prodyn Documentation

Release 0.1

Dannis Atabay

May 28, 2017

Contents

Maintainer Dennis Atabay, <dennis.atabay@tum.de>

Organization Institute for Energy Economy and Application Technology, Technische Universität München

Version 0.1

Date May 28, 2017

CHAPTER 1

Contents

This documentation contains the following pages:

Overview

An overview explains the basic procedure of the dynamic programming implementation in the random example. It also introduces files, which are involved in the implementation process, and clarifies main functions inside these files.

Simplified diagram of the process and connection between documents involved in it are shown in the Figure 1.

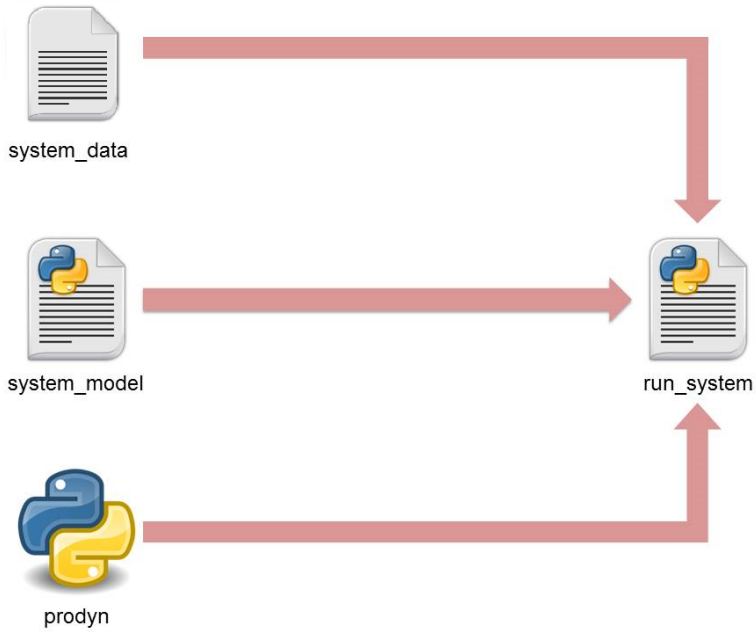

Fig. 1.1: Figure 1: Scheme of the dynamic programming implementation

According to the diagram any implemetation consists of four files. Three of them (**system_data**, **system_model** and **run_system**) are specified and should be created for each current example. The fourth one (**prodyn**) is autonomous and can be used with any example without modifications.

System Data

System_data stores an information about a system, which operation should be controlled in an optimal way. All data is split between four parts: **Time-Series**, **Constants**, **DP-States** and **DP-Decisions**. Each of this part is described below. `Chp_data` written in excel-form is taken as an example.

Note: **System_data** doesn't have to be always written in excel-form. Other formats are also possible. In addition, **system_data** doesn't have to exist in the form of file. It can be typed by the user through the code or interface and so on.

Time-Series

A series of values for parameters, which described the system, is shown here. Values are obtained at successive times and with equal intervals between them. Small part of **Time-Series** from `chp_data` is illustrated in the Figure 2.

	A	B	C	D	E
1	Time	el_demand	heat_demand	el_cost	el_feed-in
2	1	0,24	5,07	0,16	0,06
3	2	0,71	3,55	0,15	0,06
4	3	0,23	2,47	0,15	0,06
5	4	1,06	2,43	0,14	0,06
6	5	1,23	3,05	0,14	0,06
7	6	1,59	3,28	0,14	0,06
8	7	1,27	3,71	0,14	0,06
9	8	2,61	5,66	0,13	0,06
10	9	2,66	3,98	0,13	0,06

Fig. 1.2: Figure 2: **Time-Series** from `chp` example

Constants

This sheet keeps all values for parameters, which doesn't change during any operation of the system.

DP-States

The part of the system, which operation should be optimized, is characterized by a number of states. Each state has min, max allowable values and number of steps between them. All these data is stored in **DP-States** sheet. **DP-States** for `chp` example is shown in the Figure 3.

As seen from the Figure 3 the system has two states. State of the **battery** can take values from **0** to **5** with a step equaled to **0,1**. Similarly **heat-storage** is changing between **0** and **10** with a step **0,1**.

	A	B	C	D
1	state	xmin	xmax	xsteps
2	battery	0	5	50
3	heat-storage	0	10	100

Fig. 1.3: Figure 3: **DP-States** from `chp` example

DP-Decisions

An operation of the system for every timestep can be influenced by one of the specific decisions, which are written in **DP-Decisions** sheet. In other words, all possibilities for system control are written here. Figure 4 illustrates decisions for the same `chp` example.

	A	B
1	Number	Decisions
2	1	off
3	2	on

Fig. 1.4: Figure 4: **DP-Decisions** from `chp` example

As seen above `chp` example has only two decisions: **off** and **on** operation of the combined heat and power plant.

System Model

System_model is a file, which is written in python and should be created specifically for the current system. Generally, it contains two functions: **read_data** and **system**. The second one, which describes the transition of the system from one timestep to another one, is the main part of this file. Figure 5 gives simplified illustration of this transition.

System's condition at timestep **i** is defined by an array **X**, which is built from the **DP-States** data. The process of **X** formation is fully described in one of the next subchapters *prepareDP*. **System** function calculates the transition from **i** to **j** in dependance of each decision from the list of possible ones **U**. Results of the calculation are an array **Xj**, which describes the condition of the system at timestep **j**, and the **cost** of the transition for each possible decision from **U**.

For your own implementation your own **system** function, which characterizes the transition from **i** to **j**, should be written. Groups of fixed inputs and outputs of this function are presented in the Figure 6.

Inputs of the **system** are:

- **u** - decision from the possible ones in list **U**.
- **x** - array containing any possible condition of the system.
- **t** - actual timestep **i**.
- **cst** - constants needed for calculation, which are taken from **Constants** in excel file **system_data**.
- **srs** - values of needed timeseries taken from **Time-Series**.
- **Data** - pandas dataframe, which keeps information about previous transitions. This is main return of the **prodyn** file.

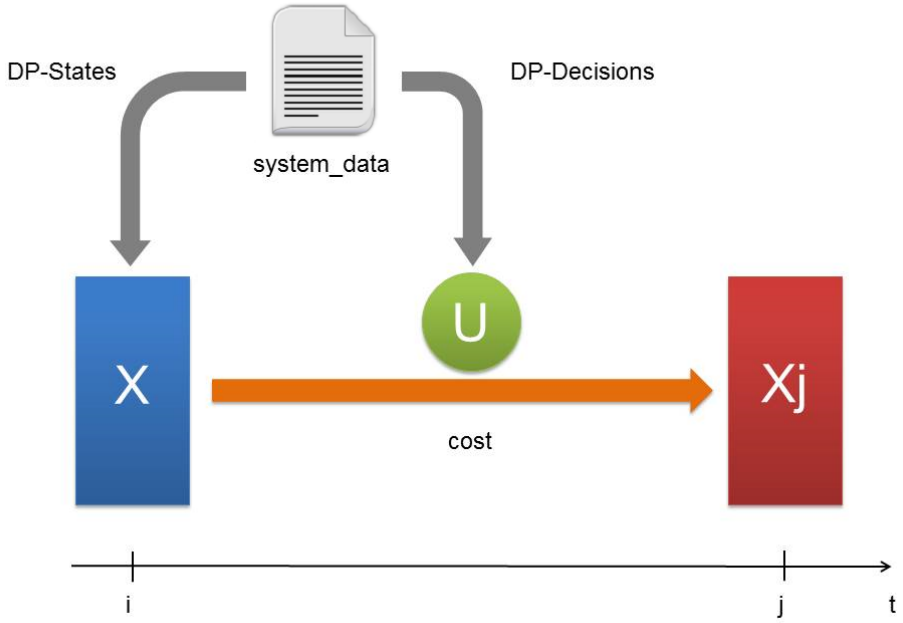

Fig. 1.5: Figure 5: System transition from timestep **i** to **j**

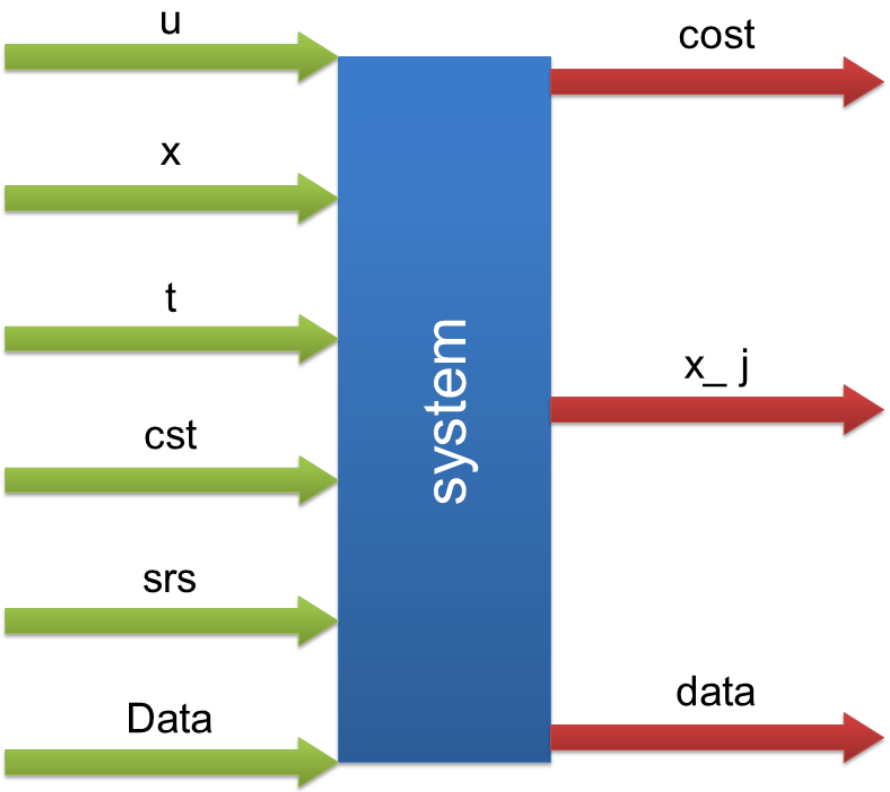

Fig. 1.6: Figure 6: **System's** necessary inputs and outputs for the description of the transition from timestep **i** to **j**

Returns of the **system** are:

- **cost** - costs of the transition from **i** to **j**.
- **x_j** - array with condition of the system at timestep **j**, which is formed due to decision **u**.
- **data** - intermediate pandas dataframe containing additional information about the transition from **i** to **j**. Needed for **Data** formation in **prodyn**.

The remaining function **read_data** is responsible for reading **system_data** and returning the following parameters, which partially form the group of **inputs** of the **system**: **cst**, **srs**, **U** and **states**. **Read_data** for all of the examples presented in the documentation is written for reading **system_data** in excel-from. However, the format or form of **system_data** can be absolutely various. In these cases **read_data** should be rewritten.

Note: Regardless of **system_data** format and it's possible absence **cst**, **srs**, **U** and **states** must always be identified according to the following standarts:

- **cst**, **srs** and **states** - Pandas Data Frames;
- **U** - 1-d numpy array.

Prodyn

Prodyn is an autonomous file, which is written in python and can be used with any example. This is the core and driving force of every dynamic programming implementation. Three main **prodyn's** functions are described below in details.

prepare DP

The goal of **prepare_DP** is a creation of several arrays, which will be used subsequently. The simplified procedure of the creation for 1 and 2 states random examples is presented in the Figure 7.

The table with states of the system, which is stored in **DP-States** sheet of *system_data*, plays a role of input for the **prepare_DP**. Three new arrays are the main returns of the function:

- **X** is an array containing every possible condition of the system. It's size depends on the number of system's states. For example, any condition of the system with 2 states is always characterized by two variables and **X** is, consequently, 2d.
- **Xidx** stores numbers corresponded to every system's condition. This array is always 1-d.
- **XX** is an array of arrays, from which **X** is built. In other words, **X** is the cartesian product of **XX**.

DP forward

Current function realizes dynamic programming algorithm in forward direction. Forward means that simulation starts from **t_start** and ends on **t_end**. A diagram, which helps to understand the process inside **DP_forward**, is shown in the Figure 8.

Let's imagine, that we have arbitrary system with 2 states. It has 4 possible conditions, all of them are defined by **X**. **U** contains only two possible decisions **{u1; u2}**, their influence on the system's condition

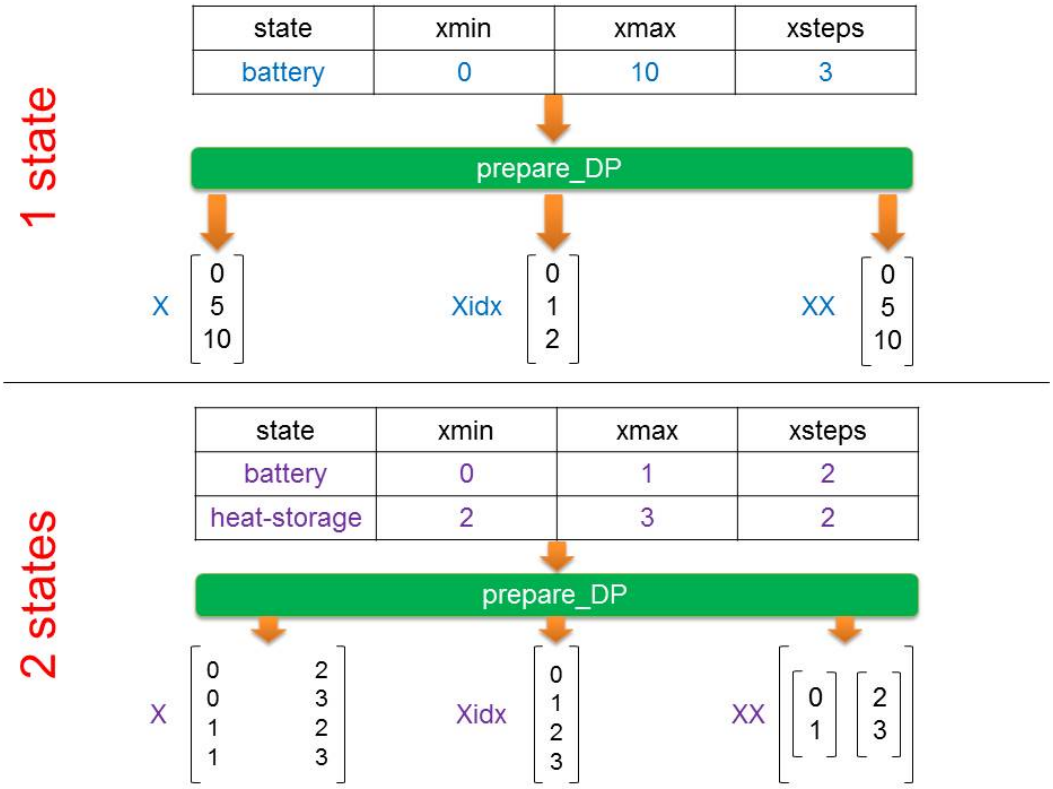

Fig. 1.7: Figure 7: Working principle of **prepare_DP** function

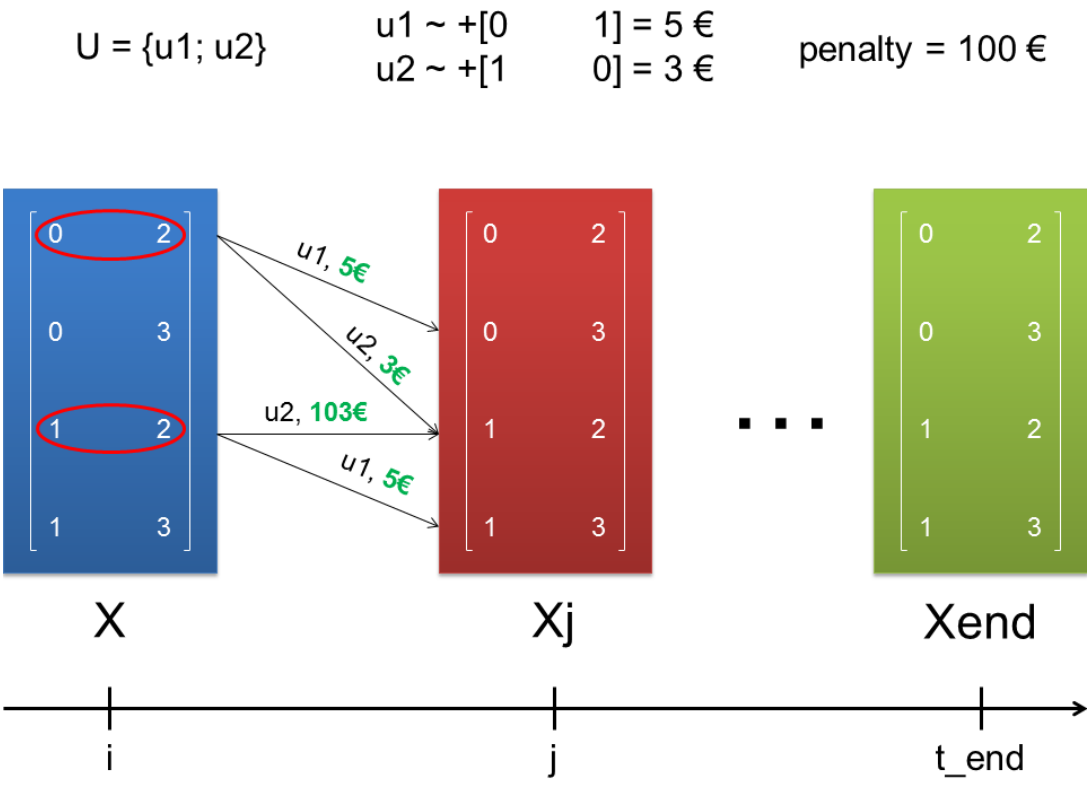

Fig. 1.8: Figure 8: Processes inside **DP_forward** function

and their **costs** are described at the top side of Figure 8. **Penalty** is an additional cost for the transition in cases, where system is pushed by **u1** or **u2** to the condition, which system can't achieve.

Note: The way **U** influences on the system, value of **costs** and possible **penalties** are always described in *system_model*. Due to this information **prodyn** knows how to make the transition from one step to another one for any possible condition of the system.

In the Figure 8 the transition for **0th** and **2nd** sytem's condition from timestep **i** to timestep **j** is shown. An idea of **penalty** is clarified very well, where **u2** is applied on the **2nd** condition. **[1 2]** is forced by **u2** to be **[2 2]**, which is impossible. In such way **prodyn** runs the system through the whole timesteps until **t_end** is reached. The one and only return of the **prodyn** is called **Data**, which structure is presented in the Figure 9.

Data

t	Xidx_end	U	cost	other parameters
t_start	0 1 2 : N			
t_start+1	0 1 2 : N			
…	…	…	…	…
t_end	0 1 2 : N			

Fig. 1.9: Figure 9: **Data** - return of the **prodyn**

Data is a pandas dataframe with two indices. **Xidx_end** represents system's conditions at the end of last timestep and **t** collects all timesteps. With **Data** we can see which decisions should be applied to the system on each timestep for achieving the desired condition at the end of simulation. There are also other system's parameters, which help to analyze the results of simulation

DP backward

DP_backward realizes dynamic programming algorithm in a same way as **DP_forward** does, but in the opposite direction. Simulation starts from **t_end** and goes backward until **t_start** will be reached.

Run System

Run_system file is a place, where all other three documents meet and interact between each other. At the end, when simulation of the system with dynamic programming algorithm is finished, an optimal way of control it is searched. All parameters, which shows the most optimal path (the cheapest path or the path with empty storage at the last timestep), are extracted from the **Data** and plotted for **results** visualisation. The procedure is illustrated in the Figure 10.

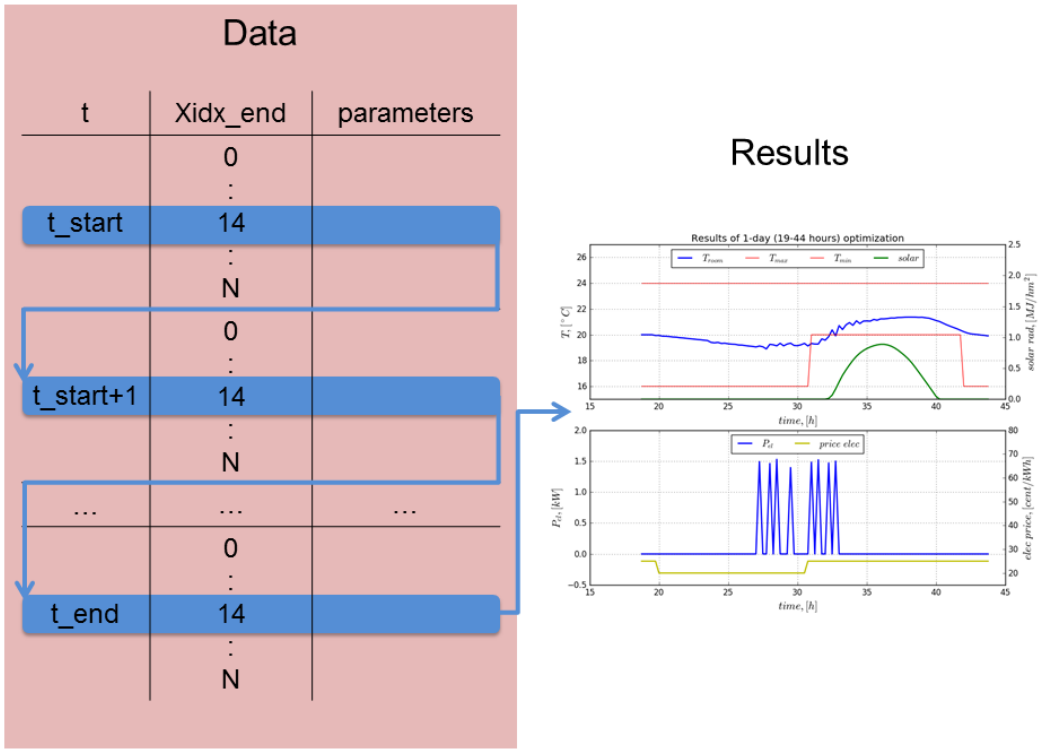

Fig. 1.10: Figure 10: The procedure of achieving results

Examples

The examples given in this chapter show how to implement dynamic programming algorithm. First example of the system (**building**) is presented very detailed. For other systems only brief description is given. However, very detailed comments through all codes will help to achieve deeper understanding. Results of optimal system control can be seen after simulations of the **run_system** codes.

Building

Description

A system in **building** example contains a model of the real building (pre-trained Neural Network) and a heat pump. The goal of the optimization is to keep room temperature **Troom** inside the range of allowed values **[Tmin; Tmax]** in a cost-efficient way. Simulation covers one day (19-44 hours) with 15 min time resolution. The picture in the Figure 11 visualizes current system.

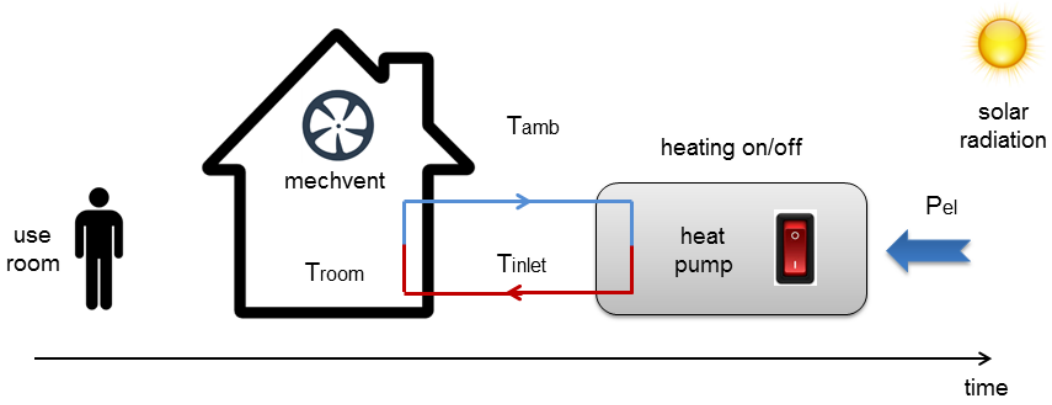

Fig. 1.11: Figure 11: Illustration of the **building** example

Dynamic Programming algorithm for optimal control of the **building** is realized with using four following files:

- **building_data.xlsx** - stores information about the system.
- **building_model.py** - reads system's data and describes transition from one timestep to another.
- **prodyn.py** - realizes dynamic programming algorithm.
- **run_building_forward.py** - runs the simulation and finds the optimal system's control.

Run_building_forward.py and **building_model.py** are described in detail below.

run_building_forward.py

There is a script of the `run_building_forward.py` (one from four dynamic programming files) is explained step by step for better understanding.

```
import numpy as np
import matplotlib.pyplot as plt
import pyrenn as prn
```

Three packages are included:

- numpy is the fundamental package for scientific computing with Python;
- matplotlib.pyplot is a plotting library which allows present results in a diagram form quite easily;
- pyrenn is a recurrent neural network toolbox for Python.

```
import building_model as model
import prodyn as prd
```

Then **building_model** and **prodyn** (two other files of dynamic programming) are imported. They assigned as **model** and **prd** respectively.

```
file = 'building_data.xlsx'
```

Gives the path to the excel-file **building_data** containing data about the current system. This is the last file of dynamic programming.

```
cst,srs,U,states = model.read_data(file)
srs['massflow'] = 0
srs['P_th'] = 0
srs['T_room'] = 20
```

Defines constants **cst**, timeseries **srs**, list of possible decisions **U** and parameters **states**, which characterize each possible **building's** state, by reading the **building_data** file. Process of reading is realized due to **read_data** function hidden in the **building_model** (model) file. To timeseries **srs** written from **building_data** some extra data is added.

```
timesteps=np.arange(cst['t_start'],cst['t_end'])
```

Sets a timeframe on which optimization will be realized.

```
net = prn.loadNN('NN_building.csv')
cst['net'] = net
```

Defines a model **net** of the real building (pre-trained Neural Network) and saves it to the constants **cst**.

```
xsteps=np.prod(states['xsteps'].values)
J0 = np.zeros(xsteps)
idx = prd.find_index(np.array([20]),states)
J0[idx] = -9999.9
```

Creates an array **J0** of initial terminal costs. **J0** will be changed from transition to transition according to list of possible decisions **U** and will keep all costs. Due to stored infromation in **J0** optimal control of the **building** can be found.

```
idx = prd.find_index(np.array([20]),states)
J0[idx] = -9999.9
```

Shifts the initial postition to index with temperature equaled to 20 degrees.

```
system=model.building
```

Defines function **building** from **building_model** for characterization the transition from one timestep to another.

```
result = prd.DP_forward(states,U,timesteps,cst,srs,system,J0=J0,
↪verbose=True,t_verbose=5)
i_mincost = result.loc[cst['t_end']-1]['J'].idxmin()
opt_result = result.xs(i_mincost,level='Xidx_end')
```

Implements dynamic programming algorithm for the chosen timeframe and saves all data to the **result**. Then finds index for cost-minimal path, extracts it from **result** and saves to **opt_result**.

```
best_massflow=opt_result['massflow'].values[:-1]
Troom=opt_result['T_room'].values[:-1]
Pel=opt_result['P_el'].values[:-1]
```

Chooses parameters, which characterize cost-efficient **building** control system, and extracts them from **opt_result**. **Best_massflow** is a schedule, which shows at which timestep heat pump is switched on and at which switched off. **Pel** defines consumed electrical power, **Troom** - room temperature inside the house, which shouldn't be out of the comfort zone **[Tmin; Tmax]**.

```
Troom=np.concatenate((srs.loc[timesteps[0]-4:timesteps[0]-1]['T_room'],
↪Troom))
Pel=np.concatenate((srs.loc[timesteps[0]-4:timesteps[0]-1]['P_th'],Pel))
```

Sums values for timesteps, which were not involved in the optimization, with those, which were extracted from **opt_result**. The remaining part of the code is responsible for plotting chosen and additional parameters. They are presented in the Figure 12.

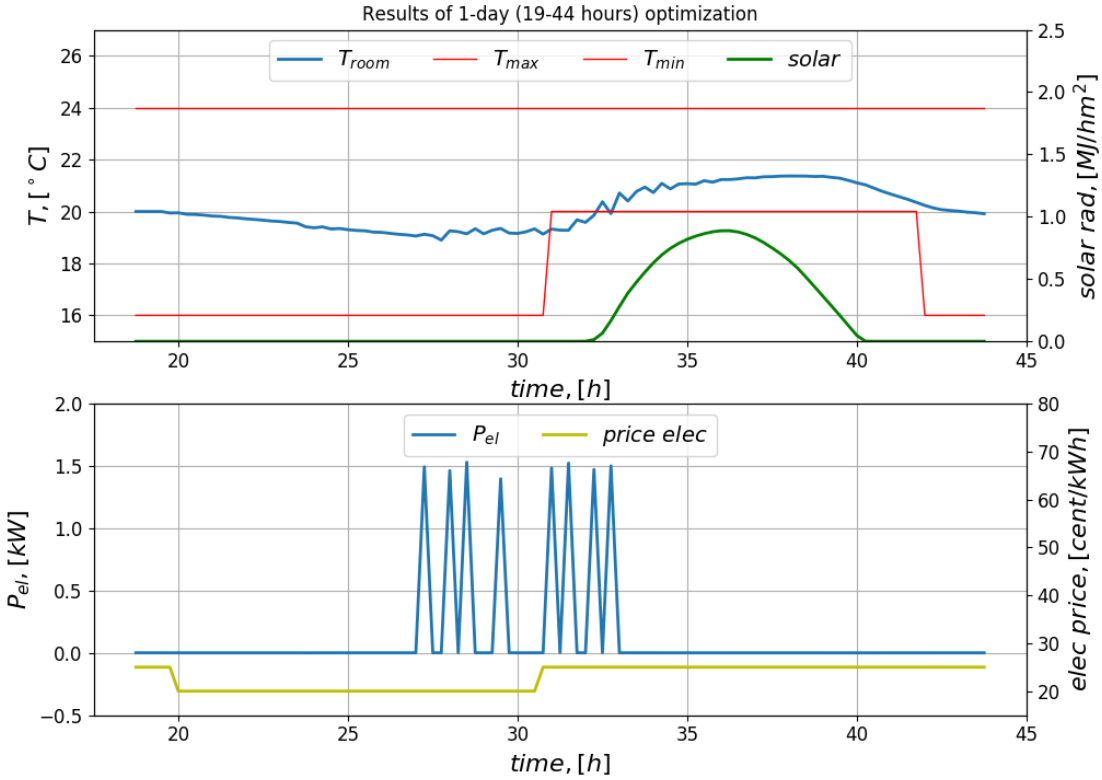

Fig. 1.12: Figure 12: Cost-minimal control of the **building** for keeping **Troom** inside **[Tmin; Tmax]**.

building_model.py

The script of the `building_model.py` is explained step by step for better understanding.

```
import pandas as pd
import numpy as np
import pyrenn as prn
import pdb
```

Three packages are included:

- pandas is a source helping to work with data structure and data analysis;
- numpy is the fundamental package for scientific computing with Python;
- pyrenn is a recurrent neural network toolbox for Python;

- pdb is a specific module, which allows to debug Python codes.

```
def read_data(file):
        xls = pd.ExcelFile(file)
        states = xls.parse('DP-States',index_col=[0])
        cst = xls.parse('Constants',index_col=[0])['Value']
        srs = xls.parse('Time-Series',index_col=[0])
        U = xls.parse('DP-Decisions',index_col=[0])['Decisions'].values
        return cst,srs,U,states
```

Read_data reads data about the **building** system from the excel-file and assigns it to different parameters.

```
def building(u,x,t,cst,Srs,Data):
        l = len(x)
        delay=4
        net = cst['net']
```

Opens function **building** responsible for the system transition. Also identifies the length **l** of the array with possible system states **x**, gives a name to the pre-trained Neural Network (NN) **net** and chooses number of timesteps **delay** for the initial input **P0** and output **Y0** needed for the NN's usage.

```
hour = Srs.loc[t]['hour']
solar = Srs.loc[t]['solar']
T_amb = Srs.loc[t]['T_amb']
user  = Srs.loc[t]['use_room']
T_inlet = Srs.loc[t]['T_inlet']
```

Creates 5 inputs for the input array **P** required for the NN's usage.

```
if u=='heating on':
        massflow = cst['massflow']
elif u=='heating off':
        massflow = 0
```

Defines the 6th and the last input of **P** in dependance of the current decision **u**.

```
P = np.array([[hour],[solar],[T_amb],[user],[massflow],[T_inlet]],dtype =␣
↪np.float)
```

Builds the input array **P** from six inputs for the current timestep **t**.

```
hour0 = Srs.loc[t-delay:t-1]['hour'].values.copy()
solar0 = Srs.loc[t-delay:t-1]['solar'].values.copy()
T_amb0 = Srs.loc[t-delay:t-1]['T_amb'].values.copy()
user0  = Srs.loc[t-delay:t-1]['use_room'].values.copy()
T_inlet0 = Srs.loc[t-delay:t-1]['T_inlet'].values.copy()
```

Creates 5 inputs for the initial input array **P0**, which is also needed for the NN's usage. The length of each input is equaled to the chosen **delay** at the beginning of the function.

```
x_j = np.zeros(l)
P_th = np.zeros(l)
costx = np.zeros(l)
```

Defines array **x_j** for the **building** states after the transition, array **P_th** for thermal power given to the **building** from heat pump and array **costx**, which will contain penalty costs for transition from each

building state in **x** to **x_j** according to current decision **u**.

```
for i,xi in enumerate(x):
        #prepare 6th input for P0 and 2 outputs for Y0
        if t-delay<cst['t_start']:

                #take all values for P0 and Y0 from timeseries
                if Data is None or t==cst['t_start']:
                        T_room0 = Srs.loc[t-delay:t-1]['T_room'].values.
↪copy()
                        P_th0 = Srs.loc[t-delay:t-1]['P_th'].values.copy()
                        massflow0 = Srs.loc[t-delay:t-1]['massflow'].
↪values.copy()

                #take part of values from timeseries and part from big Data
                else:
                        tx = t-cst['t_start']
                        T_room0 = np.concatenate([Srs.loc[t-delay:t-tx-1][
↪'T_room'].values.copy(),Data.loc[t-tx-1:t1].xs(i,level='Xidx_end')['T_
↪room'].values.copy()])
                        P_th0 = np.concatenate([Srs.loc[t-delay:t-tx-1]['P_
↪th'].values.copy(),Data.loc[t-tx-1:t-1].xs(i,level='Xidx_end')['P_th'].
↪values.copy()])
                        massflow0 = np.concatenate([Srs.loc[t-delay:t-tx-
↪1]['massflow'].values.copy(),Data.loc[t-tx-1:t-1].xs(i,level='Xidx_end')[
↪'massflow'].values.copy()])

        #take all values for P0 and Y0 from big Data
        else:
                T_room0 =Data.loc[t-delay:t-1].xs(i,level='Xidx_end')['T_
↪room'].values.copy()
                P_th0 = Data.loc[t-delay:t-1].xs(i,level='Xidx_end')['P_th
↪'].values.copy()
                massflow0 = Data.loc[t-delay:t-1].xs(i,level='Xidx_end')[
↪'massflow'].values.copy()
```

Loop for every possible state of the **building** from **x** opens. All other strings are responsible for prepairing the 6th input **massflow0** for the input array **P0** and two outputs **T_room0**, **P_th0** for the initial output array **Y0**. In dependance of relation between current timestep **t** and **t_start** (initial timestep, from which optimal **builidng** control should be found) these three parameters are created with values from the timeseries **srs** and **Data**, which keeps all information about the previous transitions. There are three cases for the **massflow0**, **T_room0** and **P_th0** creation. Supporting commentaries in this part split these cases.

```
T_room0[-1] = xi
P0 = np.array([hour0,solar0,T_amb0,user0,massflow0,T_inlet0],dtype = np.
↪float)
Y0 = np.array([T_room0,P_th0],dtype = np.float)
```

Corrects last value of **T_room0** and builds initial input **P0** and initial output **Y0** arrays.

```
if np.any(P0!=P0) or np.any(Y0!=Y0):
        #if P0 or Y0 not valid use valid values and apply penalty costs
        costx[i] = 1000*10
        x_j[i] = xi
        P_th[i] = 0
else:
```

```
        x_j[i],P_th[i] = prn.NNOut(P,net,P0=P0,Y0=Y0)

if x_j[i] != x_j[i] or P_th[i] != P_th[i]:
        pdb.set_trace()
```

Runs NN for one timestep. Checks if **P0** and **Y0** are valid. Two outputs of the NN usage are array **x_j**, which keeps all possible states of the **building** after transition, and array **P_th**, which stores data about delivered thermal power from the pump to the building. In the case of mistake a Python debugger will be open. Here the loop for every possible state of the **building** from **x** closes.

```
Tmax = Srs.loc[t]['Tmax']
Tmin = Srs.loc[t]['Tmin']

costx = (x_j>Tmax)*(x_j-Tmax)**2*1000 + (x_j<Tmin)*(Tmin-x_j)**2*1000+costx
```

Selects borders for the allowed **T_room** and calculates penalty costs **costx** if any state of **x_j** is out of chosen borders.

```
x_j=np.clip(x_j,x[0],x[-1])
```

Corrects **x_j**. Values smaller than **x[0]** become **x[0]**, and values larger than **x[-1]** become **x[-1]**.

```
P_el = P_th*T_inlet/(T_inlet-T_amb)
cost = P_el * Srs.loc[t]['price_elec']*0.25 + costx
```

Calculates **cost** of the transition by summing electricity and penalty costs.

```
data = pd.DataFrame(index = np.arange(l))
data['P_th'] = P_th
data['P_el'] = P_el
data['T_room'] = x_j
data['massflow'] = massflow
data['cost'] = cost
data['costx'] = costx

return cost, x_j, data
```

Defines parameters, which will be put in **data** used in **prodyn** file. Returns of the **building** function are costs of the transition **cost**, new array with **building** states **x_j** and **data**.

Building with Storage

The presence of the **heat storage** makes this system different to the **building**. Due to this **building_with_storage** has 4 decisions and 2 states (the room temperature **Troom** and energy content of the storage **E**). The goal and period of simulation are identical to the **building** example. In the Figure 13 schematic picture the **building_with_storage** system is given.

By reason of long-time simulation the results are already given in the folder related to this example.

Note: **Building** and **building_with_storage** examples can be simulated only in **forward** direction.

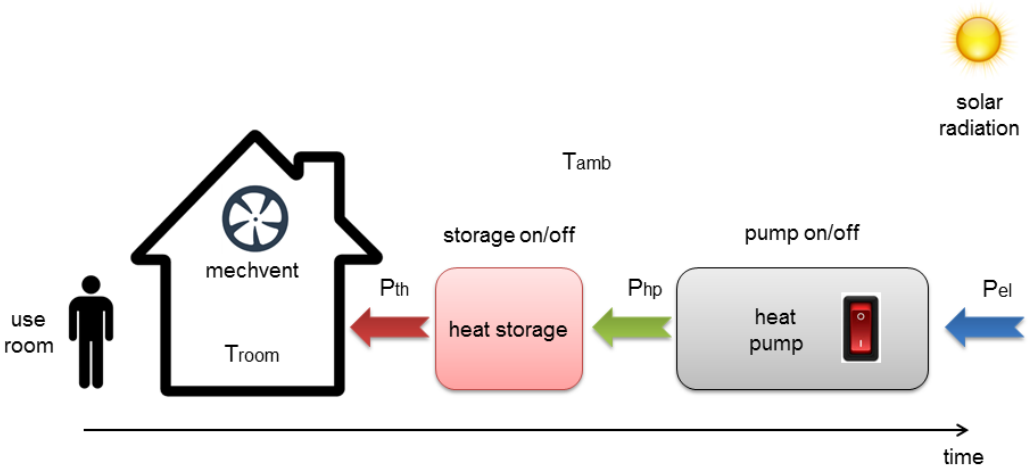

Fig. 1.13: Figure 13: Illustration of the **building_with_storage** example

CHP

Grid, gas-boiler, chp power plant, battery and heat storage are components of the system, which should cover given heat and electical demand. Energy contents of the battery and heat storage are 2 states of the system. When **chp** is **on**, it covers the demand. Surplus of electricity is stored in the battery and sold to the grid. Surplus of the heat is stored in the heat storage. When **chp** is **off**, at first both demands are covered by storages, then by the grid and gas boiler. The goal of optimization is to find the path, where both storages will be empty at the final timestep. Figure 14 shows simplified scheme of the **chp** system.

PV Storage

Photovoltaic system with storage form the system for covering given electrical demand. Energy content of the storage is the only state of the system. List **U** contains three possible decisions. With **normal** system operates without participation of the storage. Possible surplus of the produced by pv power can be saved in the storage with **charge** decision. With **discharge** system tries to cover the residual demand by stored energy. After each possible system's decision **grid load** is checked. This residual power is covered by or fed into the grid. The main goal is to find the result, where the storage is empty at the end. Illustration of the current example is presented in the Figure 15.

Pv_storage_model, which describes the transition from **i** to **j** according to each possible decision **u**, is written in two ways. In first case the transition is applied for the whole **array X**, which characterizes the system. In the second case - for each possible condition of **X**. Calculation for each condition and jump from one to another are realized inside the **loop**.

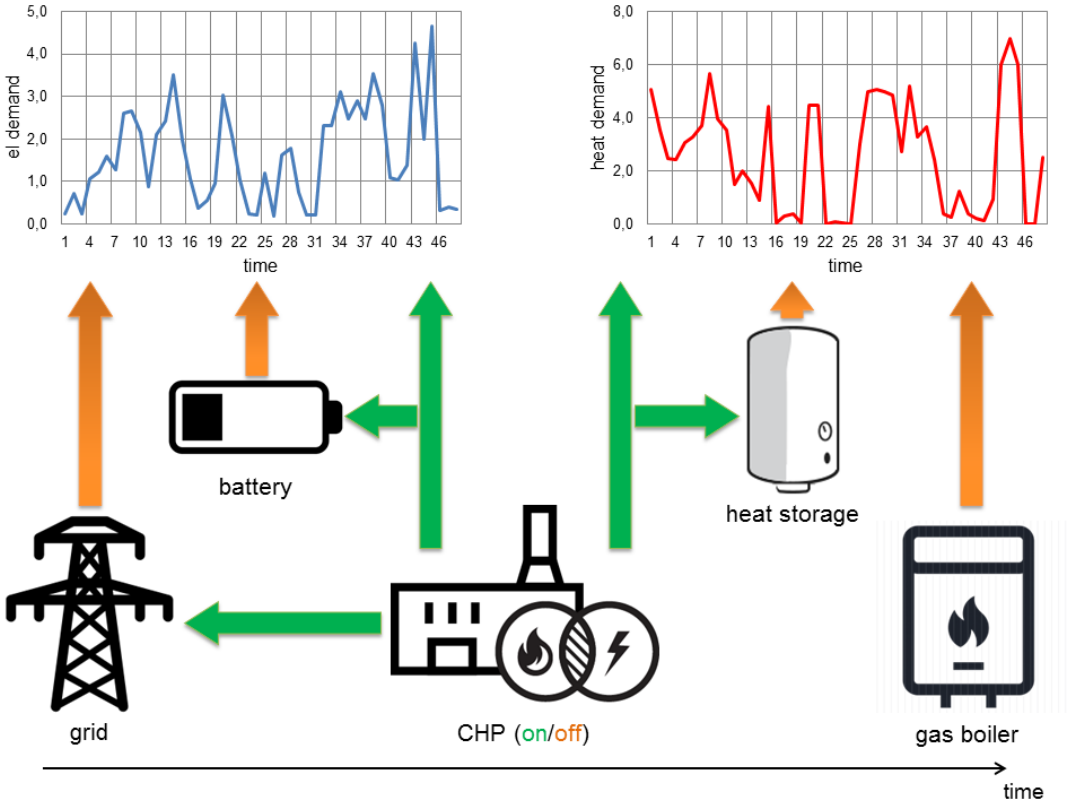

Fig. 1.14: Figure 14: Illustration of the **chp** example

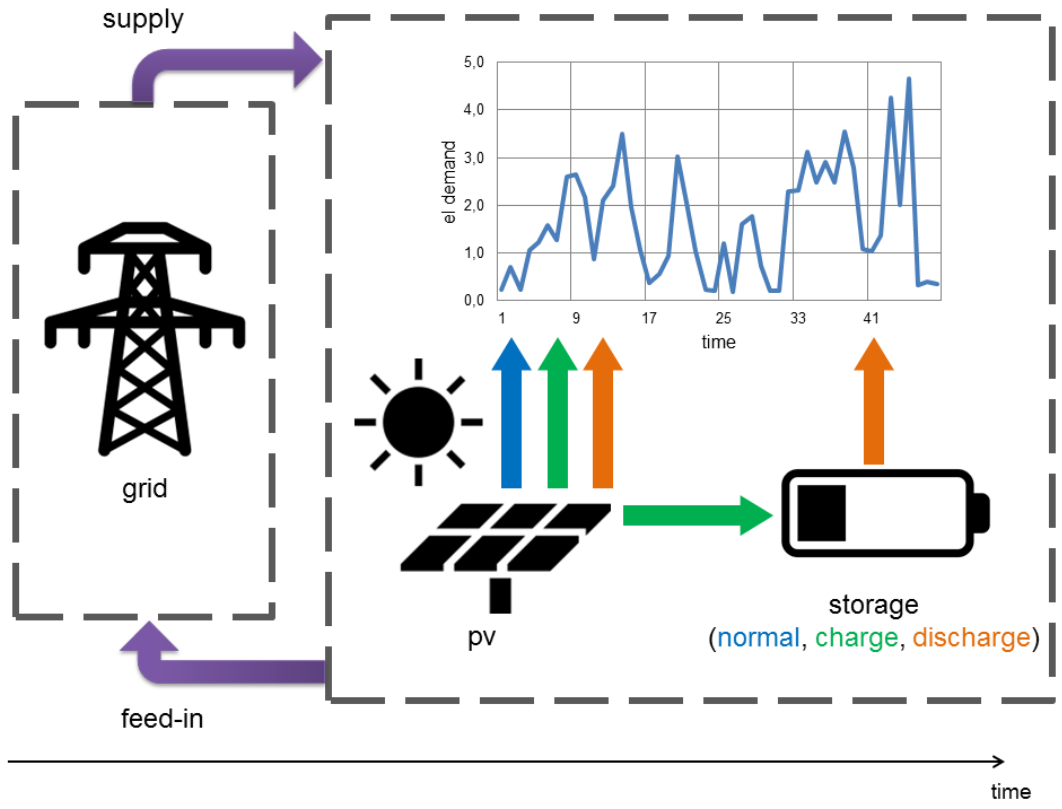

Fig. 1.15: Figure 15: Illustration of the **pv_storage** example

CHAPTER 2

Features

- mutiple states...

CHAPTER 3

Get Started

1. download or clone (with git) this repository to a directory of your choice.
2. Copy the `prodyn.py` file in the `prodyn` folder to a directory which is already in python's search path or add the `prodyn` folder to python's search path (sys.path) (how to)
3. Run the given examples in the *examples* folder.
4. Implement your own system function.

CHAPTER 4

Dependencies (Python)

- numpy for mathematical operations
- pandas only for using the examples

Python Module Index

p

Index

P