Digital Receipt

This receipt acknowledges that Turnitin received your paper. Below you will find the receipt information regarding your submission.

The first page of your submissions is displayed below.

Submission author:	Auralius Manurung
Assignment title:	No repository
Submission title:	YADPF: A reusable deterministic dynamic programming impl...
File name:	B.A.JI.6_Uddin_2022_Software_X.pdf
File size:	781.93K
Page count:	6
Word count:	4,618
Character count:	22,889
Submission date:	10-Apr-2022 06:43AM (UTC-0700)
Submission ID:	1806624123

SoftwareX 17 (2022) 101001

Contents lists available at ScienceDirect

SoftwareX

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

Original software publication

YADPF: A reusable deterministic dynamic programming implementation in MATLAB

Auralius Manurung [a,*], Lisa Kristiana [b], Nur Uddin [c]

[a] Universitas Pertamina, Jakarta, 12220, Indonesia.
[b] Institut Teknologi Nasional Bandung, Bandung, 40124, Indonesia
[c] Universitas Pembangunan Jaya, Banten, 15413, Indonesia

ARTICLE INFO

Article history:
Received 24 November 2021
Received in revised form 15 January 2022
Accepted 25 January 2022

Keywords:
Dynamic programming
Optimal control
Dynamic optimization
Reinforcement learning

ABSTRACT

This paper introduces the YADPF package, a collection of reusable MATLAB functions to solve deterministic discrete-time optimal control problems using a dynamic programming algorithm. For finite- and infinite-horizon optimal control problems, two types of dynamic programming algorithms are implemented: backward dynamic programming and value iteration. Like other implementations, users must provide the discretized state and input variables, the model dynamic equation, the terminal cost function, and the stage cost function. To more motivate users to use this MATLAB function package, we also provide more than ten academic case studies on how the YADPF function package can solve dynamic optimization problems with detailed step-by-step instructions. The provided guides and examples are expected to help users, especially, students and researchers initiate instant dynamic programming experiences with minimal coding expertise.

Current code version	v1.0.10
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-21-00222
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	–
If available Link to developer documentation/manual	https://auralius.github.io/yadpf/
Support email for questions	auralius.manurung@eee.org

1. Motivation and significance

Dynamic programming (DP) was first introduced in [1] to solve optimal control problems (OCPs) where the solution is a sequence of inputs within a predefined time horizon that maximizes or minimizes an objective function. This is known as dynamic optimization or multistage decision problem. There are many examples of how DP are used in real applications, such as in energy management systems [2] and in resource allocation problems [3].

* Corresponding author.
E-mail address: auralius.manurung@eee.org (Auralius Manurung).

Since the introduction of DP, there have been many variations of DP. In a broader sense, DP can be classified into two categories: exact dynamic programming (EDP) and approximate dynamic programming (ADP). There are very few EDP implementations as MATLAB reusable functions, such as in [4,5]. Besides these two implementation which is only designed for deterministic OCPs, there is also a more sophisticated toolbox with several DP algorithms implemented that can be used for both stochastic and deterministic OCPs [6].

Besides EDP and ADP, other methods which use nonlinear programming (NLP) techniques can also be used to solve OCPs. In fact, ADP and NLP are more suitable for a complex system, as opposed to EDP. On another hand, EDP is more common in

https://doi.org/10.1016/j.softx.2022.101001
2352-7110/

YADPF: A reusable deterministic dynamic programming implementation in MATLAB

by Auralius Manurung

Submission date: 10-Apr-2022 06:43AM (UTC-0700)
Submission ID: 1806624123
File name: B.A.JI.6_Uddin_2022_Software_X.pdf (781.93K)
Word count: 4618
Character count: 22889

SoftwareX 17 (2022) 101001

Contents lists available at ScienceDirect

SoftwareX

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

Original software publication

YADPF: A reusable deterministic dynamic programming implementation in MATLAB

Auralius Manurung [a,*], Lisa Kristiana [b], Nur Uddin [c]

[a] Universitas Pertamina, Jakarta, 12220, Indonesia,
[b] Institut Teknologi Nasional Bandung, Bandung, 40124, Indonesia
[c] Universitas Pembangunan Jaya, Banten, 15413, Indonesia

ARTICLE INFO

Article history:
Received 24 November 2021
Received in revised form 15 January 2022
Accepted 25 January 2022

Keywords:
Dynamic programming
Optimal control
Dynamic optimization
Reinforcement learning

ABSTRACT

This paper introduces the YADPF package, a collection of reusable MATLAB functions to solve deterministic discrete-time optimal control problems using a dynamic programming algorithm. For finite- and infinite-horizon optimal control problems, two types of dynamic programming algorithms are implemented: backward dynamic programming and value iteration. Like other implementations, users must provide the discretized state and input variables, the model dynamic equation, the terminal cost function, and the stage cost function. To more motivate users to use this MATLAB function package, we also provide more than ten academic case studies on how the YADPF function package can solve dynamic optimization problems with detailed step-by-step instructions. The provided guides and examples are expected to help users, especially, students and researchers initiate instant dynamic programming experiences with minimal coding expertise.

Current code version	v1.0.10
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-21-00222
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	–
If available Link to developer documentation/manual	https://auralius.github.io/yadpf/
Support email for questions	auralius.manurung@ieee.org

1. Motivation and significance

Dynamic programming (DP) was first introduced in [1] to solve optimal control problems (OCPs) where the solution is a sequence of inputs within a predefined time horizon that maximizes or minimizes an objective function. This is known as dynamic optimization or multistage decision problem. There are many examples of how DP are used in real applications, such as in energy management systems [2] and in resource allocation problems [3].

Since the introduction of DP, there have been many variations of DP. In a broader sense, DP can be classified into two categories: exact dynamic programming (EDP) and approximate dynamic programming (ADP). There are very few EDP implementations as MATLAB reusable functions, such as in [4,5]. Besides these two implementation which is only designed for deterministic OCPs, there is also a more sophisticated toolbox with several DP algorithms implemented that can be used for both stochastic and deterministic OCPs [6].

Besides EDP and ADP, other methods which use nonlinear programming (NLP) techniques can also be used to solve OCPs. In fact, ADP and NLP are more suitable for a complex system, as opposed to EDP. On another hand, EDP is more common in

* Corresponding author.
E-mail address: auralius.manurung@ieee.org (Auralius Manurung).

https://doi.org/10.1016/j.softx.2022.101001

an academic environment for less complex systems due to the exactness of the provided solutions and the guarantee for global optimality [7].

As previously mentioned, when the systems are complex, ADP and NLP are more actually favorable with many implementations available both commercially and non-commercially. The reason for such a popularity is because ADP and NLP are more computationally efficient than EDP [8]. EDP visits all possible state values and tests them with all possible input values, which makes EDP a very resource-demanding method [7].

Additionally, many researchers often choose to develop their own EDP implementation, which is tailored specifically to solve one particular dynamic optimization problem. Their implementations are often equipped with advanced features, such as with adaptive discretization [9]. However, there is no publicly available implementations.

Therefore, we decided to add a new item into the database of reusable exact dynamic programming functions by proposing another implementation of DP (backward DP and value iteration). We named our implementation with the YADPF: Yet Another Dynamic Programming Function. We strictly based our implementation on Bellman's basic dynamic programming algorithm for deterministic OCPs. Thus, our work will be in the same group as in [4,5].

We selected MATLAB since it is prevalent among control engineers and researchers. In our MATLAB implementation, we strive for fast and efficient performance by exploiting the vectorization features that MATLAB offers. We also include the YADPF package with many academic examples which we will discuss in later part of this paper.

2. Software description

Backward DP in YADPF package is used to solve an OCP with the following formulation.

$$\mathrm{P}: \begin{cases} \min\limits_{u_k} & g_N(x_N) + \sum_0^{N-1} g_k(x_k, u_k) \\ \text{subject to} & x_{k+1}^{\uparrow M} = f(x_k^{\uparrow M}, u_k^{\uparrow M}, k) \\ & x_0 = A \quad \text{and} \quad x_N = B \\ & k = 0, 1, \ldots, N-1 \end{cases} \tag{1}$$

The results obtained from solving the OCP P above is a control sequence u_k defined along a finite time horizon (horizon length of $N+1$, from $k=0$ to $k=N$).

In Eq. (1), we refer $x_k^{\uparrow M}$ as the signal x up-sampled by factor of M using the zero order hold (ZOH) interpolation. Further, $x_{k+1} = f(x_k, u_k, k)$ describes the system's dynamics whereas the stage cost and the terminal cost are described by g_k and g_N, respectively. As for x_k and u_k, these two variables represent the state variables and the input variables, respectively. The initial state is given by x_0 and the final state is given by x_N. The state variables, the input variables, and the time are bounded and discretized. The discretized states are called the nodes and the discretized time are called the stages.

Besides backward DP, the YADPF package is also equipped with value iteration algorithm to solve an OCP that does not explicitly have a predefined horizon length and terminal cost. Such an OCP is formulated in Eq. (2).

$$\mathrm{Q}: \begin{cases} \min\limits_{u_k} & \lim\limits_{N\to\infty} \sum_0^{N-1} \gamma g_k(x_k, u_k) \\ \text{subject to} & x_{k+1}^{\uparrow M} = f(x_k^{\uparrow M}, u_k^{\uparrow M}, k) \\ & x_0 = A \\ & k = 0, 1, \ldots \\ & 0 < \gamma \leq 1 \end{cases} \tag{2}$$

In Eq. (2), γ is the discounting factor which reflects how the future values are being valued. The smaller γ value contributes to the faster convergence but less-accurate solutions.

2.1. Software architecture

Backward DP starts from the terminal stage and moves to the initial stage. The algorithm visits all nodes in each step and calculates the stage cost. Before calculating the stage cost, the algorithm first calculates the system's dynamic one step forward. This whole process is iterative and can be made faster.

To make the process above faster, in our implementation, we first calculate the system's dynamic one step forward for all existing nodes. We visit all nodes and apply all inputs to the system's dynamical model. We then keep the results in a lookup table for later use during the stage cost calculation. With this lookup table, we can avoid this repetitive computation of the system's dynamics and avoid using a for-loop. As a result, we now have a matrix operation instead of having a for-loop. Thus, we can unleash the full potential of MATLAB's engine for matrix computations.

However, the process of creating a look-up table can easily consume all computation power, including memory space causing the whole system to be unresponsive. This issue is likely to happen when working with a high-dimension system in a less-powerful computation system. It is also interesting to notice that this process might be pretty similar to bootstrapping process in reinforcement learning. However, in reinforcement learning, the bootstrapping process involves a more complex estimation process [10].

Besides backward DP, in YADPF, we also implement value iteration, which is in principal a variation of DP implementation. Unlike backward DP, the iteration in value iteration is not stage-based. The iteration terminates based on the convergence of some calculated costs. Thus, value iteration is an infinite horizon DP. More details on value iteration can be found in a popular textbook by Bertsekas' [11].

With value iteration, we can build a control table (policy matrix) such that a dynamical system optimally evolves from any given initial state to a targeted terminal state. Compared to backward DP, value iteration demands less memory and requires no predefined horizon length. Therefore, it becomes a better alternative for infinite horizon optimization problem.

2.2. Software functionalities

The implementation of DP requires a discretized simulation environment. In this discretized environment, the states, inputs, and time are all discretized. When using the YADPF package, users are responsible for the discretization process.

In addition to the discretization process, users must create three functions that look like the following.

```
function x_next = state_update_fn(X, U, dt)
  % Describe the system dynamics here
end

function J = stage_cost_fn(X, U, k, dt)
  % Describe the stage cost function here
end

function J = terminal_cost_fn(X)
  % Desired terminal state cost function here
end
```

Listing 1: The structure of the state update, stage cost, and terminal cost functions.

In Listing 1, `state_update_fn`, `stage_cost_fn`, and `terminal_cost_fn` are the state update function, stage cost function and the terminal cost function, respectively. The parameters: `X`, `U`, and `dt` represent state variables, input variables, and time interval for the OCP, respectively. It is important to notice that the function `state_update_fn` can only accept non-autonomous dynamical system since there is no information on the current stage available. However, in `stage_cost_fn` function, there is parameter `k`, which is the number of the current stage and can be used to handle time-weighted objective function.

In the next step, handles to the three previously mentioned functions are then registered to a data structure (named as `dpf`). We then send this data structure to the DP solver, as shown in the Listing 2 below.

```
% Setup: 2 states and 1 input
P = -1.2  : 0.001  : 0.5;
V = -0.07 : 0.0001 : 0.07;
U = [-1 0 1];

dpf.states           = {P V};
dpf.inputs           = {U};
dpf.T_ocp            = 1;
dpf.T_dyn            = 1;
dpf.n_horizon        = 100;
dpf.state_update_fn  = @state_update_fn;
dpf.stage_cost_fn    = @stage_cost_fn;
dpf.terminal_cost_fn = @terminal_cost_fn;

% Create and run the solver
dpf = yadpf_solve(dpf);

% Trace, initial states: [-0.5 0]
dpf = yadpf_trace(dpf, [-0.5 0]);

% Plot the results
yadpf_plot(dpf, '-');
```

Listing 2: In backward DP, a structure is used to hold a necessary information.

In Listing 2, line 9 to 16, we can see a simple data structure `dpf` is being used to hold all information on the OCP. Next, `yadpf_solve` function is called where the DP is implemented. Notice that backward DP solves an OCP for all possible initial state values (nodes). Thus, we have to trace the specific solution for the initial state values that we are interested with. This process is done with the `yadpf_trace` function. Finally, we can plot the results by using `yadpf_plot` function.

As for value iteration, we must also discretize the OCP as in backward DP. However, unlike in backward DP, value iteration requires only two user-defined functions: the state update function, and the stage cost function. The prototypes of these two functions are identical with those in backward DP (see Listing 1). Horizon length is no longer needed. Instead, a new variable is introduced for setting the maximum number of iterations (see Listing 3, line 10).

```
% Setup: 2 states and 1 input
P = -1.2  : 0.001  : 0.5;
V = -0.07 : 0.0001 : 0.07;
U = [-1 0 1];

dpf.states           = {P V};
dpf.inputs           = {U};
dpf.T_ocp            = 1;
dpf.T_dyn            = 1;
dpf.max_iter         = 10000;
dpf.state_update_fn  = @state_update_fn;
dpf.stage_cost_fn    = @stage_cost_fn;

% Create and run the solver
dpf = yadpf_visolve(dpf, 0.99);

% Trace, initial states: [-0.5 0]
dpf = yadpf_vitrace(dpf, [-0.5 0]);

% Plot the results
yadpf_plot(dpf, '-');
```

Listing 3: MATLAB code for value iteration in the YADPF is very similar to backward DP.

Value iteration is executed in line 19 of Listing 2. Here, we selected $\gamma = 0.9$. Like backward DP, value iteration solves an OCP for all possible initial state values (nodes). Thus, we have to trace the specific solution for the initial state values that we are interested in by using the `yadpf_vitrace` function. Finally, plotting the results can be done by using the same plotting function as in backward DP: `yadpf_plot`.

3. Illustrative examples

In this section, we present two sets of complete working codes to demonstrate the functionalities of the YADPF package: the mass–damper's time-optimal control problem and the Sutton's mountain car problem. In both problems, we use the YADPF package to plan for time-optimal motions to reach the given targets.

Besides these two problems, the YADPF package includes several more academic-oriented examples, such as the stabilization of an F8 aircraft [12–14], Dubin's car [15,16], hanging piecewise mass–spring system [17], Lotka–Volterra fishery [5,18], a stirred-tank mixer [19], and finding the shortest path on a terrain.

3.1. The mass–damper's optimal control problem

Assume that we have a mass ($m = 1$ [kilogram]) and a damper ($b = 0.1$ [Newton×second/meter]) with two state variables (position x_k [meter] and velocity v_k [meter/second]), and one input variable (external force f_k [Newton]). The applied force that can be applies ranges from -4 Newtons to 4 Newtons. Our goal is to move the mass from $x = 0$ to $x = 0.5$ as fast as possible but with minimal input (energy). When the mass arrives at the target position, its speed should be as close as possible to zero.

Based on the statements above, we can formulate the OCP as follows.

$$
\begin{aligned}
&\min_{f_k, x_N, v_N} \ \alpha_1 \sum_{k=0}^{N-1} f_k^2 + \alpha_2 \left(x_f - x_N\right)^2 + \alpha_3 \left(v_f - v_N\right)^2 \\
&\text{subject to :} \\
&\begin{bmatrix} x_{k+1} \\ v_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 - \frac{b}{m}\Delta t \end{bmatrix} \begin{bmatrix} x_k \\ v_k \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{m}\Delta t \end{bmatrix} f_k \\
&f_k = F_k \in \{-4, -3.9, -3.8, \ldots, 3.8, 3.9, 4\} \\
&x_k = X_k \in \{0, 0.001, 0.002, \ldots, 1\} \\
&v_k = V_k \in \{0, 0.001, 0.002, \ldots, 1\} \\
&x_f = 0.5 \\
&v_f = 0
\end{aligned}
\tag{3}
$$

In Eq. (3). α_1, α_2, and α_3 are the control gains for the force input, the position and the velocity, respectively, whose values are tuned heuristically. Let us set the sampling period for the OCP to 0.1 s and for the dynamic simulation to 0.01 s. Listing 4 contains MATLAB implementation of the OCP in Eq. (3) by using the YADPF package.

```
% Setup the states and the inputs
X =  0 : 0.001 : 1; % Position
V =  0 : 0.001 : 1; % Velocity
F = -4 : 0.1   : 4; % Applied force

% Setup the horizon
Tf = 1;
T_ocp = 0.1;
t  = 0 : T_ocp : Tf;
```

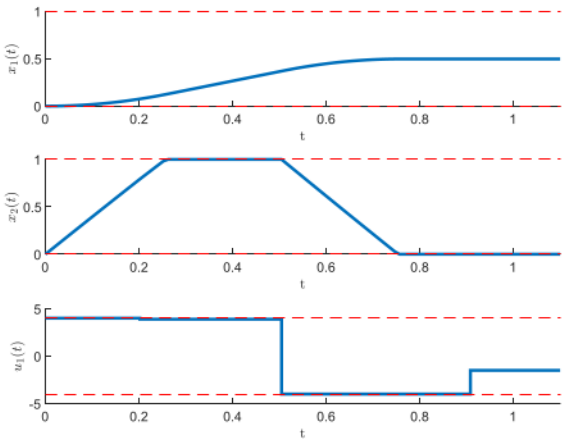

Fig. 1. Time-optimal motion of a mass–damper system, computed by backward DP.

```
dpf.states            = {X, V};
dpf.inputs            = {F};
dpf.T_ocp             = T_ocp;
dpf.T_dyn             = 0.01;
dpf.n_horizon         = length(t);
dpf.state_update_fn   = @state_update_fn;
dpf.stage_cost_fn     = @stage_cost_fn;
dpf.terminal_cost_fn = @terminal_cost_fn;

% Run, trace, and plot
% Initial states: [0 0]
dpf = yadpf_solve(dpf);
dpf = yadpf_trace(dpf, [0 0]);
yadpf_plot(dpf, '-');

% Optional: draw the reachability plot
yadpf_rplot(dpf, [0.5 0], 0.1);

%% The state unpdate funtion
function X = state_update_fn(X, F, dt)
m = 1;   % Mass
b = 0.1; % Damping coefficient

X{1} = X{1} + dt*X{2};
X{2} = X{2} - b/m*dt.*X{2} + dt/m.*F{1};
end

%% The stage cost function
function J = stage_cost_fn(X, F, k, dt)
J = dt*F{1}.^2;
end

%% The terminal cost function
function J = terminal_cost_fn(X)
xf = [0.5 0];

% Control gains
alpha1 = 1000;
alpha2 = 100;

J = alpha1*(X{1}-xf(1)).^2 + ...
    alpha2*(X{2}-xf(2)).^2;
end
```

Listing 4: Time-optimal motion of a mass–damper system with backward DP.

We first guess the horizon length in Listing 4, lines 7 to 9 since it is not known yet. In a typical time-optimal control problem, we can avoid guessing by using value iteration. Value iteration and backward DP on the problem described in Eq. (3) give us very similar results, as shown in Fig. 1. While backward DP generates a reachability plot (see Fig. 2), value iteration generates a policy matrix that can also be presented as a plot (see Fig. 3).

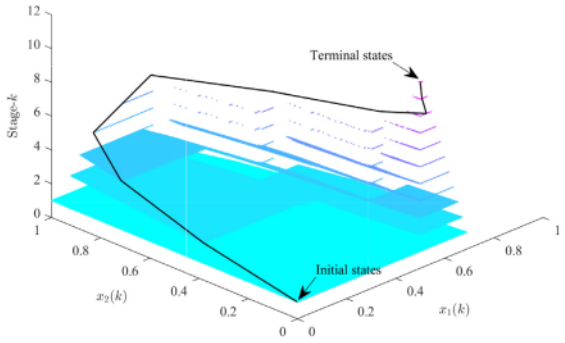

Fig. 2. Reachability plot by backward DP for the time-optimal motion of the mass–damper system.

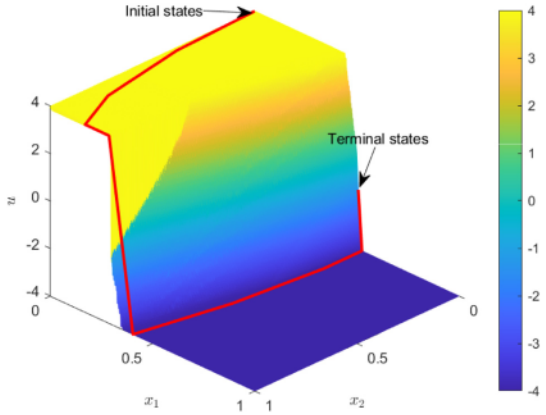

Fig. 3. Policy matrix generated by value iteration for the time-optimal motion of the mass–damper system.

Importantly, we would like to point out that the present OCP is not a solid time-optimal control problem. Implementing a time-optimal control problem in DP is not a straightforward process. The applied objective function here minimizes the sum of squared errors along a predefined time horizon. However, switching actions appear for the input sequence by applying very high control gains. Bang–bang action typically appears in a time-optimal control problem. Nevertheless, this argument requires further validation.

3.2. Sutton's mountain car problem

The mountain car problem is a widespread problem that was initially proposed by Moore in [20] and popularized by Sutton and Barto in their textbook [21]. Since then, it has become a common toy problem for reinforcement learning algorithm testings with many variations. The problem that we use in this paper is similar to the problem found in [21]. Reformulating Sutton's mountain car problem as an infinite-horizon OCP with a discount factor of

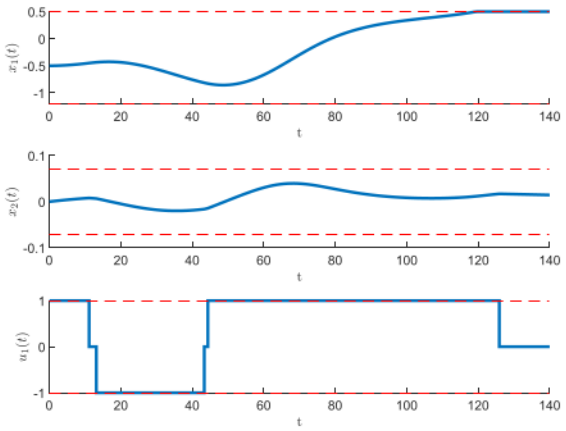

Fig. 4. The optimal solution on Sutton's mountain car problem, computed by backward DP.

γ gives us Eq. (4).

$$\min_{u_k} \gamma \left\{ \alpha_1 (x_N - 0.5)^2 + \alpha_2 \dot{x}_N \right\}$$
$$\text{subject to:}$$
$$x_{k+1} = x_k + \dot{x}_{k+1}$$
$$\dot{x}_{k+1} = \dot{x}_k + 0.001u_k - 0.0025\cos(3x_k) \quad (4)$$
$$x_k \in X_k = \{-1.2, -1.199, \ldots, 0.5\}$$
$$\dot{x}_k \in \dot{X}_k = \{-0.07, -0.0069, \ldots, 0.07\}$$
$$u_k \in U_k = \{-1, 0, 1\}$$
$$k = 0, 1, \ldots$$

As can bee seen from Eq. (4), γ is the discount factor which is set to 0.99. The car has two state variables: the car's position (x_k) and velocity (v_k); and one input variable: the car's engine throttle (u_k). Two control gains are introduced, α_1 and α_2, in order to regulate the car's position and velocity, respectively. These control gains are tuned heuristically. Listing 5 contains MATLAB implementation of the OCP in Eq. (4) by using the YADPF package. The results are presented in Figs. 4 and 5.

```
% Setup the states and the inputs
P = -1.2  : 0.001  : 0.5;
V = -0.07 : 0.0001 : 0.07;
U = [-1 0 1];

dpf.states      = {P V};
dpf.inputs      = {U};
dpf.T_ocp       = 1;
dpf.T_dyn       = 1;
dpf.max_iter    = 1500;
dpf.state_update_fn = @state_update_fn;
dpf.stage_cost_fn   = @stage_cost_fn;

% Run and trace
% From [-0.5 0] to [0.5 0]
dpf = yadpf_visolve(dpf, 0.99);
dpf = yadpf_vitrace(dpf, [-0.5 0], [0.5 0]);
yadpf_plot(dpf, '-');

% Policy plot
yadpf_pplot(dpf)

%% The state update function
function X = state_update_fn(X, U, ~)
X{2} = X{2} + 0.001*U{1} - 0.0025*cos(3*X{1});
X{1} = X{1} + X{2};

% Hitting the left wall
[r,c] = find(X{1}(:,:) <= -1.2);
X{2}(r,c) = 0.001; % Inelastic wall

% Hitting the right wall
[r,c] = find(X{1}(:,:) >= 0.5);
X{2}(r,c) = 0; %  Stop!
end

%% The stage cost function
function J = stage_cost_fn(X, U, k, ~)
alpha1 = 1000;
alpha2 = 1000;

J = alpha1*(X{1}-0.5).^2 + alpha2*X{2}.^2;
end
```

Listing 5: Sutton's mountain car with value iteration.

4. Impact

DP provides golden standards in dynamic optimization due to the exactness of the solution that it provides. The sub-optimality of the solutions is caused by the limitations introduced during the discretization process. However, DP requires a large memory capacity, making it unsuitable for complex systems.

Therefore, our DP implementation is oriented towards the academic environment: for learning, teaching, and research. We also add extra capabilities to generate two technical plots: reachability and policy-matrix plots. These plots can be generated for low dimension systems with one and two state variables. Moreover, with value iteration implemented in addition to backward DP, the YADPF package can address both finite and infinite horizon OCPs.

We are currently using the YADPF package for theoretical research in optimal controls. We have more flexibility in implementing time-optimal control for nonlinear systems thanks to separate sampling periods for the OCP and the dynamic simulation. The selection of OCP's sampling period may affect the

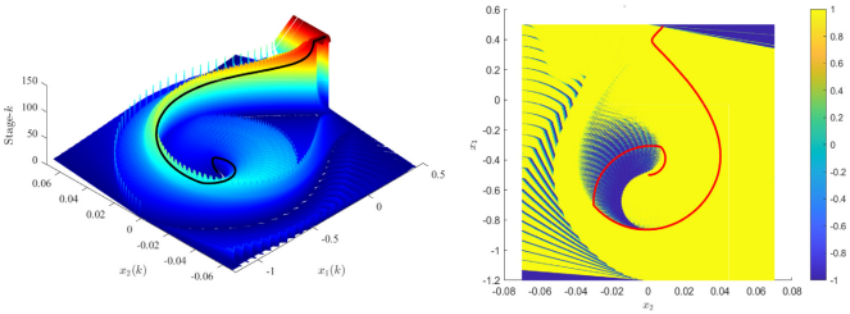

Fig. 5. Left figure shows the very complicated reachability plot with many unconnected regions generated by backward DP. The right figure shows the policy matrix generated by value iteration. Very similar optimal state trajectories are observed with the two methods.

switching actions that typically appear in a time-optimal control problem.

The YADPF source code had been made public to the MATLAB community, along with detailed documentation on how to use it. Thus, we are sure that the YADPF package can benefit researchers in dynamic programming and optimization, especially those with relatively weak backgrounds in optimal control theory.

5. Conclusion

This paper promotes dynamic programming in general and specifically the YADPF package for a generic dynamic programming implementation in MATLAB. The introduced YADPF package enables students and researchers to solve dynamic optimization problems with finite and infinite horizons. In this paper, we have also shown that it is relatively easy to use the YADPF package. Therefore, it can become an efficient tool for research, learning, and teaching in the area of dynamic optimization as well as in reinforcement learning.

In the near future, we plan to use the YADPF package for teaching advanced elective courses at the university while continuously adding more solved problems in the documentation as examples. We expect to expose the YADPF package to many different use scenarios. Further, we already have several development ideas for the YADPF package for our long-term plan. The first is to implement a selected variation of ADP and the second is to implement a stochastic DP.

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] Bellman R. Dynamic programming. Princeton, New Jersey, USA: Princeton University Press; 1957.

[2] Jamal S, Tan NML, Pasupuleti J. A review of energy management and power management systems for microgrid and nanogrid applications. Sustainability 2021;13(18):10331. http://dx.doi.org/10.3390/su131810331.

[3] Forootani A, Iervolino R, Tipaldi M, Neilson J. Approximate dynamic programming for stochastic resource allocation problems. IEEE/CAA J Automat Sinica 2020;7(4):975–90. http://dx.doi.org/10.1109/JAS.2020.1003231.

[4] Miretti F, Misul D, Spessa E. DynaProg: Deterministic dynamic programming solver for finite horizon multi-stage decision problems. SoftwareX 2021;14:100690. http://dx.doi.org/10.1016/j.softx.2021.100690.

[5] Sundstrom O, Guzzella L. A generic dynamic programming matlab function. In: 18th IEEE international conference on control applications, no. 7. Saint Petersburg, Russia; 2009, p. 1625–30. http://dx.doi.org/10.1109/CCA.2009.5281131.

[6] Chadès I, Chapron G, Cros M-J, Garcia F, Sabbadin R. MDPtoolbox: a multi-platform toolbox to solve stochastic dynamic programming problems. Ecography 2014;37(9):916–20. http://dx.doi.org/10.1111/ecog.00888.

[7] Elbert P, Ebbesen S, Guzzella L. Implementation of dynamic programming for n-dimensional optimal control problems with final state constraints. IEEE Trans Control Syst Technol 2013;21(3):924–31. http://dx.doi.org/10.1109/TCST.2012.2190935.

[8] O'Connell JF, Mumford CL. An exact dynamic programming based method to solve optimisation problems using GPUs. In: 2014 Second international symposium on computing and networking. 2014, p. 347–53. http://dx.doi.org/10.1109/CANDAR.2014.27.

[9] Grüne L, Semmler W. Using dynamic programming with adaptive grid scheme for optimal control problems in economics. J Econ Dyn Control 2004;28(12):2427–56. http://dx.doi.org/10.1016/j.jedc.2003.11.002.

[10] Osband I, Blundell C, Pritzel A, Van Roy B. Deep exploration via bootstrapped DQN. In: Advances in neural information processing systems 29. Barcelona, Spain; 2016, arXiv:1602.04621.

[11] Bertsekas DP. Dynamic programming and optimal control, Vol. I. 3rd ed.. Belmont, MA, USA: Athena Scientific; 2005.

[12] Garrard WL, Jordan JM. Design of nonlinear automatic flight control systems. Automatica 1977;13(5):497–505. http://dx.doi.org/10.1016/0005-1098(77)90070-X.

[13] Banks SP, Mhana KJ. Optimal control and stabilization for nonlinear systems. IMA J Math Control Inf 1992;9(2):179–96. http://dx.doi.org/10.1093/imamci/9.2.179.

[14] Kaya CY, Noakes JL. Computations and time-optimal controls. Optim Control Appl Methods 1996;17(3):171–85. http://dx.doi.org/10.1002/(SICI)1099-1514(199607/09)17:3<171::AID-OCA571>3.0.CO;2-9.

[15] Dubins LE. On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. Am J Math 1957;79(3):497. http://dx.doi.org/10.2307/2372560.

[16] Wolek A, Cliff EM, Woolsey CA. Time-optimal path planning for a kinematic car with variable speed. J Guid, Control, Dyn 2016;39(10):2374–90. http://dx.doi.org/10.2514/1.G001317.

[17] Lobo MS, Vandenberghe L, Boyd S, Lebret H. Applications of second-order cone programming. Linear Algebra Appl 1998;284(1–3):193–228. http://dx.doi.org/10.1016/S0024-3795(98)10032-0.

[18] Sundström O, Ambühl D, Guzzella L. On implementation of dynamic programming for optimal control problems with final state constraints. Oil Gas Sci Technol – Rev IFP 2010;65(1):91–102. http://dx.doi.org/10.2516/ogst/2009020.

[19] Hasdorff L. Gradient optimization and nonlinear control. A Wiley-Interscience Publication; 1976.

[20] Moore AW. Efficient memory-based learning for robot control (Ph.D. thesis), University of Cambridge; 1990.

[21] Sutton RS, Barto AG. Reinforcement learning: An introduction. 2nd ed.. The MIT Press; 2018.

YADPF: A reusable deterministic dynamic programming implementation in MATLAB

ORIGINALITY REPORT

14%	14%	0%	0%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	auralius.github.io Internet Source	10%
2	doaj.org Internet Source	4%

Exclude quotes	Off	Exclude matches	< 3%
Exclude bibliography	On		