Day 2: Supply I

We talked today about how electricity markets work.

We will learn today how to build a simple model of an electricity market using **JuMP**.

The data and code are based on the paper "The Efficiency and Sectoral Distributional Implications of Large-Scale Renewable Policies," by Mar Reguant.

We first load relevant libraries.

Compared to day 1, we will be adding the libraries `JuMP` and the solvers `Ipopt` (non-linear solver) and `Cbc` (mixed linear integer solver). We will also be using the clustering library `Clustering`.

Note: I often prefer to use commercial solvers (Gurobi or CPLEX), which are available under an academic license. I use solvers that are readily available here without a license for simplicity and to ensure that everyone can access the code.

```
begin
    using DataFrames
    using CSV
    using JuMP
    using Ipopt   , Cbc
    using Clustering
    using Plots
    using StatsPlots
    using Statistics   , StatsBase
    using Printf
    using Random
end
```

We load the data using the CSV syntax (`CSV.read`) into a data frame called `df`. Here we need to do some cleaning of the variables, rescaling and dropping missing entries.

	year	month	day	hour	price	imports	q_commercial	q_industrial	q_reside
1	2011	1	2	1	29.5397	4.502	8.38001	2.05659	10.6404
2	2011	1	2	2	27.9688	4.363	8.34789	2.06558	9.80354
3	2011	1	2	3	26.5258	4.089	8.54809	2.11851	9.5554
4	2011	1	2	4	25.5872	3.783	8.56002	2.13467	9.31031
5	2011	1	2	5	25.9229	3.969	8.61251	2.17499	9.4285

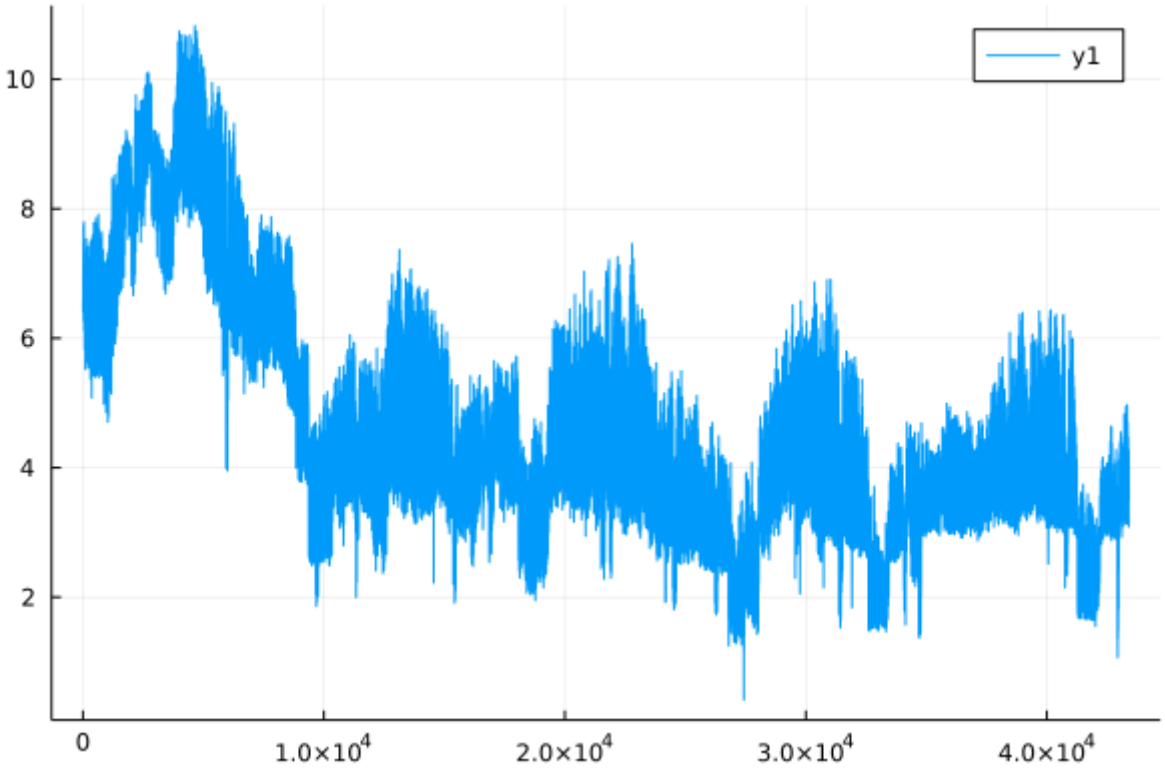
```
begin
    # We read the data and clean it up a bit
    df = CSV.read("data_jaere.csv", DataFrame)
    df = sort(df,["year","month","day","hour"])
    df = dropmissing(df)
    df.nuclear = df.nuclear/1000.0
    df.hydro = df.hydro/1000.0
    df.imports = df.imports/1000.0
    df.q_commercial = df.q_commercial/1000.0
    df.q_industrial = df.q_industrial/1000.0
    df.q_residential = df.q_residential/1000.0
    df.hydronuc = df.nuclear + df.hydro
    df = select(df,Not(["nuclear","hydro"]))
    first(df, 5)
end
```

	year	month	day	hour	price	imports	q_commercial	q_industrial	q_r
1	2011	1	2	1	29.5397	4.502	8.38001	2.05659	10.
2	2011	1	2	2	27.9688	4.363	8.34789	2.06558	9.8
3	2011	1	2	3	26.5258	4.089	8.54809	2.11851	9.5
4	2011	1	2	4	25.5872	3.783	8.56002	2.13467	9.3
5	2011	1	2	5	25.9229	3.969	8.61251	2.17499	9.4
6	2011	1	2	6	27.8414	4.141	8.81962	2.23299	9.7
7	2011	1	2	7	27.8229	4.381	8.78951	2.29248	10.
8	2011	1	2	8	28.093	4.74	8.24397	2.30747	11.
9	2011	1	2	9	30.9623	5.298	8.09124	2.30744	13.
10	2011	1	2	10	33.2964	5.536	8.56532	2.25805	13.
more									
43408	2015	12	31	24	29.2134	6.387	9.28724	2.67757	12.

```
df
```

	variable	mean	min	median	max	nmissing	eltype
1	:year	2013.0	2011	2013.0	2015	0	Int64
2	:month	6.5478	1	7.0	12	0	Int64
3	:day	15.7486	1	16.0	31	0	Int64
4	:hour	12.5064	1	13.0	24	0	Int64
5	:price	35.5433	-13.9395	34.5713	172.352	0	Float64
6	:imports	7.41422	1.571	7.446	11.674	0	Float64
7	:q_commercial	12.1097	6.91611	11.458	26.0133	0	Float64
8	:q_industrial	3.91344	1.95289	3.65441	8.53145	0	Float64
9	:q_residential	10.5988	3.87066	9.8203	24.9831	0	Float64
10	:wind_cap	0.323496	0.00689445	0.301532	0.949153	0	Float64
11	:solar_cap	0.185062	0.0	0.0	0.984559	0	Float64
12	:hydronuc	4.5316	0.415	3.951	10.824	0	Float64

```
describe(df)
```




```
plot(rownumber.(eachrow(df)),df.hydronuc)
```

```
Enter cell code...
```

Clustering our data

When modeling electricity markets, oftentimes the size of the problem can make the solver slow.

Here we will be using a clustering algorithm to come up with a (much) smaller synthetic dataset that we will use for the purposes of our main analysis.

Note: We ignore the time variables when we cluster.

```
8×100 Matrix{Float64}:
 52.6844    34.654       32.3196     34.4721    …  47.21       50.3771    45.8027
  9.32972    8.09594      8.41267     6.68697       6.48427     6.51761    8.1556
 19.4425    11.0544      14.0695     15.0617       11.8391     13.5027    10.3674
  3.83276    3.24581      3.43187     6.14058       4.36533     4.87782    3.19997
 13.9897    13.4298       9.80489     8.93031      10.0406      8.31149   13.7591
  0.31759    0.256715     0.136232    0.259152  …   0.137213    0.295722   0.0935896
  0.650792   0.00651777   0.0571869   0.718544      0.0313887   0.37557    0.00801751
  9.46003    4.0713       6.70831     3.68863       3.38879     2.61535    4.08204
```

```
begin
    n = 100
    X = transpose(Array(select(df,Between(:price,:hydronuc))));

    # We scale variables to improve kmeans performance. For that, we take the mean and std of each row (dim=2) and you repeat it by the number of columns (rows in df)
    Xs = (X.- repeat(mean(X,dims=2),1,nrow(df)))./repeat(std(X,dims=2),1,nrow(df));

    #we set seed because kmeans picks random samples to generate clusters
    Random.seed!(2020)
    R = kmeans(Xs, n);

    # Get the cluster centers rescaling again
    M = R.centers .* repeat(std(X,dims=2),1,n) .+ repeat(mean(X,dims=2),1,n);

    # R = kmeans(X, n);
    # M = R.centers;
end
```

	price	imports	q_commercial	q_industrial	q_residential	wind_cap	solar_cap	
1	52.6844	9.32972	19.4425	3.83276	13.9897	0.31759	0.650792	9
2	34.654	8.09594	11.0544	3.24581	13.4298	0.256715	0.00651777	4
3	32.3196	8.41267	14.0695	3.43187	9.80489	0.136232	0.0571869	6
4	34.4721	6.68697	15.0617	6.14058	8.93031	0.259152	0.718544	3
5	37.2787	6.17066	9.65466	3.29223	9.21121	0.449678	0.0259009	2

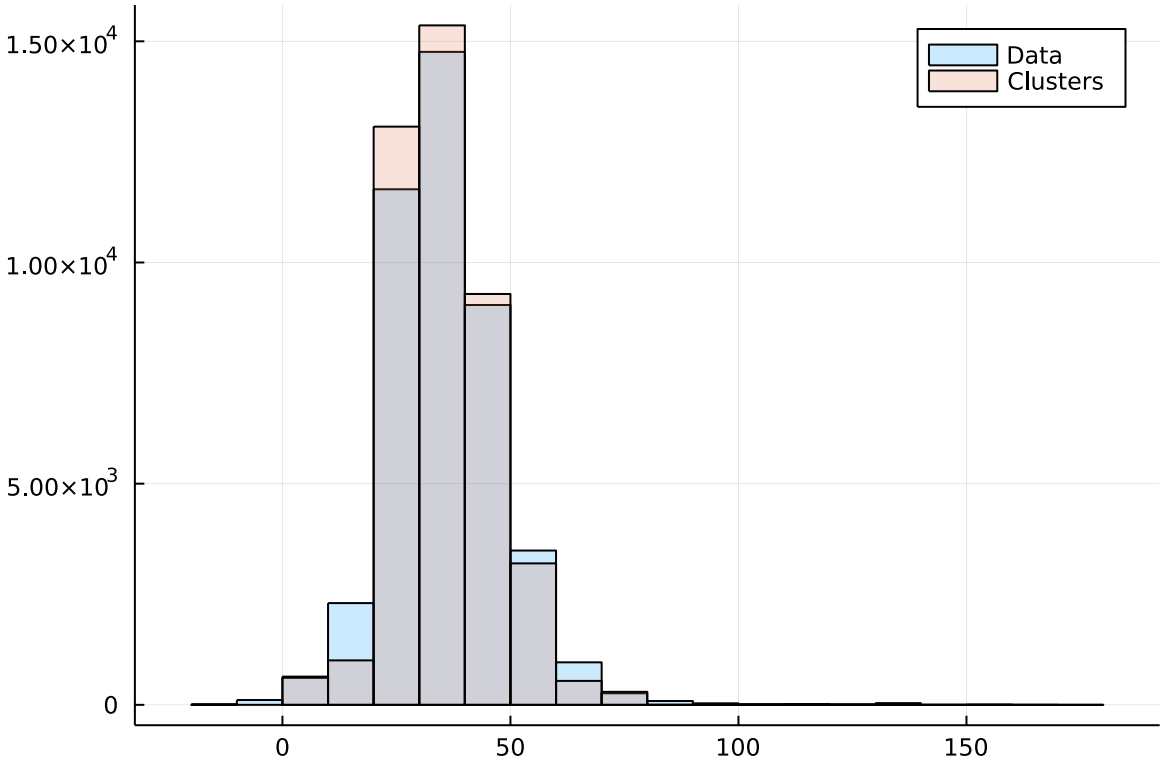
```
begin
    dfclust = DataFrame(transpose(M),
    ["price", "imports", "q_commercial", "q_industrial", "q_residential",
            "wind_cap", "solar_cap", "hydronuc"]);
    dfclust.weights = counts(R);
    first(dfclust, 5)

end
```

	price	imports	q_commercial	q_industrial	q_residential	wind_cap	solar_cap
1	52.6844	9.32972	19.4425	3.83276	13.9897	0.31759	0.650792
2	34.654	8.09594	11.0544	3.24581	13.4298	0.256715	0.00651777
3	32.3196	8.41267	14.0695	3.43187	9.80489	0.136232	0.0571869
4	34.4721	6.68697	15.0617	6.14058	8.93031	0.259152	0.718544
5	37.2787	6.17066	9.65466	3.29223	9.21121	0.449678	0.0259009
6	26.7867	7.03938	13.6814	5.02118	7.77317	0.168569	0.432051
7	32.5801	8.27651	14.7801	5.37046	5.95472	0.48007	0.0181763
8	67.0488	7.45997	11.647	4.13495	12.0611	0.226027	0.00981431
9	40.0577	9.16049	13.6316	3.62875	18.3038	0.489334	0.0277282
10	38.0546	7.59731	11.4813	3.01846	13.1705	0.423228	0.0102609
	more						
100	45.8027	8.1556	10.3674	3.19997	13.7591	0.0935896	0.00801751

```
dfclust
```

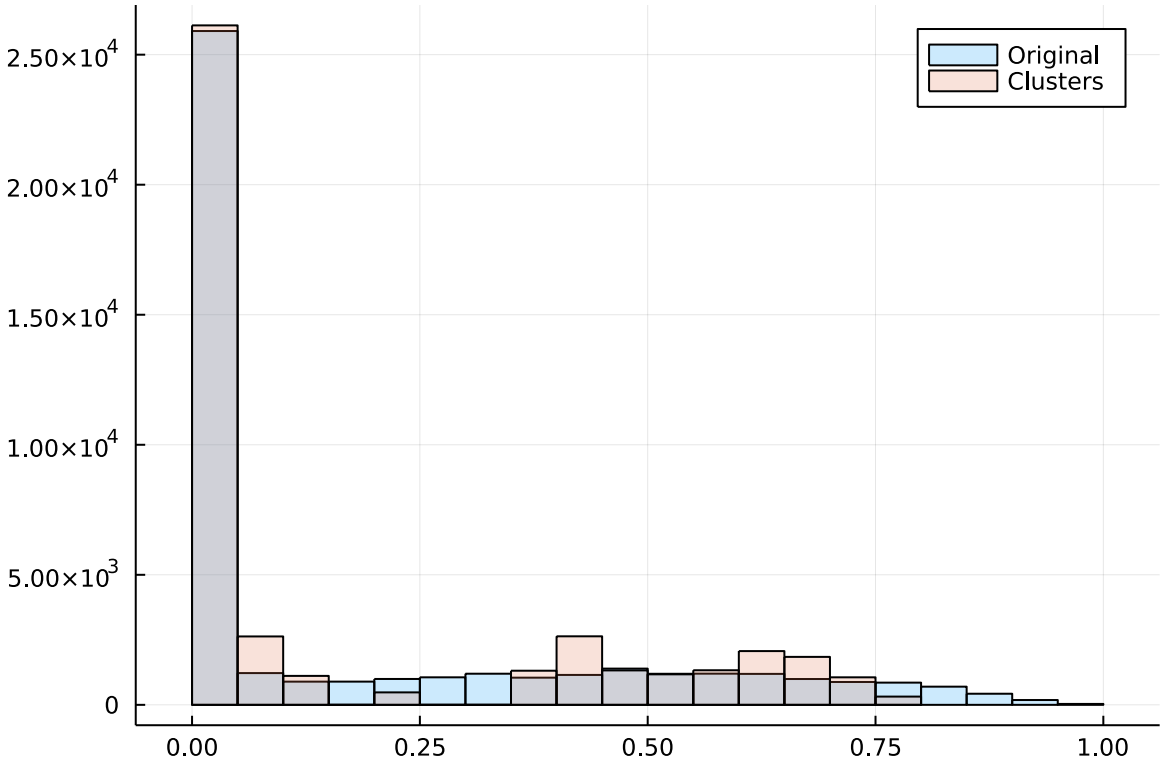
We can compare the distribution of outcomes between the original dataset and the new dataset.

Here is an example with prices. The two distributions are very similar.



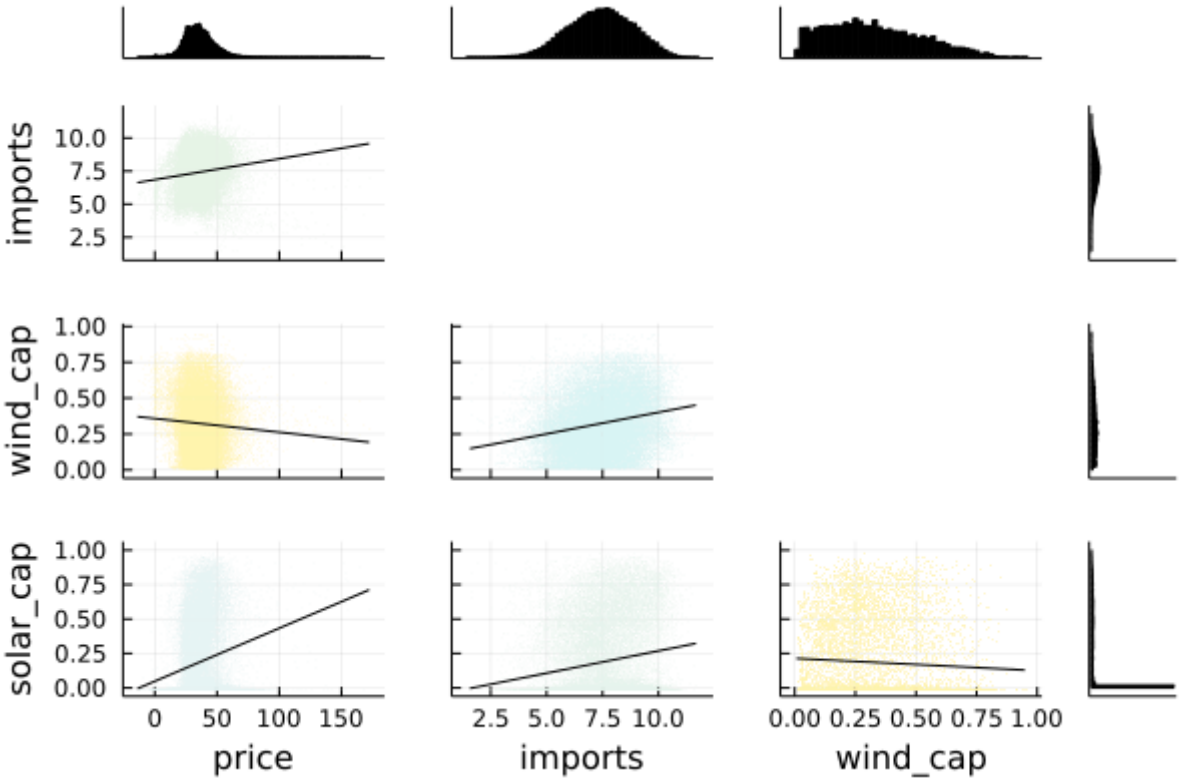

```
begin
    histogram(df.price, fillalpha=.2, nbins=20, label="Data")
    histogram!(dfclust.price,weights=dfclust.weights,
    fillalpha=.2, nbins=20,
        label="Clusters")
end
```

It is also relatively well matched for the case for solar, although it is harder there.

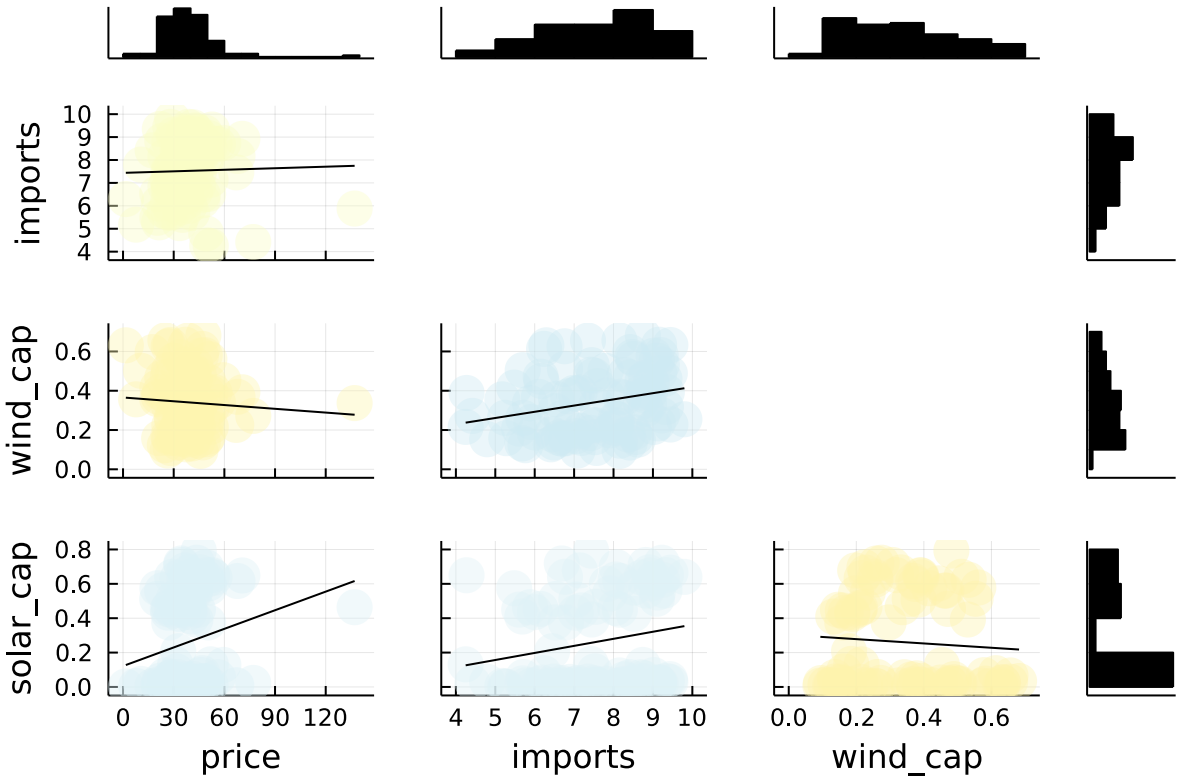



```
begin
    histogram(df.solar_cap, fillalpha=.2, nbins=20, label="Original")
    histogram!(dfclust.solar_cap, weights=dfclust.weights, fillalpha=.2, nbins=20,
    label="Clusters")
end
```

We can also check that the correlation between the main variables of interest remains similar.




```
begin
    # with original data
    @df df cornerplot(cols([:price, :imports,:wind_cap,:solar_cap]), grid = false,
    compact=true)
end
```




```
begin
    # with synthetic data, note issue that weights are not allowed in Julia function
    @df dfclust cornerplot(cols([:price, :imports,:wind_cap,:solar_cap]), grid =
    false,compact = true)
end
```

We can visualize the correlations directly, allowing for a correction for weights.

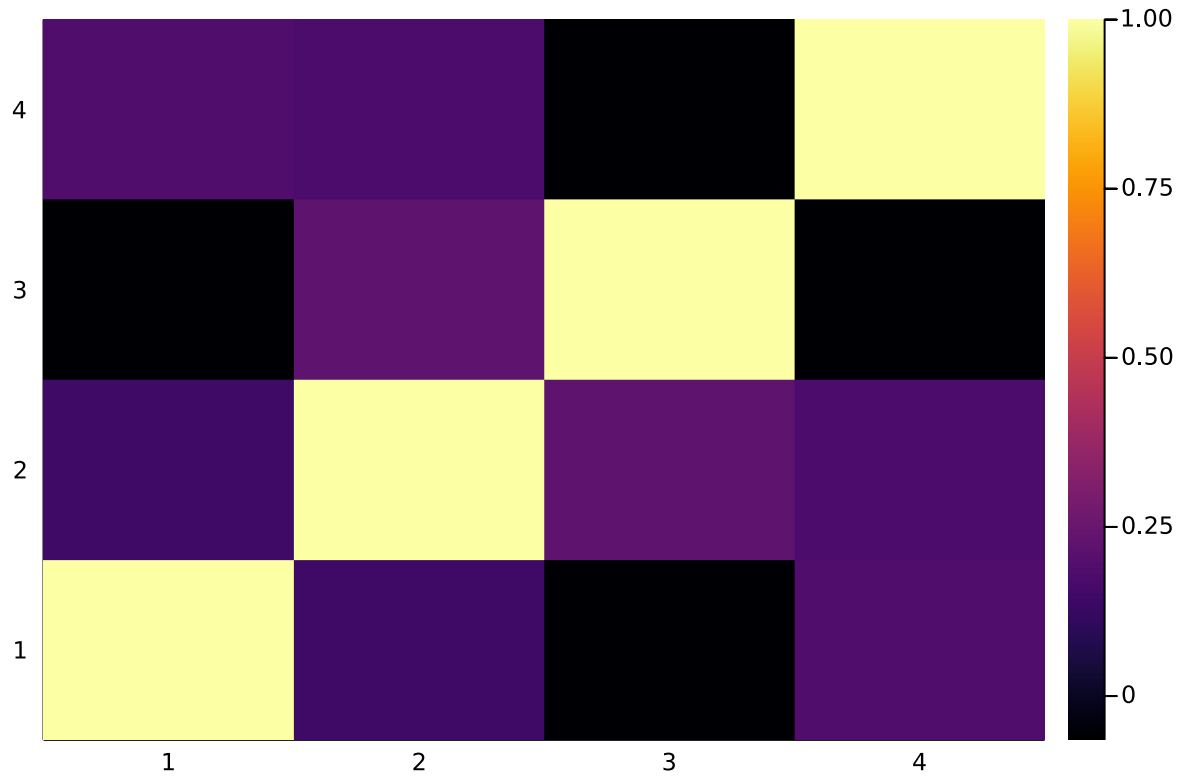
We can see that the overall correlation patterns are quite good, capturing mot of the relationships in the data accurately.

```
MatOriginal = 4×4 Matrix{Float64}:
 1.0        0.141085  -0.0628841   0.183123
 0.141085   1.0        0.221393    0.172096
-0.0628841  0.221393   1.0        -0.0647719
 0.183123   0.172096  -0.0647719   1.0
```

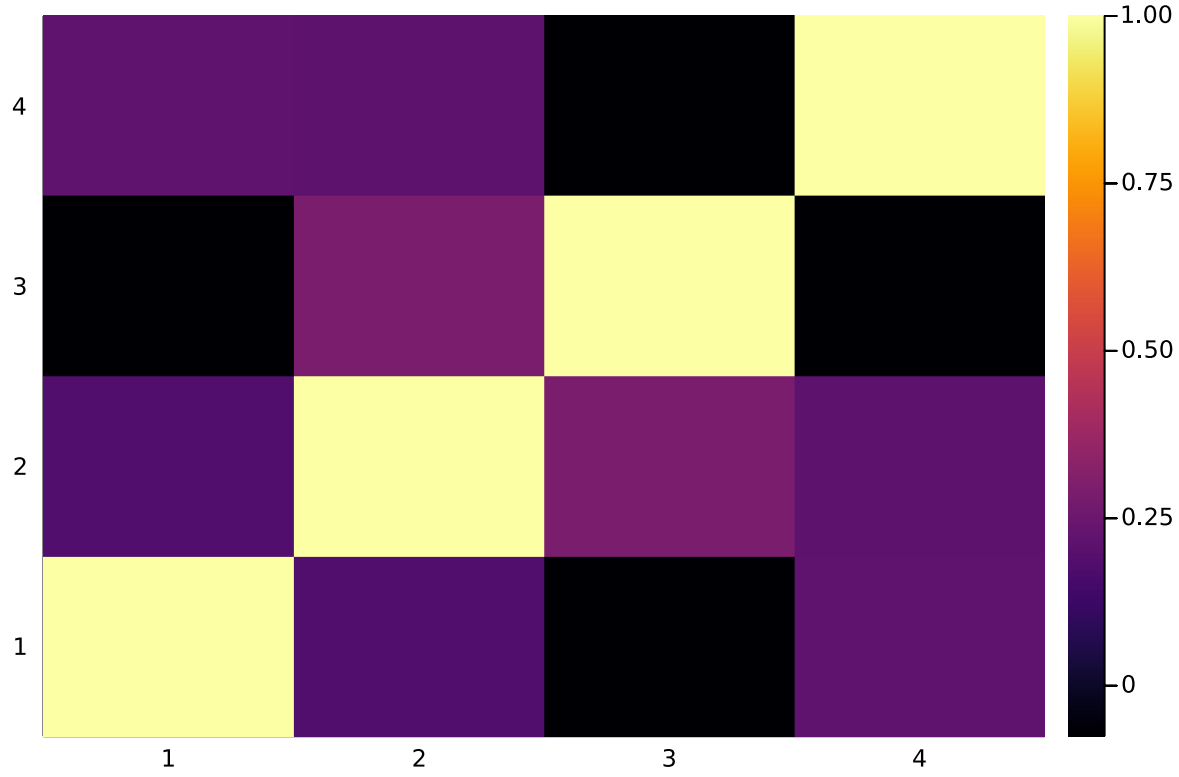
```
MatOriginal = cor(Array(select(df,[:price,:imports,:wind_cap,:solar_cap])))
```

```
MatClust = 4×4 Matrix{Float64}:
 1.0        0.18033   -0.0754344   0.217719
 0.18033    1.0        0.289014    0.210133
-0.0754344  0.289014   1.0        -0.0730127
 0.217719   0.210133  -0.0730127   1.0
```

```
MatClust = cor(Array(select(dfclust,[:price, :imports,:wind_cap,:solar_cap])),
        Weights(dfclust.weights))
```




```
heatmap(MatOriginal)
```




```
heatmap(MatClust)
```

Building the model

Now that we have clustered our data, we will build our model with the data that we have.

The model that we will build today is a simplification from the original paper.

In the original paper, the model needed to solve for:

1. Endogenous retail prices (in a demand model, iterated to find equilibrium)
2. Endogenous investment (in same supply model, with more equations)

Here we will be simply building a simple model of market clearing.

Before building the model, we define some model parameters related to:

- Number and costs of different technologies (loaded from a small dataset)
- Elasticity of demand and imports

tech =

	techname	heatrate	heatrate2	capUB	thermal	e	e2	c
1	"Hydro/Nuclear"	10.0	0.0	1.0	0	0.0	0.0	10.0
2	"Existing 1"	6.67199	0.0929123	11.5	1	0.360184	0.0048861	23.35
3	"Existing 2"	9.79412	0.286247	14.5	1	0.546134	0.0110777	34.27
4	"Existing 3"	13.8181	20.5352	0.578	1	0.816768	0.234476	48.36
5	"Wind"	0.0	0.0	100.0	0	0.0	0.0	0.0
6	"Solar"	0.0	0.0	100.0	0	0.0	0.0	0.0

```
tech = CSV.read("data_technology_simple.csv", DataFrame)
```

To calibrate demand, one can use different strategies. Here we compute the slope for the demand curve that is consistent with the assumed elasticity of demand.

Notice that this is a local elasticity approximation, but it has the advantage of being a linear demand curve, which is very attractive for the purposes of linear programming.

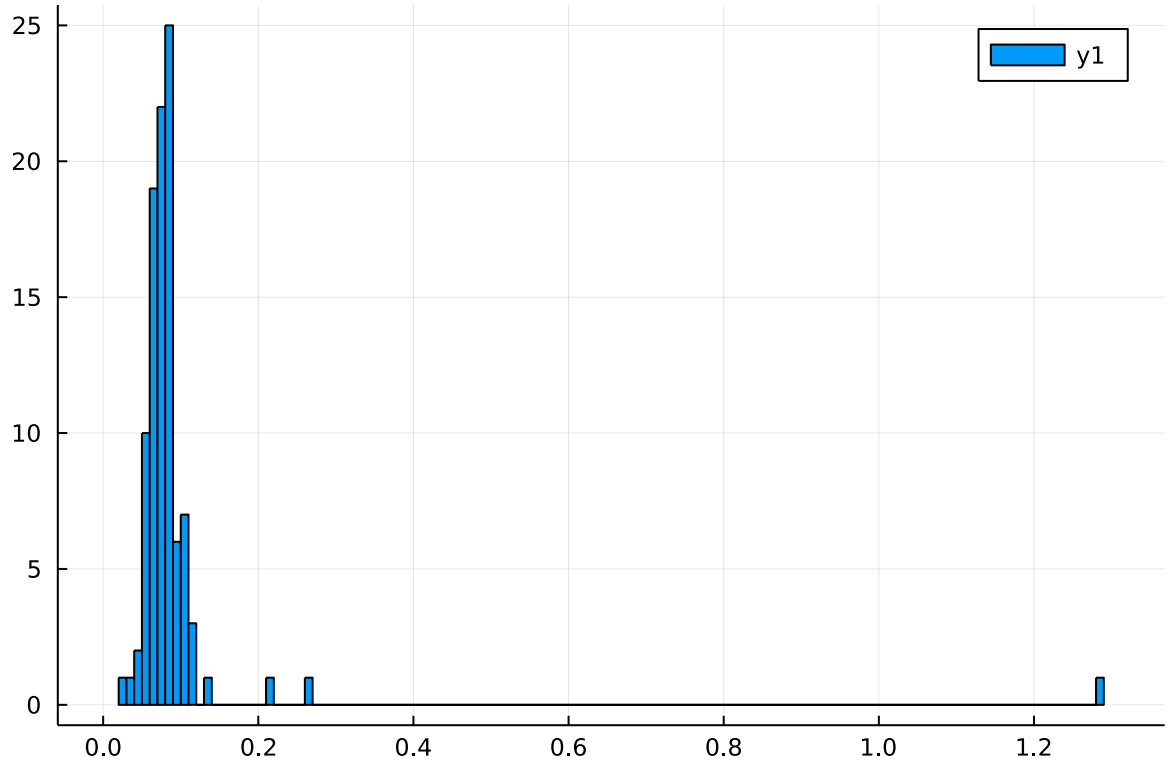
The demand is: $q = a - b\,p$

So the elasticity becomes: $b\frac{p}{q}$, which we set equal to an assumed parameter.

Once we have b, we can back out a. An analogous procedure is done for imports.

```
[6.5308, 5.66716, 5.88887, 4.68088, 4.31946, 4.92756, 5.79356, 5.22198, 6.41234, 5.31812
```

```
begin
    # Re-scaling
    dfclust.weights = dfclust.weights / sum(dfclust.weights);

    # Here only one demand type to make it easier
    dfclust.demand = dfclust.q_residential + dfclust.q_commercial +
    dfclust.q_industrial;

    # Calibrate demand based on elasticities (using 0.1 here as only one final
demand)
    elas = [.1, .2, .5, .3];
    dfclust.b = elas[1] * dfclust.demand ./ dfclust.price;  # slope
    dfclust.a = dfclust.demand + dfclust.b .* dfclust.price;  # intercept

    # Calibrate imports (using elas 0.3)
    dfclust.bm = elas[4] * dfclust.imports ./ dfclust.price;  # slope
    dfclust.am = dfclust.imports - dfclust.bm .* dfclust.price;  # intercept
end
```




```
histogram(dfclust.b)
```

Non-linear solver

We are now ready to clear the market. We will **maximize welfare** using a non-linear solver.

$$\max\ CS - Costs$$

$$\text{s.t. operational constraints, market clearing.}$$

We will then consider an approach **based on FOC**, which is useful to extend to strategic firms as in Bushnell, Mansur, and Saravia (2008) and Ito and Reguant (2016).

In perfect competition, the two approaches should be equivalent–and they are in my computer!

```
clear_market_min (generic function with 1 method)
```

```
## Clear market based on cost minimization
function clear_market_min(data::DataFrame, tech::DataFrame;
        wind_gw = 5.0, solar_gw = 2.0)

    # We declare a model
    model = Model(
        optimizer_with_attributes(
            Ipopt.Optimizer)
        );

    # Set useful indexes
    I = nrow(tech);  # number of techs
    T = nrow(data);  # number of periods
    S = 1;  # we will only be using one sector to keep things simple

    # Variables to solve for
    @variable(model, price[1:T]);
    @variable(model, demand[1:T]);
    @variable(model, imports[1:T]);
    @variable(model, quantity[1:T, 1:I] >= 0);

    # Maximize welfare including imports costs
    @NLobjective(model, Max, sum(data.weights[t] * (
                (data.a[t] - demand[t]) * demand[t] / data.b[t]
            + demand[t]^2/(2*data.b[t])
        - sum(tech.c[i] * quantity[t,i]
                    + tech.c2[i] * quantity[t,i]^2/2 for i=1:I)
        - (imports[t] - data.am[t])^2/(2 * data.bm[t])) for t=1:T));

    # Market clearing
    @constraint(model, [t=1:T],
        demand[t] == data.a[t] - data.b[t] * price[t]);
    @constraint(model, [t=1:T],
        imports[t] == data.am[t] + data.bm[t] * price[t]);
    @constraint(model, [t=1:T],
        demand[t] == sum(quantity[t,i] for i=1:I) + imports[t]);

    # Constraints on output
    @constraint(model, [t=1:T],
        quantity[t,1] <= data.hydronuc[t]);
    @constraint(model, [t=1:T,i=2:4],
        quantity[t,i] <= tech[i,"capUB"]);
    @constraint(model, [t=1:T],
        quantity[t,5] <= wind_gw * data.wind_cap[t]);
    @constraint(model, [t=1:T],
        quantity[t,6] <= solar_gw * data.solar_cap[t]);

    # Solve model
    optimize!(model);

    status = @sprintf("%s", JuMP.termination_status(model));

    if (status=="LOCALLY_SOLVED")
        p = JuMP.value.(price);
        avg_price = sum(p[t] * data.weights[t] for t=1:T);
        q = JuMP.value.(quantity);
        imp = JuMP.value.(imports);
```

```
        d = JuMP.value.(demand);
        cost = sum(data.weights[t] * (sum(tech.c[i] * q[t,i]
                + tech.c2[i] * q[t,i]^2 / 2 for i=1:I)
                + (imp[t] - data.am[t])^2/(2 * data.bm[t])) for t=1:T);
        results = Dict("status" => @sprintf("%s",JuMP.termination_status(model)),
            "avg_price" => avg_price,
            "price" => p,
            "quantity" => q,
            "imports" => imp,
            "demand" => d,
            "cost" => cost);
        return results
    else
        results = Dict("status" => @sprintf("%s",JuMP.termination_status(model)));
        return results
    end

end
```

```
results_min =
  Dict("avg_price" ⇒ 33.5402, "cost" ⇒ 427.72, "price" ⇒ [39.9526, 36.7374, 31.6412, 39
```

```
results_min = clear_market_min(dfclust, tech)
```

```
******************************************************************************
This program contains Ipopt, a library for large-scale nonlinear optimizatio
n.
 Ipopt is released as open source code under the Eclipse Public License (EP
L).
         For more information visit https://github.com/coin-or/Ipopt
******************************************************************************

This is Ipopt version 3.13.4, running with linear solver mumps.
NOTE: Other linear solvers might be more efficient (see Ipopt documentatio
n).

Number of nonzeros in equality constraint Jacobian...:     1200
Number of nonzeros in inequality constraint Jacobian.:      600
Number of nonzeros in Lagrangian Hessian.............:      800

Total number of variables............................:      900
                     variables with only lower bounds:      600
                variables with lower and upper bounds:        0
                     variables with only upper bounds:        0
Total number of equality constraints.................:      300
Total number of inequality constraints...............:      600
        inequality constraints with only lower bounds:        0
   inequality constraints with lower and upper bounds:        0
        inequality constraints with only upper bounds:      600

iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr
ls
   0 -2.1845710e+02 4.60e+01 3.97e-01  -1.0 0.00e+00    -  0.00e+00 0.00e+00
0
   1  2.1436860e+03 2.92e+01 1.86e+00  -1.0 9.08e+02    -  2.18e-01 3.66e-01
f  1
   2  3.1054063e+03 2.03e+01 5.01e+00  -1.0 5.56e+02    -  3.89e-02 3.06e-01
f  1
   3  3.9286886e+03 1.19e+01 8.69e+00  -1.0 3.65e+02    -  1.03e-01 4.11e-01
f  1
   4  4.4529596e+03 7.27e+00 1.09e+01  -1.0 1.71e+02    -  1.59e-01 3.91e-01
f  1
   5  4.8561697e+03 4.23e+00 1.24e+01  -1.0 1.73e+02    -  2.22e-01 4.19e-01
f  1
   6  5.1926732e+03 1.64e+00 1.65e+01  -1.0 2.26e+02    -  2.38e-01 6.11e-01
f  1
   7  5.3374677e+03 3.96e-01 1.43e+01  -1.0 2.84e+02    -  4.60e-01 7.59e-01
f  1
   8  5.3896784e+03 7.11e-15 7.97e+00  -1.0 1.32e+02    -  6.59e-01 1.00e+00
```

```
r  1
   9  5.4046026e+03 7.99e-15 5.14e-03  -1.0 5.33e+01    -  9.99e-01 1.00e+00
f  1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr
ls
  10  5.4299011e+03 8.44e-15 4.16e+00  -1.7 4.98e+01    -  9.19e-01 1.00e+00
f  1
  11  5.4362263e+03 7.11e-15 2.00e-07  -1.7 2.11e+01    -  1.00e+00 1.00e+00
f  1
  12  5.4431361e+03 7.11e-15 9.18e-01  -3.8 2.32e+01    -  8.22e-01 8.94e-01
f  1
  13  5.4445852e+03 7.11e-15 4.00e-01  -3.8 1.11e+01    -  7.41e-01 8.63e-01
f  1
  14  5.4449286e+03 7.11e-15 6.76e-02  -3.8 4.05e+00    -  8.70e-01 1.00e+00
f  1
  15  5.4449486e+03 7.11e-15 1.50e-09  -3.8 5.81e-01    -  1.00e+00 1.00e+00
f  1
  16  5.4450227e+03 8.66e-15 1.08e-02  -5.7 4.72e-01    -  8.02e-01 9.73e-01
f  1
  17  5.4450255e+03 7.99e-15 1.84e-11  -5.7 1.46e-01    -  1.00e+00 1.00e+00
f  1
  18  5.4450256e+03 7.22e-15 1.84e-11  -5.7 8.23e-02    -  1.00e+00 1.00e+00
f  1
  19  5.4450266e+03 7.11e-15 2.96e-06  -8.6 4.66e-02    -  9.95e-01 1.00e+00
f  1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr
ls
  20  5.4450266e+03 7.11e-15 2.53e-14  -8.6 2.07e-02    -  1.00e+00 1.00e+00
f  1
  21  5.4450266e+03 9.33e-15 2.53e-14  -8.6 6.02e-03    -  1.00e+00 1.00e+00
h  1
  22  5.4450266e+03 7.22e-15 6.66e-16 -12.9 5.94e-04    -  1.00e+00 1.00e+00
f  1

Number of Iterations....: 22

                                   (scaled)                 (unscaled)
Objective...............:  -5.4450265638658057e+03    5.4450265638658057e+03
Dual infeasibility......:   6.6613381477509392e-16    6.6613381477509392e-16
Constraint violation....:   7.2164496600635175e-15    7.2164496600635175e-15
Complementarity.........:   2.2410212464133491e-10   -2.2410212464133491e-10
Overall NLP error.......:   2.2410212464133491e-10    7.2164496600635175e-15


Number of objective function evaluations             = 23
Number of objective gradient evaluations             = 23
Number of equality constraint evaluations            = 23
Number of inequality constraint evaluations          = 23
Number of equality constraint Jacobian evaluations   = 1
Number of inequality constraint Jacobian evaluations = 1
Number of Lagrangian Hessian evaluations             = 22
Total CPU secs in IPOPT (w/o function evaluations)   =      2.112
Total CPU secs in NLP function evaluations           =      1.224

EXIT: Optimal Solution Found.
```

33.54016556018911

```
results_min["avg_price"]
```

427.7201337962463

```
results_min["cost"]
```

Mixed integer solver

The key to the FOC representation is to model the marginal cost of power plants. The algorithm will be using power plants until $MC = Price$.

Note: In the market power version of this algorithm, it sets $MR = MC$.

We will be using **integer variables** to take into consideration that FOC are not necessarily at an interior solution in the presence of capacity constraints.

If $Price < MC(0)$, a technology will not produce.

If $Price > MC(K)$, a technology is at capacity and can no longer increase output. In such case, the firm is earning a markup even under perfect competition. We define the shadow value as:

$$\psi = Price - MC$$

Shadow values define the rents that firms make. These are directly used in an expaded version of the model with investment.

We will define these conditions using binary variables (0 or 1):

- u_1 will turn on when we use a technology.
- u_2 will turn on when we use a technology at capacity.
- ψ can only be positive if $u_2 = 1$.

Compared to the previous approach:

- There will not be an objective function.
- We will use a solver for mixed integer programming (`Cbc`).

```
clear_market_foc (generic function with 1 method)
```

```
## Clear market based on first-order conditions
function clear_market_foc(data::DataFrame, tech::DataFrame;
        wind_gw = 5.0, solar_gw = 2.0,theta=1)

    # We declare a model
    model = Model(
        optimizer_with_attributes(
            Cbc.Optimizer)
        );

    # Set useful indexes
    I = nrow(tech);  # number of techs
    T = nrow(data);  # number of periods
    S = 1;  # we will only be using one sector to keep things simple

    # Variables to solve for
    @variable(model, price[1:T]);
    @variable(model, demand[1:T]);
    @variable(model, imports[1:T]);
    @variable(model, quantity[1:T, 1:I] >= 0);
    @variable(model, shadow[1:T, 1:I] >= 0);  # price wedge if at capacity
    @variable(model, u1[1:T, 1:I], Bin);  # if tech used
    @variable(model, u2[1:T, 1:I], Bin);  # if tech at max

    @objective(model, Min, sum(price[t] * data.weights[t] for t=1:T));

    # Market clearing
    @constraint(model, [t=1:T],
        demand[t] == data.a[t] - data.b[t] * price[t]);
    @constraint(model, [t=1:T],
        imports[t] == data.am[t] + data.bm[t] * price[t]);
    @constraint(model, [t=1:T],
        demand[t] == sum(quantity[t,i] for i=1:I) + imports[t]);

    # Capacity constraints
    @constraint(model, [t=1:T],
        quantity[t,1] <= u1[t,1] * data.hydronuc[t]); #we can only use the
        technology if u1 = 1
    @constraint(model, [t=1:T,i=2:4],
        quantity[t,i] <= u1[t,i] * tech[i,"capUB"]);
    @constraint(model, [t=1:T],
        quantity[t,5] <= u1[t,5] * wind_gw * data.wind_cap[t]);
    @constraint(model, [t=1:T],
        quantity[t,6] <= u1[t,6] * solar_gw * data.solar_cap[t]);

    @constraint(model, [t=1:T],
        quantity[t,1] >= u2[t,1] * data.hydronuc[t]); #if u2 = u1 = 1, hydronuc <=
        q <= hydronuc
    @constraint(model, [t=1:T,i=2:4],
        quantity[t,i] >= u2[t,i] * tech[i,"capUB"]);
    @constraint(model, [t=1:T],
        quantity[t,5] >= u2[t,5] * wind_gw * data.wind_cap[t]);
    @constraint(model, [t=1:T],
        quantity[t,6] >= u2[t,6] * solar_gw * data.solar_cap[t]);

    @constraint(model, [t=1:T,i=1:I], u1[t,i] >= u2[t,i]);

```

```
    # Constraints on optimality
    M = 1e3;
    @constraint(model, [t=1:T,i=1:I],
        price[t] -  theta/(data.b[t]+data.bm[t])*quantity[t,i] - tech.c[i] -
        tech.c2[i]*quantity[t,i] - shadow[t,i]
        >= -M * (1-u1[t,i]));
    @constraint(model, [t=1:T,i=1:I],
        price[t] - theta/(data.b[t]+data.bm[t])*quantity[t,i] - tech.c[i] -
        tech.c2[i]*quantity[t,i] - shadow[t,i]
        <= 0.0);
    @constraint(model, [t=1:T,i=1:I], shadow[t,i] <= M*u2[t,i]);


    # Solve model
    optimize!(model);

    status = @sprintf("%s", JuMP.termination_status(model));

    if (status=="OPTIMAL")
        p = JuMP.value.(price);
        avg_price = sum(p[t] * data.weights[t] for t=1:T);
        q = JuMP.value.(quantity);
        imp = JuMP.value.(imports);
        d = JuMP.value.(demand);
        cost = sum(data.weights[t] * (sum(tech.c[i] * q[t,i]
                + tech.c2[i] * q[t,i]^2 / 2 for i=1:I)
                + (imp[t] - data.am[t])^2/(2 * data.bm[t])) for t=1:T);
        shadow = JuMP.value.(shadow);
        u1 = JuMP.value.(u1);
        u2 = JuMP.value.(u2);
        results = Dict("status" => @sprintf("%s",JuMP.termination_status(model)),
            "avg_price" => avg_price,
            "price" => p,
            "quantity" => q,
            "imports" => imp,
            "demand" => d,
            "cost" => cost,
            "shadow" => shadow,
            "u1" => u1,
            "u2" => u2);
        return results
    else
        results = Dict("status" => @sprintf("%s",JuMP.termination_status(model)));
        return results
    end

end
```

```
Enter cell code...
```

```
results_foc = clear_market_foc(dfclust, tech,theta=0);
```

```
33.54016573880132
```

```
results_foc["avg_price"]
```

```
427.72013566067966
```

```
results_foc["cost"]
```

We can check that u1, u2 and the shadow values are correct. For example, at hour 1, tech 4 is not producing, while tech 3 is setting the price and therefore it does not have inframarginal rents (shadow = 0)

	u1	u2	shadow
1	1.0	1.0	29.9526
2	1.0	1.0	12.8609
3	1.0	0.0	0.0
4	0.0	0.0	0.0
5	1.0	1.0	39.9526
6	1.0	1.0	39.9526

```
begin
    df_results = DataFrame(u1=results_foc["u1"][1,:],u2=results_foc["u2"]
    [1,:],shadow=results_foc["shadow"][1,:])
end
```

Discussion of pros and cons:

- Mixed integer programming has advantages due to its robust finding of global solutions.
- Here, we are using first-order conditions, so a question arises regarding the validity of such conditions to fully characterize a unique solution in more general settings.
- Non-linear solvers explore the objective function but do not tend to be global in nature.
- Non-linear solvers cannot deal with an oligopolistic setting in a single model, as several agents are maximizing profits. We would need to iterate.

Follow-up exercises

1. Imagine each technology is a firm, which now might exercise market power. Can you modify clear_market_foc to account for market power as in BMS (2008)?

2(*). The function is prepared to take several amounts of solar and wind. What are the impacts on prices as you increase solar and wind? Save prices for different values of wind or solar investment and plot them. Does your answer depend a lot on the number of clusters?

3(*). [Harder] Making some assumptions on the fixed costs of solar and wind, can you expand the model to solver for investment? This will require a FOC for the zero profit entry condition. In Bushnell (2011) and Reguant (2019), that FOC might not be satisfied (zero investment), so it is also a complementarity problem.