ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institute for Dynamic Systems and Control
IDSC
Institut für Dynamische Systeme und Regelungstechnik

151-0563-01 **Dynamic Programming and Optimal Control** (Fall 2023)

Programming Exercise Topic: Infinite Horizon Problems

Issued: Nov 01, 2023 Due: Dec 18, 2023

Balloon Control for Internet on Mars

It is the year 2083, and humans are just starting to colonize Mars. Across the Martian surface, the most attractive spots are growing into dense cities. The internet infrastructure as we know it on the planet Earth is not available. Instead, a single super-light balloon enables communication between cities.

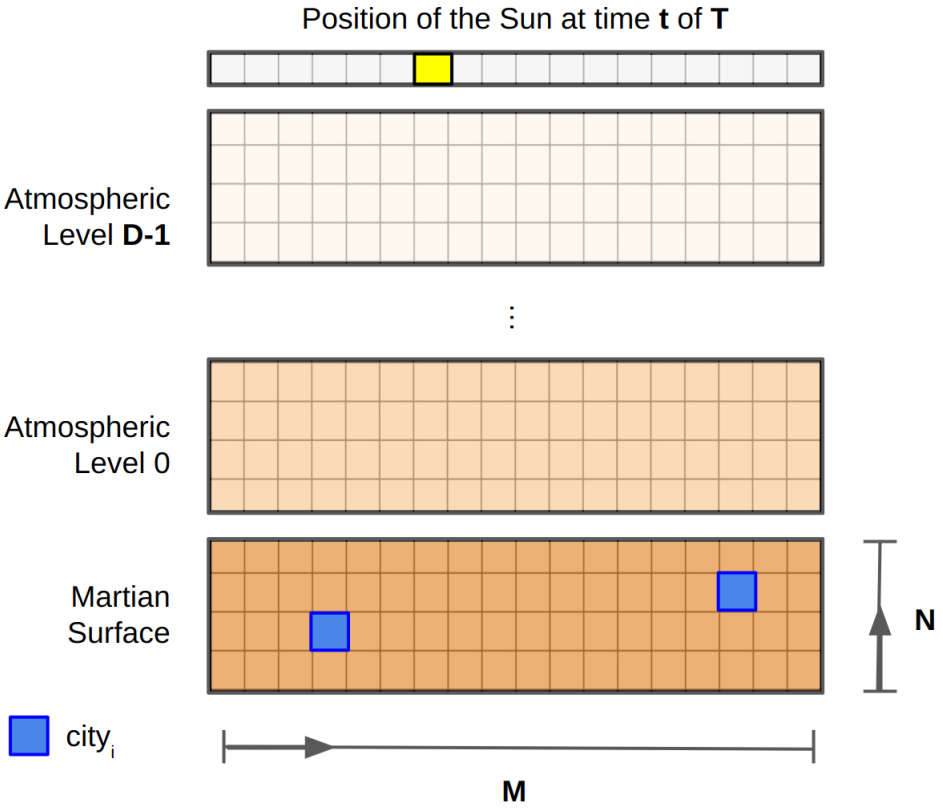
The balloon is solar-powered and floats in the Martian atmosphere, drifting according to the atmospheric winds. Without any control, the balloon would stray away from the cities, severely affecting the communication link between them. The strength of the communication link between a city and the balloon depends on the distance between them.

As a control engineer at MASA (Martian Aeronautics and Space Administration), your responsibility is to develop an optimal policy for controlling the balloon to maximize the strength of the communication between the cities and the solar power absorption.

To reduce the hardware and energy requirements, the balloon can only be controlled in the vertical direction[1]. Therefore, the balloon must take advantage of the wind blowing in different directions at different altitudes to position itself and, because of the variable sun position, the optimal location will change throughout the day!

[1]This setting is inspired by the Loon project at X, the moonshot factory: `https://x.company/projects/loon`

Problem Setup



Since resources on Mars are scarce, we want each balloon to last as long as possible, and we care about its performance at all times: A perfect fit with the infinite horizon framework. In the following, we describe how the engineers at MASA modeled the problem. We use the `teletype` font to refer to the names the variables have in the code we provide.

State Space

- We discretize the Martian surface into a $M \times N$ grid, where M (`Constants.M`) is the number of columns, indexed from west to east, and N (`Constants.N`) is the number of rows, indexed from south to north. There are D (`Constants.D`) atmospheric levels, indexed from the level above the ground upwards. The balloon is thus located in a three-dimensional grid of size $M \times N \times D$.

- The position of the sun is not fixed! In fact, it changes throughout the day and this affects the power absorbed by the balloon, depending on its relative position. The engineers at MASA have figured out the dynamics of the sun as a function of time, but the state of the balloon has to account for time, since the standard formulation of infinite horizon problems we have seen in class requires time-independence.

- The state space is thus as follows:

$$\begin{aligned} S = \{(t, z, y, x) \mid\ & t = 0, 1, \ldots, T-1 \\ & z = 0, 1, \ldots, D-1 \\ & y = 0, 1, \ldots, N-1 \\ & x = 0, 1, \ldots, M-1\}, \end{aligned} \tag{1}$$

where t is the time of the Martian day (which we discretize in T (`Constants.T`) time periods), z is the balloon's height, y the balloon's position along the south-north axis and x the balloon's position along the west-east axis.

Note: One way to compute the state space in Python from the constants `Constants.T`, `Constants.D`, `Constants.M`, `Constants.N` is the following:

```
import numpy as np
import itertools

t = np.arange(0, Constants.T)
z = np.arange(0, Constants.D)
y = np.arange(0, Constants.N)
x = np.arange(0, Constants.M)
state_space = np.array(list(itertools.product(t, z, y, x)))
```

Make sure that the indexing you use for each state is such that the state space is an array equivalent to the one obtained with the snippet above.

- The balloon has the required sensors to know its current position and a clock to know the current time of the day: The processor on the balloon has full knowledge of its current state and you can implement a feedback policy.

Input Space

- The control input is either to go **up** (`Constants.V_UP`), go **down** (`Constants.V_DOWN`) or **stay** (`Constants.V_STAY`) at the same level.
- We do not want to make the balloon leave the atmosphere, nor do we want it to crash to the ground. Thus, we set the constraints so that **up** is not allowed when $z = D - 1$ and that **down** is not allowed when $z = 0$.
- All together, the input space reads:

$$\mathcal{U}((t,z,y,x)) = \begin{cases} \{\texttt{Constants.V\_STAY}, \texttt{Constants.V\_UP}\} & \text{if } z = 0 \\ \{\texttt{Constants.V\_DOWN}, \texttt{Constants.V\_STAY}\} & \text{if } z = D-1 \\ \{\texttt{Constants.V\_DOWN}, \texttt{Constants.V\_STAY}, \texttt{Constants.V\_UP}\} & \text{otherwise.} \end{cases} \tag{2}$$

- In the code, you can generate the input space as follows:

```
input_space = [Constants.V_DOWN, Constants.V_STAY, Constants.V_UP]
```

Disturbances

In this setting, the disturbances are both unwanted and helpful! As we will discuss soon in detail, to some extent they hamper our actuation and yet allow us to move: The balloon would, otherwise, be able only to move vertically!

- The balloon moves up and down heating and cooling the air (which changes its density!). The process is prone to error: when the applied input is **up** or **down** the balloon moves in this direction with a probability of p_u and stays in the same level with a probability of $1 - p_u$. In the code, p_u is obtained as:

```
p_u = Constants.P_V_TRANSITION[1]    # 1 - p_u = Constants.P_V_TRANSITION[0]
```

However, it is guaranteed that the balloon will stay at the same level if the control input applied is **stay**.

- The discretization of the atmosphere in D (`Constants.D`) levels is related to the wind distribution. The wind can move the balloon in four directions:

  i) **north** (`Constants.H_NORTH`);

  ii) **south** (`Constants.H_SOUTH`);

  iii) **east** (`Constants.H_EAST`);

  iv) **west** (`Constants.H_WEST`).

  When there is no wind (`Constants.H_STAY`), the balloon is not displaced horizontally (i.e., within the same atmospheric level; it may still happen that the balloon changes level, depending on the input applied).

- The probabilities for all levels are collected in the array `Constants.P_H_TRANSITION`. The wind probabilities at the level `z` are collected in the array `Constants.P_H_TRANSITION[z]`:

```
Constants.P_H_TRANSITION              # Wind PDF for all levels.

p_h = Constants.P_H_TRANSITION[z]     # Wind PDF at level z = 0, 1, .., D-1.

p_h.P_WIND[Constants.H_STAY]          # Probability of no wind at level z.
p_h.P_WIND[Constants.H_NORTH]         # Probability of wind in the
                                      #   H_NORTH direction at level z.
p_h.P_WIND[Constants.H_SOUTH]         # Probability of wind in the
                                      #   H_SOUTH direction at level z.
p_h.P_WIND[Constants.H_WEST]          # Probability of wind in the
                                      #   H_WEST direction at level z.
p_h.P_WIND[Constants.H_EAST]          # Probability of wind in the
                                      #   H_EAST direction at level z.
```

Dynamics

- We discretize the Martian day into T (`Constants.T`) time intervals (reflecting the impact of the sun on the energy harvested). Thus, the time variable is circular, and its dynamics are

$$t_{k+1} = \begin{cases} t_k + 1 & t_k < T - 1 \\ 0 & t_k = T - 1. \end{cases}$$

- The wind disturbance that affects the balloon is the one at the atmospheric level where the balloon is located before we apply an input. That is, the sequence of events in one time period is as follows:

  1. The atmospheric wind is applied and the balloon is displaced within the same atmospheric level, along the **south** to **north** axis or along the **west** to **east** axis.
  2. The input is applied, and the vertical disturbance acts.

- There are safety features implemented on the balloon that geofence it within the grid. Along the **south** to **north** axis, the balloon has been constrained to stay within the grid using emergency thrusters that activate only if the balloon is going to leave the grid. You can assume that if the balloon tries to leave the grid along either the south or the north boundary it stays in place instead. However, along the **west** to **east** axis the world is circular. That is, the map is all around Mars along the **west** to **east** axis, but does not

cover the whole **south** to **north** axis. Thus, crossing the most eastern or most western boundaries brings the balloon to the opposite side of the map.

Consider for clarity the following two examples:

1. Suppose the balloon is at position x_k on the **west** to **east** axis, the input is `Constants.V_STAY` and the probability of being disturbed in the **east** direction is 1 (i.e. the probability of being disturbed in all the other directions is 0). Then, the dynamics along the x-axis are

$$x_{k+1} = \begin{cases} x_k + 1 & x_k < M - 1 \\ 0 & x_k = M - 1 \end{cases}$$

2. Suppose the balloon is at position y_k on the **south** to **north** axis, the input is `Constants.V_STAY` and the probability of being disturbed in the **south** direction is 1 (i.e. the probability of being disturbed in all the other directions is 0). Then, the dynamics along the **south** to **north** axis are

$$y_{k+1} = \begin{cases} y_k - 1 & y_k > 0 \\ 0 & y_k = 0 \end{cases}$$

- In the code, if `P` is the $(TDMN) \times (TDMN) \times 3$ matrix of the transition probabilities (with the ordering as prescribed by Sections **State Space** and **Input Space**) and the input `input_space[a]` corresponding to the index `a` is not allowed at the state `state_space[i]` corresponding to the index `i`, write:

```
P[i, j, a] = 0
```

If you follow our convention in the ordering and `Constants.V_UP` is not allowed at the state `state_space[i]`, you can write for all states `state_space[j]`:

```
P[i, j, Constants.V_UP] = 0
```

Your solution will be marked as incorrect if you do not adhere to this convention!

Cost Function

The engineers at MASA have realized that to obtain an optimal behavior that trades off connectivity strength and energy levels sufficient to guarantee (infinitely) long missions of the balloon, the control algorithm needs to minimize a cost function made of two parts:

- *Energy.* The sun traverses the map from east to west,

$$x_{\text{sun}}(t) = \left\lfloor (M-1)\, \frac{T-1-t}{T-1} \right\rfloor, \tag{3}$$

where $\lfloor \cdot \rfloor$ is the floor function defined as $\lfloor x \rfloor = \max\{m \in \mathbb{Z} \mid m \leq x\}$.

Hint: Use the Python function `np.floor`.

To compute the distance between the balloon and the sun, you need to account for the fact that the world is circular along the west to east axis. One way to do this is to create two fictitious images of the balloon, each shifted by $\pm M$ on the x position:

$$x_c = x + c \cdot M \quad \forall\, c \in \{-1, 0, 1\}.$$

Then, the component of the cost related to the energy harvesting from the sun is

$$\begin{aligned} g_{\text{solar}}((t,z,y,x),u) &= \min_{c\in\{-1,0,1\}} (x_c - x_{\text{sun}})^2 \\ &= \min_{c\in\{-1,0,1\}} ((x+cM) - x_{\text{sun}})^2. \end{aligned} \tag{4}$$

That is, the energy component of the cost depends exclusively on the x position of the balloon.

- *Connectivity.* The strength of the connection depends on the distance from the cities:

$$g_{\text{cities}}((t,z,y,x),u) = \sum_{i=0}^{i=N_{cities}-1} \left(\min_{c\in\{-1,0,1\}} \sqrt{(x+cM-x_{\text{city},i})^2 + (y-y_{\text{city},i})^2} + \lambda_{\text{level}} z \right),$$

where

  i) We account for the circular world by the factor cM, as we did in the *Energy* component of the cost;
  ii) N_{cities} (`Constants.N_CITIES`) is the number of cities; and
  iii) λ_{level} (`Constants.LAMBDA_LEVEL`) is a parameter that accounts for the discretization of the atmosphere.

The locations of the cities is stored in `Constants.CITIES_LOCATIONS`, and they are generated randomly:

```
# [(y,x),(y,x),...] coordinates of the cities
CITIES_LOCATIONS = np.stack((np.random.randint(0,N,N_CITIES),
                             np.random.randint(0,M,N_CITIES)), axis=1)
                   .tolist()
```

Thus, the location of the city `i` is `Constants.CITIES_LOCATIONS[i]`.

- The total stage cost is then the weighted sum of the two above:

$$g((t,z,y,x),u) = g_{\text{cities}}((t,z,y,x),u) + \lambda_{\text{timezone}} \cdot g_{\text{solar}}((t,z,y,x),u),$$

where $\lambda_{\text{timezone}}$ (`Constants.LAMBDA_TIMEZONE`) is a parameter that balances the influence of the two costs.

- Notice that the stage cost as we defined it so far depends only on the current state. However, not all inputs are allowed at every state (see Section **Input Space**). To account for this, the stage cost is modified as:

$$g((t,z,y,x),u) = \begin{cases} g_{\text{cities}}((t,z,y,x),u) \quad +\lambda_{\text{timezone}} \cdot g_{\text{solar}}((t,z,y,x),u) \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{if } u \in \mathcal{U}((t,z,y,x)) \\ +\infty \qquad\qquad\qquad\qquad\qquad\qquad \text{otherwise.} \end{cases} \tag{5}$$

In the code, if `G` is the $(TDMN) \times 3$ matrix of the stage costs (with the ordering as prescribed by Sections **State Space** and **Input Space**) and the input `input_space[a]` corresponding to the index `a` is not allowed at the state `state_space[i]` corresponding to the index `i`, write:

```
    G[i, a] = np.inf
```

If you follow our convention in the ordering and `Constants.V_UP` is not allowed at the state `state_space[i]`, you can write:

```
    G[i, Constants.V_UP] = np.inf
```

Your solution will be marked as incorrect if you do not adhere to this convention!

Discount Factor

There are multiple things that can go wrong when deploying a robot on a hostile environment like Mars. For instance, the balloon may crash in a Martian sandstorm, or the electronics may fail. Therefore, there is a chance that the balloon is no longer in operation after each time period. This is reflected by a discount factor α (`Constants.ALPHA`).

Tasks

Your tasks as a MASA engineer are the following.

- *Guided Solution.* The subtasks are (all are needed, but it helps in making progress to complete them separately):

  (a) Implement the function

  `compute_transition_probabilities` in `ComputeTransitionProbabilites.py`

  to return the transition probability matrix $\texttt{P} \in \mathbb{R}^{K \times K \times 3}$, where K is the number of possible states. To compute `P`, adhere to the ordering convention in Section **State Space** and Section **Input Space**:

  ```
  print(len(state_space))      # K
  print(len(input_space))      # L
  state_space[i]               # state with index i
  input_space[j]               # input with index j
  ```

  For instance, `P[i,j,Constants.V_STAY]` is the probability of transitioning from `state_space[i]` to `state_space[j]` when applying the input `Constants.V_STAY`.

  (b) Implement the function

  `compute_stage_cost` in `ComputeStageCosts.py`

  to return the stage cost matrix $G \in \mathbb{R}^{K \times L}$. The same indexing as the transition probability matrix must be used.

  (c) Implement the function

  `solution` in `Solver.py`

  to compute the optimal cost $\texttt{J} \in \mathbb{R}^{K}$ and the optimal policy $\texttt{policy} \in \mathbb{N}^{K}$. The same indexing as the transition probability matrix must be used. The optimal cost and input at the state `state_space[i]` are `J[i]` and `policy[i]`, respectively. If `Constants.V_STAY` is the optimal input for the state `state_space[i]`, you can write

  ```
  policy[i] = Constants.V_STAY
  ```

- *Freestyle Solution.* Implement the function:

  `freestyle_solution` in `Solver.py`

  You are free to re-use the functions you implemented for the *Guided Solution*, or follow another approach that does not adhere to the structure of the *Guided Solution*. Here the only constraint is that the peak memory usage must be below 250 MiB. The map we will test will have `Constants` in the ranges:

  $$
  \begin{aligned}
  0 &\leq \texttt{Constants.T} \leq 10 \\
  0 &\leq \texttt{Constants.D} \leq 4 \\
  0 &\leq \texttt{Constants.N} \leq 15 \\
  0 &\leq \texttt{Constants.M} \leq 20 \\
  0.95 &\leq \texttt{Constants.ALPHA} \leq 0.99 \\
  1 &\leq \texttt{Constants.N\_CITIES} \leq 5
  \end{aligned}
  $$

  All the other parameters may vary arbitrarily as long as they make sense mathematically and physically.

Your solution will be marked as incorrect if you do not adhere to the described conventions!

Evaluation and Scoring

1. You get 0 points if you are found to plagiarize from another team.
2. You get 0 points in a task in which you submit a solution that is not correct. When floating points comparisons are required to check correctness, we use `numpy.allclose(a, b, rtol=1e-04, atol=1e-07)`, where `a` and `b` are the values to compare.
3. You get 0 points in the *Freestyle Solution* task if you get a peak memory above 250 MiB. We compute the memory footprint of the *Freestyle Solution* as follows:

```
tracemalloc.start()

J_opt, u_opt = freestyle_solution(Constants)

current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()
print("Peak memory usage in MiB: {:.4}".format(peak / 2**20))
```

4. We rank the submitted solutions based on the average run time over 10 instances for the *Guided Solution* task. You get p_1 points based on your ranking as follows:

Ranking	Points
1	10
2	7
3	5
4	3
5	2
> 5	1

5. We rank the submitted solutions based on the average run time over 10 instances for the *Freestyle Solution* task. You get p_2 points based on your ranking as follows:

Ranking	Points
1	5
2	1
> 3	0

6. Your total score is $p = p_1 + p_2$. The final ranking is based on the score, where the higher the value of p the better your result.
7. For the top three submissions according to the score p, we will issue prizes including a tour of Verity with Prof. D'Andrea, signed certificates and gift vouchers of 100 CHF.

The judgement of the TA is final.

Python Files Provided

A set of Python files is provided on the class website. Use them to solve the above problem. Follow the structure strictly as the grading is automated for fairness reasons.

`Constants.py`	An instance of the Python class `Constants` will contain the constants used in the problem.
`ComputeTransitionProbabilities.py`	Contains `compute_transition_probabilities`, the Python function that has to be implemented to calculate the transition probability matrix `P`.
`ComputeStageCosts.py`	Contains `compute_stage_cost`, the Python function that has to be implemented to calculate the stage cost matrix `G`.
`Solver.py`	Contains `solution` and `freestyle_solution`, the Python functions that have to be implemented to solve the problem.
`main.py`	Python script that loads a test case (described by `Constants.py`), executes the implemented algorithms and generates the results to be visualized at a later time.
`test.py`	We provide three test cases that you can use to check your transition probability matrix, stage cost matrix and optimal cost by running `test.py`. You will not be evaluated on these test cases.
`viz_policy.py`	Python script to visualize interactively the policy and the value function (see Figure 1 and Figure 2). Run `main.py` before running this script. The visualization refers to the latest working output of `main.py`.
`viz_rollout.py`	Python script that creates a video of a rollout (obtained sampling the disturbances) of the latest policy. The video is saved in the folder `videos`. Run `main.py` before running this script. The visualization refers to the latest working output of `main.py`.

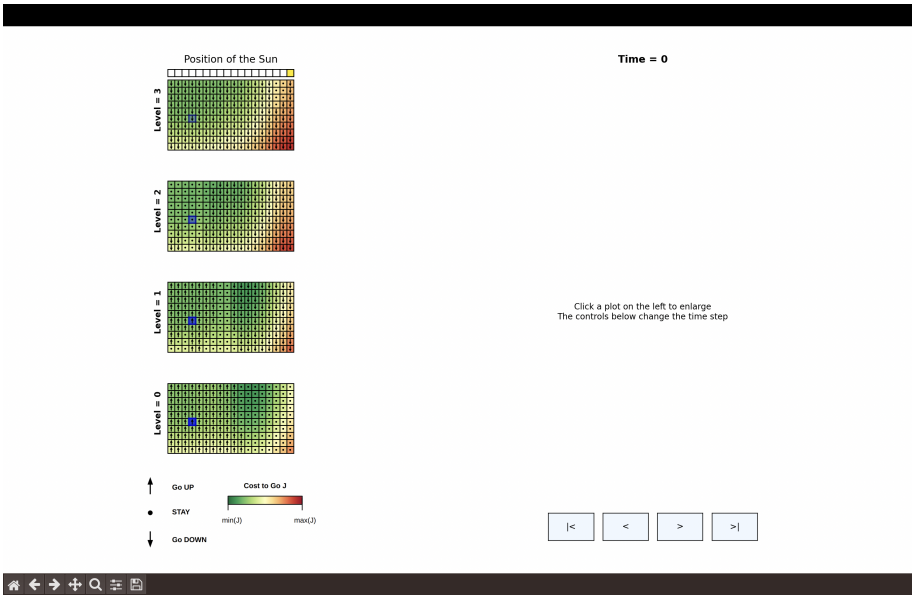


Figure 1: Visualization window that results from running `viz_policy.py`*. To visualize a level (enlarged) on the right, press the zoom or drag button and then the desired level on the left.*

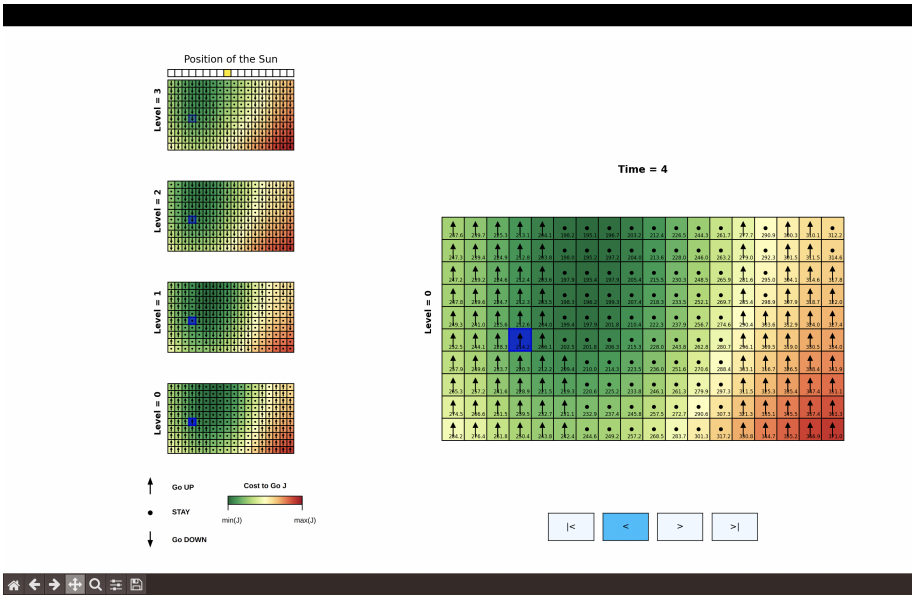


Figure 2: Pressing the arrow buttons you can visualize the time evolution throughout the day. Notice the position of the sun changing on the top left.

Deliverables

A maximum of two students can work as one team. Hand in one submission per team by e-mail to aterpin@ethz.ch by the due date (18^{th} of December, everywhere on Earth) with the subject `[programming exercise submission 2023]`, containing your Python implementation of the following files (only submit these files!):

- `ComputeTransitionProbabilites.py`
- `ComputeStageCosts.py`
- `Solver.py`

Make sure all your code is within the functions:

- `compute_transition_probabilities` (in `ComputeTransitionProbabilites.py`)
- `compute_stage_cost` (in `ComputeStageCosts.py`)
- `solution` (in `Solver.py`)
- `freestyle_solution` (in `Solver.py`)

If you need to declare additional functions or import additional packages, do so within the above functions. For instance,

```
    def compute_transition_probabilities(Constants):
        import itertools
        t = np.arange(0, Constants.T)
        z = np.arange(0, Constants.D)
        y = np.arange(0, Constants.N)
        x = np.arange(0, Constants.M)
        state_space = np.array(list(itertools.product(t, z, y, x)))
        ...
```

Include all files in one `zip`-archive, named `DPOCEx_Name1_Number1(_Name2_Number2).zip`, where `Name` is the full name of the student who worked on the solution and `Number` is the student identity number. (e.g `DPOCEx_JaneDoe_12345678_JohnDoe_12333678.zip`).

We will test your implementation with `Python 3.11.4` in a virtual environment created with the following instructions:

```
python3 -m venv .

source bin/activate

pip install -r requirements.txt
```

You can check your Python version with `python3 --version`. Make sure not to use packages that are not installed with the above list of commands.

We will send you a confirmation upon receiving your e-mail, but we will not check that your code is working and respects the required format. We will discard submissions that do not run in the above mentioned environment or do not fulfill the format required.

You are ultimately responsible that we receive your working solution in time.

Submission Checklist

- ☐ Your code runs with the original `main.py` script in a virtual environment created with the commands above and with `Python 3.11.4`.
- ☐ You did not modify the function signatures (name and arguments) in the submission files
  - `ComputeTransitionProbabilites.py`
  - `ComputeStageCosts.py`
  - `Solver.py`
- ☐ All your code is within the functions:
  - `compute_transition_probabilities`
  - `compute_stage_cost`
  - `solution`
  - `freestyle_solution`
- ☐ You only submit the files
  - `ComputeTransitionProbabilites.py`
  - `ComputeStageCosts.py`
  - `Solver.py`