Comparative Study of AMPL, Pyomo and JuMP Optimization Modeling Languages on a Flood Control Problem Example

Andrzej Karbowski, Krzysztof Wyskiel
Warsaw University of Technology, Faculty of Electronics and Information Technology, Nowowiejska 15/19, 00-665 Warszawa

Abstract: The purpose of this work is a comparative study of three languages (environments) of optimization modeling: AMPL, Pyomo and JuMP. The comparison will be based on three implementations of an optimal discrete-time flood control problem formulated as a nonlinear programming problem. The codes for individual models and differences between them will be presented and discussed. Various aspects will be taken into account, e.g. simplicity and intuitiveness of implementation.

Keywords: optimization, modeling languages, programming, flood control problem, nonlinear programming, optimal control

1. Introduction

The aim of this work is a comparative study of three popular optimization modeling languages: AMPL, Pyomo and JuMP and their capabilities to solve a flood control problem. In our previous paper [2] the structures used, the available elements and the methods of their construction have already been described in detail. Even though the models for flood control problem are longer than those for the shortest path problem in the graph [2], there is not much new to it. Most of a model in every language is actually a large number of parameters, constraints and relationships between them, that occur in the mathematical expressions. Therefore, in the descriptions of individual implementations, first of all, new things will be discussed, concerning, i.a., loading data from multiple files into one model.

2. Flood wave control

Optimization of releases from retention reservoirs during floods is quite an important issue for water management in many countries. The nonlinear model described in this chapter was developed in [3–5, 1]. The considered river system is presented in Fig. 1.

It is a system with one retention reservoir, two sections of the main river and a side inflow. Three characteristic points

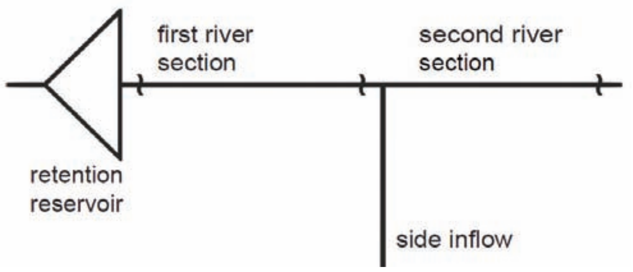


Fig. 1. A single reservoir river system
Rys. 1. System rzeczny z jednym zbiornikiem

are marked with wavelets: the spot just below the reservoir, the end of the first section and the end of the second section. In these places a flood wave culmination will be calculated. Our goal will be the minimization of flood damages approximated by the sum of the wave culminations Q_i^{cul} in these points. A control is calculated for the time horizon of T stages. The data given at the input regarding the tank inflow, side inflow, distributed side inflow on the first and the second section (about 10% of the main inflows), are denoted successively as: d, d_b, q^1 and q^2. An additional condition is that at the end of the simulation the retention reservoir should be completely filled. To solve the problem we will apply the first approach mentioned in [6], where system equations are discretized a'priori with respect to time. The mathematical model is as follows [5]:

$$\min_{u,w,Q} \sum_{j=1}^{3} p_j \cdot Q_j^{cul} \tag{1}$$

$$Q_1^{cul} = \max_{t=0,\dots,T-1} u(t) \tag{2}$$

$$Q_j^{cul} = \max_{t=0,\dots,T-1} Q_{wy}^{j-1}(t), \quad j = 2,3 \tag{3}$$

subject to:

Autor korespondujący:
Andrzej Karbowski, A.Karbowski@ia.pw.edu.pl

Artykuł recenzowany
nadesłany 16.06.2021 r., przyjęty do druku 20.09.2021 r.

$$w(t+1) = w(t) + \tau \cdot (d(t) - u(t)), \quad t = 0, \ldots, T-1 \tag{4}$$

$$w(0) = w_0 \tag{5}$$

$$w(T) = w_{\max} \tag{6}$$

$$w_{\min} \le w(t) \le w_{\max}, \; t = 0, \ldots, T-1 \tag{7}$$

$$u_{\min} \le u(t) \le u_{\max}(w(t)) = a + b \cdot w(t), \; t = 0, \ldots, T-1 \tag{8}$$

$$Q^1_{we_1}(t) = u(t), \quad t = 0, \ldots, T-1 \tag{9}$$

$$Q^j_{wy_i}(t+1) - Q^j_{wy_i}(t) = \tau \frac{1}{n_j k_j} Q^j_{wy_i}(t)^{1-n_j} \cdot \left[Q^j_{we_i}(t) - Q^j_{wy_i}(t) \right]$$
$$j = 1,2;\; i = 1,\ldots,E;\; t = 0,\ldots,T-1 \tag{10}$$

$$Q^j_{wy_i}(0) = Q^j_0 \tag{11}$$

$$Q^j_{we_i}(t) = Q^j_{wy_{i-1}}(t), \quad i = 2,\ldots,E;\; t = 0,\ldots,T-1 \tag{12}$$

$$Q^1_{wy}(t) = Q^1_{wy_{10}}(t) + q^1(t), \quad t = 0,\ldots,T-1 \tag{13}$$

$$Q^2_{we}(t) = Q^1_{wy}(t) + d_b(t), \quad t = 0,\ldots,T-1 \tag{14}$$

$$Q^2_{wy}(t) = Q^2_{wy_{10}}(t) + q^2(t), \quad t = 0,\ldots,T-1 \tag{15}$$

In the above model a and b are coefficients describing the characteristics of the reservoir. The p vector contains weighting coefficients for peaks. A release (trajectory) from the reservoir is denoted as u, a retention of the reservoir as w, and flows as Q. The maximum value of the release is directly related to the construction of the retention reservoir, so this value depends on the level of its current storage $u_{\max}(w(t))$.

An important element of the above model is the flood wave transformation function in the j-th section of the river (10). This function has been derived from Saint-Venant PDE, discretized in space into E pieces for every section and the first-order Runge-Kutta method with the integration step τ [5, 3]. Inputs and outputs for the j-th section of the river are denoted as $Q^j_{we_1}$ and Q^j_{wy}, respectively. The τ coefficient is given in seconds, and its value depends on the frequency with which we will integrate (i.e. in how many minutes next values of the reservoir storage and water levels in the river will be calculated). Variables with the argument 0 mean the initial values.

2.1. AMPL

The implementation of this problem in AMPL is presented in Listing 1.

At the beginning, we declare the standard and indexed parameters necessary for the model. For some of them, however, we also assign appropriate values. For `tau`, this is the value of dividing 3600 (the full hour in seconds) by the `iperh` parameter, which, as we can see, has not yet been assigned a value. However, we do not have to worry about this, because the appropriate value will be assigned after reading the data from a file and running the `solve` command. The situation is similar for the parameters `T` and `hour`. Additionally, we calculate the values for this parameter using the `ceil` function, which rounds the result of the given expression up to a whole integer. For this purpose, instead of specifying only the index range $1..T$, as in the case of declaring other visible parameters, we provide the full index expression with an iterator that we use to generate the next values. Successively, we declare a few more necessary parameters, including those related to tributaries. Loading data for them will be discussed at the end of this section.

When declaring variables, we can give them a lower or upper bound or both. In the last case, we can write both constraints in the same line after comma (it does not matter whether we

```
param N;          # control horizon in hours
param E;
param k{1..2};
param n{1..2};
param p{1..3};
param a; param b;
param iperh;      # division of an hour
param tau := 3600/iperh;
param T:= N*iperh;
param hour{t in 1..T} := ceil(t*tau/3600);
param wmin;  param wmax;  param w_0;
param Q1_0;
param umin;
param d{1..N};
param db{1..N};
param q1{1..N};
param q2{1..N};

var w{1..T} >= wmin, <= wmax;
var u{1..T} >= umin;
var umax{t in 1..T} = a + b*w[t]*1e-6;
var Qwe{1..2,1..E,1..T} >= 1e-6;
var Qwy{1..2,1..E,1..T} >= 1e-6;
var Qcul{i in 1..3} >= 1e-6;

minimize Ju:
    sum{j in 1..3} p[j]*Qcul[j];

subject to Qcul1{t in 1..T}:    Qcul[1] >= u[t];
subject to Qcul2{t in 1..T}:    Qcul[2] >= Qwy[1,E,t] + q1[hour[t]];
subject to Qcul3{t in 1..T}:    Qcul[3] >= Qwy[2,E,t] + q2[hour[t]];

subject to flow_between_elements{j in 1..2, i in 2..E, t in 1..T}:
    Qwe[j,i,t] = Qwy[j,i-1,t];
subject to Qwe1{t in 1..T}:
    Qwe[1,1,t] = u[t];
subject to Qwe2{t in 1..T}:
    Qwe[2,1,t] = Qwy[1,E,t] + q1[hour[t]] + db[hour[t]];

subject to w_initial:   w[1] = w_0;
subject to w_final:     w[T] = wmax;
subject to res_fill{t in 1..T-1}:
                        w[t+1] = w[t] + tau*(d[hour[t]] - u[t]);
subject to u_during_flood{t in 1..T}:  u[t] <= umax[t];
subject to Qwy1_2_section_initial{j in 1..2, i in 1..E}:    Qwy[j,i,1] = Q1_0;
subject to flow_transformation_rk1{j in 1..2, i in 1..E, t in 1..T-1}:
    Qwy[j,i,t+1]-Qwy[j,i,t] =
    tau*(1/(n[j]*k[j]))*Qwy[j,i,t]^(1-n[j])
    *(Qwe[j,i,t]-Qwy[j,i,t]);
```

Listing 1. Flood control problem implementation in AMPL
Listing 1. Implementacja zadania sterowania falą powodziową w AMPLu

give first the lower or the upper one), as we can see in the case of the `w` variable. We also declare the `umax` variable (the maximum value of the release from the reservoir at a given moment in time) and use the equality operator to give an expression describing the characteristics of the reservoir to which this variable will be equivalent (this is a so-called defined variable in AMPL, which is not a decision variable itself, but a kind of a macro definition). Thanks to this, we can use this variable instead of the entire expression in the constraint on the maximum release from the reservoir (in both cases the model passed to a solver will be identical and it will return the same result after the same number of iterations). In this model, such a variable is used only in one constraint, but in models where it is used many times, it would definitely affect the readability of the code. Finally, we declare the remaining indexed variables `Qwe`, `Qwy` and `Qcul`.

There are no new things in the definitions of the objective function and constraints as compared to the models discussed in [2]. We create them in a standard way, using the structures described in [2], in accordance with the mathematical model and the description of changes made at the beginning of this article.

The only thing left is to load the data and run the `solve` command. Data delivery to the model will consist of two parts: the first is to load most of the parameters from the .dat file using the `data 'name.dat'`; command and the second, to load the data for the `d`, `db`, `q1`, and `q2`. The data for each of these four parameters is in a separate text file. Successive values are separated by spaces, ten values per line (for the reader's convenience). Successive values are written from left to right and then down. To read this data, we use the `read` function:

```
read {it in 1..N} d[it] < d_param.txt;
read {it in 1..N} q1[it] < q1_param.txt;
read {it in 1..N} q2[it] < q2_param.txt;
read {it in 1..N} db[it] < db_param.txt;
```

Listing 2. The function read in AMPL
Listing 2. Funkcja read w języku AMPL

The syntax is simple: we give the keyword, then the index expression, the variable into which we load (along with the index), operator < and the name of the file being read (or the full path, which we must then put in quotation marks). We put these calls under the `data 'name.dat'`; command. The number of spaces in the file is irrelevant, and the values are read exactly in the order previously described. Interestingly, putting the calls to the `read` function in the .dat file worked fine. This is in a sense a "trick", as there is no description of such a method in the AMPL book. Nevertheless, it allows you to put things related to loading data in one place, and we only load a single .dat file to solve the model.

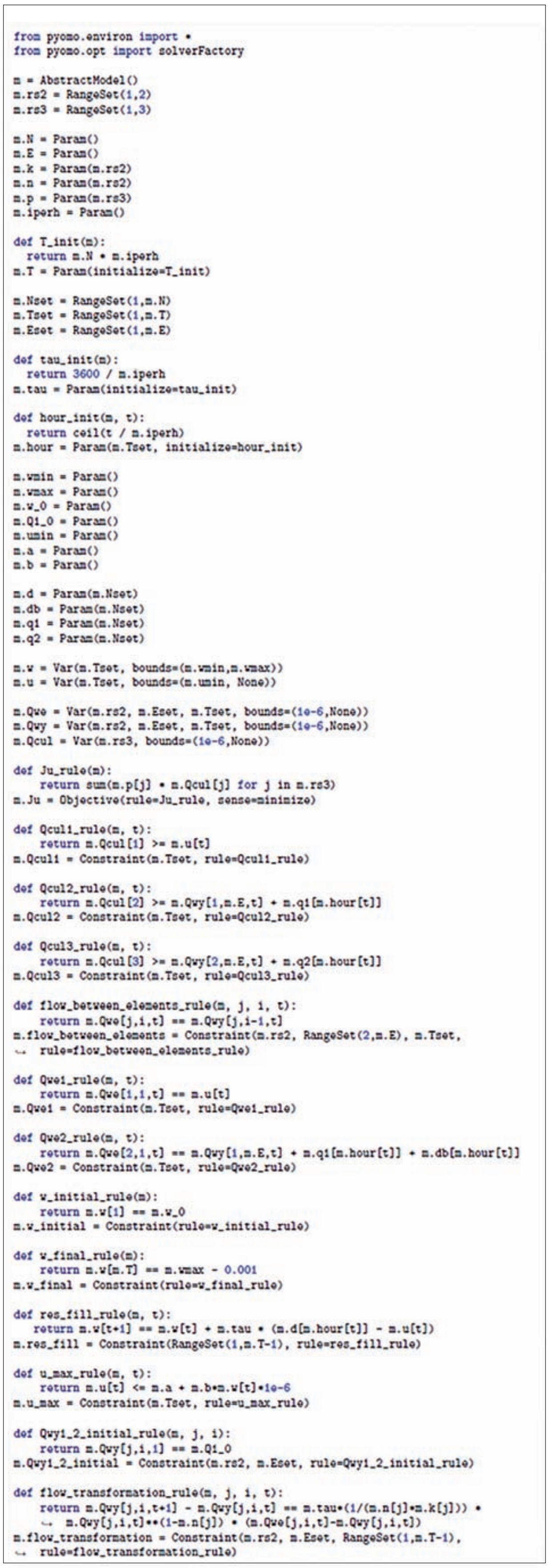
2.2. Pyomo

The implementation of this problem in Pyomo is shown in Listing 3.

Traditionally, at the beginning we import the necessary Pyomo libraries. As in the case of the graph model from paper [2], this model will be abstract (`AbstractModel()` method), because we do not need specific data when creating it. Next, we add to the model a few auxiliary files, that will be used for indexing (remembering, that some files are created only after declaring the parameters that are used to create these sets), and the necessary parameters. For some of them we need to set values returned by expressions depending on other parameters, that is for `m.tau`, `m.T` and `m.hour`. We do this by providing the `Param()` method with the `initialize` argument, which takes the name of the method generating the appropriate expression. We can also give the expression itself (in parentheses), which will shorten a bit the code. However, we can only do this for simple parameters (without indices). Due to the fact that most of the constraints and parameters are indexed, for some consistency, instead of the direct expression a method is used here. The parameters will be built and set to the appropriate values when instantiated using the `m.create_instance()` method. For indexed `hour` we simply add the set in `Param()` and include the variable for the index number in the `hour_init` method. The values themselves are generated using the Python-built `ceil` method, which rounds the result of the expression in parentheses up to an integer number. We also declare the missing parameters, including those concerning inflows, for which the data loading method will be discussed at the end of the section.

When declaring variables, we can limit them by passing the `bounds` argument in the `Var()` method, where in parentheses we give the lower and upper bounds (in this order) separated by a comma. For example, if we want to limit a variable only from the bottom, we pass the keyword `None` after comma. Contrary to AMPL, there is no analogous structure for `umax` (maximum value of release from the reservoir at a given moment in time), so the expression describing the characteristics of the reservoir will be used directly in the limitation of the maximum release for a given reservoir capacity (constraint `mu.u_max`). Finally, we add the remaining `Qwe`, `Qwy` and `Qcul` indexed variables to the model.

The objective function and constraints contain the constructs already discussed in [2] and are created in a similar way. However, there was a problem with the `m.w_final` constraint, and more specifically with the `m.w_final_rule` method (full filling at the end). The `m.w` variable has an upper bound defined with `bounds` as `m.wmax`, which is also a condition for `m.w_final`. As a result, the solver was not able to solve the passed model then (it looked as if it was "stuck" in a certain area of local minima, from which it could not "jump"). Fortunately, the solution to the problem turned out to be quick and easy. As we can see in the code, it was enough to subtract from `m.wmax` a relatively small number 0.001. Thanks to this, the solver was able to find and return the correct result. However, giving a too small number did not work. For example, for 10^{-6}, the solver needed about 10 times more iterations and the resulting solution left a lot to be desired (huge amplitudes of dumps in consecutive moments of time). Instead of subtracting here 0.001, we can also add this value to the upper limit for the reservoir capacity (in the `bounds` argument when declaring the `m.w` variable). However, the solver then needs a bit more iterations to find the result, and it is slightly worse.



```
from pyomo.environ import *
from pyomo.opt import solverFactory

m = AbstractModel()
m.rs2 = RangeSet(1,2)
m.rs3 = RangeSet(1,3)

m.N = Param()
m.E = Param()
m.k = Param(m.rs2)
m.n = Param(m.rs2)
m.p = Param(m.rs3)
m.iperh = Param()

def T_init(m):
  return m.N * m.iperh
m.T = Param(initialize=T_init)

m.Nset = RangeSet(1,m.N)
m.Tset = RangeSet(1,m.T)
m.Eset = RangeSet(1,m.E)

def tau_init(m):
  return 3600 / m.iperh
m.tau = Param(initialize=tau_init)

def hour_init(m, t):
  return ceil(t / m.iperh)
m.hour = Param(m.Tset, initialize=hour_init)

m.wmin = Param()
m.wmax = Param()
m.w_0 = Param()
m.Q1_0 = Param()
m.umin = Param()
m.a = Param()
m.b = Param()

m.d = Param(m.Nset)
m.db = Param(m.Nset)
m.q1 = Param(m.Nset)
m.q2 = Param(m.Nset)

m.w = Var(m.Tset, bounds=(m.wmin,m.wmax))
m.u = Var(m.Tset, bounds=(m.umin, None))

m.Qwe = Var(m.rs2, m.Eset, m.Tset, bounds=(1e-6,None))
m.Qwy = Var(m.rs2, m.Eset, m.Tset, bounds=(1e-6,None))
m.Qcul = Var(m.rs3, bounds=(1e-6,None))

def Ju_rule(m):
    return sum(m.p[j] * m.Qcul[j] for j in m.rs3)
m.Ju = Objective(rule=Ju_rule, sense=minimize)

def Qcul1_rule(m, t):
    return m.Qcul[1] >= m.u[t]
m.Qcul1 = Constraint(m.Tset, rule=Qcul1_rule)

def Qcul2_rule(m, t):
    return m.Qcul[2] >= m.Qwy[1,m.E,t] + m.q1[m.hour[t]]
m.Qcul2 = Constraint(m.Tset, rule=Qcul2_rule)

def Qcul3_rule(m, t):
    return m.Qcul[3] >= m.Qwy[2,m.E,t] + m.q2[m.hour[t]]
m.Qcul3 = Constraint(m.Tset, rule=Qcul3_rule)

def flow_between_elements_rule(m, j, i, t):
    return m.Qwe[j,i,t] == m.Qwy[j,i-1,t]
m.flow_between_elements = Constraint(m.rs2, RangeSet(2,m.E), m.Tset,
↪  rule=flow_between_elements_rule)

def Qwe1_rule(m, t):
    return m.Qwe[1,1,t] == m.u[t]
m.Qwe1 = Constraint(m.Tset, rule=Qwe1_rule)

def Qwe2_rule(m, t):
    return m.Qwe[2,1,t] == m.Qwy[1,m.E,t] + m.q1[m.hour[t]] + m.db[m.hour[t]]
m.Qwe2 = Constraint(m.Tset, rule=Qwe2_rule)

def w_initial_rule(m):
    return m.w[1] == m.w_0
m.w_initial = Constraint(rule=w_initial_rule)

def w_final_rule(m):
    return m.w[m.T] == m.wmax - 0.001
m.w_final = Constraint(rule=w_final_rule)

def res_fill_rule(m, t):
   return m.w[t+1] == m.w[t] + m.tau * (m.d[m.hour[t]] - m.u[t])
m.res_fill = Constraint(RangeSet(1,m.T-1), rule=res_fill_rule)

def u_max_rule(m, t):
    return m.u[t] <= m.a + m.b*m.w[t]*1e-6
m.u_max = Constraint(m.Tset, rule=u_max_rule)

def Qwy1_2_initial_rule(m, j, i):
    return m.Qwy[j,i,1] == m.Q1_0
m.Qwy1_2_initial = Constraint(m.rs2, m.Eset, rule=Qwy1_2_initial_rule)

def flow_transformation_rule(m, j, i, t):
    return m.Qwy[j,i,t+1] - m.Qwy[j,i,t] == m.tau*(1/(m.n[j]*m.k[j])) *
    ↪  m.Qwy[j,i,t]**(1-m.n[j]) * (m.Qwe[j,i,t]-m.Qwy[j,i,t])
m.flow_transformation = Constraint(m.rs2, m.Eset, RangeSet(1,m.T-1),
↪  rule=flow_transformation_rule)
```

Listing 3. Flood control problem implementation in Pyomo
Listing 3. Implementacja zadania sterowania falą powodziową w Pyomo

Finally, you have to load the data, create an instance and solve the model. For reading data from multiple files, Pyomo offers the `DataPortal()` object used in the following way:

```
data = DataPortal()
data.load(filename="flood-data.dat", model=m)
data.load(filename="d_param.tab", param=m.d)
data.load(filename="db_param.tab", param=m.db)
data.load(filename="q1_param.tab", param=m.q1)
data.load(filename="q2_param.tab", param=m.q2)
instance = m.create_instance(data)
```

Listing 4. DataPortal() object in Pyomo
Listing 4. Obiekt DataPortal() w Pyomo

After creating the mentioned object, we call the `load()` method on it as many times as many files we have to load. In the `filename` argument we provide the name of the loaded file. If it is a .dat file (the construction for the reminder is the same as for the AMPL counterpart), then additionally in the `model` argument we provide (as you can guess) the object of our model. In the case of files with data on inflows to the reservoir, etc., we pass the `param` argument instead, in which we pass a specific parameter from the model. These files have a .tab extension and have their own syntax. In this case, the first column contains the indices 1 to N, while the second column contains specific values. Additionally, in the first line there are one-word names of these columns (which are only informative for the person checking the content of the file). Finally, the entire `data` object is passed to the `create_instance()` method, and the created instance is passed to the `solve` method of the solver object.

2.3. JuMP

The implementation of this problem in JuMP is shown in Listing 5.

```
1   using JuMP, KNITRO
2
3   p_vals = Dict()
4   for i in map(it -> split(it, ":"), readlines("flood-params.txt"))
5      p_vals[strip(i[1])] = parse.(split(strip(i[2]), " "))
6   end
7
8   N = p_vals["N"][1]
9   E = p_vals["E"][1]
10  k = p_vals["k"]
11  n = p_vals["n"]
12  p = p_vals["p"]
13
14  a = p_vals["a"][1]
15  b = p_vals["b"][2]
16
17  iperh = p_vals["iperh"][1]
18
19  tau = 3600 / iperh
20  T = N * iperh
21  hour = [ceil(Int32, t/iperh) for t in 1:T]
22
23  wmin = p_vals["wmin"][1]
24  wmax = p_vals["wmax"][1]
25  w_0 = p_vals["w_0"][1]
26
27  umin = p_vals["umin"][1]
28  Q1_0 = p_vals["Q1_0"][1]
29
30  d = vec(readdlm("d_param.txt", ' ', Int64)')
31  db = vec(readdlm("db_param.txt", ' ', Int64)')
32  q1 = vec(readdlm("q1_param.txt", ' ', Int64)')
33  q2 = vec(readdlm("q2_param.txt", ' ', Int64)')
34
35  m = Model(solver = KnitroSolver())
36
37  @variable(m, wmin <= w[1:T] <= wmax)
38  @variable(m, u[1:T] >= umin)
39
40  @variable(m, Qwe[1:2,1:E,1:T] >= 1e-6)
41  @variable(m, Qwy[1:2,1:E,1:T] >= 1e-6)
42  @variable(m, Qcul[1:3] >= 1e-6)
43
44  @objective(m, Min, sum(p[j] * Qcul[j] for j in 1:3))
45  @constraint(m, Qcul1_t[t=1:T], Qcul[1] >= u[t])
46  @constraint(m, Qcul2_t[t=1:T], Qcul[2] >= Qwy[1,E,t] + q1[hour[t]])
47  @constraint(m, Qcul3_t[t=1:T], Qcul[3] >= Qwy[2,E,t] + q2[hour[t]])
48
49  @constraint(m, flow_between_elements[j=1:2,i=2:E,t=1:T], Qwe[j,i,t] ==
    ↪  Qwy[j,i-1,t])
50  @constraint(m, Qwe1[t=1:T], Qwe[1,1,t] == u[t])
51  @constraint(m, Qwe2[t=1:T], Qwe[2,1,t] == Qwy[1,E,t] + q1[hour[t]] +
    ↪  db[hour[t]])
52
53  @constraint(m, w[1] == w_0)
54  @constraint(m, w[T] == wmax - 0.001)
55  @constraint(m, res_fill[t=1:T-1], w[t+1] == w[t] + tau * (d[hour[t]] - u[t]))
56
57  @constraint(m, u_t_max[t=1:T], 1200 + 15*w[t]*1e-6 >= u[t])
58  @constraint(m, sec_init[j=1:2,i=1:E], Qwy[j,i,1] == Q1_0)
59  @NLconstraint(m, flow_transformation[j=1:2,i=1:E,t=1:T-1], Qwy[j,i,t+1] -
    ↪  Qwy[j,i,t] == tau*(1/(n[j]*k[j]))
    ↪  *Qwy[j,i,t]^(1-n[j])*(Qwe[j,i,t]-Qwy[j,i,t]))
```

Listing 5. Flood control problem implementation in JuMP
Listing 5. Implementacja zadania sterowania falą powodziową w JuMP

We start with importing the appropriate packages. Due to the fact, that in JuMP there are no typical constructs for parameters and sets, which are also used in indexing, we have to load the data for the model at the very beginning (similarly to the graph problem model in [2]). It was also necessary to come up with a clear syntax for the input file with parameters data, which, when loaded into the program, will be easy to parse and prepare. Ultimately, the data in this file looks like this:

```
N:     300
k:     13700   5000
n:     0.756   0.915
p:     0.0146  0.0119  0.1149
...
```

Listing 6. A fragment of the input file with wave data
Listing 6. Fragment pliku wejściowego z danymi fali

In each line, we start with the name of the parameter, followed by a colon, and then the value or values separated by spaces. This gives us a readable input file and is easy to parse. Thanks to the extensive constructs and methods in native Julia, this can be done in a short way as shown in the model code above (lines 3-6). We create the dictionary variable `p_vals` (using `Dict()`). We then provide an expression for the loop. The `map()` method maps the elements from the array given in the second argument using the expression given in the first. Let's start with the second argument passed to this method. The `readlines()` function reads lines from a given text file and creates an array of strings from them. In the first `map()` argument we pass an anonymous function of the form:

```
(parameter_name1, parameter_name2, ...)
            → (function body)
```

In our case, there is only one parameter, so we can omit the parentheses. The name of the parameter is irrelevant, it is essential that we use it in the function body. Here, the parameter is a text line that is separated with the split() method in place of the colon (the colon itself is lost). The result is a pair of (`name, string with values`). After the mapping is done, we have a list of pairs that serve as the iterator for the `for` loop. Inside, we create the appropriate elements in the `p_vals` variable. On the left side of the assignment, in square brackets, we give the key by reading it from the first position of the `i` iterator, which we additionally place in the `strip()` method, removing spaces from both ends of the string passed to it. On the right side of the assignment, starting from the very center, we read the second element (a text string with numerical values) from the iterator. We remove white spaces from both ends, and then use `split` to separate the values from each other, getting an array. Finally, we use the parse.`()` method to cast a string value to a numeric one (which is detected automatically because the resulting strings contain only numbers). Note here the characteristic dot after the `parse` method name, before the brackets. It means the execution of the method in a vector way, i.e., it will be called for each element from the array that we passed as an argument (the created array with numbers). It is a very compact solution, owing to which we do not need to use an additional loop. Finally, the `p_vals` variable contains keys that are variable names and values that are lists of numbers.

Then we create the parameters by reading the appropriate values from our dictionary through a key. If it is a simple, non-array parameter, then we additionally extract it from a one-element list, adding "1" in square brackets (i.e., simply the first index of the array). We calculate the `tau` and `T` parameters using standard mathematical operations. For `hour` we generate the values in the array using the `ceil` method and the expression `for`. The `ceil` function itself, that rounds the result up to the whole number, in the first argument takes the type (we give `Int32` so that the number is an integer; by default it returns a floating point with a zero fraction), and in the second a mathematical expression or a fractional number.

The parameters for `d`, `db`, `q1` and `q2` tributaries can be read from separate text files using the `readdlm()` method (similarly to the graph problem [2]). Successive values are separated by spaces, ten values per line (for the reader's convenience). They are written from left to right and then down. We must remember that after reading the data will be in the matrix. We can "flatten" it down to one dimension using the `vec()` method, which reads the matrix column from top to bottom. This means, that we need to transpose the matrices. In the above code, behind the parenthesis ending the `readdlm()` method, we can see a characteristic apostrophe, by which we are just transposing the matrix.

We create a model object as standard, taking into account the solver passed in the constructor argument of the `Model()` method. As in the previous models, we declare the variables and constraints according to the previously described structures. As with Pyomo, there was a problem with the final reservoir filling constraint `w[T] == wmax`. It turned out that the solution to the problem is identical, when we just subtract (as we can see in the code) a relatively small number 0.001. Likewise, we can instead add this value in the upper bound on the reservoir capacity, with the same consequences as in Pyomo. It is quite an interesting situation that the same behavior occurred in both these languages, but it did not occur in AMPL.

Another limitation that draws attention is the wave transformation on river sections (`flow_transformation`). JuMP has a `@NLconstraint` method for nonlinear constraints. Using it is just the same as using normal constraints, the only point is that it is necessary because JuMP does not allow us to write a nonlinear constraint using the `@constraint()` method. This in some way increases the readability of the model, because we can immediately see what nonlinear constraints are, without analyzing them. There is also a similar method for the objective function (`NLobjective()`), but in this model the function is linear.

Finally, all that's left to do is to call `solve(m)` and to read the results.

3. Tests

We solved a flood wave control problem using data concerning Dunajec river with Rożnów reservoir and Biała Tarnowska side inflow. The following values of parameters were used:

`N` = 300 h (time horizon),

`iperh` = 4 (#steps in 1h),

`k` = [13700, 5000],	`n` = [0.756, 0.915],
`p` = [0.0146, 0.0119, 0.1149],	
`a` = 1200,	`b` = 15,
`umin` = 40 m^3	`w_0` = 84.9 mln m^3,
`wmin` = 40.3 mln m^3,	`wmax` = 171.2 mln m^3,
`Q1_0` = 100 m^3,	`E` = 10.

The values of inflows $d(.)$, $d_b(.)$, $q^1(.)$, $q^2(.)$ are given in tables in [5]. The resulting number of decision variables was 50001. In all implementations we used Knitro solver. In every case after 918 iterations and about 400 s on a PC with Intel Core i7-2600 3.40 GHz processor we got the same results. They are presented in Figs. 2, 3. The obtained curves are very similar to those presented in [5], despite the coding effort was much smaller, because of using high level modeling languages and a given, efficient commercial nonlinear solver.

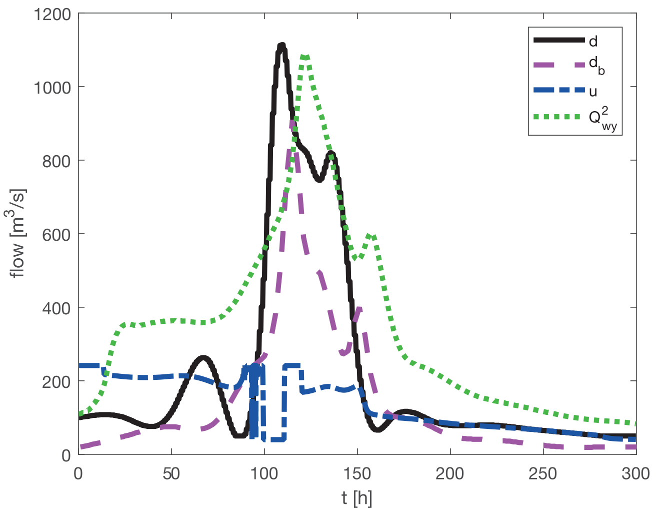


Fig. 2. Inflows, the optimal release from the reservoir and the outflow from the system
Rys. 2. Dopływy, optymalny zrzut oraz przepływ na wyjściu systemu

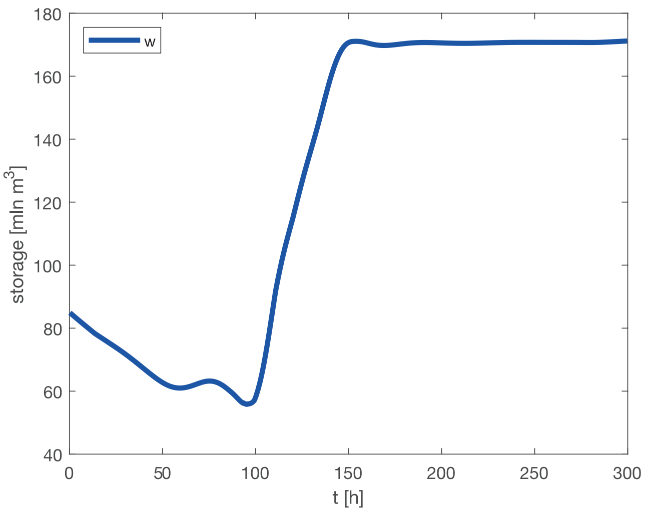


Fig. 3. Reservoir storage trajectory for the optimal release
Rys. 3. Trajektoria napełnienia zbiornika dla zrzutu optymalnego

4. Conclusions

The AMPL, Pyomo and JuMP environments presented in this paper proved to be effective tools for discrete-time optimal control problems modeling. AMPL, as a language dedicated to optimization modeling, allowed for the easiest and most convenient, resembling mathematical formulation, without programming overheads, implementation of models. It also provided the shortest code of all the environments discussed here. In terms of the number of characters and lines we need to write, undoubtedly the implementation in Pyomo is the worst. JuMP is between the two. Despite the lack of constructs for parameters and sets in JuMP and Pyomo one can successfully use the standard functionalities offered by Julia and Python languages. The creation of mathematical expressions was quite comfortable and intuitive in these environments. JuMP offers built-in operations on vectors and arrays, which are missing in AMPL. Moreover, both Pyomo and JuMP offer all constructs of their native languages what is most important, when optimization is only a part of a bigger application.

References

1. Karbowski A., *FC-ROS – decision support system for reservoir operators during flood*, "Environmental Software", Vol. 6, No. 1, 1991, 11–15, DOI: 10.1016/0266-9838(91)90012-F.
2. Karbowski A., Wyskiel K., *Comparative study of AMPL, Pyomo and JuMP optimization modeling languages on a network linear programming problem example.* „Pomiary Automatyka Robotyka", R. 25, Nr 3, 2021, 23-30, DOI: 10.14313/PAR_241/23.
3. Niewiadomska-Szynkiewicz E., Malinowski K., Karbowski A., *Predictive methods for real-time control of flood operation of a multireservoir system: Methodology and comparative study.* "Water Resources Research", Vol. 32, No. 9, 1996, 2885–2895, DOI: 10.1029/96WR01443.
4. Pytlak R., Malinowski K., *Optimal scheduling of reservoir releases during flood: Deterministic optimization problem, Part 1, Procedure.* "Journal of Optimization Theory and Applications", Vol. 61, No. 3, 1989, 409–432, DOI: 10.1007/BF00941827.
5. Pytlak R., Malinowski K., *Optimal scheduling of reservoir releases during flood: Deterministic optimization problem, Part 2, Case Study.* "Journal of Optimization Theory and Applications", Vol. 61, No. 3, 1989, 433–449, DOI: 10.1007/BF00941828.
6. Pytlak R., Blaszczyk J., Karbowski A., Krawczyk K., Tarnawski T., *Solvers chaining in the IDOS server for dynamic optimization.* [In:] Proceedings of 52nd IEEE Annual Conference on Decision and Control (CDC), Florence, Italy, Dec. 10–13, 2013, 7119–7124, DOI: 10.1109/CDC.2013.6761018.

Studium porównawcze języków modelowania optymalizacyjnego AMPL, Pyomo i JuMP na przykładzie zadania sterowania falą powodziową

Streszczenie: Celem pracy jest badanie porównawcze trzech języków (środowisk) modelowania optymalizacyjnego: AMPL, Pyomo i JuMP. Porównanie jest oparte na trzech implementacjach zadania optymalnego sterowania falą powodziową z czasem dyskretnym, sformułowanego jako zadanie programowania nieliniowego. Przedstawione i omówione zostaną kody poszczególnych modeli oraz różnice między nimi. Uwzględnione zostaną różne aspekty, m.in. prostota i intuicyjność implementacji.

Słowa kluczowe: optymalizacja, języki modelowania, programowanie, sterowanie falą powodziową, programowanie nieliniowe, sterowanie optymalne

Andrzej Karbowski, PhD, DSc
A.Karbowski@ia.pw.edu.pl
ORCID: 0000-0002-8162-1575

Andrzej Karbowski received PhD (1990) and habilitation (2012) in Automatic Control and Robotics from Warsaw University of Technology, Faculty of Electronics and Information Technology. Currently he is an assistant professor at the Institute of Control and Computation Engineering of Warsaw University of Technology and at NASK National Research Institute in Warsaw. He is the editor and the co-author of two books (on parallel and distributed computing), the author and the co-author of two e-books (on grid computing and optimal control synthesis), the editor of a special issue of the „Energies" journal („Mixed-Integer Linear and Nonlinear Programming Methods for Energy Aware Traffic Control in Stationary Networks and Clouds") and the author over 150 journal and conference papers. His research interests concentrate on optimal control, MIP and MINLP methods, energy-aware data networks management, cybersecurity, decomposition and parallel implementation of optimization algorithms on multicore computers, clusters and clouds.

Krzysztof Wyskiel, BSc
k.wyskiel@protonmail.com
ORCID: 0000-0001-6851-9755

Krzysztof Wyskiel is an MSc student of Computer Science at the Faculty of Electronics and Information Technology of the Warsaw University of Technology.