Secondary control: A residential microgrid

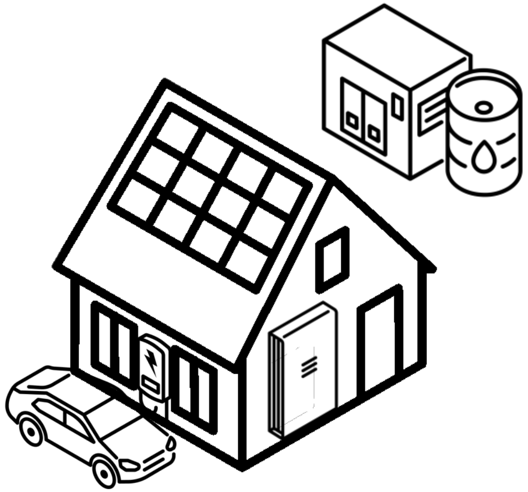
Bertrand Cornélusse, Thomas Stegen

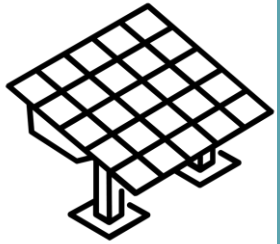
12 March 2025

I. Case study

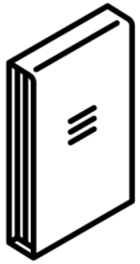
Introduction to the physical problem

Representation of the house

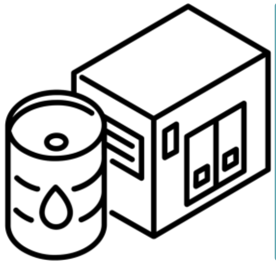




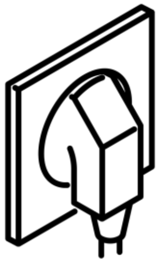
Photovoltaic panels are the main energy source



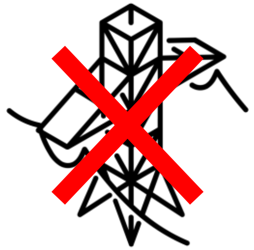
Battery storage system can be used as a slack



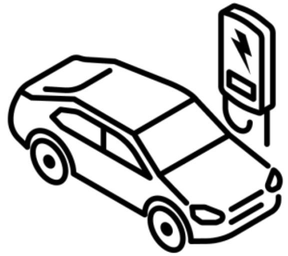
A diesel genset can also provide energy if needed



Load is fixed and should be always satisfied

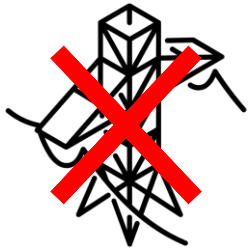


The system is offgrid, which creates the need of control

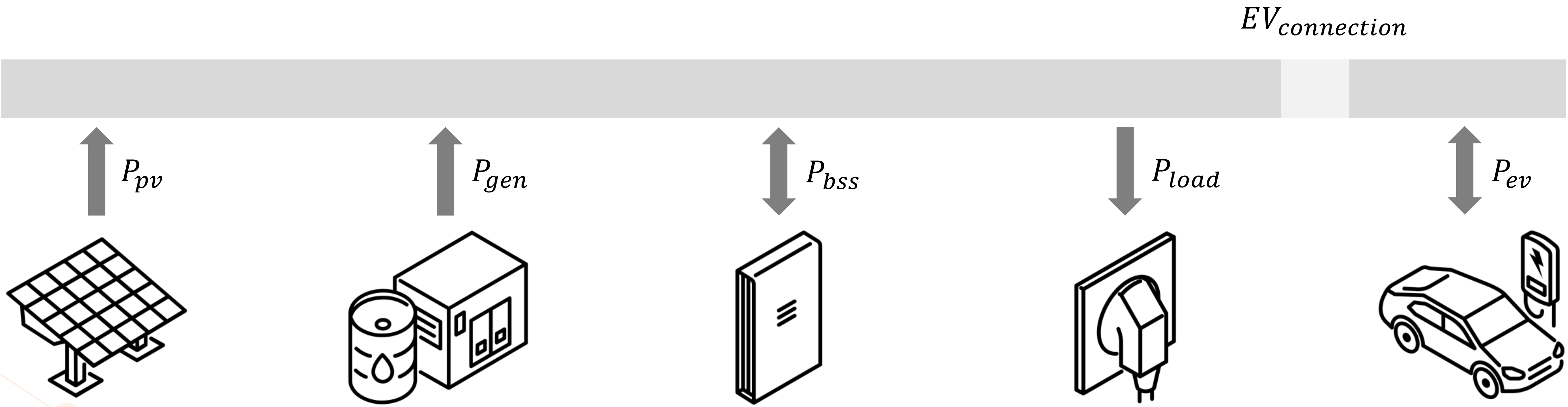


Electric vehicle connects, then should be charged

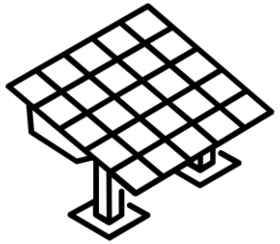
Bus view



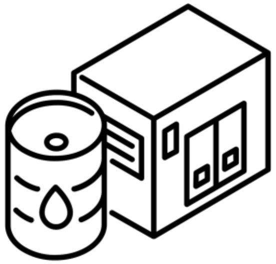
No external connection:
→ production and consumption should always match



Power generation



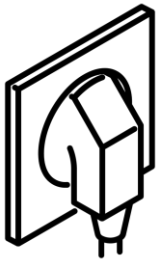
- Main source of energy
- Profile provide for MPPs
  - $P_{pv,max}$
- Connected via inverter
  - $P_{pv,nom}$
- Free energy
- Usage should be maximized



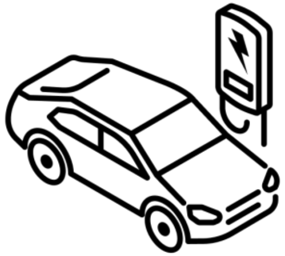
- Contingency usage
- Maximum/minimum power
  - $P_{max,gen}, P_{min,gen}$
- Fuel costs when used
- On-off switches avoided
- Power output should maximize efficiency

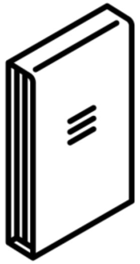
Power usage

- Not always connected
  - $EV_{connection}$ (binary)
- Charging time left accessible
  - $EV_{remaining\ time}$
- Should be charged when leaving
  - $SOC_{ev\ target}$
- Power limited by charger
  - $P_{ev\ nom}$
- Fixed capacity and efficiency
  - C_{ev}, eff_{ev}
- Bounded SOCs
  - $[SOC_{min\ ev},\ SOC_{max\ ev}]$



- Represents plugged-in appliances
- Defined by a profile
  - P_{load}
- Should always be satisfied





- Should mitigate imbalance
- Power limited by inverter
  - $P_{bss\ nom}$
- Fixed capacity
  - C_{bss}
- Constant charging efficiency
  - eff_{bss}
- Bounded SOCs
  - $[SOC_{min\ bss},\ SOC_{max\ bss}]$

Objectives of the controller

This is what a good controller should achieve, in decreasing order of priority

Ensure feasibility: Power/energy bounds, demand satisfaction,…

Approach optimality: final cost, asset management, genset operation,…

Improve reliability: robustness, more realistic model of appliances,…

II. Rule-based controller

Description of the first part of the homework

Python control_decision() function

This function is used to determine the power setpoints for the different components of the system

It is called at each time step to determine the power setpoints for the next time step

Inputs

- t: Current time step
- P_load: Demand [kW]
- P_pv_max: Maximum available PV power [kW]
- EV_connected: Presence of EV {0;1}
- EV_remaining_time: Time before EV leaves [h]
- P_gen_prev: Previous power of the genset [kW]
- SOC_bss: Current battery state of charge [0, 1]
- SOC_EV: Current EV state of charge [0, 1]

Outputs

- P_pv: Power output of the PV system [kW]
- P_gen: Power output of the generator [kW]
- P_bss: Power output of the battery [kW]
- P_ev: Power output of the EV [kW]

These values are used to update the state of the system and to calculate the cost of the system operation

Python control_decision() function ctd.

You should not change the input and outputs of this function.

You can use the given parameters and add some more

Parameters

- delta_t, time_steps, total_hours
- P_nom_pv, P_max_gen, P_min_gen,
- C_bss, SOC_min_bss, SOC_max_bss,
- eff_bss, P_nom_bss
- C_ev, SOC_min_ev, SOC_max_ev,
- eff_ev, P_nom_ev, SOC_target_ev

Code

```
MODEL_BASED = False
Scenario = 'S1' # Change to S2, S3, S4 to test the different scenarios

def control_decision(P_load, P_pv_max, EV_connected,
                     EV_remaining_time, P_gen_prev, SOC_bss, SOC_ev):
    # Define the ouput, you should write your controller here
    P_pv = 0
    P_gen = 0
    P_bss = 0
    P_ev = 0
    return P_pv, P_gen, P_bss, P_ev

run.run_sim(control_decision, Scenario, MODEL_BASED, FORECAST=False)
```

Rule-based controller guidelines

1. Construct a block diagram of your controller
2. Transcript your diagram in python in the function “control_decision”.
3. Test your code
4. Verify the results and go back to step 1 or step 2 if needed

5. Check, print results and make plots in utils.py

Python utils.py functions

- check(res) verifies physical constraints for each time step
- print(res) prints general information about simulation results (do not spam)
- plot(res) plots time dependent variables (do not make to many plots for the report)

res object

Python structure for data storing

Access with:

res.P_pv	res.SOC_bss
res.P_bss	res.P_pv_max
res.P_ev	res.P_load
res.P_gen	res.EV_connected
res.SOC_ev	res.t

to get array of outputs, results, parameters and time_steps

Code

```python
def check(res):
    # TODO: check that all your results are within physical limits
    # Print something if there is an error
    # Do it in a way to help you debug
    return

def print(res):
    # TODO: print informations on the overall simulation results
    # This should
    return

def plot(res):
    # TODO: plot the powers and other results for the simulation
    # You can access data with functions like this
    # Load = res.P_load
    # P_pv_max = res.P_pv_max
    # etc.
    return
```

III. Optimization-based controller

Description of the second part of the homework

Python control_decision() function

This function is used to determine the power setpoints for the different components of the system

It is called at each time step to determine the power setpoints for the next time step

Inputs

- t: Current time step
- P_load: Demand [kW]
- P_pv_max: Maximum available PV power [kW]
- EV_connected: Presence of EV {0;1}
- EV_remaining_time: Time before EV leaves [h]
- P_gen_prev: Previous power of the genset [kW]
- SOC_bss: Current battery state of charge [0, 1]
- SOC_EV: Current EV state of charge [0, 1]
- Model: Optimization model [pyomo object]

Outputs

- P_pv: Power output of the PV system [kW]
- P_gen: Power output of the generator [kW]
- P_bss: Power output of the battery [kW]
- P_ev: Power output of the EV [kW]

These values are used to update the state of the system and to calculate the cost of the system operation

Python control_decision() function ctd.

You should not change the input and outputs of this function

This function now uses two additional functions

Code

```python
def control_decision(P_load, P_pv_max, EV_connected, EV_remaining_time, P_gen_prev, SOC_bss, SOC_ev, model):
    # You should not change the input and outputs of this function
    model = update_model(model, SOC_bss, SOC_ev, P_load, P_pv_max, EV_connected, EV_remaining_time, P_gen_prev)

    solve_model(model)

    P_pv = model.P_pv.value
    P_gen = model.P_gen.value
    P_bss = model.P_charge_bss.value - model.P_discharge_bss.value
    P_ev = model.P_charge_ev.value - model.P_discharge_ev.value
    return P_pv, P_gen, P_bss, P_ev
```

1. Update mutable parameters in the model object
2. Solve the model
3. Retrieve variable values

Python update_model() function

This function updates the model with the current inputs

This function could also do some preprocessing if you want to add other varying parameters which are function of the inputs

Code

```python
def update_model(model, SOC_bss, SOC_ev, P_load, P_pv_max, EV_connected, EV_remaining_time, P_gen_prev):
    # You can change this function if you add mutable parameters in the model and you want to update them
    model.SOC_ev = SOC_ev
    model.SOC_bss = SOC_bss
    model.P_load = P_load
    model.P_pv_max = P_pv_max
    model.EV_connected = EV_connected
    model.EV_time_remaining = EV_remaining_time
    model.Pgen_prev = P_gen_prev
    return model
```

Python create_model() function

This function creates the model, with parameters, variables, objective and constraints_

This is where you will implement your controller

Code

```python
def create_model():
    # Create a concrete model
    model = ConcreteModel()

    # Mutable parameters that will be updated
    model.SOC_ev = Param(mutable=True)
    model.SOC_bss = Param(mutable=True)
    model.P_load = Param(mutable=True)
    model.P_pv_max = Param(mutable=True)
    model.EV_connected = Param(mutable=True, within=Binary)
    model.EV_time_remaining = Param(mutable=True)
    model.Pgen_prev = Param(mutable=True, initialize=0)

    # Variables
    model.P_pv = Var(within=NonNegativeReals)
    model.P_gen = Var(within=NonNegativeReals)
    model.P_charge_bss = Var(within=NonNegativeReals)
    model.P_discharge_bss = Var(within=NonNegativeReals)
    model.P_charge_ev = Var(within=NonNegativeReals)
    model.P_discharge_ev = Var(within=NonNegativeReals)
    # Binary variables for mode control
    model.gen_status = Var(within=Binary, initialize=0)

    # Define the objective function ----------------------------
    model.objective = Objective(sense=minimize,
    expr=0) #TODO

    #Constraints ------------------------------------------------
    # Power balance constraint:
    model.P_bal_cstr = Constraint(expr= model.P_load==model.P_pv)

    # PV constraints:#TODO

    # Battery constraints: #TODO

    # EV constraints: #TODO

    # Generator power constraints: #TODO

    return model
```

Optimization guidelines

1. Construct a basic physically-correct model
2. Tune your objective or constraints
3. Test your code
4. Verify the results and go back to step 1 or step 2 if needed

5. Compare to rule-based controller

IV. Report

Report guidelines

1. Maximum 6 pages
2. Include block diagram for rule-based
3. Include mathematical model for optimization based
4. Pay attention to the graphs and results you present
5. Be concise, focus on your decisions and results, not on the case study