DynaProg: Deterministic Dynamic Programming solver for finite horizon multi-stage decision problems

SoftwareX 14 (2021) 100690

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

DynaProg: Deterministic Dynamic Programming solver for finite horizon multi-stage decision problems

Federico Miretti *, Daniela Misul, Ezio Spessa

IC Engines Advanced Laboratory, Dipartimento Energia, Politecnico di Torino, c.so Duca degli Abruzzi 24, 10129 Torino, Italy

ARTICLE INFO

Article history:
Received 18 February 2021
Received in revised form 23 March 2021
Accepted 23 March 2021

Keywords:
Dynamic Programming
Optimal control
Decision problem

ABSTRACT

DynaProg is an open-source MATLAB toolbox for solving multi-stage deterministic optimal decision problems using Dynamic Programming. This class of optimal control problems can be solved with Dynamic Programming (DP), which is a well-established optimal control technique suited for highly non-linear dynamic systems. Unfortunately, the numerical implementation of Dynamic Programming can be challenging and time consuming, which may discourage researchers from adopting it. The toolbox addresses these issues by providing a numerically fast DP optimization engine wrapped in a simple interface that allows the user to set up an optimal control problem in a straightforward yet flexible environment, with no restrictions on the controlled system's simulation model. Therefore, it enables researchers to easily explore the usage of Dynamic Programming in their fields of expertise. Thorough documentation and a set of step-by-step examples complete the toolbox, thus allowing for easy deployment and providing insight of the optimization engine. Finally, the source code's class-oriented design allows researchers experienced in Dynamic Programming to extend the toolbox if needed.

Code metadata

Current code version	v1.1
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-21-00036
Code Ocean compute capsule	–
Legal Code License	MIT license
Code versioning system used	git
Software code languages, tools, and services used	MATLAB
Compilation requirements, operating environments & dependencies	MATLAB R2020a or newer
If available Link to developer documentation/manual	Documentation is included in the software package
Support email for questions	federico.miretti@polito.it

Software metadata

Current software version	v1.1
Permanent link to executables of this version	https://github.com/fmiretti/DynaProg
Legal Software License	MIT license
Computing platforms/Operating Systems	Linux, OS X, Microsoft Windows
Installation requirements & dependencies	MATLAB R2020a or newer
If available, link to user manual – if formally published include a reference to the publication in the reference list	Documentation is included in the software package
Support email for questions	federico.miretti@polito.it

1. Motivation and significance

Multi-stage optimal decision problems describe a wide class of control problems where decisions must be made in stages in

* Corresponding author.
E-mail addresses: federico.miretti@polito.it (Federico Miretti), daniela.misul@polito.it (Daniela Misul), ezio.spessa@polito.it (Ezio Spessa).

https://doi.org/10.1016/j.softx.2021.100690

order to minimize a certain total cost. As the stages progress, the system evolves based on its own dynamics which are influenced by the decisions themselves. If the system's evolution for each stage is fully predictable, the problem is deterministic. Instead, if the system's evolution is influenced by some random phenomena, the problem is stochastic. Moreover, the decision problem may be defined over a finite horizon or an infinite horizon, based on whether the number of stages is finite or not.

Dynamic Programming (DP) is a technique that is applied to the very wide field of optimal control in multi-stage decision problems [1]. Its implementation however is not straightforward: even in a deterministic scenario, its original form (Exact Dynamic Programming) can only be applied to a very limited number of cases because of its computational complexity. In most cases at least one technique from the broad field of Approximate Dynamic Programming must be adopted to obtain a numerical solution to the optimization problem (see e.g. [2]), mitigating the so-called *curse of dimensionality* (the issue of computational complexity easily exploding as the dimensionality of the problem increases) which affects all DP-based algorithms. Because of this, building an algorithm based on Dynamic Programming often involves tailoring it to a specific optimization problem (see e.g. the comprehensive set of methods and applications discussed in [3]) and requires both technical knowledge of the problem at hand and a deep understanding of DP and its implementation challenges.

DynaProg is a general-purpose MATLAB software package that allows to solve deterministic, finite horizon multi-stage decision problems with a user-friendly interface. The rationale behind its development is to provide a simple interface that allows the user to define an optimization problem of this class in a straightforward manner, without having to deal with the implementation of the optimization algorithm itself. Moreover, a fundamental aim of DynaProg is to provide a computationally fast optimization algorithm, as computational time is one of the main factors that limits the usage of Dynamic Programming algorithms.

This particular class of problems is found in many engineering and economic applications, and Dynamic Programming algorithms have been used with great benefits by many research communities studying various subjects such as vehicle routing problems [4], optimal control of hybrid-electric vehicles [5,6], battery health management [7], resource allocation problems [8], reservoir systems operation [9,10], hydrothermal coordination in power systems [11], sewer network management [12], and even natural ecosystems preservation [13].

Indeed, speed is DynaProg's main strength. The software was developed with state-of-the-art algorithms and it relies on features (e.g. vectorization, implicit expansion) and best coding practices (e.g. argument validation) which allow to exploit MATLAB's computational engine at its best as well as improve user-friendliness, based on our experience in the practical implementation of Dynamic Programming algorithms. Another prominent feature of the software is flexibility and ease of use. The user can define the problem structure with a simple and consistent naming structure, and he/she can define the system's dynamics with a simple MATLAB function (*m-file* [14]). Furthermore, advanced users can take advantage of DynaProg's well-documented, object-oriented design to fully customize and reuse the source code.

2. Software description

The core of the software is the DynaProg class, which allows to store the problem structure with its properties and solve it with its methods. The basic problem elements are a model for the system dynamics, a stage cost, and the initial state of the system. The state dynamics is a model which takes the form

$$x_{k+1} = f(x_k, u_k), \qquad k = 0, 1, \ldots, N-1, \tag{1}$$

where k indicates the current stage, N is the number of stages (the control horizon), x is the state of the system, and u is the control variable (the decision to be taken). Both x and u can be scalar or vector, if the system is characterized by several state variables or several control variables must be controlled. The stage cost takes the form $g(x_k, u_k)$, so that the total cost incurred in the entire process is

$$J(x_0, u_0, \ldots, u_{N-1}) = g_N(x_N) + \sum_{k=0}^{N-1} g(x_k, u_k), \tag{2}$$

where $g_N(x_N)$ is a terminal cost which may be incurred based on the final state of the system.

The main output of DynaProg are the optimal value of the total cost incurred in the entire decision problem

$$J^*(x_0) = \min_{\substack{u_k \in U_k(x_k) \\ k=0,\ldots,N-1}} J(x_0, u_0, \ldots, u_{N-1}) \tag{3}$$

and the optimal control sequence $u_0^*, \ldots, u_{N-1}^*$, that is the sequence of control variables that minimizes the total cost, subject to the constraint that each u_k^* must belong to the set of admissible control variables at stage k, i.e. $U_k(x_k)$.

The core of DynaProg is a deterministic Dynamic Programming optimization algorithm, which is divided in a *backward* phase and a *forward* phase.

In the backward phase, the algorithm iteratively builds the optimal *cost-to-go* for each stage:

$$J_k^*(x_k) = \min_{u_k \in U_k(x_k)} \left(g_k(x_k, u_k) + J_{k+1}^*\left(f_k(x_k, u_k)\right)\right). \tag{4}$$

The optimal cost-to-go function for each stage is the optimal cost associated to the tail sub-problem which involves only the stages from that to the last. In other words, the cost-to-go function $J_k^*(x_k)$ is the minimum cost incurred if the system must evolve from stage k to stage N, expressed as a function of the initial state x_k.

In general, it may not be possible to obtain an analytical expression for $J_k^*(x_k)$, which would be required for a solution based on Exact Dynamic Programming. For this reason, DynaProg requires the user to define discrete computational grids for the state and control variables. Then, for each iteration k during the backward phase, it evaluates $g_k(x_k, u_k) + J_{k+1}^*\left(f_k(x_k, u_k)\right)$ for all points belonging to the computational grids and it constructs a numerical approximation of $J_k^*(x_k)$ by linear interpolation.

In the backward phase, the system's evolution is simulated starting from the initial state x_0. For each stage, the optimal control variables are determined as

$$u_k^*(x_k) = \operatorname*{argmin}_{u_k \in U_k(x_k)} \left(g_k(x_k, u_k) + J_{k+1}^*\left(f_k(x_k, u_k)\right)\right) \tag{5}$$

and the simulation is advanced to the next stage, until the last stage is reached.

2.1. Software architecture

The interface by which the user is able to define and solve a multi-stage deterministic optimal decision problem is the DynaProg class. In order to set up the problem and its settings, an instance of the class can be created by calling the class constructor:

```
prob = DynaProg(__);
```

The input arguments of the class constructor are used to define some mandatory arguments such as the dynamic system to be controlled, the computational grids mentioned in Sections 2 and 2.2.2 and initial state values, as well as other optional arguments. The dynamic system in particular is passed to the class constructor as a *function handle* which points to a function contained in

an external *m-file*. This allows the user to code the dynamics of the system with the greatest amount of flexibility.

The problem object is returned as the output of the class constructor, and the settings defined by the user are stored as properties of that object. The problem structure can then be modified, if needed, by changing the values of the appropriate properties.

2.2. Software functionalities

2.2.1. Defining the system's dynamics

The user can define the system's dynamics and stage cost by writing a function in an *m-file*, with the following structure.

```
[x_next, stage_cost] = myfun(x, u)
```

Then, the model function is passed to the DynaProg class constructor as a *function handle*.

```
problem = DynaProg(__, @myfun)
```

If the model function is not time-invariant or, in other words, it is also a function of some time-dependent exogenous input w, the system's dynamics and stage cost are specified with the following alternate structure.

```
[x_next, stage_cost] = myfun(x, u, w)
```

The exogenous inputs w can include all variables that do not depend on the state and control variables, but may depend on time. It may also be simply time itself.

For some problems, the user may want to define constraints on the state and control variables that can be reached or selected by the system. To do this, DynaProg allows to define a third output to the model function which define, in the form of a logical variable, unfeasible states and/or control variables.

```
[x_next, stage_cost, unfeas] = myfun(x, u, w)
```

2.2.2. Defining the computational grids

DynaProg can be used to solve multi-stage optimal decision problems whether the state and control variables are discrete or continuous. However, the user must specify a discrete computational grid for each of the state and control variables, though for different reasons.

Control variables must be discretized and the resulting optimal control state trajectory will be restricted to the discrete grid specified by the user. This simplifies the numerical solution as the min and argmin operations mentioned in Eqs. (4) and (5) simplify to finding minimum values over finite sets.

State variables, on the other hand, do not get discretized in the forward simulation. However, building the cost-to-go function in the backward phase (as mentioned in Section 2) requires sampling it at a certain number of values for the state variables. For this reason, a discretized computational grid for the state variables is needed. The discretization of this computational grid for the state variables affects the computation of the cost-to-go function and, as such, it is a source of sub-optimality. Selecting the proper discretization level for the state variables grid usually requires some understanding of the physics behind the system under analysis and possibly some trial-and-error.

Both computational grids are entirely user-defined. The user can therefore decide whether to adopt uniform grids or experiment with non-uniform grids, which may allow to reduce the grid size but may also bias the solution if not properly designed. However, DynaProg does not currently include any functionality to automatically design a non-uniform computational grid and it is left to the user to do so based on his/her field expertise in the optimization problem he/she is studying.

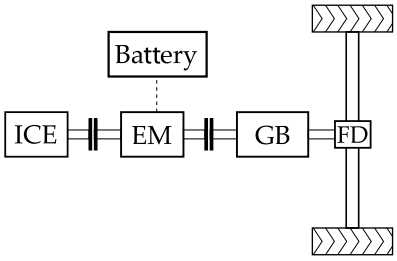

Fig. 1. Scheme of a p2 parallel HEV powertrain.

3. Illustrative example: optimal control of an hybrid electric vehicle

3.1. Problem definition

Consider a p2 parallel Hybrid Electric Vehicle (HEV) powertrain schematized in Fig. 1.

Assume that the vehicle must drive following a speed trace in time, such as the speed trace defined by a regulatory driving cycle, while minimizing the fuel consumption in order to maximize the benefits of hybridization.

The objective of this example is to design a control strategy which defines the gear number to be engaged by the gearbox and the torque that the electrical machine (EM) must provide or absorb. Furthermore, we must make sure that the battery's state of charge at the end of the drive cycle is equal to its initial value.

The vehicle speed and acceleration in time is treated as an exogenous input, while the control variables are the gear number and the EM torque ratio, defined as:

$$\tau = \frac{T_{EM}}{T_{req}}, \quad -1 \leq \tau \leq 1 \tag{6}$$

where T_{EM} is the torque provided (if positive) or absorbed (if negative) by the electrical machine and T_{req} is the torque requested at the powertrain level to make the vehicle following the given speed trace. As such, T_{req} is a function of the vehicle's speed and the gear number.

Since we must set a constraint on the terminal value of the battery's state of charge (SOC), we must be able to assign an initial condition to it and track its evolution in time (in other words, throughout the decision problem's stages). Thus, the battery SOC is set as a state variable, and we must define its dynamics.

The SOC dynamics is derived using a simple battery internal resistance model:

$$I_{batt} = \frac{V_{OC} - \sqrt{V_{OC}^2 - 4R_{eq}P_{batt}}}{2R_{eq}}, \tag{7}$$

$$\dot{SOC} = \frac{I_{batt}}{C_{batt}}. \tag{8}$$

Here I_{batt}, V_{OC}, R_{eq} and C_{batt} are the battery current, open-circuit voltage, equivalent resistance and capacity.

The battery power P_{batt} is evaluated as

$$P_{batt} = \begin{cases} \eta_{inv} P_{EM} & \text{if } P_{EM} \geq 0, \\ \frac{1}{\eta_{inv}} P_{EM} & \text{if } P_{EM} < 0, \end{cases} \tag{9}$$

and the power absorbed or generated by the electrical machine P_{EM} is evaluated as

$$P_{EM} = \begin{cases} \eta_{EM} \omega_{EM} T_{EM} & \text{if } T_{EM}\omega_{EM} \geq 0, \\ \frac{1}{\eta_{EM}} \omega_{EM} T_{EM} & \text{if } T_{EM}\omega_{EM} < 0, \end{cases} \tag{10}$$

with η_{EM} and η_{inv} being the efficiencies of the electrical machine and the inverter (or any other power electronics).

Finally, the engine's fuel consumption is evaluated based on the engine speed and torque with its fuel consumption map $\dot{m}_{fuel}(\omega_{eng}, T_{eng})$.

In order for the optimization problem to be meaningful, we must set a series of constraints that reflect the operational constraints of an actual powertrain.

The engine's torque cannot exceed its limit torque (which is in turn dependent on its speed):

$$T_{eng} = (1 - \tau)\, T_{req} \leq T_{eng,lim}(\omega_{eng}) \tag{11}$$

The electrical machine's torque must stay within its limit torque in generation and motor mode:

$$T_{EM,gen,lim} \leq T_{EM} \leq T_{EM,mot,lim} \tag{12}$$

When braking (i.e. $T_{req} < 0$), the electrical machine should never operate in motor mode.

The terminal SOC must be equal to its initial value:

$$SOC_0 = SOC_N \tag{13}$$

Also, the battery is characterized by a maximum discharge power and a maximum charge current that must not be exceeded:

$$P_{batt} \leq P_{batt,max} \tag{14}$$

$$I_{batt} \geq I_{batt,chrg,lim} \tag{15}$$

3.2. Defining the model function

First, the model function must be created by the user. In this example, the vehicle speed and acceleration are treated as exogenous inputs. Therefore, the basic function signature accepts three inputs (the state variables, the control variables, and the exogenous inputs) and three outputs (the updated state variables, the stage cost, and the unfeasibility tensor which is used to set the model constraints).

Listing 1: Basic model function signature.

```
function [x_new, stageCost, unfeas] = hev(x, u, w)
    ...
end
```

Also, for practical purposes, it is convenient to define all constant parameters that characterize the powertrain (such as the engine limit torque characteristic $T_{eng,lim}(\omega_{eng})$ and fuel consumption map $\dot{m}_{fuel}(\omega_{eng}, T_{eng})$, the electrical machine efficiency map $\eta_{EM}(\omega_{EM}, T_{EM})$, the gearbox speed ratios, etc.) outside of the model function and pass them to it as additional inputs. In this example, the vehicle data is stored in six structures (`veh`, `fd`, `gb`, `eng`, `em` and `batt`).

Listing 2: Model function signature modified to accept additional inputs.

```
function [x_new, stageCost, unfeas] = hev(x, u, w, veh, fd,
    gb, eng, em, batt)
    ...
end
```

Finally, it might be interesting to also include in the optimization results the time profiles of physical quantities other than the state variables, control variables and cost. This can be done by changing the function signature to return additional outputs starting from the fourth positional output.

Listing 3: Model function signature modified to return additional outputs.

```
function [x_new, stageCost, unfeas, engTrq, emTrq] = hev(x, u
    , w, veh, fd, gb, eng, em, batt)
    ...
end
```

The model function must perform all operations required to evaluate the updated state variables, the stage cost and to define constraints via the unfeasibility tensor. The full code for this example is included in the software documentation and can be accessed by entering `open('hev')` in MATLAB's command window.

3.3. Setting up and solving the optimization problem

The optimization problem must be set up and solved in a separate script. The script must define the discrete computational grids for state and control variables as well as initial conditions and (optionally) terminal constraints for the state variable. The optimization settings for this example are shown in Listing 4.

Listing 4: Preparing the optimization problem settings.

```
% State variable grid
SVnames = SOC;
x_grid = {0.4:0.005:0.7};
% Initial state
x_init = {0.6};
% Final state constraints
x_final = {[0.6 0.6]};

% Control variable grid
CVnames = [Gear Number, Torque split];
u1_grid = [1 2 3 4 5];
u2_grid = 1:0.1:1;
u_grid = {u1_grid, u2_grid};

% Load a drive cycle
load UDDS % contains velocity and time vectors
dt = time_s(2)   time_s(1);
% Create exogenous input
w{1} = speed_kmh./3.6;
w{2} = [diff(w{1})/dt; 0];

% Number of stages (time intervals)
Nint = length(time_s);

% Generate and store vehicle data
[veh, fd, gb, eng, em, batt] = data();
```

All the information that is needed to define the optimization problem is then used to create a problem structure, that is an instance of the `DynaProg` class, as shown in Listing 5. Note how the vehicle speed and acceleration are passed to the constructor as exogenous inputs.

Listing 5: Constructing the problem structure.

```
prob = DynaProg(x_grid, x_init, x_final, u_grid, Nint, @(x, u
    , w) hev(x, u, w, veh, fd, gb, eng, em, batt), '
    ExogenousInput', w);
```

The optimization problem can then be solved by using the `run` method, which returns the problem structure with simulation results. The same problem object can then be passed to the plot method to visualize the optimal state variables, control variables and cumulative cost trajectories (see Fig. 2), as shown in Listing 6.

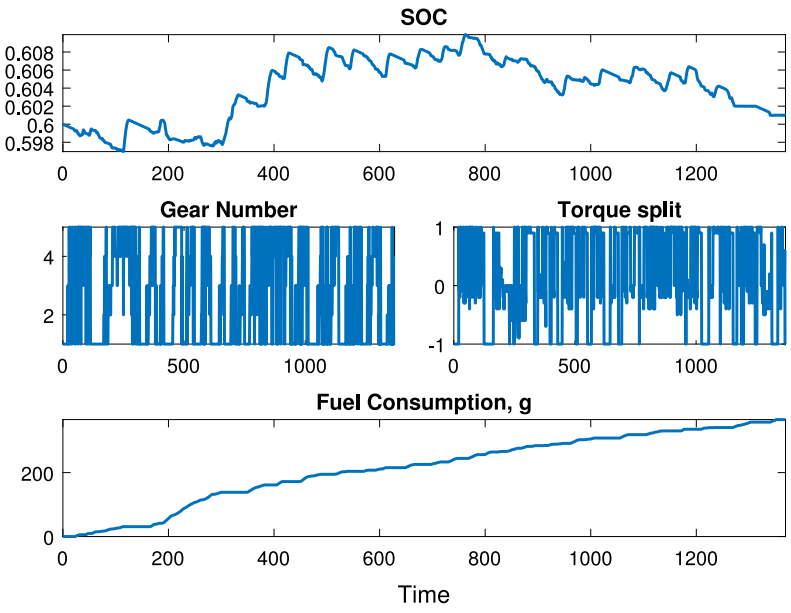

Fig. 2. Optimization results for the p2 HEV example.

Listing 6: Running the optimization problem.

```
prob = run(prob);
plot(prob)
```

4. Impact

There are many fields where Dynamic Programming is already a well-established method for solving the class of decision problems such as the ones listed in Section 1. For researchers in these fields, DynaProg may speed up research activity by allowing them to set up their own optimization problem and solve it with a fast algorithm without having to deal with the implementation challenges of Dynamic Programming. For the same reasons, DynaProg may also enable researchers to develop and experiment with new methods even in research fields where Dynamic Programming has not been previously used.

Currently, DynaProg is being used by sustainable mobility groups at Politecnico di Torino to investigate optimal control strategies for hybrid-electric vehicles and optimal design of hybrid-electric powertrains, and it is also under evaluation at other institutions. In this context, DynaProg has already proven its worth by enabling relatively complex models to be explored with significantly reduced computational times.

The DynaProg package provides an easy, flexible, well-documented and computationally fast tool that allows researchers to obtain the (approximate) global solution for any finite horizon, multi-stage deterministic optimal decision problems, regardless of the field of application. The package syntax and documentation was carefully designed to prevent it from being tied to a specific research topic. The authors hope that these features should stimulate the adoption of Dynamic Programming-based optimization in new research fields.

Moreover, DynaProg's code was specifically designed with transparency and reusability in mind. One of the main design goals was allowing to extend and customize both its user interface and computational core. The reason for this is that this will allow researchers who work in fields where Dynamic Programming optimization algorithms are already an established practice to improve their understanding of their computational hazards and even develop their own optimization algorithm tailored to their own research objectives.

5. Conclusions

DynaProg was designed to provide a reliable, versatile and well documented tool in multi-stage deterministic optimal decision problems. Its development stems from the authors' experience in optimal control of Hybrid-Electric Vehicles, but the tool and the documentation was specifically designed to make it problem-independent.

We hope that this will enable many researchers facing this class of optimization problems to exploit Dynamic Programming in their research without having to invest a long time in becoming experienced in this technique.

By making the code fully open source, we also hope that those researchers who are instead experienced in Dynamic Programming will be able to contribute with their own extensions or improvements. Examples of potential extensions include replacing the functional approximator for the cost-to-go function mentioned in Section 2 with more sophisticated solutions (such as the ones mentioned in [15]) or embedding tools for assisted/automated creation of non-uniform computational grids.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] Bertsekas DP. Dynamic programming and optimal control, vol. 1. 4th ed. Belmont, Massachusetts: Athena Scientific; 2016.
[2] Bertsekas DP. Dynamic programming and optimal control, vol 2. 4th ed. Belmont, Massachusetts: Athena Scientific; 2012.
[3] Brandimarte P. From shortest paths to reinforcement learning: a MATLAB-based tutorial on dynamic programming. 2021.
[4] Novoa C, Storer R. An approximate dynamic programming approach for the vehicle routing problem with stochastic demands. European J Oper Res 2009;196(2):509–15. http://dx.doi.org/10.1016/j.ejor.2008.03.023.
[5] Pérez LV, Bossio GR, Moitre D, García GO. Optimization of power management in an hybrid electric vehicle using dynamic programming. In: Applied and computational mathematics - selected papers of the fifth PanAmerican workshop - June 21–25, 2004, Tegucigalpa, Honduras. Math Comput Simulation 2006;73(1):244–54. http://dx.doi.org/10.1016/j.matcom.2006.06.016.

[6] Brahma A, Guezennec Y, Rizzoni G. Optimal energy management in series hybrid electric vehicles. In: Proceedings of the 2000 American control conference, vol. 1. 2000, p. 60–4. http://dx.doi.org/10.1109/ACC.2000.878772.
[7] Moura SJ, Forman JC, Bashash S, Stein JL, Fathy HK. Optimal control of film growth in lithium-ion battery packs via relay switches. IEEE Trans Ind Electron 2011;58(8):3555–66. http://dx.doi.org/10.1109/TIE.2010.2087294.
[8] Forootani A, Iervolino R, Tipaldi M, Neilson J. Approximate dynamic programming for stochastic resource allocation problems. IEEE/CAA J Autom Sin 2020;7(4):975–90. http://dx.doi.org/10.1109/JAS.2020.1003231.
[9] Rani D, Moreira MM. Simulation–optimization modeling: A survey and potential application in reservoir systems operation. Water Resour Manag 2010;24(6):1107–38. http://dx.doi.org/10.1007/s11269-009-9488-0.
[10] Zhang Z, Zhang S, Wang Y, Jiang Y, Wang H. Use of parallel deterministic dynamic programming and hierarchical adaptive genetic algorithm for reservoir operation optimization. Comput Ind Eng 2013;65(2):310–21. http://dx.doi.org/10.1016/j.cie.2013.02.003.
[11] Yang Jin-Shyr, Chen Nanming. Short term hydrothermal coordination using multi-pass dynamic programming. IEEE Trans Power Syst 1989;4(3):1050–6. http://dx.doi.org/10.1109/59.32598.
[12] Abraham DM, Wirahadikusumah R, Short TJ, Shahbahrami S. Optimization modeling for sewer network management. J Constr Eng Manag 1998;124(5):402–10. http://dx.doi.org/10.1061/(ASCE)0733-9364(1998)124:5(402).
[13] Odom DIS, Cacho OJ, Sinden JA, Griffith GR. Policies for the management of weeds in natural ecosystems: the case of scotch broom (Cytisus scoparius, L.) in an Australian national park. Ecol Econom 2003;17.
[14] The Mathworks Inc. MATLAB primer (user's guide). 2020.
[15] Gaggero M, Gnecco G, Sanguineti M. Dynamic programming and value-function approximation in sequential decision problems: Error analysis and numerical results. J Optim Theory Appl 2013;156(2):380–416. http://dx.doi.org/10.1007/s10957-012-0118-2.