A

Code listings and file snippets

This appendix elaborates on the code per section. For traceability and clearness, the code was subdivided into several parts, and each parts resulted in certain data-files that are used in subsequent parts. First the Powerfactory data extraction scripts and files are described, whereafter scripting of the flexibility reaction model and calculation of ultimate percentages is elaborated on.

A.1. Coding workflow for 5.2 on congestion signals and financial incentives

For the creation of the network and creation of congestion signals, python files 1a-e were created, next to the files in the 'Hierarchical network' python directory. Besides, python scripting was used to extract the right data with regard to network elements and distribution-substation 'agents' with their loads from the power flow modeling tool in the previous step. The following steps were performed:

- First, the network was recreated with the Networkx library, through extraction of nodes and edges from powerfactory. Therefore, the specific busbars from the distribution substations that represent the node 'agents' needed to be selected from powerfactory. This selection required specific search strings to select the right busbar elements in powerfactory, to separate the busbars in the network from the other elements. To this end, a list of regex patterns was used for string matching as visible in the 'Hierarchical network directory' in the 'network elements creation.py' file. These regex patterns are also visible in the snippets below. A likewise approach was used for the edges. However some edges that exist for the N-1 criterion and connect the different distribution areas had to be manually deleted, as powerfactory cannot filter between closed or open switches. To identify these closed switches that are not open during normal operations a replica network could already be constructed in networkx in which loops were identified as in the 'Loop identification.py' file in the 'Hierarchical network directory'. Based on this, the wrongly generated edges could be deleted. Ultimately, this led to files with all edges and nodes in the network, visible in the 'edges updated' and 'manual nodes updated' datafiles. The resulting list of edges and nodes were combined in the '1b PF Hierarchical network.py' file, into the final radial network representing all possible congestion contributors in a Kamada-Kawai layout, which is shown in the snippets.
- Secondly, a chain of congestion contributors per element was made, based on a list in which the dependencies on congested elements for each of the node 'agents' are listed. These lists with chains per node were constructured in '1e congestionchain.py', and led to the resulting file 'final_congestion_components.csv' in which the parents for each node were listed.
- Thirdly, now that the congestion contributors structure is known, the congestion signals per node can be calculated. The agents below the congested component, can receive the congestion signal from the most limiting congested component. It implies that a node can either receive congestion from its own value or from several, higher-level elements that are more congested. The signal will then correspond with the highest congested nodes in it's dependency. This struc-

ture was identified, and the according overloading percentages were filled in. A file was created in which each element has congestion values of its own in the 'node congestion vertical filled' datafile. This was done in the '1c congestion import' scripting file. Then, with the corresponding congestion values from the '1c congestion import' scripting file, the highest congestion value from superior nodes was calculated per node in the '1d Bus congestion' scripting file. This resulted in the 'max per parent rounded' file. Here, actual congestion loading percentages per node per timestep are visible, which is the resulting output of this part. It is visible that some nodes have exactly the same loading percentage. This corresponds with these having the same 'parents' that receive a high congestion value, therefore the 'kids' also have this congestion percentage. This congestion percentage per node per timestep is impacting the congestion signal, which is impacts the incentive structure and therewith the flexibility reaction of the agent. An example of this file is shown in the snippets below.

To summarize, next to usage of the load profiles, the main outputs of extracted from the the power flow model with python scripts are the following datafiles:

- **The edges and node data** These are in the datafile 'edges updated.py' en 'manual nodes updated.py'. These were used to create the replica network model.
- **Loading per node per timestep as congestion signal**. Here, each node has the a loading percentage per timestep. The loading percentage is the highest percentage among its own loading or loadings above it, as it is hierarchically dependent on the above nodes and components. This is included in the datafile 'max per parent rounded.py'.

A.1.1. Snippets for section 5.2 on congestion signals

The snippet below, from the 'network elements creation.py' file in the 'Hierarchical network directory' shows the search strings needed to find the right elements from the powerfactory simulation data elements to include nodes in the replica network.

```
big_busbar_patterns = [
BLURRED
]
big_busbars = [
    name for name in names
    if any(re.match(pattern, name) for pattern in big_busbar_patterns)
]
big_busbars = sorted(set(big_busbars))
```

The same was done for the edges, after which a file existed for nodes and edges, from which the graph was made with the following code

```
# --- Build undirected graph ---
G = nx.Graph()

# Add nodes with type info
for node, t in node_types.items():
    G.add_node(node, node_type=t)

# Track invalid edges
invalid_edges = []

for _, row in edges_df.iterrows():
    src, tgt = row["Source"], row["Target"]

    if src in G.nodes and tgt in G.nodes:
        G.add_edge(src, tgt, edge_name=row["Edge_Name"], edge_type=row["Edge_Type"])
    else:
        invalid_edges.append((src, tgt, row["Edge_Name"], row["Edge_Type"]))

# --- Find isolated nodes (no edges) ---
isolated_nodes = list(nx.isolates(G))

# --- Layout (Kamada-Kawai) ---
pos = nx.kamada_kawai_layout(G, scale=10)

# Node color mapping
```

```
color_map = {
    "big_busbar": "orange",
    "eln": "lightblue",
}

node_colors = []
for n, d in G.nodes(data=True):
    node_colors.append(color_map.get(d.get("node_type", "unknown"), "gray"))
```

The file output below shows the example output of the '1e congestionchain.py' output in a sample of the 'final_congestion_components.csv' file output. In here, the differences in dependencies on congestion between nodes early and later in the hierarchical chain are clearly visible

```
Node,Full_Congestion_Path
BLURRED
and so forth
```

The file output below shows the example output of 'max_per_parent_rounded.csv' in which all the agents have a specific congestion signal per timestep, which corresponds with the highest loading of the elements above it, hence to which the loading of the agents contributes. As visible, some agents have the same loading. This means that they are both contributors to the same the above lying congested component

```
Parent;2024.06.01 00.00.00;2024.06.01 01.00.00;2024.06.01 02.00.00;2024.06.01
    03.00.00;2024.06.01
8400;18.94;18.3;17.84;18.01;18.71;19.32
6162;18.94;18.3;17.84;18.01;18.71;19.32
3095;12.39;11.74;11.61;11.59;12.93;15.07
```

A.2. Coding workflow for 5.4 on spatial correlation and clustering

Several steps were needed to account for spatial correlation in the simulated flexibility reactions. After the development of the probabilistic envelopes, specific flexibility reactions per agent per timestep were drawn. These reactions were coupled to correspond with the agents in the network. Then, the direct neighbors of each agent were identified, for only the specific one-directional neighbor was selected. From all the agents, a specific list was made, so that the interactions could be performed in the one-directional order starting at the agents at ends of the feeders and propagating upwards. Then, the interactions were performed, and the flexibility values per agent per timestep were updated chronologically. For this question, files from the python code directory were used, being the '2 Envelopes.py' file from step 6 and onward. The code does the following things:

- In step 6, it imports the edges from the network graph and assigns the agents to these edges while marking them as neighbors. A mapping from agents and neighbors is created
- In step 7, it retrieves the initial flexibility reactions from the envelopes.
- In step 8, the specific order of the agent and their interactions was made in a list with sequential neighbors. In this way, it was ensured that adjacent agents were interacting chronologically so that spatial clustering could propagate hierarchically. This specific code block is included in the snippet below, in which the file pair reorder list is shown through a function.
- In step 9 the new flexibility reactions were added to the electricity profiles per agent per timestep in csv files. However, first, the normalized weighted sum was performed in which random weights were used, as also shown in the snippet below.

A.2.1. Snippets for section 5.4 on spatial clustering

The snippet below, from step 8 of the '2 Envelopes.py' file, reorders the file pairs in a list with an order needed to ensure that adjacent agents were interacting chronologically. Herewith, spatial clustering could propagate hierarchically, from the feeder ends to the beginning

```
def extract_numbers(path):
    """Extract all numeric parts from a filename."""
    base = os.path.basename(path)
    return re.findall(r'\d+', base)
```

```

def has_overlap(pair1, pair2):
    """Check if two tuples share any numeric ID."""
    nums1 = set(sum([extract_numbers(p) for p in pair1], []))
    nums2 = set(sum([extract_numbers(p) for p in pair2], []))
    return not nums1.isdisjoint(nums2)

def reorder_file_pairs(file_pairs):
    """Reorder file_pairs so similar tuples are adjacent."""
    ordered = []
    remaining = file_pairs.copy()

    while remaining:
        current = remaining.pop(0)
        ordered.append(current)

        # Find next tuple that shares something with the current one
        for i, candidate in enumerate(remaining):
            if has_overlap(current, candidate):
                # Move that tuple right after the current one
                next_pair = remaining.pop(i)
                remaining.insert(0, next_pair)
                break  # continue chaining similarity

    return ordered
```

The snippet below, from step 9 of the '2 Envelopes.py' file, shows the normalized weighted sum used to assign the spatial correlation between direct neighbouring agents, based on a random weight. Herewith, spatial clustering could occur randomly between neighbours, and subsequently propagate and distribute through the network with random probabilities per agent-neighbor pair.

```
        # Random weight
        w = round(random.random() **0.5, 2) # varying this in scenarios

        # Weighted sum
        weighted_sum = agent_col * w + neighbor_col * (1 - w)
```

A.3. Coding workflow for 6.1 on factorial envelope and state development

The development of the ultimate states as a results of the in-between states that are the result of multiplicative factorial reductions with envelopes is was implemented via the hierarchical logic. The code to develop the final states is elaborated on below per factor.

To this end, a double for-loop was used, that performed the envelope development for each agent and for each timestep. Most of the information could be represented directly in the code, except for the congestion signals. These were extracted as elaborated on in section 5.2 and the corresponding code. Once applicable under congestion, for each agent, transactive congestion signal value from the loading/congestion file was imported per timestep, to use in the incorporation of the 'incentive structure' factor later on.

In the end, for each timestep the final flexibility state was calculated per agent, as a result of the multiplicative reduction of the inbetween factorial states. The factorial envelopes and states are elaborated on individually below. However first, the initialized values that remain constant over time are elaborated on. The values can be found in table 7.1 in the text, and the actual code snippet is provided below.

1. **Initializations** Before the for-loop for the timestep begins, the initialized values are initialized already per agent as they are in the agent for-loop. Specifically, the initial beginning state from 'physical capacity' factor is initialized as asset_capacity. Besides, the duck curve multipliers and daily pattern, that influences the opportunity cost in the 'incentive structure' are initialized here. Furthermore, the R3 that largely determines the influence of the capability factor, is determined here, as it does not vary over time.
2. **Physical Capacity** First, physical capacity is created. It calculates the remaining physical capacity state, after subtraction of the historical usage that impacts the current timestep by 'flex_memory'.

This historical usage incorporates two envelope values, R11 and R12, as apparent in the code below. Besides, it subtracts external influences of the remaining asset capacity, with the envelope R1. This leads to remaining physical capacity.

3. **Capability** Then, only if there is congestion, a reaction actually occurs. There is only a flexibility reaction if there if there is incentive hence if there is congestion at loading above 80%. So, the transactive congestion signal per agent per timestep is checked in an if-statement, and only once true the rest of the development continues, otherwise a value of 0, meaning no flexibility, is outputted at the end. Within the continuous development in the if-statement, the outcome of state physical capacity is then diminished through subtraction of external influences influencing the capability, which is the value R2 that is chosen once per agent in initialization. However a small part varies randomly over time, R21.
4. **Incentive structure** Subsequently, the incentivation with the granular CMM occurs, resulting in incentivized and capable physical capacity. Actually, this is where, the transactive signal comes into play, but since the uniform price already applies once there is congestion, no detailed information from the transactive value is implemented here. The state is adjusted for the influence of the magnitude of the incentive, and the effect of the incentive on the opportunity cost. This is R3, and varies in scenarios the two scenarios. Besides, the daily varying opportunity costs also impact the result of incentive, which is captured through the influence of R21. Both R2 and R21 reduce the previous state and develop into the new, third factorial state.
5. **Behavior** In the end, the third state is influenced by behavior and develops into the final fourth state. The effect of external influences is accounted for and reduces stat by the envelope R4.

These steps result in the final flexibility state per agent per timestep. Eventually, these are converted into dataframes as csv files in which for each agent at each timestep the original profile is stated, the percentual flexibility reaction and the congestion value, as elaborated on in 5.3 and the appendix on 5.3 Step 5 of this code saves and creates these files.

A.3.1. Snippets for section 6.4 on factorial envelope development

The snippet below, from step 1 to 5 of the '2 Envelopes.py' file, shows the code that was the result of the logic provided above.

```

for all agents:
    merged_rows = []
    prev_reaction = [0,0]  # so take two rounds before in account
    duck_curve_multipliers = [1.067, 1.080, 1.100, 1.120, 1.133, 1.167, 1.200, 1.167, 1.067,
        0.900, 0.800, 0.733, 0.700, 0.700, 0.733, 0.833, 0.967, 1.133, 1.233, 1.300, 1.267,
        1.167, 1.100, 1.053]
    Asset_capacity = round(0.1 + random.betavariate(2, 5) * (0.5 - 0.1), 2)  # zodat eenmalig
         geïnitieerd en niet elke tijdstap
    R3 = round(random.uniform(0, 0.8), 2)  # information on capability here?

    for i, (timestep, original_value) in enumerate(original_profile_rows):
        hour_index = i % 24
        R41 = duck_curve_multipliers[hour_index]

        if i < len(congestion_percentages):
            cp = congestion_percentages[i]
            if cp == '' or pd.isna(cp):
                congestion_percentage = 0
            else:
                congestion_percentage = float(cp)
            congestion_percentage = int(congestion_percentage)
        else:
            congestion_percentage = 0  # fallback if fewer congestion values than timesteps

        # HERE IS THE CALCULATION OF THE ACTUAL FLEXIBILITY REACTION
        R11 = round(0.3 + random.betavariate(5, 2) * (0.8 - 0.3), 2)  # previous round
        R12 = round(random.betavariate(5, 2) * 0.3, 2)  # round before that one
        Flex_memory = prev_reaction[0] * R11 + prev_reaction[1] * R12  # takes two rounds
            before into account
        inter_asset_cap = Asset_capacity - Flex_memory
        R2 = round(random.betavariate(2, 6) * 0.4, 2)
```

```
        Externalities_capacity_loss = inter_asset_cap * R2
        Physical_capacity = Asset_capacity - Flex_memory - Externalities_capacity_loss

        if congestion_percentage > 80:
            R31 = round(random.uniform(0.95, 1.05), 2)
            Externalities_capability_loss = Physical_capacity * R3 * R31
            Capable_physical_capacity = Physical_capacity - Externalities_capability_loss

            R4 = round(random.betavariate(1.5, 3) * 0.7, 2)  # CMM uncertainty decision
            Incentivized_capable_physical_capacity = Capable_physical_capacity * R4 * R41

            R5 = round(random.betavariate(1.5, 3) * 0.4, 2)
            Externalities_behavior_loss = Incentivized_capable_physical_capacity * R5
            Incentivized_behavior_capable_physical_capacity =
                Incentivized_capable_physical_capacity - Externalities_behavior_loss

            output_congestion = congestion_percentage
        else:
            Incentivized_behavior_capable_physical_capacity = 0
            output_congestion = 0

        prev_reaction = [Incentivized_behavior_capable_physical_capacity, prev_reaction[0]]
        merged_rows.append([timestep, original_value,
            Incentivized_behavior_capable_physical_capacity, output_congestion])
```

A.4. Coding workflow for 7.2 on obtaining flexibility reaction states

In scripting file '2 Envelopes.py' the actual flexibility reactions as described above are calculated, after the implementation and development in 6.1. In this file, several steps were performed, being the following:

- First, in step 1,2 and 3 each node 'agent' got assigned its reference profile load in a the 'loads_combined.csv' datafile, based on the earlier extracted information file that includes a mapping of agents and with their corresponding loads from the load flow. These original profiles with corresponding timesteps were prepared to create new files with the adapted loads, after the implementation of the flexibility from the factorial envelopes. The output of this were the first parts of the csv-files per agent in which the timesteps and the original load data was listed.
- Secondly, in step 4 of the code, actual flexibility envelopes with probabilities and ranges were prepared to later on draw a value that could be added to the electricity load profile files of the agent, per timestep. The specific values for the probabilities and ranges of the envelopes is elaborated on in chapter 6 on the model implementation. The code used for the envelope development is shown in the snippet below.
- Thirdly, the relative flexibility values that arise from the envelope are adjusted. Since the flexibility output from the envelopes is a relative value per timestep, while the load values are absolute values, the flexibility values are also translated into active values. This is done in the file '3 Results processing.py'. To scale the flexibility relatively to the load, the relative flexibility value is multiplied with a weight factor. This weight factor is the maximum load in the year divided by the load at the current timestep. The snippet of this formula is provided in the snippet below.

A.4.1. Snippets for section 7.2 on flexibility visualization and required vs provided flexibility reactions

```
ref sum_reqflex_minus_actflex(path: str) -> float:
    """
    Read CSV and return the total amount by which reqflex (second-to-last column)
    exceeds actflex (last column), summed across all rows.
    """
    df = pd.read_csv(path)
    if df.shape[1] < 2:
        return 0.0

    actflex_col = df.columns[-1]      # last column
    reqflex_col = df.columns[-2]      # second-to-last column

```

```
    actflex = pd.to_numeric(df[actflex_col], errors="coerce")
    reqflex = pd.to_numeric(df[reqflex_col], errors="coerce")

    # Calculate positive differences only
    differences = (reqflex - actflex).clip(lower=0)

    return float(differences.sum())
```

Snippets for section 7.2 on flexibility envelope reactions

The snippet below from the file '3 Results processing.py' shows the formula used, in the last line, to adapt the relative flexibility reactions from the envelope towards scaled relative values towards the maximum load from the original profile from the agent. In this way the values can be compared with required flex, but only if congestion is above 80%.

```
# Reference maximum
    ref_max = df['OriginalProfile'].max()

    # --- reqflex ---
    df['reqflex'] = 0.0
    df.loc[df['CongestionPercentage'] > 80, 'reqflex'] = (
        (df['CongestionPercentage'] - 80) / 100
    )

    # --- actflex ---
    if ref_max == 0:
        df['actflex'] = 0.0
    else:
        df['actflex'] = (ref_max / df['OriginalProfile']) * df[target_col]
```