REINFORCEMENT LEARNING

Part 2: Markov Decision Processes, Policies and Planning

@SSRoy

How to use this tutorial

This is Part 2 of the RL series. Part 1 explained the RL framework: agent, environment, state, action, reward, return G_t, discount γ, policy π and value functions v_π and q_π. Here we answer three questions:

1. **MDP:** how do we write down an RL problem exactly, as a mathematical object?
2. **Policy and optimality:** what does the *best* policy mean, and what equations does it satisfy?
3. **Planning:** if we know how the environment works, how do we compute the best policy? (policy evaluation, policy iteration, value iteration, linear programming)

We use **one small example throughout**, the corridor from Part 1, so every number can be checked by hand. Each phase ends with a short exercise and a full solution. Try the exercise first.

Phase	Topic	Key question
1	Quick recap of Part 1	What do we already know?
2	The Markov Decision Process (MDP)	What are the five ingredients of an MDP?
3	Policies in an MDP	How do we write a policy as a table?
4	Bellman expectation equation	How are the values of neighbouring states linked?
5	Optimal values and optimal policy	What is the best policy and how do we recognise it?
6	What is planning?	What do planning algorithms do?
7	Iterative policy evaluation	How do we compute v_π for a given policy?
8	Policy improvement	How do we make a policy better?
9	Policy iteration	Evaluate, improve, repeat
10	Value iteration	A faster shortcut to the optimal values
11	Linear programming (brief)	Planning as an optimisation problem
12	Comparison and summary	Which method, when?

1. Quick Recap of Part 1

Symbol	Meaning
S_t, A_t, R_{t+1}	State, action at time t, and the reward that follows
$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$	Return: discounted sum of future rewards
$G_t = R_{t+1} + \gamma G_{t+1}$	Recursive form of the return
γ (gamma), $0 \le \gamma \le 1$	Discount factor
$\pi(a \mid s)$	Policy: probability of choosing action a in state s
$v_\pi(s)$	Expected return starting in s and following π
$q_\pi(s, a)$	Expected return taking a in s first, then following π

The running example (corridor). Four cells A, B, C, D in a row. D is the goal and is **terminal** (the episode ends there). In A, B and C the agent can move **Left** or **Right**. Moving Left in A hits the wall, so the agent stays in A. Every move costs −1, except a move that enters D, which earns +10. We use **γ = 0.9** unless stated otherwise.

Exercise 1.1 — Warm-up

(a) What is the value of a terminal state? (b) In Part 1 we found v(A) = 6.2 for the policy "always Right". Is this a reward or a value?

Solution

(a) **0**, because no rewards follow a terminal state.

(b) A **value**: it is the long-run expected return from A (−1 − 0.9 + 8.1 = 6.2), not a single immediate reward.

2. The Markov Decision Process (MDP)

A **Markov Decision Process** is the standard mathematical description of an RL problem. It lists *everything* about the environment: which situations exist, what the agent can do, what happens after each action, and what reward is given. It assumes the **Markov property** from Part 1: the next state and reward depend only on the current state and action, not on the earlier history.

2.1 The five ingredients

An MDP is written as a 5-tuple (an ordered list of five things):

$$\text{MDP} = \langle \mathcal{S},\ \mathcal{A},\ P,\ R,\ \gamma \rangle \qquad (1)$$

Symbol	Name	Meaning	Corridor example

Symbol	Name	Meaning	Corridor example
$\mathcal{S}$	State set	All possible states	{A, B, C, D}
A	Action set	All possible actions	{Left, Right}
P	Transition probability	Probability of moving to each next state	Right from B goes to C with probability 1
R	Reward function	Reward for each transition	−1 per move, +10 on entering D
γ	Discount factor	Weight of future rewards	0.9

2.2 Transition probability P

$$P(s' \mid s, a) = \Pr\{\, S_{t+1} = s' \mid S_t = s,\ A_t = a \,\} \qquad (2)$$

- **P(s' | s, a)**: the probability that the next state is s', given that the agent is in state s and takes action a.
- **s**: current state; **a**: action taken; **s'** ("s-prime"): a possible next state.
- **Pr{ … }**: "the probability that …"; the bar "|" is read "given".

For each state–action pair, the probabilities over all next states form a probability distribution, so:

$$P(s' \mid s, a) \geq 0 \quad \text{and} \quad \sum_{s' \in \mathcal{S}} P(s' \mid s, a) = 1 \qquad (3)$$

In words: no probability is negative, and **each row of the transition table adds up to 1**, because the agent must end up *somewhere*.

2.3 Reward function R

R(s, a, s') is the reward received when action **a** in state **s** leads to state **s'.**

*** (Some books, including Mohri et al., write the reward as r(s, a), the expected reward for taking a in s. Part 1 used the joint form p(s', r | s, a). When the reward is fixed for each transition, all three describe the same thing.)*

2.4 The corridor as an MDP

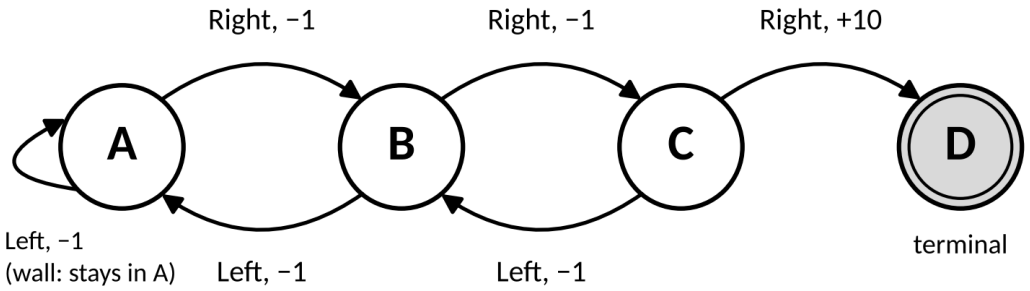


Figure 1. State-transition diagram of the corridor MDP. Each arrow is labelled with the action and the reward. Left in A hits the wall and loops back to A. D (double circle) is terminal.

Transition table P(s′ | s, a) (each row is one state–action pair; each row sums to 1):

(s, a)	s′ = A	s′ = B	s′ = C	s′ = D	Row sum
(A, Left)	1	0	0	0	1
(A, Right)	0	1	0	0	1
(B, Left)	1	0	0	0	1
(B, Right)	0	0	1	0	1
(C, Left)	0	1	0	0	1
(C, Right)	0	0	0	1	1

Reward table R(s, a, s′) (only transitions that can happen are listed):

s	a	s′	R(s, a, s′)
A	Left	A	−1
A	Right	B	−1
B	Left	A	−1
B	Right	C	−1
C	Left	B	−1
C	Right	D	+10

Deterministic and stochastic MDPs

In the corridor every row has a single 1, so the outcome of each action is certain: the MDP is **deterministic**. In a **stochastic** MDP an action can lead to different next states with different probabilities. Figure 2 shows a slippery floor: in B, Right reaches C with probability 0.8 and slips (stays in B) with probability 0.2.

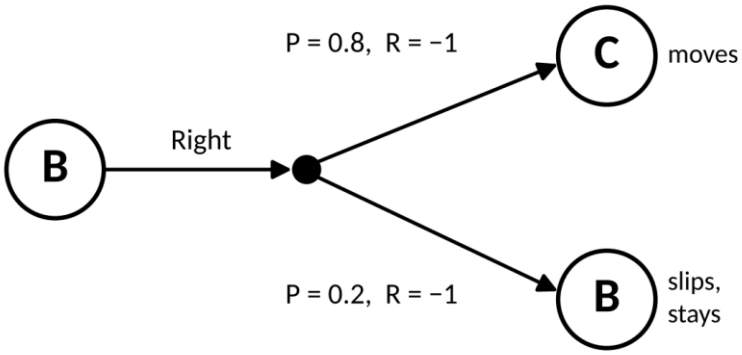


Figure 2. A stochastic transition. From B, action Right leads to C with probability 0.8 and back to B with probability 0.2; both give reward −1.

Exercise 2.1 — Write and check a transition row

(a) Write the table row for (B, Right) in the slippery corridor of Figure 2. (b) Is it a valid row? (c) A student writes the row for (C, Right) as: A 0, B 0.1, C 0, D 0.8. What is wrong?

Solution

(a) (B, Right): A = 0, B = 0.2, C = 0.8, D = 0.

(b) **Valid**: all entries are ≥ 0 and 0 + 0.2 + 0.8 + 0 = 1.

(c) The row sums to 0.9, not 1. Probability 0.1 is missing; the agent must end up somewhere. A correct version could be B = 0.2, D = 0.8.

3. Policies in an MDP

A policy tells the agent what to do in every non-terminal state. In a small MDP we can write it as a table.

State	π_R (always Right)	π_{rand} (random)
A	$\pi(\text{Right} \mid A) = 1$	$\pi(\text{Left} \mid A) = 0.5,\ \pi(\text{Right} \mid A) = 0.5$
B	$\pi(\text{Right} \mid B) = 1$	$\pi(\text{Left} \mid B) = 0.5,\ \pi(\text{Right} \mid B) = 0.5$
C	$\pi(\text{Right} \mid C) = 1$	$\pi(\text{Left} \mid C) = 0.5,\ \pi(\text{Right} \mid C) = 0.5$

π_R is **deterministic** (one action per state, probability 1). π_{rand} is **stochastic**. Both are *stationary*: the choice depends only on the state, not on the time step. For a finite MDP, searching among stationary policies is enough to find the best one.

3.1 How many deterministic policies are there?

A deterministic policy picks one of the |A| actions in each of the |S| states, independently. So:

$$\text{number of deterministic policies} = |\mathcal{A}|^{|\mathcal{S}|} \tag{4}$$

Here |A| is the number of actions and |S| the number of (non-terminal) states where a choice is made. For the corridor: 2^3 = **8** policies. This number grows very quickly, which is why we need smart algorithms instead of trying every policy.

Exercise 3.1 — Counting policies

(a) List all 8 deterministic policies of the corridor as (action in A, action in B, action in C). (b) A grid world has 20 non-terminal states and 4 actions. How many deterministic policies does it have?

Solution

(a) (L,L,L), (L,L,R), (L,R,L), (L,R,R), (R,L,L), (R,L,R), (R,R,L), (R,R,R).

(b) 4^{20} = 1,099,511,627,776 (about 1.1×10^{12}). Checking each one is hopeless; this is the motivation for planning algorithms.

4. The Bellman Expectation Equation

In Part 1 we found $G_t = R_{t+1} + \gamma G_{t+1}$: *return now = next reward + γ × return from the next step*. Taking averages of both sides gives the most important equation of this tutorial, named after Richard Bellman. It links the value of a state to the values of the states that follow it.

4.1 Value of an action, $q_\pi(s, a)$

If we take action a in state s, we land in some s′ with probability P(s′ | s, a), get reward R(s, a, s′), and from s′ onwards we expect $v_\pi(s')$. Averaging over the possible next states:

$$q_\pi(s,a) = \sum_{s' \in \mathcal{S}} P(s' \mid s,a) \left[R(s,a,s') + \gamma\, v_\pi(s') \right] \tag{5}$$

- $\sum_{s' \in \mathbb{S}}$: add up over every possible next state s′.
- **P(s′ | s, a)**: how likely that next state is.
- **R(s, a, s′) + γv_π(s′)**: immediate reward plus discounted value of where we land.

4.2 Value of a state, $v_\pi(s)$

The policy chooses action a with probability π(a | s), so the value of the state is the policy-weighted average of the action values (as in Part 1):

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s)\, q_\pi(s, a) \tag{6}$$

Putting Equation (5) into Equation (6) gives the **Bellman expectation equation for v_π**:

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s) \sum_{s' \in \mathcal{S}} P(s' \mid s, a) \left[R(s, a, s') + \gamma\, v_\pi(s') \right] \tag{7}$$

Read it as two averages: first average over the actions the policy might choose (weights π(a | s)), then over the next states the environment might produce (weights P(s′ | s, a)). Inside: *reward now + γ × value next*. Figure 3 shows this as a tree called a **backup diagram**: the value at the top is "backed up" from the values at the bottom.

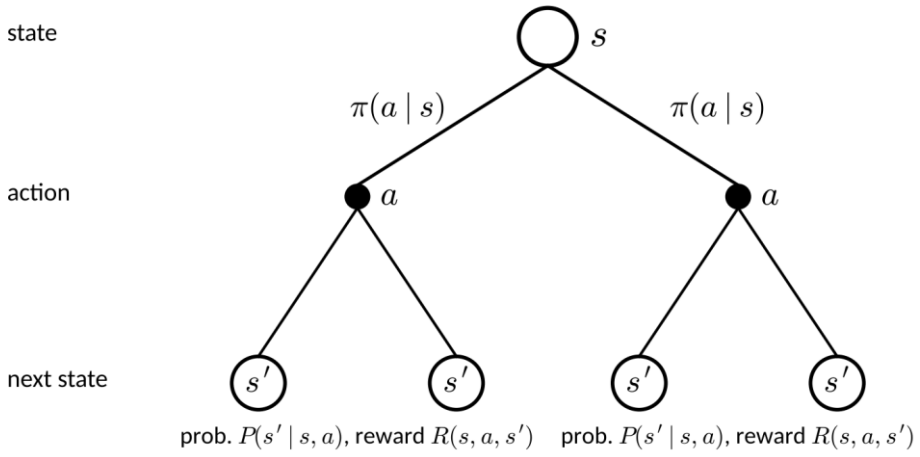


Figure 3. Backup diagram for $v_\pi(s)$. Open circles are states, solid dots are actions. The policy picks the branch to an action; the environment picks the branch to a next state.

For a **deterministic policy in a deterministic MDP** both sums have a single term, and the equation becomes very simple:

$$v_\pi(s) = R(s, a, s') + \gamma\, v_\pi(s') \qquad \text{where } a = \pi(s) \tag{8}$$

4.3 Solved example 1: always Right (π_R)

State	Bellman equation	Value
D	terminal	0
C	v(C) = +10 + 0.9 × v(D) = 10 + 0	10
B	v(B) = −1 + 0.9 × v(C) = −1 + 9	8
A	v(A) = −1 + 0.9 × v(B) = −1 + 7.2	6.2

These match the values found in Part 1 by adding up rewards, as they must.

4.4 Solved example 2: checking the random policy (π_{rand})

Later (Phase 7) we will compute $v_{\pi rand}(A) = -1.42$, $v_{\pi rand}(B) = 0.48$, $v_{\pi rand}(C) = 4.72$ (rounded). Let us check state C with Equation (7):

Action	π(a | C)	Next state	R + γv(s')	π × (R + γv)
Left	0.5	B	−1 + 0.9 × 0.48 = −0.568	−0.284
Right	0.5	D	+10 + 0.9 × 0 = 10	5.000
			v(C) = sum	**4.716 ≈ 4.72 ✓**

Exercise 4.1 — One stochastic backup

In the slippery corridor (Figure 2), suppose $v_\pi(B) = 5$ and $v_\pi(C) = 10$, with $\gamma = 0.9$. Compute $q_\pi(B, Right)$.

Solution

Use Equation (5) with two possible next states:

Move to C (prob. 0.8): $0.8 \times [-1 + 0.9 \times 10] = 0.8 \times 8 = 6.4$

Slip, stay in B (prob. 0.2): $0.2 \times [-1 + 0.9 \times 5] = 0.2 \times 3.5 = 0.7$

$q_\pi(B, Right) = 6.4 + 0.7 =$ **7.1**. (Without slipping it would be −1 + 9 = 8; the risk of slipping lowers it.)

Exercise 4.2 — A policy that never finishes

Find the values of the policy "always Left" in the corridor ($\gamma = 0.9$) using Equation (8).

Solution

A: Left keeps the agent in A, so v(A) = −1 + 0.9 v(A). Then 0.1 v(A) = −1, so **v(A) = −10**.

B: Left leads to A, so v(B) = −1 + 0.9 × (−10) = **−10**.

C: Left leads to B, so v(C) = −1 + 0.9 × (−10) = **−10**.

The agent never reaches the goal and pays −1 forever: −1/(1 − 0.9) = −10, the constant-reward formula from Part 1. Note that the Bellman equation can be solved directly as a small equation, without adding infinitely many rewards.

5. Optimal Values and the Optimal Policy

Different policies give different values. A policy π is **better than or equal to** another policy π′ if its value is at least as large in **every** state. There is always at least one policy that is better than or equal to all others: an **optimal policy**, written π_*.

$$v_*(s) = \max_\pi \; v_\pi(s), \qquad q_*(s,a) = \max_\pi \; q_\pi(s,a) \tag{9}$$

- **$v_*(s)$**: the optimal state value, the largest expected return achievable from s by any policy.
- **$q_*(s, a)$**: the optimal action value, the expected return of taking a in s and behaving optimally afterwards.
- **$\max_\pi$**: the maximum over all policies.

5.1 Bellman optimality equations

An optimal agent does not average over actions; it takes the **best** one. So the policy average in Equation (7) is replaced by a **max**:

$$v_*(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P(s' \mid s, a) \left[R(s,a,s') + \gamma \, v_*(s') \right] \tag{10}$$

$$q_*(s,a) = \sum_{s' \in \mathcal{S}} P(s' \mid s, a) \left[R(s,a,s') + \gamma \max_{a'} q_*(s', a') \right] \tag{11}$$

Here a′ is an action in the next state s′. In words: *the best value of a state = the value of its best action*.

5.2 From q_* to the optimal policy

Once q_* is known, acting optimally is easy: in each state pick the action with the largest q_*:

$$\pi_*(s) = \arg\max_{a \in \mathcal{A}} \; q_*(s,a) \tag{12}$$

5.3 Solved example: optimal values of the corridor

The optimal values are $v_*(A) = 6.2$, $v_*(B) = 8$, $v_*(C) = 10$ (we will compute them in Phases 9–10). Compute every $q_*(s, a) = R + 0.9 \times v_*(s')$:

State	$q_*(s,$ Left)	$q_*(s,$ Right)	max = $v_*(s)$	$\pi_*(s)$
A	−1 + 0.9 × 6.2 = 4.58	−1 + 0.9 × 8 = 6.2	6.2 ✓	Right
B	−1 + 0.9 × 6.2 = 4.58	−1 + 0.9 × 10 = 8	8 ✓	Right
C	−1 + 0.9 × 8 = 6.2	+10 + 0.9 × 0 = 10	10 ✓	Right

In every state, the max of the q_* values equals v_*, so these values satisfy the Bellman optimality equation, and the optimal policy is **Right everywhere**.

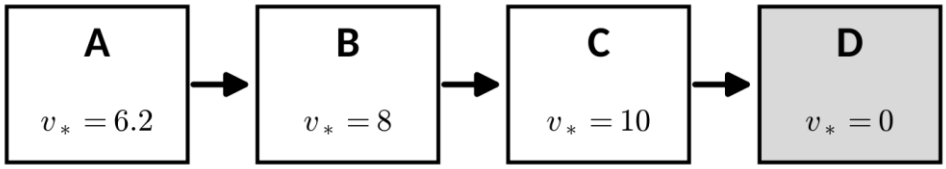


Figure 4. Optimal values and the optimal policy of the corridor (γ = 0.9).

Exercise 5.1 — Read off the optimal policy

A small MDP has two states X and Y. Its optimal action values are: $q_*(X, a_1) = 3$, $q_*(X, a_2) = 7$, $q_*(Y, a_1) = 5$, $q_*(Y, a_2) = 2$. Find $v_*(X)$, $v_*(Y)$ and π_*.

Solution

X: max(3, 7) = 7, so $\mathbf{v_*(X) = 7}$, $\mathbf{\pi_*(X) = a_2}$.

Y: max(5, 2) = 5, so $\mathbf{v_*(Y) = 5}$, $\mathbf{\pi_*(Y) = a_1}$.

Exercise 5.2 — Check optimality with a different γ

With γ = 0.5, a student claims $v_*(A) = 1$, $v_*(B) = 4$, $v_*(C) = 10$. Check the Bellman optimality equation in state A.

Solution

Left: $-1 + 0.5 \times v(A) = -1 + 0.5 = -0.5$. Right: $-1 + 0.5 \times v(B) = -1 + 2 = 1$.

$\max(-0.5, 1) = 1 = v(A)$ ✓ The equation holds in A (and the same check works in B and C).

6. What Is Planning?

Planning means computing a good (ideally optimal) policy when the **model of the MDP is known**: we know P(s' | s, a) and R(s, a, s'). The agent does not need to try actions in the real world; it can "think ahead" on paper. The planning methods in this tutorial are also called **dynamic programming** (DP) methods, because they solve a big problem by reusing the answers for smaller pieces (the values of next states).

Task	Input	Output	Method in this tutorial
Prediction (evaluation)	MDP + a fixed policy π	v_π	Iterative policy evaluation (Phase 7)

Task	Input	Output	Method in this tutorial
Control (optimisation)	MDP only	π_* and v_*	Policy iteration (Phase 9), value iteration (Phase 10), linear programming (Phase 11)

Planning vs learning

Planning needs the model (P and R). When the model is **not** known, the agent must learn from experience instead; Q-learning (Part 3) does exactly that. Both aim at the same target: the Bellman optimality equation.

Exercise 6.1 — Prediction or control? Planning or learning?

(a) Given the corridor model and π_{rand}, compute how good each state is. (b) Given the corridor model, find the best way to reach D. (c) A robot in an unknown building improves by trial and error.

Solution

(a) **Prediction**, by planning (model known, policy fixed).

(b) **Control**, by planning (model known, best policy wanted).

(c) **Learning**, not planning, because the model is unknown.

7. Iterative Policy Evaluation

Goal: given a policy π, compute $v_\pi(s)$ for every state. Equation (7) gives one equation per state. Instead of solving them all at once, we start with a guess (all zeros) and repeatedly apply Equation (7) as an **update rule**:

$$v_{k+1}(s) = \sum_a \pi(a \mid s) \sum_{s' \in \mathcal{S}} P(s' \mid s, a) \left[R(s, a, s') + \gamma \, v_k(s') \right] \tag{13}$$

- **k**: the iteration (sweep) number, k = 0, 1, 2, …
- **$v_k(s)$**: our estimate of $v_\pi(s)$ after k sweeps. We start with $v_0(s) = 0$ for all s.
- **$v_{k+1}(s)$**: the new estimate, computed using only the *old* values v_k. The terminal state always keeps value 0.

Each sweep updates every state once. The estimates get closer and closer to v_π (for $\gamma < 1$ this is guaranteed). We stop when the biggest change in a sweep is tiny:

$$\Delta = \max_{s \in \mathcal{S}} \left| \, v_{k+1}(s) - v_k(s) \, \right| < \theta \tag{14}$$

Δ (delta) is the largest change in any state during the sweep, and **θ** (theta) is a small threshold we choose, such as 0.001.

Algorithm 1: Iterative policy evaluation

```
Input: policy π, model P and R, discount γ, threshold θ
Set v(s) = 0 for every state s
Repeat:
  Δ = 0
  For each non-terminal state s:
    new(s) = ∑_a π(a|s) ∑_s' P(s'|s,a) [R(s,a,s') + γ v(s')]
    Δ = max(Δ, |new(s) − v(s)|)
  v = new        (replace all values after the sweep)
Until Δ < θ
Output: v ≈ v_π
```

7.1 Solved example: evaluating the random policy

For π_{rand} (each action with probability 0.5) in the corridor, Equation (13) becomes, for each state:

- $v_{k+1}(A) = 0.5[-1 + 0.9\, v_k(A)] + 0.5[-1 + 0.9\, v_k(B)]$ (Left stays in A)
- $v_{k+1}(B) = 0.5[-1 + 0.9\, v_k(A)] + 0.5[-1 + 0.9\, v_k(C)]$
- $v_{k+1}(C) = 0.5[-1 + 0.9\, v_k(B)] + 0.5[+10 + 0.9 \times 0]$

Sweep 1 (from $v_0 = 0, 0, 0$): A = 0.5(−1) + 0.5(−1) = −1; B = −1; C = 0.5(−1) + 0.5(10) = 4.5.

Sweep 2 (from $v_1 = -1, -1, 4.5$):

- A = 0.5[−1 + 0.9(−1)] + 0.5[−1 + 0.9(−1)] = 0.5(−1.9) + 0.5(−1.9) = −1.9
- B = 0.5[−1 + 0.9(−1)] + 0.5[−1 + 0.9(4.5)] = 0.5(−1.9) + 0.5(3.05) = 0.575
- C = 0.5[−1 + 0.9(−1)] + 0.5(10) = −0.95 + 5 = 4.05

Sweep k	$v_k(A)$	$v_k(B)$	$v_k(C)$	$v_k(D)$	Δ
0	0	0	0	0	—
1	−1	−1	4.5	0	4.5
2	−1.9	0.575	4.05	0	1.575
3	−1.596	−0.033	4.759	0	0.708
10	−1.497	0.429	4.682	0	small
20	−1.433	0.475	4.713	0	smaller
∞ (converged)	−1.424	0.482	4.717	0	0

The values settle at $v_{\pi rand} \approx (-1.42, 0.48, 4.72)$. Wandering randomly is much worse than always going Right (6.2, 8, 10).

Exercise 7.1 — Evaluate a looping policy

Policy π_2: Right in A, Left in B, Right in C ($\gamma = 0.9$). (a) Do sweeps 1, 2 and 3 of iterative policy evaluation. (b) Find the exact values by solving the Bellman equations.

Solution

Updates: A → B (−1), B → A (−1), C → D (+10).

Sweep	v(A) = −1 + 0.9 v(B)	v(B) = −1 + 0.9 v(A)	v(C) = 10
0	0	0	0
1	−1	−1	10
2	−1.9	−1.9	10
3	−2.71	−2.71	10

(b) By symmetry v(A) = v(B) = x, so x = −1 + 0.9x, giving **x = −10**; and **v(C) = 10**. The iterative values (−1, −1.9, −2.71, …) slowly approach −10: A and B bounce the agent back and forth for ever.

8. Policy Improvement

Goal: given a policy π and its values v_π, find a better policy. **Idea:** in each state, look one step ahead with Equation (5) and choose the action with the largest $q_\pi(s, a)$. This is called acting **greedily** with respect to v_π:

$$\pi'(s) = \arg\max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P(s' \mid s, a) \left[R(s, a, s') + \gamma \, v_\pi(s') \right] = \arg\max_a q_\pi(s, a) \tag{15}$$

π′ ("pi-prime") is the new, improved policy. The **policy improvement theorem** guarantees that the new policy is never worse:

$$v_{\pi'}(s) \;\geq\; v_\pi(s) \quad \text{for every state } s \tag{16}$$

If the greedy policy is the same as the old one (no state changes its action), then π already satisfies the Bellman optimality equation, so it is **optimal**.

Ties

If two actions have the same q-value, either may be chosen. A useful rule is: *on a tie, keep the old action*. This prevents the policy from switching back and forth for ever between equally good actions.

8.1 Solved example: improving the random policy

Using $v_{\pi rand}$ = (−1.42, 0.48, 4.72) from Phase 7:

State	q(s, Left)	q(s, Right)	Greedy π'(s)
A	−1 + 0.9 × (−1.42) = −2.28	−1 + 0.9 × 0.48 = −0.57	Right
B	−1 + 0.9 × (−1.42) = −2.28	−1 + 0.9 × 4.72 = 3.25	Right
C	−1 + 0.9 × 0.48 = −0.57	+10 + 0.9 × 0 = 10	Right

One improvement step already gives "always Right", whose values (6.2, 8, 10) are higher in every state than (−1.42, 0.48, 4.72), exactly as the theorem promises.

Exercise 8.1 — Improve "always Left"

The policy "always Left" has v = (−10, −10, −10) for (A, B, C) (Exercise 4.2). Do one policy improvement step. Use the rule "on a tie, keep the old action".

Solution

State	q(s, Left)	q(s, Right)	New action
A	−1 + 0.9(−10) = −10	−1 + 0.9 v(B) = −10	tie → keep **Left**
B	−1 + 0.9 v(A) = −10	−1 + 0.9 v(C) = −10	tie → keep **Left**
C	−1 + 0.9 v(B) = −10	+10 + 0 = 10	**Right**

New policy: (Left, Left, Right). Only C changes, because C is the only state that can "see" the goal one step ahead. The next phase continues from here.

9. Policy Iteration

Policy iteration simply alternates the two previous steps until the policy stops changing:

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \longrightarrow \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_* \qquad (17)$$

Here **E** means *evaluate* (Phase 7) and **I** means *improve* (Phase 8). Each improvement gives a policy at least as good as the last, and a finite MDP has only finitely many deterministic policies (Phase 3), so the process must stop, and when it stops the policy is optimal.

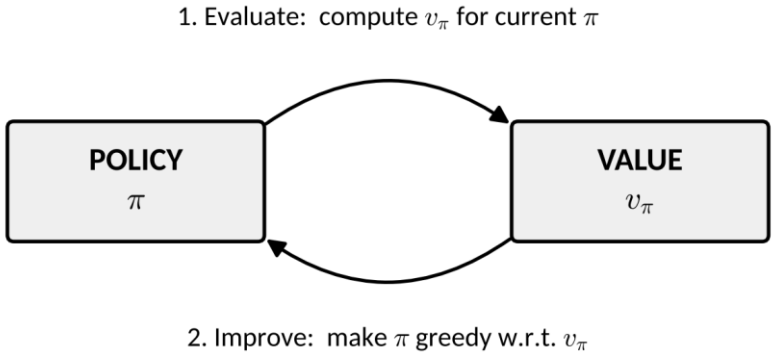


Figure 5. Policy iteration: evaluation and improvement repeat until the policy no longer changes.

Algorithm 2: Policy iteration

```
Start with any policy π (e.g., random or all-Left)
Repeat:
  1. Policy evaluation: compute v_π  (Algorithm 1, or solve the equations)
  2. Policy improvement: for each state s
       π'(s) = argmax_a ∑_s' P(s'|s,a)[R(s,a,s') + γ v_π(s')]
  3. If π' = π (no change in any state): STOP, π is optimal
     Otherwise set π = π' and repeat
```

9.1 Solved example: policy iteration from "always Left"

Policies are written as (action in A, action in B, action in C). Ties keep the old action. In this deterministic corridor each evaluation can be solved exactly, as in Exercise 4.2.

Iter.	Policy π	Evaluate: v(A), v(B), v(C)	Improve: key comparison	New policy
0	(L, L, L)	−10, −10, −10	C: Right 10 > Left −10; A, B tie	(L, L, R)
1	(L, L, R)	−10, −10, 10	B: Right −1 + 0.9(10) = 8 > Left −10	(L, R, R)
2	(L, R, R)	−10, 8, 10	A: Right −1 + 0.9(8) = 6.2 > Left −10	(R, R, R)
3	(R, R, R)	6.2, 8, 10	No state improves (see table in 5.3)	(R, R, R) → STOP

Check of iteration 1 evaluation: with (L, L, R), A loops on Left (v(A) = −10), B goes to A (−1 + 0.9(−10) = −10), and C goes to D (10). **Check of iteration 2:** with (L, R, R), v(B) = −1 + 0.9 × 10 = 8.

What we see: the good news from the goal travels backwards one state per iteration, C first, then B, then A. After 4 evaluations the optimal policy "always Right" and v_* = (6.2, 8, 10) are found.

Exercise 9.1 — Policy iteration from the random policy

Run policy iteration starting from π_{rand}. Use the results of Phases 7 and 8. How many improvement steps are needed?

Solution

Iter.	Policy	Values (A, B, C)	Greedy policy
0	π_{rand}	−1.42, 0.48, 4.72	(R, R, R) (Phase 8.1)
1	(R, R, R)	6.2, 8, 10	(R, R, R): no change → STOP

Only **one** real improvement step was needed; the second improvement step only confirms that nothing changes. The starting policy affects how many iterations are needed, but not the final answer.

10. Value Iteration

Policy iteration fully evaluates every policy, which can be slow. **Value iteration** skips the separate evaluation: it applies the Bellman *optimality* equation (10) directly as an update rule, taking the **max over actions** in every sweep:

$$v_{k+1}(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P(s' \mid s, a) \left[R(s, a, s') + \gamma \, v_k(s') \right] \tag{18}$$

- **$v_k(s)$**: the estimate after k sweeps, starting from $v_0(s) = 0$.
- **$\max_a$**: instead of averaging over a policy, take the best action.
- Compare with Equation (13): the only change is that $\sum_a \pi(a \mid s)$ has become $\max_a$.

The estimates v_k converge to v_*. When the change Δ is below θ, we stop and read off the optimal policy greedily, using Equation (12) with the final values.

Algorithm 3: Value iteration

```
Set v(s) = 0 for every state s
Repeat:
  Δ = 0
  For each non-terminal state s:
    new(s) = max_a Σ_s' P(s'|s,a) [R(s,a,s') + γ v(s')]
    Δ = max(Δ, |new(s) − v(s)|)
  v = new
Until Δ < θ
Output policy: π(s) = argmax_a Σ_s' P(s'|s,a) [R(s,a,s') + γ v(s')]
```

10.1 Solved example: value iteration on the corridor (γ = 0.9)

In each sweep, compute both action values from the *previous* row, and keep the larger.

k	A: Left / Right	$v_k(A)$	B: Left / Right	$v_k(B)$	C: Left / Right	$v_k(C)$
0	—	0	—	0	—	0
1	−1 / −1	−1	−1 / −1	−1	−1 / 10	10
2	−1.9 / −1.9	−1.9	−1.9 / 8	8	−1.9 / 10	10
3	−2.71 / 6.2	6.2	−2.71 / 8	8	6.2 / 10	10
4	4.58 / 6.2	6.2	4.58 / 8	8	6.2 / 10	10

How row 2 was computed (using row 1: v(A) = −1, v(B) = −1, v(C) = 10):

- A: Left = −1 + 0.9(−1) = −1.9; Right = −1 + 0.9 v(B) = −1 + 0.9(−1) = −1.9; max = −1.9
- B: Left = −1 + 0.9 v(A) = −1.9; Right = −1 + 0.9 v(C) = −1 + 9 = 8; max = 8
- C: Left = −1 + 0.9 v(B) = −1.9; Right = 10 + 0 = 10; max = 10

Row 4 equals row 3, so Δ = 0 and we stop. Final values (6.2, 8, 10) = v_*, and Right is the larger action value in every state, so π_* = always Right. Figure 6 shows how the values spread backwards from the goal.

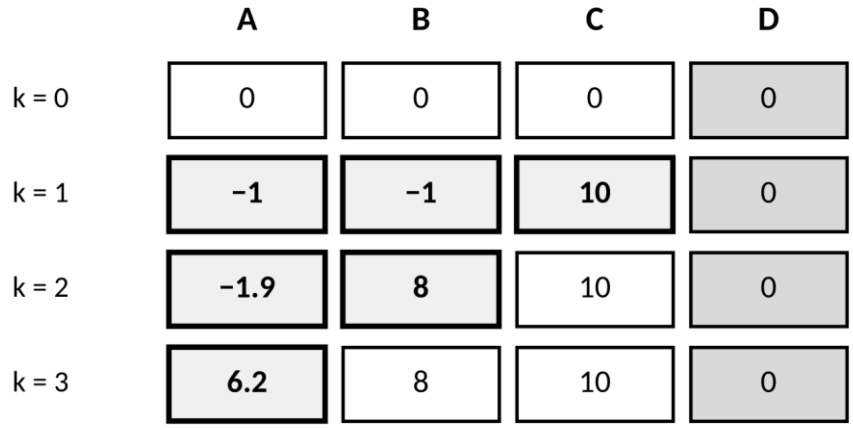


Figure 6. Value iteration on the corridor. Each sweep carries the goal's reward one cell further back.

Why did it finish so quickly?

The corridor is deterministic and very short, so the exact optimal values appear after 3 sweeps. In general (especially with loops and randomness) value iteration only *approaches* v_* step by step, and we stop when $\Delta < \theta$.

Exercise 10.1 — Value iteration with γ = 0.5

Repeat value iteration on the corridor with γ = 0.5. Fill in sweeps 1 to 4 and give the optimal policy.

Solution

k	A: L / R	v(A)	B: L / R	v(B)	C: L / R	v(C)
1	−1 / −1	−1	−1 / −1	−1	−1 / 10	10
2	−1.5 / −1.5	−1.5	−1.5 / 4	4	−1.5 / 10	10
3	−1.75 / 1	1	−1.75 / 4	4	1 / 10	10
4	−0.5 / 1	1	−0.5 / 4	4	1 / 10	10

Example (sweep 3, A): Left = −1 + 0.5(−1.5) = −1.75; Right = −1 + 0.5 × 4 = 1.

Converged: v_* = (1, 4, 10), π_* = always Right. These match Part 1, Exercise 12.1. A smaller γ lowers the values far from the goal, but here the best policy is unchanged.

Exercise 10.2 — One stochastic value-iteration update

Slippery corridor (Figure 2): in B, Right reaches C with probability 0.8 and stays in B with probability 0.2; Left reaches A with probability 1. All these moves give −1. With γ = 0.9 and current estimates $v_k(A) = 0$, $v_k(B) = 5$, $v_k(C) = 10$, compute $v_{k+1}(B)$.

Solution

Left: $1 \times [-1 + 0.9 \times 0] = -1$.

Right: $0.8 \times [-1 + 0.9 \times 10] + 0.2 \times [-1 + 0.9 \times 5] = 0.8 \times 8 + 0.2 \times 3.5 = 6.4 + 0.7 = 7.1$.

$v_{k+1}(B) = \max(-1, 7.1) =$ **7.1**.

11. Planning as a Linear Program (Brief)

A third planning method writes v_* as the solution of a **linear program (LP)**, an optimisation problem with a linear objective and linear constraints:

$$\min_{v} \sum_{s \in \mathcal{S}} \alpha(s)\, v(s) \quad \text{subject to} \quad v(s) \;\geq \sum_{s' \in \mathcal{S}} P(s' \mid s, a)\big[\, R(s, a, s') + \gamma\, v(s') \,\big] \quad \text{for all } s, a \tag{19}$$

- **v(s)**: the unknowns (one per state).
- **α(s) > 0**: any positive weights (for example, all equal to 1).
- **Constraints:** for every state s and every action a, v(s) must be at least the one-step value of taking a. So v(s) is at least the value of the *best* action.

Why it works: the constraints force v(s) to be at least the best action value, and minimising pushes v(s) down until it touches that value. The minimum is reached exactly at v_*. LP is useful in theory and for small problems; policy and value iteration are usually used in practice.

Exercise 11.1 — Write the constraints

(a) Write the two LP constraints for state B of the corridor (γ = 0.9). (b) Check that $v_* = (6.2, 8, 10)$ satisfies them. Which one is tight (holds with equality)?

Solution

(a) Left: $v(B) \geq -1 + 0.9\, v(A)$. Right: $v(B) \geq -1 + 0.9\, v(C)$.

(b) Left: $8 \geq -1 + 0.9 \times 6.2 = 4.58$ ✓ Right: $8 \geq -1 + 0.9 \times 10 = 8$ ✓(**tight**).

The tight constraint belongs to the optimal action (Right). This is always the case at the LP solution.

12. Comparison and Summary

12.1 Policy iteration vs value iteration

	Policy iteration	**Value iteration**
Equation used	Bellman expectation (evaluate) + greedy step (improve)	Bellman optimality, every sweep
Each iteration	Full evaluation of the current policy, then improvement	One sweep with a max over actions
Cost per iteration	High (evaluation can take many sweeps)	Low
Number of iterations	Usually few	Usually more
Stops when	Policy does not change	$\Delta < \theta$
Corridor result	4 evaluations from “always Left”	3 sweeps (4th confirms)
Needs the model?	Yes	Yes

12.2 Notation at a glance

Symbol	**Meaning**
$\langle \mathcal{S}, \mathcal{A}, P, R, \gamma \rangle$	An MDP: states, actions, transition probabilities, rewards, discount
$P(s' \mid s, a)$	Probability that action a in state s leads to state s’
$R(s, a, s')$	Reward for that transition
v_π, q_π	Values of a given policy π
v_*, q_*, π_*	Optimal state values, action values, policy

Symbol	Meaning
v_k	Estimate after k sweeps
Δ, θ	Largest change in a sweep; stopping threshold
π'	Improved (greedy) policy
$\alpha(s)$	Positive weights in the LP objective

12.3 Key equations

Name	Equation
Bellman expectation	$v_\pi(s) = \sum_a \pi(a\|s) \sum_{s'} P(s'\|s,a)[R + \gamma v_\pi(s')]$
Bellman optimality	$v_*(s) = \max_a \sum_{s'} P(s'\|s,a)[R + \gamma v_*(s')]$
Optimal policy	$\pi_*(s) = \text{argmax}_a\, q_*(s, a)$
Policy evaluation update	$v_{k+1}(s) = \sum_a \pi(a\|s) \sum_{s'} P(s'\|s,a)[R + \gamma v_k(s')]$
Value iteration update	$v_{k+1}(s) = \max_a \sum_{s'} P(s'\|s,a)[R + \gamma v_k(s')]$

12.4 Common mistakes to avoid

- Forgetting that each row of the transition table must sum to 1.
- Giving the terminal state a non-zero value, or updating it.
- Using new values in the middle of a sweep when the method says to use the old row v_k. (Updating in place also converges, but gives different intermediate numbers.)
- Averaging over actions in value iteration. Value iteration takes the **max**; policy evaluation takes the **policy-weighted average**.
- Thinking planning needs trial and error. Planning uses the known model; learning is for unknown models.
- Changing the action on a tie during policy improvement, which can make policy iteration flip-flop.

12.5 Quick self-test

Exercise 12.1 — Five quick questions

1. Name the five parts of an MDP.
2. How many deterministic policies does an MDP with 5 non-terminal states and 3 actions have?
3. What is the one difference between the policy-evaluation update and the value-iteration update?
4. When does policy iteration stop?
5. In the corridor, which action has $q_*(B, \cdot) = 4.58$?

Solution

1. States , actions , transition probabilities P, rewards R, discount γ.
2. 3^5 = **243**.
3. Policy evaluation averages over actions with $\pi(a \mid s)$; value iteration takes the **max** over actions.
4. When policy improvement leaves the policy unchanged in every state.
5. **Left** ($-1 + 0.9 \times 6.2 = 4.58$).

12.6 What comes next

All methods here need the model P and R. **Part 3: Q-learning** removes that need: the agent learns q_* directly from experience (s, a, r, s'), using a sampled version of the Bellman optimality equation (11) and the ε-greedy exploration from Part 1.

References

- R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed., MIT Press, 2018. Chapter 3 (Finite Markov Decision Processes) and Chapter 4 (Dynamic Programming).
- M. Mohri, A. Rostamizadeh and A. Talwalkar, *Foundations of Machine Learning*, 2nd ed., MIT Press, 2018. Reinforcement Learning chapter: MDP model, policy, planning algorithms (value iteration, policy iteration, linear programming).