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
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_resident
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_res
1	2011	1	2	1	29.5397	4.502	8.38001	2.05659	10.64
2	2011	1	2	2	27.9688	4.363	8.34789	2.06558	9.803
3	2011	1	2	3	26.5258	4.089	8.54809	2.11851	9.555
4	2011	1	2	4	25.5872	3.783	8.56002	2.13467	9.310
5	2011	1	2	5	25.9229	3.969	8.61251	2.17499	9.428
6	2011	1	2	6	27.8414	4.141	8.81962	2.23299	9.739
7	2011	1	2	7	27.8229	4.381	8.78951	2.29248	10.40
8	2011	1	2	8	28.093	4.74	8.24397	2.30747	11.65
9	2011	1	2	9	30.9623	5.298	8.09124	2.30744	13.03
10	2011	1	2	10	33.2964	5.536	8.56532	2.25805	13.52
more									
43408	2015	12	31	24	29.2134	6.387	9.28724	2.67757	12.03

```
df
```

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}:
 28.4991    35.1206    54.0689    69.1331      …  58.0201     48.5971    55.8251
  6.82532    7.16396    9.10176    5.89943         8.47609     5.1078     9.30882
 10.5077    14.8845    15.5234    11.4068         12.5062     12.9664    19.4634
  3.05102    3.48671    3.77813    3.88588         4.20635     5.61552    3.82047
 10.2875     8.90182   20.3081    13.0717         13.0578     17.2486    14.7831
  0.146433   0.249785   0.451964   0.194571    …   0.171732    0.556942   0.339089
  0.382221   0.577303   0.54136    0.00506296      0.0160437   0.306562   0.633465
  3.38689    8.90913    6.28418    3.44948         4.02199     5.39365    9.56588
```

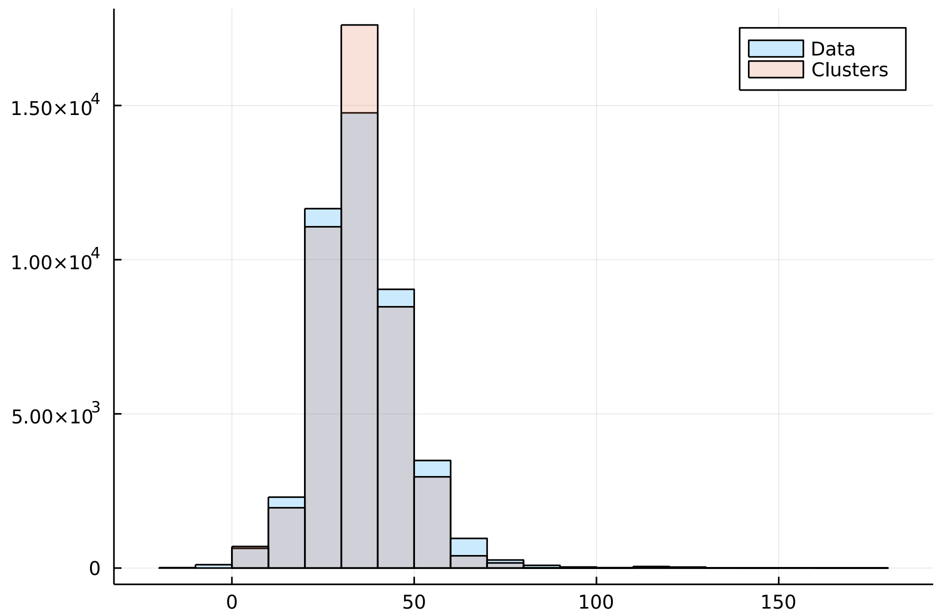
```
begin
    n = 100
    X = transpose(Array(select(df,Between(:price,:hydronuc))));

    # We scale variables to improve kmeans performance
    Xs = (X.- repeat(mean(X,dims=2),1,nrow(df)))./repeat(std(X,dims=2),1,nrow(df));
    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	hy
1	28.4991	6.82532	10.5077	3.05102	10.2875	0.146433	0.382221	3.
2	35.1206	7.16396	14.8845	3.48671	8.90182	0.249785	0.577303	8.
3	54.0689	9.10176	15.5234	3.77813	20.3081	0.451964	0.54136	6.
4	69.1331	5.89943	11.4068	3.88588	13.0717	0.194571	0.00506296	3.
5	30.0975	7.37008	10.9111	2.69482	11.0674	0.168375	0.451678	6.

```
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
```

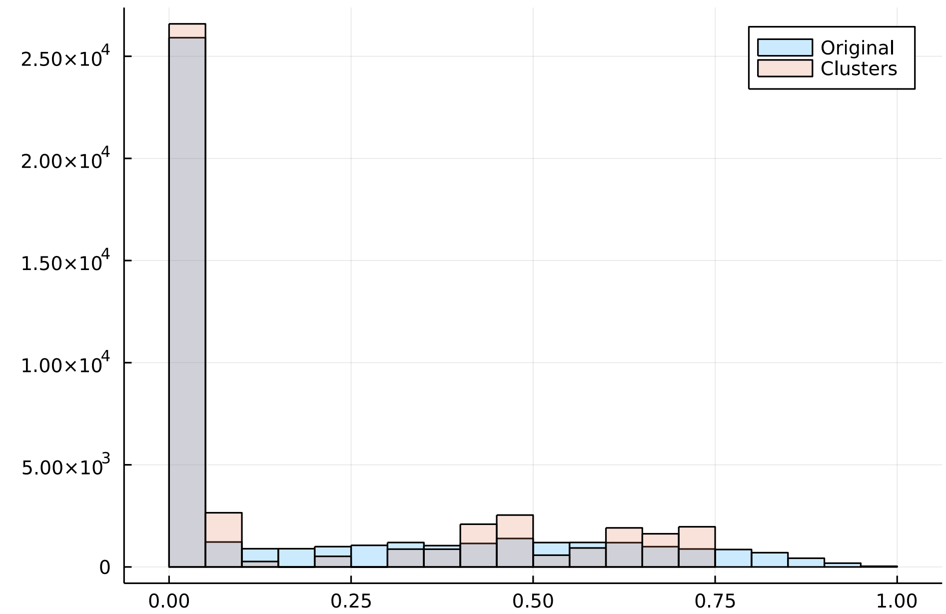
We can compare the distribution of outcomes between the original dataset and the new dataset.

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



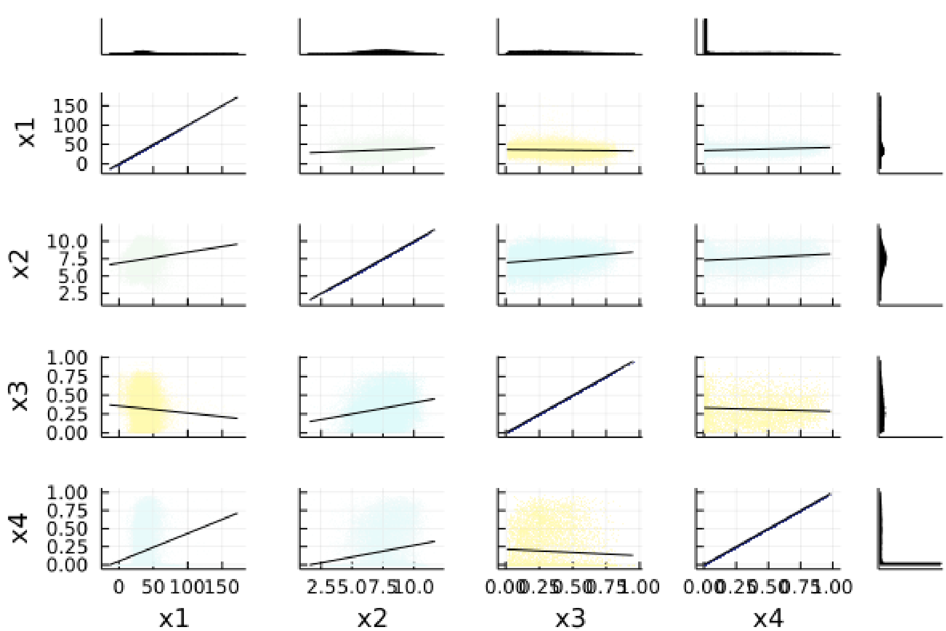

```
begin
    histogram(df.price, fillalpha=.2, nbins=20, label="Data")
    histogram!(dfclust.price,weights=dfclust.weights, fillalpha=.2, nbins=20,
        label="Clusters")
end
```

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

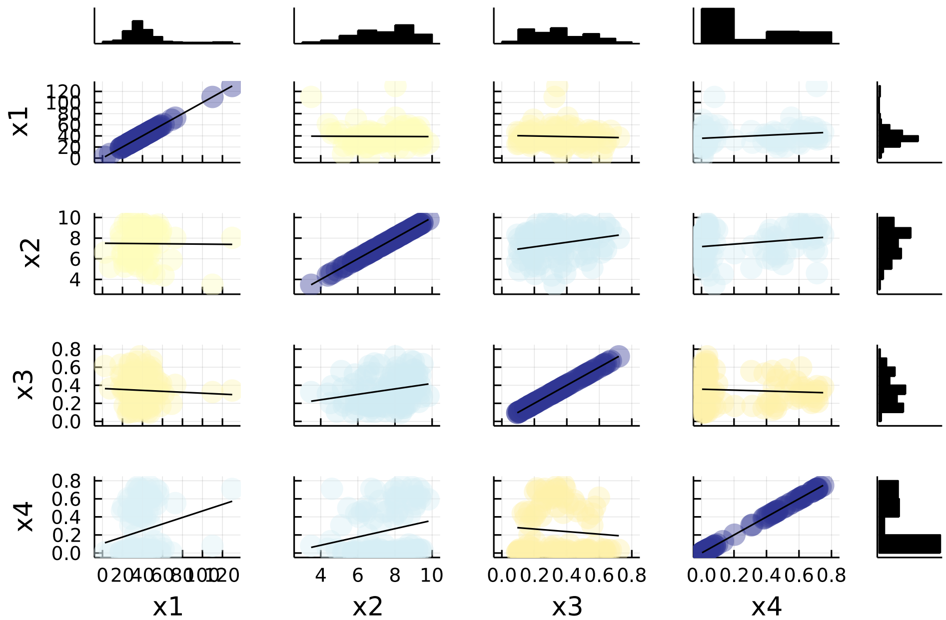



```
begin
    histogram(df.solar_cap, fillalpha=.2, nbins=20, label="Original")
    histogram!(dfclust.solar_cap, weights=dfclust.weights, fillalpha=.2, nbins=20,
label="Clusters")
end
```

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




```
begin
    # with original data
    cornerplot(Array(select(df,[:price, :imports,:wind_cap,:solar_cap])))
end
```




```
begin
    # with synthetic data, note issue that weights are not allowed in Julia function
    cornerplot(Array(select(dfclust,[:price,:imports,:wind_cap,:solar_cap])))
end
```

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

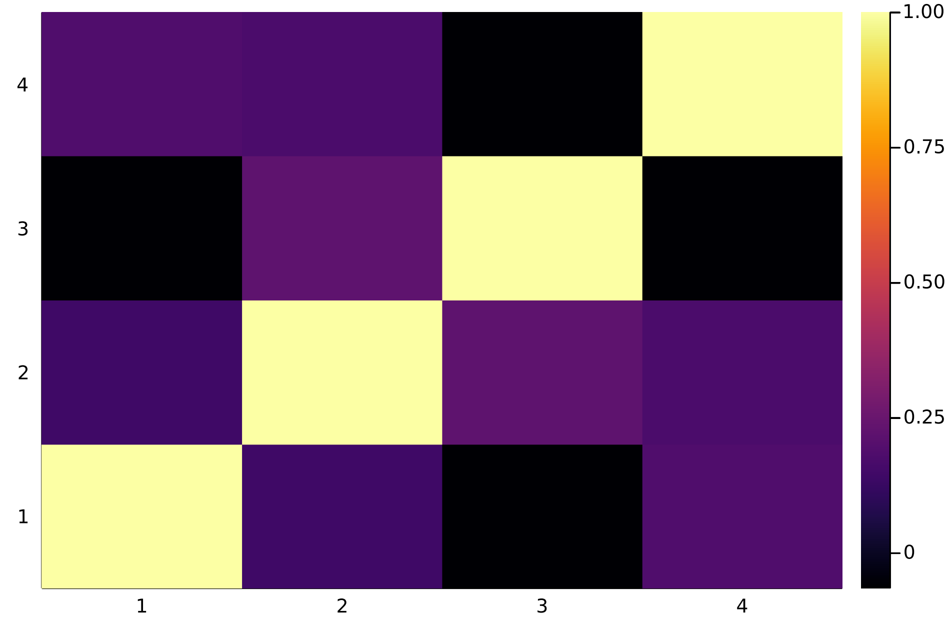
We can see that the overall correlation patterns are quite good, capturing mot of the relationships in the data accurately.

```
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
```

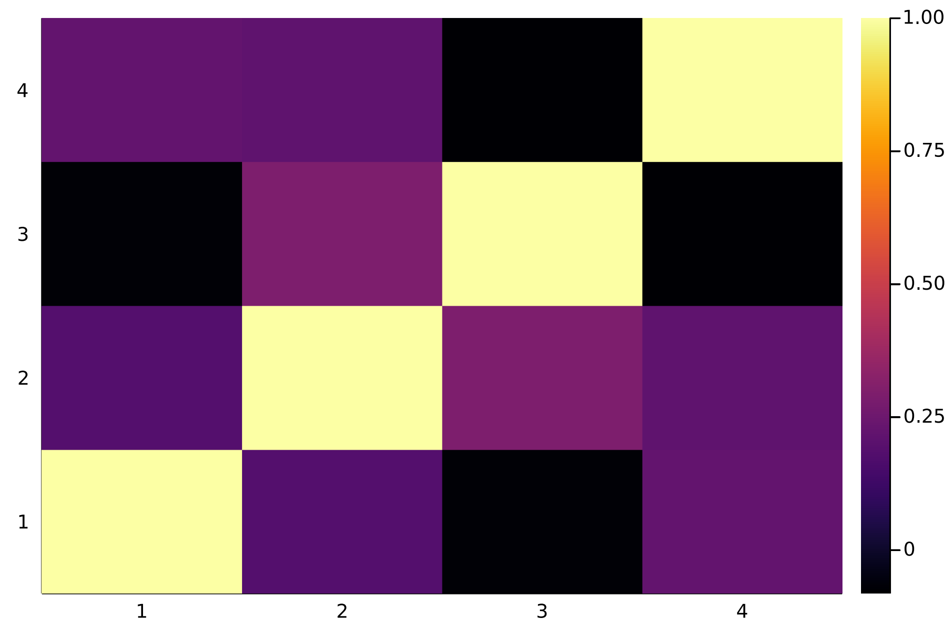
```
MatOriginal = cor(Array(select(df,[:price,:imports,:wind_cap,:solar_cap])))
```

```
MatClust = 4×4 Matrix{Float64}:
              1.0        0.181258  -0.074098    0.222193
              0.181258   1.0        0.291775    0.21099
             -0.074098   0.291775   1.0        -0.0813579
              0.222193   0.21099   -0.0813579   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	23.352
3	"Existing 2"	9.79412	0.286247	14.5	1	0.546134	0.0110777	34.2794
4	"Existing 3"	13.8181	20.5352	0.578	1	0.816768	0.234476	48.3634
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.

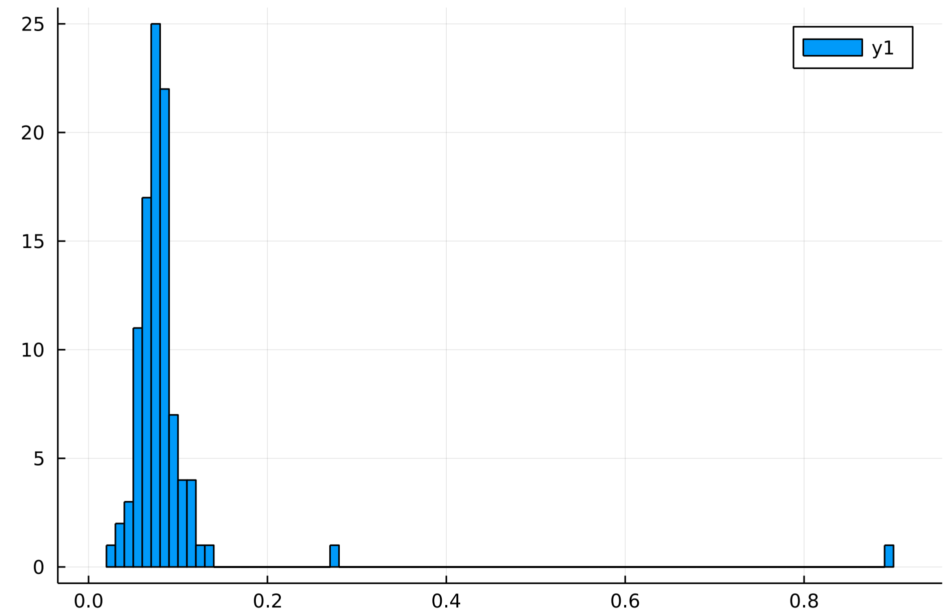
The demand is: $q = a - b\,p$

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.

```
[4.77773, 5.01477, 6.37123, 4.1296, 5.15905, 5.3769, 4.13765, 5.82753, 6.29428, 6.43658, 4
```

```
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.

$\max\ CS - Costs$

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" ⇒ 33.547, "cost" ⇒ 427.812, "price" ⇒ [32.5972, 26.5974, 44.8243, 42.5
```

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

```
33.547018924068574
```

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

```
427.81167924250354
```

```
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)

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

    # 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, Max, 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]);
    @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]);
    @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] - 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] - 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);
```

```julia
        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
```

```julia
results_foc = clear_market_foc(dfclust, tech);
```

```
33.54701942808367
```

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

```
427.81167929403694
```

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

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.