

SOP: Running the GEP Model in Jupyter Notebook

1. Introduction

This document provides step-by-step instructions to run the GEP model in Jupyter Notebook. The objective is to simulate energy dispatch scenarios, calculate shadow prices, and generate visualizations for further analysis.

2. Preparing the Environment

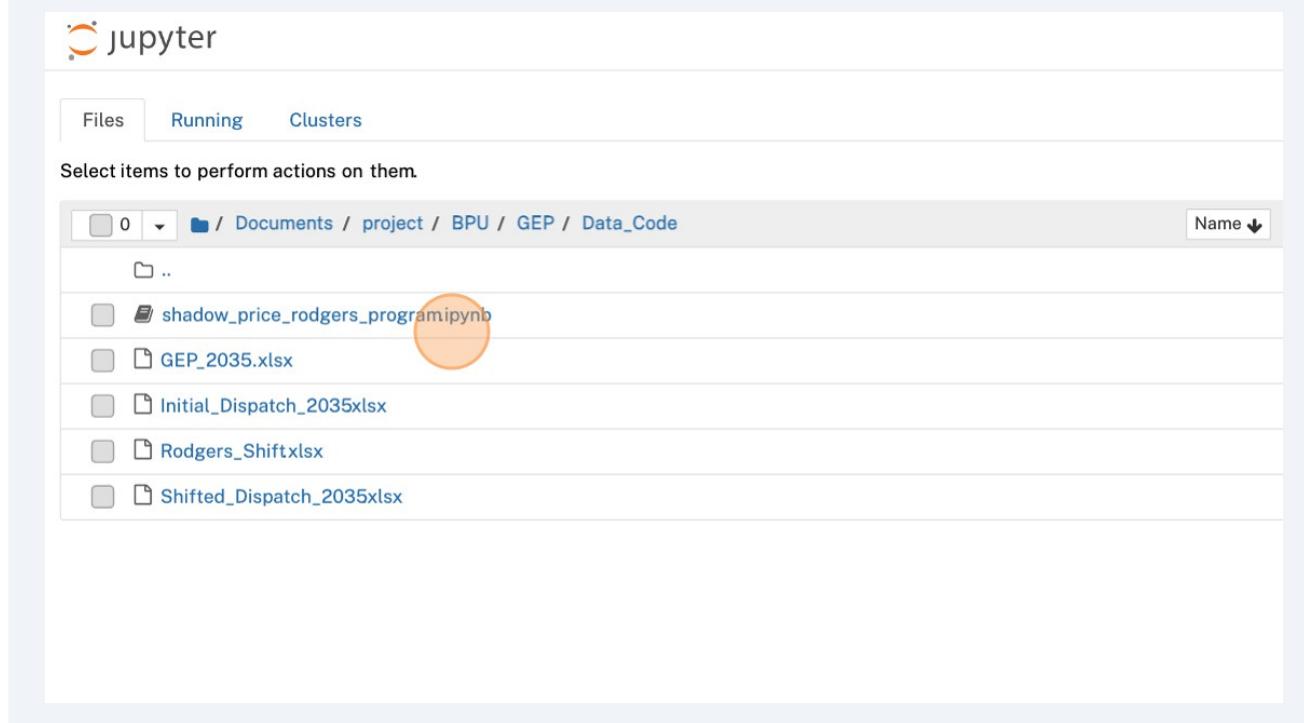
- 1) Install Python and Required Libraries:
 - a) Ensure Python 3.9 or above is installed.
 - b) Install required libraries.
- 2) Set Up the Working Directory:
Place the following data files in the same directory as the Jupyter Notebook:
 - a) GEP_2035.xlsx: Input data file for GEP parameters.
 - b) Initial_Dispatch_2035.xlsx: Initial energy dispatch data.
 - c) Shifted_Dispatch_2035.xlsx: Shifted energy demand profiles.

3. Step-by-Step Instructions

Step 1: Open the Jupyter Notebook

Locate the file shadow_price_rodgers_program.ipynb and open it in Jupyter Notebook.

- 1 Click "shadow_price_rodgers_program.ipynb"



Step 2: Import Required Libraries

Click the cell labeled import and run it to load all dependencies.

2

Click "import"

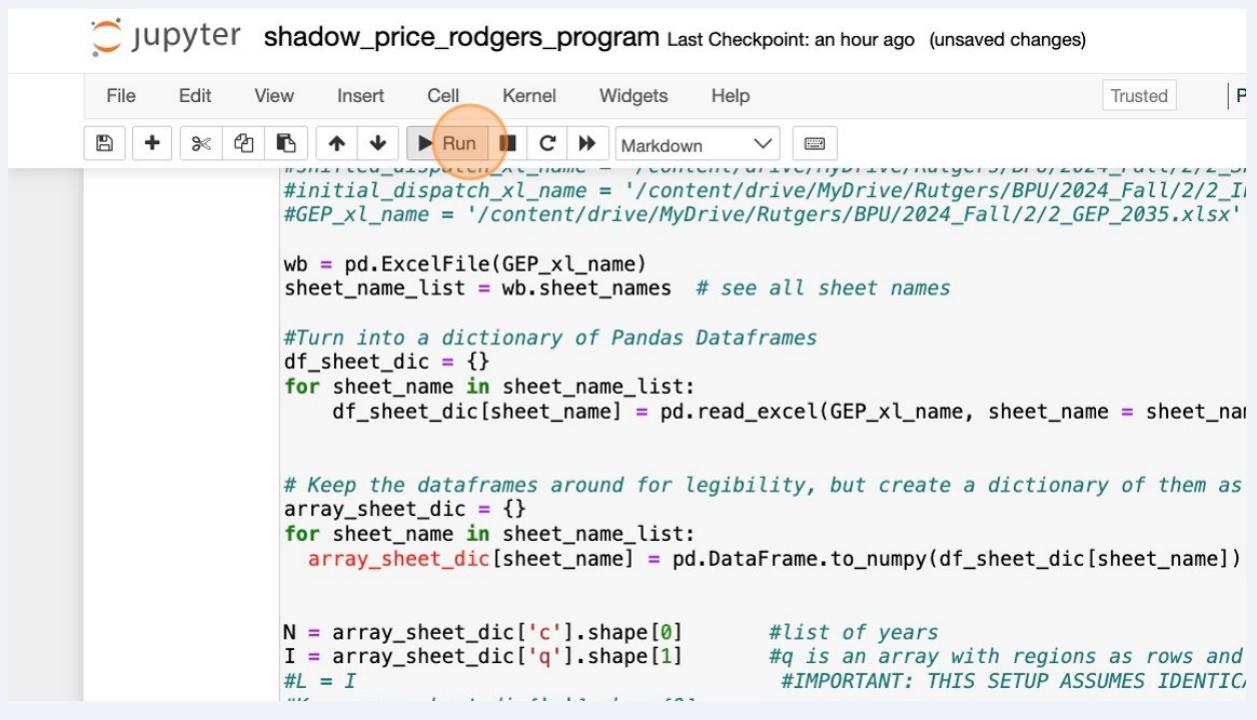
```
##  
#####  
  
# !apt-get install -y -qq glpk-utils #for the GLPK solver in pulp, used to make  
# !pip install pulp  
# !pip install pandas openpyxl  
# !pip install xlsxwriter  
  
#####  
##  
## Import initial libraries. Not all are needed.  
##  
#####  
  
import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
from math import *  
import time  
import random  
from numpy import linalg as LA  
import pickle  
import warnings  
warnings.filterwarnings("ignore", category=UserWarning)  
import scipy  
from scipy import optimize as opt  
import csv  
import os  
import nlpn
```

Step 3: This step loads the Excel input files required for the GEP model. The files include:

- i. `GEP_2035.xlsx`: Contains parameters for the GEP model.
- ii. `Initial_Dispatch_2035.xlsx`: Initial energy dispatch schedule.
- iii. `Shifted_Dispatch_2035.xlsx`: Data for shifted demand.

Ensure the files are located in the specified directory and have no missing fields.

- 3 Click "Run"



```
#initial_dispatch_xl_name = '/content/drive/MyDrive/Rutgers/BPU/2024_Fall/2/2_Initial_Dispatch_2035.xlsx'
#GEP_xl_name = '/content/drive/MyDrive/Rutgers/BPU/2024_Fall/2/2_GEP_2035.xlsx'

wb = pd.ExcelFile(GEP_xl_name)
sheet_name_list = wb.sheet_names # see all sheet names

#Turn into a dictionary of Pandas Dataframes
df_sheet_dic = {}
for sheet_name in sheet_name_list:
    df_sheet_dic[sheet_name] = pd.read_excel(GEP_xl_name, sheet_name = sheet_name)

# Keep the dataframes around for legibility, but create a dictionary of them as
array_sheet_dic = {}
for sheet_name in sheet_name_list:
    array_sheet_dic[sheet_name] = pd.DataFrame.to_numpy(df_sheet_dic[sheet_name])

N = array_sheet_dic['c'].shape[0]           #list of years
I = array_sheet_dic['q'].shape[1]           #q is an array with regions as rows and
#L = I                                     #IMPORTANT: THIS SETUP ASSUMES IDENTICAL
```

- 4 Click "1) GEP"

4) Dispatch with Reallocation of Load

Shifted Demand Plotter

Shifted Demand Dispatcher Master Function

Run to Show with Unmet Demand Shifted

1) GEP

Initialize

```
In [30]: GEP_xl_name = './GEP_2035.xlsx'
initial_dispatch_xl_name = './Initial_Dispatch_2035.xlsx'
shifted_dispatch_xl_name = './Shifted_Dispatch_2035.xlsx'
```

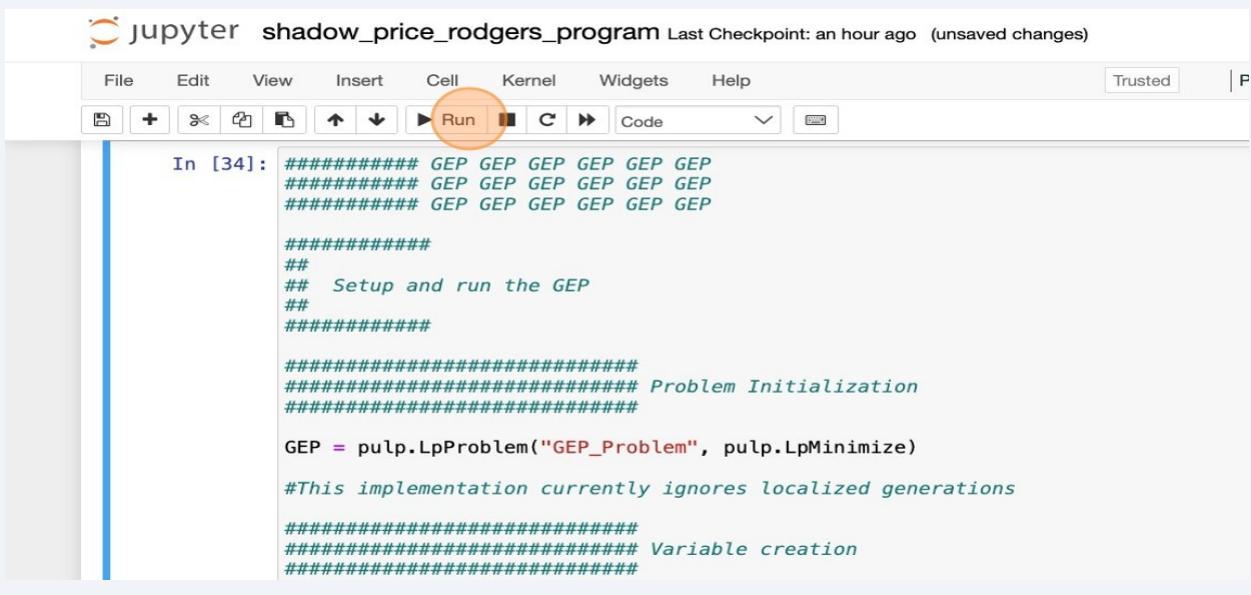
```
In [2]: ##### GEP GEP GEP GEP GEP GEP
#####
##
```

Step 4: Model Optimization Results

After running the optimization model, the key results are displayed directly in the output cell. These results include:

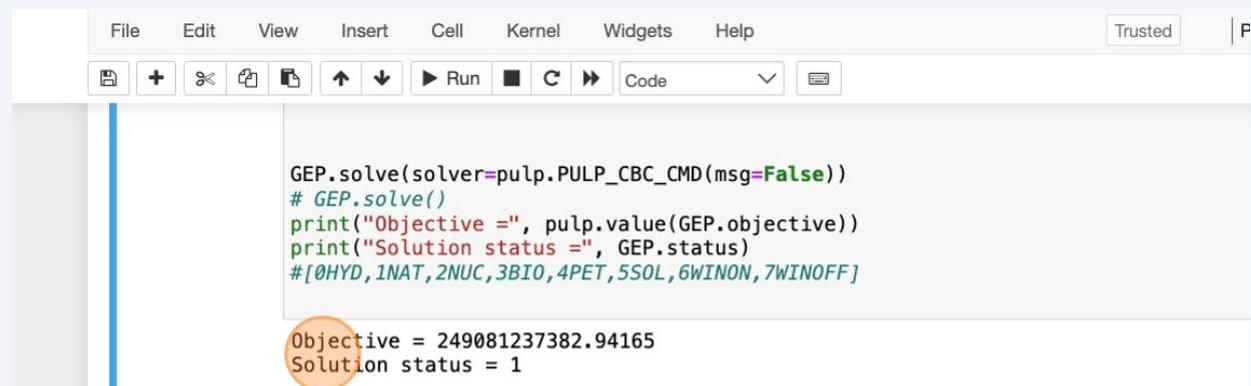
- **Objective Value:** The total cost or value the model aims to optimize.
- **Solution Status:** Indicates whether the optimization was successfully completed (1 = success). These outputs are critical for validating the model's execution and ensuring its feasibility.

5 Click "Run"



```
#----#
# GEP GEP GEP GEP GEP GEP GEP
#----#
# GEP GEP GEP GEP GEP GEP GEP
#----#
# GEP GEP GEP GEP GEP GEP GEP
#----#
## Setup and run the GEP
## Problem Initialization
## Variable creation
GEP = pulp.LpProblem("GEP_Problem", pulp.LpMinimize)
#This implementation currently ignores localized generations
#----#
#----#
```

6



```
GEP.solve(solver=pulp.PULP_CBC_CMD(msg=False))
# GEP.solve()
print("Objective =", pulp.value(GEP.objective))
print("Solution status =", GEP.status)
#[0HYD,1NAT,2NUC,3BIO,4PET,5SOL,6WINON,7WINOFF]
```

Objective = 249081237382.94165
Solution status = 1

Store and Plot Results

```
In [35]: # obtain results from solved GEP PuLP variables
x_mat = np.zeros((N,I))
for n in range(N):
    for i in range(I):
        x_mat[n][i] = x[n][i].varValue

w_mat = np.zeros((N,I))
```

Step 5: Visualization - Generation Timeline

This visualization provides an overview of the power generation timeline broken down by different energy sources.

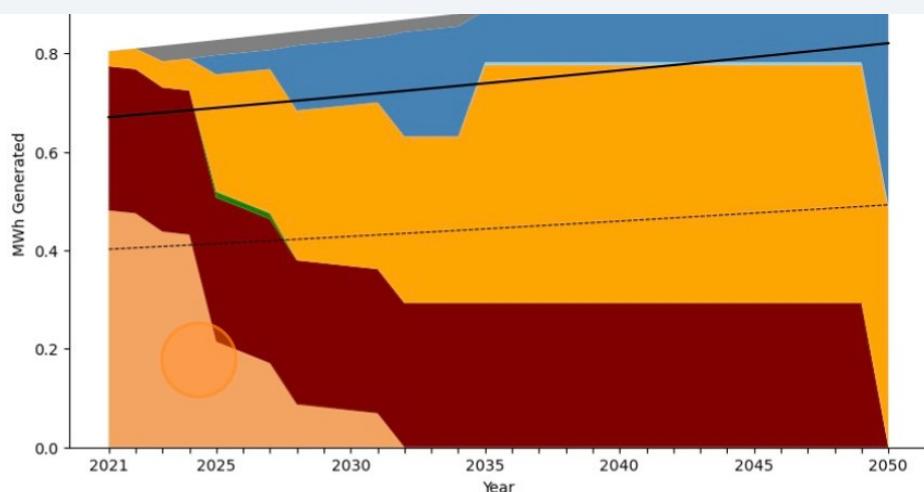
- The chart shows how each energy source contributes to the total generation over the years.
- It helps identify trends and assess the feasibility of energy policies.

7 Click "Store and Plot Results"

jupyter shadow_price_rodgers_program Last Checkpoint: an hour ago (unsaved changes)

```
File Edit View Insert Cell Kernel Widgets Help Trusted P  
Store and Plot Results 1  
In [35]: # obtain results from solved GEP PuLP variables  
  
x_mat = np.zeros((N,I))  
for n in range(N):  
    for i in range(I):  
        x_mat[n][i] = x[n][i].varValue  
  
w_mat = np.zeros((N,I))  
for n in range(N):  
    for i in range(I):  
        w_mat[n][i] = w[n][i].varValue  
  
y_mat = np.zeros((N))  
for n in range(N):  
    y_mat[n] = y[n].varValue  
  
#investment_mat = np.zeros((N,I+1))
```

8



```
In [37]:  
#####  
##  
## Plot the cumulative investments  
##  
#####
```

Step 6: Visualization - Cumulative Capacity

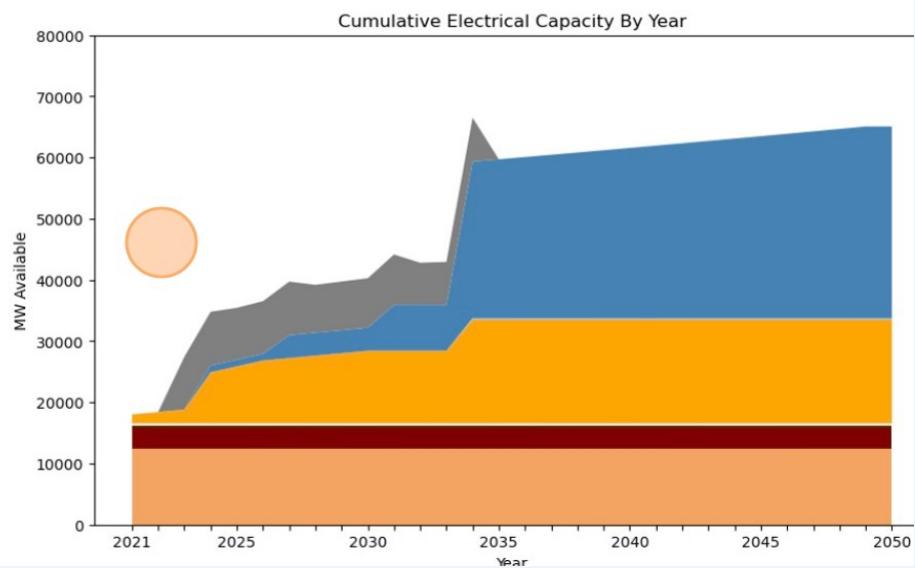
This chart displays the cumulative electrical capacity by year for different energy sources.

- It aggregates the installed capacity over time, providing insights into the overall growth of the energy system.
- Different colors represent various energy sources, offering a clear breakdown of contributions.

9

Click this image.

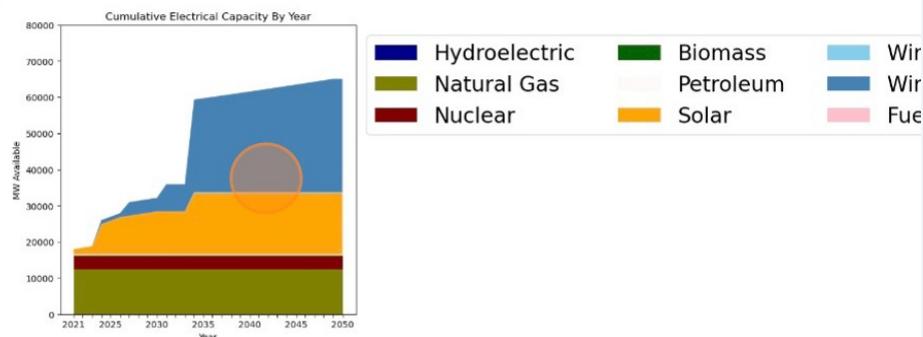
```
plt.show()
```



10

Click this image.

```
tech_list = ["Hydroelectric", "Natural Gas", "Nuclear", "Biomass", "Petroleum", "Solar"]
plt.legend(tech_list)
plt.legend(tech_list, loc='upper left', bbox_to_anchor=(1, 1), ncol = 3, prop =
plt.show()
```



```
In [40]: sum(dic['alpha'][n][i]*w[n][i].varValue for i in range(I) for n in range(N))
```

```
Out[40]: 130595512345.50388
```

```
In [41]: # Obtain and store proportion of demand recommended to be met by external purch
```

Step 7: Initial Dispatch Module

- The Initial Dispatch module initializes data and parameters for the energy dispatch model.
- This step prepares the required inputs for the GEP model's optimization.
- It ensures all data files are correctly loaded and pre-processed.

11 Click "2) Initial Dispatch"

The screenshot shows a Jupyter Notebook interface with the title bar 'jupyter shadow_price_rodgers_program Last Checkpoint: an hour ago (unsaved changes)'. The menu bar includes File, Edit, View, Insert, Cell, Kernel, Widgets, Help, and a Trusted button. Below the menu is a toolbar with various icons. A status bar at the bottom shows numerical values for capacity and other metrics. The main notebook area has a section header '2) Initial Dispatch' with a yellow circle icon. Below it, a code cell is labeled 'Initialize'. The code in the cell is:

```
In [19]: ##### dispatch dispatch dispatch
##### dispatch dispatch dispatch
##### dispatch dispatch dispatch

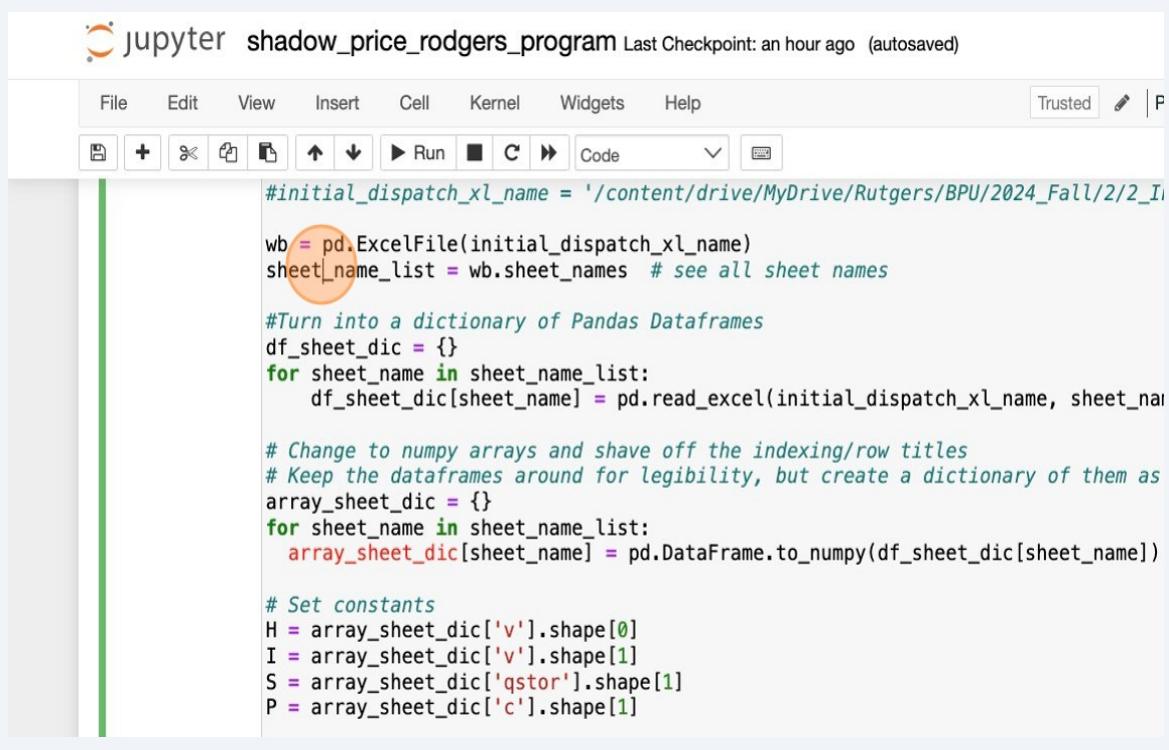
#####
## 
## Reset imports and remount
##
##### 
# drive.mount("/content/drive", force_remount=True)
```

Step 8: Loading Excel Sheets

The model reads Excel sheets containing energy dispatch and demand data.

- The sheet_name_list variable stores the names of all sheets in the workbook.
- These sheets are converted into a dictionary of pandas DataFrames for easier manipulation.

12 Click "sheet_name_list"



```
#initial_dispatch_xl_name = '/content/drive/MyDrive/Rutgers/BPU/2024_Fall/2/2_I.xlsx'
wb = pd.ExcelFile(initial_dispatch_xl_name)
sheet_name_list = wb.sheet_names # see all sheet names

#Turn into a dictionary of Pandas Dataframes
df_sheet_dic = {}
for sheet_name in sheet_name_list:
    df_sheet_dic[sheet_name] = pd.read_excel(initial_dispatch_xl_name, sheet_name)

# Change to numpy arrays and shave off the indexing/row titles
# Keep the dataframes around for legibility, but create a dictionary of them as arrays
array_sheet_dic = {}
for sheet_name in sheet_name_list:
    array_sheet_dic[sheet_name] = pd.DataFrame.to_numpy(df_sheet_dic[sheet_name])

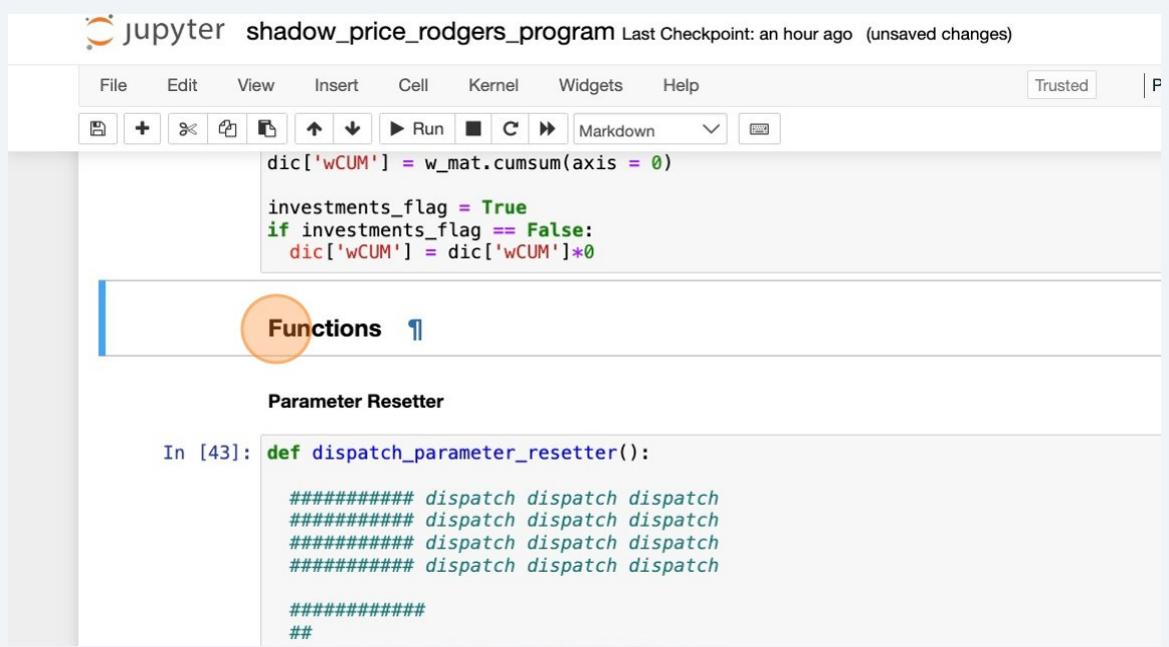
# Set constants
H = array_sheet_dic['v'].shape[0]
I = array_sheet_dic['v'].shape[1]
S = array_sheet_dic['qstor'].shape[1]
P = array_sheet_dic['c'].shape[1]
```

Step 9: Function Initialization

The notebook includes several pre-defined functions to streamline the execution of the GEP model.

- `dispatch_parameter_resetter`: Resets all model parameters to ensure consistency.
- Additional functions handle data pre-processing and result storage.
- These sheets are converted into a dictionary of pandas DataFrames for easier manipulation.

13 Click "Functions"



```
dic['wCUM'] = w_mat.cumsum(axis = 0)

investments_flag = True
if investments_flag == False:
    dic['wCUM'] = dic['wCUM']*0
```

Functions 

Parameter Resetter

```
In [43]: def dispatch_parameter_resetter():

    ##### dispatch
    ##### dispatch
    ##### dispatch
    ##### dispatch
    #####
    ##
```

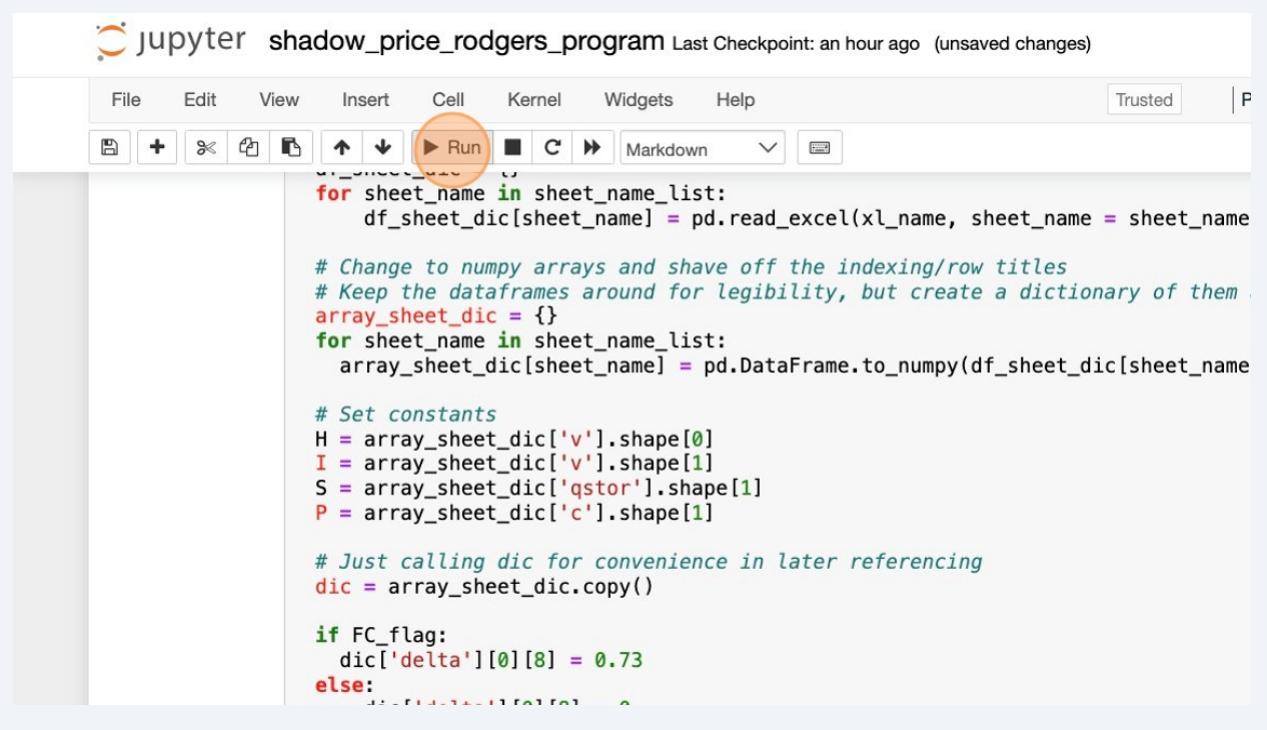
Step 10: Final Module Execution

This step involves executing the final module of the GEP model.

All functions and data have been prepared in the previous steps.

The final module integrates these components and runs the optimization model to completion.

14 Click "Run"



```
# Import required libraries
import pandas as pd
import numpy as np

# Read Excel file containing data
xl_name = 'path_to_your_file.xlsx'
sheet_name_list = ['Sheet1', 'Sheet2', 'Sheet3']

# Process each sheet
for sheet_name in sheet_name_list:
    df_sheet_dic[sheet_name] = pd.read_excel(xl_name, sheet_name = sheet_name)

# Change to numpy arrays and shave off the indexing/row titles
# Keep the dataframes around for legibility, but create a dictionary of them
array_sheet_dic = {}
for sheet_name in sheet_name_list:
    array_sheet_dic[sheet_name] = pd.DataFrame.to_numpy(df_sheet_dic[sheet_name])

# Set constants
H = array_sheet_dic['v'].shape[0]
I = array_sheet_dic['v'].shape[1]
S = array_sheet_dic['qstor'].shape[1]
P = array_sheet_dic['c'].shape[1]

# Just calling dic for convenience in later referencing
dic = array_sheet_dic.copy()

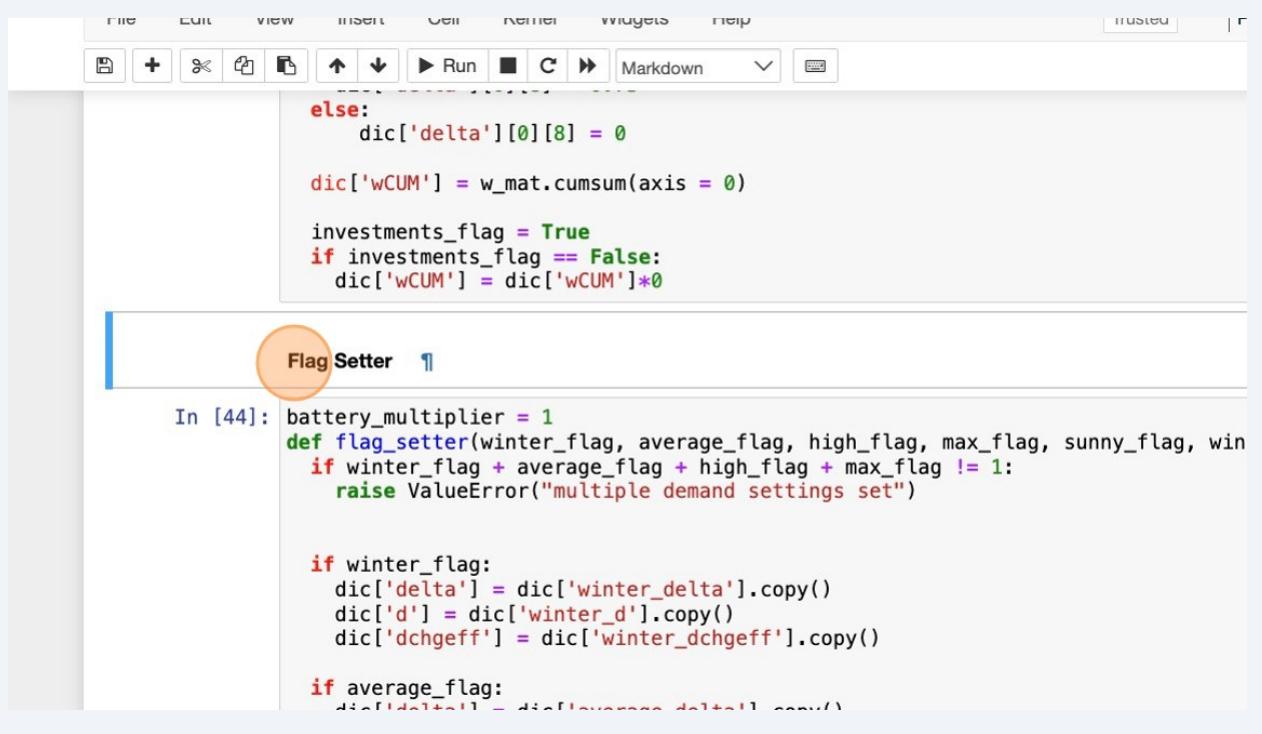
if FC_flag:
    dic['delta'][0][8] = 0.73
else:
    dic['delta'][0][8] = 0.5
```

Step 11: Flag Setter

This module defines flags to control different scenarios for the GEP model.

- Flags like winter_flag, sunny_flag, and windy_flag allow users to test specific conditions.
- These flags are used throughout the model to modify constraints or adjust calculations.

15 Click "Flag Setter"



The screenshot shows a Jupyter Notebook interface with a menu bar at the top. Below the menu is a toolbar with various icons. A code cell is open, and the title "Flag Setter" is highlighted with a yellow circle. The code in the cell is as follows:

```
else:  
    dic['delta'][0][8] = 0  
  
dic['wCUM'] = w_mat.cumsum(axis = 0)  
  
investments_flag = True  
if investments_flag == False:  
    dic['wCUM'] = dic['wCUM']*0
```

In [44]:

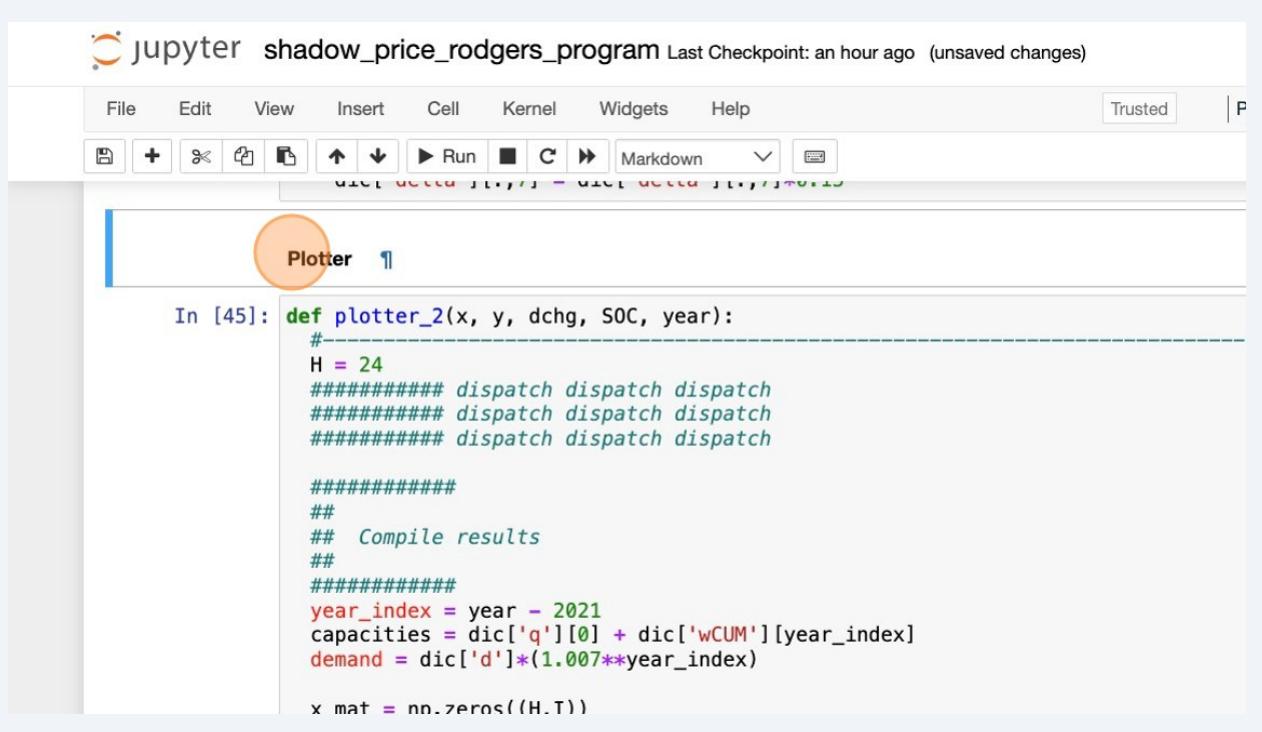
```
battery_multiplier = 1  
def flag_setter(winter_flag, average_flag, high_flag, max_flag, sunny_flag, win  
    if winter_flag + average_flag + high_flag + max_flag != 1:  
        raise ValueError("multiple demand settings set")  
  
    if winter_flag:  
        dic['delta'] = dic['winter_delta'].copy()  
        dic['d'] = dic['winter_d'].copy()  
        dic['dchgeff'] = dic['winter_dchgeff'].copy()  
  
    if average_flag:  
        dic['delta'] = dic['average_delta'].copy()
```

Step 12: Plotter Function

This function is used to generate visualizations of model outputs.

- It plots key metrics, such as dispatch timelines and cumulative capacity, based on the results.

16 Click "Plotter"



The screenshot shows a Jupyter Notebook interface with a menu bar at the top. Below the menu is a toolbar with various icons. A code cell is open, and the title "Plotter" is highlighted with a yellow circle. The code in the cell is as follows:

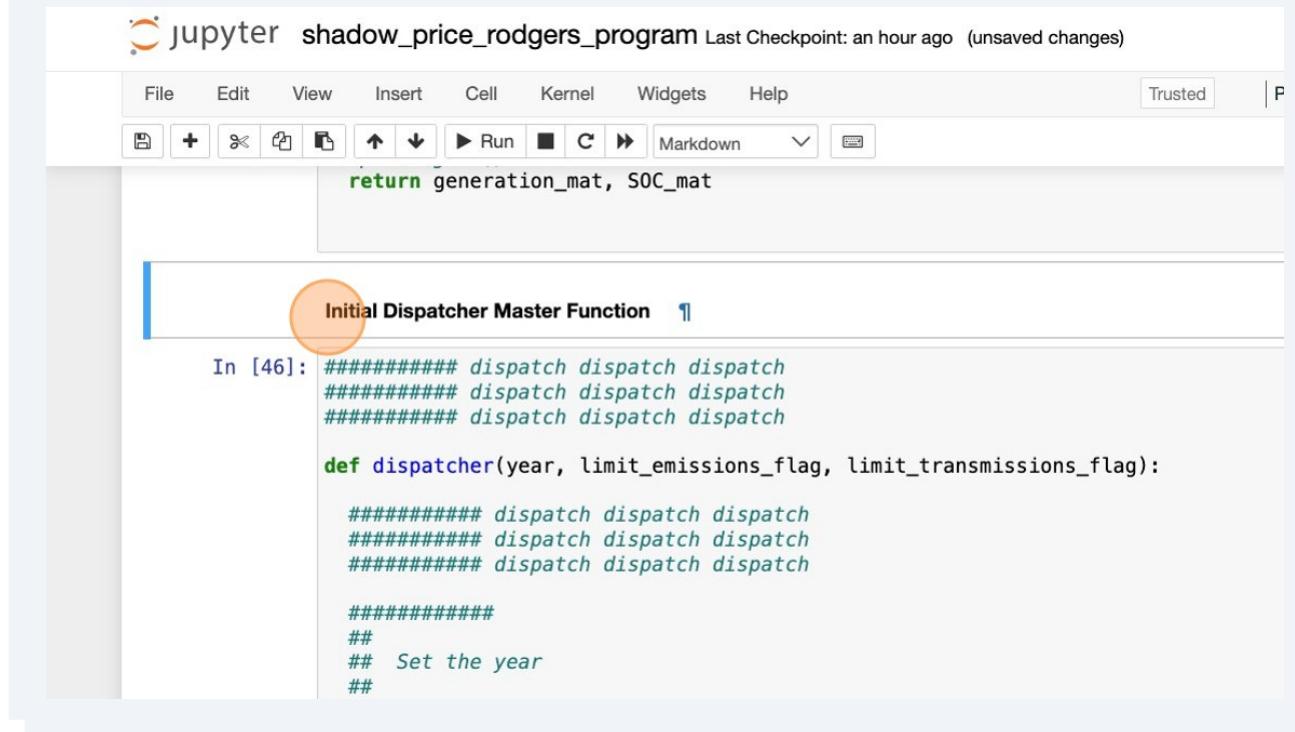
```
#  
H = 24  
##### dispatch dispatch dispatch  
##### dispatch dispatch dispatch  
##### dispatch dispatch dispatch  
  
#####  
##  
##  Compile results  
##  
#####  
year_index = year - 2021  
capacities = dic['q'][0] + dic['wCUM'][year_index]  
demand = dic['d']*(1.007**year_index)  
  
x_mat = np.zeros((H,T))
```

Step 13: Dispatcher Master Function

This function serves as the main controller for running the energy dispatch model.

- It initializes key parameters and calls sub-functions to compute results.
- The output includes optimized dispatch schedules and shadow prices.

17 Click "Initial Dispatcher Master Function"



The screenshot shows a Jupyter Notebook interface. The title bar reads "jupyter shadow_price_rodgers_program Last Checkpoint: an hour ago (unsaved changes)". The menu bar includes File, Edit, View, Insert, Cell, Kernel, Widgets, and Help. Below the menu is a toolbar with various icons. The code cell contains the following Python code:

```
return generation_mat, SOC_mat

Initial Dispatcher Master Function 11

In [46]: ##### dispatch dispatch dispatch
##### dispatch dispatch dispatch
##### dispatch dispatch dispatch

def dispatcher(year, limit_emissions_flag, limit_transmissions_flag):

    ##### dispatch dispatch dispatch
    ##### dispatch dispatch dispatch
    ##### dispatch dispatch dispatch

    #####
    ##
    ## Set the year
    ##
```

Step 14: Chosen Example Setup

In this step, the model allows you to set specific conditions and flags for testing scenarios.

- Examples of flags include:
 - winter_flag, sunny_flag: Used to simulate seasonal variations.
 - limit_emissions_flag: Applies constraints to limit emissions.
- The chosen example combines these flags to create a specific scenario for testing the model.

18 Click "Chosen example"

```
GEP.solve(solver = pulp.GLPK(msg=False))
solver_choice = 'GLPK'
else:
    GEP.solve()
solver_choice = 'CBC'

print("Objective =", pulp.value(GEP.objective))
print("Solution status =", GEP.status)

return x, y, dchg, SOC
```

Chosen example

```
In [47]: winter_flag = False
average_flag = False
high_flag = True
max_flag = False

sunny_flag = True
windy_flag = False

limit_emissions_flag = True
limit_transmissions_flag = False
#dispatch_parameter_resetter()
flag_setter(winter_flag, average_flag, high_flag, max_flag, sunny_flag, windy_f
```

Step 15: Dispatch with Removed Load

This module tests the model's response to load removal scenarios.

- It calculates the unmet demand vector, simulating cases where load shedding is necessary.
- The output includes dispatch schedules adjusted to meet these conditions while minimizing unmet demand.

19 Click "3) Dispatch with removed load"

```
File Edit View Insert Cell Kernel Widgets Help
Trusted P
Store unmet demand vector
In [32]: # get the unmet demand vector
unmet_demand = np.zeros((30,1))
#unmet_demand[:24] = generation_mat[:,11].reshape(24,1)
#total_unmet_demand = sum(unmet_demand)[0]
```

3) Dispatch with removed load

We are pretending there is no unmet demand, this file is to test shadow prices. But there was the question about how the 3rd dispatch is the one that incorporates an actual decision for load shedding. How do we represent shifting?

So I guess shedding is a decision variable in form of generation, but shifting would just be based on shadow pricing. I guess we'll see...

Removed Demand Plotter

Step 16: Removed Demand Plotter Definition

This function defines the process for visualizing dispatch schedules with removed demand.

- It computes the unmet demand and generates plots showing dispatch changes.
- The function ensures visual representation of how demand is met under specific constraints.

20 Click "Removed Demand Plotter"

The screenshot shows a Jupyter Notebook interface. The title bar says "jupyter shadow_price_rodgers_program Last Checkpoint: an hour ago (unsaved changes)". The menu bar includes File, Edit, View, Insert, Cell, Kernel, Widgets, Help, Trusted, and P. Below the menu is a toolbar with icons for file operations like Open, Save, Run, and Cell Type. A text cell contains a note: "So I guess shedding is a decision variable in form of generation, but shifting would just be based on shadow pricing. I guess we'll see...". A code cell is highlighted with a blue border and has a circled orange header "Removed Demand Plotter" with a small info icon. The code in the cell is:

```
In [49]: def removed_demand_plotter(x, y, dchg, SOC, year, unmet_demand):
    #
    H = 24
    ##### dispatch dispatch dispatch
    ##### dispatch dispatch dispatch
    ##### dispatch dispatch dispatch

    #####
    ##
    ##  Compile results
    ##
    #####
    year_index = year - 2021
    capacities = dic['n101 + dic['wCIM1[year_index]
```

Step 17: Results with Unmet Demand Removed

This step runs the model to analyze scenarios where unmet demand is removed.

- The output includes the optimized objective value and solution status.
- Additionally, a visualization of hourly dispatch results for the year 2035 is generated.

21

```
File Edit View Insert Cell Kernel Widgets Help Trusted P
print("Objective =", pulp.value(GEP.objective))
print("Solution status =", GEP.status)

return x, y, dchg, SOC, GEP
```

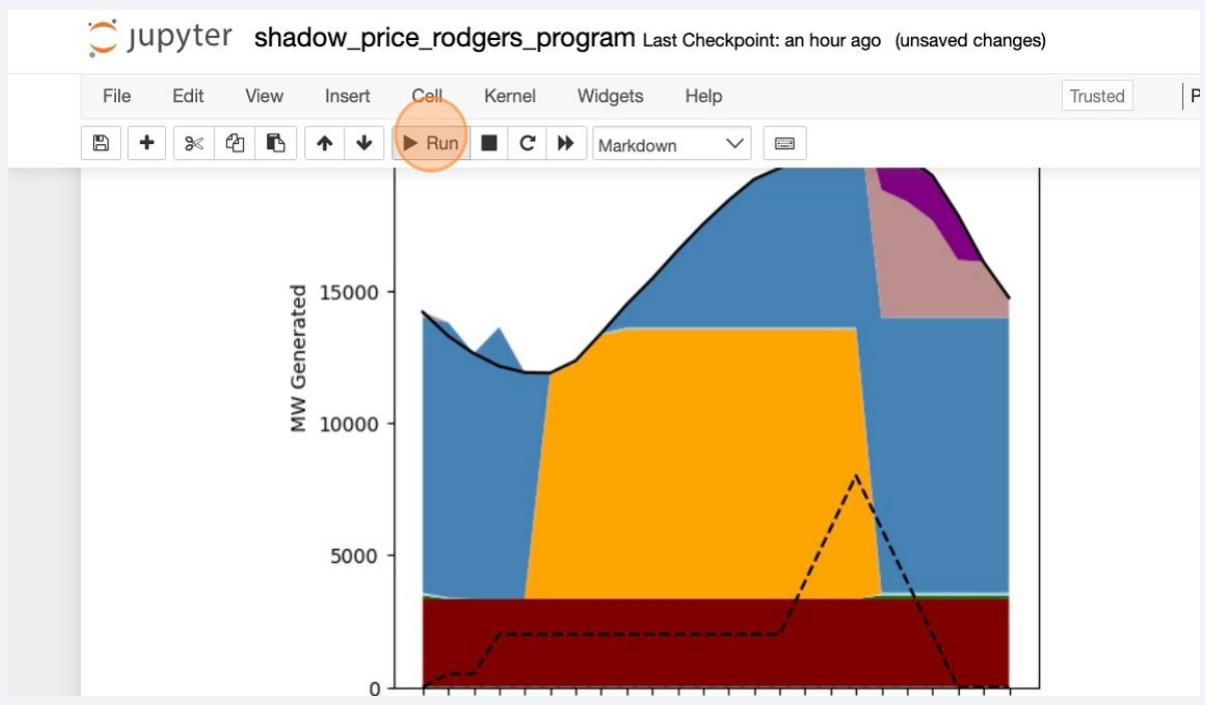
Run to show with unmet demand removed

In [51]:

```
x, y, dchg, SOC, model = removed_demand_dispatcher(year, limit_emissions_flag,
generation_mat, SOC_mat = removed_demand_plotter(x, y, dchg, SOC, year, unmet_d
Objective = 9206566.74048
Solution status = 1
```

Hourly Dispatch: Year = 2035

22 Click "Run"

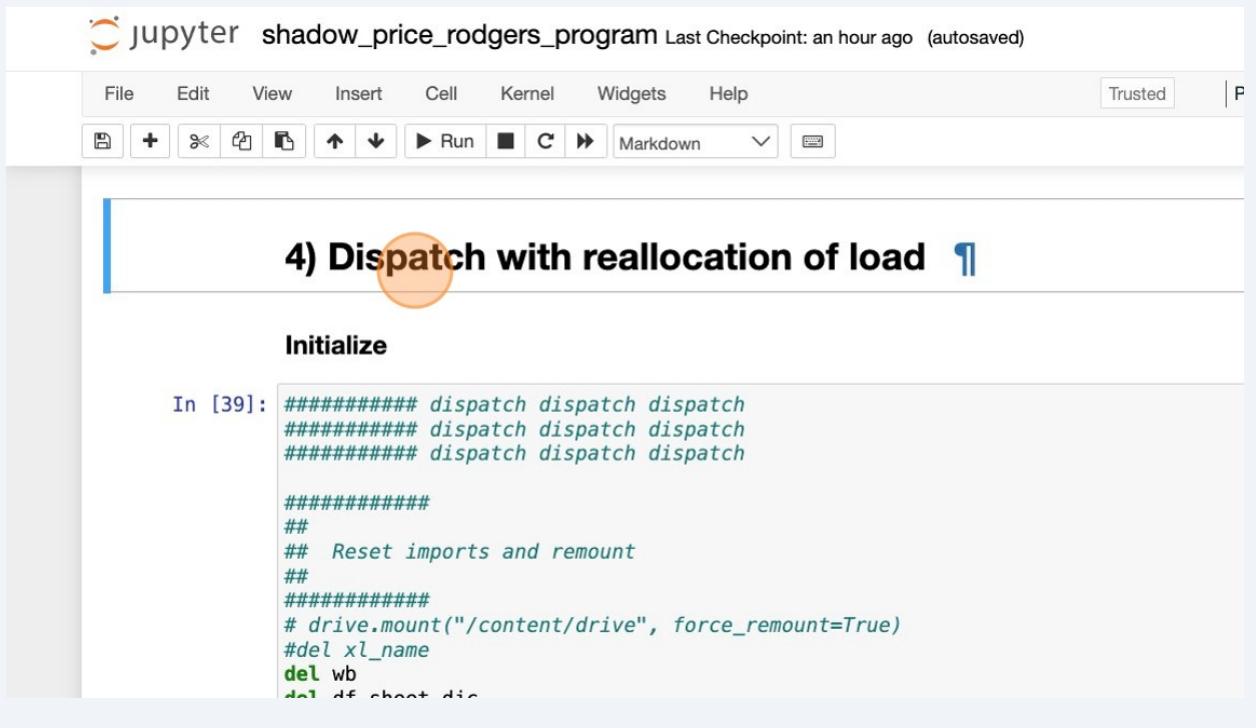


Step 18: Dispatch with Reallocation of Load

This module tests the reallocation of energy dispatch under shifted demand conditions.

- The reallocation process adjusts dispatch schedules to minimize unmet demand.
- This step prepares the model for further analysis and result generation.

- 23 Click "4) Dispatch with reallocation of load"



The screenshot shows a Jupyter Notebook interface with the title bar "jupyter shadow_price_rodgers_program Last Checkpoint: an hour ago (autosaved)". The menu bar includes File, Edit, View, Insert, Cell, Kernel, Widgets, and Help. Below the menu is a toolbar with icons for file operations like Open, Save, and Run. A cell in the notebook is highlighted with a blue border and has the title "4) Dispatch with reallocation of load". The cell content is titled "Initialize" and contains Python code:

```
In [39]: ##### dispatch dispatch dispatch
##### dispatch dispatch dispatch
##### dispatch dispatch dispatch

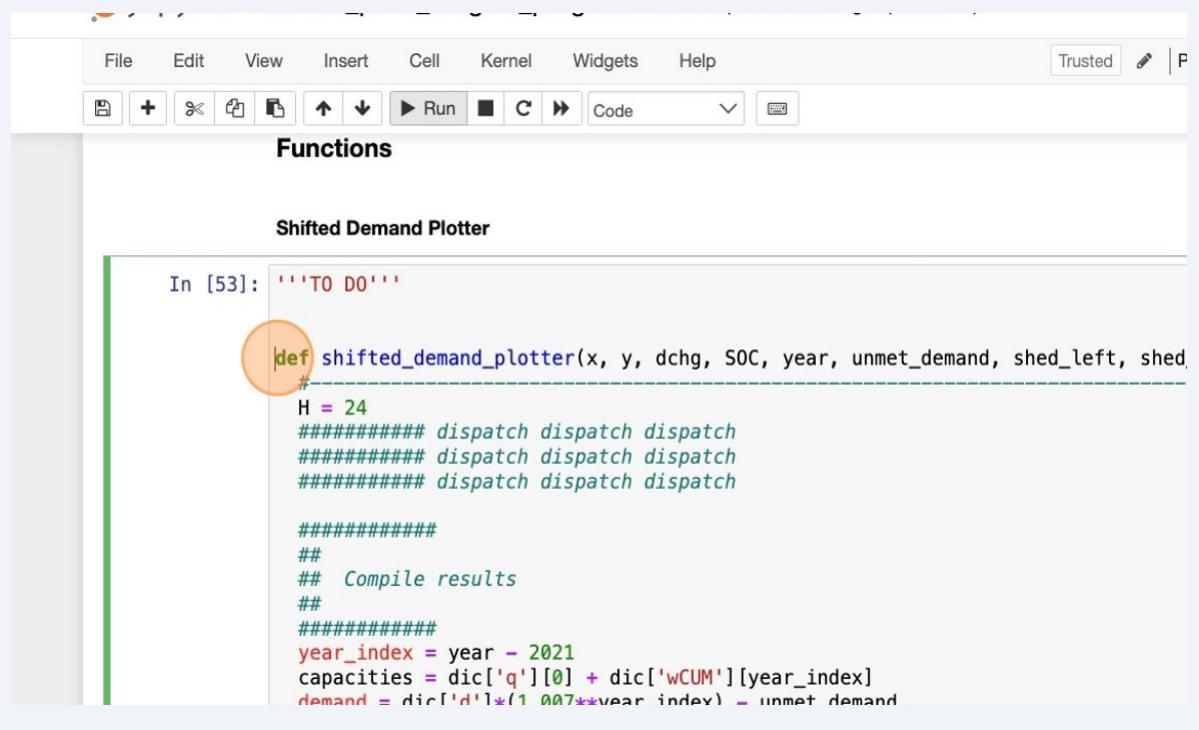
#####
##
## Reset imports and remount
##
#####
# drive.mount("/content/drive", force_remount=True)
#del xl_name
del wb
del df_sheet dic
```

Step 19: Shifted Demand Plotter Function

This function (shifted_demand_plotter) is designed to visualize the impact of shifted demand on energy dispatch.

- It takes key parameters such as unmet demand, shifted load, and system constraints as inputs.
- The output includes plots that represent adjusted dispatch schedules and other critical metrics.

24 Click "def"



```
In [53]: '''TO DO'''

def shifted_demand_plotter(x, y, dchg, SOC, year, unmet_demand, shed_left, shed_right):
    H = 24
    ##### dispatch dispatch dispatch
    ##### dispatch dispatch dispatch
    ##### dispatch dispatch dispatch

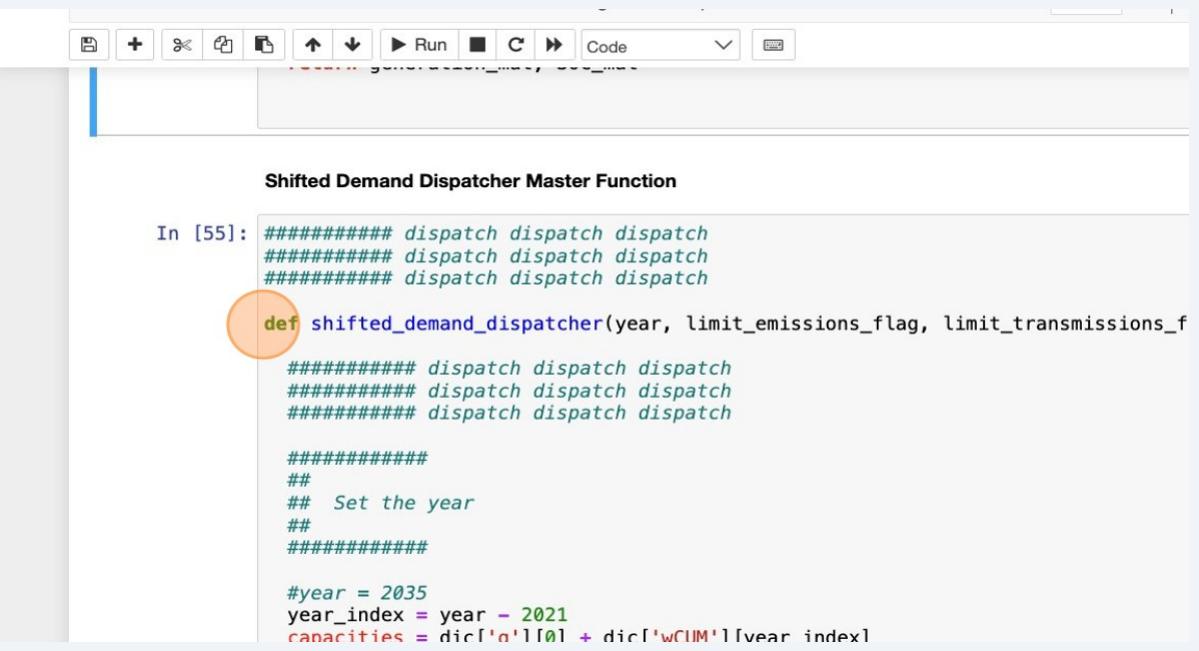
    #####
    ## Compile results
    ##
    #####
    year_index = year - 2021
    capacities = dic['q'][0] + dic['wCUM'][year_index]
    demand = dic['d']*(1.007**year_index) - unmet_demand
```

Step 20: Shifted Demand Dispatcher Function

This function (shifted_demand_dispatcher) is responsible for handling the reallocation of demand in the dispatch model.

- It modifies the dispatch schedules based on the shifted demand conditions provided.
- The function integrates key constraints such as emissions limits and transmission constraints.

25 Click "def"



```
In [55]: ##### dispatch dispatch dispatch
##### dispatch dispatch dispatch
##### dispatch dispatch dispatch

def shifted_demand_dispatcher(year, limit_emissions_flag, limit_transmissions_flag):
    ##### dispatch dispatch dispatch
    ##### dispatch dispatch dispatch
    ##### dispatch dispatch dispatch

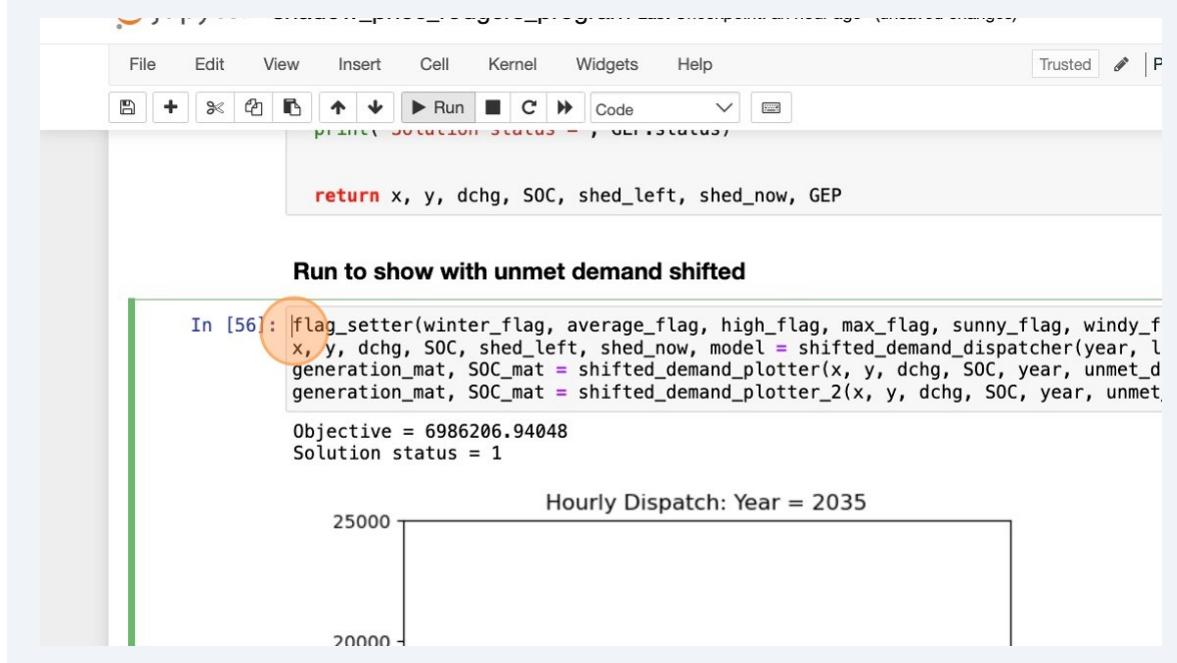
    #####
    ##
    ## Set the year
    ##
    #####
    #year = 2035
    year_index = year - 2021
    capacities = dic['n'][0] + dic['wCUM'][year_index]
```

Step 21: Results with Shifted Demand Reallocation

This step generates the final results for the reallocated load scenario.

- The output includes the optimized objective value and solution status.
- Additionally, a visualization of hourly dispatch under the shifted demand condition is provided.

26 Click "flag_setter"

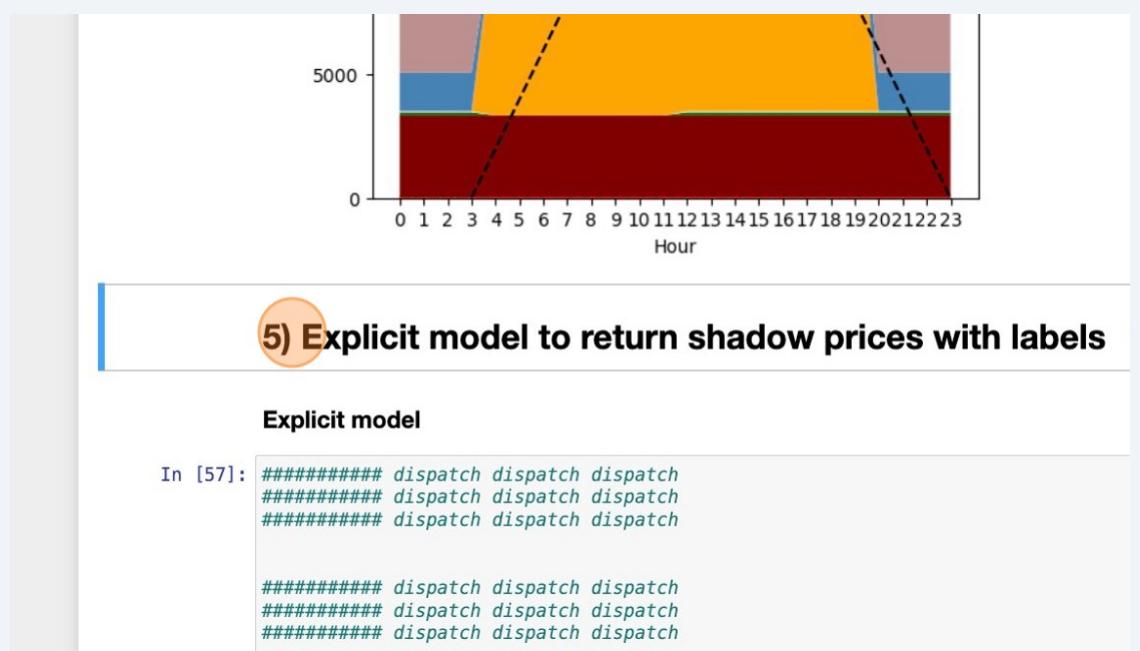


Step 22: Explicit Model for Shadow Prices

This module focuses on calculating shadow prices for various constraints in the dispatch model.

- Shadow prices help quantify the value of relaxing constraints in the optimization problem.
- The module outputs shadow prices labeled for easier interpretation and analysis.

27 Click "5) Explicit model to return shadow prices with labels"

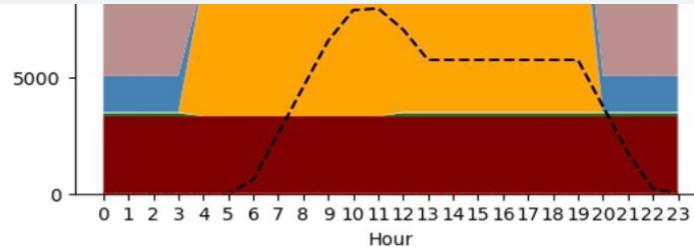


Step 23: Shadow Price Results

This step displays the computed shadow prices for constraints in the dispatch model.

- Each shadow price represents the marginal value of relaxing a specific constraint.
- These values are critical for understanding bottlenecks in the optimization process.

28 Click "run"



```
In [58]: for constraint in dispatch_model.constraints.values():
    print(f"{constraint.name}: Shadow price = {constraint.pi}")

balance_0: Shadow price = 100.0
balance_1: Shadow price = 100.0
balance_2: Shadow price = 100.0
balance_3: Shadow price = 70.0
balance_4: Shadow price = 0.0
balance_5: Shadow price = 0.0
balance_6: Shadow price = 0.0
balance_7: Shadow price = 0.0
balance_8: Shadow price = 0.0
balance_9: Shadow price = 0.0
balance_10: Shadow price = 0.0
balance_11: Shadow price = 0.0
```

Step 24: Calculation of Shadow Prices Over Time

This step involves creating a time-based array to compute shadow prices associated with balance constraints at each hour.

- `time_array` is used to index through 24 hours.
- `balance_shadow_prices` captures the shadow price of maintaining balance in the power grid for each hour.

29 Click "time_array"

```
1,2) + 5.06*x_{11,3} + 4.92*x_{11,4} + 0.62*x_{11,8} + 1.46*x_{12,0} + 1.96*x_{12,1}
+ 2.48*x_{12,2} + 5.06*x_{12,3} + 4.92*x_{12,4} + 0.62*x_{12,8} + 1.46*x_{13,0} + 1.9
2.48*x_{13,2} + 5.06*x_{13,3} + 4.92*x_{13,4} + 0.62*x_{13,8} + 1.46*x_{14,0} +
1) + 2.48*x_{14,2} + 5.06*x_{14,3} + 4.92*x_{14,4} + 0.62*x_{14,8} + 1.46*x_{15
_{15,1}} + 2.48*x_{15,2} + 5.06*x_{15,3} + 4.92*x_{15,4} + 0.62*x_{15,8} + 1.46*
1.96*x_{16,1} + 2.48*x_{16,2} + 5.06*x_{16,3} + 4.92*x_{16,4} + 0.62*x_{16,8} +
0) + 1.96*x_{17,1} + 2.48*x_{17,2} + 5.06*x_{17,3} + 4.92*x_{17,4} + 0.62*x_{17
_{18,0}} + 1.96*x_{18,1} + 2.48*x_{18,2} + 5.06*x_{18,3} + 4.92*x_{18,4} + 0.62*
1.46*x_{19,0} + 1.96*x_{19,1} + 2.48*x_{19,2} + 5.06*x_{19,3} + 4.92*x_{19,4} +
0) + 1.96*x_{20,0} + 1.96*x_{20,1} + 2.48*x_{20,2} + 5.06*x_{20,3} + 4.92*x_{20,4} +
0) + 1.96*x_{21,0} + 1.96*x_{21,1} + 2.48*x_{21,2} + 5.06*x_{21,3} + 4.92*x_{21,4} +
0) + 1.96*x_{22,0} + 1.96*x_{22,1} + 2.48*x_{22,2} + 5.06*x_{22,3} + 4.92*x_{22,4} +
0) + 1.96*x_{23,0} + 1.96*x_{23,1} + 2.48*x_{23,2} + 5.06*x_{23,3} + 4.92*x_{23,4} +
0)
```

```
In [61]: #for constraint in dispatch_model.constraints.values():
#    print(f"{constraint.name}: Shadow price = {constraint.pi}")

time_array = np.array(range(0,24))
balance_shadow_prices = np.zeros(24)
for time in time_array:
    balance_shadow_prices[time] = dispatch_model.constraints[f'balance_{time}'].p
dfer(balance_shadow_prices)
```

```
Out[61]:
0
0 100.000
1 100.000
2 100.000
```