


Universitatea Politehnica Bucureşti
Facultatea de Automatică si Calculatoare
Departamentul de Automatică şi Ingineria Sistemelor

LUCRARE DE LICENŢĂ

Clasificare defecte într-o rețea de apă de mari dimensiuni

Absolvent
Cazan Cristian-Claudiu

Coordonator
Conf. dr. ing. Florin Stoican

Bucureşti, 2018

Cuprins

Listă de figuri

Listă de tabele

Listă de algoritmi

1. Detecția și izolarea defectelor

1.1. Definirea defectelor

Defectele sunt simulate modificând parametrul C din ecuația emitter-ului (**??**). Modalitatea prin care se execută în cod simularea unui defect este prin apelarea metodei:

```
1    def set_emitter(self, node_index, emitter_val):
         if self.network.nodes[node_index].node_type is EN_JUNCTION:
3            ENSim._getncheck(self.ENsetnodevalue(node_index, EN_EMITTER, emitter_val
     ))
```

Listing 1.1: Funcție pentru simularea defectelor

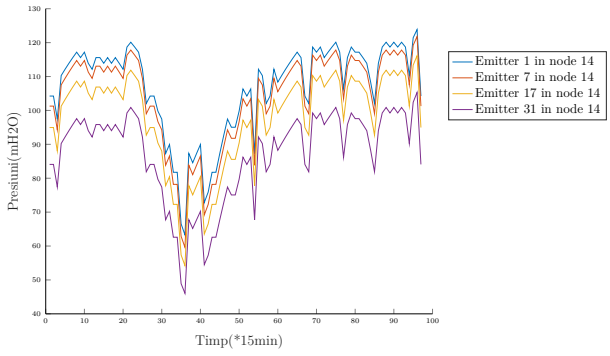
parametrii funcției *set_emitter* sunt:

- node_inde - indexul nodului în care se simulează defectul
- emitter_val - magnitudinea defectului

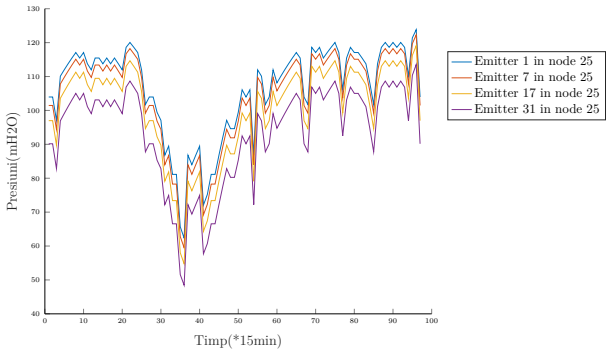
Metoda mai întâi vefică dacă nodul cu indexul $node_index$ reprezintă doar o jonncțiune apoi setează magnitudinea defectului în nodul primit cu ajutorul funcției de bibliotecă **ENsetnodevalue**

1.2. Simulare dinamică pentru defecte în diferite noduri

În continuare vom considera un scenariu de defect pentru rețea care constă în modificarea succesivă a parametrului de proporționalitate din relația de calcul a debitului de emitter (**??**). În imaginile următoare voi considera mai multe magnitudini de defect într-un anumit nod și voi reprezenta grafic răspunsul în timp al rețelei în același nod.



(a) Profile cu defect în nodul 14



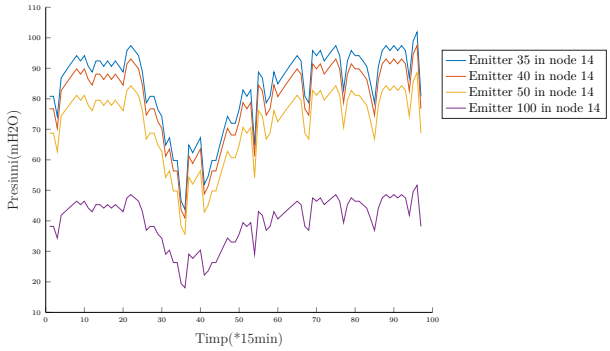
(b) Profil cu defect în nodul 25

Figura 1.1.: Rezultate simulări defecte ușoare

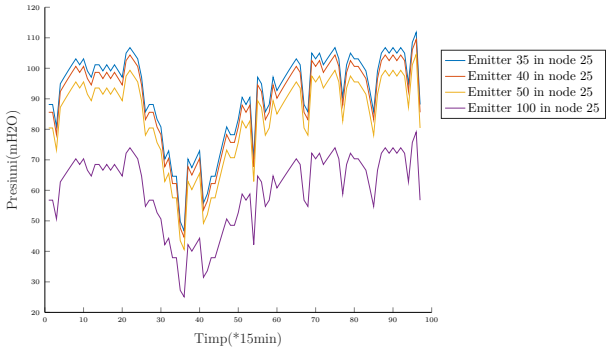
După cum se poate observa în imaginile 1.1 variația emitter-ului într-un nod produce în mod evident o modificarea a modului comun al caracteristicii $timp - presiune$. Din punctul de vedere al magnitudinolor de simulare pentru defecte, am considerat 2 clase de defecte, anume:

- defecte ușoare (soft faults) - cu valorile coeficientului de emitter mai mici de 35
- defecte puternice (hard faults) - cu valorile emitter mai mari de 35

Cele din urmă produc și modificări ale caracteristicii dinamice, introducând distorsiuni sau aplatizări ale mărimilor măsurate. Reprezentarea defectelor hard este reprezentată în figurile de mai jos:



(a) Profile cu defect puternic în nodul 14



(b) Profil cu defect puternic în nodul 25

Figura 1.2.: Rezultate simulări defecte puternice

Se observă de exemplu că pentru o valoare a emitter-ului de 100 caracteristica dinamică este deja modificată din cauza scurgerilor puternice din nod.

Este relevantă împărțirea defectelor în mai multe clase de magnitudini pentru a putea valida un model de clasificare. Spre exemplu este normal să se întrebe dacă un model antrenat pe baza unui set date corespunzător unor magnitudini normale $C \in (0, 35)$ poate da rezultate semnificative pentru un set de date cu magnitudini ale emitter-ului puternice $C \geqslant 35$.

1.3. Preprocesarea datelor

În urma extragerii datelor din rețea este extrem de importantă etapa de prelucrare și preprocesare a datelor. Domeniul de preprocesare a datelor este unul extrem de vast și important în domeniul de învățare automată (engl. Machine Learning) și procesare de semnal. Preprocesarea datelor este etapa în care datele de intrare pentru un algoritm sunt aduse la o formă optimă pentru desfășurarea procesului impus, de exemplu în domeniul clasificării este important ca algoritmii să primească date care să fie scalate într-un anumit domeniu, pentru a asigura convergența[5], [2]. Alegerea metodei de preprocesare este strâns legată de tipul de date disponibile și de starea acestora. În cazul rețelelor de apă, unde am ales caracteristica presiunii ca mărime de intrare pentru algoritm și ținând cont de răspunsul în timp al rețelei am considerat ca fiind necesare următoarele operații:

- eliminarea frontului comun și extragerea diferenței dintre semnalul nominal și cel măsurat în rețea
- filtrarea semnalului obținut anterior

1.4. Nomenclatura mărimilor alese

Pentru a menține rigurozitatea și eleganța metodelor folosite este nevoie de o definire matematică pentru toate mărimile și metodele de filtrare folosite.

1.4.1. Presiunea în regim dinamic

Reprezintă o funcție de timp:

$$p_i : \mathbb{R} \longrightarrow \mathbb{R}^n, i \in V \tag{1.1}$$

unde n reprezintă numărul de noduri al rețelei, iar i reprezintă indexul nodului. Deoarece cazurile tratate în această lucrare reprezintă momente discrete de timp este important să definim presiunea măsurată în intervalele discrete în care este simulat procesul:

$$\mathbf{p}_i \in \mathbb{R}^{n \times p_{sim}} \tag{1.2}$$

unde p_{sim} reprezintă numărul de eșantioane pentru fiecare măsurătoare. Mergând mai departe în analiza simulării este de asemenea important să definim mărimea afectată de un defect în nodul j, de magnitudine m și măsurată în nodul i:

$$\mathbf{p}_i^{j,m} \in \mathbb{R}^{n \times p_{sim}} \tag{1.3}$$

Pentru cazul în care magnitudinea m ia valori nule, atunci vom considera notația mărimii nominale:

$$\mathbf{p}_i^{j,0} = \mathbf{p}_i^{nom}, \forall j \in V \tag{1.4}$$

Pentru valorile presiunii recoltate din rețea în nodul i despre care nu se cunoaște nici o informație, vom considera notația

$$\widehat{\mathbf{p}}_i \tag{1.5}$$

1.4.2. Presiunea în regim static

Considerând o plajă de momente de timp situate între indicii $rs_1 : rs_2$ unde se afla valorile de regim staționar ale procesului, putem defini o medie a regimului static în felul următor:

$$\overline{\mathbf{p}}_i^{j,m} = \frac{1}{rs_1 - rs_2 + 1} \sum_{k=rs_1}^{rs_2} \mathbf{p}_i^{j,m}[k] \tag{1.6}$$

În aceeași manieră definim și media presiunii nominale în regim static:

$$\overline{\mathbf{p}}_i^{j,0} = \overline{\mathbf{p}_i}^{nom}, \forall j \in V \tag{1.7}$$

Media presiunii măsurată în nodul i și despre care nu se cunosc informații în legătură cu valoarea și poziția defectului:

$$\overline{\widehat{\mathbf{p}}}_i = \frac{1}{rs_1 - rs_2 + 1} \sum_{k=rs_1}^{rs_2} \widehat{\mathbf{p}}_i[k] \tag{1.8}$$

1.4.3. Reziduuri

Așa cum a fost discutat în secțiunea 1.3, preprocesarea datelor are un rol important iar în cazul analizei și clasificării defectelor în rețelele cu apă, este nevoie să definim caracteristica prelucrată care va fi folosită mai apoi în procesul de izolare a defectelor. Reziduul absolut

reprezintă diferența dintre valoarea măsurată în rețea și valoarea nominală, aici putem discerne două cazuri: Reziduu temporal:

$$\mathbf{r}_i^{j,m} = \mathbf{p}_i^{j,m} - \mathbf{p}_i^{nom} \tag{1.9}$$

Reziduu atemporal, calculat ca diferența dintre cele două valori mediate pe intervalul staționar al caracteristicii:

$$r_i^{j,m} = \overline{\mathbf{p}}_i^{j,m} - \overline{\mathbf{p}}_i^{nom} \tag{1.10}$$

iar pentru valorile reziduului despre care nu se cunosc încă lucruri folosim notația din stilul anterior:

$$\widehat{r}_i = \overline{\widehat{\mathbf{p}}}_i - \overline{\mathbf{p}}_i^{nom} \tag{1.11}$$

Alte tipuri de reziduuri preprocesate sunt relative:

$$rrelativ_i^{j,m} = \frac{r_i^{j,m}}{\overline{\mathbf{p}}_i^{nom}} \tag{1.12}$$

Reziduurile normate:

$$rnorm_i^{j,m} = \frac{r_i^{j,m}}{\left\| r_{1:n}^{j,m} \right\|} \tag{1.13}$$

Reziduurile scalate:

$$rscal_i^{j,m} = \frac{r_i^{j,m} - \min r_{1:n}^{j,m}}{\max r_{1:n}^{j,m} - \min r_{1:n}^{j,m}} \tag{1.14}$$

Ca semnificație notațiile prezentate în 1.4 care conțin simbolul $\widehat{\ }$ fac referire la datele folosite pentru validarea modelului iar valorile unde se specifică nodul defectului și magnitudinea acestuia sunt considerate ca fiind date de antrenare și testare. Astfel în contextul definirii setului de dat pe care vom aplica algoritmii de calsificare va trebui să definim matricea:

$$\mathbf{R} < \mathbf{tip} >^{j,m} \in \mathbb{R}^{n_d \times n} \tag{1.15}$$

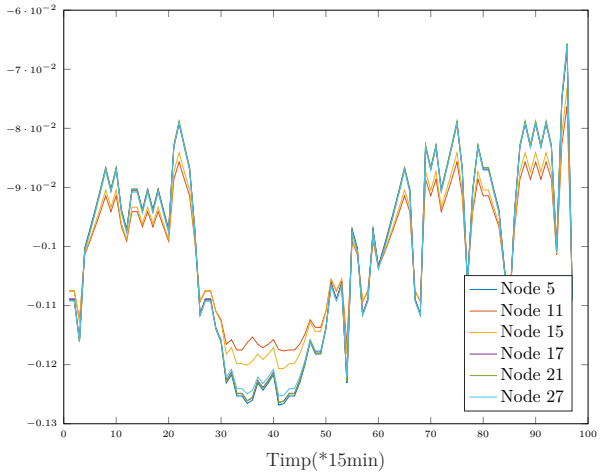
Unde croșetele din formulă reprezintă un înlocuitor pentru metoda de reziduu folosită iar n_d reprezintă numărul de defecte tratate în setul de date. De asemenea pentru fiecare linie a matricei (1.15) putem defini perechea

$$(\mathbf{R} < tip > (d, :), y_d) \tag{1.16}$$

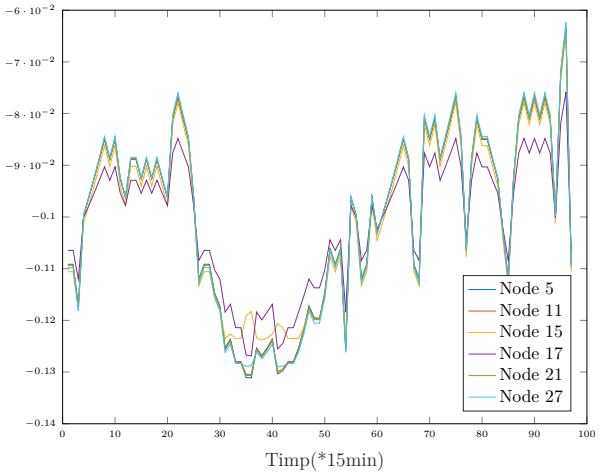
Unde $\mathbf{R} < tip > (d, :)$ reprezintă răspunsul rețelei prin reziduuri la defectul d. Iar y_d reprezintă eticheta pentru acest set de date, anume, nodul în care a avut loc defectul.

1.5. Calcul și prezentare reziduuri

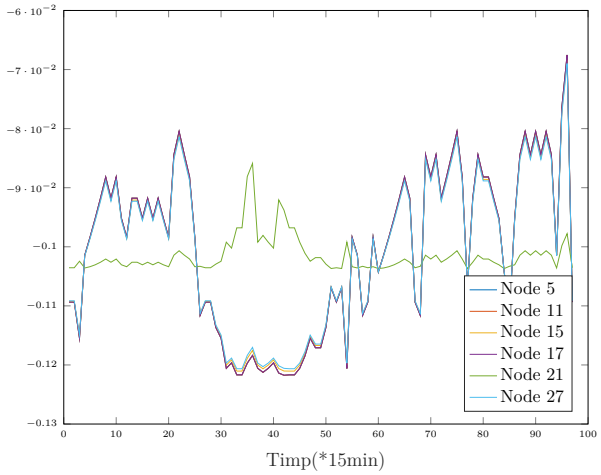
În continuare vom prezenta grafic reziduurile temporale normalizate care apar în rețea pentru diferite scenarii ale defectelor definite anterior. Astfel vom considera nodurile de măsurătoare ca o submulțime a lui $V' \subset V$ și $V = \{5, 11, 15, 17, 21, 27\}$, nodurile în care s-au simulat defecte sunt $V_d = \{11, 17, 27, 29\}$.



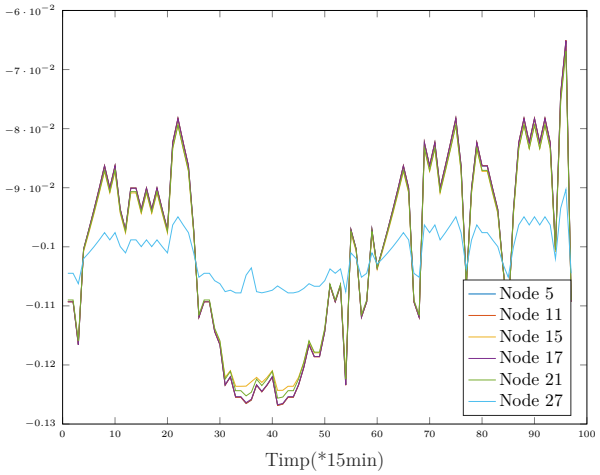
(a) Reziduuri pentru defect în nodul 11, magnitudine 29



(b) Reziduuri pentru defect în nodul 17, magnitudine 29



(c) Reziduuri pentru defect în nodul 21, magnitudine 29



(d) Reziduuri pentru defect în nodul 27, magnitudine 29

Figura 1.3.: Reziduuri rețea

Se poate observa că în figurile 1.3 reziduul cel mai pronunțat ca funcție de timp se găsește în nodul în care se simulează și defectul - lucru natural și de așteptat. O caracteristică importantă a acestei rețele de apă este faptul că există o dependență între diferitele răspunsuri în timp ale caracteristicii de presiune, fapt care ne permite să exploatăm redundanțele din rețea și să prezicem cu o acuratețe relativ ridicată defectele.

Este necesar acum să prezentăm profilurile reziduurilor atemporale,

Anexe

A. Fișiere sursă

```
# epanet toolkit
import json

import numpy as np
import pandas as pd
from epanettools.epanet2 import *
from epanettools.epanettools import *
from plotly import tools
from plotly.graph_objs import *
# plotting imports
from plotly.offline import plot

# extending the EPANetSimulation class to ease acces to
# simulation routines
class ENSim(EPANetSimulation):
    EN_INIT = 10

    def __init__(self, network_file, pdd=False):
        # careful when using pdd = true for residues simulations
        self.json_sim = {}
        super().__init__(network_file, pdd)

    def set_emitter(self, node_index, emitter_val):
        if self.network.nodes[node_index].node_type is EN_JUNCTION:
            ENSim._getncheck(self.ENsetnodevalue(node_index, EN_EMITTER, emitter_val
    ))

    def set_basedemand(self, node_index, demand_val):
        if self.network.nodes[node_index].node_type is EN_JUNCTION:
            ENSim._getncheck(self.ENsetnodevalue(node_index, EN_BASEDEMAND,
    demand_val))

    def set_emitters(self, emitter_info=None):

        if emitter_info is None:
            # if arg is none reset emitter values
            for node_index in self.network.nodes:
                self.set_emitter(node_index, 0)
        else:

            for node_index, emitter_val in emitter_info:
                self.set_emitter(node_index, emitter_val)

    def get_nodes_data(self, data_query, emitter=(1, 0)):

        no_nodes = ENSim._getncheck(self.ENgetcount(EN_NODECOUNT)) - ENSim.
    _getncheck(self.ENgetcount(EN_TANKCOUNT))
        t_step = 1
        node_values = {}

        for queries in data_query:
            node_values[queries] = [[] for _ in range(no_nodes)]

        node_values["EMITTER_NODE"] = emitter[0]
        node_values["EMITTER_VAL"] = emitter[1]

        # initialize network for hydraulic process
```

```
56          ENSim._getncheck(self.ENinitH(ENSim.EN_INIT))

58          while t_step > 0:

60              self.ENrunH()

62              for node_index in range(1, no_nodes + 1):
                    for query_type in data_query:
64                      ret_val = ENSim._getncheck(self.ENgetnodevalue(node_index, eval(
    query_type)))
                        node_values[query_type][node_index - 1].append(ret_val)
66
                t_step = ENSim._getncheck(self.ENnextH())
68
            for key in node_values:
70              node_values[key] = np.transpose(node_values[key]).tolist()

72          return node_values

74      def get_links_data(self, data_query, emitter=(1, 0)):

76          no_links = self.ENgetcount(EN_LINKCOUNT)[1]
            t_step = 1
78          link_values = {}

80          for queries in data_query:
                link_values[queries] = [[] for _ in range(no_links)]
82
            link_values["EMITTER_NODE"] = emitter[0]
84          link_values["EMITTER_VAL"] = emitter[1]

86          # initialize network for hydraulic process

88          ENSim._getncheck(self.ENinitH(ENSim.EN_INIT))

90          while t_step > 0:
                ENSim._getncheck(self.ENrunH())
92
                for link_index in range(1, no_links + 1):
94                  for query_type in data_query:
                        ret_val = ENSim._getncheck(self.ENgetlinkvalue(link_index, eval(
    query_type)))
96                      link_values[query_type][link_index - 1].append(ret_val)

98              t_step = ENSim._getncheck(self.ENnextH())

100         for key in link_values:
                link_values[key] = np.transpose(link_values[key]).tolist()
102
            return link_values
104
        def query_network(self, sim_dict):
106         """
            :param sim_dict: a dict containing info about the network
108         has the form
            {
110             simulation_name : "name",
                simulation_type: "H" or "Q"
112             emitter_values : [ (node_index, emitter_value) ]
                query : {
114                 nodes : [ "EN_PRESSURE"
                    ]
116                 links : [ "EN_VELOCITY"
                    ]
118             }
```

```
        }
120     :return: JSON with required data
        output_json format:
122     {
            SIM_NAME = simulation-name
124         NODE_VALUES = [
                {
126                 "EN_PRESSURE" : [ [values for each node]]
                    "EMITTER_VAL" :
128                 "EMITTER_NODE" :
                }
130         ]

132     }

134     """

136     # for the moment i'll treat only hydraulic simulations :)

138     # initialize network simulaton

140     ENSim._getncheck(self.ENopenH())

142     # initialize session
        ENSim._getncheck(self.ENinitH(ENSim.EN_INIT))
144
        node_query = False
146     link_query = False
        simulations = False
148
        # check json for querried data
150
        # node info:
152     try:
            if sim_dict["query"]["nodes"]:
154             node_query = True
        except KeyError:
156         node_query = False

158     # link info
        try:
160         if sim_dict["query"]["links"]:
                link_query = True
162     except KeyError:
            link_query = False
164
        # emitter info:
166     try:
            simulations = sim_dict["emitter_values"]
168     except KeyError:
            simulations = False
170
        if simulations:
172         node_values = []
            link_values = []
174
            # in order to plot residues we need to simulate the case where there is
    no leakage
176
            self.set_emitters()
178         if node_query:
                node_values.append(
180                 self.get_nodes_data(sim_dict["query"]["nodes"]))

182         if link_query:
                link_values.append(
```

```
184                     self.get_links_data(sim_dict["query"]["links"]))
186             for node_index, emitter_value in simulations:
                    print("Simulating emitter in node no {} with value {}".format(
    node_index, emitter_value))
188
                    self.set_emitter(node_index, emitter_value)
190
                    if node_query:
192                     node_values.append(
                            self.get_nodes_data(sim_dict["query"]["nodes"], emitter=(
    node_index, emitter_value)))
194
                    if link_query:
196                     link_values.append(
                            self.get_links_data(sim_dict["query"]["links"], emitter=(
    node_index, emitter_value)))
198
                    # reset emitter values everywhere in network
200                 self.set_emitters()
202         else:
204             if node_query:
                    node_values = [self.get_nodes_data(sim_dict["query"]["nodes"])]
206             else:
                    node_values = []
208
                if link_query:
210                 link_values = [self.get_links_data(sim_dict["query"]["links"])]
                else:
212                 link_values = []
214         ENSim._getncheck(self.ENcloseH())
            ENSim._getncheck(self.ENclose())
216
            self.__init__(self.OriginalInputFileName)
218         self.json_sim = {
                "SIM_NAME": sim_dict["simulation_name"],
220             "NODE_VALUES": node_values,
                "LINK_VALUES": link_values
222         }
            return self.json_sim
224
    def get_time_step(self, pattern_id=1):
226         """
            returns the time_step of the network in minutes
228         :param pattern_id:
            :return:
230         """
            return (24 * 60) / ENSim._getncheck(self.ENgetpatternlen(pattern_id))
232
    def plot(self, json_data, residues=False):
234         """
            utility function used to plot data from network simulations
236         WIP
            :param json_data:
238         :return:
            """
240         values = json_data["NODE_VALUES"]
            date_range = pd.date_range('1/1/2018', periods=97, freq='15min')
242
            if residues:
244             # consider the first value of the JSON as the reference
                ref = values[0]
246             for emitter in values:
```

```
                trace = []
248             data = np.transpose(emitter["EN_PRESSURE"]) - np.transpose(ref["
    EN_PRESSURE"])

250
                for node_index, vals in enumerate(data):
252                 trace.append(Scatter(
                        x=date_range,
254                     y=vals,
                        name="node{}".format(node_index+1)
256                 )
                    )
258             layout = dict(
                    title = "Ressidues with emitter in node {}, val = {}".format(
    emitter["EMITTER_NODE"], emitter["EMITTER_VAL"])
260             )

262             fig = dict(data=trace, layout=layout)
                plot(fig, filename= "Plot_node{}val{}".format(emitter["EMITTER_NODE"
    ], emitter["EMITTER_VAL"]))
264
        else:
266         for emitter in values:
                trace = []
268             data = np.transpose(emitter["EN_PRESSURE"])
                for node_index, vals in enumerate(data):
270                 trace.append(Scatter(
                        x=date_range,
272                     y=vals,
                        name="node{}".format(node_index+1)
274                 )
                    )
276             layout = dict(
                    title = "Pressure with emitter in node {}, val = {}".format(
    emitter["EMITTER_NODE"], emitter["EMITTER_VAL"])
278             )

280             fig = dict(data=trace, layout=layout)
                plot(fig, filename= "Plot_node{}val{}".format(emitter["EMITTER_NODE"
    ], emitter["EMITTER_VAL"]))
282
    def save_data(self, path=None):
284
        try:
286         with open(path, "wt") as file:
                file.write(json.dumps(self.json_sim))
288     except IOError:
            print("Could not write to file {}".format(path))
290
    @staticmethod
292 def write_json(output_json, path):

294     json_data = json.dumps(output_json)
        with open(path, "wt") as f:
296         f.write(json_data)

298 @staticmethod
    def _getncheck(ret_val):
300
        # check the return code
302     if isinstance(ret_val, list):
            if ret_val[0] == 0:
304             # everything OK
                return ret_val[1]
306         else:
                err_msg = ENgeterror(ret_val[0], 100)
```

```
                raise EpanetError(err_msg)
        else:
            if ret_val is not 0:
                err_msg = ENgeterror(ret_val, 100)
                raise EpanetError(err_msg)


def en_check(func):
    def func_wrapper(*args):
        ret_val = func(args)

        # check the return code
        if isinstance(ret_val, list):
            if ret_val[0] == 0:
                # everything OK
                return ret_val[1]
            else:
                err_msg = ENgeterror(ret_val[0], 100)
                raise EpanetError(err_msg)
        else:
            if ret_val is not 0:
                err_msg = ENgeterror(ret_val, 100)
                raise EpanetError(err_msg)


class EpanetError(Exception):

    def __init__(self, err_msg):
        super().__init__(err_msg)


def run_simulation(network, pdd, query_dict):

    es = EPANetSimulation(network, pdd)

    print("Running {}".format(query_dict["simulation_name"]))
    ret_vals = []

    print(query_dict)
    for emitter, emitter_val in query_dict["emitter_values"]:
        print("for node {} simulating emitter_Val {}".format(emitter, emitter_val))

        # modify current network and and save inp temp file

        es.ENsetnodevalue(emitter, EN_EMITTER, emitter_val)

        es.ENsaveinpfile("temp.inp")
        print(es.ENsetnodevalue(emitter, EN_EMITTER, 0))

        e2 = EPANetSimulation("temp.inp", pdd)
        e2.ENsetnodevalue(emitter,EN_EMITTER,emitter_val)

        e2.run()

        node_vals = {}
        link_vals = {}
        for node_query in query_dict["query"]["nodes"]:
            node_vals[node_query] = []

        for link_query in query_dict["query"]["links"]:
            link_vals[link_query] = []

        for node_query in query_dict["query"]["nodes"]:
            for node in e2.network.nodes:
                node_vals[node_query].append(e2.network.nodes[node].results[eval(
    node_query)])
```

```
374         for link_query in query_dict["query"]["links"]:
                for link in e2.network.links:
376                 link_vals[link_query].append(e2.network.links[link].results[eval(
    link_query)])


378
            ret_vals.append({
380             "EMITTER_VAL" : emitter_val,
                "EMITTER_NODE" : emitter,
382             "NODE_VALS" : np.transpose(node_vals).tolist(),
                "LINK_VALS" : np.transpose(link_vals).tolist()
384         })
        return ret_vals
386
#TODO metoda pentru modificarea demand-ului pe nod!!!
388 if __name__ == '__main__':
        es = ENSim("data/hanoi.inp", pdd=False)
390
        test_vals = [val for val in range(32) if val % 2 == 0]
392     train_vals = [val for val in range(32) if val % 2 == 1]


394
        intense_leak = [35, 40, 50, 60, 100]
396
        nodes = list(range(1, 32))
398
        emitter_test = [ (node, val) for node in nodes for val in test_vals]
400     emitter_train = [(node, val) for node in nodes for val in train_vals]
        emitter_intense_leak = [(node, val) for node in nodes for val in intense_leak]
402
        train_dataset = {
404         "simulation_name": "Hanoi train simulation",
            "simulation_type": "H",
406         "emitter_values": emitter_train,
            "query": {
408
                "nodes": ["EN_PRESSURE", "EN_DEMAND"],
410             "links": ["EN_VELOCITY"]
            }
412
        }
414
        test_dataset = {
416         "simulation_name": "Hanoi test simulation",
            "simulation_type": "H",
418         "emitter_values": emitter_test,
            "query": {
420
                "nodes": ["EN_PRESSURE", "EN_DEMAND"],
422             "links": ["EN_VELOCITY"]
            }
424
        }
426
        test2_dataset = {
428
            "simulation_name": "Hanoi intense leak simulation",
430         "simulation_type": "H",
            "emitter_values": emitter_intense_leak,
432         "query": {

434             "nodes": ["EN_PRESSURE", "EN_DEMAND"],
                "links": ["EN_VELOCITY"]
436         }
```

```
438     }

440     data_train = es.query_network(train_dataset)
        es.save_data("train_set.json")
442     data_test = es.query_network(test_dataset)
        es.save_data("test_set.json")
444     data_test2 = es.query_network(test2_dataset)
        es.save_data("test2_set.json")
```

Listing A.1: Wrapper EPANET – fișier complet

Bibliography

[1] Donald F Elger and John A Roberson. *Engineering fluid mechanics*. Wiley Hoboken (NJ), 2016.

[2] *Garbage in Garbage out.* 2016. URL: https://en.wikipedia.org/wiki/Garbage_in,_garbage_out.

[3] Richard M Karp. „On the computational complexity of combinatorial problems“. In: *Networks* 5.1 (1975), pp. 45–68.

[4] Rex Klopfenstein Jr. „Air velocity and flow measurement using a Pitot tube“. In: *ISA transactions* 37.4 (1998), pp. 257–263.

[5] SB Kotsiantis, D Kanellopoulos, and PE Pintelas. „Data preprocessing for supervised leaning“. In: *International Journal of Computer Science* 1.2 (2006), pp. 111–117.

[6] Assela Pathirana. *EPANET calling API for python.* 2016. URL: https://github.com/asselapathirana/epanettools.

[7] Lewis A Rossman et al. „EPANET 2: users manual“. In: (2000).

[8] Gerard Sanz Estapé. „Demand modeling for water networks calibration and leak localization“. In: (2016).