Control in buildings

author: stephane.ploix@grenoble-inp.fr

reviewer:
mircea_stefan.simoiu@upb.ro

Case based reasoning for generating advices without explicit model

Developing a model is a long and difficult process. A less knowledge demanding approach consists in using a case based reasoning.

It consists in archiving every days as cases K_i and to evaluate the performance of each case in terms of compromise between satisfaction of the occupants and costs. The recorded variables used in he cases must be defined in terms of context (uncontrollable causes), actions (controllable causes) and effects (outcomes).

For the case based reasoning, the base of cases must be complete i.e. same causes (context and actions) of cases must lead to the same effects.

It assumes relevant distances have been defined to compare causes, actions and effects.

Let's denote $d_{CA,w}(K_i, K_j)$, a weighted CA-distance to compare causes, $d_{E,\sigma}(K_i, K_j)$ is a sensitive distance to compare effects and α is a bounding function. The sensitive distance is used to compare effects because it takes into account the inhabitant sensitivity between 2 effects.

The problem can be solved by finding a bounding function:

$$d_{E,\sigma}(\Pi_E(K_i), \Pi_E(K_j)) \leq \alpha(d_{CA,w}(\Pi_{CA}(K_i), \Pi_{CA}(K_j)) + \epsilon)$$

α can be found by searching for:

$$\alpha(w) = argmax_{(K_i,K_j)\in\mathbb{K}^2} \left(\frac{d_{E,\sigma}(\Pi_E(K_i), \Pi_E(K_j))}{d_{CA,w}(\Pi_{CA}(K_i), \Pi_{CA}(K_j)) + \epsilon} \right)$$

The smaller $\alpha(w)$, the complete is the base of cases. Note that α depends on the weighted CA-distance.

Local completeness

If α is too large, it's possible to define a maximum acceptable value for α in a neighborhood $\mathcal{V}_{\mathbb{K}}(C)$ of the current context C defined by a distance d_w such as:

$$\forall K_i, K_j / \left(\Pi_C(K_i), \Pi_C(K_j)\right) \in \mathcal{V}_{\mathbb{K}}(C)^2 \frac{d_{E,\sigma}(\Pi_E(K_i), \Pi_E(K_j))}{d_{CA,w}(\Pi_{CA}(K_i), \Pi_{CA}(K_j)) + \epsilon} \leq \alpha(w)$$

Computing a weighted distance

The weights of the CA distance must minimize the incompleteness i.e.

$$w^* = argmin_w \alpha(w)$$

subject to $\sum_k w_k = 1$

calculate the barycentre of several action data for adaptation to current context

Let $K_i; i \in N$ be the cases located in the neighborhood of a context and $(\Pi_A(K_i), d_C(C, \Pi_C(K_i)), p(K_i)); i \in N$ be respectively, the action data, the distance with the current context, and $p(K_i) \in [0, \infty[$ the performance of the case K_i. i.e. the bigger the best.

A neighbor case is "all the more interesting when is distance $d_C(C, \Pi_C(K_i))$ is as small as possible and the performance $p(K_i)$ is as small as possible". Let's denote $\hat{d}_C = max_{i \in N}(d_C(C, \Pi_C(K_i)))$, the biggest distance to the current context. If $\hat{d}_C \neq 0$, the weight ω_i for each case actions of the barycentric approach is directly drawn from the intentions in between double quotations:

$$\omega_i = \frac{\hat{d}_C - d_C(C, \Pi_C(K_i))}{\hat{d}_C} \times p(K_i)$$

Then, the suggested action is given by:

$$(A)_i = \sum_i \frac{\omega_i}{\Omega} (\Pi_A(K_i))_i$$

If $\hat{d}_C = 0$, it means all the neighbors have exactly the same context, so the most performant case ix selected.

Core concepts of the buildingenergy project

The buildingenergy project can model and control building systems modeled by nonlinear state space models. It represents both the thermal phenomena and the CO2 concentration depending both on ventilation air flows. It's a dynamic simulator: it represents the inertia of a multi-zone buildings by decomposing walls into several layers depending on the wall materials. The generated state space model dealing both with temperatures and CO2 concentrations is integrated over 1 hour into a recurrent time-varying discrete state space model. The simulation time is reduced thanks to a dynamic programming algorithm.

System variables have to be distinguished from the **data**: their roles are different as well as their names. System variables are symbolic variables that can be let symbolic until a simulation set a series of values. Data are a single value or a series of values coming either from measurements (weather, recorded values) or from the generated signals (profiles, set-points).

The naming convention for system variables and for data are different.

Naming of system variables

A room is related to system variables, with predefined names. Let's consider a room named "room1". It's related variables are:

- TZroom1: the air temperature inside the room
- PZroom1: the total power coming into the room (sun power, body metabolism power, HVAC power, thermal power coming from electric appliances,...)
- PCO2room1: the CO2 production (by breath) in room 1
- CCO2room1: the CO2 concentration in room 1

When a zone is simulated, say "room1", it's volume ($50m^3$) has to be defined by:

```
dp.add_parameter('room1:volume', 50)
```

The airflow between 2 connected zones "room1" and "room2" must be named:

- room1-room2:Q_any_name or room2-room1:Q_any_name

A system variable "window_position" located at the interface between 2 rooms, say "room1" and "room2" must be named: room1-room2:window_position, or alternatively, room2-room1:window_position.

Naming of data

A data (an invariant parameter, or a time series) "occupancy" related to a room named "room1" should be named: room1:occupancy

Binding a system variable with a data

With the binding class, it is possible to associate a data i.e. a time series or a time-invariant parameter value depicted by its name, with a related system variable. For instance, the system variable "TZoutdoor" can be bound to "weather_temperature" coming from weather data with:

```
bindings = Bindings()
bindings('TZoutdoor', 'weather_temperature')
```

weather

A building is set in a specific weather. Historical data are automatically collected from the open-meteo.com website according to the provided GPS coordinates (it uses the data of the closest meteorological station). It collects automatically the elevation of the specified location and load data into a data provider.

Weather variables at a given gps location are loaded: their naming pattern is: "weather_variable" where "variable" can be a temperature (so weather_temperature), "humidity", "apparent_temperature", "precipitation", "rain", "snowfall", "snow_depth", "surface_pressure", "cloud_cover", "wind_speed_10m", "wind_direction_10m", "wind_gusts_10m", "shortwave_radiation", "direct_radiation",

solar gains

A building is also a solar system that provides solar gains to rooms through different windows. The skyline composed by the surrounded landscape is automatically downloaded. Additional solar masks can be added to represent masks specific to window. For a collector, a solar mask is added to represent the surrounding wall of the window. For instance,

```
surface_window = 2.2*.9
direction_window = -180  # North directed
solar_protection = 90  # no protection
solar_factor = 0.56

# use the weather data, including cloudiness to compute the solar gains
solar_model = SolarModel(dp.weather_data)
solar_system = SolarSystem(solar_model)
```

Building systems

A building system is composed of zones (rooms) that can be simulated with a 1 hour time step, or not. All the zones (but not "outdoor" zone, which is implicitly existing) must be declared with, for instance:

```
state_model_maker = BuildingStateModelMaker('cabinet', 'toilet', ...)
```

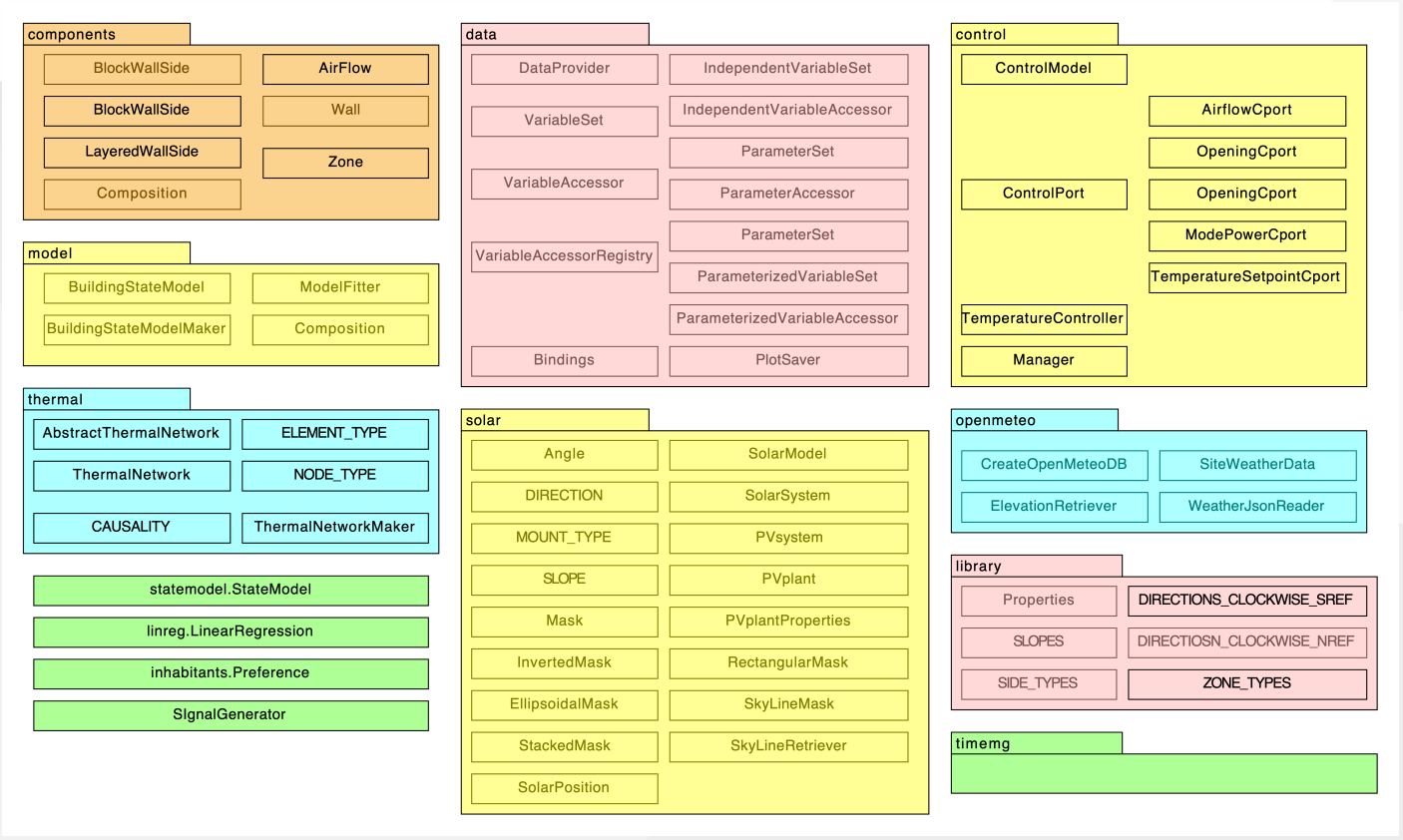
The simulated zones have to be declared with, for instance:

```
state_model_maker.simulate_zone('cabinet', 'toilet')
```

The zones with airflow passing from one to another have to be connected with:

code architecture

Here are represented the different modules of the buildingenergy project with contained classes, functions, enumerations,...



the data module

It manages all what is related to data, including parameters, time series coming from weather history but also time series designed as occupancy scenarios, set-points and much more.

The data provider is the facade class: it is the one to be used by building modeler (classes ending by Set or Accessor are used internally by the data provider).

A data can be overloaded: for instance, if you load a recorded temperature for a room and you simulate it with a controller that leads to other values: writing the simulated values in the DataProvider will overload the existing values (the overloaded can still be obtained).

A data can be a time-invariant parameter, a time-series (named data) or a parameterized data that relies on a formula mixing parameters and data (time-series)

Upper and lower bounds for parameters (only those that can be adjusted to match recorded data) and variables must be provided.

Any data whose time slots, called reference datetimes, are consistent with the existing data can be added as an external variable.

All the data from a data provider can be plotted and saved.

the solar module

It manages the calculation of solar radiations from the current GPS coordinates, the weather, the surrounding masks, the tilt and slope of the collecting surface... SolarModel contains a geometrical model of the sun: it can compute heliodon. It is used by SolarSystem that can compute solar gains through a window but also by PVplant, that represents photovoltaic cells disposal with shading phenomena depending of the panel mount.

the component module

It contains different components to create a building envelope, composed of walls made of sides (a wall is usually composed of several sides, depending on the layered materials,..):

- a composition describes the thicknesses of materials for each different layer
- a layered wall side is related to a composition and to a surface
- a block wall side is related to a component like a thermal bridge

the library module

It contains a set of physical properties (thermal conductivity, density,...) coming from the Microsoft Excel file (located at the project root folder) named "propertiesDB.xslx". The properties of some materials are preloaded. The library can be imported with:

```
from buildingenergy.library import properties
```

Loading another material, for instance 'Asphalt' from the Excel file can be done by referring to its row (#4) in the Excel table and by naming it 'goudron':

```
properties.store('goudron', 'thermal', 4)
```

Constants are also defined in this module like SLOPES, SIDE_TYPES and DIRECTIONS_CLOCKWISE_SREF used by the solar module.

the timemg module

It contains lots of functions to convert a time from a format to another.

the weather module

It can read weather data coming from 'open-meteo.com'. If historical weather data are requested for a location, it will automatically download the data from open-meteo.com and save them locally next to the root folder of the project, in a folder named 'data'. The weather data contains data from January 1, 1980 till the day before the request but a reference year has generally to be specified depending on the application. The requested data are encapsulated in a `weather_data_site` object, that is incorporated into a data provider.

the thermal module

The thermal module is a low-level module that transforms RC diagrams with power sources, modeled as thermal networks, into static or dynamic state models. It should be used directly except for creating new HVAC appliances or new kind of envelopes.

the model module

It can generates high-level state space models by aggregating the thermal state space model coming from thermal network defined with the previous module, and a CO2 model. The StateModelMaker has to be called at each simulation step because the global state space model is time-varying. If a model has already been computed, it won't be recalculated: it will be reused from a cache memory according to the principle of dynamic programming. This model contains also a model fitter that can performed Morris sensitivity analysis and adjust the model parameters to better match the simulated data with the recorded ones.

the signal generator

It's a signal generator behaving like factory for generating signals that can be profiles (occupancy, window opening,...) or setpoints for control. It will be detailed later.

the control module

It contains a set of control ports that restricts the possible actions of the data in the data provider. Without using control port, any value can be assigned to a data but with, for example, in case of absence of inhabitants, only the value 0 is possible, for example, because the window can't be opened. The module contains also temperature controller, that adjusts the power to reach a predefined set-point as fast as possible (remembering the time slot is 1 hour). Eventually, the manager implements several temperature controllers.

A commented example (GreenCAB2)

All the needed imports are:

```
from __future__ import annotations
import numpy
from buildingenergy.solar import SolarModel, SolarSystem
from buildingenergy.data import DataProvider, Bindings
from buildingenergy.model import BuildingStateModelMaker
from buildingenergy.components import SideFactory
from buildingenergy.control import TemperatureSetpointCport, TemperatureController, ControlModel
from buildingenergy.control import AirflowCport, ModePowerCport, Manager
from buildingenergy.signal import SignalGenerator
from buildingenergy.inhabitants import Preference
from buildingenergy.library import SIDE_TYPES
import time
```

The bindings between system variables and data appears at the beginning:

```
print("linking model variables with recorded data...")
bindings = Bindings()
bindings('TZoutdoor', 'weather_temperature')
bindings('PZcabinet', 'cabinet:Pheat')
bindings('PCO2cabinet', 'cabinet:PCO2')
bindings('PCO2toilet', 'toilet:PCO2')
bindings('CCO2cabinet', 'cabinet_CO2_concentration')
bindings('cabinet-outdoor:z_window', 'window_opening')
bindings('cabinet:occupancy', 'occupancy')
```

Load the weather data (download them from the gps coordinates if the json weather file named "Saint-Julien-en-Saint-Alban.json" does not exist in the data folder next to the root folder) and define the time period for control simulation:

```
print('Loading data...')
dp = DataProvider(location='Saint-Julien-en-Saint-Alban', latitude_north_deg=44.71407488275519,
 longitude_east_deg=4.633318302898348, starting_stringdate="1/1/2022", ending_stringdate="31/12/2022",
  bindings=bindings, albedo=0.1, pollution=0.1, number_of_levels=4)
```

The number of levels is used for reusing calculated state space model to save time (3 to 5 is a good comprise between accuracy and computation time).

Compute solar gains (through window on north side):

```
print("Displaying data solar gain passing through window...")
surface_window = 2.2*.9
direction_window = -180
solar_protection = 90  # no protection
solar_factor = 0.56

solar_model = SolarModel(dp.weather_data)
solar_system = SolarSystem(solar_model)
solar_system.add_collector('main', surface=surface_window, exposure_in_deg=direction_window,
slope_in_deg=90, solar_factor=solar_factor)
solar_gains_with_mask, _ = solar_system.solar_gains_in_Wh()
dp.add_external_variable('Psun_window', solar_gains_with_mask)
```

Some not-adjustable related to the greenCAP dimensions are defined here:

```
# Dimensions
insulation_thickness = 150e-3
surface_cabinet: float = 5.85*2.14
volume_cabinet: float = surface_cabinet*2.29
surface_toilet: float = 1.18*2.14
volume_toilet: float = surface_toilet*2.29
surface_cabinet_wall: float = (2 * 6000e-3 + 2440e-3) * 2590e-3 - surface_window
```

Use signal generators to define the occupancy profiles, the related presence and free heat gain coming into the cabinet:

```
body_metabolism = 100
occupant_consumption = 200
occupancy_sgen = SignalGenerator(dp.series('datetime'))
occupancy_sgen.daily([0, 1, 2, 3, 4], {0: 0, 8: 3, 18: 0})  # 12: 0, 13: 3,
occupancy: list[float] = occupancy_sgen()
dp.add_external_variable('occupancy', occupancy)
presence: list[int] = [int(occupancy[k] > 0) for k in range(len(dp))]
dp.add_external_variable('presence', presence)

dp.add_external_variable('PZcabinet', [occupancy[k] * (body_metabolism +
occupant_consumption) + dp('Psun_window', k) for k in dp.ks])
dp.add_parameter('PZtoilet', 0)
```

Generate the temperature set-points for summer and winter and the mode (heating=1, off=0, cooling=-1) of the heat pomp:

```
temperature_sgen = SignalGenerator(dp.series('datetime'), None)
temperature_sgen.daily([0, 1, 2, 3, 4], {0: None, 7: 20, 19: None}, merger=Merger(min, 'r'))
heating_period = temperature_sgen.seasonal('16/11', '15/3', 1, merger=Merger(max, 'b'))
temp_sgen = SignalGenerator(dp.series('datetime'), None)
temp_sgen.daily([0, 1, 2, 3, 4], {0: None, 7: 22, 19: None}, merger=Merger(min, 'r'))
cooling_period = temp_sgen.seasonal('16/3', '15/11', 1, merger=Merger(max, 'b'))
temperature_sgen.combine(temp_sgen(), merger=Merger(min, 'n'))
dp.add_external_variable('TZcabinet_setpoint', temperature_sgen())

hvac_modes_sgen = SignalGenerator(dp.series('datetime'))
hvac_modes_sgen.combine(heating_period, merger=Merger(max, 'l'))
hvac_modes_sgen.combine(cooling_period, merger=Merger(lambda x, y: x - y, 'n'))
dp.add_external_variable('mode', hvac_modes_sgen())
```

Definition of ventilation scenarios still using signal generators. 3 levels of ventilation are defined: one corresponding to air infiltration when the fan is off, another one to the normal ventilation when fan is on (from 7am to 18pm during weekdays only) and a latter one, which is triggered by the reactive control of the energy **Manager** when the CO2 concentration passes a given level (3000ppm).
CO2 is produced by occupants' breath is the cabinet ; the production in toilet is neglected. The ventilation airflow depends on the occupancy as well.

```
q_infiltration: float = volume_cabinet/3600
q_ventilation = 6 * volume_cabinet/3600
q_freecooling: float = 15 * volume_cabinet/3600
body_PCO2 = 7

dp.add_parameter('CCO2outdoor', 400)
```

The normal ventilation profiles are added (but not the freecooling, which will be trigger by the manager during the simulation under some conditions).

```
dp.add_external_variable('ventilation', ventilation)
dp.add_external_variable('cabinet-outdoor:Q', [q_infiltration + ventilation[k]*q_ventilation for k in range(len(dp))])
dp.add_external_variable('toilet-cabinet:Q', [q_infiltration + ventilation[k]*q_ventilation for k in range(len(dp))])
```

A ventilation control port is created to define what are the permitted ventilation airflows between the cabinet and outdoor depending of the heating/cooling mode, but also on the presence. The tuple of integers corresponding to an airflow level is given by:
$(mode, presence)$
where mode is equal to 1, 0, -1 (resp. heating, off and cooling periods) and presence (0 or 1):

```
ventilation_cport = AirflowCport(dp, 'cabinet-outdoor:Q', 'mode', 'presence', {-2: (q_infiltration,),
 -1: (q_infiltration, q_freecooling), 0: (q_infiltration,),
  1: (q_infiltration, q_ventilation, q_freecooling), 2: (q_infiltration,),
  3: (q_infiltration, q_ventilation)})
```

A state model generator is then created, considering 2 zones (cabinet and toilet), a periodic depth of penetration of 1h and a maximum state model order equal to 5 in order to reduce the computation time:

```
state_model_maker = BuildingStateModelMaker('cabinet', 'toilet', data_provider=dp,
periodic_depth_seconds=3600, state_model_order_max=5)
```

Wall side compositions can then be created (accordingly to the content of the material thermal properties loaded into the library module):

```
wall = SideFactory(('wood', 3e-3), ('polystyrene', insulation_thickness), ('steel', 5e-3), ('wood', 3e-3))
floor = SideFactory(('wood', 10e-3), ('polystyrene', insulation_thickness), ('steel', 5e-3))
ceiling = SideFactory(('wood', 3e-3), ('polystyrene', insulation_thickness), ('steel', 5e-3))
glazing = SideFactory(('glass', 4e-3), ('air', 10e-3), ('glass', 4e-3), ('air', 10e-3), ('glass', 4e-3))
internal = SideFactory(('wood', 9e-3), ('air', 20e-3), ('wood', 9e-3))
```

The different sides of the cabinet wall are then described:

```
state_model_maker.make_side(wall('cabinet', 'outdoor', SIDE_TYPES.WALL, surface_cabinet_wall))
state_model_maker.make_side(floor('cabinet', 'outdoor', SIDE_TYPES.FLOOR, surface_cabinet))
state_model_maker.make_side(ceiling('cabinet', 'outdoor', SIDE_TYPES.CEILING, surface_cabinet))
state_model_maker.make_side(glazing('cabinet', 'outdoor', SIDE_TYPES.GLAZING, surface_window))
```

Then the internal wall separating the 2 zones:

```
state_model_maker.make_side(internal('cabinet', 'toilet', SIDE_TYPES.WALL, 2138e-3 * 2290e-3))
```

And eventually, the different sides of the toilet wall:

```
state_model_maker.make_side(wall('toilet', 'outdoor', SIDE_TYPES.WALL, (1315e-3 * 2 + 2138e-3) * 2290e-3))
state_model_maker.make_side(floor('toilet', 'outdoor', SIDE_TYPES.FLOOR, 2440e-3 * 1330e-3))
state_model_maker.make_side(ceiling('toilet', 'outdoor', SIDE_TYPES.CEILING, 2440e-3 * 1330e-3))
```

The 2 zones to be simulated are specified together with the airflow names connecting zones.
A first state space model is generated using nominal airflow values (level 0) used to compute the projection matrices used by the model reduction algorithm:

```
state_model_maker.simulate_zone('cabinet', 'toilet')
state_model_maker.connect_airflow('cabinet', 'outdoor', dp('cabinet-outdoor:Q'))  # nominal value
state_model_maker.connect_airflow('toilet', 'cabinet', dp('toilet-cabinet:Q'))  # nominal value

nominal_state_model = state_model_maker.make_k()
```

The inhabitants' preferences are then specified to assess the performance of the simulation results: the indoor temperatures belonging to the interval $[19, 24]$ are considered as perfect (thermal comfort satisfaction in 100%) while those out of the interval $[16, 29]$ are considered as critical: the satisfaction is lower or equal to 0 in case of presence otherwise, in case of absence, the temperature value is not accounted. Same for the CO2 but with a critical value over 1500ppm and a perfect value equal to 500ppm or lower. The relative weight between thermal and CO2 satisfaction is given as well as the weight dealing with energy cost and aggregated comfort. In order to compute the final energy the COP of each mode is given.

```
preference = Preference(preferred_temperatures=(19, 24), extreme_temperatures=(16, 29),
 preferred_CO2_concentration=(500, 1500), temperature_weight_wrt_CO2=0.5,
  power_weight_wrt_comfort=0.5, mode_cop={1: 2, -1: 2})
```

A direct manager is then created: it creates several control ports, which restrict the possible values for a variable depending on some conditions:

- an airflow port, with 3 possible levels of ventilation depending on a presence and on the heating/cooling/off mode
- a temperature set-point port, which defines the possible temperature values depending on the heating/cooling mode
- a power port that defines maximum bounds on the HVAC power
  It defines a controller with a control variable (the HVAC power) and a target set_point to reach, corresponding to a temperature set-point port.

A control method makes it possible to introduce control strategy based on heuristics.

```
manager = DirectManager(dp, state_model_maker)
```

Finally, the simulation is performed thanks to ControlModel that connects a control manager to the building system. The resulting performances are assessed and some variables are added to the data provider for plotting purpose.

```
control_model = ControlModel(state_model_maker, manager)
control_model.simulate()

Pheater = dp.series('PZcabinet_control')
occupancy = dp.series('occupancy')
preference.print_assessment(dp.series('datetime'), Pheater=Pheater,
 temperatures=dp.series('TZcabinet'), CO2_concentrations=dp.series('CCO2cabinet'),
  occupancies=dp.series('occupancy'), action_sets=(), modes=dp.series('mode'), list_extreme_hours=True)
electricity_needs = [abs(Pheater[k])/2 + occupancy[k] * occupant_consumption for k in dp.ks]
dp.add_external_variable('electricity needs', electricity_needs)
```

the control manager

As explained above, a control manager defines control ports to restrict the possible control values and defines a single control value for each time slot.

Here is the code of the direct control manager used for the GreenCAB2 project:

```
class DirectManager(Manager):

    def __init__(self, dp: DataProvider, building_state_model_maker: BuildingStateModelMaker) -> None:
        super().__init__(dp, building_state_model_maker)

    def make_control_ports(self) -> None:
        self.airflow_cport = AirflowCport(dp, 'cabinet-outdoor:Q', 'mode', 'presence',
         {-2: (q_infiltration,), -1: (q_infiltration, q_freecooling),
          0: (q_infiltration,), 1: (q_infiltration, q_ventilation, q_freecooling), 2: (q_infiltration,), 3: (q_infiltration, q_ventilation)})

        self.temperature_setpoint_cport = TemperatureSetpointCport(dp, 'TZcabinet_setpoint', 'TZcabinet', mode='mode',
        mode_value_domain={1: (13, 19, 20, 21, 22, 23), 0: None, -1: (24, 25, 26, 28, 29, 32)})

        self.mode_power_cport = ModePowerCport(dp, 'PZcabinet_control', 'PZcabinet', max_heating_power=3000, max_cooling_power=3000, mode='mode', full=False)

    def make_controllers_and_initvals(self) -> dict[TemperatureController, float]:
        return {self.temperature_controller(self.temperature_setpoint_cport, self.mode_power_cport): 0}
```

```
    def controls(self, k: int, X_k: numpy.matrix, current_output_dict: dict[str, float]) -> None:
        Tin: float = current_output_dict['TZcabinet']
        Tout: float = self.dp('weather_temperature', k)
        if self.dp('presence', k) == 1:
            if 20 <= Tout <= 23 or self.dp('CCO2cabinet', k) > 3000:
                self.airflow_cport(k, q_freecooling)
            elif Tin > 23 and Tout < 20:
                self.airflow_cport(k, q_freecooling)
```

Let's examine how the controller are implemented: there are 2 kinds of controllers considered: those that react instantaneously (but rather not existing in practice) and those that reach their set-point at the next time slot.

The system to control is written like:

$$Y_k = C_k X_k + D_k U_k + D_k^i \delta U_k^i$$

$$X_{k+1} = A_k X_k + B_k U_k + B_k^i \delta U_k^i$$

The aim is control control Y_k^j (a temperature for us) by acting on the input (a heating/cooling power for us) U_k^i.

It can be modeled by: $\delta U_k^i / Y_k^j = S_j^T Y_k = \tilde{Y}_k^j$, with S_j a selection vector.

$$\tilde{Y}_k^j = S_j^T C_k X_k + S_j^T D_k U_k + S_j^T D_k^i \delta U_k^i$$

If the following condition is satisfied: $S_j^T D_k^i \neq 0$, the input value that leads to the output consistent with the set-point is:

$$\delta U_k^i = \frac{\tilde{Y}_{k,j} - S_j^T C_k X_k - S_j^T D_k U_k}{S_j^T D_k^i}$$

If this condition cannot be satisfied, the instantaneous control cannot be done.

Therefore, a one time slot delay can be considered. The aims is given by:

$$\delta U_k^i / Y_{k+1}^j = S_j^T Y_{k+1} = \tilde{Y}_{k+1}^j \text{ and } S_j^T D_{k+1}^i = 0$$

Developing the observation equation of the state space model leads to:

$$Y_{k+1} = C_{k+1} X_{k+1} + D_{k+1} U_{k+1} + D_{k+1}^i \delta U_k^i$$

Introducing the state recurrent equation yields:

$$\leftrightarrow Y_{k+1} = C_{k+1}A_kX_k + C_{k+1}B_kU_k + C_{k+1}B_k^i\delta U_k^i + D_{k+1}U_{k+1} + D_{k+1}^i\delta U_{k+1}^i$$

$$\rightarrow \tilde{Y}_{k+1}^j = S_j^T C_{k+1}A_kX_k + S_j^T C_{k+1}B_kU_k + S_j^T C_{k+1}B_k^i\delta U_k^i + S_j^T D_{k+1}U_{k+1}$$

It yields, if the following conditions are satisfied: $S_j^T C_{k+1}B_k^i \neq 0$ and $S_j^T D_{k+1}^i = 0$, the one slot delay control is given by:

$$\delta U_k^i = \frac{\tilde{Y}_{k+1}^j - S_j^T C_{k+1}A_kX_k - S_j^T C_{k+1}B_kU_k - S_j^T D_{k+1}U_{k+1}}{S_j^T C_{k+1}B_k^i}$$

where U_k stands for the input variable without control.

If C_k and D_k are time-invariant and if the following conditions are satisfied $S_j^T C B_k^i \neq 0$ and $S_j^T D^i = 0$, the control input value must be equal to:

$$\delta U_k^i = \frac{\tilde{Y}_{k+1}^j - S_j^T C A_k X_k - S_j^T C B_k U_k - S_j^T D U_{k+1}}{S_j^T C B_k^i}$$

This situation is the common situation for HVAC control.

the signal generator

Generating a signal starts by specifying the series of consecutive reference datetimes, with a one hour time slot:

```
signal_generator = SignalGenerator(datetimes: list[datetime.datetime], constant: None | float = 0)
```

The initial signal is made of a repetition of the provided constant value.

The signal generator is a signal factory: it contains a signal under construction that can be modified by different processing, whose order matter. Most of the time a processing consists in generating a new signal and to combine it to the signal under construction thanks to a merger.

Merging is composed of 2 kinds of actions:

- defining a merging operator to combine the values of the signal under construction with the newly generated one, for each time slot where both values are not None: any binary operator can be used like `min` or `max` but any binary operators can also be designed using a Python lambda function, like `lambda x, y: x - y`.

- defining a dominance rule that defines how the signal's None values are handled during the merge operation. The following table summarizes the 4 dominance rules ('x' stands for any operator here):

signal1	signal2	left	right	both	noone
val1	val2	val1 x val2	val1 x val2	val1 x val2	val1 x val2
None	val2	None	val2	None	val1
val1	None	val1	None	None	val2
None	None	None	None	None	None

A merger must be used to specified how the new signal (considered at the right side) will be merged with the signal under construction (considered at the left side):

```
Merger(op, 'left'/'right'/'both'/'noone')
```

For instance, `Merger(max, 'l')` or `Merger(lambda x, y: x - y, 'n')`. Note that the first letter for the dominance is enough

The methods related to a signal generator that use a merger are (a default merger is specified for each method):

- `signal_generator.offset(value, merger=Merger(lambda x, y: x + y, dominance='l'))`
  : add a constant value
- `signal_generator.seasonal(dm_start: str, dm_end: str, in_value: float = 1, out_value: float = None, merger: Merger = Merger(max, dominance='r'))`
  : create a period delimited by 2 dates in a year (format `dd/mm)`), `dm_start` and `dm_end` taking the in_value inside the period and out_value outside

- `signal_generator.daily(weekdays: list[int], hour_setpoints: dict[int, float], merger: Merger = Merger(max, dominance='l'))`
  : add a repeated day sequence where weekdays define the days where the day sequence appears (0=Monday,..., 6=Sunday), and the hour setpoints specify each hour in a day a new value (or None) must be set.
- `signal_generator.long_absence(high_setpoint: float, long_absence_setpoint: float, number_of_days: int, presence: list[float], merger: Merger = Merger(min, dominance='b'))`
  : detect long absence from a presence vector accordingly to the specified number of days and set the `long_absence_setpoint` during long absence and `high_setpoint` otherwise.

- `signal_generator.combine(*signals: list[float | None], merger: Merger = Merger(lambda x, y: x + y, dominance='n'))`
  : combine an external signal with the signal under-construction according to the specified merger

Other processing methods do not use merger like:

- `signal_generator.cap(capped_value: float, threshold: float, capping_values: list[float] = None)` replace the values greater than the specified threshold applied, either to the signal under construction itself (if `capping_values` is None), or to the corresponding `capping_values` else, by the specified `capped_value`
- `signal_generator.cup(cupped_value: float, threshold: float, cupping_values: list[float] = None)` replace the values lower than the specified threshold applied, either to the signal under construction itself (if `cupping_values` is None), or to the corresponding `cupping_values` else, by the specified `cupped_value`

- `signal_generator.amplify(alpha: float)` multiply the signal under construction by a scalar alpha
- `signal_generator.integerize(none_value: int = 0)` replace each None value in the signal under construction by the specified `none_value`

An any moment, the signal under construction can be obtained by:

```
signal_generator()
```

from thermal network to state model

Definition and Properties of a selection matrix A selection matrix $S_1 = S(m_1, n)$ is a full row rank matrix composed of 0, -1 and 1 with at most one non-zero value per column and one, and only one, non-null value per row. It satisfies:

$$S_1 \times S_1^T = I_{m_1}$$

Moreover, $S_2 = S_2(m_2, n)$ $(m_1 + m_2 \leq n)$ is a complementary selection matrix if it satisfies: $S_1 S_2^T = S_2 S_1^T = 0$.

$\{S_1, S_2, \ldots, S_p\}$ is a set of partitioning matrices if

- S_i is a selection matrix $\forall i \in \{1, \ldots, p\}$
- $\forall i \neq j$, S_j is complementary to S_i
- $\sum_i m_i = n$

Therefore, the partioning matrices $\{S_i\}$ satisfy:

$$\begin{bmatrix} S_1 \\ S_2 \\ \dots \\ S_p \end{bmatrix} \times \begin{bmatrix} S_1^T & S_2^T & \dots & S_p^T \end{bmatrix} = I_n$$

$$S_1 X = Y \leftrightarrow X = S_1^T Y.$$

Capacitance assumption: All the capacitances are modelled by:

$$C_i \frac{d}{dt} \tau_i = \varphi_i \text{ with } \tau_i \in T$$

Problem statement

Let T be the vector of the n temperatures appearing in nodes. Let's define a triplet of partitioning matrices:

$$\begin{bmatrix} T_{in} \\ \bar{T}_{in} \end{bmatrix} = \begin{bmatrix} S_{in} \\ \bar{S}_{in} \end{bmatrix} T \leftrightarrow T = S_{in}^T T_{in} + \bar{S}_{in}^T \bar{T}_{in}$$

with $S_{in}(n_{in}, n)$ and $\bar{S}_{in}(n - n_{in}, n)$, partitioning matrices.

It yields:

$$T_{state}T_{out} = \begin{bmatrix} S_{state} \\ S_{out} \end{bmatrix} \bar{T}_{in} \leftrightarrow \bar{T}_{in} = S_{state}^{T} T_{state} + S_{out}^{T} T_{out}$$

with $S_{state}(n_{state}, n)$, $S_{out}(n_{out}, n)$, $S_{out}(n - n_{state} - n_{out}, n)$, partitioning matrices.

- T_{in} gathers known temperatures
- T_{state} gathers temperatures with derivations
- T_{out} gathers temperatures whose values must be estimated
- T_{rem} gathers remaining temperatures without much interest

It can be rewritten as:

$$T = S_{in}^T T_{in} + S_{state}^T T_{state} + S_{out}^T T_{out}$$

Temperature transformation for component delta temperatures e_R (dimension n_R) and e_C (dimension n_{state}):

$$e_R = \mathcal{M}_R T e_C = \mathcal{M}_C T = \mathcal{M}_C S_{state}^T T_{state}$$

Capacitance model $\mathcal{C} = \mathcal{C}(n_{state}, n_{state})$ diagonal and invertible:

$$\mathcal{C}\frac{de_C}{dt} = \varphi_C$$

Heat balance at each node

$$\Gamma_R\varphi_R + \Gamma_C\varphi_C = \Gamma_P\varphi_P$$

with $\Gamma_R = \Gamma_R(n_B, n_R), \Gamma_C = \Gamma_C(n_B, n_{state})$ and $\Gamma_P = \Gamma_P(n_B, n_P)$. Because capacitances are necessary bound to the 0 reference temperature, each capacitance can be involved in one, and only one power balance node. It comes out that Γ_C^T is a selection matrix.

Regarding Γ_R, each resistance can be involved either in one node balance (when connected to the 0 reference temperature) or to 2 node balances, and there is one and only one heat flow per thermal resistance. Therefore, Γ_R is necessary full row rank.

Solving the static problem

Because there are no capacitance in a static problem, we have:

$$\begin{bmatrix} T_{in} \\ \bar{T}_{out} \\ \bar{T}_{ignore} \end{bmatrix} = \begin{bmatrix} S_{in} \\ S_{out} \\ S_{ignore} \end{bmatrix} T$$

$$T = S_{in}^T T_{in} + S_{out}^T T_{out} + S_{ignore}^T T_{ignore}$$

$$e_R = \mathcal{M}_R T$$

$$\mathcal{R}\varphi_R = e_R$$

$$\Gamma_R \varphi_R = \Gamma_P \varphi_P$$

$$\varphi_R = \mathcal{R}^{-1} e_R = \mathcal{R}^{-1} \mathcal{M}_R T$$

Heatflow static equation

We have $\varphi_R = \mathcal{R}^{-1}\mathcal{M}_R T$ with
$T = S_{in}^T T_{in} + S_{out}^T T_{out} + S_{ignore}^T T_{ignore}.$

Therefore,

$$\varphi_R = \mathcal{R}^{-1}\mathcal{M}_R S_{in}^T T_{in} + \mathcal{R}^{-1}\mathcal{M}_R S_{out}^T T_{out} + \mathcal{R}^{-1}\mathcal{M}_R S_{ignore}^T T_{ignore}$$

Using the previous output equation to remove T_{out} yields:

$$\varphi_R = \mathcal{R}^{-1}\mathcal{M}_R(I_n - S_{out}^T(\Psi_R S_{out}^T)^+\Psi_R)S_{in}^T T_{in} + \mathcal{R}^{-1}\mathcal{M}_R S_{out}^T(\Psi_R S_{out}^T)^+\Gamma_P\varphi_P$$

Dynamic problem solving

We have:

$$e_R = \mathcal{M}_R S_{in}^T T_{in} + \mathcal{M}_R S_{state}^T T_{state} + \mathcal{M}_R S_{out}^T T_{out}$$

$$e_C = \mathcal{M}_C \bar{S}_{int}^T S_{state}^T T_{state}$$

$$\mathcal{R}\varphi_R = e_R$$

$$\mathcal{C}\frac{de_C}{dt} = \varphi_C$$

$$\Gamma_R \varphi_R + \Gamma_C \varphi_C = \Gamma_P \varphi_P$$

With $\Psi_R = \Gamma_R \mathcal{R}^{-1} \mathcal{M}_R$ and $\Psi_C = \Gamma_C \mathcal{C} \mathcal{M}_C$, it can be rewritten as:

$$\Psi_R S_{in}^T T_{in} + \Psi_R S_{state}^T T_{state} + \Psi_R S_{out}^T T_{out} + \Psi_C S_{state}^T \frac{dT_{state}}{dt} = \Gamma_P \varphi_P$$

Output dynamic equation

Multiplying by $\overline{S_{state}\Psi_C^T}$ on the left to remove the derivatives and get the output equation leads to:

$$\Psi_R S_{in}^T T_{in} + \Psi_R S_{state}^T T_{state} + \Psi_R S_{out}^T T_{out} + \Psi_C S_{state}^T \frac{dT_{state}}{dt} = \Gamma_P \varphi_P$$

$$\overline{S_{state}\Psi_C^T}\Psi_R S_{out}^T T_{out} = -\overline{S_{state}\Psi_C^T}\Psi_R S_{state}^T T_{state} - \overline{S_{state}\Psi_C^T}\Psi_R S_{in}^T T_{in} + \overline{S_{state}\Psi_C^T}\Gamma_P \varphi_P$$

with $\Phi = \overline{S_{state}\Psi_C^T}\Psi_R S_{out}^T$.
It leads to the output equation:

$$T_{out} = -\Phi^+ \overline{S_{state}\Psi_C^T}\Psi_R S_{state}^T T_{state} - \Phi^+ \overline{S_{state}\Psi_C^T}\Psi_R S_{in}^T T_{in} + \Phi^+ \overline{S_{state}\Psi_C^T}\Gamma_P \varphi_P$$

State equation

Let's now focus on the state equation:

$$\Psi_C S_{state}^T \frac{dT_{state}}{dt} = -\Psi_R S_{state}^T T_{state} - \Psi_R S_{in}^T T_{in} - \Psi_R S_{out}^T T_{out} + \Gamma_P \varphi_P$$

The derivative must be isolated by multiplying by $(\Psi_C S_{state}^T)^+$:

$$\frac{dT_{state}}{dt} = -(\Psi_C S_{state}^T)^+ \Psi_R S_{state}^T T_{state} - (\Psi_C S_{state}^T)^+ \Psi_R S_{in}^T T_{in} - (\Psi_C S_{state}^T)^+ \Psi_R S_{out}^T T_{out} + (\Psi_C S_{state}^T)^+ \Gamma_P \varphi_P$$

Let $\Pi = (\Psi_C S_{state}^T)^+ (I_n - \Psi_R S_{out}^T \Phi^+ \overline{S_{state} \Psi_C^T})$, T_{out} is removed using the observation equation:

$$\frac{dT_{state}}{dt} = -\Pi \Psi_R S_{state}^T T_{state} - \Pi \Psi_R S_{in}^T T_{in} + \Pi \Gamma_P \varphi_P$$

Heatflow dynamic equation

We have $\varphi_R = \mathcal{R}^{-1}\mathcal{M}_R T$ with $T = S_{in}^T T_{in} + S_{state}^T T_{state} + S_{out}^T T_{out}$.

Therefore,

$$\varphi_R = \mathcal{R}^{-1}\mathcal{M}_R S_{state}^T T_{state} + \mathcal{R}^{-1}\mathcal{M}_R S_{in}^T T_{in} + \mathcal{R}^{-1}\mathcal{M}_R S_{out}^T T_{out}$$

Using the output equation to remove T_{out} yields:

$$\varphi_R = \mathcal{R}^{-1}\mathcal{M}_R(I + S_{out}^T\Phi^+\overline{S_{state}\Psi_C^T}\Psi_R)S_{state}^T T_{state} \dots$$

$$+\mathcal{R}^{-1}\mathcal{M}_R(I + S_{out}^T\Phi^+\overline{S_{state}\Psi_C^T}\Psi_R)S_{in}^T T_{in} \dots$$

$$-\mathcal{R}^{-1}\mathcal{M}_R S_{out}^T\Phi^+\overline{S_{state}\Psi_C^T}\Gamma_P\varphi_P$$

We have also $\varphi_C = \mathcal{C}\mathcal{M}_C S_{state}\frac{dT_{state}}{dt}$. Introducing the state differential equation yields:

$$\varphi_C = -\mathcal{C}\mathcal{M}_C S_{state}^T\Pi\Psi_R S_{state}^T T_{state} - \mathcal{C}\mathcal{M}_C S_{state}^T\Pi\Psi_R S_{in}^T T_{in} + \mathcal{C}\mathcal{M}_C S_{state}^T\Pi\Gamma_P\varphi_P$$

Selection of output

We had:

$$T_{state}T_{out} = \begin{bmatrix} S_{state} \\ S_{out} \end{bmatrix} \bar{T}_{in} \leftrightarrow \bar{T}_{in} = S_{state}^T T_{state} + S_{out}^T T_{out}$$

with $S_{state}(n_{state}, n)$, $S_{out}(n_{out}, n)$, $S_{out}(n - n_{state} - n_{in}, n)$, partitionning matrices.

We are interested in $T_{sel} \subset T_{out}$ i.e. $T_{sel} = S_{sel}Tout$ with $S_{sel}(n_{sel}, n - n_{state} - n_{in})$

Computing CO2 concentration from heat flows

CO2 concentration

A multi-zonal model is going to be used[1]. This model divides the building into several zones and represents the air exchanges between the zones by resistances. It relies on the fact that, considering a 1 hour time slot, the dynamics of airflows are much faster than the dynamics of CO_2 concentrations. Therefore, airflows are steady airflows during each time slot.

Airflows are composed of 2 components: the exchanged airflows $Q^D_{i,j}$, which are bi-directional, and the transported airflows $Q^T_{i,j}$, which are uni-directional, with $Q_{i,j} = Q^D_{i,j} + Q^T_{i,j}$.

1. Haijing Wang, Lihua Xie, Shuai Liu and Juanjuan Xu. A model-based control of CO2 concentration in multi-zone air-conditioning systems, 12th IEEE International Conference on Control & Automation (ICCA) Kathmandu, Nepal, June 1-3, 2016

No matter the components in airflows, the mass of CO_2 in a zone i is given by: $m_i = \rho C_i V_i$, where ρ stands for volumic mass of CO_2, C_i for the CO_2 concentration in zone i and V_i the air volume in zone i. In presence of n_i persons, there is a continuous CO2 production of $S_i = n_i S_{breath}$ in the zone but pressure is not affected considering that one person is producing permanently $S_{breath}\ ppm.m^3/s$ in breathing.

Because pressures are considered in steady state, the permanent air flows are constant too. The variation of m_i in a zone i satisfies:

$$\frac{d}{dt}C_i = \frac{S_{breath}}{V_i}n_i + \frac{C_{out} - C_i}{V_i}Q_{i,out} + \sum_{j/j\neq i}\frac{C_j - C_i}{V_i}Q_{i,j}$$

Generally speaking, a simulated room i connected to a set of simulated rooms $j \in N_i$:

$$\forall i, \frac{d}{dt}C_i = -\left(\frac{Q_{i,out}}{V_i} + \sum_{j \in N_i} \frac{Q_{i,j}}{V_i}\right) C_i + \frac{Q_{i,out}}{V_i} C_{out} + \sum_{j \in N_i} \frac{Q_{i,j}}{V_i} C_j + \frac{S_{breath}}{V_i} n_i$$

with $Q_{i,j} = Q^D_{i,j} + Q^T_{i,j}$ where:

- $Q^D_{i,j} = -Q^D_{j,i}$
- $Q_{i,out} = \sum_{j \in N_i} Q^T_{i,j}$

Let's introduce the steady state pressure P_i in zone i. The transported airflows have to satisfied:

$$\forall i, \forall j \in N_j, P_i - P_j = R_{i,j} Q_{i,j}^T$$

$$\forall i, P_i - P_{out} = R_{i,out} Q_{i,out}^T$$

Because the zone pressures are not easy to guess, nor the resistances, we do not consider transportation phenomena.

Self-check questionnaire

12 questions — control in buildings

Choose the best answer each time.

(In class: discuss in pairs; answers on the following slides.)

Q1 — Case-based reasoning (CBR)

Case-based reasoning on the slides avoids a full explicit model by:

- A. Archiving cases (context + actions + effects), comparing distances, and adapting actions from neighbour cases when the case base is sufficiently complete
- B. Fitting only a static U_{bat} with no historical data
- C. Using HDD/CDD exclusively for control
- D. Replacing all measurements with PMV votes

Q2 — System variables vs data

In buildingenergy, system variables differ from data because:

- A. Variables are symbolic model quantities (e.g. `TZroom1`, `CCO2room1`); data are measured or generated time series linked via bindings
- B. They are always the same object with identical names
- C. Data exist only in the thermal network, not in control
- D. Variables cannot be bound to weather files

Q3 — Room naming (example)

For a zone named `room1`, the slides list system variables including:

- A. `TZroom1` (air temperature), `PZroom1` (power in), `PCO2room1` / `CCO2room1` (CO_2 production / concentration)
- B. Only `Uroom1` and `Rroom1`
- C. HDD and CDD only
- D. Compressor enthalpy `H_comp`

Q4 — State model maker

BuildingStateModelMaker on the slides:

- A. Builds a time-varying discrete state model (thermal + CO_2), recomputed when needed but cached (dynamic programming) to save time
- B. Is computed once for all years with no ventilation dependence
- C. Replaces bindings and weather data
- D. Only implements ARX black-box models

Q5 — Control ports

Control ports (e.g. `AirflowCport`, `TemperatureSetpointCport`) are used to:

- A. Restrict which control values are allowed each hour (e.g. ventilation levels vs mode/presence)
- B. Download weather from GPS automatically only
- C. Define wall layer materials in `SideFactory`
- D. Compute Morris elementary effects only

Q6 — Control time step

Building control simulation on the slides typically uses a time slot of:

- A. 1 hour (state model integrated over 1 h; controllers act per slot)
- B. 1 second only
- C. 1 year per step
- D. No discrete time index k

Q7 — Instantaneous vs delayed temperature control

For output Y^j (temperature) via input δU^i (HVAC power), instantaneous control needs roughly:

- A. $S_j^T D_k^i \neq 0$; one-slot-delay control uses $S_j^T D_{k+1}^i = 0$ and $S_j^T C_{k+1} B_k^i \neq 0$ (common HVAC case)
- B. Always $D^i = 0$ for both schemes
- C. No state vector X_k
- D. COP Carnot only, no state space

Q8 — SignalGenerator

SignalGenerator is described as a factory to build:

- A. Profiles (occupancy, ventilation, set-points) by daily/seasonal rules and mergers (`min` / `max` , dominance left/right/…)
- B. Only steady-state heat flows with no time series
- C. Refrigerant Mollier diagrams
- D. Thermal bridge FEM meshes only

Q9 — Thermal network → state model (partitioning)

Selection / partitioning matrices S_{in}, S_{state}, S_{out} serve to:

- A. Split node temperatures into known inputs, states with derivatives, and outputs to solve for
- B. Replace $\mathcal{R}$ and $\mathcal{C}$ with identity always
- C. Eliminate all heat balances
- D. Convert COP to SCOP only

Q10 — Static vs dynamic thermal solve

In the static problem (no capacitance dynamics), the slides set:

- A. No $\mathcal{C}\, de_C/dt$ terms; solve algebraic heat balance for temperatures/flows
- B. Full state equation with dT_{state}/dt unchanged from dynamic case
- C. Only CO_2 without temperatures
- D. Only eigenvalues of A

Q11 — CO_2 model (time scale)

The multi-zone CO_2 model assumes over each 1 h slot:

- A. Airflows are steady (fast air dynamics) while CO_2 concentration evolves more slowly
- B. CO_2 is constant; airflows vary every second within the hour
- C. No ventilation between zones
- D. Pressure-driven transport is always fully modeled

Q12 — Control manager role

The Manager in the GreenCAB2 example:

- A. Instantiates control ports and controllers, and sets one control value per time slot (e.g. ventilation, power) from rules/state
- B. Only prints Morris μ and σ
- C. Replaces the state-space matrices with ARX
- D. Computes HDD for regulatory ThBAT only

Answer key (1/2)

Q	Answer	Short rationale
1	A	CBR: cases, distances, completeness α, adapted actions.
2	A	Symbolic variables vs bound data series.
3	A	TZ, PZ, PCO2, CCO2 naming pattern.
4	A	Hourly TV SS model + cache / DP.
5	A	Ports restrict feasible controls.
6	A	1 h simulation/control step.

Answer key (2/2)

Q	Answer	Short rationale
7	A	Instantaneous vs next-slot HVAC control conditions.
8	A	Profiles/set-points via mergers.
9	A	Partition $T_{\text{in}}, T_{\text{state}}, T_{\text{out}}$.
10	A	Static: no capacitance ODE.
11	A	Steady Q over 1 h; slower C_i dynamics.
12	A	Manager + ports + per-slot controls.