Homework 4 Solutions

2024-11-03

> Due Date
>
> Thursday, 11/07/23, 9:00pm

Overview

Load Environment

The following code loads the environment and makes sure all needed packages are installed. This should be at the start of most Julia scripts.

```
import Pkg
Pkg.activate(@__DIR__)
Pkg.instantiate()
```

```
using JuMP
using HiGHS
using DataFrames
using Plots
using Measures
using CSV
using MarkdownTables
```

Problems (Total: 50/60 Points)

Problem 1 (20 points)

The first step in formulating the optimization problem is to identify the decision variables. A straightforward set of variables are S_i, C_i, and W_i, where these are the hectares of soybeans,

corn, and wheat treated with pesticide rate $i = 0, 1, 2$ kg/ha.

> **Alternative Variable Specifications**
>
> We could also combine these crop variables into a single matrix variable A_{ji}, where j is an index for the crop. This would let us specify all of the objectives and constraints using matrix notation, but may make the problem harder to read and debug. The final problem will be equivalent but the writeup may look slightly different.

Next, let's formulate the optimization problem. The goal is to maximize profit, so we want to calculate the profit associated with any given planting and pesticide strategy.

The profit from producing soybeans is

$$\begin{aligned}
&\overbrace{0.36\,(2900S_0 + 3800S_1 + 4400S_2)}^{revenue} - \overbrace{350\,(S_0 + S_1 + S_2)}^{productioncost} - \overbrace{70\,(S_1 + 2S_2)}^{pesticidecost} \\
&= (1044 - 350)S_0 + (1368 - 350 - 70)S_1 + (1584 - 350 - 140)S_2 \\
&= 694S_0 + 948S_1 + 1094S_2.
\end{aligned}$$

Similarly, the profit from producing wheat is

$$665W_0 + 757W_1 + 714W_2$$

and from corn

$$908C_0 + 1014C_1 + 1208C_2.$$

So the overall objective is

$$\max_{S_i, W_i, C_i} 694S_0 + 948S_1 + 1094S_2 + 665W_0 + 757W_1 + 714W_2 + 908C_0 + 1014C_1 + 1208C_2.$$

For constraints, we have the non-negativity constraints

$$S_i, W_i, C_i \geq 0.$$

The total planted area cannot exceed 130 ha, so

$$S_0 + S_1 + S_2 + C_0 + C_1 + C_2 + W_0 + W_1 + W_2 \leq 130.$$

We would also get no revenue from producing more than 250,000 kg of any crop, so we will add that in as a set of constraints:

$$\begin{aligned}
2900S_0 + 3800S_1 + 4400S_2 &\leq 250000 \\
3500W_0 + 4100W_1 + 4200W_2 &\leq 250000 \\
5900C_0 + 6700C_1 + 7900C_2 &\leq 250000.
\end{aligned}$$

Finally, we have the pesticide application constraints. These are a little tricky because the most direct way of writing them down does not result in a linear constraint, so we will need to do some algebra. Let's illustrate this with the soybean constraint. The average application rate cannot exceed 0.8 kg/ha, which means

$$\frac{S_1 + 2S_2}{S_0 + S_1 + S_2} \leq 0.8.$$

As noted, this is not linear, but we can turn it into a linear constraint by multiplying through by $S_0 + S_1 + S_2$ and moving everything over to the left-hand side. This yields

$$-0.8S_0 + 0.2S_1 + 1.2S_2 \leq 0.$$

Similarly, the wheat and corn constraints are, respectively,

$$\begin{aligned} -0.7W_0 + 0.3W_1 + 1.3W_2 &\leq 0 \\ -0.6C_0 + 0.4C_1 + 1.4C_2 &\leq 0. \end{aligned}$$

Now let's implement this program in `JuMP`. We will use matrix-vector notation for our implementation, but you could also write out the constraints one variable at a time as well; the disadvantage of this is that it does not scale well as the number of variables increases.

```
crop_model = Model(HiGHS.Optimizer)
@variable(crop_model, S[1:3] >= 0)                                          ①
@variable(crop_model, W[1:3] >= 0)
@variable(crop_model, C[1:3] >= 0)
@objective(crop_model, Max, [694; 948; 1094]' * S + [665; 757; 714]' * W +
 ↪  [908; 1014; 1208]' * C) # <2>
@constraint(crop_model, land, sum(S) + sum(W) + sum(C) <= 130)
@constraint(crop_model, soy_demand, [2900; 3800; 4400]' * S <= 250_000)
@constraint(crop_model, wheat_demand, [3500; 4100; 4200]' * W <= 250_000)
@constraint(crop_model, corn_demand, [5900; 6700; 7900]' * C <= 250_000)
@constraint(crop_model, soy_pesticide, [-0.8; 0.2; 1.2]' * S <= 0)
@constraint(crop_model, wheat_pesticide, [-0.7; 0.3; 1.3]' * W <= 0)
@constraint(crop_model, corn_pesticide, [-0.6; 0.4; 1.4]' * C <= 0)
print(crop_model)                                                           ③
```

① Even though we defined the indices i to start at 0, Julia's indexing starts at 1, so we will need to bear that in mind when we formulate the problem and interpret the solution.

② When using matrix-vector syntax, be careful about the dimensions and making sure they match; any mistakes should come out as you formulate the model and print it out, but may be hard to tell if the problem is sufficiently large. You could also directly compute the dot product with syntax like `sum([694; 948; 1094] .* S)`.

③ I would always `print` your model just to make sure there are no obvious typos. I made one while writing up the solution and printing the model let me see it!

```
Max 694 S[1] + 948 S[2] + 1094 S[3] + 665 W[1] + 757 W[2] + 714 W[3] + 908 C[1] + 1014 C[2] +
Subject to
 land : S[1] + S[2] + S[3] + W[1] + W[2] + W[3] + C[1] + C[2] + C[3]   130
 soy_demand : 2900 S[1] + 3800 S[2] + 4400 S[3]   250000
 wheat_demand : 3500 W[1] + 4100 W[2] + 4200 W[3]   250000
 corn_demand : 5900 C[1] + 6700 C[2] + 7900 C[3]   250000
 soy_pesticide : -0.8 S[1] + 0.2 S[2] + 1.2 S[3]   0
 wheat_pesticide : -0.7 W[1] + 0.3 W[2] + 1.3 W[3]   0
 corn_pesticide : -0.6 C[1] + 0.4 C[2] + 1.4 C[3]   0
 S[1]   0
 S[2]   0
 S[3]   0
 W[1]   0
 W[2]   0
 W[3]   0
 C[1]   0
 C[2]   0
 C[3]   0
```

Now, let's find the solution.

```
optimize!(crop_model)
```

```
Running HiGHS 1.7.2 (git hash: 5ce7a2753): Copyright (c) 2024 HiGHS under MIT licence terms
Coefficient ranges:
  Matrix [2e-01, 8e+03]
  Cost   [7e+02, 1e+03]
  Bound  [0e+00, 0e+00]
  RHS    [1e+02, 2e+05]
Presolving model
7 rows, 9 cols, 27 nonzeros  0s
7 rows, 9 cols, 27 nonzeros  0s
Presolve : Reductions: rows 7(-0); columns 9(-0); elements 27(-0) - Not reduced
Problem not reduced by presolve: solving the LP
Using EKK dual simplex solver - serial
  Iteration        Objective     Infeasibilities num(sum)
          0    -1.8170297740e+02 Ph1: 7(28.3601); Du: 9(181.703) 0s
          7     1.1674116702e+05 Pr: 0(0) 0s
```

```
Model   status      : Optimal
Simplex   iterations: 7
Objective value     :  1.1674116702e+05
HiGHS run time      :          0.00
```

We found a solution, which is a good sign that we didn't mis-specify our constraints (which can result in an unbounded problem, which would throw an error). The optimal planting and pesticide strategy is then:

```
@show value.(S);
@show value.(W);
@show value.(C);
```

```
value.(S) = [13.812154696132604, 55.24861878453038, 0.0]
value.(W) = [6.7433064173395625, 15.734381640458977, 0.0]
value.(C) = [26.92307692307692, 0.0, 11.538461538461547]
```

Summarizing in a table:

Pesticide Rate (kg/ha)	Soy Area (ha)	Corn Area (ha)	Wheat Area (ha)
0	13.8	6.7	26.9
1	55.2	15.7	0
2	0	0	11.5

The resulting profit is:

```
@show round(objective_value(crop_model); digits=0);
```

```
round(objective_value(crop_model); digits = 0) = 116741.0
```

To evaluate whether the farmer should buy the land, we can look at the shadow price.

```
@show shadow_price(land);
```

```
shadow_price(land) = 729.4
```

The shadow price is non-zero, so the land constraint is binding and the farmer could make more money by buying land. Since the amount of land is relatively small, we can take the shadow price and multiply it by 10 to get an estimate of the value to the farmer of buying the land, which is approximately \$7294.

Does this result make sense? Looking at the other constraints, wheat is the only crop for which the demand constraint isn't binding, so any additional land would have to be used to grow wheat. From the solution, it seems clear that there is no point in increasing W_2, as wheat treated with two kg/ha of pesticide is less profitable than wheat treated with one, and the additional pesticide would limit the amount we could grow. As the wheat pesticide constraint is already binding, to stay in compliance we would need to allocate no more than 7 ha to W_1 and 3 ha to W_0 based on the constraint. And then if we plug those values into the wheat profit equation that went into our objective, $\$757 \times 7 + \$665 \times 3 = \$7294$, which is the same as the additional profit we estimated using the shadow price.

Problem 2 (30 points)

First, let's load the data for demand, the generators, and renewable variability.

```
# load the data, pull Zone C, and reformat the DataFrame
NY_demand = DataFrame(CSV.File("data/2013_hourly_load_NY.csv"))                ①
rename!(NY_demand, :"Time Stamp" => :Date)
demand = NY_demand[:, [:Date, :C]]
rename!(demand, :C => :Demand)
demand[:, :Hour] = 1:nrow(demand)

# generator data
gens = DataFrame(CSV.File("data/generators.csv"))

# load capacify factors into a DataFrame
cap_factor = DataFrame(CSV.File("data/wind_solar_capacity_factors.csv"))
```

① We load these tabular files into a `DataFrame` to improve our ability to manipulate them.

	Hour	Wind	Solar
	Int64	Float64	Float64
1	1	0.0456225	0.0
2	2	0.0908019	0.0
3	3	0.177524	0.0
4	4	0.196106	0.0
5	5	0.163481	0.0
6	6	0.129312	0.0
7	7	0.131448	0.0
8	8	0.162792	0.0
9	9	0.341911	0.115556
10	10	0.222848	0.530593
11	11	0.030222	0.747114
12	12	0.0143295	0.874793
13	13	0.0377381	0.94319
14	14	0.0552232	0.962
15	15	0.00819863	0.948126
16	16	0.000661809	0.852887
17	17	0.00255233	0.687016
18	18	0.0296442	0.440206
19	19	0.0534046	0.0
20	20	0.276383	0.0
21	21	0.296339	0.0
22	22	0.278992	0.0
23	23	0.28697	0.0
24	24	0.331498	0.0
...	...	...	...

From lecture, the decision variables for a "greenfield" capacity expansion are:

- x_g: capacity installed (MW) for generator type g;
- $y_{g,t}$: generated power (MWh) by generator type g in hour t;
- NSE_t: non-served demand (MWh) in hour t.

The optimization problem is:

$$
\begin{aligned}
\min_{x,y,NSE} \quad & \sum_{g\in\mathcal{G}} \text{FixedCost}_g \times x_g + \sum_{t\in\mathcal{T}}\sum_{g\in\mathcal{G}} \text{VarCost}_g \times y_{g,t} \\
& + \sum_{t\in\mathcal{T}} \text{NSECost} \times NSE_t \\
\text{subject to:} \quad & \sum_{g\in\mathcal{G}} y_{g,t} + NSE_t \geq d_t \qquad \forall t \in \mathcal{T} \\
& y_{g,t} \leq x_g \times c_{g,t} \qquad \forall g \in G, \forall t \in \mathcal{T} \\
& x_g, y_{g,t}, NSE_t \geq 0 \qquad \forall g \in G, \forall t \in \mathcal{T},
\end{aligned}
$$

where d_t is the demand in hour t, and $c_{g,t}$ is the capacity factor in hour t for generator class g.

To put this into `JuMP`, the our first task is to decide how we want to handle the difference between the renewable generating technologies (which have time-varying capacity factors) and the thermal technologies (which have constant capacity factors). We can either create a joint capacity factor `DataFrame` which we can then use to construct all of our capacity constraints, or we can create two different capacity constraints, one for renewables and one for non-renewables. The eventual problem will be the same, but the code would look slightly different. In this solution, we will do take the former approach to show what it looks like.

To construct a single capacity factor `DataFrame`:

```
# define sets
G = 1:nrow(gens)                                                    ①
T = 1:nrow(demand)

# capacity factor matrix
cf_constant = ones(T[end], G[end-2])                                ②
# set geothermal capacity
cf_constant[:, 1] .= 0.85                                           ③
# append wind and solar capacity factors
cf = hcat(cf_constant, cap_factor[!, :Wind], cap_factor[!, :Solar]) ④
```

① We need to define these index sets in any case to help set up arrays of constraints (we don't, for example, want to define all of the individual demand constraints for every hour individually).

② The last two generators in the `gens` DataFrame are solar and wind, so we want to drop those as we will replace them with the varying capacity factor data that we loaded. The `end` keyword is just syntax to automate looking up the last index, so we can just subtract values from it.

③ We broadcast the `=` operator to assign element-wise; Julia will throw an error if we try to assign a scalar to a vector because it doesn't know if we want to do element-wise assigning or if this is an error. Requiring broadcasting forces us to be explicit about our intent.

④ You can use either `cap_factor[!, :Wind]` or `cap_factor[:, :Wind]` to refer to a "slice" of the DataFrame with column name `Wind`. The difference is that `!` does not allocate a temporary DataFrame, which makes it more memory efficient but also means it can't be manipulated. You could also use a column index number instead of the name if you didn't have meaningful names or didn't know what they were.

```
8760×6 Matrix{Float64}:
 0.85  1.0  1.0  1.0  0.0456225  0.0
 0.85  1.0  1.0  1.0  0.0908019  0.0
 0.85  1.0  1.0  1.0  0.177524   0.0
 0.85  1.0  1.0  1.0  0.196106   0.0
 0.85  1.0  1.0  1.0  0.163481   0.0
 0.85  1.0  1.0  1.0  0.129312   0.0
 0.85  1.0  1.0  1.0  0.131448   0.0
 0.85  1.0  1.0  1.0  0.162792   0.0
 0.85  1.0  1.0  1.0  0.341911   0.115556
 0.85  1.0  1.0  1.0  0.222848   0.530593

 0.85  1.0  1.0  1.0  0.249119   0.560314
 0.85  1.0  1.0  1.0  0.171991   0.445911
 0.85  1.0  1.0  1.0  0.234215   0.232936
 0.85  1.0  1.0  1.0  0.32073    0.0
 0.85  1.0  1.0  1.0  0.166379   0.0
 0.85  1.0  1.0  1.0  0.252252   0.0
 0.85  1.0  1.0  1.0  0.276054   0.0
 0.85  1.0  1.0  1.0  0.111131   0.0
 0.85  1.0  1.0  1.0  0.208158   0.0
```

Now we can implement our model.

```
# define NSECost
NSECost = 10_000

# set up model object
gencap = Model(HiGHS.Optimizer) # use the HiGHS LP solver

# define variables
@variable(gencap, x[g in G] >= 0) # installed capacity
```

```
@variable(gencap, y[g in G, t in T] >= 0) # generated power
@variable(gencap, nse[t in T] >= 0) # unserved energy

# define objective: minimize sum of fixed costs, variable costs of
 ↪  generation,
# and non-served energy penalty
@objective(gencap, Min, sum(gens[!, :FixedCost] .* x) +
    sum(gens[!, :VarCost] .* [sum(y[g, :]) for g in G]) +
    NSECost * sum(nse))

# define constraints
@constraint(gencap, capacity[g in G, t in T], y[g, t] <= x[g] * cf[t, g]) #
 ↪  capacity constraint
@constraint(gencap, demand_met[t in T], sum(y[:, t]) + nse[t] >=
 ↪  demand.Demand[t]) # demand constraint
print(gencap)                                                              ①
```

① Notice that here we've hit the limit of what we can read directly from the model printing, but hopefully it gives us an idea of how everything looks.

```
Min 450000 x[1] + 220000 x[2] + 82000 x[3] + 65000 x[4] + 91000 x[5] + 70000 x[6] + 24 y[2,1]
Subject to
 demand_met[1] : y[1,1] + y[2,1] + y[3,1] + y[4,1] + y[5,1] + y[6,1] + nse[1]   1678.3
 demand_met[2] : y[1,2] + y[2,2] + y[3,2] + y[4,2] + y[5,2] + y[6,2] + nse[2]   1596.7
 demand_met[3] : y[1,3] + y[2,3] + y[3,3] + y[4,3] + y[5,3] + y[6,3] + nse[3]   1522.8
 demand_met[4] : y[1,4] + y[2,4] + y[3,4] + y[4,4] + y[5,4] + y[6,4] + nse[4]   1497.7
 demand_met[5] : y[1,5] + y[2,5] + y[3,5] + y[4,5] + y[5,5] + y[6,5] + nse[5]   1507.8
 demand_met[6] : y[1,6] + y[2,6] + y[3,6] + y[4,6] + y[5,6] + y[6,6] + nse[6]   1540.2
 demand_met[7] : y[1,7] + y[2,7] + y[3,7] + y[4,7] + y[5,7] + y[6,7] + nse[7]   1591.2
 demand_met[8] : y[1,8] + y[2,8] + y[3,8] + y[4,8] + y[5,8] + y[6,8] + nse[8]   1657.3
 demand_met[9] : y[1,9] + y[2,9] + y[3,9] + y[4,9] + y[5,9] + y[6,9] + nse[9]   1677.1
 demand_met[10] : y[1,10] + y[2,10] + y[3,10] + y[4,10] + y[5,10] + y[6,10] + nse[10]   1809.
 demand_met[11] : y[1,11] + y[2,11] + y[3,11] + y[4,11] + y[5,11] + y[6,11] + nse[11]   1895.
 demand_met[12] : y[1,12] + y[2,12] + y[3,12] + y[4,12] + y[5,12] + y[6,12] + nse[12]   1944.
 demand_met[13] : y[1,13] + y[2,13] + y[3,13] + y[4,13] + y[5,13] + y[6,13] + nse[13]   1959.
 demand_met[14] : y[1,14] + y[2,14] + y[3,14] + y[4,14] + y[5,14] + y[6,14] + nse[14]   1949.
 demand_met[15] : y[1,15] + y[2,15] + y[3,15] + y[4,15] + y[5,15] + y[6,15] + nse[15]   1937.
 demand_met[16] : y[1,16] + y[2,16] + y[3,16] + y[4,16] + y[5,16] + y[6,16] + nse[16]   1952
 demand_met[17] : y[1,17] + y[2,17] + y[3,17] + y[4,17] + y[5,17] + y[6,17] + nse[17]   2061.
 demand_met[18] : y[1,18] + y[2,18] + y[3,18] + y[4,18] + y[5,18] + y[6,18] + nse[18]   2256.
 demand_met[19] : y[1,19] + y[2,19] + y[3,19] + y[4,19] + y[5,19] + y[6,19] + nse[19]   2273.
 demand_met[20] : y[1,20] + y[2,20] + y[3,20] + y[4,20] + y[5,20] + y[6,20] + nse[20]   2251.
```

```
 demand_met[21] : y[1,21] + y[2,21] + y[3,21] + y[4,21] + y[5,21] + y[6,21] + nse[21]   2176.
 demand_met[22] : y[1,22] + y[2,22] + y[3,22] + y[4,22] + y[5,22] + y[6,22] + nse[22]   2038.
 demand_met[23] : y[1,23] + y[2,23] + y[3,23] + y[4,23] + y[5,23] + y[6,23] + nse[23]   1908.
 demand_met[24] : y[1,24] + y[2,24] + y[3,24] + y[4,24] + y[5,24] + y[6,24] + nse[24]   1800.
 demand_met[25] : y[1,25] + y[2,25] + y[3,25] + y[4,25] + y[5,25] + y[6,25] + nse[25]   1742.
 demand_met[26] : y[1,26] + y[2,26] + y[3,26] + y[4,26] + y[5,26] + y[6,26] + nse[26]   1711.
 demand_met[27] : y[1,27] + y[2,27] + y[3,27] + y[4,27] + y[5,27] + y[6,27] + nse[27]   1665.
 demand_met[28] : y[1,28] + y[2,28] + y[3,28] + y[4,28] + y[5,28] + y[6,28] + nse[28]   1690.
 demand_met[29] : y[1,29] + y[2,29] + y[3,29] + y[4,29] + y[5,29] + y[6,29] + nse[29]   1741.
 demand_met[30] : y[1,30] + y[2,30] + y[3,30] + y[4,30] + y[5,30] + y[6,30] + nse[30]   1864
 demand_met[31] : y[1,31] + y[2,31] + y[3,31] + y[4,31] + y[5,31] + y[6,31] + nse[31]   2115.
 demand_met[32] : y[1,32] + y[2,32] + y[3,32] + y[4,32] + y[5,32] + y[6,32] + nse[32]   2253.
 demand_met[33] : y[1,33] + y[2,33] + y[3,33] + y[4,33] + y[5,33] + y[6,33] + nse[33]   2279
 demand_met[34] : y[1,34] + y[2,34] + y[3,34] + y[4,34] + y[5,34] + y[6,34] + nse[34]   2287.
 demand_met[35] : y[1,35] + y[2,35] + y[3,35] + y[4,35] + y[5,35] + y[6,35] + nse[35]   2294.
 demand_met[36] : y[1,36] + y[2,36] + y[3,36] + y[4,36] + y[5,36] + y[6,36] + nse[36]   2298.
 demand_met[37] : y[1,37] + y[2,37] + y[3,37] + y[4,37] + y[5,37] + y[6,37] + nse[37]   2256.
 demand_met[38] : y[1,38] + y[2,38] + y[3,38] + y[4,38] + y[5,38] + y[6,38] + nse[38]   2243.
 demand_met[39] : y[1,39] + y[2,39] + y[3,39] + y[4,39] + y[5,39] + y[6,39] + nse[39]   2225.
 demand_met[40] : y[1,40] + y[2,40] + y[3,40] + y[4,40] + y[5,40] + y[6,40] + nse[40]   2231.
 demand_met[41] : y[1,41] + y[2,41] + y[3,41] + y[4,41] + y[5,41] + y[6,41] + nse[41]   2314.
 demand_met[42] : y[1,42] + y[2,42] + y[3,42] + y[4,42] + y[5,42] + y[6,42] + nse[42]   2485.
 demand_met[43] : y[1,43] + y[2,43] + y[3,43] + y[4,43] + y[5,43] + y[6,43] + nse[43]   2496.
 demand_met[44] : y[1,44] + y[2,44] + y[3,44] + y[4,44] + y[5,44] + y[6,44] + nse[44]   2464.
 demand_met[45] : y[1,45] + y[2,45] + y[3,45] + y[4,45] + y[5,45] + y[6,45] + nse[45]   2404.
 demand_met[46] : y[1,46] + y[2,46] + y[3,46] + y[4,46] + y[5,46] + y[6,46] + nse[46]   2273.
 demand_met[47] : y[1,47] + y[2,47] + y[3,47] + y[4,47] + y[5,47] + y[6,47] + nse[47]   2111.
 demand_met[48] : y[1,48] + y[2,48] + y[3,48] + y[4,48] + y[5,48] + y[6,48] + nse[48]   1974.
 demand_met[49] : y[1,49] + y[2,49] + y[3,49] + y[4,49] + y[5,49] + y[6,49] + nse[49]   1882.
 demand_met[50] : y[1,50] + y[2,50] + y[3,50] + y[4,50] + y[5,50] + y[6,50] + nse[50]   1835.
[[...122546 constraints skipped...]]
 nse[8711]   0
 nse[8712]   0
 nse[8713]   0
 nse[8714]   0
 nse[8715]   0
 nse[8716]   0
 nse[8717]   0
 nse[8718]   0
 nse[8719]   0
 nse[8720]   0
 nse[8721]   0
 nse[8722]   0
```

```
nse[8723]   0
nse[8724]   0
nse[8725]   0
nse[8726]   0
nse[8727]   0
nse[8728]   0
nse[8729]   0
nse[8730]   0
nse[8731]   0
nse[8732]   0
nse[8733]   0
nse[8734]   0
nse[8735]   0
nse[8736]   0
nse[8737]   0
nse[8738]   0
nse[8739]   0
nse[8740]   0
nse[8741]   0
nse[8742]   0
nse[8743]   0
nse[8744]   0
nse[8745]   0
nse[8746]   0
nse[8747]   0
nse[8748]   0
nse[8749]   0
nse[8750]   0
nse[8751]   0
nse[8752]   0
nse[8753]   0
nse[8754]   0
nse[8755]   0
nse[8756]   0
nse[8757]   0
nse[8758]   0
nse[8759]   0
nse[8760]   0
```

Now we optimize.

```
optimize!(gencap)
```

```
Running HiGHS 1.7.2 (git hash: 5ce7a2753): Copyright (c) 2024 HiGHS under MIT licence terms
Coefficient ranges:
  Matrix [5e-05, 1e+00]
  Cost   [2e+01, 4e+05]
  Bound  [0e+00, 0e+00]
  RHS    [1e+03, 3e+03]
Presolving model
56856 rows, 56862 cols, 153048 nonzeros  0s
56853 rows, 56859 cols, 153042 nonzeros  0s
Presolve : Reductions: rows 56853(-4467); columns 56859(-4467); elements 153042(-8934)
Solving the presolved LP
Using EKK dual simplex solver - serial
  Iteration        Objective     Infeasibilities num(sum)
          0     0.0000000000e+00 Pr: 8760(4.1747e+06) 0s
      42008     6.5458487038e+08 Pr: 0(0); Du: 0(2.80224e-10) 2s
Solving the original LP from the solution after postsolve
Model   status      : Optimal
Simplex   iterations: 42008
Objective value     :  6.5458487038e+08
HiGHS run time      :          1.88
```

We can find how much generating capacity we want to build for each plant type by querying the relevant decision variable `x`. We will turn this into a DataFrame to make the presentation easier.

```
built_cap = value.(x).data
DataFrame(Plant=gens.Plant, Capacity=round.(built_cap; digits=0))
```

	Plant	Capacity
	String15	Float64
1	Geothermal	-0.0
2	Coal	0.0
3	NG CCGT	1658.0
4	NG CT	880.0
5	Wind	485.0
6	Solar	1958.0

Similarly, we can find the total amount of non-served energy by adding up `nse[t]`.

```
@show sum(value.(nse).data);
```

```
sum(value.(nse).data) = 256.83377442738583
```

So this plan results in 257 MWh of non-served energy throughout the year.

Finally, to get the total cost of the system we use `objective_value`:

```
@show objective_value(gencap);
```

```
objective_value(gencap) = 6.545848703815409e8
```

So the cost of operating this system (fixed and variable costs) for a year is $6.5e8.

Next, to find the total annual generation from each plant, we want to sum up the values of the variable `y` along the time dimension.

```
annual_gen = [sum(value.(y[g, :]).data) for g in G]
```

```
6-element Vector{Float64}:
      0.0
      0.0
      8.65664708400426e6
 449375.8891095482
      1.4272043521322047e6
      5.83442174097956e6
```

Notice that this means that the cost of the system per MWh generated is $40/MWh.

We can then convert this into fractions of total generation, which we can compare to fractions of built capacity.

```
annual_gen_frac = annual_gen ./ sum(annual_gen)
built_frac = built_cap ./ sum(built_cap)
DataFrame(Plant=gens[!, :Plant], Built_Perc=100 * round.(built_frac;
 ↪  digits=2), Generated_Perc=100 * round.(annual_gen_frac; digits=2))
```

	Plant	Built_Perc	Generated_Perc
	String15	Float64	Float64
1	Geothermal	-0.0	0.0
2	Coal	0.0	0.0
3	NG CCGT	33.0	53.0
4	NG CT	18.0	3.0
5	Wind	10.0	9.0
6	Solar	39.0	36.0

One observation is that that we have to overbuild the fraction of combustion turbine gas plants (NG CT) relative to the fraction of times in which they are used, as these are needed when wind and solar is low, but otherwise are less commonly used. We also have to slightly overbuild wind and solar relative to the power generated by these technologies, as although they are free to generate, they can be severely constrained in terms of capacity in a given hour.

Finally, to plot the electricity price for each hour, we can look at the absolute value of the shadow prices for the demand constraints (absolute value since the shadow prices are negative, as "relaxing" the demand constraint by reducing the demand by one MWh reduces the objective).

```
elec_price = abs.(shadow_price.(demand_met).data)
p = plot(demand.Hour, elec_price, xlabel="Hour of Year", ylabel="Electricity
 ↪  Price (\$/MWh)", label=:false)
```

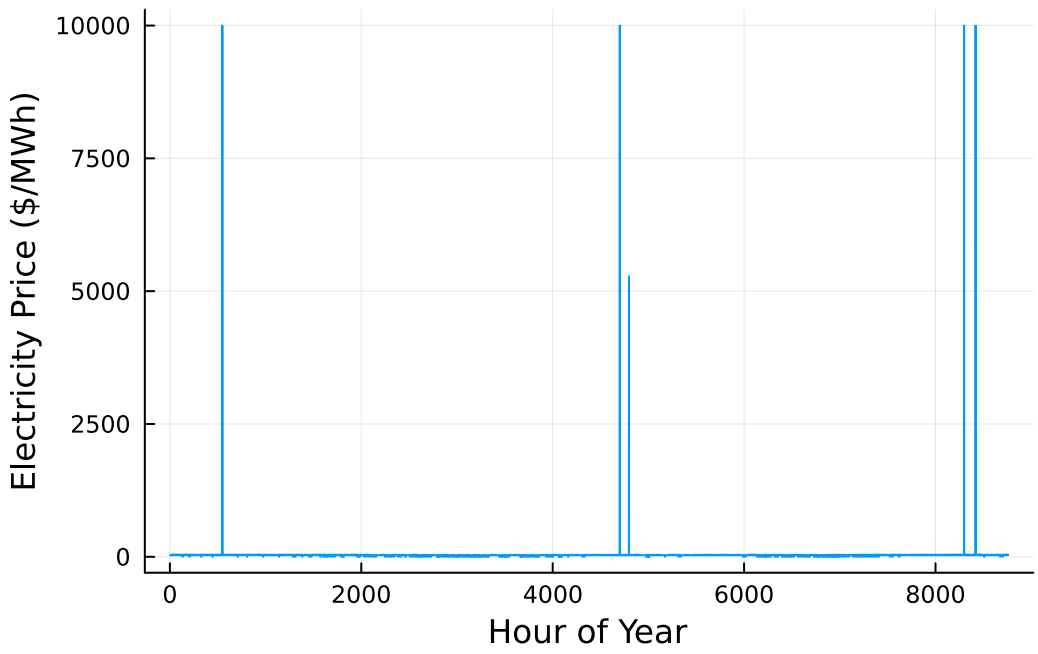

Figure 1: Price of electricity for each hour in the year.

Notice that the prices in Figure 1 go up to $10,000/MWh, as these are the hours in which energy is non-served. Let's restrict the y limits in this plot to see any other trends.

```
ylims!(p, (-5, 50))
```

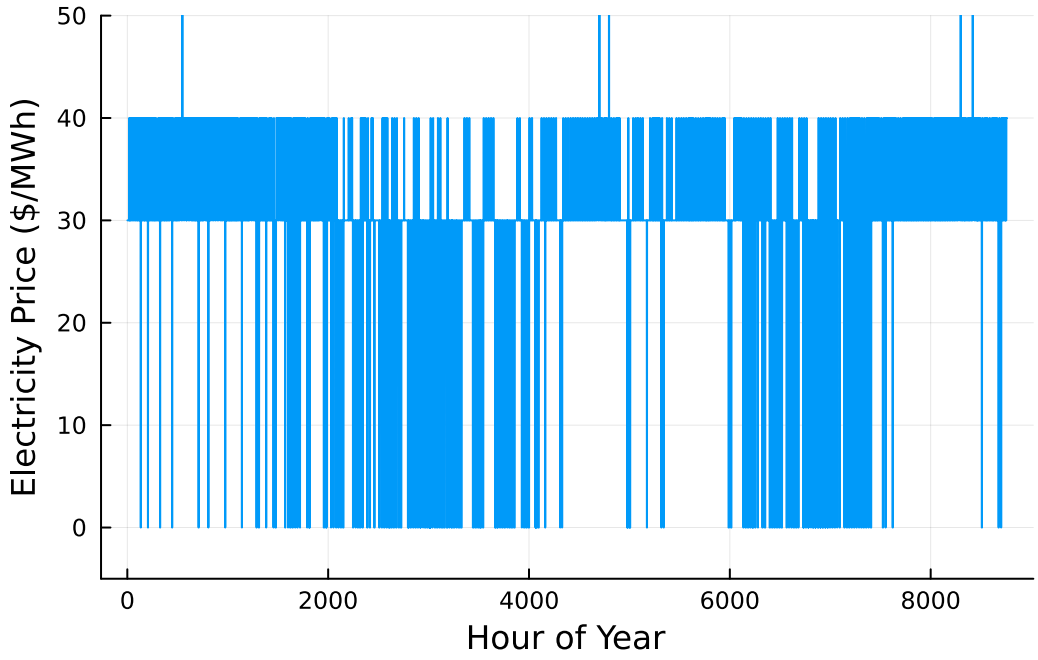

Figure 2: Price of electricity for each hour in the year.

Figure 2 shows that the price can vary between \$0/MWh and between \$30-40/MWh, which depends on whether we can meet demand entirely due to renewable generation or when we need to rely on natural gas.

Problem 3 (10 points)

This problem is only required for students in BEE 5750.

The NY state legislature is considering enacting an annual CO_2 limit, which for the utility would limit the emissions in its footprint to 1.5 MtCO_2/yr.

In this problem:

- What would the value to the utility be of allowing it to emit an additional 1000 tCO_2/yr? An additional 5000?

The only change needed to the LP from Problem 2 is to add in a constraint for the CO_2 limit. Let $emis_g$ be the CO_2 emissions factor (tCO_2/MWh generated) for plant g. Then this constraint is:

$$\sum_{g \in G} emis_g \times \sum_{t \in T} y_{g,t} \leq 1,500,000.$$

Instead of creating a whole new model, we can actually just add a new constraint to the JuMP model (just be careful when evaluating the notebook cells to not jump back to Problem 2 without re-evaluating everything!), but if you re-formulated the model object with a new name, that works as well.

```
# add in the new constraint
@constraint(gencap, co2, sum(gens[:, :Emissions] .* [sum(y[g, :]) for g in
 ↪  G]) <= 1500000);
```

Finding the new solution:

```
optimize!(gencap)
built_cap_co2 = value.(x).data
DataFrame(Plant=gens.Plant, New_Capacity=round.(built_cap_co2; digits=0),
 ↪  Old_Capacity=round.(built_cap; digits=0))
```

```
Coefficient ranges:
  Matrix [5e-05, 1e+00]
  Cost   [2e+01, 4e+05]
  Bound  [0e+00, 0e+00]
  RHS    [1e+03, 2e+06]
Solving LP without presolve, or with basis, or unconstrained
Using EKK dual simplex solver - serial
  Iteration        Objective     Infeasibilities num(sum)
          0     6.5458538900e+08 Pr: 1(617379); Du: 0(2.11155e-06) 0s
       3982     6.7522866671e+08 Pr: 1049(2.50139e+06); Du: 0(4.44609e-06) 5s
       8548     6.8112749404e+08 Pr: 6553(2.63109e+06); Du: 0(4.62437e-06) 10s
      12316     7.0641506462e+08 Pr: 721(4.11196e+06); Du: 0(5.48107e-06) 15s
      15989     7.4200754019e+08 Pr: 9334(2.97015e+07); Du: 0(8.30066e-06) 20s
      20970     7.8473854135e+08 Pr: 2405(35057.8); Du: 0(9.37279e-06) 26s
      22000     7.8535295894e+08 Pr: 0(0); Du: 0(8.52409e-11) 27s
Model   status      : Optimal
Simplex   iterations: 22000
Objective value     :  7.8535295894e+08
HiGHS run time      :         26.87
```

	Plant	New_Capacity	Old_Capacity
	String15	Float64	Float64
1	Geothermal	286.0	-0.0
2	Coal	0.0	0.0
3	NG CCGT	1509.0	1658.0
4	NG CT	703.0	880.0
5	Wind	2549.0	485.0
6	Solar	2104.0	1958.0

To meet the emissions constraint, we build a reduced amount of natural gas, which creates some interesting changes in the rest of the mix. This natural gas capacity is replaced by a combination of 286 MW of geothermal (which was previously zero), an additional 2100 MW of wind capacity, and 150 MW of solar. These massive increases in wind are required to ensure adequate generation when solar is low, since we can no longer rely on as much gas generation for that purpose.

To find the value to the utility of relaxing the CO_2 constraint, we can use the shadow price:

```
@show shadow_price(co2);
```

```
shadow_price(co2) = -182.77193609185798
```

Thus, every tCO_2 we allow will be worth \$183, so allowing an extra 1,000 tCO_2 will be worth \$183000 and allowing an extra 5,000 tCO_2 \$915000.