement and solve this instance of the model in Julia/JuMP. Display the optimal objective value and the optimal

(a) Let $B$ be the set of beers. Let $N$ be the set of locations. Define $c_{ij}$ as the cost of brewing beer $i \in B$ at location $j \in N$. Let $t_i$ be the time it takes to brew beer $i \in B$. Let $\ell_i$ and be the minimum requirement of each beer type $i \in B$. Finally, let $T_j$ represent the total time (days) available for brewing at location $j \in N$.

Let $x_{ij}$ be the kegs of beer $i$ brewed at location $j$.

Constraints limit how much we can brew at each location and require we meet demand for each type of beer. We cannot brew negative numbers of kegs.

Now we can write our model:

$$\begin{aligned} \min\ & \sum_{i \in B, j \in N} c_{ij} x_{ij} \\ \text{s.t. } & \sum_{j \in N} x_{ij} \geq \ell_i \ \forall i \in B \\ & \sum_{i \in B} t_i x_{ij} \leq T_j \ \forall j \in N \\ & x_{ij} \geq 0 \ \forall i \in B, j \ \in N. \end{aligned}$$

In [4]:

```julia
#You might need to run "Pkg.add(...)" before using these packages
using DataFrames, CSV, NamedArrays

#Load the data file
df = CSV.read("brewery.csv",DataFrame,delim=',');

# create a list of beers
beers = convert(Array,df[1:end,1])

# create a list of locations
locs = 1:4

# create a dictionary of the total time it takes to brew each type of beer
time_to_brew = Dict(zip(beers,df[1:end,2]))

# create a dictionary of the minimum gallons required for each beer
min_req = Dict(zip(beers,df[1:end,3]))

brew_cost_matrix = Matrix(df[1:end,4:end])

brew_cost_array = NamedArray(brew_cost_matrix, (beers, locs),("beers","locs"))
;
```

In [5]:

```julia
#You might need to run "Pkg.add(...)" before using these packages
using DataFrames, CSV, NamedArrays

#Load the data file
df = CSV.read("brewery.csv",DataFrame,delim=',');

# create a list of beers
beers = convert(Array,df[1:end,1])
# create a list of locations
locs = 1:4
# create a dictionary of the total time it takes to brew each type of beer
time_to_brew = Dict(zip(beers,df[1:end,2]))
# create a dictionary of the minimum gallons required for each beer
min_req = Dict(zip(beers,df[1:end,3]))

brew_cost_matrix = Matrix(df[1:end,4:end])
# rows are roasting methods, columns are flavors
brew_cost_array = NamedArray(brew_cost_matrix, (beers, locs),("beers","locs"))
;

using JuMP, Clp

m = Model(Clp.Optimizer)

@variable(m, x[beers,locs] >= 0)

@objective(m, Min, sum(brew_cost_array[i,j]*x[i,j] for i in beers, j in locs)) # minimize costs

@constraint(m, meet_dem[i in beers], sum(x[i,j] for j in locs) >= min_req[i]) # meet the minimum requirements
@constraint(m, time_const[j in locs], sum(time_to_brew[i]*x[i,j] for i in beers) <= 35000) # don't exceed brew

optimize!(m)

sol = [value(x[i,j]) for i in beers, j in locs]

println("Solution: brew beer at each location according to number of kegs in the following table:")
sol_array = NamedArray(sol, (beers,locs), ("beers","locs"))
println(sol_array)
println("This has a cost of: \$", objective_value(m))
```

```
Solution: brew beer at each location according to number of kegs in the following table:
10×4 Named Matrix{Float64}
beers ╲ locs │       1        2        3        4
─────────────┼───────────────────────────────────
1            │     0.0    500.0      0.0      0.0
2            │     0.0    628.0      0.0      0.0
3            │ 613.828      0.0      0.0  64.1724
4            │     0.0      0.0    547.0      0.0
5            │     0.0      0.0  515.407  188.593
6            │     0.0   540.87  14.1304      0.0
7            │     0.0      0.0    488.0      0.0
8            │     0.0      0.0      0.0    737.0
9            │     0.0      0.0      0.0    534.0
10           │   637.0      0.0      0.0      0.0
This has a cost of: $55945.0
Coin0506I Presolve 14 (0) rows, 40 (0) columns and 80 (0) elements
Clp0006I 0  Obj 0 Primal inf 6008 (10)
Clp0006I 13  Obj 55945
Clp0000I Optimal - objective value 55945
Clp0032I Optimal objective 55945 - 13 iterations time 0.002
```

## Problem 4 - Standard Form

Consider the following LP:

$$\begin{aligned} \min\ & 2x_1 - 3x_2 + 4x_3 - 2x_4 + 5x_5 \\ \text{s.t.}\quad & 3x_2 - x_3 + 4x_4 - x_5 \geq 12 \\ & 4x_1 - 7x_2 + 2x_3 \quad + 11x_5 = 5 \\ & 2x_1 \quad - 3x_3 + 8x_4 + \quad \leq 3 \\ & x_1, x_4, x_5 \geq 0 \\ & x_2, x_3 \text{ Unrestricted in Sign (Free)} \end{aligned}$$

(a) Convert the problem to standard form.

(b) What are $A$, $b$, $c$,and $x$ in this problem? Clearly indicate how the decision variables of your transformed LP relate to those of the original LP.

(c) Solve the standard-form LP in Julia and report the objective value and the value of each decision variable in an optimal solution to the original LP.

Solution (a)

- Let $x_2 = v_2 - w_2$. Let $x_3 = v_3 - w_3$. Then $x_1$, $v_2$, $w_2$, $v_3$, $w_3$, $x_4$, and $x_5 \geq 0$. Replace everywhere $x_2$ and $x_3$ appear:

$$\begin{aligned} \min\ & 2x_1 - 3(v_2 - w_2) + 4(v_3 - w_3) - 2x_4 + 5x_5 \\ \text{s.t.}\quad & 3(v_2 - w_2) - (v_3 - w_3) + 4x_4 - x_5 \geq 12 \\ & 4x_1 - 7(v_2 - w_2) + 2(v_3 - w_3) \quad + 11x_5 = 5 \\ & 2x_1 \quad - 3(v_3 - w_3) + 8x_4 + \quad \leq 3 \\ & x_1, v_2, w_2, v_3, w_3 x_4, x_5 \geq 0 \end{aligned}$$

- Simplify and fix the objective:

$$\begin{aligned} -\max\ & -2x_1 + 3v_2 - 3w_2 - 4v_3 + 4w_3 + 2x_4 - 5x_5 \\ \text{s.t.}\quad & 3v_2 - 3w_2 - v_3 + w_3 + 4x_4 - x_5 \geq 12 \\ & 4x_1 - 7v_2 + 7w_2 + 2v_3 - 2w_3 \quad + 11x_5 = 5 \\ & 2x_1 \quad - 3v_3 + 3w_3 + 8x_4 + \quad \leq 3 \\ & x_1, v_2, w_2, v_3, w_3 x_4, x_5 \geq 0 \end{aligned}$$

- Fix the constraints:

$$\begin{aligned} -\max\ & -2x_1 + 3v_2 - 3w_2 - 4v_3 + 4w_3 + 2x_4 - 5x_5 \\ \text{s.t.}\quad & -3v_2 + 3w_2 + v_3 - w_3 - 4x_4 + x_5 \leq -12 \\ & 4x_1 - 7v_2 + 7w_2 + 2v_3 - 2w_3 \quad + 11x_5 \leq 5 \\ & -4x_1 + 7v_2 - 7w_2 - 2v_3 + 2w_3 \quad - 11x_5 \leq -5 \\ & 2x_1 \quad - 3v_3 + 3w_3 + 8x_4 + \quad \leq 3 \\ & x_1, v_2, w_2, v_3, w_3 x_4, x_5 \geq 0 \end{aligned}$$

(b) $x = \begin{bmatrix} x_1 \\ v_2 \\ w_2 \\ v_3 \\ w_3 \\ x_4 \\ x_5 \end{bmatrix}$, $A = \begin{bmatrix} 0 & -3 & 3 & 1 & -1 & -4 & 1 \\ 4 & -7 & 7 & 2 & -2 & 0 & 11 \\ -4 & 7 & -7 & -2 & 2 & 0 & -11 \\ 2 & 0 & 0 & -3 & 3 & 8 & 0 \end{bmatrix}$, $c = \begin{bmatrix} -2 & 3 & -3 & -4 & 4 & 2 & -5 \end{bmatrix}$, and $b = \begin{bmatrix} -12 \\ 5 \\ -5 \\ 3 \end{bmatrix}$.

And $x_2 = v_2 - w_2$ and $x_3 = v_3 - w_3$.

In [6]:

```julia
using JuMP, Clp

m = Model(Clp.Optimizer)

@variable(m, x[1:5])

@objective(m, Min, 2x[1] - 3x[2] + 4x[3] -2x[4] + 5x[5])

@constraint(m, x[1] >= 0)
@constraint(m, x[4] >= 0)
@constraint(m, x[5] >= 0)
@constraint(m, 3x[2] -x[3] +4x[4] - x[5]>= 12)
@constraint(m, 4x[1]-7x[2] +2x[3]+11x[5] == 5)
@constraint(m, 2x[1] -3x[3]+8x[4] <=3)

optimize!(m)

println("objective: ", objective_value(m))
println("solution: ", value.(x))
A = [0 -3 3 1 -1 -4 1; 4 -7 7 2 -2 0 11; -4 7 -7 -2 2 0 -11; 2 0 0 -3 3 8 0]

c = [-2; 3; -3; -4; 4; 2; -5]

b= [-12; 5; -5; 3]

using JuMP, Clp

m = Model(Clp.Optimizer)

@variable(m, x[1:7] >= 0)
@objective(m, Max, c'*x)
@constraint(m, A*x .<= b)

optimize!(m)

println("x1 = ", value(x[1]))
println("x2 = ", value(x[2])-value(x[3]))
println("x3 = ", value(x[4])-value(x[5]))
println("x4 = ", value(x[6]))
println("x5 = ", value(x[7]))
println("objective: ", -(objective_value(m)))
```

```
objective: 0.0769230769230802
solution: [0.0, 4.9230769230769225, -1.0, 0.0, 3.769230769230769]
x1 = 0.0
x2 = 4.92307684971736
x3 = -1.0
x4 = 0.0
x5 = 3.7692304391127363
objective: 0.0769261213449326
Coin0506I Presolve 0 (-6) rows, 0 (-5) columns and 0 (-14) elements
Clp3002W Empty problem - 0 rows, 0 columns and 0 elements
Clp0000I Optimal - objective value 0.076923077
Coin0511I After Postsolve, objective 0.076923077, infeasibilities - dual 0 (0), primal 0 (0)
Clp0032I Optimal objective 0.07692307692 - 0 iterations time 0.002, Presolve 0.00
Coin0506I Presolve 4 (0) rows, 7 (0) columns and 22 (0) elements
Clp0006I 0  Obj 0 Primal inf 3.4545453 (2) Dual inf 16.666666 (3)
Clp0006I 3  Obj -0.076926121
Clp0000I Optimal - objective value -0.076926121
Clp0032I Optimal objective -0.07692612134 - 3 iterations time 0.002
```