Homework 1

June 24, 2020

1 Homework1, CS 524, Lorraine Chen

1.1 Problem 1

Using Clp Solver

[36]:

```julia
# always specify which packages you're going to use
using JuMP, Clp

# create a new model object
m = Model()

# specify the solver you want to use to solve Model m
set_optimizer(m, Clp.Optimizer)

# we need variables x1,...,x4
# format is (<model name>, <variable name>). we can optionally
# include bounds on each variable.
@variable(m, x1 >= 0)
@variable(m, x2 <= 0)
@variable(m, x3 >= 0)
@variable(m, x4 == 5)

# objective is to maximize profit
# format is (<model name>, <Max or Min>, <algebraic function>)
@objective(m, Min, 0.5*x1 - 7*x2 + (1/3)*x3)

# constraints
# format is (<model name>, <constraint name>, <algebraic constraint>)
@constraint(m, con1, x1 + x2 - 2*x3 + 0.5*x4   10)
@constraint(m, con2, - x1 + 3*x2   -2)

# use the @time macro to measure the amount of time it takes to solve m
println("Time to solve this model using Clp: ")
@time(optimize!(m))

println("x1 = ", value(x1))
println("x2 = ", value(x2))
```

```
println("x3 = ", value(x3))
println("x4 = ", value(x4))
println("The minimum value of the objective function is ", objective_value(m))
```

```
Time to solve this model using Clp:
  0.001072 seconds (1.80 k allocations: 122.953 KiB)
x1 = 7.5
x2 = 0.0
x3 = 0.0
x4 = 5.0
The minimum value of the objective function is 3.75
Coin0506I Presolve 0 (-2) rows, 0 (-4) columns and 0 (-6) elements
Clp3002W Empty problem - 0 rows, 0 columns and 0 elements
Clp0000I Optimal - objective value 3.75
Coin0511I After Postsolve, objective 3.75, infeasibilities - dual 0 (0), primal
0 (0)
Clp0032I Optimal objective 3.75 - 0 iterations time 0.002, Presolve 0.00
```

Using ECOS Solver

[38]:
```
# using package ECOS
import Pkg
Pkg.add("ECOS")
using ECOS

# use the @time macro to measure the amount of time it takes to solve m
println("Time to solve this model using ECOS: ")
set_optimizer(m, ECOS.Optimizer)
@time(optimize!(m))

println("x1 = ", value(x1))
println("x2 = ", value(x2))
println("x3 = ", value(x3))
println("x4 = ", value(x4))
println("The minimum value of the objective function is ", objective_value(m))
```

```
 Resolving package versions…
  Updating `~/.julia/environments/v1.3/Project.toml`
 [no changes]
  Updating `~/.julia/environments/v1.3/Manifest.toml`
 [no changes]
Time to solve this model using ECOS:
  0.003075 seconds (3.12 k allocations: 198.813 KiB)
x1 = 7.500000000019637
x2 = 2.7672727138375614e-11
x3 = -3.736247973302586e-11
x4 = 5.0
The minimum value of the objective function is 3.7499999998036553
```

```
ECOS 2.0.5 - (C) embotech GmbH, Zurich Switzerland, 2012-15. Web:
www.embotech.com/ECOS

It     pcost       dcost      gap   pres   dres    k/t    mu     step   sigma
IR    |   BT
 0  -9.263e-01  +9.593e+00  +3e+01  5e-01  5e-01  1e+00  5e+00    ---    ---
1  1  - |  -  -
 1  +3.968e+00  +5.340e+00  +2e+00  5e-02  7e-02  3e-01  4e-01  0.9219  2e-03
0  0  0 |  0  0
 2  +3.737e+00  +3.764e+00  +5e-02  9e-04  1e-03  5e-03  8e-03  0.9785  8e-04
0  0  0 |  0  0
 3  +3.750e+00  +3.750e+00  +5e-04  1e-05  1e-05  5e-05  9e-05  0.9890  1e-04
1  0  0 |  0  0
 4  +3.750e+00  +3.750e+00  +6e-06  1e-07  2e-07  6e-07  1e-06  0.9890  1e-04
1  0  0 |  0  0
 5  +3.750e+00  +3.750e+00  +7e-08  1e-09  2e-09  7e-09  1e-08  0.9890  1e-04
1  0  0 |  0  0
 6  +3.750e+00  +3.750e+00  +7e-10  1e-11  2e-11  8e-11  1e-10  0.9890  1e-04
1  0  0 |  0  0

OPTIMAL (within feastol=2.0e-11, reltol=1.9e-10, abstol=7.3e-10).
Runtime: 0.000070 seconds.
```

Using SCS Solver

[18]:
```julia
# use the package SCS
import Pkg
Pkg.add("SCS")
using SCS

# use the @time macro to measure the amount of time it takes to solve m
println("Time to solve this model using SCS: ")
set_optimizer(m, SCS.Optimizer)
@time(optimize!(m))

println("x1 = ", value(x1))
println("x2 = ", value(x2))
println("x3 = ", value(x3))
println("x4 = ", value(x4))
println("The minimum value of the objective function is ", objective_value(m))
```

```
 Resolving package versions…
  Updating `~/.julia/environments/v1.3/Project.toml`
 [no changes]
  Updating `~/.julia/environments/v1.3/Manifest.toml`
 [no changes]
```

```
Time to solve this model using SCS:
  0.002039 seconds (2.55 k allocations: 167.797 KiB)
x1 = 7.5
x2 = 0.0
x3 = 0.0
x4 = 5.0
The minimum value of the objective function is -9.999999999632589
----------------------------------------------------------------------------
        SCS v2.1.1 - Splitting Conic Solver
        (c) Brendan O'Donoghue, Stanford University, 2012
----------------------------------------------------------------------------
Lin-sys: sparse-indirect, nnz in A = 8, CG tol ~ 1/iter^(2.00)
eps = 1.00e-05, alpha = 1.50, max_iters = 5000, normalize = 1, scale = 1.00
acceleration_lookback = 10, rho_x = 1.00e-03
Variables n = 2, constraints m = 5
Cones:  linear vars: 5
Setup time: 3.14e-05s
----------------------------------------------------------------------------
 Iter | pri res | dua res | rel gap | pri obj | dua obj | kap/tau | time (s)
----------------------------------------------------------------------------
     0| 1.62e+19  2.94e+18  7.76e-01 -1.37e+20 -1.73e+19  5.25e+19  1.53e-05
    13| 1.55e-11  5.00e-11  5.28e-12 -1.00e+01 -1.00e+01  7.52e-16  7.65e-05
----------------------------------------------------------------------------
Status: Solved
Timing: Solve time: 7.74e-05s
        Lin-sys: avg # CG iterations: 1.00, avg solve time: 3.33e-07s
        Cones: avg projection time: 4.51e-08s
        Acceleration: avg step time: 3.48e-06s
----------------------------------------------------------------------------
Error metrics:
dist(s, K) = 6.1766e-17, dist(y, K*) = 0.0000e+00, s'y/|s||y| = 6.4119e-17
primal res: |Ax + s - b|_2 / (1 + |b|_2) = 1.5475e-11
dual res:   |A'y + c|_2 / (1 + |c|_2) = 4.9982e-11
rel gap:    |c'x + b'y| / (1 + |c'x| + |b'y|) = 5.2845e-12
----------------------------------------------------------------------------
c'x = -10.0000, -b'y = -10.0000
============================================================================
```

1. **Which solver is most accurate?** Clp is most accurate, because it is a dedicated LP solver.

2. **Which is fastest (use the @time macro)? (Note: you should run each solver several times to get an average time.) Can you speculate as to why?** Clp is fastest, since Clp model is most specialized. The other two solvers are more generalized and broadly used models, so they are slower.

3. **If there is no clear difference between the solvers, can you think of some factors that might contribute to solver speed differences?** Repeating times: by repeating running these solvers, I found they complete faster and the speed differences among them become less clear. (other factors may be the size of the problem, the algorithm)

1.2 Problem 2

(a) Formulate a linear program to help Prof. Smith figure out how many pounds of each type of material (iron and steel) she should purchase to minimize her character's "slowness." State the math model, then code and solve the model using Julia.

The math model: x_1 denotes the amount of steel, x_2 denotes the amount of iron objective function:

$$\min x_1 - 3x_2$$

subject to:

$$x_1 - x_2 \geq -2$$
$$x_1 + 2x_2 \geq 6$$
$$2x_1 + x_2 \leq 8$$
$$x_1, x_2 \geq 0$$

[21]:

```
# packages
using JuMP, Clp

# create a new model object, specifying the solver
m = Model(Clp.Optimizer)

# we need variables for the total amount of steel and iron (pounds)
# format is (<model name>, <variable name>). we can optionally
# include bounds on each variable.
@variable(m, steel >= 0)
@variable(m, iron >= 0)

# objective is to minimize the "slowness"
# format is (<model name>, <Max or Min>, <algebraic function>)
@objective(m, Min, steel - 3*iron)

# constraint on total damage, protection and surface area
# total damage is at least -2 points
# total protection is at least 6 points
# maximum surface area is 8 m^2
# format is (<model name>, <constraint name>, <algebraic constraint>)
@constraint(m, damage, steel - iron >= -2)
@constraint(m, protection, steel + 2*iron >= 6)
@constraint(m, surarea, 2*steel + iron <= 8)

# solve the instance of the problem
optimize!(m)

# display solution information
println("steel: ", value.(steel), " pounds.")
println("iron: ", value.(iron), " pounds.")
println("slowness(minimum): ", objective_value(m))
```

```
steel: 2.0 pounds.
iron: 4.0 pounds.
slowness(minimum): -10.0
Coin0506I Presolve 3 (0) rows, 2 (0) columns and 6 (0) elements
Clp0006I 0  Obj 0 Primal inf 5.9999999 (1) Dual inf 2.9999999 (1)
Clp0006I 2  Obj -10
Clp0000I Optimal - objective value -10
Clp0032I Optimal objective -10 - 2 iterations time 0.002
```

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

[24]:
```
# Problem Data

materials = [:steel, :iron] # these are 2 possible materials

characteristics = [:damage, :protection, :surface_area] # 3 characteristics for␣
↪materials

slowness = Dict( zip(materials, [1,-3] ) ) # slowness = steel(pounds) -␣
↪3*iron(pounds)

char_bounds = Dict( zip(characteristics, [-2, 6, -8] ) ); # bounds of each type␣
↪of characteristic (here I changed 8 -> -8)

# we use the NamedArrays package (you'll need to Pkg.add it first)
using Pkg
Pkg.add("NamedArrays")
using NamedArrays

# create a matrix (Array) of the "recipe" for each material type.
# each row is a material type, each column is a resource (should be ordered the␣
↪same as the characteristics array).
# we read this as: "material type 1 (:steel) causes 1 of characteristics 1 (:
↪damage), 1 of characteristics 2
# (:protection), 2 of characteristics 3 (:surface_area)"
# row 2 is similar, but for material type :iron.
# Notes: change the sign of surface area to match the model afterwards for␣
↪one-side char_bounds
material_char_matrix = [1 1 -2; -1 2 -1]

# create NamedArray that contains info on how much of each resource each trophy␣
↪uses.
# syntax is (<"recipe" matrix>, (<row indices>,<column indices>),(<row␣
↪name>,<column name>))
```

```
material_char_NA = NamedArray(material_char_matrix, (materials,␣
↪characteristics), ("Material","characteristics"))

# check out the output to see how NamedArrays are structured:
```

```
Resolving package versions…
 Updating `~/.julia/environments/v1.3/Project.toml`
[no changes]
 Updating `~/.julia/environments/v1.3/Manifest.toml`
[no changes]
```

[24]:

```
2×3 Named Array{Int64,2}
Material   characteristics        :damage    :protection  :surface_area

:steel                                  1              1             -2
:iron                                  -1              2             -1
```

Notes: here, I change the sign of surface area to match the one-side char_bounds in the model.

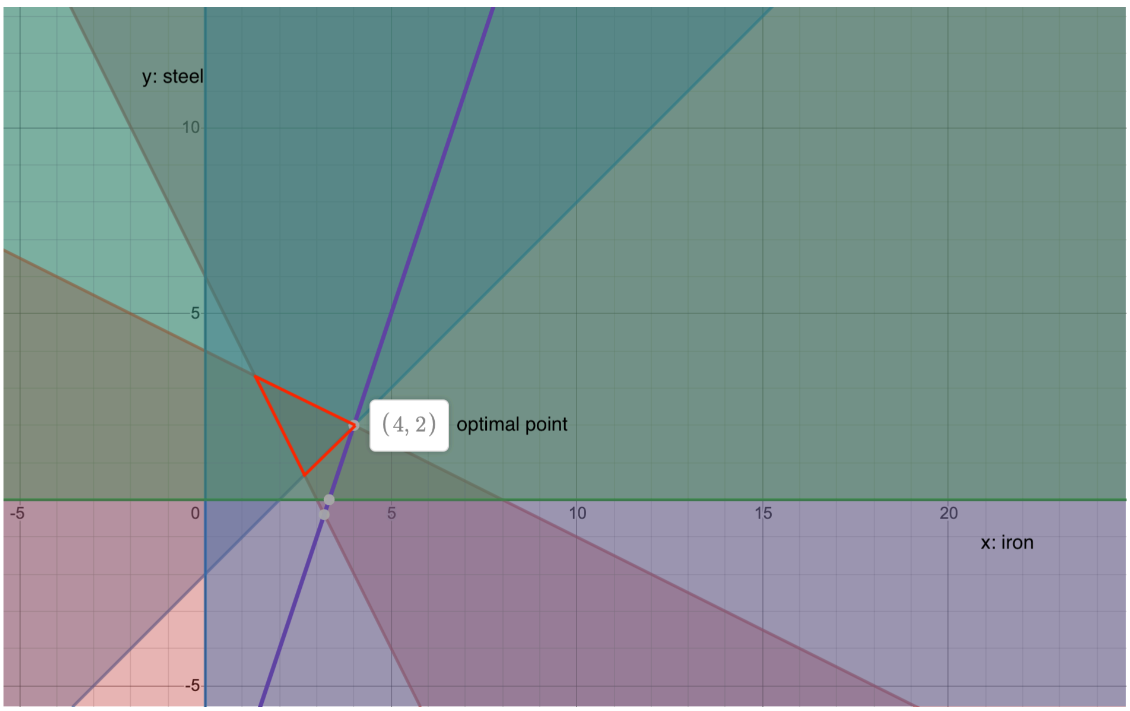
[25]:

```
# Model

# always specify which packages you're going to use
using JuMP, Clp

# create a new model object, specifying the solver
m = Model(Clp.Optimizer)

# variable object is now a Dictionary indexed over material types (elements are␣
↪variables)
@variable(m, material[materials] >= 0)

# use an expression object to calculate the total "slowness"
@expression(m, tot_slowness, sum(slowness[i] * material[i] for i in materials) )

# our material/char NamedArray allows us to create a Dictionary of constraints.
# indices are characteristics, and elements are constraint objects.
@constraint(m, constr[i in characteristics], sum(material_char_NA[t, i] *␣
↪material[t] for t in materials) >= char_bounds[i] )

# our objective is to minimize the total "slowness"
@objective(m, Min, tot_slowness)

# solve the instance of the problem
optimize!(m)

# display solution information
println("steel: ", value.(steel), " pounds.")
```

```
println("iron: ", value.(iron), " pounds.")
println("slowness(minimum): ", objective_value(m))
```

```
steel: 2.0 pounds.
iron: 4.0 pounds.
slowness(minimum): -10.0
Coin0506I Presolve 3 (0) rows, 2 (0) columns and 6 (0) elements
Clp0006I 0  Obj 0 Primal inf 5.9999999 (1) Dual inf 2.9999999 (1)
Clp0006I 2  Obj -10
Clp0000I Optimal - objective value -10
Clp0032I Optimal objective -10 - 2 iterations time 0.002
```

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



1.3 Problem 3

(a) Convert the problem to standard form.

[27]:
```
#x1 = y1
#x2 = -y2
#x3 = y3
#x4 = 5 + y4
```

objective function

$$-\max_{y_1,y_2,y_3,y_4} -\frac{1}{2}y_1 - 7y_2 - \frac{1}{3}y_3$$

constraints

$$-y_1 + y_2 + 2y_3 - 0.5y_4 \leq -7.5$$

$$-y_1 - 3y_2 \leq -2$$

$$y_4 \leq 0$$

$$y_1 \geq 0, y_2 \geq 0, y_3 \geq 0, y_4 \geq 0$$

(b) What are A, b, c,and x? Be sure to explain how the decision variables of your transformed LP relate to those of the original LP.

Decision variables relations:

$$x_1 = y_1, y_1 \geq 0$$

$$x_2 = -y_2, y_2 \geq 0$$

$$x_3 = y_3, y_3 \geq 0$$

$$x_4 = 5 + y_4, y_4 \geq 0, y_4 \leq 0$$

In transformed LP,

[29]:
```
using JuMP, Clp
m = Model(Clp.Optimizer)

var = [:y1, :y2, :y3, :y4]

A = [-1 1 2 -0.5; -1 -3 0 0; 0 0 0 1]
x = [var[1];var[2];var[3];var[4]]
c = [-0.5 -7 -(1/3) 0]'
b = [-7.5; -2; 0]
display(A)
display(x)
display(c)
display(b)
```

```
3×4 Array{Float64,2}:
 -1.0   1.0  2.0  -0.5
 -1.0  -3.0  0.0   0.0
  0.0   0.0  0.0   1.0


4-element Array{Symbol,1}:
 :y1
 :y2
 :y3
 :y4
```

```
4×1 LinearAlgebra.Adjoint{Float64,Array{Float64,2}}:
 -0.5
 -7.0
 -0.3333333333333333
  0.0
```

```
3-element Array{Float64,1}:
 -7.5
 -2.0
  0.0
```

(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.

[31]:
```
# always specify which packages you're going to use
using JuMP, Clp

# create a new model object, specifying the solver
m = Model(Clp.Optimizer)

# four nonnegative variables
@variable(m, y1 >= 0)
@variable(m, y2 >= 0)
@variable(m, y3 >= 0)
@variable(m, y4 == 0)

# two less than or equal to constraints
@constraint(m, - y1 + y2 + 2*y3 - 0.5*y4 <= - 7.5 )
@constraint(m, - y1 - 3*y2 <= -2 )

# maximize the objective
@objective(m, Max, - 0.5*y1 - 7*y2 - (1/3)*y3  )

# solve the standard form model
optimize!(m)

# display the model and solution
println(m)
println()
# remember to convert back to x1, x2, x3 and x4!
println("x1 = ", value(y1), ", y1 = ", value(y1))
println("x2 = ", value(-y2), ", y2 = ", value(y2) )
println("x3 = ", value(y3), ", y3 = ", value(y3) )
println("x4 = ", value(y4 + 5), ", y4 = ", value(y4) )
# remember min f(x) = -max -f(x), so we need to report the negative of our␣
 ↪objective value
```

```
println("objective = ", -objective_value(m) )
```

```
Max -0.5 y1 - 7 y2 - 0.3333333333333333 y3
Subject to
 -y1 + y2 + 2 y3 - 0.5 y4  -7.5
 -y1 - 3 y2  -2.0
 y4 = 0.0
 y1  0.0
 y2  0.0
 y3  0.0


x1 = 7.5, y1 = 7.5
x2 = 0.0, y2 = 0.0
x3 = 0.0, y3 = 0.0
x4 = 5.0, y4 = 0.0
objective = 3.75
Coin0506I Presolve 0 (-2) rows, 0 (-4) columns and 0 (-6) elements
Clp3002W Empty problem - 0 rows, 0 columns and 0 elements
Clp0000I Optimal - objective value -3.75
Coin0511I After Postsolve, objective -3.75, infeasibilities - dual 0 (0), primal
0 (0)
Clp0032I Optimal objective -3.75 - 0 iterations time 0.002, Presolve 0.00
```

1.4 Problem 4

(a) Formulate a linear program that MineCo can use to determine how many tons of ore to extract from each site today in order to maximize their total value. Give a general form (no numbers) of the math model.

objective function

$$\max_{mine_1, mine_2, ..., mine_{40}} mines.' \cdot vals$$

constraints

$$(mines.' \cdot attributes).'[n] \leq maxpercent[n], n = 1, 2, 3, ..., 7$$

$$(mines.' \cdot attributes).'[n] \geq minpercent[n], n = 1, 2, 3, ..., 7$$

$$mine_1 + mine_2 + ... + mine_{40} \leq 1000$$

$$mine_1, mine_2, ..., mine_{40} \geq 0$$

Terminologies: $mine_i$ denotes the total number of tons of ores from $mine_i$. mines denotes a vector $(40 * 1)$ consisting of $mine_i$, $i \in [1, 40]$. vals denotes a vector $(40 * 1)$ consisting of every value from each mine. attributes is a matrix $(40 * 7)$ consisting of all the mine-attribute pairs $(mine_i, attribute_j)$. maxpercent and minpercent are vectors $(7 * 1)$ which set bounds for each attribute.

(b) Implement and solve this instance of the model in Julia/JuMP. Display the optimal objective value and the optimal mining plan (in tons extracted from each site).

[32]:

```julia
# Problem data

using Pkg
Pkg.add("DataFrames")
Pkg.add("CSV")

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

#Load the data file
raw = CSV.read("mineco.csv");

# turn DataFrame into an array
mine_array = convert(Array,raw);

# the names of the DataFrame (header) are the attributes
attributes = names(raw[3:end]);

# create a list of mining sites from the mine array
sites = mine_array[3:end,1];

# create a dictionary of the value of each mining site's ore (per ton)
ore_val = Dict(zip(sites,mine_array[3:end,2]));

# create a dictionary of the value of min and max % of each attribute
min_percent = Dict(zip(attributes,mine_array[1,3:end]));
max_percent = Dict(zip(attributes,mine_array[2,3:end]));

# create a NamedArray that specifies the % of each attribute at each site
using NamedArrays
mine_attribute_matrix = mine_array[3:end,3:end]

# rows are sites, columns are attributes
mine_attribute_array = NamedArray(mine_attribute_matrix, (sites,␣
 ↪attributes),("sites","attributes"))


# note this syntax uses some deprecated commands, so you'll get a warning␣␣
 ↪ ↪message when you run it
# the code should still work (you can ignore the warning)
```

```
Resolving package versions…
 Updating `~/.julia/environments/v1.3/Project.toml`
[no changes]
 Updating `~/.julia/environments/v1.3/Manifest.toml`
[no changes]
Resolving package versions…
```

```
 Updating `~/.julia/environments/v1.3/Project.toml`
 [no changes]
 Updating `~/.julia/environments/v1.3/Manifest.toml`
 [no changes]
```

```
 Warning: `lastindex(df::AbstractDataFrame)` is deprecated, use `ncol(df)`
instead.
   caller = top-level scope at In[32]:15
 @ Core In[32]:15
 Warning: `getindex(df::DataFrame, col_inds::Union{AbstractVector, Regex,
Not})` is deprecated, use `df[:, col_inds]` instead.
   caller = top-level scope at In[32]:15
 @ Core In[32]:15
```

[32]:

```
40×7 Named Array{Any,2}
sites   attributes     Gold (%)  Carbon (%)  …    Coal (%)  Silica (%)

1                             6          16  …           7          12
2                            10          12              8          10
3                             4           9             19          19
4                            20           8              8          16
5                            14           9              8           8
6                            11           5             11          19
7                             3          14             17          10
8                            18           2             15          11
9                             7          13             10          11
10                           17          18              6          19
11                            8          13             18          15
12                            2          19              1          17

29                           14          13             20           8
30                           19          18              1          11
31                            7           7              4          11
32                           11           6              0           8
33                           14          16              4           3
34                            5           6             14           9
35                            4           6              8           6
36                            5          18              1          13
37                            8           6              7           5
38                           16          12              5           2
39                            6          15             15           7
40                           19           3  …           1          13
```

[33]:

```julia
# Math model

using JuMP, Clp

# create a new model object, specifying the solver
```

```
m = Model(Clp.Optimizer)

# variables are the list of the number (tons) of each site
@variable(m, num[sites] >= 0)

# maximize total value by summing (Value/ton * # site) for each site
@objective(m, Max, sum(ore_val[i] * num[i] for i in sites) )

# contraints are the min and max, and 1000 total tons
@constraint(m, meet_req[n in attributes], sum(mine_attribute_array[i,n] *␣
↪num[i] for i in sites) <= max_percent[n])
@constraint(m, meet_req2[n in attributes], sum(mine_attribute_array[i,n] *␣
↪num[i] for i in sites) >= min_percent[n])
@constraint(m, sum(num[i] for i in sites) <= 1000) #MineCo can extract up to␣
↪1000 total tons today

# solve the standard form model
optimize!(m)

# display the model and solution
println("objective = ", objective_value(m) )
println("How many tons of ore in each mining site should be mined today?")
for i in sites
    if value(num[i]) > 10e-5
        println(i ,":", value(num[i]))
    end
end
```

```
objective = 2544.5124821989257
How many tons of ore in each mining site should be mined today?
14:0.3089194104627416
17:0.1462433299589934
18:0.01427946399464688
20:0.2708165331228662
27:0.11942161522227668
34:0.23695159823619322
Coin0506I Presolve 12 (-3) rows, 40 (0) columns and 462 (-114) elements
Clp0006I 0  Obj -0 Primal inf 0.61578447 (5) Dual inf 474486.65 (40)
Clp0006I 10  Obj 2544.5125
Clp0000I Optimal - objective value 2544.5125
Coin0511I After Postsolve, objective 2544.5125, infeasibilities - dual 0 (0),
primal 0 (0)
Clp0032I Optimal objective 2544.512482 - 10 iterations time 0.002, Presolve 0.00
```

[ ]:
```
# other sites: 0 tons of ore
```