# EXERCISE 2
# Dynamical Programming

Tue Herlau
tuhe@dtu.dk

14 February, 2025

**Objective:** The goal of this exercise is to introduce the dynamical programming (DP) algorithm in its most general form (backward-dp). To practically implement it, we need to introduce a model-class to represent the various terms in the DP problem such as $f_k$ and $g_k$. (36 lines of code)
**Exercise code:** `https://lab.compute.dtu.dk/02465material/02465students.git`
**Online documentation:** 02465material.pages.compute.dtu.dk/02465public/exercises/ex02.html

## Contents



# 1 Conceptual question: A windy walk on the line

☞ Consider a simple game where the agent can walk right and left on a line, but is sometimes blown one step by wind.

We formally define this as consisting of states $\mathcal{S}_k = \mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$ and in each state take an actions $\mathcal{A}_k(x_k) = \{-1, 1\}$ (i.e., move right or left). When the agent

takes an action, what happens is that our simulation of the environment generates $m$ random numbers:

$$v_1, \ldots, v_m \in \{0, 1\}$$

(generated i.i.d. with probability 0.5). Suppose we let

$$w_k = v_1 + v_2 + \cdots + v_m$$

and the update rule is then simply

$$x_{k+1} = f_k(x_k, u_k, w_k) = x_k + u_k + w_k$$

For instance, if $m = 2$ and we go right $u_k = 1$, but the wind blow us as $v_1 = 0$ and $v_2 = 1$ the next state is:

$$x_{k+1} = x_k + 1 + (0 + 1) = x_k + 2.$$

**(a.)** Assume that $m = 1$. Determine $P_W(w_k|x_k, u_k)$

**(b.)** Assume that $m = 2$. Determine $P_W(w_k|x_k, u_k)$

**(c.)** Assume that $m = 1$, and that we select

$$x_{k+1} = f_k(x_k, u_k, w'_k) = w'_k.$$

Determine a suitable choice for the noise distribution $P_W(w'_k|x_k, u_k)$ so that this new model is in fact equivalent to the one considered in the two first questions. (*Hint: As the notation indicate, you need to re-define what values $w'_k$ can take*).

**The last formulation, i.e. setting $f_k(x_k, u_k, w_k) = w_k$ and letting $p_W$ do all the work, is in fact how we will end up addressing the Pacman problem in project 1.**

## 2 A deterministic variant of the inventory-control problem (`deterministic_inventory.py`)

This section will consider a deterministic variant of the inventory-control problem which we will use in the next exercise when we implement a simpler (deterministic) variant of the DP algorithm (but don't worry, we will get to the full version soon!).

The inventory-control problem you will consider here has the same dynamics and cost-function, but we assume that the number of customers on day $k$ is exactly $w_k = k+1$. In the language of probabilities it means $w_k$ can take one value, $k + 1$, with probability 1:

$$P_W(w_k = k + 1|x_k, u_k) = 1.$$

Your task is to implement this variant of the inventory environment. To do that start with the file `deterministic_inventory.py`. The model inherits from our general inventory model given below; once all functions are filled out correctly, the model is complete and can be used for dynamical programming later.

```python
# dp_model.py
class DPModel:
    def __init__(self, N):
        self.N = N  # Store the planning horizon.

    def f(self, x, u, w, k: int):
        raise NotImplementedError("Return f_k(x,u,w)")

    def g(self, x, u, w, k: int) -> float:
        raise NotImplementedError("Return g_k(x,u,w)")

    def gN(self, x) -> float:
        raise NotImplementedError("Return g_N(x)")

    def S(self, k: int):
        raise NotImplementedError("Return state space as set S_k = {x_1, x_2, ...}")

    def A(self, x, k: int):
        raise NotImplementedError("Return action space as set A(x_k) = {u_1, u_2,
        ↪  ...}")

    def Pw(self, x, u, k: int):
        # Compute and return the random noise disturbances here.
        # As an example:
        return {'w_dummy': 1/3, 42: 2/3}  # P(w_k="w_dummy") = 1/3, P(w_k =42)=2/3.
```

**Problem 1** *Deterministic inventory control*

Implement the (deterministic) model by making sure the `Pw(self, x, u, k)` function is filled out correctly. It should return a dictionary, with one element, that represents the deterministic dynamics. You can find a complete implementation of the inventory-control problem (with probabilities) near the end of today's slides.

**Info:** Use the tests to check your implementation.

# 3 Implementing deterministic DP (`dp.py`)

The main goal today is to implement the dynamical programming algorithm. We will first implement a simpler version suitable for deterministic problems (and test it on the deterministic inventory control problem from the previous exercise) and then later generalize it to the complete case.

Since we consider a deterministic variant of the DP algorithm, this means that you can ignore the noise terms $w$. Concretely, in the DP algorithm [Her25, algorithm 1] line 9–12 can be simplified to:

$$Q_u = g_k(x_k, u_k, 0) + J_{k+1}(f_k(x_k, u_k, 0))$$

Problem 2 *Deterministic DP*

Implement the DP algorithm as described in [Her25, algorithm 1], using comments in the exercise and in the pseudo code. I recommend that you first implement the version described in [Her25, section 6.2.1] where the above simplification is applied to line 9–12.

Verify the first steps of the solution agrees with the expected output; if one of the cost function terms $J_k(x)$ differ, you have a mistake!

**Info:** Since the DP algorithm starts at $k = N$ and proceeds backwards you should focus on the first $k$ where the output differs from the expected output in [Her25, section 6.2.1]. Carefully follow the standard debugging recipe of using breakpoints/stepping the code to find the first time a quantity is updated wrongly, then figure out why it is updated wrongly, and fix the problem. When done, you should obtain the following output:

```
J_0(0) = 6.0, J_0(1) = 5.0, J_0(2) = 5.0
J_1(0) = 5.0, J_1(1) = 4.0, J_1(2) = 3.0
J_2(0) = 3.0, J_2(1) = 2.0, J_2(2) = 1.0
J_3(0) = 0.0, J_3(1) = 0.0, J_3(2) = 0.0
Total cost when starting in state x_0 = 2: J[0][2]=5 (and should be 5)
```

**Any** differences means you have a bad implementation and the following exercises will fail.

# 4 Implementing stochastic DP (`inventory.py`)

Once done, we can increase the complexity slightly by including the noise distribution (obviously, you might have implemented this already, but now we will test it). Recall we represent the noise distribution as a dictionary: `{..., w:pw, ...}`. The implementation should be quite similar to the above, except it should include an extra loop to account for the average over the noise parameters, see [Her25, algorithm 1]. If you are having problems debugging the code, consult [Her25, section 6.2.3] for a detailed example of the intermediate states of the algorithm.

Problem 3 *Stochastic DP*

- First complete the function `pW` in `inventory.py`. The functions should return the random noise disturbances and their probabilities as a dictionary (see the online documentation).
- Next, ensure your DP algorithm implementation in `dp.py` includes the loop over noise terms. When done, run the code and inspect the output to get a sense of how it represents the optimal policy and value function.

**Info:** The code should produce the following output

```
Inventory control optimal policy/value functions
 J_0(x_0=0) = 3.70,  J_0(x_0=1) = 2.70,  J_0(x_0=2) = 2.82
 J_1(x_1=0) = 2.50,  J_1(x_1=1) = 1.50,  J_1(x_1=2) = 1.68
 J_2(x_2=0) = 1.30,  J_2(x_2=1) = 0.30,  J_2(x_2=2) = 1.10
pi_0(x_0=0) = 1, pi_0(x_0=1) = 0, pi_0(x_0=2) = 0
pi_1(x_1=0) = 1, pi_1(x_1=1) = 0, pi_1(x_1=2) = 0
pi_2(x_2=0) = 1, pi_2(x_2=1) = 0, pi_2(x_2=2) = 0
```

# 5 Exam question: Counting states

Suppose the Dynamical Programming algorithm is applied to a problem where the following is known:

- $N = 10$
- The size of the action spaces are $|\mathcal{A}_k(x_k)| = 4$
- The size of the states spaces are $|\mathcal{S}_0| = 1$, $|\mathcal{S}_N| = 2$ and otherwise $|\mathcal{S}_k| = 10$
- There are exactly two random noise disturbances, $w = 0$ and $w = 1$, available in any state/action combination:

$$P_W(w = 0|x, u) = P_W(w = 1|x_k, u_k) = \frac{1}{2}.$$

How many times does the dynamical programming algorithm need to evaluate $f_k$ in order to find the optimal policy?

**a.** 736

**b.** 744

**c.** 730

**d.** 728

**e.** Don't know.

# 6 The DP agent (`dp_agent.py`)

We are now ready to build the first serious agent, namely an agent which plan using the DP algorithm. In other words, we need both an environment, and a DP model that corresponds to that environment. The agent should then plan using the dp model to obtain an optimal policy $\pi* = \{\mu_k^*\}_{k=0}^{N-1}$, and then in step $k$ use policy function $\mu_k^*$. Since the Inventory control problem is the only one where we both have a model and an environment it will provide a good testbed. Once done, the interaction will look as follows:

```
# dp_agent.py
env = InventoryEnvironment(N=3)
inventory_model = InventoryDPModel(N=3)
agent = DynamicalProgrammingAgent(env, model=inventory_model)
stats, _ = train(env, agent, num_episodes=5000)
```

**Problem 4** *Dynamical Programming Agent*

Implement the missing functionality from the DP agent. Once done, verify the (sample estimate) of the optimal value function agrees with the (exact) DP result.

**Info:** You should expect the following output:

```
Estimated reward using trained policy and MC rollouts -3.6964
Reward as computed using DP -3.6999999999999997
```

Having to specify both an agent and an environment, when it is quite apparent we can derive the environment from the agent, is obviously not ideal. Subsequent exercises will fix this problem.

# 7 Exam question: The flower-store (`flower_store.py`) ★★

This problem focuses on a variant of the inventory control problem discussed in [Her25, section 5.1.2]. This inventory problem represents a flower-store such that $x_k$ denotes the

number of flower bouquets in stock at planning round $k$. The original inventory control model and the dynamical programming algorithm is included in the exam folder.

The following tasks can be solved by implementing suitable variants of the inventory control problem, and then applying dynamical programming to determine the optimal policy $\mu_0^*(x_0)$ and cost-function $J^*(x_0)$ in the starting state.

The flower store problem is equivalent to the inventory control problem on a horizon of $N$ with two changes:[1]:

- $g_k(x_k, u_k, w_k) = cu + |x_k + u_k - w_k|$
- The distribution of the number of items customers buy $w_k$ is:

$$p_W(w_k = 0|x_k, u_k) = 0.1, \quad p_W(w_k = 1|x_k, u_k) = 0.3, \quad p_W(w_k = 2|x_k, u_k) = 0.6.$$

**(a.)** Complete `def a_get_policy(N: int, c: float, x0 : int)` : This function is given a value of $N$, $c$ and a starting state $x_0$, and should return the action the optimal policy computes in $x_0$, i.e. $\mu_0^*(x_0)$, as an `int`.

**(b.)** Complete `def b_prob_one(N : int, x0 : int)` : For every policy and starting state $x_0$, there is a certain chance $p(x_N = 1|x_0)$ we will end up with a single item (bouquet) on the last day $N$ when following the policy. The clerk operating the store would very much like to bring this last bouquet home with her, and so she is solely concerned with determining the policy which maximize $p(x_N = 1|x_0)$, i.e. the chance she can bring home a single bouquet at the end of the planning period.

Determine what this chance is when we follow the policy which is *solely* concerned with with maximizing the chance that $x_N = 1$. The function should accept $N$ and $x_0$ as input argument, and return the value of $p(x_N = 1|x_0)$ as a `float`.

*Hint: Alter the cost-functions so that the optimal solution maximize this probability. The Pacman-problem where the winning probability is computed may provide inspiration.*

# References

[Her25] Tue Herlau. Sequential decision making. (Freely available online), 2025.

[1] The expression $|x|$ return the absolute value, i.e. $|4| = |-4| = 4$