1. Stigler's Diet [10 pts]

by Roumen Guha, on Sunday, February 12th, 2017

True story! In 1945, American economist (and future Nobel laureate) George Stigler published a paper investigating the composition of an optimal diet; minimizing total cost while meeting the recommended daily allowance (RDA) of several nutrients. To answer this question, Stigler tabulated a list of 77 foods and their nutrient content for 9 nutrients: calories, protein, calcium, iron, vitamin A, thiamine, riboflavin, niacin, and ascorbic acid.

To make calculations easier, Stigler normalized his data so each row shows the nutrients contained in $1's worth of the given food item. Back then, $1 could buy you quite a lot! When Stigler posed his diet problem, the simplex method had not yet been invented. In his paper, he wrote: "... the procedure is experimental because there does not appear to be any direct method of finding the minimum of a linear function subject to linear conditions." Nevertheless, through a combination of "trial and error, mathematical insight, and agility", he eventually arrived at a solution: a diet costing only $39.93 per year. Though he confessed: "There is no reason to believe that the cheapest combination was found, for only a handful of the [many] possible combinations of commodities were examined."

(a) Formulate Stigler's diet problem as an LP and solve it. To get you started, Stigler's original data is provided in stigler.csv, and the IJulia notebook stigler.ipynb imports the data and puts it into a convenient array format. How does your cheapest diet compare in annual cost to Stigler's? What foods make up your optimal diet?

In [78]:
```
using NamedArrays      # make sure you run Pkg.add("NamedArrays") first!

# import Stigler's data set
raw = readcsv("stigler.csv")
(m, n) = size(raw)

n_nutrients = 2:n      # columns containing nutrient names
n_foods = 3:m          # rows containing food names

nutrients = raw[1, n_nutrients][:]   # the list of nutrients (convert to 1-D array)
foods = raw[n_foods, 1][:]           # the list of foods (convert to 1-D array)

# min_requirements[i] is the minimum daily requirement of nutrient i.
min_requirements = Dict(zip(nutrients, raw[2, n_nutrients]))

# data[f,i] is the amount of nutrient i contained in food f.
data = NamedArray(raw[n_foods, n_nutrients], (foods, nutrients), ("foods", "nutrients"))
;
```

In [79]:
```
using JuMP
m = Model()

@variable(m, quant_food[1:77] >= 0.0)
# Number of servings of food
for i = 1:9
    count[i] = 0
    for j = 1:77
        count[i] = count[i] + data[j,i]*quant_food[j]
    end
    @constraint(m, count[i] >= raw[2, n_nutrients][i])
end

@objective(m, Min, sum(quant_food))

solve(m)
```

In [68]:
```
getobjectivevalue(m)*365
```

Out[68]: 39.66173154546625

In [69]:
```
println("The optimal diet consists of:")
println("")

for i = 1:77
    t = getvalue(quant_food[i])
    if t > 1e-5
        println("item ", i, ": ", raw[i + 2, 1], " --> ", t)
    end
end
```

```
The optimal diet consists of:

item 1: Wheat Flour (Enriched) --> 0.029519061676488 27
item 30: Liver (Beef) --> 0.0018925572907052643
item 46: Cabbage --> 0.011214435246144865
item 52: Spinach --> 0.005007660466725203
item 69: Navy Beans, Dried --> 0.061028563526693246
```

(b) Suppose we wanted to find the cheapest diet that was also vegan and gluten-free. How much would that cost per year, and what foods would be used?

Of the 5 items that the optimal solution contains, note that 2 of the foods are not vegan nor are they gluten-free. Therefore, we introduce constraints to remove these items.

In [75]:
```
using JuMP
m2 = Model()

data[1, :] = zeros(9) # set wheat flour to have zero nutritional value
# interestingly enough, setting this to zero changed the solution enough that it
# was not necessary to set beef liver to zero. Go figure.

@variable(m2, quant_food[1:77] >= 0.0) # Number of servings of food

for i = 1:9
    count[i] = 0
    for j = 1:77
        count[i] = count[i] + data[j,i]*quant_food[j]
    end
    @constraint(m2, count[i] >= raw[2, n_nutrients][i])
end

@objective(m2, Min, sum(quant_food))

solve(m2)
```

Out[75]:
```
:Optimal
```

In [76]:
```
getobjectivevalue(m2)*365
```

Out[76]:
```
45.55617276801373
```

In [77]:
```
println("The optimal vegan and gluten-free diet consists of:")
println()

for i = 1:77
    t = getvalue(quant_food[i])
    if t > 1e-5
        println("item ", i, ": ", raw[i + 2, 1], " --> ", t)
    end
end
```

```
The optimal vegan and gluten-free diet consists of:

item 24: Lard --> 0.003609110117837147
item 46: Cabbage --> 0.011221011438294
item 52: Spinach --> 0.005355495313175868
item 69: Navy Beans, Dried --> 0.1046258153718265
```

2. Construction with Constraints [10 pts]

by Roumen Guha, on Sunday, February 12th, 2017

During the next 4 months, a construction firm must complete three projects. Each project has a deadline as well as labor requirements.

- Project 1 must be completed no later than 3 months from now and requires 8 worker-months of labor.
- Project 2 must be completed no later than 4 months from now and requires 10 worker-months of labor.
- Project 3 must be completed no later than 2 months from now and requires 12 worker-months of labor.

Project	**1**	**2**	**3**
Deadline (Months)	3	4	2
Required Labor (Worker-Months)	8	10	12

Each month, **8** workers are available. During a given month, no more than **6** workers can work on a single job. Determine whether all three projects can be completed on time.

Solution (without Julia)

Given the constraints, we can see an immediate solution. The first two months will have 6 workers working on Project 3 to complete it in 2 months. These first two months will have the remaining 2 available workers working on Project 1, so that it require only 4 worker-months of labor by the start of the third month. So, in month 3, 4 workers will work on Projects 1 and 2, and by the end of the third month Project 1 will be complete and Project 2 will require 6 more worker-months of labor. This leaves 6 workers working on Project 2 for the last month, allowing the project to meet its deadlines. We therefore know that the goal is at least attainable, if not surpassable.

Solution (with Julia)

In [24]:
```
using JuMP

available_monthly_workers = 8
max_workers_per_job = 6
required_labor = [8, 10, 12]

m = Model()

# arrays contain the number of workers in that month who worked on that project
@variable(m, 0 <= labor_project1[1:3] <= max_workers_per_job)
@variable(m, 0 <= labor_project2[1:4] <= max_workers_per_job)
@variable(m, 0 <= labor_project3[1:2] <= max_workers_per_job)

# set constraints on the distribution (and the deadlines, inherently) of work
@expression(m, labor_month1, labor_project1[1] + labor_project2[1] + labor_project3[1])
@constraint(m, 0 <= labor_month1 <= available_monthly_workers)
@expression(m, labor_month2, labor_project1[2] + labor_project2[2] + labor_project3[2])
@constraint(m, 0 <= labor_month2 <= available_monthly_workers)
@expression(m, labor_month3, labor_project1[3] + labor_project2[3])
@constraint(m, 0 <= labor_month3 <= available_monthly_workers)
@expression(m, labor_month4, labor_project2[4])
@constraint(m, 0 <= labor_month4 <= available_monthly_workers)

# set constraints on the amount of work needed for each project
@constraint(m, sum(labor_project1) == required_labor[1])
@constraint(m, sum(labor_project2) == required_labor[2])
@constraint(m, sum(labor_project3) == required_labor[3])

solve(m)
```

Out[24]:
```
:Optimal
```

In [20]:
```
getvalue(labor_project1)
```

Out[20]:
```
3-element Array{Float64,1}:
 0.0
 2.0
 6.0
```

In [21]:
```
getvalue(labor_project2)
```

Out[21]:
```
4-element Array{Float64,1}:
 2.0
 0.0
 2.0
 6.0
```

In [22]:
```
getvalue(labor_project3)
```

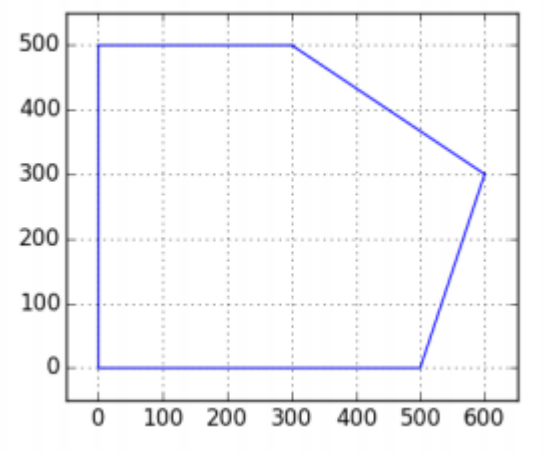
Out[22]:
```
2-element Array{Float64,1}:
 6.0
 6.0
```

It looks like Julia found a different solution than the one proposed earlier, but it is equally as optimal. Interestingly, we didn't need a maximize or minimize objective.

3. Museum Site Planning [10 pts]

by Roumen Guha, on Sunday, February 12th, 2017

A site is being investigated as a potential location for a new museum. An aerial plan of the site is shown in the figure below (in **feet**). The museum will have a **circular** footprint and law mandates that there be **at least 50 feet of clearance between the building and any road**. If we want the largest possible museum, where should it be located? What is its optimal radius? Re-plot the figure below along with the optimally designed museum.



In [ ]:

4. Electricity Grid with Storage [15 pts]

by Roumen Guha, on Sunday, February 12th, 2017

The town of Hamilton buys its electricity from the Powerco utility, which charges for electricity on an hourly basis. If less than 50 MWh is used during a given hour, then the cost is $100 per MWh. Any excess beyond 50 MWh used during the hour is charged at the higher rate of $400 per MWh. The maximum power that Powerco can provide in any given hour is 75 MWh. Here is what the average daily electricity demand looks like for Hamilton during the month of January:

Hour of Day (AM)	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Demand (MWh)	43	40	36	36	35	38	41	46	49	48	47	47

Hour of Day (PM)	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Demand (MWh)	48	46	45	47	50	63	75	75	72	66	57	50

The mayor of Hamilton is concerned because the high electricity use during evening hours is costing the city a lot of money. There is also risk of black-outs at around 7pm because the average demand is dangerously close to Powerco's 75 MWh limit. To address these issues, the mayor purchased a large battery with a storage capacity of 30 MWh. The idea is that extra electricity could be purchased early in the day (at the lower rate), stored in the battery, and used later in the day when demand (and prices) are high.

In [14]:
```
low_demand_price = 100  # when less than 50MWh is used per hour
high_demand_price = 400 # when more than 50MWh is used per hour

maximum_power_supplied = 75
battery_capacity = 30

# hourly demand in an average day
d = [43, 40, 36, 36, 35, 38, 41, 46, 49, 48, 47, 47, 48, 46, 45, 47, 50, 63, 75, 75, 72, 66, 57, 50]
;
```

(a) How much money can the town of Hamilton save per day thanks to the battery? Assume that the battery begins the day completely drained. Also, to be safe from possible black-outs, limit the amount of electricity purchased every hour to a maximum of 65 MWh.

In [54]:
```
daily_cost_without_battery = 0
for i in 1:24
    if d[i] <= 50
        daily_cost_without_battery = daily_cost_without_battery + d[i]*low_demand_price
    else
        daily_cost_without_battery = daily_cost_without_battery + d[i]*low_demand_price + (d[i] - 50)*high_demand_price
    end
end

using JuMP

m = Model()

max_hourly_energy = 65

@variable(m, 0 <= x[1:24] <= 50)  # energy produced with regular rate
@variable(m, 0 <= y[1:24] <= 65)  # energy produced with premium rate
@variable(m, 0 <= b[1:25] <= 30)  # energy stored in battery
@constraint(m, b[1] == 0)
@constraint(m, flow[i in 1:24], x[i] + y[i] + b[i] == d[i] + b[i+1])        # conservation of energy
@objective(m, Min, low_demand_price*sum(x) + high_demand_price*sum(y))      # minimize costs

solve(m)

println("Without the battery, the citizens of Hamilton would pay \$", daily_cost_without_battery, ".")
println("With the battery, the citizens of Hamilton would pay \$", getobjectivevalue(m), ".")

println()
println("The battery saves them \$", daily_cost_without_battery - getobjectivevalue(m), " daily.")
```

```
Without the battery, the citizens of Hamilton would pay $163200.
With the battery, the citizens of Hamilton would pay $143400.0.

The battery saves them $19800.0 daily.
```

(b) How much money would be saved if the battery had an infinite capacity? In this scenario, how much of the battery's capacity is actually used?

In [61]:
```
using JuMP

m2 = Model()

@variable(m2, 0 <= x[1:24] <= 50) # energy produced with regular rate
@variable(m2, 0 <= y[1:24] <= 65) # energy produced with premium rate
@variable(m2, b[1:25] >= 0)       # energy stored in battery
@constraint(m2, b[1] == 0)
@constraint(m2, flow[i in 1:24], x[i] + y[i] + b[i] == d[i] + b[i+1])        # conservation of energy
@objective(m2, Min, low_demand_price*sum(x) + high_demand_price*sum(y))      # minimize costs

solve(m2)

println("With a battery of infinite capacity, the citizens of Hamilton would pay \$", getobjectivevalue(m2), ".")

println()
println("The infinite battery saves them \$", daily_cost_without_battery - getobjectivevalue(m2), " daily.")

print(getvalue(b))
```

```
With a battery of infinite capacity, the citizens of Hamilton would pay $120000.0.

The infinite battery saves them $43200.0 daily.
[0.0,7.0,17.0,31.0,45.0,60.0,72.0,81.0,85.0,86.0,88.0,91.0,94.0,96.0,100.0,105.0,108.0,108.0,95.0,70.0,45.0,23.0,7.0,0.
0,0.0]
```

Out[61]: (25,)

(c) Make a plot that shows (i) the typical energy demand vs time of day (ii) the electricity purchased using the strategy found in part **(a)** vs time of day, and (iii) the battery capacity used as a function of time (draw all three plots on the same axes).

In [63]:
```
using PyPlot
x = 1:5
y = [3, 5, 4, 2, 6]
bar(x,y,align="center")
```

```
ArgumentError: haskey of NULL PyObject

 in haskey(::PyCall.PyObject, ::String) at C:\Users\roume\.julia\v0.5\PyCall\src\PyCall.jl:286
 in #bar#19(::Array{Any,1}, ::Function, ::UnitRange{Int64}, ::Vararg{Any,N}) at C:\Users\roume\.julia\v0.5\PyPlot\src\P
yPlot.jl:169
 in (::PyPlot.#kw##bar)(::Array{Any,1}, ::PyPlot.#bar, ::UnitRange{Int64}, ::Array{Int64,1}) at .\<missing>:0
```

(d) Comment on whether the solutions you found are unique. Are other solutions possible? Why? Suggest a way of finding another optimal solution.

The solution found with an upper limit for the battery's capacity is probably not unique, simply because we are held back by the constraint of the battery's capacity.

For the solution found with a battery of infinite capacity, there is only really 1 possible solution with any given set of paramaters, therefore it is unique. But if we added other constraints, such as by taking into account creep-discharge typical of such large batteries and the associated cost, the solution would change, but under these new constraints, it should still be unique.

In [ ]:
```

```