PRINCETON CONSULTANTS

Information Technology and Management Consulting

GUROBI ALLIANCE | PREMIER SERVICE PARTNER

Data-First Optimization Development with Gurobi and Pandas

Irv Lustig, PhD, Optimization Principal, Princeton Consultants
Simon Bowly, PhD, Software Developer, Gurobi

PRINCETON CONSULTANTS
Information Technology and Management Consulting
www.princeton.com

GUROBI ALLIANCE | PREMIER SERVICE PARTNER

Princeton New Jersey

Information Technology & Management Consulting

AI & Optimization *Strategy*
Software Development *Process Improvement*

New York City

- **Mission**: Help our clients optimize their performance by integrating AI into their core business processes

- **Track Record**: 43 years in business
  Over 1800 projects successfully implemented into full production across multiple industries
  Earned long-term strategic relationships with top transportation clients

- **Experience and Retention**: Senior Staff (top 28 consultants) average 15+ years at Princeton; Firm Leadership averages 20+ years; Client Retention > 90% annually

Dr. Irv Lustig, Optimization Principal, Princeton Consultants

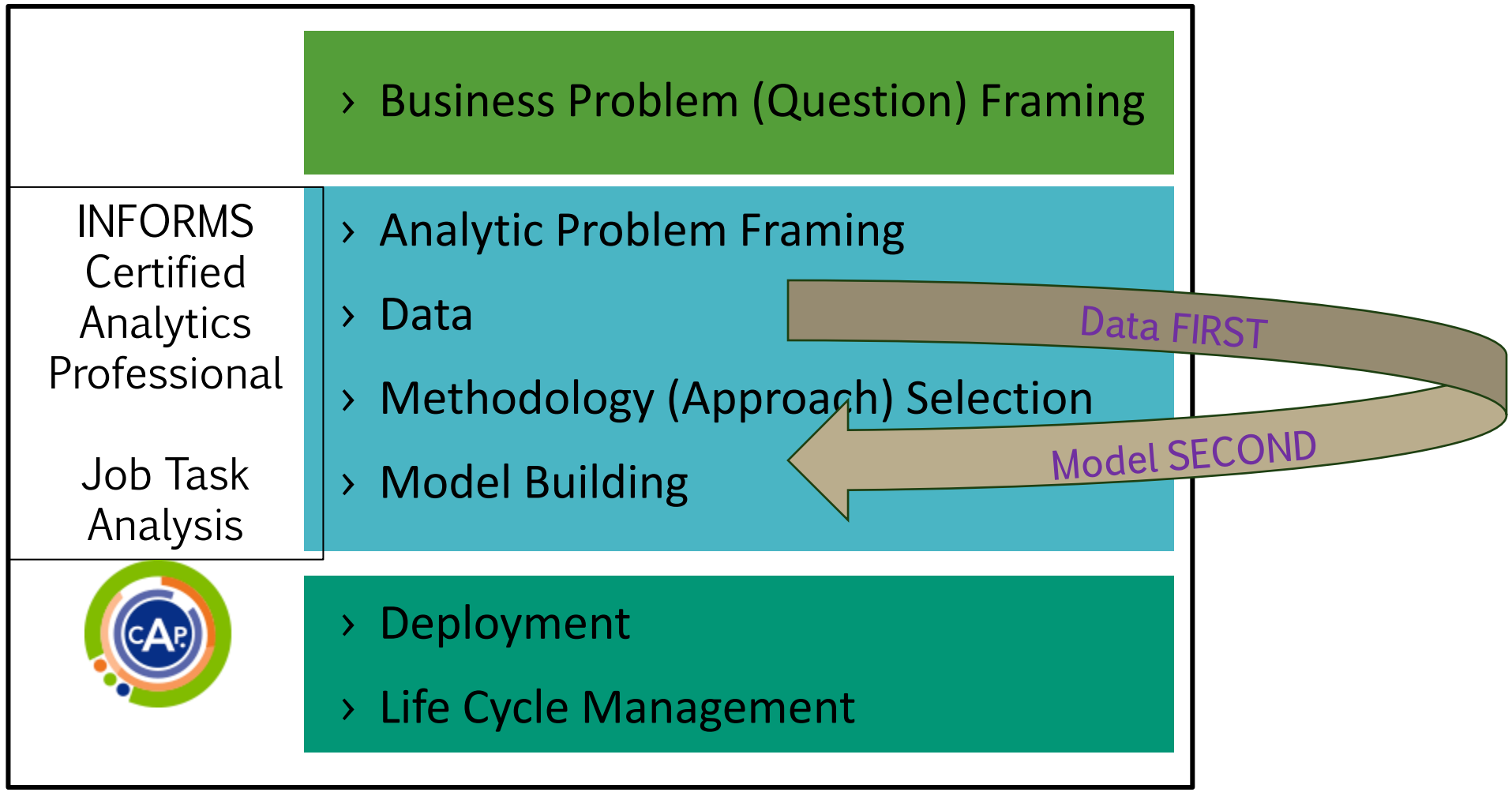
- Director of Numerical Optimization, CPLEX Optimization, Inc.
- ILOG Optimization Product Manager
- ILOG Optimization Evangelist
- ILOG Direct Director of Technical Services
- Vice President, ILOG Direct
- Worldwide Optimization and Supply Chain Technical Sales Manager, IBM
- Manager, Optimization and Mathematical Software, IBM Research

INFORMS Certified Analytics Professional

INFORMS Fellow

pandas Core Team

The Solution Development Life Cycle



See https://info.informs.org/jta for more info on the JTA

DATA FIRST MODEL DEVELOPMENT

A 7-step approach that improves the likelihood of success in optimization projects. This approach:

- Creates useful documentation
- Shortens the overall development time
- Improves code maintainability
- Provides a natural feedback loop to business requirements

1. Understand the Data
2. Define and Document Datasets
3. Collect Data from the Client
4. Document Data Transformations
5. Create Model Documentation
6. Code Optimization Model
7. Validate Results and Iterate

A Simple Example

Gurobipy Example: `netflow.py`

- From the Gurobi Supplied Examples
- A simple multicommodity network flow problem
- Two products ('Pencils' and 'Pens')
  - Produced in 2 cities ('Detroit' and 'Denver')
  - Must be sent to warehouses in 3 cities ('Boston', 'New York', and 'Seattle')
- Satisfy supply/demand
- Flows on the transportation network must respect arc capacity constraints
- The objective is to minimize the sum of the arc transportation costs

Gurobi Example: A Poor Data Representation!

```
# Base data
commodities = ["Pencils", "Pens"]
nodes = ["Detroit", "Denver", "Boston", "New York", "Seattle"]

arcs, capacity = gp.multidict(
    {
        ("Detroit", "Boston"): 100,
        ("Detroit", "New York"): 80,
        ("Detroit", "Seattle"): 120,
        ("Denver", "Boston"): 120,
        ("Denver", "New York"): 120,
        ("Denver", "Seattle"): 120,
    }
)

# Cost for triplets commodity-source-destination
cost = {
    ("Pencils", "Detroit", "Boston"): 10,
    ("Pencils", "Detroit", "New York"): 20,
    ("Pencils", "Detroit", "Seattle"): 60,
    ("Pencils", "Denver", "Boston"): 40,
    ("Pencils", "Denver", "New York"): 40,
    ("Pencils", "Denver", "Seattle"): 30,
    ("Pens", "Detroit", "Boston"): 20,
    ("Pens", "Detroit", "New York"): 20,
    ("Pens", "Detroit", "Seattle"): 80,
    ("Pens", "Denver", "Boston"): 60,
    ("Pens", "Denver", "New York"): 70,
    ("Pens", "Denver", "Seattle"): 30,
}
```

```
# Supply (> 0) and demand (< 0) for pairs of
commodity-city
inflow = {
    ("Pencils", "Detroit"): 50,
    ("Pencils", "Denver"): 60,
    ("Pencils", "Boston"): -50,
    ("Pencils", "New York"): -50,
    ("Pencils", "Seattle"): -10,
    ("Pens", "Detroit"): 60,
    ("Pens", "Denver"): 40,
    ("Pens", "Boston"): -40,
    ("Pens", "New York"): -30,
    ("Pens", "Seattle"): -30,
}
```

What's Poor About This?

- Data inside code
- Dependent on Gurobi `multidict`
- Commodities and Nodes repeated
  - These can be inferred!
- Mixing Supply and Demand Data
- If there were transshipment nodes, `inflow` could have many zero values

A Better Data Representation: Tables

1. Understand the Data
3. Collect Data from the Client

	A	B	C
1	From	To	capacity
2	Detroit	Boston	100
3	Detroit	New York	80
4	Detroit	Seattle	120
5	Denver	Boston	120
6	Denver	New York	120
7	Denver	Seattle	120

capacity

	A	B	C	D
1	Product	From	To	cost
2	Pencils	Detroit	Boston	10
3	Pencils	Detroit	New York	20
4	Pencils	Detroit	Seattle	60
5	Pencils	Denver	Boston	40
6	Pencils	Denver	New York	40
7	Pencils	Denver	Seattle	30
8	Pens	Detroit	Boston	20
9	Pens	Detroit	New York	20
10	Pens	Detroit	Seattle	80
11	Pens	Denver	Boston	60
12	Pens	Denver	New York	70
13	Pens	Denver	Seattle	30
14				

cost

	A	B	C
1	Product	Node	supply
2	Pencils	Detroit	50
3	Pencils	Denver	60
4	Pens	Detroit	60
5	Pens	Denver	40
6			

supply

	A	B	C
1	Product	Node	demand
2	Pencils	Boston	50
3	Pencils	New York	50
4	Pencils	Seattle	10
5	Pens	Boston	40
6	Pens	New York	30
7	Pens	Seattle	30
8			

demand

```
import pandas as pd
raw_data = pd.read_excel("mcfdata.v1.xlsx", sheet_name=None)
data = {
    k: df.set_index(df.columns[:-1].to_list())[df.columns[-1]]
    for k, df in raw_data.items()
}
```

Pandas Representation: A Dictionary of Series

1. Understand the Data
3. Collect Data from the Client

capacity

From	To	capacity
Detroit	Boston	100
	New York	80
	Seattle	120
Denver	Boston	120
	New York	120
	Seattle	120

cost

Product	From	To	cost
Pencils	Detroit	Boston	10
		New York	20
		Seattle	60
	Denver	Boston	40
		New York	40
		Seattle	30
Pens	Detroit	Boston	20
		New York	20
		Seattle	80
	Denver	Boston	60
		New York	70
		Seattle	30

supply

Product	Node	supply
Pencils	Detroit	50
	Denver	60
Pens	Detroit	60
	Denver	40

demand

Product	Node	demand
Pencils	Boston	50
	New York	50
	Seattle	10
Pens	Boston	40
	New York	30
	Seattle	30

Document the Data Model

Sets

Notation	Meaning	Table Locations

$\mathcal{N}$	Set of network nodes	`cost`: Columns `From`, `To`
		`capacity`: Columns `From`, `To`
		`supply`: Column `Node`
		`demand`: Column `Node`

$\mathcal{P}$	Set of products (commodities)	`cost`: Column `Product`
		`supply`: Column `Product`
		`demand`: Column `Product`

$\mathcal{A}$	Set of arcs (n_f, n_t), $n_f, n_t \in \mathcal{A}$	`cost`: Columns `From`, `To`
$\mathcal{P}_a$	Set of products $p \in \mathcal{P}$ that can be carried on arc $a \in \mathcal{A}$	`cost`: Columns `Product`, `From`, To
$\mathcal{A}_p$	Set of arcs $a \in \mathcal{A}$ that can carry product $p \in \mathcal{P}$	`cost`: Columns `Product`, `From`, To

Numerical Input Values

The input data is converted to pandas `Series`, so the name of each `Series` is also the name of the value.

Notation	Meaning	Table Name/Value Column	Index Columns
κ_a	Capacity of arc $a \in \mathcal{A}$	`capacity`	`From`, `To`
π_{ap}	Cost of carrying product p on arc $a \in \mathcal{A}$, $p \in \mathcal{P}_a$,	`cost`	`Product`, `From`, `To`
σ_{pn}	Supply of product $p \in \mathcal{P}$ at node $n \in \mathcal{N}$. Defaults to 0	`supply`	`Node`
δ_{pn}	Demand of product $p \in \mathcal{P}$ at node $n \in \mathcal{N}$. Defaults to 0	`demand`	`Node`

Computing Sets from Data

- In a Data First approach, sets are *inferred* from data
  - The set $\mathcal{P}$ of products can appear in any of the tables: `supply`, `demand` and `cost` .
  - The set $\mathcal{N}$ of nodes can appear in any of the tables: `capacity`, `supply`, `demand` and `cost`

```
commodities = set(
    pd.concat(
        [
            data[dfname].index.to_frame()["Product"]
            for dfname in ["supply", "demand", "cost"]
        ]
    ).unique()
)
nodes = set(
    pd.concat(
        [
            data[dfname].index.to_frame()[fromto].rename("Node")
            for dfname in ["capacity", "cost"]
            for fromto in ["From", "To"]
        ]
        + [data[dfname].index.to_frame()["Node"] for dfname in ["supply", "demand"]]
    ).unique()
)
```

Computing the Net Flow Per Node

- The net flow μ_{pn} for each product $p \in \mathcal{P}$ and node $n \in \mathcal{N}$ is the sum of the supply less the demand. For transshipment nodes, this value is 0. This is called `inflow` in the code.

```
inflow = pd.concat(
    [
        data["supply"].rename("net"),
        data["demand"].rename("net") * -1,
        pd.Series(
            0,
            index=pd.MultiIndex.from_product(
                [commodities, nodes], names=["Product", "Node"]
            ),
            name="net",
        ),
    ]
).groupby(["Product", "Node"]).sum()
```

- Note that the above allows supply and demand to co-exist at any node, and for any node to act as transshipment nodes
- This is not true in the original Gurobi example.

Time to Model

5. Create Model Documentation

Decision Variables

- The model will have one set of decision variables: X_{pa} for $p \in \mathcal{P}$, $a \in \mathcal{A}_p$ represents the amount shipped of product p on arc a. We will call this variable `flow` in the code.
- The cost of shipment is π_{ap}.
- This defines the objective function:

$$\text{minimize} \quad \sum_{a \in \mathcal{A}} \sum_{p \in \mathcal{P}_a} \pi_{ap} X_{pa}$$

Constraints

- **Flow on each arc is capacitated**

$$\sum_{p \in \mathcal{P}_a} X_{pa} \leq \kappa_a \qquad \forall a \in \mathcal{A}$$

- **Conservation of Flow**
- For each node and each product, the flow out of the node, less the flow into the node is equal to the net flow.

$$\sum_{(n,n_t) \in A_p} X_{p(n,n_t)} - \sum_{(n_f,n)} X_{p(n_f,n)} = \mu_{pn} \qquad \forall p \in \mathcal{P}, n \in \mathcal{N}$$

Using gurobipy-pandas: Decision Variables


```
import gurobipy as grb
import gurobipy_pandas as gppd
m = grb.Model("netflow")
flow = gppd.add_vars(m, data["cost"], obj=data["cost"], name="flow")
```

Decision Variables Share Index with pandas Series

Series has the values for the objective

```
Product  From      To
Pencils  Detroit   Boston        <gurobi.Var flow[Pencils,Detroit,Boston]>
                   New York     <gurobi.Var flow[Pencils,Detroit,New_York]>
                   Seattle       <gurobi.Var flow[Pencils,Detroit,Seattle]>
         Denver    Boston          <gurobi.Var flow[Pencils,Denver,Boston]>
                   New York      <gurobi.Var flow[Pencils,Denver,New_York]>
                   Seattle       <gurobi.Var flow[Pencils,Denver,Seattle]>
Pens     Detroit   Boston           <gurobi.Var flow[Pens,Detroit,Boston]>
                   New York       <gurobi.Var flow[Pens,Detroit,New_York]>
                   Seattle         <gurobi.Var flow[Pens,Detroit,Seattle]>
         Denver    Boston           <gurobi.Var flow[Pens,Denver,Boston]>
                   New York       <gurobi.Var flow[Pens,Denver,New_York]>
                   Seattle         <gurobi.Var flow[Pens,Denver,Seattle]>
Name: flow, dtype: object
```

Using gurobipy-pandas: Capacity Constraints

$$\sum_{p \in \mathcal{P}_a} X_{pa} \leq \kappa_a \qquad \forall a \in \mathcal{A}$$

```
pd.concat(
    [flow.groupby(["From", "To"]).agg(grb.quicksum), data["capacity"]], axis=1
).gppd.add_constrs(m, "flow <= capacity", name="cap")
```

		flow	capacity	cap
From	To			
Denver	Boston	flow[Pencils,Denver,Boston] + flow[Pens,Denver...	120	<gurobi.Constr cap[Denver,Boston]>
	New York	flow[Pencils,Denver,New_York] + flow[Pens,Denv...	120	<gurobi.Constr cap[Denver,New_York]>
	Seattle	flow[Pencils,Denver,Seattle] + flow[Pens,Denve...	120	<gurobi.Constr cap[Denver,Seattle]>
Detroit	Boston	flow[Pencils,Detroit,Boston] + flow[Pens,Detro...	100	<gurobi.Constr cap[Detroit,Boston]>
	New York	flow[Pencils,Detroit,New_York] + flow[Pens,Det...	80	<gurobi.Constr cap[Detroit,New_York]>
	Seattle	flow[Pencils,Detroit,Seattle] + flow[Pens,Detr...	120	<gurobi.Constr cap[Detroit,Seattle]>

Using gurobipy-pandas: Flow Constraints

$$\sum_{(n,n_t)\in A_p} X_{p(n,n_t)} - \sum_{(n_f,n)} X_{p(n_f,n)} = \mu_{pn} \qquad \forall p \in \mathcal{P}, n \in \mathcal{N}$$

```
pd.concat(
    [
        flow.rename_axis(index={"From": "Node"})
        .groupby(["Product", "Node"])
        .agg(grb.quicksum)
        .rename("flowout"),

        flow.rename_axis(index={"To": "Node"})
        .groupby(["Product", "Node"])
        .agg(grb.quicksum)
        .rename("flowin"),

        inflow,
    ],
    axis=1,
).fillna(0).gppd.add_constrs(m, "flowout - flowin == net", name="node")
```

$\sum_{(n,n_t)\in A_p} X_{p(n,n_t)}$

$\sum_{(n_f,n)} X_{p(n_f,n)}$

μ_{pn}

Flow Constraint in pandas

Product	Node	flowout	flowin	net	node
Pencils	Denver	flow[Pencils,Denver,Boston] + flow[Pencils,Den...	0	60	<gurobi.Constr node[Pencils,Denver]>
	Detroit	flow[Pencils,Detroit,Boston] + flow[Pencils,De...	0	50	<gurobi.Constr node[Pencils,Detroit]>
Pens	Denver	flow[Pens,Denver,Boston] + flow[Pens,Denver,Ne...	0	40	<gurobi.Constr node[Pens,Denver]>
	Detroit	flow[Pens,Detroit,Boston] + flow[Pens,Detroit,...	0	60	<gurobi.Constr node[Pens,Detroit]>
Pencils	Boston	0	flow[Pencils,Detroit,Boston] + flow[Pencils,De...	-50	<gurobi.Constr node[Pencils,Boston]>
	New York	0	flow[Pencils,Detroit,New_York] + flow[Pencils,...	-50	<gurobi.Constr node[Pencils,New_York]>
	Seattle	0	flow[Pencils,Detroit,Seattle] + flow[Pencils,D...	-10	<gurobi.Constr node[Pencils,Seattle]>
Pens	Boston	0	flow[Pens,Detroit,Boston] + flow[Pens,Denver,B...	-40	<gurobi.Constr node[Pens,Boston]>
	New York	0	flow[Pens,Detroit,New_York] + flow[Pens,Denver...	-30	<gurobi.Constr node[Pens,New_York]>
	Seattle	0	flow[Pens,Detroit,Seattle] + flow[Pens,Denver,...	-30	<gurobi.Constr node[Pens,Seattle]>

› Note the 0 values are created via `fillna(0)` because there is no flow into supply nodes and flow out of demand nodes, so the `pd.concat` operation would create NA values, which cannot be passed to Gurobi

Optimize and Get Results


```
m.optimize()
soln = flow.gppd.X
soln.to_frame().query("flow > 0").sort_index()
```

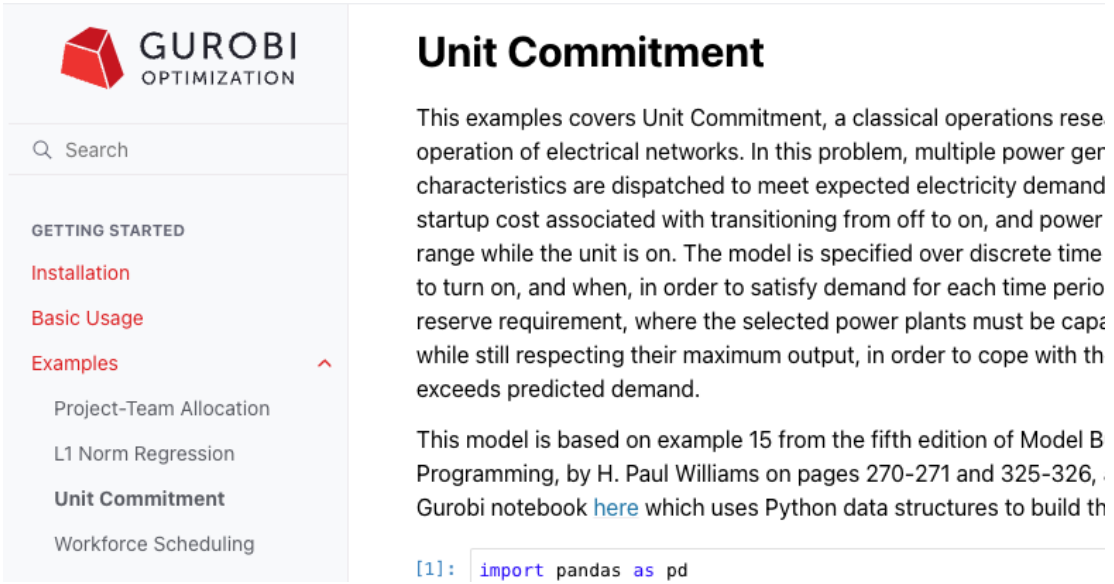
- Note the use of the `gppd` accessor to get the attribute `X` of the decision variables `flow` use
  - All Gurobi attributes can be accessed in this way!

			flow
Product	**From**	**To**	
Pencils	Denver	New York	50.0
		Seattle	10.0
	Detroit	Boston	50.0
Pens	Denver	Boston	10.0
		Seattle	30.0
	Detroit	Boston	30.0
		New York	30.0

Energy Systems Modeling: Unit Commitment Problem

- Dispatch generators to meet expected demand
- Keep a target level of available capacity online
- Minimize overall generation cost
- Many more complex constraints in real life!

gurobipy-pandas.readthedocs.io



Pandas Representation: DataFrames

GUROBI OPTIMIZATION

`generator_data`

generator_class	num_available	min_output	max_output	cost_per_hour	marginal_cost	startup_cost	state0
thermal1	12	850.0	2000.0	1000.0	2.0	2000.0	0
thermal2	10	1250.0	1750.0	2600.0	1.3	1000.0	0
thermal3	5	1500.0	4000.0	3000.0	3.0	500.0	0

- Generators have:
  - Fixed costs and marginal costs
  - Startup costs
  - Lower and upper generation limits when online
- Each time period has:
  - Demand to be met
  - A minimum online capacity
- Model captures state of each class in each time period

`time_period_data`

time_period	expected_demand	minimum_active_capacity
2024-07-19 06:00:00	15000.0	17250.0
2024-07-19 07:00:00	30000.0	34500.0
2024-07-19 08:00:00	25000.0	28750.0
2024-07-19 09:00:00	40000.0	46000.0
2024-07-19 10:00:00	27000.0	31050.0

Decision Variables: gurobipy-pandas

```
generators = (
    # Decision variables for each class at each time period
    pd.DataFrame(index=pd.MultiIndex.from_product([
        generator_data.index,
        time_period_data.index,
    ]))
    .join(generator_data)
    # Three classes of variables capture output and on/off state
    .gppd.add_vars(model, name="output")
    .gppd.add_vars(
        model,
        vtype=GRB.INTEGER,
        ub="num_available",
        name="num_active",
    )
    .gppd.add_vars(model, vtype=GRB.INTEGER, name="num_startup”)
)
```

Combined time-expanded index (with generator data)

Each accessor call creates one variable per row

Time-Expanded Format

Time-expanded pandas MultiIndex

		output	num_active	num_startup	cost_per_hour	startup_cost	min_output	max_output
generator_class	time_period							
thermal1	2024-07-19 06:00:00	output[thermal1,0600]	num_active[thermal1,0600]	num_startup[thermal1,0600]	1000.0	2000.0	850.0	2000.0
	2024-07-19 07:00:00	output[thermal1,0700]	num_active[thermal1,0700]	num_startup[thermal1,0700]	1000.0	2000.0	850.0	2000.0
	2024-07-19 08:00:00	output[thermal1,0800]	num_active[thermal1,0800]	num_startup[thermal1,0800]	1000.0	2000.0	850.0	2000.0
	2024-07-19 09:00:00	output[thermal1,0900]	num_active[thermal1,0900]	num_startup[thermal1,0900]	1000.0	2000.0	850.0	2000.0
	2024-07-19 10:00:00	output[thermal1,1000]	num_active[thermal1,1000]	num_startup[thermal1,1000]	1000.0	2000.0	850.0	2000.0
thermal2	2024-07-19 06:00:00	output[thermal2,0600]	num_active[thermal2,0600]	num_startup[thermal2,0600]	2600.0	1000.0	1250.0	1750.0
	2024-07-19 07:00:00	output[thermal2,0700]	num_active[thermal2,0700]	num_startup[thermal2,0700]	2600.0	1000.0	1250.0	1750.0
	2024-07-19 08:00:00	output[thermal2,0800]	num_active[thermal2,0800]	num_startup[thermal2,0800]	2600.0	1000.0	1250.0	1750.0
	2024-07-19 09:00:00	output[thermal2,0900]	num_active[thermal2,0900]	num_startup[thermal2,0900]	2600.0	1000.0	1250.0	1750.0

Decision variables share this common index

Generator data is expanded (makes model building easier)

Constraints: Generator Limits

Time-expanded dataframe

```
# Generators must operate within their limits
generators.gppd.add_constrs(
    model,
    "output >= min_output * num_active",
    name="lower_limit",
).gppd.add_constrs(
    model,
    "output <= max_output * num_active",
    name="upper_limit",
)
```

One constraint per time-expanded row

New constraint defines a relationship between columns

		output	num_active	min_output	max_output	lower_limit
generator_class	time_period					
thermal1	2024-07-19 06:00:00	output[thermal1,0600]	num_active[thermal1,0600]	850.0	2000.0	<gurobi.Constr lower_limit[thermal1,0600]>
	2024-07-19 07:00:00	output[thermal1,0700]	num_active[thermal1,0700]	850.0	2000.0	<gurobi.Constr lower_limit[thermal1,0700]>

Constraints: Satisfying Demand

```
# Predicted demand must be exactly satisfied
demand_constraint = gppd.add_constrs(
    model,
    generators.groupby("time_period")["output"].sum(),
    GRB.EQUAL,
    time_period_data["expected_demand"],
)
```

These two expressions have compatible indexes

		output
generator_class	time_period	
thermal1	2024-07-19 06:00:00	output[thermal1,0600]
	2024-07-19 07:00:00	output[thermal1,0700]
	2024-07-19 08:00:00	output[thermal1,0800]
	2024-07-19 09:00:00	output[thermal1,0900]
	2024-07-19 10:00:00	output[thermal1,1000]
thermal2	2024-07-19 06:00:00	output[thermal2,0600]
	2024-07-19 07:00:00	output[thermal2,0700]
	2024-07-19 08:00:00	output[thermal2,0800]
	2024-07-19 09:00:00	output[thermal2,0900]

groupby + sum creates a constraint across generators for one time period

Constraints: Reserve Capacity

```
# Ensure that the active generators during each time
# period are capable of meeting the reserve demand.
active_capacity = (
    (generators["max_output"] * generators["num_active"])
    .groupby("time_period").sum()
)
active_capacity_constraint = gppd.add_constrs(
    model,
    active_capacity,
    GRB.GREATER_EQUAL,
    time_period_data["minimum_active_capacity"],
)
```

Keep this for results step

time_period	
2024-07-19 06:00:00	2000.0 num_active[thermal1,0600] + 1750.0 num_...
2024-07-19 07:00:00	2000.0 num_active[thermal1,0700] + 1750.0 num_...
2024-07-19 08:00:00	2000.0 num_active[thermal1,0800] + 1750.0 num_...
2024-07-19 09:00:00	2000.0 num_active[thermal1,0900] + 1750.0 num_...
2024-07-19 10:00:00	2000.0 num_active[thermal1,1000] + 1750.0 num_...

Expression measuring total active capacity in a given time period

Constraints: Startup Costs

```python
def count_startups(group):
    return gppd.add_constrs(
        model,
        group["num_startup"].iloc[1:],
        GRB.GREATER_EQUAL,
        group["num_active"].diff().dropna(),
        name="startup",
    ).to_frame()

startup = generators.groupby("generator_class").apply(count_startups)
```

diff() captures the change from one time period to the next

		num_active	num_startup
generator_class	time_period		
thermal1	2024-07-19 06:00:00	num_active[thermal1,0600]	num_startup[thermal1,0600]
	2024-07-19 07:00:00	num_active[thermal1,0700]	num_startup[thermal1,0700]
	2024-07-19 08:00:00	num_active[thermal1,0800]	num_startup[thermal1,0800]
	2024-07-19 09:00:00	num_active[thermal1,0900]	num_startup[thermal1,0900]
	2024-07-19 10:00:00	num_active[thermal1,1000]	num_startup[thermal1,1000]
thermal2	2024-07-19 06:00:00	num_active[thermal2,0600]	num_startup[thermal2,0600]
	2024-07-19 07:00:00	num_active[thermal2,0700]	num_startup[thermal2,0700]
	2024-07-19 08:00:00	num_active[thermal2,0800]	num_startup[thermal2,0800]
	2024-07-19 09:00:00	num_active[thermal2,0900]	num_startup[thermal2,0900]
	2024-07-19 10:00:00	num_active[thermal2,1000]	num_startup[thermal2,1000]

groupby creates a set of constraints per generator class

Objective: Total Cost

```
# Minimize total cost objective
model.setObjective(
    (
        # Fixed hourly costs for started generators
        generators["cost_per_hour"] * generators["num_active"]
        # Marginal hourly cost of generation above the minimum
        + generators["marginal_cost"] * (
            generators["output"] -
            generators["num_active"] * generators["min_output"]
        )
        # Startup costs for newly active generators in each period
        + generators["startup_cost"] * generators["num_startup"]
    ).sum(),
    sense=GRB.MINIMIZE
)
```

Creates a cost expression per period and class

Aggregate objective

		output	num_active	num_startup	cost_per_hour	startup_cost	min_output	max_output
generator_class	time_period							
thermal1	2024-07-19 06:00:00	output[thermal1,0600]	num_active[thermal1,0600]	num_startup[thermal1,0600]	1000.0	2000.0	850.0	2000.0
	2024-07-19 07:00:00	output[thermal1,0700]	num_active[thermal1,0700]	num_startup[thermal1,0700]	1000.0	2000.0	850.0	2000.0

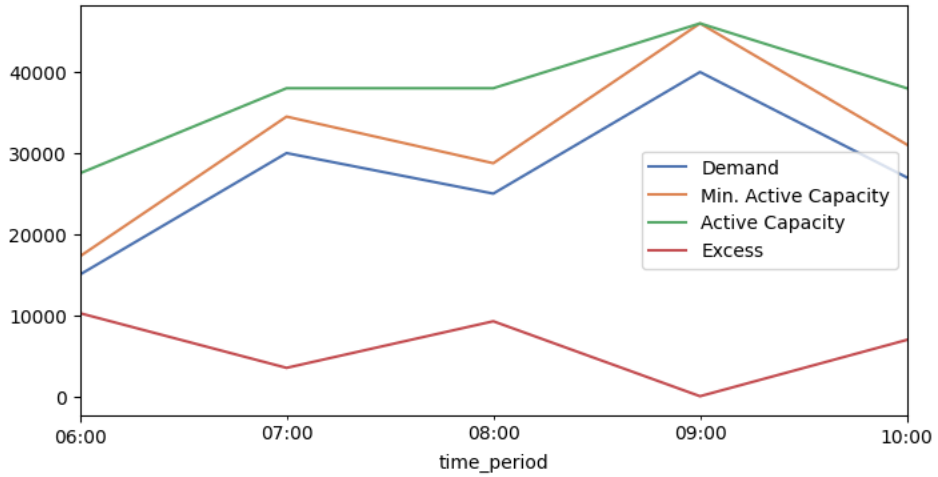
Optimize and Extract Results

```
model.optimize()

results = pd.DataFrame({
    "Demand": time_period_data["expected_demand"],
    "Min. Active Capacity": time_period_data["minimum_active_capacity"],
    "Active Capacity": active_capacity.gppd.get_value(),
    "Excess": (-active_capacity_constraint.gppd.Slack),
})

results.plot.line()
```

Value of expressions in the solution



Slack in constraints (Gurobi attribute query)

CONCLUSION

- Data First Model Development Focuses on the Data
  - A proven practice for delivering high quality optimization applications
- `gurobipy-pandas`
  - Takes advantage of pandas for managing data
  - Allow Gurobi objects to be placed into pandas `DataFrame`s and `Series`
  - Provides capability to write code that executes quickly when creating models
  - Requires good knowledge of `pandas` to understand how to map mathematical formulations into `gurobipy-pandas`

1. Understand the Data
2. Define and Document Datasets
3. Collect Data from the Client
4. Document Data Transformations
5. Create Model Documentation
6. Code Optimization Model
7. Validate Results and Iterate

Thanks!

- More info:
  - https://gurobipy-pandas.readthedocs.io/en/stable/
  - https://www.princetonoptimization.com/blog/blog/rapid-optimization-model-development-python-and-pandas-7-steps