Day 3: Supply I - Building the model

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 2, we will be adding the packages `pyomo` and the solvers `Ipopt` (non-linear solver) and `HiGHS` (mixed linear integer solver).

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.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from pyomo.environ import (
    ConcreteModel, RangeSet, Var, Set, NonNegativeReals, Reals, Binary,
    Constraint, Objective, maximize, minimize, SolverFactory, value
)

# Note: you probably need to install these packages via pip if you haven't already:
# pip install pyomo
# pip install highspy
# conda install -c conda-forge cyipopt
```

Python

```python
dirpath = "/Users/marreguant/Dropbox/TEACHING/BSE/Electricity2026/day3/practicum/"
```

Python

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.

We load the clustered data from the previous session using the CSV syntax. We combine all demand into one variable.

To calibrate the demand curve, 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.

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.

```
def load_and_prepare(dirpath: str):
    dfclust = pd.read_csv(f"{dirpath}/data_jaere_clustered.csv")
    tech = pd.read_csv(f"{dirpath}/data_technology_simple.csv")

    # Re-scaling
    dfclust["weights"] = dfclust["weights"] / dfclust["weights"].sum()

    # One demand type (sum of components)
    dfclust["demand"] = (
        dfclust["q_residential"] + dfclust["q_commercial"] + dfclust["q_industrial"]
    )

    # Calibrate demand curve: demand = a - b * price
    elas = np.array([0.1, 0.2, 0.5, 0.3])
    dfclust["b"] = elas[0] * dfclust["demand"] / dfclust["price"]
    dfclust["b"] = dfclust["b"].mean()  # constant slope across t
    dfclust["a"] = dfclust["demand"] + dfclust["b"] * dfclust["price"]

    # Calibrate imports: imports = am + bm * price
    dfclust["bm"] = elas[3] * dfclust["imports"] / dfclust["price"]
    dfclust["am"] = dfclust["imports"] - dfclust["bm"] * dfclust["price"]

    # Set index names for tech
    tech.index = ["hydronuc","gas1","gas2","gas3","wind","solar"]

    return dfclust, tech

dfclust, tech = load_and_prepare(dirpath)
```

I average slope to avoid outliers

```
dfclust.describe()
```

	price	imports	q_commercial	q_industrial	q_residential	wind_cap	solar_cap	hydronuc	weights	demand	b	a	bm	am
count	100.000000	100.000000	100.000000	100.000000	100.000000	100.000000	100.000000	100.000000	100.000000	100.000000	1.000000e+02	100.000000	100.000000	100.000000
mean	38.505476	7.524257	12.953196	4.122400	11.075335	0.340955	0.260549	5.034605	0.010000	28.150932	9.232864e-02	31.706090	0.077144	5.266980
std	16.032924	1.354806	2.816034	1.154978	3.270312	0.161457	0.273990	1.934417	0.005392	5.057071	1.394770e-17	6.100057	0.110954	0.948364
min	1.663115	4.247898	9.262359	2.600491	4.399502	0.093590	0.001559	2.363158	0.000898	20.605153	9.232864e-02	21.322098	0.012867	2.973528
25%	28.996742	6.536902	10.746648	3.223270	8.850752	0.198401	0.017448	3.439219	0.005846	24.280314	9.232864e-02	27.223763	0.049859	4.575831
50%	37.279651	7.570554	12.388104	3.833469	10.007544	0.334676	0.067964	4.568591	0.008996	27.093761	9.232864e-02	31.136484	0.061616	5.299387
75%	45.780669	8.700303	14.679611	4.744508	13.288205	0.468172	0.545671	6.094806	0.012843	30.784737	9.232864e-02	34.317781	0.076286	6.090212
max	137.115844	9.798136	22.176628	7.620034	20.492173	0.681777	0.793407	9.680505	0.026009	41.809606	9.232864e-02	50.379412	1.135820	6.858695

```
tech
```

	techname	heatrate	heatrate2	capUB	thermal	e	e2	c	c2
hydronuc	Hydro/Nuclear	10.000000	0.000000	1.000	0	0.000000	0.000000	10.000000	0.000000
gas1	Existing 1	6.671990	0.092912	11.500	1	0.360184	0.004886	23.351965	0.325193
gas2	Existing 2	9.794118	0.286247	14.500	1	0.546134	0.011078	34.279413	1.001866
gas3	Existing 3	13.818120	20.535160	0.578	1	0.816768	0.234476	48.363420	71.873060
wind	Wind	0.000000	0.000000	100.000	0	0.000000	0.000000	0.000000	0.000000
solar	Solar	0.000000	0.000000	100.000	0	0.000000	0.000000	0.000000	0.000000

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!

```
# ----------------------------
# Helpers
# ----------------------------

TECHS = ["hydronuc", "gas1", "gas2", "gas3", "wind", "solar"]
```

✓ 0.0s

```
# ----------------------------
# Model 1: Clear market via welfare maximization (Ipopt)
# ----------------------------

def clear_market_min(data, tech, wind_gw=5.0, solar_gw=2.0, solver_name="ipopt"):
    """
    Welfare maximization (NLP), JuMP -> Pyomo.
    tech is indexed by tech name (TECHS).
    """
    T = len(data)
```

Notice we give the function some options.

We set default optimizer to IPopt

We define some indices, together with TECHS

```python
# 1-indexed time parameters (tiny helper to keep Pyomo clean)
w  = {t: float(data.iloc[t-1]["weights"]) for t in range(1, T+1)}
a  = {t: float(data.iloc[t-1]["a"])       for t in range(1, T+1)}
b  = {t: float(data.iloc[t-1]["b"])       for t in range(1, T+1)}
am = {t: float(data.iloc[t-1]["am"])      for t in range(1, T+1)}
bm = {t: float(data.iloc[t-1]["bm"])      for t in range(1, T+1)}

hydronuc = {t: float(data.iloc[t-1]["hydronuc"]) for t in range(1, T+1)}
wind_cap = {t: float(data.iloc[t-1]["wind_cap"]) for t in range(1, T+1)}
solar_cap= {t: float(data.iloc[t-1]["solar_cap"])for t in range(1, T+1)}

m = ConcreteModel()
m.T = RangeSet(1, T)
m.I = Set(initialize=TECHS, ordered=False)

m.price   = Var(m.T, domain=Reals)
m.demand  = Var(m.T, domain=Reals)
m.imports = Var(m.T, domain=Reals)

m.q = Var(m.T, m.I, domain=NonNegativeReals)
```

Some housekeeping to get the data

This is where we start creating a pyomo model!

We define important indices (time and techs)

We create variables for our model specifying the relevant indices and whether they are integers, reals, non-negative, etc.

```python
# Objective: sum_t w_t * (gross_surplus_t - costs_t), written directly (no extra vars)
def obj_rule(mm):
    expr = 0.0
    for t in mm.T:
        D = mm.demand[t]
        # same algebra as Julia:
        gross_surplus = (a[t] - D) * D / b[t] + (D**2) / (2.0 * b[t])
        gen_cost = sum(
            float(tech.at[k, "c"])  * mm.q[t, k]
            + float(tech.at[k, "c2"]) * (mm.q[t, k] ** 2) / 2.0
            for k in mm.I
        )
        imp_cost = (mm.imports[t] - am[t]) ** 2 / (2.0 * bm[t])
        expr += w[t] * (gross_surplus - (gen_cost + imp_cost))
    return expr

m.obj = Objective(rule=obj_rule, sense=maximize)
```

We define the objective function as an expression based on the variables, we keep adding up all time periods.

Gross demand surplus minus costs

Notice the weights!!!

This is how we define the objective of the model, we let it know we are *maximizing* welfare.

The constraints can be used to define equilibrium outcomes

```
# Demand/import schedules + market clearing
m.demand_curve  = Constraint(m.T, expr={t: m.demand[t]  == a[t]  - b[t]  * m.price[t] for t in m.T})
m.imports_curve = Constraint(m.T, expr={t: m.imports[t] == am[t] + bm[t] * m.price[t] for t in m.T})
m.market_clear  = Constraint(m.T, expr={t: m.demand[t] == sum(m.q[t,k] for k in m.I) + m.imports[t] for t in m.T})

# Capacity constraints
m.cap_hydronuc = Constraint(m.T, expr={t: m.q[t,"hydronuc"] <= hydronuc[t] for t in m.T})
m.cap_gas1     = Constraint(m.T, expr={t: m.q[t,"gas1"]     <= float(tech.at["gas1","capUB"]) for t in m.T})
m.cap_gas2     = Constraint(m.T, expr={t: m.q[t,"gas2"]     <= float(tech.at["gas2","capUB"]) for t in m.T})
m.cap_gas3     = Constraint(m.T, expr={t: m.q[t,"gas3"]     <= float(tech.at["gas3","capUB"]) for t in m.T})
m.cap_wind     = Constraint(m.T, expr={t: m.q[t,"wind"]     <= wind_gw  * wind_cap[t] for t in m.T})
m.cap_solar    = Constraint(m.T, expr={t: m.q[t,"solar"]    <= solar_gw * solar_cap[t] for t in m.T})
```

And to ensure that limits to production are respected

```
res = SolverFactory(solver_name).solve(m, tee=False)
term = str(res.solver.termination_condition)

if term=="optimal":
    # Extract
    price  = np.array([value(m.price[t]) for t in m.T])
    demand = np.array([value(m.demand[t]) for t in m.T])
    imports= np.array([value(m.imports[t]) for t in m.T])
    q = {k: np.array([value(m.q[t,k]) for t in m.T]) for k in TECHS}

    w_arr = data["weights"].to_numpy()
    avg_price = float(np.sum(price * w_arr) / np.sum(w_arr))

    # Cost accounting
    gen_cost_t = np.zeros(T)
    for k in TECHS:
        gen_cost_t += float(tech.at[k, "c"]) * q[k] + float(tech.at[k, "c2"]) * (q[k] ** 2) / 2.0
    imp_cost_t = (imports - data["am"].to_numpy()) ** 2 / (2.0 * data["bm"].to_numpy())
    total_cost = float(np.sum(w_arr * (gen_cost_t + imp_cost_t)))

    return {
        "status": term,
        "avg_price": avg_price,
        "price": price,
        "demand": demand,
        "imports": imports,
        "q": q,              # dict: tech -> (T,) array
        "cost": total_cost,
    }
else:
    return {
        "status": term
    }
```

Solving is very easy (if the model is alright!), just call solve (note tee=True prints more info)

Here I construct total cost as a relevant metric

For imports, just the triangle formula

Here we can choose what to report back

```python
results_min = clear_market_min(dfclust, tech)
print("MIN status:", results_min["status"], "avg_price:", results_min.get("avg_price"), "cost:", results_min.get("cost"))
```

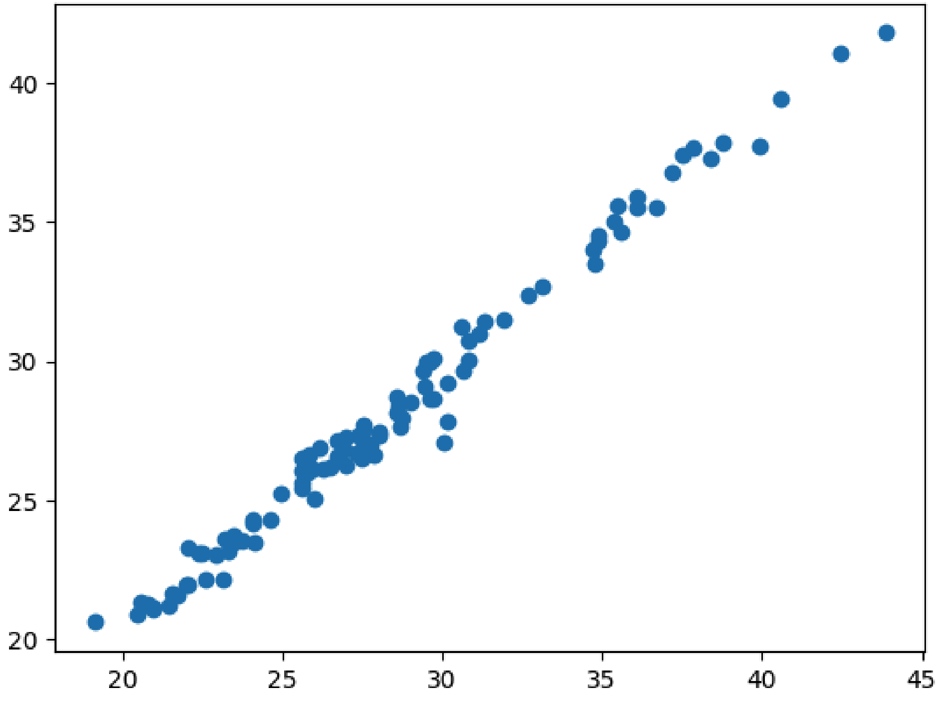
0.1s

```
MIN status: optimal avg_price: 33.78531467481889 cost: 432.3604268698247
```

We can compute the results and check the changes for alternative values of the inputs

```python
plt.scatter(results_min["demand"],dfclust.demand)
plt.show()
```

0.0s




```python
results_more_wind = clear_market_min(dfclust, tech, wind_gw=15.0)
print("MORE WIND status:", results_more_wind["status"], "avg_price:", results_more_wind.get("av
```

1]

0.1s

```
MORE WIND status: optimal avg_price: 30.258364878709536 cost: 343.30107814844746
```

Increased wind, lower prices

Mixed integer solver

More advanced bonus material

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 (`HiGHS`).

```python
# ----------------------------
# Model 2: Clear market via FOCs (HiGHS MILP)
# ----------------------------

def clear_market_foc(data, tech, wind_gw=5.0, solar_gw=2.0, theta=0.0, M=1e3, solver_name="highs"):
    """
    FOC-based equilibrium (MILP), JuMP -> Pyomo.
    tech is indexed by tech name (TECHS).
    """
    T = len(data)

    w  = {t: float(data.iloc[t-1]["weights"]) for t in range(1, T+1)}
    a  = {t: float(data.iloc[t-1]["a"])       for t in range(1, T+1)}
    b  = {t: float(data.iloc[t-1]["b"])       for t in range(1, T+1)}
    am = {t: float(data.iloc[t-1]["am"])      for t in range(1, T+1)}
    bm = {t: float(data.iloc[t-1]["bm"])      for t in range(1, T+1)}

    hydronuc = {t: float(data.iloc[t-1]["hydronuc"]) for t in range(1, T+1)}
    wind_cap = {t: float(data.iloc[t-1]["wind_cap"]) for t in range(1, T+1)}
    solar_cap= {t: float(data.iloc[t-1]["solar_cap"])for t in range(1, T+1)}

    m = ConcreteModel()
    m.T = RangeSet(1, T)
    m.I = Set(initialize=TECHS, ordered=False)

    m.price   = Var(m.T, domain=Reals)
    m.demand  = Var(m.T, domain=Reals)
    m.imports = Var(m.T, domain=Reals)

    m.q = Var(m.T, m.I, domain=NonNegativeReals)
    m.shadow = Var(m.T, m.I, domain=NonNegativeReals)
    m.u1 = Var(m.T, m.I, domain=Binary)  # used
    m.u2 = Var(m.T, m.I, domain=Binary)  # at cap
```

Notice we changed the optimizer/solver

Mixed integer models have 0-1 variables (Bin)

```python
# Tech-specific caps (UB and LB with binaries)
def cap_value(t, k):
    if k == "hydronuc":
        return hydronuc[t]
    if k == "wind":
        return wind_gw * wind_cap[t]
    if k == "solar":
        return solar_gw * solar_cap[t]
    # gas1/2/3
    return float(tech.at[k, "capUB"])

m.cap_ub = Constraint(m.T, m.I, expr={(t, k): (m.q[t, k] <= m.u1[t, k] * cap_value(t, k)) for t in m.T for k in m.I})
m.cap_lb = Constraint(m.T, m.I, expr={(t, k): (m.q[t, k] >= m.u2[t, k] * cap_value(t, k)) for t in m.T for k in m.I})
m.u_link = Constraint(m.T, m.I, expr={(t, k): (m.u1[t, k] >= m.u2[t, k]) for t in m.T for k in m.I})
```

Capacity tells us if a technology is used or if it is used to the max (our binary variables)

```python
# FOC envelope
def foc_expr(mm, t, k):
    return (
        mm.price[t]
        - float(tech.at[k, "c"])
        - float(tech.at[k, "c2"]) * mm.q[t, k]
        - float(theta) * mm.q[t, k] / (b[t] + bm[t]) # markup if theta > 0
        - mm.shadow[t, k]
    )

m.foc_lb = Constraint(m.T, m.I, expr={(t, k): foc_expr(m, t, k) >= -M * (1 - m.u1[t, k])for t in m.T for k in m.I})
m.foc_ub = Constraint(m.T, m.I, expr={(t, k): foc_expr(m, t, k) <= 0.0 for t in m.T for k in m.I})
m.shadow_cap = Constraint(m.T, m.I, expr={(t, k): m.shadow[t, k] <= M * m.u2[t, k] for t in m.T for k in m.I})
```

When theta=1, each unit (here a technology) charges a full markup

The shadow value captures the rents that a technology is earning above their FOC: P > their MC (competitive) or P > their MR (with market power)

Binary variables define the FOC, which can now include market power.
M is a large number that allows the conditions to not be satisfied (e.g., if the first order condition is not binding because a power plant hits its max capacity).
When this happens, the shadow value is allowed to be positive.

```python
# FOC / MILP
results_foc = clear_market_foc(dfclust, tech)
print("FOC status:", results_foc["status"], "avg_price:", results_foc.get("avg_price"), "cost:", results_foc.get("cost"))
```

✓ 0.2s

```
FOC status: optimal avg_price: 33.78531489446415 cost: 432.3604286119609
```

Double checking FOC are equivalent to social planner
(competitive case, theta = 0)

Setting theta=1 to get markups

```
results_foc_mp = clear_market_foc(dfclust, tech, theta=1.0)
print("FOC status:", results_foc_mp["status"], "avg_price:", results_foc_mp.get("avg_price"), "cost:", results_foc_mp.get("cost"))
```

✓ 0.1s

```
FOC status: optimal avg_price: 60.052486740173 cost: 421.35818197663565
```

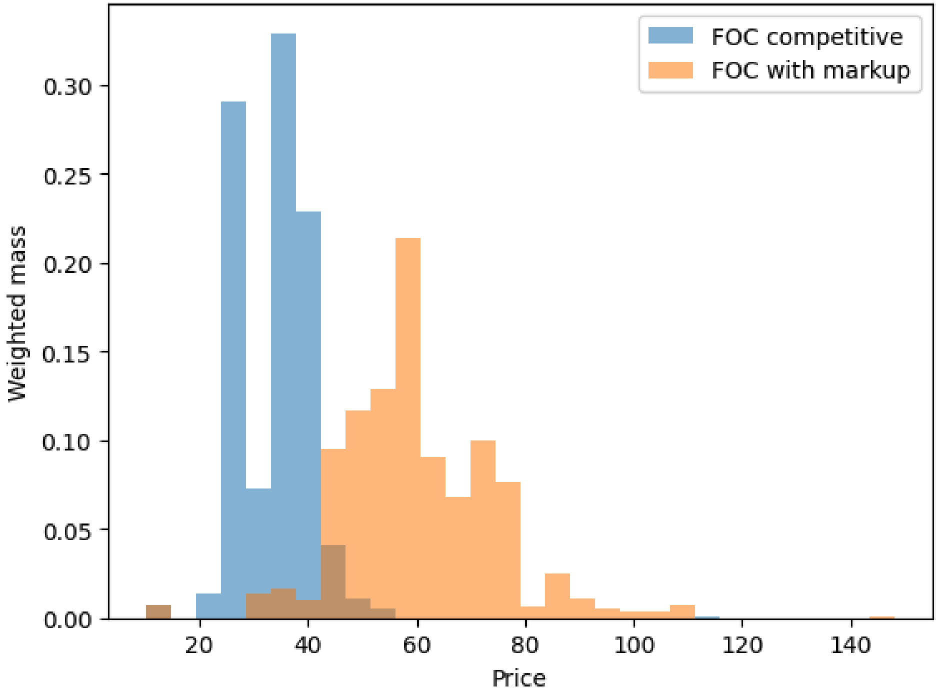
Market power in this toy model double prices!

```
p1 = results_foc["price"]
p2 = results_foc_mp["price"]
w  = dfclust["weights"].to_numpy()

# common bins across both distributions
bins = np.histogram_bin_edges(np.r_[p1, p2], bins=30)

plt.hist(p1, bins=bins, weights=w, alpha=0.5, label="FOC competitive")
plt.hist(p2, bins=bins, weights=w, alpha=0.5, label="FOC with markup")
plt.xlabel("Price")
plt.ylabel("Weighted mass")
plt.legend()
plt.show()
```

✓ 0.0s



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)? [in class]
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 solve for investment? For the FOC approach, this will require a FOC for the zero profit entry condition.