

Numerical dynamic programming for discrete states

Prof. Paolo Brandimarte
Dip. di Scienze Matematiche – Politecnico di Torino
e-mail: `paolo.brandimarte@polito.it`
URL: `https://staff.polito.it/paolo.brandimarte`

This version: May 7, 2025

NOTE: For internal teaching use within the Masters' Program in Mathematical Engineering. Do not post or distribute.

References

These slides are taken from my book:

P. Brandimarte. *From Shortest Paths to Reinforcement Learning: A MATLAB-Based Introduction to Dynamic Programming*. Springer, 2021.

Other references:

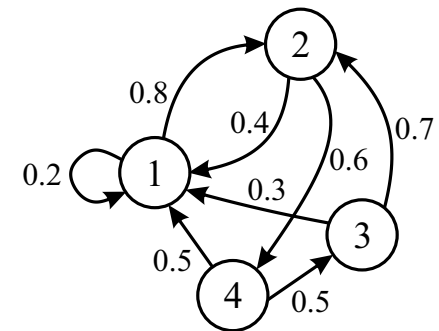
- D.P. Bertsekas. *Dynamic Programming and Optimal Control Vol. 2*, (4th ed.). Athena Scientific, 2012.
- R. Howard. *Dynamic Programming and Markov Processes*. MIT Press, 1960.
- W.B. Powell. *Approximate Dynamic Programming: Solving the Curses of Dimensionality* (2nd ed.). Wiley, 2011.
- M.L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, 2005.
- S. Ross. *Introduction to Stochastic Dynamic Programming*. Academic Press, 1983.

Discrete-time Markov chains

A discrete-time Markov chain is a stochastic process with a state variable s_t , $t = 0, 1, 2, 3, \dots$, taking values on a discrete set.

If the state space is countable, we may associate states with integer numbers and represent the process by a network. In the figure, the state space is the set $\mathcal{S} = \{1, 2, 3, 4\}$.

Transitions are represented by directed arcs, labeled by transition probabilities. For instance, if we are in state 3 now, at the next step we will be in state 2 with probability 0.7 or in state 1 with probability 0.3.



Transition probabilities are conditional probabilities and depend only on the current state:

$$\pi(i, j) \doteq \mathbb{P}\{s_{t+1} = j \mid s_t = i\},$$

Homogeneous (time invariant) chains are used in the case of infinite-horizon problems.

We may describe a discrete-time Markov chain by collecting the set of transition probabilities into the square single-step transition probability matrix $\mathbf{\Pi}$, where element π_{ij} gives the probability of a one-step transition from the row state i to the column state j .

For the Markov chain in the figure, we have

$$\mathbf{\Pi} = \begin{bmatrix} 0.2 & 0.8 & 0 & 0 \\ 0.4 & 0 & 0 & 0.6 \\ 0.3 & 0.7 & 0 & 0 \\ 0.5 & 0 & 0.5 & 0 \end{bmatrix}.$$

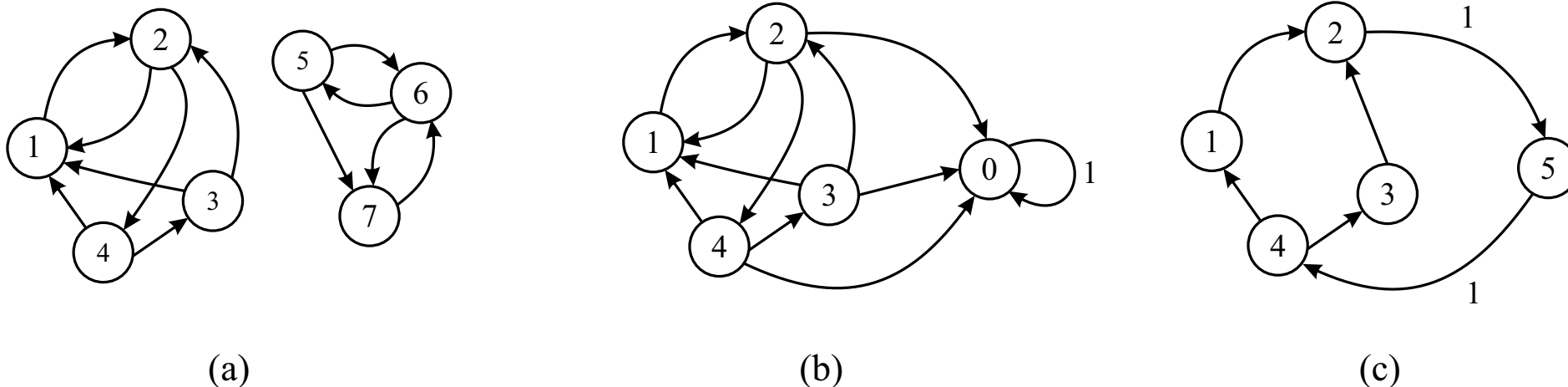
After a transition, we must land somewhere within the state space. Therefore, each and every row in matrix $\mathbf{\Pi}$ adds up to 1:

$$\sum_{j=1}^N \pi(i, j) = 1, \quad \forall i.$$

In MDPs, transitions are partially controlled by selecting actions. At state i , there is a finite set of feasible actions $\mathcal{A}(i)$, and for each action $a \in \mathcal{A}(i)$, we have a set of transition probabilities $\pi(i, a, j)$.

To evaluate the performance of a stationary control policy, we could use long-term (stationary) probabilities of being in each state, denoted by $q(i)$, $i \in \mathcal{S}$, collected into vector \mathbf{q} .

Such probabilities may not exist if the Markov chain is not “nice” enough.



In case (a) the chain is not irreducible, and the long-term probabilities depend on the initial state. We will assume a single and connected chain, a *unichain*, such that every state can be visited infinitely often in the long term, and every state can be reached from any other state in finite time with positive probability.

In case (b), state 0 is an *absorbing* state. Eventually, we will get to that state and stay there forever, and all the other states are *transient*. On the contrary, all states are *recurrent* in the first chain we have shown.

We may have both a set of recurrent and a set of transient states, in general. *Per se*, this may not be a stumbling block in finding a stationary distribution, but it may be a problem in making sure that certain learning algorithms work, as we should be able to visit every state often enough.

Finally, (c) is an example of a periodic chain. When there is a periodicity in the states, we cannot find a stationary distribution. As a rule, aperiodic chains are needed for certain nice theorems to hold.

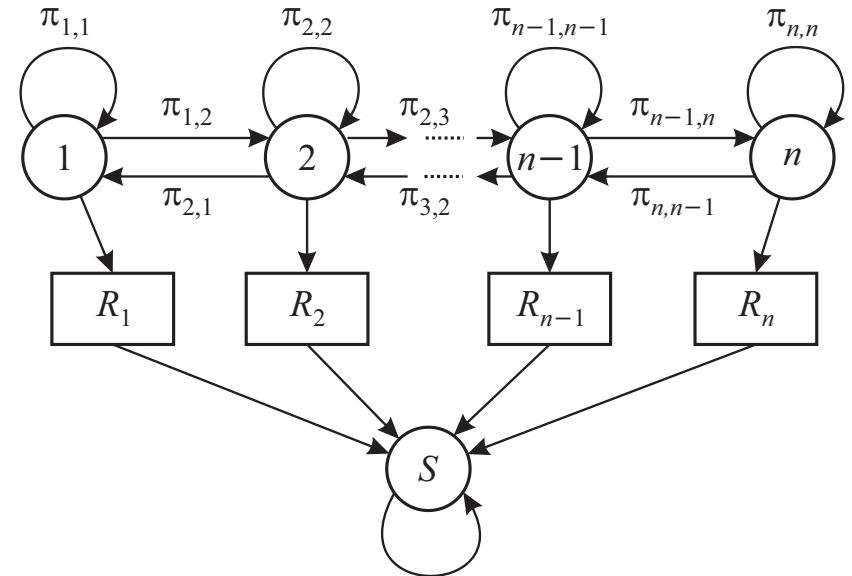
In the following, we will take for granted that the Markov process we are dealing with is well-behaved for any choice of the control policy.

For a finite-horizon MDP, the DP recursion is

$$V_t(i) = \underset{a \in \mathcal{A}(i)}{\text{opt}} \left\{ f(i, a) + \gamma \sum_j \pi(i, a, j) V_{t+1}(j) \right\}, \quad i \in \mathcal{S}. \quad (1)$$

Let us consider an optimal stopping example, comprising n states, $i = 1, \dots, n$, arranged in a linear chain representing a random walk.

For each interior state, $i \in \{2, \dots, n-1\}$, we stay there with probability $\pi_{i,i}$, we move up with probability $\pi_{i,i+1}$, and down with probability $\pi_{i,i-1}$. From boundary states 1 and n , we can only move up and down, respectively, or stay there.



At each time instant $t = 0, 1, \dots, T$ we have to choose whether to let the system behave randomly, without doing anything, or to stop the process. Let us denote these actions as wait and stop, respectively.

If we wait and do nothing, the state will change according to the random walk mechanism at no cost. The transition probabilities $\pi_{i,j}$ are just a shorthand for $\pi(i, \text{wait}, j)$.

We may stop the process and earn a state-dependent reward R_i , after which the process moves to a terminal state S , where the system will remain until the terminal time instant T .

We may interpret this mechanism as the problem of optimally selling an asset, whose price follows a random walk, which is not influenced by our action (the state is an informational one).

We sway a bit from the usual notation: we have to choose the action up to time instant T included, rather than the usual $T - 1$. Time instant T is the latest time instant at which we may sell the asset. This suggests that the optimal policy will not be stationary.

We need to find the value functions $V_t(i)$, $i = 1, \dots, n$, $t = 0, 1, 2, \dots, T$, subject to boundary conditions. Since, at time T , we should just sell the asset at the current price, we have

$$V_T(i) = R_i, \quad i = 1, \dots, n.$$

The DP equations may be written as follows:

$$\begin{aligned}
V_t(1) &= \max \left\{ R_1, \gamma(\pi_{1,1}V_{t+1}(1) + \pi_{1,2}V_{t+1}(2)) \right\} \\
V_t(i) &= \max \left\{ R_i, \gamma(\pi_{i,i-1}V_{t+1}(i-1) + \pi_{i,i}V_{t+1}(i) + \pi_{i,i+1}V_{t+1}(i+1)) \right\}, \\
&\quad i = 2, 3, \dots, n-1, \\
V_t(n) &= \max \left\{ R_n, \gamma(\pi_{n,n-1}V_{t+1}(n-1) + \pi_{n,n}V_{t+1}(n)) \right\}.
\end{aligned}$$

These equations may be represented in compact form by building a tridiagonal transition matrix,

$$\mathbf{\Pi} = \begin{bmatrix} \pi_{1,1} & \pi_{1,2} & & & & \\ \pi_{2,1} & \pi_{2,2} & \pi_{2,3} & & & \\ & \ddots & \ddots & \ddots & & \\ & & \pi_{n-1,n-2} & \pi_{n-1,n-1} & \pi_{n-1,n} & \\ & & & \pi_{n,n-1} & \pi_{n,n} & \end{bmatrix}.$$

Then, in vector form, we have

$$\mathbf{V}_t = \max \{ \mathbf{R}, \gamma \mathbf{\Pi} \mathbf{V}_{t+1} \},$$

where vectors \mathbf{V}_t and \mathbf{V}_{t+1} in \mathbb{R}^n collect the values of states $i = 1, \dots, n$, max should be interpreted componentwise, and vector \mathbf{R} collects the immediate rewards.

```

function [valueTable, decisionTable] = FindPolicyFiniteRW(transMatrix, ...
    payoffs, timeHorizon, discount)
payoffs = payoffs(:); % make sure inputs are columns
numStates = length(payoffs);
valueTable = zeros(numStates, timeHorizon+1);
decisionTable = zeros(numStates, timeHorizon+1);
% precompute invariant discounted probabilities
dprobs = discount*transMatrix;
% initialize
valueTable(:,timeHorizon+1) = payoffs;
decisionTable(:,timeHorizon+1) = ones(numStates,1);
for t = timeHorizon:-1:1
    valueWait = dprobs * valueTable(:, t+1);
    valueTable(:,t) = max(valueWait, payoffs);
    decisionTable(:,t) = valueWait <= payoffs;
end

```

The MATLAB function `FindPolicyFiniteRW` receives the tridiagonal transition matrix `transMatrix`, the reward vector `payoffs`, the time limit `timeHorizon`, and the discount factor `discount`.

As an output, we obtain the value functions stored in the matrix `valueTable`, as well as the binary decisions stored in the binary matrix `decisionTable`, where the value 0 corresponds to wait and 1 to stop.

The following script prepares a numerical case:

```
probUp = 0.1;
probDown = 0.1;
probStay = 1-probUp-probDown;
payoffs = [9; 10; 15; 20; 25; 40];
numStates = length(payoffs);
timeHorizon = 12;
discount = 0.99;
transMatrix = diag(probStay*ones(numStates,1)) + ...
    diag(probUp*ones(numStates-1,1),+1) + ...
    diag(probDown*ones(numStates-1,1),-1);
transMatrix(1,1) = transMatrix(1,1)+probDown;
transMatrix(end,end) = transMatrix(end,end)+probUp;
[valueTable, decisionTable] = FindPolicyFiniteRW(transMatrix, payoffs, ...
    timeHorizon, discount);
display(decisionTable);
```

Here, the asset price can move up or down, but the rewards are increasing along the chain.

We obtain the following decision table:

decisionTable =

0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	1	1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	1	1	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0	1
1	1	1	1	1	1	1	1	1	1	1	1	1

Here, we have six rows corresponding to states $i = 1, \dots, 6$ and 13 columns corresponding to time instants $t = 0, \dots, 12$, respectively. We observe, as expected, that the policy is not stationary.

In state $i = 1$, it is never optimal to stop the process early, as the price can only increase.

At time instant $t = 0$, when there is plenty of time to reach better states, we should only stop at the best state.

We notice that we should stop early at some states.

Apart from state 6, where the largest reward is earned, so that it is optimal to stop immediately, the earliest stop decision occurs at state 3. This may be understood by observing that, in this numerical example, the random walk is symmetric in terms of probabilities, but not in terms of payoffs.

In the case of an infinite-horizon MDP, the value function is implicitly given by equations like

$$V(i) = \operatorname{opt}_{a \in \mathcal{A}(i)} \left\{ f(i, a) + \gamma \sum_j \pi(i, a, j) V(j) \right\}, \quad i \in \mathcal{S}, \quad (2)$$

or

$$V(i) = \operatorname{opt}_{a \in \mathcal{A}(i)} \sum_{j \in \mathcal{S}} \pi(i, a, j) \{ h(i, a, j) + \gamma V(j) \}, \quad i \in \mathcal{S}. \quad (3)$$

Since we deal with finite MDPs, the state space may be identified with a finite set of integer numbers, $\mathcal{S} \equiv \{1, \dots, n\}$ and the value function $V : \mathcal{S} \rightarrow \mathbb{R}$ is a vector in \mathbb{R}^n , which we will denote by \mathbf{V} , with components $V(i)$, $i \in \mathcal{S}$.

There are two basic strategies for solving the above equations numerically and find the optimal policy:

- **Value iteration**, which relies on computationally cheap iterations, but gives the optimal value function only in the limit.
- **Policy iteration**, which requires more expensive iterations, but it will converge in finite time for a finite MDP, since there is a finite number of policies.

It is useful to define the following two operators:

- Given a generic value function represented by vector $\tilde{\mathbf{V}}$, not necessarily the optimal one, we define the operator \mathcal{T} :

$$[\mathcal{T}\tilde{\mathbf{V}}](i) = \operatorname{opt}_{a \in \mathcal{A}(i)} \sum_{j \in \mathcal{S}} \pi(i, a, j) \left\{ h(i, a, j) + \gamma \tilde{V}(j) \right\}, \quad i \in \mathcal{S}. \quad (4)$$

Note that $\mathcal{T}\tilde{\mathbf{V}}$ is itself a function over \mathcal{S} , in this case just another vector.

- Given a generic value function $\tilde{\mathbf{V}}$ and a generic stationary policy μ , not necessarily optimal, we define the operator \mathcal{T}_μ as follows:

$$[\mathcal{T}_\mu \tilde{\mathbf{V}}](i) = \sum_{j \in \mathcal{S}} \pi(i, \mu(i), j) \left\{ h(i, \mu(i), j) + \gamma \tilde{V}(j) \right\}, \quad i \in \mathcal{S}. \quad (5)$$

The operator \mathcal{T} plays a key role in value iteration, whereas the operator \mathcal{T}_μ does so for policy iteration. To understand why, let us observe (without any formal proof) that:

1. The optimal value vector \mathbf{V} is a fixed point of \mathcal{T} , i.e., it is the solution of the equation

$$\mathbf{V} = \mathcal{T}\mathbf{V}. \quad (6)$$

2. The value function \mathbf{V}_μ of a stationary policy μ is the fixed point of \mathcal{T}_μ , i.e.,

$$\mathbf{V}_\mu = \mathcal{T}_\mu \mathbf{V}_\mu. \quad (7)$$

Given a stationary policy, $V_\mu(i)$ gives the expected discounted performance obtained if we start from state i and apply that policy.

The existence of a value function and of an optimal stationary policy should not be taken for granted in general.

They apply to the case of a finite MDP, with strict discounting (i.e., when $\gamma < 1$) and bounded immediate contributions per stage (i.e., when there exists a constant M such that $|h(i, a, j)| \leq M$, a condition that is easily verified in the case of a finite MDP).

The key ingredient in the proofs is related to the fact that both \mathcal{T} and \mathcal{T}_μ are contraction operators, and thus they feature a unique fixed point.

For our purposes, it is sufficient to understand that the operator \mathcal{T} provides us with a characterization of the optimal stationary policy.

Furthermore, the operator \mathcal{T} , which involves a single optimization step, allows us to improve a nonoptimal stationary policy μ . To this aim, we need the value V_μ of policy μ , and the operator \mathcal{T}_μ allows us to find it.

According to Eq. (6), a finite MDPs with strict discounting ($\gamma < 1$) may be tackled by looking for a fixed point of the operator \mathcal{T} , defined by Eq. (4).

To understand why, let us consider a problem with a finite horizon and terminal value function given by vector $\mathbf{V}^{(0)}$. If we apply the operator \mathcal{T} to $\mathbf{V}^{(0)}$, we obtain a new value function

$$V^{(1)}(i) = [\mathcal{T} V^{(0)}](i) = \operatorname{opt}_{a \in \mathcal{A}(i)} \sum_{j \in \mathcal{S}} \pi(i, a, j) \left\{ h(i, a, j) + \gamma V^{(0)}(j) \right\}, \quad i \in \mathcal{S}.$$

Note that the superscript should be interpreted as the number of steps to go, and we may regard $\mathbf{V}^{(1)}$ as the value function for a single-stage problem and terminal value function $\mathbf{V}^{(0)}$.

This may be repeated iteratively, mapping $\mathbf{V}^{(k)}$ into $\mathbf{V}^{(k+1)}$:

$$V^{(k+1)}(i) = [\mathcal{T} V^{(k)}](i) = \operatorname{opt}_{a \in \mathcal{A}(i)} \sum_{j \in \mathcal{S}} \pi(i, a, j) \left\{ h(i, a, j) + \gamma V^{(k)}(j) \right\}, \quad i \in \mathcal{S}.$$

Again, we may consider this as the recursive equation for a finite-horizon MDP with terminal value function $\mathbf{V}^{(0)}$.

Intuition suggests that, with strict discounting, the contribution of the terminal value function $\mathbf{V}^{(0)}$ should be vanishing, as it is multiplied by a discount factor γ^k that goes to zero for increasing k .

If the sequence of value functions, where we interpret k as an iteration counter, converges to a limit \mathbf{V} , then we have found a fixed point of \mathcal{T} .

Generally speaking, given an operator $H(\cdot)$ mapping \mathbb{R}^n into itself, imagine that we want to solve the fixed-point equation $\mathbf{y} = H(\mathbf{y})$, where $\mathbf{y} \in \mathbb{R}^n$. A simple approach to find a fixed point is the iterative scheme

$$\mathbf{y}^{(k+1)} = H(\mathbf{y}^{(k)}), \quad k = 0, 1, 2, 3, \dots,$$

starting from an initial guess $\mathbf{y}^{(0)}$.

Clearly, there is no reason to believe that the above scheme will always converge to a fixed point. Actually, we may not even take for granted that a fixed point exists and that it is unique, in general.

Luckily, the aforementioned feature of \mathcal{T} , which is a contraction mapping, makes sure that a fixed point exists and is unique. This justifies the value iteration algorithm.

- 1: Choose an initial value function $V^{(0)}$. If we have no better clue, we may just set $V^{(0)}(i) = 0$ for all states $i \in \mathcal{S}$.
- 2: Choose a tolerance parameter ϵ .
- 3: Set the iteration counter $k = 0$ and the termination flag $\text{stop} = \text{false}$
- 4: **while** $\text{stop} \neq \text{true}$ **do**
- 5: For each state $i \in \mathcal{S}$, compute the next approximation of the value function:

$$V^{(k+1)}(i) = \underset{a \in \mathcal{A}(i)}{\text{opt}} \left\{ f(i, a) + \gamma \sum_{j \in \mathcal{S}} \pi(i, a, j) V^{(k)}(j) \right\}$$

- 6: **if** $\|V^{(k+1)} - V^{(k)}\|_{\infty} < \epsilon$ **then**
- 7: $\text{stop} = \text{true}$
- 8: **else**
- 9: $k = k + 1$
- 10: **end if**
- 11: **end while**
- 12: Let $\hat{V} = V^{(k+1)}$ be the estimate of the optimal value function.
- 13: Find the estimated optimal policy (choosing an arbitrary action if the set of optimal actions is not a singleton):

$$\hat{\mu}(i) \in \underset{a \in \mathcal{A}(i)}{\text{arg opt}} \left\{ f(i, a) + \gamma \sum_j \pi(i, a, j) \hat{V}(j) \right\}, \quad i \in \mathcal{S}.$$

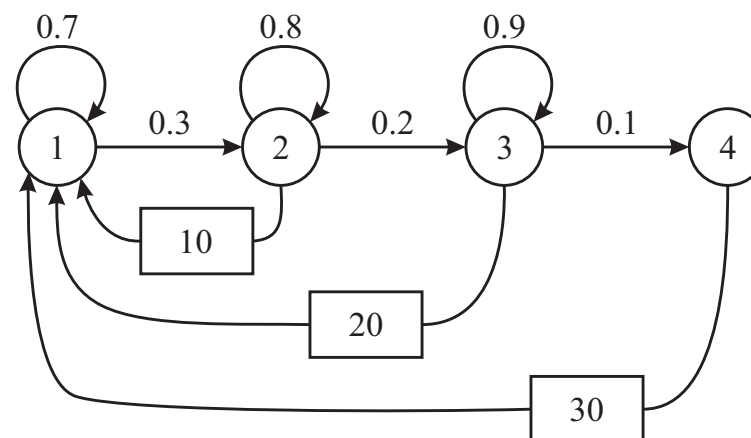
- 14: Return \hat{V} and $\hat{\mu}$.

A numerical example of value iteration

Consider a toy Markov chain modeling a recurring optimal stopping problem, with four states $k = 1, 2, 3, 4$.

We have one-way transition probabilities, and the possibility of resetting the state earning a state-dependent payoff.

Note that, when we move along the chain, the probability of moving to a better state is diminishing. Hence, the tradeoff is unclear and is also affected by the discount factor γ .



There are two possible implementation approaches to solve a MDP by value iteration:

1. Write a generic function implementing value iteration for an MDP with an arbitrary transition probability matrix $\mathbf{\Pi}$ for each feasible action.
2. Write a function for the specific problem, taking advantage of its peculiar structure.

In this case, let us adopt the ad hoc strategy.

We have two feasible actions, wait and reset, and we actually need only the transition matrix $\mathbf{\Pi}$ for the former action, since the state transition is deterministic with the latter one. Moreover, since most elements of $\mathbf{\Pi}$ are zero, we just need to specify the vector of “no transition” probabilities $\pi_{kk} \equiv \pi(k, \text{wait}, k)$, $k \in \mathcal{S}$, and the payoffs.

```
function [stateValues, policy, count] = PW311_FindPolicy(probs, payoffs, ...
    discount, myeps, maxIter)
% make sure inputs are columns
probs = probs(:);
payoffs = payoffs(:);
% set default values for tolerance and max iterations
if nargin < 4, myeps=0.00001; end
if nargin < 5, maxIter=1000; end
numStates = length(payoffs);
oldV = zeros(numStates,1);
newV = zeros(numStates,1);
stopit = false;
count = 0; % iteration counter
% precompute invariant discounted probs
dprobStay = discount*probs;
dprobMove = discount*(1-probs(:));
% we need to make a decision only in interior states
idxInteriorStates = 2:(numStates-1);
```

```

% value iteration
while (~stopit)
    count = count + 1;
    % update: first and last state are apart
    newV(1) = dprobStay(1)*oldV(1)+dprobMove(1)*oldV(2);
    newV(numStates) = payoffs(numStates) + discount*oldV(1);
    % now update interior states
    valueReset = payoffs(idxInteriorStates) + discount*oldV(1);
    valueWait = dprobStay(idxInteriorStates).*oldV(idxInteriorStates) + ...
        dprobMove(idxInteriorStates).*oldV(idxInteriorStates+1);
    newV(idxInteriorStates) = max(valueWait, valueReset);
    if ( (norm(newV-oldV,Inf) < myeps) || (count > maxIter) )
        stopit = true;
    else
        oldV = newV;
    end
end
stateValues = newV;
policy = [0; valueReset > valueWait; 1];

```

The inputs are the vector `probs` of probabilities of remaining in each state, the vector `payoffs` of immediate rewards and the discount factor `discount`. We also need to set the tolerance ϵ and the maximum number of iterations.

The function returns the vectors `stateValues` of state values and `policy` containing the optimal policy, where 0 and 1 correspond to wait and reset, respectively.

We may also wish to keep track of the number of iterations, stored into variable `count`, to check the convergence behavior.

The following snapshot finds the optimal solution for the example with discount factor $\gamma = 0.8$.

```
>> probs = [0.7, 0.8, 0.9, 0];
>> payoff = [0, 10, 20, 30];
>> discount = 0.8;
>> [stateValues, policy, count] = PW311_FindPolicy(probs, payoff, discount);
>> policy'
ans =
     0     1     1     1
>> stateValues'
ans =
    9.6774    17.7419    27.7419    37.7419
>> count
count =
    57
```

In this very simple case, convergence is rather fast, and the values of states 2, 3, and 4 is that they differ by 10, which is exactly the increment in the payoff if we start from a better state and then apply the optimal policy.

With the above data, the best policy is greedy, in the sense that we should always go for the immediate payoff, without waiting for better opportunities.

This depends on two factors: the relatively low probability of moving to a better state and the heavy discount factor.

Let us play with numbers to check our intuition.

- If we set `probs = [0.6, 0.6, 0.6, 0]` and `discount = 0.95`, we obtain

```
>> stateValues'  
ans =  
    60.5195    68.4826    77.4935    87.4935  
>> policy'  
ans =  
     0     0     1     1  
>> count =  
    248
```

- If we increase the discount factor to `discount = 0.99`, we obtain

```
>> stateValues'  
ans =  
    342.1122    350.7518    359.6096    368.6910  
>> policy'  
ans =  
     0     0     0     1  
>> count =  
    1001
```

Apart from finding the optimal policy, it is also quite instructive to check its application by Monte Carlo simulation and to compare the estimated state values with the exact ones.

A Monte Carlo simulation is also useful to check the robustness of the policy against model misspecifications.

```
function [value, confInt] = PW311_Simulate(probs, payoff, discount, ...
    startState, numRepl, numSteps, policy)
numStates = length(probs);
sumDiscRewards = zeros(numRepl, 1);
discFactors = discount.^(0:(numSteps-1));
for k = 1:numRepl
    state = startState;
    Rewards = zeros(numSteps, 1);
    for t = 1:numSteps
        if (state == 1) % state 1, no choice
            if rand > probs(state)
                state = 2;
            end
        elseif state == numStates
            % last state, collect reward and back to square 1
            state = 1;
            Rewards(t) = payoff(numStates);
        end
    end
    sumDiscRewards(k) = sum(Rewards .* discFactors);
end
value = sumDiscRewards / numRepl;
confInt = [value - 1.96 * sqrt(sumDiscRewards^2 / (numRepl * numSteps)), ...
    value + 1.96 * sqrt(sumDiscRewards^2 / (numRepl * numSteps))];
```

```

    else % at interior states, apply policy
        if policy(state) == 1
            % collect reward and go back to square 1
            Rewards(t) = payoff(state);
            state = 1;
        else % random transition
            if rand > probs(state)
                state = state + 1;
            end
        end
    end
end % end if on states
end % each replication
sumDiscRewards(k) = dot(Rewards,discFactors);
end % all replications
[value,~,confInt] = normfit(sumDiscRewards);

```

Due to discounting, there is little point in simulating a large number of steps. For instance, $0.8^{40} = 0.0001329228$, which suggests that with heavy discounting the effective time horizon is small.

```
probs = [0.7, 0.8, 0.9, 0];
payoff = [0, 10, 20, 30];
discount = 0.8;
[stateValues, policy] = PW311_FindPolicy(probs, payoff, discount);
rng('default') % reset state of random number generator for repeatability
startState = 1;
numRepl = 1000;
numSteps = 40;
[value, confInt] = PW311_Simulate(probs, payoff, discount, ...
    startState, numRepl, numSteps, policy);
```

The estimated value of state 1 is in good agreement with the result from value iteration:

```
>> stateValues(1)
ans =
    9.6774
>> value
value =
    9.6023
>> confInt
confInt =
    9.3233
    9.8814
```

Given the limited number of replications and the short effective horizon, the confidence interval does not look too tight.

Policy iteration

With value iteration it may happen that we have already found the optimal policy, based on approximate state values, but we are still far from assessing its true value.

On the contrary, the policy iteration approach takes a radical step towards the assessment of the exact value of a policy.

Let us consider again the mapping \mathcal{T}_μ defined in Eq. (5):

$$[\mathcal{T}_\mu \tilde{V}](i) = f(i, \mu(i)) + \gamma \sum_{j \in \mathcal{S}} \pi(i, \mu(i), j) \tilde{V}(j), \quad i \in \mathcal{S}.$$

Unlike mapping \mathcal{T} , \mathcal{T}_μ is associated with a specific stationary policy μ and it does not involve any optimization.

The value function V_μ of a stationary policy μ is a fixed point of \mathcal{T}_μ , and it can be found by solving a system of linear equations:

$$V_\mu(i) = f(i, \mu(i)) + \gamma \sum_{j \in \mathcal{S}} \pi(i, \mu(i), j) V_\mu(j), \quad i \in \mathcal{S}.$$

This set of equations has a unique solution, which we may find by collecting the transition probabilities induced by policy μ into the matrix

$$\mathbf{\Pi}_\mu \doteq \begin{bmatrix} \pi(1, \mu(1), 1) & \pi(1, \mu(1), 2) & \cdots & \pi(1, \mu(1), n) \\ \pi(2, \mu(2), 1) & \pi(2, \mu(2), 2) & \cdots & \pi(2, \mu(2), n) \\ \vdots & \vdots & \ddots & \vdots \\ \pi(n, \mu(n), 1) & \pi(n, \mu(n), 2) & \cdots & \pi(n, \mu(n), n) \end{bmatrix}, \quad (8)$$

and the immediate contributions into the vector

$$\mathbf{f}_\mu \doteq \begin{bmatrix} f(1, \mu(1)) \\ f(2, \mu(2)) \\ \vdots \\ f(n, \mu(n)) \end{bmatrix}. \quad (9)$$

To find the fixed point of \mathcal{T}_μ , taking for granted that it exists and is unique, we have to solve a system of linear equations,

$$\mathbf{V}_\mu = \mathcal{T}_\mu \mathbf{V}_\mu = \mathbf{f}_\mu + \gamma \mathbf{\Pi}_\mu \mathbf{V}_\mu,$$

which may be rewritten as

$$(\mathbf{I} - \gamma \mathbf{\Pi}_\mu) \mathbf{V}_\mu = \mathbf{f}_\mu,$$

where $\mathbf{I} \in \mathbb{R}^{n \times n}$ is the identity matrix (n is the number of states in \mathcal{S}). Formally, this is solved by matrix inversion:

$$\mathbf{V}_\mu = (\mathbf{I} - \gamma \mathbf{\Pi}_\mu)^{-1} \mathbf{f}_\mu.$$

Direct methods like Gaussian elimination may be prohibitive for large matrices. Furthermore, the involved matrix is likely to be sparse, a property which is destroyed in Gaussian elimination. Hence, iterative methods are usually adopted.

Policy μ need not be optimal, but a surprising fact is that it is easy to find a way to improve μ , if it is not optimal.

Given a stationary policy μ and its value function V_μ , let us define an alternative policy $\tilde{\mu}$ as follows:

$$\tilde{\mu}(i) \in \arg \operatorname{opt}_{a \in \mathcal{A}(i)} \left\{ f(i, a) + \gamma \sum_j \pi(i, a, j) V_\mu(j) \right\}, \quad i \in \mathcal{S}.$$

This kind of operation is called policy **improvement**, and it can be shown that the new policy cannot be worse than μ , i.e.,

$$V_\mu(i) \leq V_{\tilde{\mu}}(i), \quad i \in \mathcal{S},$$

for a maximization problem (reverse the inequality for a minimization problem).

If μ is not optimal, strict inequality applies to at least one state.

Since, for a finite MDP, there is a finite number of stationary and deterministic policies, it must be the case that by a sequence of policy improvements we end up with an optimal policy. This is precisely the idea behind the algorithm of policy iteration.

- 1: Define an arbitrary initial stationary policy $\mu^{(0)}$.
- 2: Set the iteration counter $k = 0$ and the termination flag $\text{stop} = \text{false}$
- 3: **while** $\text{stop} \neq \text{true}$ **do**
- 4: Evaluate policy $\mu^{(k)}$ by solving

$$\left(\mathbf{I} - \gamma \mathbf{\Pi}_{\mu^{(k)}}\right) \mathbf{V}_{\mu^{(k)}} = \mathbf{f}_{\mu^{(k)}}.$$

- 5: Find a new stationary policy $\mu^{(k+1)}$ by policy improvement:

$$\mu^{(k+1)}(i) \in \arg \underset{a \in \mathcal{A}(i)}{\text{opt}} \left\{ f(i, a) + \gamma \sum_j \pi(i, a, j) V_{\mu^{(k)}}(j) \right\}, \quad i \in \mathcal{S}.$$

- 6: **if** the new policy is the same as the incumbent one **then**
- 7: $\text{stop} = \text{true}$
- 8: **else**
- 9: $k = k + 1$
- 10: **end if**
- 11: **end while**
- 12: Return the optimal value function and optimal stationary policy

```

function [stateValues, policy] = FindPolicyPI(payoffArray, transArray, ...
    feasibleActions, discount, initialPolicy, optDir, verboseFlag)
numStates = length(initialPolicy);
oldPolicy = initialPolicy(:); % make sure it is a column
newPolicy = zeros(numStates,1);
stopit = false;
count = 1;
while (~stopit)
    % build vector and matrix for the current policy
    [rhs,matrix] = makeArrays(oldPolicy);
    % evaluate current policy (we use Gaussian elimination here)
    oldV = matrix\rhs;
    % improve policy
    for j = 1:numStates
        if strcmpi(optDir, 'min')
            newPolicy(j) = findMin(j);
        else
            newPolicy(j) = findMax(j);
        end
    end % for states
    if verboseFlag
        printPolicy(oldPolicy);
    end
end

```

```

    if all(oldPolicy == newPolicy)
        stopit = true;
    else
        count = count + 1;
        oldPolicy = newPolicy;
    end
end
stateValues = oldV;
policy = oldPolicy;

% nested functions
% build system of linear equations
function [outVet, outMatrix] = makeArrays(policy)
    outVet = zeros(numStates,1);
    auxMatrix = zeros(numStates,numStates);
    for k=1:numStates
        outVet(k) = payoffArray(k,policy(k));
        auxMatrix(k,:) = transArray(k,:,policy(k));
    end % for
    outMatrix = eye(numStates) - discount*auxMatrix;
end

```

```

% print policy
function printPolicy(vet)
    fprintf(1,'Policy at step %d:  ',count);
    aux = [sprintf('%d,', vet(1:(end-1))), sprintf('%d', vet(end))];
    fprintf(1,'%s\n',aux);
end
% find min cost action
function best = findMin(k)
    best = NaN;
    valBest = inf;
    act = feasibleActions{k};
    for a = 1:length(act)
        aux = contribArray(k,act(a)) + ...
                discount * dot(transArray(k,:,act(a)), oldV);
        if aux < valBest
            valBest = aux;
            best = act(a);
        end
    end
end
% find max reward action
function best = findMax(k)
    ...
end

```

This is a generic, but very naive implementation. The main function `FindPolicyPI` produces vectors `stateValues` and `policy`, collecting the values and the optimal action for each state, respectively. Actions are numbered starting from 1, to allow for MATLAB indexing.

The function requires the following inputs:

- `payoffArray` is a matrix collecting immediate payoffs, where rows correspond to states and columns to actions.
- `transArray` is a three-dimensional array collecting transition probabilities; the last index corresponds to actions, and for each action we have a square transition probability.
- `feasibleActions` is a cell array, collecting the array of feasible actions for each state; we need a cell array, since the number of feasible actions need not be the same across states.
- `initialPolicy` is a vector giving, for each state, the selected action for the initial policy $\mu^{(0)}$.
- `discount` is the discount factor.
- `optDir` is a character string, which may be either `min` or `max`.

- `verboseFlag` is a Boolean flag; when set to `true`, the current policy is displayed at each step.

We use nested functions:

- `makeArrays` builds the vector \mathbf{f}_μ and the matrix $\mathbf{\Pi}_\mu$ that we need for policy iteration.
- `printPolicy` prints the iteration number and the current policy.
- `findMin` and `findMax` find the index of an optimal action, based on the current value of V_μ .

A numerical example of policy iteration

To illustrate policy iteration, let us consider again the simple MDP for recurring optimal stopping.

```
probs = [0.7; 0.8; 0.9; 0];
payoff = [0; 10; 20; 30];
discount = 0.8;
initialPolicy = [1;1;1;2];
payoffArray = [zeros(4,1), payoff];
% build transition matrices
transReset = [ones(4,1), zeros(4, 3)];
auxProbs = probs(1:(end-1)); % the last one is not relevant
transWait = diag([auxProbs;1]) + diag(1-auxProbs,1);
transArray(:,:,1) = transWait;
transArray(:,:,2) = transReset;
feasibleActions = {1, [1;2], [1:2], 2};
[stateValues, policy] = FindPolicyPI(payoffArray, transArray, ...
    feasibleActions, discount, initialPolicy, 'max', true);
```

Due to MATLAB indexing, action 1 corresponds to wait and action 2 corresponds to reset. The payoff array `payoffArray` is

$$\begin{bmatrix} 0 & 0 \\ 0 & 10 \\ 0 & 20 \\ 0 & 30 \end{bmatrix}.$$

Actually, the zeros in positions (1,2) and (4,1) are not relevant, as we cannot reset in state 1 and we cannot wait in state 4 (or there is no point in doing so, if you prefer).

The two transition matrices, collected into the three-dimensional array `transArray` are

$$\mathbf{\Pi}(1) = \begin{bmatrix} 0.7 & 0.3 & 0 & 0 \\ 0 & 0.8 & 0.2 & 0 \\ 0 & 0 & 0.9 & 0.1 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{\Pi}(2) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix},$$

where we observe that the last row of $\mathbf{\Pi}(1)$ and the first row of $\mathbf{\Pi}(2)$ are not relevant.

To illustrate the notation, for the initial policy $\mu^{(0)} = (1, 1, 1, 2)$ that always prescribes wait when feasible, we will have

$$\mathbf{f}_{\mu^{(0)}} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 30 \end{bmatrix}, \quad \mathbf{\Pi}_{\mu^{(0)}} = \begin{bmatrix} 0.7 & 0.3 & 0 & 0 \\ 0 & 0.8 & 0.2 & 0 \\ 0 & 0 & 0.9 & 0.1 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

Since we set the verbose flag on, we obtain the MATLAB output

Policy at step 1: 1,1,1,2

Policy at step 2: 1,2,2,2

showing that we are done after a single improvement step.

At first sight, value and policy iteration look very different:

- Value iteration relies on a possibly large number of computationally cheap iterations; policy iteration, on the contrary, relies on a (hopefully) small number of possibly expensive iterations.
- Convergence may only be obtained in the limit for value iteration, whereas finite convergence is obtained for policy iteration.

However, it is possible to bridge the gap between the two.

Imagine that we evaluate a stationary policy μ by fixed-point iteration of operator \mathcal{T}_μ :

$$V_\mu^{(k+1)}(i) = f(i, \mu(i)) + \gamma \sum_{j \in \mathcal{S}} \pi(i, \mu(i), j) V_\mu^{(k)}(j), \quad i \in \mathcal{S}. \quad (10)$$

In practice, this may be needed because the application of Gaussian elimination to solve a large-scale system of linear equations is not feasible.

Before applying the improvement step, we should wait for convergence of the above scheme. Now, imagine that we stop prematurely the iterations in Eq. (10), *optimistically* assuming that we have correctly evaluated the current policy μ .

After a sufficient number of iterations, we stop and use the current values as an estimate. Hence, we set $\widehat{V}_\mu(i) = V_\mu^{(k+1)}(i)$, $i \in \mathcal{S}$ and apply the improvement step:

$$\tilde{\mu}(i) \in \arg \operatorname{opt}_{a \in \mathcal{A}(i)} \left\{ f(i, a) + \gamma \sum_j \pi(i, a, j) \widehat{V}_\mu(j) \right\}, \quad i \in \mathcal{S}.$$

This kind of approach is called **optimistic policy iteration** and leads to methods broadly labeled under the umbrella of **generalized policy iteration**.

Depending on how we are optimistic about our ability to assess the value of the policy, we may iterate Eq. (10) for a large or small number of steps.

In value iteration we always keep changing the policy, which is implicit in the updated value function:

$$V^{(k+1)}(i) = \operatorname{opt}_{a \in \mathcal{A}(i)} \sum_{j \in \mathcal{S}} \pi(i, a, j) \left\{ h(i, a, j) + \gamma V^{(k)}(j) \right\}, \quad i \in \mathcal{S}.$$

In a sense, we are so optimistic about our assessment of the implied policy, that we apply \mathcal{T} repeatedly without even a single application of \mathcal{T}_μ . On the contrary, policy iteration would apply a possibly very large number of iterations of \mathcal{T}_μ before switching to another policy.

Therefore, value and policy iteration can be considered as the extreme points of a continuum of related approaches.

This difference becomes especially relevant in the context of reinforcement learning (RL).

When the transition probabilities (and possibly the immediate contributions) are not known, we need to switch to model-free DP. In that context, the value function is typically replaced by Q -factors $Q(s, a)$ depending on both states and actions.

We will see that Q -learning is the RL counterpart of value iteration and that it is an **off-policy** learning scheme. By this we mean that we apply a policy to learn another one.

On the contrary, the RL counterparts of policy iteration rely on an **on-policy** learning scheme, as we aim at learning the value of the very policy that we are applying. One well-known learning approach is SARSA.

In RL, we cannot afford to learn the value function of a policy exactly, especially if this requires costly Monte Carlo runs or online experiments, and we will have to perform an improving step sooner or later.

The precise way in which this is implemented leaves room to quite a few variants on the theme, which leads to a bewildering (and somewhat confusing) array of RL strategies.