```python
import pandas as pd

def get_data_csv():

  """Reads data from a CSV file and returns it as a pandas Dataframe."""

  data = pd.read_csv("data.csv")
  return data
```

```python
def get_departure_times(truck_availability):
    """
    Returns a list of departure times based on truck availability data. When the truck is there (1)
    and then not there (0), it indicates a departure.
    """
    departures = []
    for i in range(1, len(truck_availability)):
        if truck_availability.iloc[i - 1] == 1 and truck_availability.iloc[i] == 0:
            departures.append(i - 1)

    # handle last period if truck still connected
    if truck_availability.iloc[-1] == 1:
        departures.append(len(truck_availability) - 1)

    return departures
```

```python
from load_csv import get_data_csv
from utils import get_departure_times
from build_model import build_model, get_default_parameters
from analysis import experiment_solar_capacity, experiment_grid_capacity, experiment_truck_charge_power, experiment_soc_target


def optimize_session(session_data):
    """
    The function builds and optimizes the EMS model for a single charging session.
    """
    model, info = build_model(session_data)
    model.optimize()

    return {
        "model": model,
        "info": info
    }


def run_all_sessions(data):
    """
    Function splits full dataset into sessions and runs optimization for each. It takes the full dataset with all sessions and returns a list of results for ea
    """
    departures = get_departure_times(data["Truck"])
    all_results = []

    start_session = 0
    for time_dep in departures:
        # extract the current session
        session_data = data.iloc[start_session:time_dep + 1]
        # optimize this session
        result = optimize_session(session_data)
        # add metadata
        result["session_start"] = start_session
        result["session_end"] = time_dep
        all_results.append(result)

        # update the start for the next session
        start_session = time_dep + 1

    return all_results

def run_analysis(data):
    """
    Runs the four analysis experiments and plots the  results.
    """
    base_params = get_default_parameters()

    print("\n--- Solar Capacity Experiment ---")
    solar_multipliers = [0.5, 0.75, 1.0, 1.25, 1.5]
    experiment_solar_capacity(data, base_params, solar_multipliers)

    print("\n--- Grid Capacity Experiment ---")
    qg_values = [100, 250, 500, 750, 1000]
    experiment_grid_capacity(data, base_params, qg_values)

    print("\n--- Truck Charging Power Experiment ---")
    qb_values = [20, 50, 75, 100, 125]
    experiment_truck_charge_power(data, base_params, qb_values)

    print("\n--- SoC Target Experiment ---")
    soc_targets = [0.6, 0.7, 0.8, 0.9, 1.0]
    experiment_soc_target(data, base_params, soc_targets)


def main():
    """
    Loads the input data, identifies charging sessions, runs the EMS model for each session, and returns all optimization results.
    """

    data = get_data_csv()                   # 1. load the data
    results = run_all_sessions(data)        # 2. run all the sessions
    print(f"Total sessions optimized: {len(results)}")  # 3. report the results

    run_analysis(data)                      # 4. run the analysis
    return results


if __name__ == "__main__":
    results = main()
```

```python
from gurobipy import Model, GRB
import pandas as pd


def get_default_parameters() -> dict:
    """
    Returns default parameters for the EMS optimization model.
    """
    return {
        "Xmax": 400.0,     # battery capacity (kWh)
        "Qb_max": 100.0,   # max charge/discharge (kW)
        "Qg_max": 535.0,   # grid power limit (kW)
        "kappa": 1000.0,   # penalty per kWh short of target
        "v_target": 0.8,   # target SoC fraction
        "X0": 0.0          # initial SoC
    }


def validate_data(data: pd.DataFrame) -> None:
    """
    Checks if the input Dataframe contains all required columns.
    """

    required_columns = [
        "Truck",
        "Solar_production_kWh",
        "Energy_consumption_kWh",
        "Price_per_kWh",
    ]
    for col in required_columns:
        if col not in data.columns:
            raise ValueError(f"Missing required column: '{col}' in input data")


def create_decision_variables(model: Model, T: int, Qb_max: float, Qg_max: float, solar) -> dict:
    """
    Add the decision variables to the model and returns them as a dictionary.
    """

    b = model.addVars(T, lb=-Qb_max, ub=Qb_max, name="b")
    x = model.addVars(T, lb=0, name="x")
    g = model.addVars(T, lb=-Qg_max, ub=Qg_max, name="g")
    a = model.addVars(T, lb=0, ub=solar, name="a")
    gamma = model.addVar(lb=0, name="gamma")

    return {"b": b, "x": x, "g": g, "a": a, "gamma": gamma}


def add_constraints(model: Model, data: pd.DataFrame, vars: dict, params: dict) -> None:
    """
    Add all constraints to the EMS optimization model.
    """

    T = len(data)
    truck = data["Truck"].values
    solar = data["Solar_production_kWh"].values
    load = data["Energy_consumption_kWh"].values

    b, x, g, a, gamma = vars["b"], vars["x"], vars["g"], vars["a"], vars["gamma"]

    # 1. Truck charge/discharge only when connected
    for t in range(T):
        model.addConstr(b[t] <= params["Qb_max"] * truck[t], name=f"charge_limit_{t}")
        model.addConstr(b[t] >= -params["Qb_max"] * truck[t], name=f"discharge_limit_{t}")

    # 2. SoC dynamics, only active when truck is connected
    model.addConstr(x[0] == params["X0"] + b[0] * truck[0], name="soc_start")
    for t in range(1, T):
        if truck[t] == 1: model.addConstr(x[t] == x[t - 1] + b[t], name=f"soc_dyn_{t}")
        else:
            # if the truck is disconnected, SoC is zero
            model.addConstr(x[t] == 0, name=f"soc_reset_{t}")

    # 3. Curtailment limit
    for t in range(T):
        model.addConstr(a[t] <= solar[t], name=f"curtail_{t}")

    # 4. Energy balance
    for t in range(T):
        model.addConstr(
            load[t] + b[t] + a[t] == solar[t] + g[t], name=f"energy_balance_{t}",)

    # 5. Departure SoC target (at end of session)
    model.addConstr(
        x[T - 1] >= params["v_target"] * params["Xmax"] - gamma, name="soc_target",)


def set_objective(model: Model, vars: dict, data: pd.DataFrame, params: dict) -> None:
    """
```

```python
    Defines the objective function for the EMS optimization model.
    """

    price = data["Price_per_kWh"].values
    g, gamma = vars["g"], vars["gamma"]
    T = len(data)

    objective = sum(price[t] * g[t] for t in range(T)) + params["kappa"] * gamma
    model.setObjective(objective, GRB.MINIMIZE)


def build_model(data: pd.DataFrame, parameters: dict | None = None):
    """
    Builds the full Gurobi EMS model for one charging session. Takes the dataframe as input and puts out the model and relevant info.
    """

    if parameters is None:
        parameters = get_default_parameters()

    validate_data(data)

    model = Model("EMS_Optimization")

    model.Params.OutputFlag = 0 # suppress Gurobi output

    T = len(data)
    vars = create_decision_variables(model, T, parameters["Qb_max"], parameters["Qg_max"], data["Solar_production_kWh"].values)
    add_constraints(model, data, vars, parameters)
    set_objective(model, vars, data, parameters)

    info = {
        "T": T,
        "parameters": parameters,
        "vars": vars
    }

    return model, info
```

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

from build_model import build_model
from utils import get_departure_times


def run_sessions(data: pd.DataFrame, parameters: dict) -> dict:
    """
    Run all sessions for a dataset and return total cost, total gamma, and session metrics.
    """

    departures = get_departure_times(data["Truck"])
    start_session = 0

    total_cost = 0.0
    total_gamma = 0.0
    session_metrics = []

    for dep in departures:
        session_data = data.iloc[start_session:dep + 1].copy()
        model, info = build_model(session_data, parameters=parameters)
        model.optimize()

        objval = model.ObjVal if model.Status == 2 else math.nan
        gamma_val = info["vars"]["gamma"].X if model.Status == 2 else math.nan

        total_cost += objval if not math.isnan(objval) else 0.0
        total_gamma += gamma_val if not math.isnan(gamma_val) else 0.0

        session_metrics.append({
            "session_start": start_session,
            "session_end": dep,
            "objval": objval,
            "gamma": gamma_val
        })

        start_session = dep + 1

    return {
        "total_cost": total_cost,
        "total_gamma": total_gamma,
        "session_metrics": session_metrics
    }


def plot_and_save(x, y_dict, xlabel, ylabel, title, filename):
    """
    Plotting function for the upcoming functions to visualize the results.
    """
    plt.figure()
    for label, y in y_dict.items():
        plt.plot(x, y, label=label)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.title(title)
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.savefig(filename)
    plt.close()
    print(f"Saved plot to {filename}")


# 1. Solar capacity experiment

def experiment_solar_capacity(data, base_params, multipliers, current_capacity_kW=1500, cost_per_kW=1250):
    """
    Vary solar capacity and compute operational + installation costs.
    Returns results DataFrame.
    """
    results = []

    for m in multipliers:
        session_data = data.copy()
        session_data["Solar_production_kWh"] = data["Solar_production_kWh"] * m

        res = run_sessions(session_data, base_params)

        added_kW = max(0, (m - 1.0) * current_capacity_kW)
        install_cost = added_kW * cost_per_kW
        net_cost = res["total_cost"] + install_cost


        results.append({
            "multiplier": m,
            "operational_cost": res["total_cost"],
            "install_cost": install_cost,
            "net_cost": net_cost
        })

    df = pd.DataFrame(results)

    # plots the results
    x = (df["multiplier"] - 1.0) * (current_capacity_kW / 1000.0)
    plot_and_save(x,
                  {"Operational cost": df["operational_cost"], "Net cost": df["net_cost"]}, "Added solar capacity (MW)", "Cost (EUR)", "Solar Capacity vs Costs

    # find the best multiplier
    best_idx = df["net_cost"].idxmin()
    best = df.loc[best_idx]
    print(f"Best solar multiplier: {best['multiplier']}, add {(best['multiplier']-1)*current_capacity_kW/1000:.2f} MW")
```

```python
    return df


# 2. Grid capacity experiment

def experiment_grid_capacity(data, base_params, qg_values):
    """
    Vary grid capacity (Qg_max) and compute average SoC miss fraction.
    """
    results = []

    for qg in qg_values:
        params = copy.deepcopy(base_params)
        params["Qg_max"] = qg

        res = run_sessions(data, params)
        session_count = len(res["session_metrics"])
        avg_gamma = res["total_gamma"] / max(1, session_count)
        avg_fraction = avg_gamma / (params["v_target"] * params["Xmax"])
        avg_cost = res["total_cost"] / max(1, session_count)

        results.append({
            "Qg_max": qg,
            "avg_gamma_kWh": avg_gamma,
            "avg_miss_fraction": avg_fraction,
            "avg_cost_per_session": avg_cost
        })
        print(f"Qg_max={qg}: avg_gamma={avg_gamma:.2f} kWh, avg_fraction={avg_fraction:.2%}, avg_cost per session={avg_cost:.2f}")

    df = pd.DataFrame(results)

    plot_and_save(df["Qg_max"], {"Avg fraction missed": df["avg_miss_fraction"]},
                  "Grid capacity (kW)", "Fraction missed", "Grid capacity vs SoC miss", "grid_capacity_vs_miss.png")
    return df


# 3. Truck charging power experiment

def experiment_truck_charge_power(data, base_params, qb_values):
    """
    Vary truck charge power (Qb_max) and compute total operational cost.
    """
    results = []

    for qb in qb_values:
        params = copy.deepcopy(base_params)
        params["Qb_max"] = qb

        res = run_sessions(data, params)
        session_count = max(1, len(res["session_metrics"]))
        avg_cost = res["total_cost"] / session_count

        results.append({"Qb_max": qb, "total_cost": res["total_cost"], "avg_cost_per_session": avg_cost})
        print(f"Qb_max={qb}: total_cost={res['total_cost']:.2f}, avg_cost per session={avg_cost:.2f}")

    df = pd.DataFrame(results)
    plot_and_save(df["Qb_max"], {"Total cost": df["total_cost"]},
                  "Truck charging power (kW)", "Total cost (EUR)", "Charging power vs cost", "charge_power_vs_cost.png")
    return df


# 4. SoC target experiment

def experiment_soc_target(data, base_params, targets):
    """
    Vary SoC target (v_target fraction) and compute total cost and avg gamma.
    """
    results = []

    for v in targets:
        params = copy.deepcopy(base_params)
        params["v_target"] = v

        res = run_sessions(data, params)
        avg_gamma = res["total_gamma"] / max(1, len(res["session_metrics"]))
        avg_cost = res["total_cost"] / max(1, len(res["session_metrics"]))
        results.append({
            "v_target": v,
            "total_cost": res["total_cost"],
            "avg_gamma_kWh": avg_gamma,
            "avg_cost_per_session": avg_cost
        })
        print(f"v_target={v:.2f}: total_cost={res['total_cost']:.2f}, avg_gamma={avg_gamma:.2f} kWh, avg_cost per session={avg_cost:.2f}")

    df = pd.DataFrame(results)
    plot_and_save(df["v_target"], {"Total cost": df["total_cost"]},
                  "SoC target fraction", "Total cost (EUR)", "SoC target vs cost", "soc_target_vs_cost.png")
    return df
```