Bachelor thesis

Artificial Intelligence

Radboud University

ABM Traffic

Author:
Joep Saris
s1094356

First supervisor:
MSc S. Chaturvedi
`Artificial Intelligence & Brain, Cognition and Behavior`
`siddharth.chaturvedi@donders.ru.nl`

Second reader:
Dr. Y. Qin
`Machine Learning & Neural Computing`
`yuzhen.qin@donders.ru.nl`

June 21, 2025

Abstract

This project is a proof of concept for a simple and scalable computational model for traffic movement, in which cars traverse a road from a random starting point to a destination, switching lanes to reach it. Navigation across roads occurs in a unique interplay between Cell and Car agents, operating in a vectorized way. This grid-world representation for traffic problems is an application of ABMax, an agent-based modeling framework using the Python library JAX, which is considered the state-of-the-art when it comes to parallel operations. Different traffic models from the past thirty years are explained and compared to each other, and to ABMTraffic. The model is the first to apply ABM to a traffic model evaluating in parallel. Though ABMTraffic is successful in proving the concept, there are a multitude of directions to take the project, such as development of new learning models to apply to the environment, optimization of the model and featuring expansion to increase models capabilities and statistical power.

Acknowledgements

- I appreciate the help of Thesis Coordinator Luc Selen, who has been available to guide me on two occasions during the intensive process of writing an alternate BSc Thesis project, a feat that requires realistic planning and the ability to replan. Talking to him was instrumental in helping me accept that I would have to make concessions and be critical of the project's timeline at all times.
- This project would not have been possible without the support of my primary supervisor Siddharth Chaturvedi. His hands-on approach to this project was invaluable and incredibly interesting! I had come into this project not knowing anything about JAX, and not having much experience with agent-modeling problems. Through our meetings, my progress was monitored and we discussed ways to move forward. What would the next problem to tackle and what tasks would come down the line? This process was good and oftentimes very fun.
- Additionally, I am thankful for having access to Siddharth's Git page for ABMax [1] as a resource, which introduced me to JIT, JAX, and ABMax. This made the steep learning curve associated with the libraries a bit more streamlined.
- Finally, I want to acknowledge the value that I got out of working closely with my friend Sybren van den Heuvel, who has also spent the past semester writing his thesis on an application for ABMax. We have spent quite some time explaining the material to each other and managing the process. When times got stressful I could always count on him, and I believe he could on me.

Contents

Chapter 1

Introduction

Traffic is an integral part of a thriving economy [2]. Transportation access is listed as one of the five pillars of economic development. If infrastructure is of such importance, it is a logical goal to optimize for traffic to maximize efficiency, which can be done through traffic modeling.
This thesis project is a model in which cars navigate across a road to find their destination. This model could then be used to simulate traffic flows and serves as a proof of concept that agent-based modeling can be used to model traffic movements in a vectorized way, breaking serial processing. By processing in parallel, we can open the door to evolutionary strategies that take advantage of these vectorized data structures [3]. This in turn allows for optimized learning, reducing cost in traffic modeling and thus simplifying the task of optimizing traffic networks in the real world.

1.1 Complex Systems

Traffic flow is a complex system. For a modeling system to be complex, it must involve numerous interacting agents whose aggregate behaviors are non-linear; hence, it cannot simply be derived from the summation of individual components' behavior [4]. Traffic jams are a non-linear emerging phenomenon in traffic models: they consist of interacting cars, and cannot be modeled by the sum of their parts, i.e. a singular car cannot form a traffic jam.
For modeling complex systems, agent-based modeling (later referred to as ABM) has been an extensively used tool. ABM is a computational modeling technique that represents individual agents in a system, along with the rules governing their behavior and interactions [5]. The advantage of ABM is that the agents can be as simple or as complex as the modeler chooses, and agents can represent anything from people and animals to companies or machines. Agents in ABM have three unique qualities. Firstly, they can make their own decisions, given their goals and perceptions of the world. This is called autonomy. Furthermore, different agents can have different parameters, a fact that leads to different behavior, facilitating diversity in the simulation. Finally, due to agents' interactivity: affecting each other and the environment, complex behavior emerges. In our case, that complex behavior is traffic jams.

1.2 Cellular Automata

In ABM, we refer to the number of states the environment can be in as the *state space*, and the *action space* is what we call the set of actions an agent may choose from [6]. A model's environment can either be *discrete* or *continuous.* In a *continuous* environment, both the state and action space are continuous and thus infinite. A *discrete* environment has a finite state and action space. When choosing between a discrete and a continuous environment, there's often a trade-off [7]. Discrete environments are more straightforward and precise, yet they are less realistic when modeling real life behavior, as in real-life, people do not move from cell to cell. The cellular automaton (CA) is a model with a discrete environment, in which the next state of a cell is dependent solely on a (fixed) certain rule applied to itself and the cells surrounding it: their *neighbourhood.* Rule 184 is a simple example of a CA [8]. The model works with a one-dimensional array of binary numbers and is used to model traffic on a single-lane road. The ruleset for Rule 184 can be seen below in Table 1.1. The model from this project is also a cellular automaton, though one with a much more complex set of rules.

Current Pattern	111	110	101	100	011	010	001	000
New State	1	0	1	1	1	0	0	0

1.3 Recent Development & Opportunity

In the past, not many agents could be simulated simultaneously in an ABM. However, with modern advancements in computational efficiency, that has changed. By making use of concurrent programming on hardware accelerators, this number could be vastly increased. JAX is a high-performance computing library for Python [9]. It offers this scalability in the set of agents by leveraging Just-in-Time (JIT) compilation, enabling simulations to run multiple orders of magnitude faster. Traditional ABM implementations often rely on slow, sequential processing, but JAX transforms Python code into optimized machine instructions, allowing for parallel execution on CPUs, GPUs, or TPUs. Key features like `vmap` automatically vectorize operations across agents, eliminating costly loops, while `pmap` distributes computations across multiple processors for even greater efficiency. By integrating JAX into ABM workflows, researchers can simulate considerably larger sets of agents in real-time, enabling deeper exploration of emergent behaviors and complex interactions. The approach removes computational bottlenecks, allowing for more extensive scenario testing and real-time experimentation. This thesis project makes use of Abmax, a library taking the exact approach that was just described. Abmax is an integration of JAX into ABM, focused on dynamic population size [1]. This library allows us to add and remove an arbitrary number of agents and search for, and sort these agents according to their traits. We can update a subset of agents to a specific state, and most notably, we can step multiple agents in parallel, in a vectorized way.

1.4 Section-wise Summary

In the **Background** (2) chapter of this thesis, notable models that have made progress in the subfield of traffic ABMs are briefly described, and it is explained how these models relate to ABMTraffic. In **Methodology** (3) we go over how the model works globally: What agents are at play, how they interact with the world, and what the order of operations is in each timestep of the simulation. The final chapter of this thesis is **Discussion & Conclusion** (4), which concludes from the results of the experiments, goes over the limitations and assumptions of this project, and describes its potential future application. The sources on which this project is based can be found in the **References** (5) section, and anything that was deemed too technical or extensive, but still supports my thesis, is referred to and listed in the **Appendix** (6).

Chapter 2

Background

This section provides context to ABMTraffic by showing the history of traffic models.

2.1 Cellular Automata approach

The NaSch model is the first notable traffic model to cover. It was released in 1992 after being constructed by German mathematician and physicist Kai Nagel and Michael Schreckenberg [10]. This model was based on Cellular Automata and stands out because it showed that traffic jams can emerge without being the result of an accident. This model can be seen as simple, as it is limited in realism: there is no overtaking between lanes, and cars only act by changing their speed [11]. Simple models are useful to impart underlying knowledge, rather than quantitative accuracy. Using their simulation however, the researchers compared traffic flow (speed of cars per timestep) between different road densities (number of cars per tile), and found a similar relation to the real-world data. The researchers concluded that opting for the discrete CA approach was both computationally advantageous to the continuous approach and instrumental to capturing the dynamic nature of traffic flow. This model does not use agents and is not optimized to work in a vectorized way.

2.2 Agent-Based Modeling approach

The NaSch model opened the door for new and more advanced traffic simulators. In 2009, a model was created by Bosnian engineer Vedran Ljubović that applied agents to the environment [12]. The researcher was critical of what features were relevant and what assumptions he made during the process of creating the model. With the release of this paper, a number of advancements have been made in modeling traffic. Just like the 1992 model, Ljubović's model works in a discrete environment and also captures car congestion (traffic jams). However, unlike in the NaSch model, multiple lanes are introduced, between which cars can switch, and cars that are spawned at entrances of the environment have their destination to exit the simulation. Finally, traffic rules have been formalized for agents so that they are aware of when lane change, turning, signalization, and parking is possible. Ljubović's model operates in a serialized order, and he mentioned the opportunities of making a traffic ABM that works in parallel. He states that the speed of the simulation would increase, the door would open for additional features and it would allow for a larger simulation size, which was maximally an area of 2x2 km with the original model. Ljubović thus saw the potential value of having the model function in a vectorized way.

2.3 Multi-GPU Parallel Simulation Framework

At the end of 2024, another significant contribution was made to the field of traffic simulation: LPSim, a Large Scale Multi-GPU Parallel Computing-based framework designed to scale microscopic traffic models using GPU architecture [13]. This framework is not an ABM. Cars in the simulation are objects with a current position, speed, and a planned route. Unlike in ABMs, Car objects in LPSim have no beliefs or knowledge. Their movements and speed changes are decided by mathematical functions and are not based on decision-making. LPSim was created to outperform the traffic models that leveraged CPU or a single GPU, a goal that was achieved. LPSim is capable of running nine million vehicle simulations in parallel in under 22 minutes. LPSim combines single-instruction-multiple-data processing with dynamic graph partitioning to distribute simulation tasks across multiple GPUs. It works as follows: Each GPU is assigned a subregion of the traffic network, and vehicles transitioning between these regions are synchronized through ghost zones ('replicated buffer areas that possess the same information across the boundaries of adjacent GPUs' [13]) ensuring consistency within simulations.
LPSim offers a vectorized and scalable simulation platform, with a more complex environment than any of the other models mentioned: there is variation in vehicles, the road includes intersections, and roads can be manipulated during simulation. Altogether, the framework leaps the field of modeling traffic forward in a major way [13].

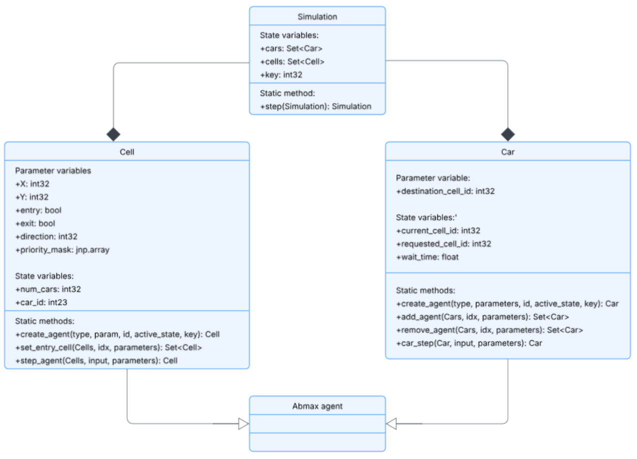
Chapter 3

Methodology

In this section, the model presented by this project is explained. Note that while the major functions and behaviors of the project are explained, this is not an exhaustive description. The complete overview of the code of this model can be read in the appendix, section 6 of this thesis. In this section, the default values for parameters is also given.

3.1 Agents

This model has two interacting agent types: car agents and cell agents. The interplay of these agents is managed in a composite agent type called simulation. Through a simulation, car agents can get from their start cell agents to their destination. The UML for these agents can be seen below in figure 3.1. An agent is defined as an autonomous computational object that interacts with others and the environment over time and space [14]. Next to their ids and transition functions, agents have states and parameters. The difference between states and parameters lies in their function: states are dynamic. They change over time, while parameters are static and govern how the agent changes over time. ABMs use these elements to simulate complex, emergent outcomes from individual-level interactions. When explaining the agent types, it is important to describe what they represent, their state, and parameter variables. The functions within these agents are explained in the next section, functionality.


Simulation
State variables:
+cars: Set<Car>
+cells: Set<Cell>
+key: int32
Static method:
+step(Simulation): Simulation
Cell
Parameter variables
+X: int32
+Y: int32
+entry: bool
+exit: bool
+direction: int32
+priority_mask: jnp.array
State variables:
+num_cars: int32
+car_id: int23
Static methods:
+create_agent(type, param, id, active_state, key): Cell
+set_entry_cell(Cells, idx, parameters): Set<Cell>
+step_agent(Cells, input, parameters): Cell
Car
Parameter variable:
+destination_cell_id: int32
State variables:'
+current_cell_id: int32
+requested_cell_id: int32
+wait_time: float
Static methods:
+create_agent(type, parameters, id, active_state, key): Car
+add_agent(Cars, idx, parameters): Set<Car>
+remove_agent(Cars, idx, parameters): Set<Car>
+car_step(Car, input, parameters): Car
Abmax agent

3.1.1 Cell agent

Cell agents are part of the road environment, and they can hold cars. Both state variables included in cell agents are about car agents: `num_cars` and `car_id`. The first one describes the number of cars in the cell agent. In the current setup this number is either zero or one, but cell agents can be modified to house more than one car agent. `car_id` is the id of the car agent that is currently in the cell, if any. If there is no car in the cell, this value will be set to -1. The parameters of cell agents describe what makes them unique from other cell agents. There are six parameters for cell agents, the first two of which describe the coordinates concerning the larger road structure: `X` and `Y`. The road is of size $(n \times c)$, with n being the number of rows the road has (the number of cell agents that have to be traveled across from entry to exit Cell), and c as the number of columns, or most often referred to as the number of lanes the road has. At the bottom left of the road (relative to the road's `direction`) the coordinates `X` and `Y` are at the origin: (0, 0). Following that, the other coordinates logically follow. If the road is vertical, the `X` increases across lanes and `Y` across rows, if it is horizontal, the dimensions are swapped so it is the other way around. Of the other four parameters of the cell agents, two inform the global function and two help with accepting and navigating car agents. The parameters `entry` and `exit` allow the cell agents to respectively be a start or end point of the simulation. The two final parameters are `direction` and `priority_mask`. `direction` informs car agent in the cell as to what way it should look ahead when seeking the next `requested_cell_id` to go to (explained in Car agent section). `priority_mask` is a vector of size eight ascribing a value for each of the cell agents surrounding it. This mask is used when determining which car agents are favored over others when there are multiple car agents interested in coming toward the Cell. At least five of the eight values in the `priority_mask` are zeroes, as at most only three cell agents have a legal move toward each other cell agent. The `priority_mask` differs based on lane. If the cell agent is in the leftmost lane, car agents can not come from the bottom left, as there are no cells there, and thus it cannot take car agents from there. Similarly, in the rightmost lane, there cannot be car agents accepted from the bottom right. For these reasons the `priority_mask` includes a zero value for the indices pointing to out of bounds surrounding cell agents. The middle lane cell agents have a positive value for all three cell agents beneath it. The cell agents have a natural priority for car agents going directly in the cell agents `direction`, this spot in the array is attributed value 3. The second priority goes toward car agents that switch lanes from the left side, as the further left, the more speedy the lane is in real-world driving behavior. The bottom left cell agent gets value 2 and the bottom right cell value 1.

3.1.2 Car agent

The car agent moves across the environment, a $(n \times c)$ structure of cell agents forming a road. The state of the car agent consists of the following variables: `current_cell_id`, `requested_cell_id`, and `wait_time`. The `current_cell_id` variable points to what cell the car agent is currently in. In each timestep of the simulation, the car agent plans to what cell it wants to go to next. This `cell.id` is stored in the `requested_cell_id` variable. Finally, there's `wait_time`, which represents the amount of time a Car has been waiting in a single Cell. Once this value gets across the patience threshold, the car agent will plan for a new `requested_cell_id`, as the current cell the car agent has requested to go to does not seem interested in having it come there. Car agents have only a single parameter `destination_cell_id`, the id of the cell agent on the road toward which the car agent is navigating during the simulation. Unlike the cell agent that is always active, a car agent can be either active or inactive. Because of this, in addition to the create and step function, car agents have functions to get activated and deactivated.

Symbol	Meaning
C	The set of car agents
S	The set of cell agents
P	Params object used by Abmax functions
I	Input object used by Abmax functions
k	Random key
k_{split}	Sub-key
n	Length of the road
c	Number of lanes and thus entry and exit cell agents

3.1.3 Simulation agent

As stated before, simulation agents represent the interplay between car and cell agents. A simulation agent has no parameters, and it has state variables `cars`, `cells`, and `key`. `Cars` and `cells` are sets of car and cell agents, and `key` is a random key. When a simulation agent gets created, the dimensions ($n \times c$) of the road `road_shape`, the maximal number of car agents `num_cars` to be active and the road's `direction` need to be specified.

3.2 Functionality

The functionality of the model's simulation can be divided into five sub-routines. Together, these sub-routines make up the step function for the simulation agent. Each of the subroutines and finally, the simulation step function have been explained individually, using both a short description and a pseudocode section. Some variables used in pseudocode are not clearly understandable, these can be found in 3.1.

3.2.1 Subroutines

Spawn car agents in start cells

Algorithm 1 Spawn cars in entry cells

```
1: function SPAWN($C, S, C_{active\_count}, k, c$)
2:     $S_{entry}, S_{exit} \leftarrow$ find_entries_exits($S$)
3:
4:     $C_{ids} \leftarrow array([0, ..., c-1]) + C_{active\_count}$
5:     $S_{exit} \leftarrow$ random_sample($c, S_{exit}, k_{split}$)
6:
7:     $S_{entry} \leftarrow$ shuffle($S_{entry}, k_{split}$)
8:     $S_{available} \leftarrow S_{entry}['num\_cars'] == 0$
9:     $S_{entry} \leftarrow$ argsort($S_{available}$)
10:
11:    $n_{S\_free} \leftarrow$ sum($S\_available$)
12:    $n_{added\_cars} \leftarrow$ random($1, n_{S\_free} + 1$)
13:
14:    $P_{car\_add} \leftarrow Params(S_{entry}, S_{exit}, C_{active\_count})$
15:    $P_{cell\_set} \leftarrow Params(S_{entry}, C_{ids})$
16:    return $P_{car\_add}, n_{added\_cars}, P_{cell\_set}$
17: end function
```

In the first part of the simulation step, cars get added to the environment. This function creates parameters for each entry cell (c). The entry cell ids get shuffled first, and then they get sorted according to availability such that the empty cells occur before the occupied ones. The destination ids get sampled with replacement. Each call, between 1 and all of the empty entry cells get filled with a car agent. Thus, there are almost always more parameters generated than car agents that end up getting added. This approach is necessary if both randomization and parallelism are part of the environment, and it takes advantage of how JIT works internally, as the loops only go to the number of car agents we want to add.

Check which car agents moved

For a car agent to move from one cell agent to the next, it needs to request the next cell agent and only when the cell agent accepts it, does it move ahead. In this subroutine, the global functionality vectorizes over all car agents to compare their id to the `car_id` of the cell agent given by the car agent's `requested_cell_id`. If the car agent was chosen, these numbers are equal. The resulting binary vector from this `vmap` is used as a parameter for the next sub-routine `car.step_agent`.

Algorithm 2 Car chosen

```
1: function CAR_CHOSEN($C$, $S$)
2:     function SINGLE_CAR_CHOSEN($car$, $S$)
3:         $requested\_id \leftarrow car.state.content['requested\_cell\_id']$
4:         $chosen\_id \leftarrow car_{single}.state.content['car\_id'][requested\_id][0]$
5:         if $requested\_id[0] \geq 0$ then
6:             $chosen \leftarrow car.id == chosen\_id$
7:         else
8:             $chosen \leftarrow$ False
9:         end if
10:        return $chosen$
11:    end function
12:    $car\_chosen \leftarrow$ jax.vmap(single_car_chosen, in_axes=(0, None))(C, S)
13:    return $car\_chosen$
14: end function
```

Step car agents

In the car step function input variable `chosen` informs a car agent whether or not it has been selected to move ahead. When the car is chosen, the time it has been waiting to move will be reset and its `current_cell_id` is set to its past request id. When it was not chosen, the car's wait time gets incremented by one unit of parameter `dt`, and the `current_cell_id` remains the same. After potentially moving, a car agent gets the opportunity to choose a new `requested_cell_id`. This is only done if the car agent just moved, or if it was rejected for five times. This next move is stochastic, and dependent on the cell options in the direction the car agent is driving in and its distance to the destination. To ensure every car agent reaches its destination, lane switching options get removed from the pool as the car gets closer to its destination. When the process of choosing a new request is complete, the new state variables get made into a State object and the agent gets replaced with this new state.

Find and remove finished car agents

To make sure all car agents are actively moving toward their destination, the simulation removes cars that reached the destination right after the car agents have stepped. As Abmax offers

Algorithm 3 Stepping car agents

```
function CAR.STEP_AGENT($P_{car\_step} = (S, n, c, dt), I_{car\_step} = (chosen)$)
    if $I_{car\_step}['chosen']$ then
        $new\_wait\_time \leftarrow 0$
        $new\_current\_cell\_id \leftarrow car.state['requested\_cell\_id']$
    else
        $new\_wait\_time \leftarrow car.state['wait\_time'] + P['dt']$
        $new\_current\_cell\_id \leftarrow car.state['current\_cell\_id']$
    end if
    if $0 < new\_wait\_time \leq 5$ then
        $new\_requested\_cell\_id \leftarrow car.state['requested\_cell\_id']$
    else
        $new\_requested\_cell\_id \leftarrow$ next_move($car.new\_current\_cell\_id, car.params['destination\_cell$
    end if
    $new\_state \leftarrow State(new\_current\_cell\_id, new\_wait\_time, requested\_cell\_id)$
    return $car$.replace($new\_state$)
end function
```

functions for selecting and removing agents, the only necessary code for this sub-routine is providing a function that returns `True` for each car agent at the destination, and `False` for all others [1]. To make that happen, the simulation vectorizes across all car agents and compares the state variable `current_cell_id` to parameter `destination` separately for each car agent.

Algorithm 4 Selecting car agents at destination

```
function SELECT_FINISHED_CARS($C$)
    function SELECT_CAR($car$)
        $current\_cell\_id \leftarrow car.state.state.content['current\_cell\_id']$
        $destination\_id \leftarrow car.params.content['destination\_cell\_id']$
        if $car.active\_state$ && $current\_cell\_id == destination\_id$ then
            $car\_arrived \leftarrow$ True
        else
            $car\_arrived \leftarrow$ False
        end if
        return $car\_arrived$
    end function
    $cars\_arrived \leftarrow$ jax.vmap(car_arrived, in_axes=(0))(C)
    return $cars\_arrived$
end function
```

Step cell agents

Stepping cell agents happens in two phases. The first phase is the donor phase, in which the cell checks whether the car it believes to have is still there. If it is not, the cell agent detaches the car from itself, resetting `car_id` and `num_cars`; else, nothing happens. In recipient, the second phase, the cell looks for car agents that want to enter it. The cell checks all its surroundings to get a vector of the maximally eight cars that could potentially pick it as the `requested_cell_id`. Given car agents' id, direction of entrance, and `wait_time`, a preferred car gets chosen. When the cell agent has an empty slot, and at least one car agent applied to go there, it will update the `car_id` and `num_cars` to the new information. The agent gets replaced with an agent with the new state information for `car_id` and `num_cars` and the function is stopped.

Algorithm 5 Stepping cell agents

```
function CELL.STEP_AGENT($P_{cell\_step} = (C, S, n, c), I_{cell\_step} = (\emptyset)$)
    $car_id \leftarrow cell.state['car\_id']$
    $my\_car \leftarrow P_{cell\_step}['C'][car\_id]$
    $cars\_cell\_id \leftarrow my\_car.state['current\_cell\_id']$
    if $cell.id == cars\_cell\_id$ then
        $new\_num\_cars \leftarrow cell.state['num\_cars']$
        $new\_car\_id \leftarrow cell.state['car\_id']$
    else
        $new\_num\_cars \leftarrow 0$
        $new\_car\_id \leftarrow -1$
    end if

    $surrounding\_relative\_coordinates \leftarrow array([[[-1], [-1]], [[0], [-1]], [[1], [-1]], [[1], [0]], [[1], [1]], [[0], [1]],$
    $surrounding\_absolute \leftarrow array([cell.params.content['X'], cell.params.content['Y']]) + surrounding\_relative\_coordinates$
    $surrounding\_cell\_ids \leftarrow$ jax.vmap($jit\_coords\_to\_id, in\_axes = (0, 0, None, None))(surrounding\_absolute[0], surrounding\_absolute[1], P_{cell\_step}[n], P_{cell\_step}[c])$

    $surrounding\_car\_ids \leftarrow$ jax.vmap($get\_car\_ids, in\_axes = (0))(surrounding\_cell\_ids)$
    $car\_options, wait\_times \leftarrow$ jax.vmap($get\_request\_match\_wait\_time, in\_axes = (0, None))(surrounding\_car\_ids, cell.id)$

    $preference\_matrix \leftarrow car\_options \cdot wait\_times \cdot cell.params['priority\_mask']$
    $chosen\_index \leftarrow$ argmax($preference\_matrix$)
    $chosen\_car\_id \leftarrow car\_options[chosen\_index]$

    if $new\_num\_cars == 0$ && $chosen\_car\_id \geq 0$ then
        $new\_num\_cars \leftarrow 1$
        $new\_car\_id \leftarrow chosen\_car\_id$
    end if
    $new\_state \leftarrow State(new\_num\_cars, new\_car\_id)$
    return $cell$.replace($new\_state$)
end function
```

3.2.2 Simulation step

With all five steps of the sub-routine written, the simulation step becomes quite easy. Once an environment has been created, the step function can be ran to develop the environment by a timestep. First, a set of between 0 and c car agents get added to the entry cells. Then, `car_chosen` informs car agents in `car.step_agent` whether or not they are allowed to move to their `requested_cell_id` and possibly generate the new one. After this, the car agents that reached their destination get taken out of the simulation, freeing up space to add new agents in the next timestep. Finally, the cell set takes in their surroundings to find and choose the best car candidate to come to them in case they will be free in the next timestep.

Algorithm 6 Simulation step

```
1:  function STEP($C, S, k$)
2:
3:      $P_{car\_add}, n_{added\_cars}, P_{cell\_set} \leftarrow$ spawn($C, S, C_{active\_count}, k_{split}$)
4:      $C \leftarrow$ add_cars($C, n_{added\_cars}, P_{car\_add}$)
5:      $S \leftarrow$ set_cells($S, n_{added\_cars}, P_{cell\_set}$)
6:
7:      $chosen \leftarrow$ car_chosen($C, S$)
8:
9:      $C \leftarrow$ jit.step_agents($car.step\_agent(), C, P_{car\_step} = (S, n, c, dt), I_{car\_step} = (chosen)$)
10:
11:     $C_{selected} \leftarrow$ select_agents($select\_finished\_cars, C$)
12:     $C \leftarrow$ remove_agents($car.remove\_agent, C_{selected}$)
13:
14:     $S \leftarrow$ jit.step_agents($cell.step\_agent(), S, P_{car\_step} = (C, S, n, c), I_{cell\_step} = (\emptyset)$)
15:     return $C, S, k$
16: end function
```

After the cell agents have been stepped, the step is complete and the random key, cell, and car agents get returned to be used in the next call of the simulation step. Additionally, the `Car.current_cell_id` vector gets used, such that the state of the road can be visualized externally. It is not possible to represent the road during the step in parallel, as the variable values are not concretized mid-step [9].

Chapter 4

Discussion & Conclusion

4.1 Limitations

By creating any simple model, one makes many assumptions. This project is no exception. Because it is a discrete model, the assumption gets made that both time and space are discrete: Cars move ahead by a discrete unit (one Cell) at a time and can do so once each timestep (also a discrete unit)[7]. As a consequence of using a discrete environment, the state space is finite, given by the set of Cells. By not including any memory for the agents, the Markov assumption gets made, stating that the future is independent of the past given the present [15]. Other, more explicit assumptions that are made mostly to keep the model simple include that Car agents are not interested in other Cars in their surroundings when planning a lane change. Additionally, the assumptions that Cells can hold at most one Car maximally and that Cars can, given no circumstances, miss their destination. All of the mentioned assumptions would not follow from real life and must be carefully taken into consideration when making comparisons between the two.

4.2 Conclusion and Future applications of this model

The field of Traffic modeling has come a long way since the release of the NaSch model, the first model capturing the complex behavior of traffic jams [10]. An ABM evaluating in serial, and a non-ABM application that worked in a vectorized way leveraging multiple GPUs have been mentioned [12][13]. It has been shown that a traffic ABM application that evaluates in parallel is possible, though it has not been explored enough. There are a multitude of directions in which this project and the field of traffic modeling can be taken. One can divide these applications into the categories of optimization, feature expansion, and learning application.

4.2.1 Model Optimization

First off, the efficiency of this model can be improved upon, as the project has situations where information is gathered at every timestep, which stays static over time, so computational optimization can still be done to reduce the time taken to simulate a road structure, something that will always be appreciated, especially as the ABM gets more complex. Eventually, this model is aimed to be used for real-world problems, which also means it would not be used by programming experts down the line. To prepare for that it would make sense to give the model an easy-to-understand user interface in which a user passes along the significant parameters, and the desired simulation would be started.

4.2.2 Feature expansion

Secondly, many more features can be added to make simulations more interesting. Compared to the parallel ABM introduced by this project, the 2009 model by Ljubović is complex, as it works with networks of roads, variations in speed, and its agents that plan their route from their current location to their destination, something the Car agents do not do this model [12]. Now that models can be made in a vectorized way using the state-of-the-art library JAX, it is time to add more features to make the simulation more interesting [9]. In this process, it is important to find the distinction between what is possible to create in a vectorized way and what is not, as working in parallel with these libraries adds unique constraints and reduces the information an agent has about other agents; When every agent is acting in serial, an agent can go off of what other agents have done in this timestep, whereas in this framework, the agent knows only where agents were before and what they are planning to do. These features can be divided into two categories: features that alter the environment: the road structure of cells, and features that alter the car agents that move across the environment.

Adding features to the environment

Having a more interesting road structure is the most important next step in this project in my opinion. Making it possible to create intersections that have multiple roads feeding into each other. This project already supports roads in all directions, so only the connecting junctions between these already introduced road sections have to be made. After adding this, the environment can be scaled to have the simulation capture a whole city block area. Having variations between these junctions such as roundabouts and intersections with and without traffic lights would be incredibly interesting, as you could compare the different ways to set up infrastructure and find the best layouts for a given situation. Another idea would be making the road structure dynamic, allowing the simulation to open and close lanes depending on need. This would help quantify the extent to which a problem is solved by adding more lanes and could factor into the learning model to differentiate between how many lanes are needed between different segments of the area.

Adding variation to cars

The reasons for increasing the characteristics car agents can have are twofold. For one, the simulation becomes more realistic to real-world applications. If the simulation ended up being used for real-world scenarios, yet some instrumental variable was not implemented, the proposed optimal solution could fail massively when applied in real life. An example would be applying a traffic model that samples no trucks to find the optimal infrastructure solution for an area that in reality is frequently used as a freight route. For that reason, adding more parameters to car agents such as maximal speed, and length is imperative. The second reason for increasing car agents' characteristics, is that it opens the door for learning more about driving behavior. A suggestion in this category is adding patience as a state variable to the Car agent. When this variable increases, a car navigates the road more aggressively. Modeling this might have psychological applications as the model is used to learn the best driving strategies for both the individual and the group, a learning model that - one could hypothesize - would work against itself. Another suggestion concerning driving behavior is adding emergency cars, which traverse the network at high speeds, and other cars have to clear the way. It would be interesting to see what behavior would be learned from this addition.

4.2.3 Learning Models

There is no learning model in place for this thesis project, though there are libraries for JAX that enable this. EvoJAX is one of such libraries [3]. It allows agents to evolve to find the best behaviors. By representing simulation as an agent and because this entire model can be vectorized, EvoJAX could be implemented with the model as it currently stands. While learning can occur even in such a small road environment, many more interesting experiments can take place when it is scaled up with some of the feature suggestions mentioned.

One might see that as you develop all of these things, useful knowledge can be accumulated for real world application. An example is adding traffic lights as an agent using a learning model to find the optimal traffic light patterns for an already constructed traffic problem given an estimated number of cars that would drive through the network, and from what directions they enter and exit. A larger scale learning model would get the same information of estimated Cars, though it would not have a constructed road to work with. The model would be tasked with finding the optimal solution: determining the number of lanes each road would require and what junction types would be best. A model like that seems achievable but is still quite a ways away, as many features need to be added, and running the model would be very computationally expensive, even if it was optimized to its max.

Using an agent-based model for traffic and deploying learning models could help us learn the optimal solutions for specific infrastructure problems without having to build a single road until the users are satisfied and decide to make it a reality. Incorporating these models into the process of expanding and optimizing infrastructure would mean saving expenses and man-hours on the planning of such a solution and on the reconstruction of infrastructure that did not end up working as intended. Perhaps most importantly, drivers' time would be saved, optimizing transportation access, and contributing to a thriving economy [2].

Chapter 5

References

[1] I-M-Iron-Man, “GitHub - i-m-iron-man/abmax: Abmax is an agent-based modelling framework in Jax, focused on dynamic population size.” https://github.com/i-m-iron-man/abmax/tree/master/abmax.

[2] D. O’Hara, “The five pillars of economic development,” 5 2018. https://docs.udc.edu/causes/Five-Pillars-DC-Final-05-2018.pdf.

[3] Y. Tang, Y. Tian, and D. Ha, “EvoJAX,” *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pp. 308–311, 7 2022. https://arxiv.org/abs/2202.05008.

[4] L. M. Rocha, “Complex systems modeling,” 8 2003. https://casci.binghamton.edu/publications/complex/csm.html.

[5] M. James, “Agent-Based Modeling: How to Simulate the Behavior of Large Systems,” *Simultech*, 10 2024. https://www.simultech.org/agent-based-modeling-how-to-simulate-the-behavior-of-large-systems/.

[6] N. W. Staff, “Discrete vs. continuous models: Choosing frameworks for dynamic systems - new york weekly,” 4 2025. https://casci.binghamton.edu/publications/complex/csm.html.

[7] A. S. Tripathi, “Types of environment in ai,” 5 2023. https://www.scaler.com/topics/artificial-intelligence-tutorial/types-of-environment-in-ai/.

[8] W. Contributors, “Rule 184,” 5 2024. https://en.wikipedia.org/wiki/Rule_184.

[9] J. Frostig and Leary, “Compiling machine learning programs via high-level tracing,” *Standford Computer Science*, 2018. https://cs.stanford.edu/~rfrostig/pubs/jax-mlsys2018.pdf.

[10] K. Nagel and M. Schreckenberg, “A cellular automaton model for freeway traffic,” *Journal de Physique I*, vol. 2, pp. 2221–2229, 12 1992. https://doi.org/10.1051/jp1:1992277.

[11] M. Channel, “Mark Newman - The Physics of Complex Systems - 02/10/18,” 2 2018. Relevant segment at 9:30-15:08. https://www.youtube.com/watch?v=2L64AhoKamE.

[12] V. Ljubovic, “Traffic simulation using agent-based models,” *Institute of Electrical and Electronics Engineers*, pp. 1–6, 10 2009. https://doi.org/10.1109/icat.2009.5348417.

[13] X. Jiang, R. Sengupta, J. Demmel, and S. Williams, “Large scale multi-GPU based parallel traffic simulation for accelerated traffic assignment and propagation,” *Transporta-*

tion Research Part C Emerging Technologies, vol. 169, p. 104873, 10 2024. https://doi.org/10.1016/j.trc.2024.104873.

[14] S. De Marchi and S. E. Page, “Agent-Based models,” *Annual Review of Political Science*, vol. 17, pp. 1–20, 3 2014. https://doi.org/10.1146/annurev-polisci-080812-191558.

[15] J. Kwisthout, “Bayesian Networks 4,” 11 2023.

Chapter 6

Appendix

6.1 Parameters

Parameter	Description	Default value
n	Length of road	7
c	Number of lanes in road	3
direction	The direction in which cell agents allow car agents to move: 0 = up, 1 = left, 2 = down, 3 = right	0
num_cars	Maximum number of cars in simulation	10
dt	Unit by which `wait_time` increases at every timestep when car agent stays in a cell	1.0

6.2 GitHub page

ABM Traffic GitHub

6.3 Complete Model Code

[1]:
```
from abmax.structs import *
from abmax.functions import *
import jax.numpy as jnp
import jax.random as random
import jax
from flax import struct
```

6.3.1 Agent classes

[2]:
```
'''
Helper functions to
    - convert between XY coordinates and cell IDs so that we dont have to
      pass the entire grid around when we actually only need to pass the cell␣
 ↪ID.

```

```
    - represent the road grid so that you don't have to go
      into the arrays in order to think about the state of the road.
'''

def XY_to_cell_id(X:jnp.array, Y:jnp.array, X_max:jnp.array, Y_max:jnp.array,␣
↪moves_vertical: jnp.array):
    """
    Convert XY coordinates to cell ID. The cell ID is a unique identifier for␣
↪each cell in the grid.
    The cell ID is calculated as cell_id = X + Y * X_max, where X and Y are␣
↪the coordinates of the cell
    in the grid, and X_max is the maximum value of X in the grid. The cell ID␣
↪is -1 if the coordinates are out of bounds.
    The function also checks if the coordinates are within the bounds of the␣
↪grid, and returns -1 if they are not.
    Args:
        X: The X coordinate of the cell. jnp.array
        Y: The Y coordinate of the cell. jnp.array
        X_max: The maximum value of X in the grid. jnp.array
        Y_max: The maximum value of Y in the grid. jnp.array
    Returns:
        cell_id: The cell ID of the cell. jnp.array
    """
    X_cond = jnp.logical_and(X[0] < X_max[0], X[0] >= 0)
    Y_cond = jnp.logical_and(Y[0] < Y_max[0], Y[0] >= 0)

    def horizontal_layout(_):
        return X + Y * X_max[0]

    def vertical_layout(_):
        return Y + X * Y_max[0]

    cell_id = jax.lax.cond(jnp.logical_and(X_cond, Y_cond),
                           lambda _: jax.lax.cond(moves_vertical[0],
                                                  horizontal_layout,
                                                  vertical_layout,
                                                  None),
                           lambda _: jnp.array([-1]),
                           None)
    return cell_id
jit_XY_to_cell_id = jax.jit(XY_to_cell_id)

def cell_id_to_XY(cell_id: int, X_max: int, Y_max: int, moves_vertical: jnp.
↪array):
    """
    Convert cell ID to XY coordinates. The cell ID is a unique identifier for␣
↪each cell in the grid.
    The XY coordinates are calculated as X = cell_id % X_max and Y = cell_id /
↪/ X_max, where cell_id is the ID of the cell,
    and X_max is the maximum value of X in the grid. The function also checks␣
↪if the cell ID is within the bounds of the grid,
    and returns (jnp.array([-1]), jnp.array([-1])) if it is not.
    Args:
        cell_id: The cell ID of the cell. jnp.array
        X_max: The maximum value of X in the grid. jnp.array
        Y_max: The maximum value of Y in the grid. jnp.array
    Returns:
        XY: The XY coordinates of the cell. jnp.array
    """
    valid_id = jnp.logical_and(cell_id[0] >= 0, cell_id[0] < jnp.
↪multiply(X_max[0], Y_max[0]))
    def horizontal_layout(_):
        return jnp.mod(cell_id, X_max[0]), jnp.floor_divide(cell_id, X_max[0])

    def vertical_layout(_):
        return jnp.floor_divide(cell_id, Y_max[0]), jnp.mod(cell_id, Y_max[0])

    XY = jax.lax.cond(valid_id,
                      lambda _: jax.lax.cond(moves_vertical[0],
                                             horizontal_layout,
                                             vertical_layout,
                                             operand=None),
                      lambda _: (jnp.array([-1]), jnp.array([-1])),
                      operand=None
    )
    return XY
jit_cell_id_to_XY = jax.jit(cell_id_to_XY)

def print_car_positions_sequence(car_positions_over_time: jnp.array, X: int,␣
↪Y: int, moves_vertical: int):
    """
    Visual representation of cars on the road. Made for printing the entire␣
↪sequence at once (necesarry when using the lax scan function, because this␣
↪function is not jittable).

    args:
        - car_cell_ids: jnp.array, a 1D array of length num_cars, each index␣
↪is a Car id and the value in this array the Cell index or -1 for inactive␣
↪cars.
        - X, Y: int, grid dimensions.
        - moves_vertical: int, True if grid is row-major (X moves faster),␣
↪False for column-major (Y moves faster)
    """
    T = car_positions_over_time.shape[0]

    for t in range(T):
        print(f"Road at timestep t={t}")
        grid = [" ."] * (X * Y)
        for car_id, cell_id in enumerate(car_positions_over_time[t]):
            if cell_id != -1:
                grid[int(cell_id)] = f"{car_id:2}"

        for row in reversed(range(Y)):
            row_cells = []
            for col in range(X):
                if moves_vertical:
                    index = row * X + col   # row-major layout
                else:
                    index = col * Y + row   # column-major layout
                row_cells.append(grid[index])
            print("\t" + " ".join(row_cells))
```

[3]:
```
@struct.dataclass
class Car(Agent):
    '''
    A Car Agent moves across Cell structures from their start until their␣
↪destination.

    Variables
        - State
            current_cell_id -> The id of the Cell the Car is currently in.
            requested_cell_id -> The id of the Cell to which the Car wants to␣
↪go in the next timesteps.
            wait_time -> A variable keeping track of how many timesteps the␣
↪Car has been stuck in a Cell for. This is used for prioritizing cars who've␣
↪been waiting for longer.
        - Params
            destination_cell_id -> The id of the Cell the Car needs to go to.

    Functions
        - create_agent() -> Create a Car agent.
        - add_agent() -> Activate a Car agent from the set of Car agents.
        - remove_agent() -> Deactivate a Car agent from the set of Car agents.
        - step_agent() -> Make a Car agent act: move ahead, choose a new␣
↪cell, eventually reach the destination.
    '''

    @staticmethod
    def create_agent(type: int, param: None, id: int, active_state: int, key:␣
↪int):
        # Setting agent state variables and object
        ## State variables
        current_cell_id = jnp.array([-1])
        requested_cell_id = jnp.array([-1])
        wait_time = jnp.array([-1.0])

        ## State object
        state_content = {'current_cell_id': current_cell_id,␣
↪'requested_cell_id': requested_cell_id, 'wait_time': wait_time}
        agent_state = State(content=state_content)


        # Setting agent param variables and object
        ## Param variables
        destination_cell_id = jnp.array([-1])

        ## Param object
        param_content = {'destination_cell_id': destination_cell_id}
        agent_params = Params(content=param_content)


        # Creating and returning Car agent
        return Car(id=id, active_state=active_state, age = 0.0,␣
↪agent_type=type, params=agent_params, state=agent_state, policy=None,␣
↪key=key)

    @staticmethod
    def add_agent(agents: Set, idx: jnp.array, add_params: Params):
        # Determining agent's slot and setting offset
        agent_to_add = jax.tree_util.tree_map(lambda x: x[idx], agents)
        num_active_cars = add_params.content['num_active_agents']

        # Setting agent state variables and content
        ## State variables
        new_current_cell_id = add_params.content['current_cell_id'][idx -␣
↪num_active_cars]
        new_wait_time = jnp.array([0.0])
        new_requested_cell_id = jnp.array([-1])

        ## State object
        new_state_content = {'current_cell_id': new_current_cell_id,
                             'requested_cell_id': new_requested_cell_id,
                             'wait_time': new_wait_time}
        new_state = State(content=new_state_content)

        # Setting agent param variables and content
        ## Param variables
        new_destination_cell_id = add_params.
↪content['destination_cell_id'][idx - num_active_cars]

        ## Param object
        new_param_content = {'destination_cell_id': new_destination_cell_id}
        new_params = Params(content=new_param_content)

        # Creating and returning Car agent
        return agent_to_add.replace(state=new_state, params=new_params, age=0.
↪0, active_state = True)
```

```
    @staticmethod
    def remove_agent(agents: Set, idx:jnp.array, remove_params: Params):
        # Determining agent's slot
        agent_to_remove = jax.tree_util.tree_map(lambda x:x[idx], agents)

        # Setting agent remove state variables and content
        ## State variables
        new_current_cell_id = jnp.array([-1])
        new_requested_cell_id = jnp.array([-1])
        new_wait_time = jnp.array([-1.0])

        ## State object
        new_state_content = {'current_cell_id': new_current_cell_id,
                             'requested_cell_id': new_requested_cell_id,
                             'wait_time': new_wait_time}
        new_state = State(content=new_state_content)


        # Setting agent remove param variables and content
        ## Param variables
        new_destination_cell_id = jnp.array([-1])

        ## Param object
        new_param_content = {'destination_cell_id': new_destination_cell_id}
        new_params = Params(content=new_param_content)

        # Integrating into removed agent and returning it
        return agent_to_remove.replace(state=new_state, params=new_params,␣
↪age=0.0, active_state = False)

    @staticmethod
    def step_agent(car: Agent, input: Signal, step_params: Params):
        '''
        In the Car.step_agent(agent, input, step_params) a distinction gets␣
↪made between the active agent and the inactive agent
        The inactive agent gets returned as is, while the active agent's␣
↪stepping behavior consists of three phases:
        Phase 1:
        The Car figures out whether or not it was chosen by the cell it chose␣
↪last timestep. It knows this based on car_chosen input.
        If the car was chosen, it will update its current_cell_id to the␣
↪requested_cell_id and reset the waiting time to 0.0.
        If not, we stay in the current_cell_id that we had before and␣
↪increase the waiting time by one unit of dt

        Phase 2:
        The Car decides whether or not it should generate a new␣
↪requested_cell_id based on waiting time.
```

```
        If it decides to not generate a new one, the phase is over here. If␣
↪it chooses a new Cell, there's two types of move choosing behavior:

            Case 1: forced_move
                Occurs when the Car is:
                     - On the destination diagonal (Marked a).
                Behavior:
                    At this point, the result of the move determines whether␣
↪or not the Car will reach its destination, so we cannot leave it up to␣
↪chance.
                    Calculates the correct trajectory the Car should be on to␣
↪eventually reach destination.
            Case 2: free_move or special sub-case partially_forced_move
            Case 2.1: Partially_forced_move
                Occurs when the Car is far enough away from destination to␣
↪not have to take it into account (Marked b).
                    - This is when the Car is more than one unit away from␣
↪the destination or the destination diagonal.
                Behavior:
                    Randomly generates a number between the lower and upper␣
↪bound, which is decided by lane.
                    - Leftmost lane: pick number in [X to X+1].
                    - Middle lane: pick number in [X-1 to X+1].
                    - Rightmost lane: pick number in [X-1 to X].

            Case 2.2: Partially_forced_move
                Occurs when the Car is in a position to switch lanes such␣
↪that it can cut through the destination diagonal (Marked b).
                    - This can only be when the Car is located right next to␣
↪the destination diagonal.
                Behavior:
                    Car decides the next location like in 2.1, but rules out␣
↪left move / right move depending on position relative to destination with␣
↪limiter variable.
                    The Lower bound will be increased by one or the upper␣
↪bound decreased by one.
            ====[Road Example]====
              [ o ]  [ o ]  [ o ]
              [ o ]  [ o ]  [ X ]
              [ o ]  [ a ]  [ c ]
              [ a ]  [ c ]  [ b ]
              [ c ]  [ b ]  [ b ]
            ----------------------
            With    X   Marked   Y
                Empty cells      o
           Destination cell      X
                Case 1 cell      a
              Case 2.1 cell      b
              Case 2.2 cell      c
            ----------------------
        Finally, the possibly updated information will get packaged into a␣
↪new Car agent to replace the old one.
        '''
        def step_inactive_agent():
            return car
        def step_active_agent():
            # Setting variables from state/params content and input/
↪step_params
            requested_cell_id = car.state.content['requested_cell_id']
            current_cell_id = car.state.content['current_cell_id']
            wait_time = car.state.content['wait_time']

            destination_cell_id = car.params.content['destination_cell_id']

            car_chosen = input.content['car_chosen']

            cells = step_params.content['cells']s
            X_max = step_params.content['X_max']
            Y_max = step_params.content['Y_max']
            dt = step_params.content['dt']

            # Phase 1: Determining whether Car moves to requested Cell based␣
↪on car_chosen input.
            new_wait_time, new_current_cell_id = jax.lax.cond(car_chosen,
                                                          # If true,␣
↪reset new wait_time to 0 and new current cell id to the requested cell id.
                                                          lambda _: (jnp.
↪array([0.0]), requested_cell_id),
                                                          # If false, set␣
↪new current cell id to the old cell id and increase wait_time by dt.
                                                          lambda _:␣
↪(wait_time + dt, current_cell_id),
                                                          None)
            direction = cells.params.
↪content['direction'][new_current_cell_id] # up/left/down/right
            moves_vertical = jnp.array([jnp.where(direction[0] % 2 == 0, 1,␣
↪0)]) # If the direction is of even number, the movement is vertical.␣
↪Otherwise it's horizontal.

            # Phase 2: Choosing a next Cell id based on the lane of␣
↪current_cell_id and the location of destination_cell_id.
            ## Finding X and Y of Car and destination and comparing the two␣
↪variables.
            X, Y = jit_cell_id_to_XY(new_current_cell_id, X_max, Y_max,␣
↪moves_vertical)
            destination_X, destination_Y =␣
↪jit_cell_id_to_XY(destination_cell_id, X_max, Y_max, moves_vertical)

            vertical_distance = (destination_Y - Y)[0]
            horizontal_distance = (destination_X - X)[0]

            steps_to_destination, lanes_to_switch, current_lane_index,␣
↪lane_max = jax.lax.cond(moves_vertical[0],
                                                              lambda _:␣
↪(vertical_distance, horizontal_distance, X, X_max),
                                                              lambda _:␣
↪(horizontal_distance, vertical_distance, Y, Y_max),
                                                              None)

            key, lane_switch_key = random.split(car.key)

            def get_request():
                def forced_move():
                    new_lane_index = lanes_to_switch//jnp.
↪abs(steps_to_destination) + current_lane_index
                    return jnp.array([new_lane_index])
                def free_move():
                    # Choosing and applying a limiter for the partial move.
                    ## Determining whether or not a limiter is needed␣
↪(whether there's one more step to destination than difference in number of␣
↪lanes).
                    partial_force_required = jnp.abs(lanes_to_switch) + 1 ==␣
↪jnp.abs(steps_to_destination)

                    ## If the limiter is required, which one would it be?
                    limiter_if_needed = jnp.where(lanes_to_switch > 0, #␣
↪Lanes_to_switch positive = need to move right to reach destination.
                                                  jnp.array([1, 0]), # The␣
↪left move will be blocked out when limiter is applied.
                                                  jnp.where(lanes_to_switch <␣
↪0, # Lanes_to_switch negative = need to move left to reach destination.
                                                          jnp.array([0,␣
↪1]), # The right move will be blocked out when limiter is applied.
                                                          jnp.array([1,␣
↪1]))) # No more moves (you are right below the destination).

                    ## Setting the limiter to the right type based on␣
↪partial_force_required.
                    limiter = jnp.where(partial_force_required,␣
↪limiter_if_needed, jnp.array([0, 0]))

                    # Randomly selecting the new value for X from a pool␣
↪based on Car's lane and limiter.
                    ## Converting lane index into the function index: ind=0␣
↪-> left lane function, ind=lane_max-1 -> right lane function, else middle␣
↪lane function.
                    lane = jnp.where(current_lane_index==lane_max-1, 2, jnp.
↪where(current_lane_index==0, 0, 1))
```

```
                    ## Setting lane functions.
                    def left_border_lane(): # If the Car is the leftmost␣
↪lane, it can stay there or move to the right.
                        new_lane_index = random.randint(lane_switch_key,␣
↪(1,), minval=current_lane_index, maxval=current_lane_index+2-limiter[1])
                        return jnp.array([new_lane_index])

                    def middle_lanes(): # If the Car is in a middle lane␣
↪(neither leftmost nor rightmost), it can stay there or move either left or␣
↪right.
                        new_lane_index = random.randint(lane_switch_key,␣
↪(1,), minval=current_lane_index-1+limiter[0],␣
↪maxval=current_lane_index+2-limiter[1])
                        return jnp.array([new_lane_index])

                    def right_border_lane(): # If the Car is the rightmost␣
↪lane, it can stay there or move to the left.
                        new_lane_index = random.randint(lane_switch_key,␣
↪(1,), minval=current_lane_index-1+limiter[0], maxval=current_lane_index+1)
                        return jnp.array([new_lane_index])
                    lane_move_types = [left_border_lane, middle_lanes,␣
↪right_border_lane]

                    ## Generating and returning the new X value.
                    new_lane_index = jax.lax.switch(lane[0], lane_move_types)
                    return new_lane_index

                ## Collecting new X and Y values.
                new_lane_index = jax.lax.cond(jnp.abs(lanes_to_switch) <␣
↪steps_to_destination,
                                              lambda _: free_move(),
                                              lambda _: forced_move(),
                                              None)[0]
                new_coords_options = jnp.array([(new_lane_index, Y + 1), (X -␣
↪1, new_lane_index), (new_lane_index, Y - 1), (X + 1, new_lane_index)])
                X_new, Y_new = new_coords_options[direction[0]] # As long as␣
↪we are in the South to North road Y always goes up by one every move.

                ## Converting X_new, Y_new to the correct cell_id and␣
↪returning it.
                new_requested_cell_id = jit_XY_to_cell_id(X_new, Y_new,␣
↪X_max, Y_max, moves_vertical)
                return (new_requested_cell_id, key)

            ## Choosing whether a new requested_cell_id has to be generated,␣
↪or whether it keeps the original.
            redraw_condition = jnp.logical_and(new_wait_time[0] > dt,␣
↪new_wait_time[0] <= (5 * dt))
```

```
            new_requested_cell_id, key = jax.lax.cond(redraw_condition, # If␣
↪waiting longer than one timestep, shorter than five timesteps
                                                      lambda _: (requested_cell_id, car.
↪key), # Re-use previously generated request
                                                      lambda _: get_request(), # Generate␣
↪a new requeset
                                                      None)

            # Packaging (new) information into a Car to replace this one in␣
↪the next timestep.
            new_state_content = {'current_cell_id': new_current_cell_id,
                                 'requested_cell_id': new_requested_cell_id,
                                 'wait_time': new_wait_time}
            new_state = State(content=new_state_content)

            return car.replace(state=new_state, key=key, age=car.age + 1.0)
        return jax.lax.cond(car.active_state,
                            lambda _: step_active_agent(),
                            lambda _: step_inactive_agent(),
                            None)
```

[4]:
```
@struct.dataclass
class Cell(Agent):
    '''
    A Cell agent is part of the larger road structure and can hold a car.

    Variables
        - State
            car_id -> id of the car that is in this Cell (or -1 if there are␣
↪none)
            num_cars -> number of cars in the Cell (might not be needed)
        - Params
            X -> X coordinate of Cell in road
            Y -> Y coordinate of Cell in road
            entry -> Whether or not Cars can get spawned in this Cell
            exit -> Whether or not Cars can have this Cell.id has their␣
↪destination_cell_id
            priority_mask -> Multiplied by the Cars interested in this Cell␣
↪to filter out illegal moves and favor the correct ones.
            direction -> In which directions Cars should look for a next Cell.

    Functions
        - create_agent() -> Create an active Cell agent. (there's no inactive␣
↪Cell)
        - set_entry_cell() -> Provide empty entry cells with Car ids
        - step_agent() -> Make a Cell agent act: Update car status, and pick␣
↪a new Car if there's requests.
    '''
    @staticmethod
```

```
  def create_agent(type: int, param: None, id: int, active_state: int, key:␣
↪int):
    # Calculating Cells coordinates and using it together with the shape␣
↪of the road to determine Cell's characteristics.
    ## Gathering X_max, Y_max (represent road dimensions ) and converting␣
↪the Cell id into (X, Y) coordinates.
    X_max = param.content['X_max']
    Y_max = param.content['Y_max']
    direction = param.content['direction'] # 0: up, 1: left, 2: down, 3:␣
↪right
    moves_vertical = jnp.array([jnp.where(direction[0] % 2 == 0, 1, 0)])

    (X, Y) = jit_cell_id_to_XY(jnp.array([id]), X_max, Y_max,␣
↪moves_vertical)




    # Determining Cell's characteristics.
    ## Setting bottom and top (left and right) to entry/exit based on␣
↪direction (Cars sample from this set as destination cells).
    entry_exit_id_per_direction = jnp.array([(jnp.array([id < X_max[0]],␣
↪dtype=jnp.int32), jnp.array([id >= Y_max[0] * X_max[0] - X_max[0]],␣
↪dtype=jnp.int32)),
                        (jnp.array([id >= X_max[0] *␣
↪Y_max[0] - Y_max[0]], dtype=jnp.int32), jnp.array([id < Y_max[0]],␣
↪dtype=jnp.int32)),
                        (jnp.array([id >= Y_max[0] *␣
↪X_max[0] - X_max[0]], dtype=jnp.int32), jnp.array([id < X_max[0]],␣
↪dtype=jnp.int32)),
                        (jnp.array([id < Y_max[0]],␣
↪dtype=jnp.int32), jnp.array([id >= X_max[0] * Y_max[0] - Y_max[0]],␣
↪dtype=jnp.int32))])
    entry, exit = entry_exit_id_per_direction[direction[0]]

    ## Setting priority mask (dependent on lane).
    ### Converting X into the function index: X=0 -> left lane function,␣
↪X=X_max-1 -> right lane function, else middle lane function.
    current_lane_index, lane_max = jax.lax.cond(moves_vertical[0], lambda␣
↪_: (X, X_max), lambda _: (Y, Y_max), None)
    lane = jnp.where(current_lane_index==lane_max-1, 2, jnp.
↪where(current_lane_index==0, 0, 1))

    ### Setting priority mask functions and picking one.
    def left_priority_mask(): # For Cells in the leftmost lane.
      return jnp.array([0, 3, 2, 0, 0, 0, 0, 0], dtype=jnp.int32)
    def right_priority_mask(): # For Cells in the middle lanes.
      return jnp.array([2, 3, 0, 0, 0, 0, 0, 0], dtype=jnp.int32)
    def center_priority_mask(): # For Cells in the rightmost lane.
      return jnp.array([2, 3, 1, 0, 0, 0, 0, 0], dtype=jnp.int32)
```

```
        choices = [left_priority_mask, center_priority_mask,␣
↪right_priority_mask]
        priority_mask =  jax.lax.switch(lane[0], choices)
        priority_mask = jnp.roll(priority_mask, shift=2*direction[0])
        '''
        Surroundings are indexed as follows:
        6 5 4
        7 o 3
        0 1 2
        Therefore, you can get the correct direction variant by moving all␣
↪priority_mask 2 units to right for every leftward rotation from going up.
        '''

        # Integrating information into Params object for Agent
        agent_params_content = { "X": X, "Y": Y, "entry": entry, "exit":␣
↪exit, "priority_mask": priority_mask, "direction": direction[0]}
        agent_params = Params(content=agent_params_content)

        # Setting Cell state variables and integrating it into State object.
        car_id = jnp.array([-1])
        num_cars = jnp.array([0])
        agent_state_content = {"car_id": car_id, "num_cars": num_cars}
        agent_state = State(content=agent_state_content)

        # Creating and returning Cell agent.
        return Cell(id = id, active_state=active_state, age = 0.0,␣
↪agent_type= type, params= agent_params, state= agent_state, policy = None,␣
↪key = key)

    @staticmethod
    def set_entry_cell(agents: Set, idx: jnp.array, set_params: Params):
        # Selecting agent to replace
        agent_to_set = jax.tree_util.tree_map(lambda x: x[idx], agents)

        # Updating state variables to show that has a car now
        num_cars_new = agent_to_set.state.content['num_cars']+1
        car_id_to_add = jnp.array([set_params.content['car_id'][idx]])

        # Integrating state variables into State object and replacing agent
        new_state_content = {'car_id': car_id_to_add, 'num_cars':␣
↪num_cars_new}
        new_state = State(content=new_state_content)
        return agent_to_set.replace(state=new_state)

    @staticmethod
    def step_agent(cell: Agent, input: Signal, step_params: Params):
        # Getting variables from step_params and state/params content
        cars = step_params.content['cars']
        cells = step_params.content['cells']
        X_max = step_params.content['X_max']
        Y_max = step_params.content['Y_max']

        num_cars = cell.state.content['num_cars']
        car_id = cell.state.content['car_id']

        priority_mask = cell.params.content['priority_mask']
        direction = cell.params.content['direction']
        moves_vertical = jnp.where(direction % 2 == 0, 1, 0)

        car_indx = jnp.argmax(jnp.where(car_id[0]==cars.id, 1, 0))
        car = jax.tree_util.tree_map(lambda x: x[car_indx], cars)
        car_cell_id = car.state.content['current_cell_id'] # In what Cell the␣
↪Car in this Cell is according to them

        # Phase 1: Donor
        '''
        Donor phase
        Check whether the Cell was set to empty one timestep ago
        If it was, then we still have a free spot.
        If it wasn't, we check if the Car that is in the Cell now is set to␣
↪active and if it believes that it is in this Cell
            If both are True, then we have a Car
            If not, then the Car has advanced to a different Cell or it was␣
↪removed from the simulation: we have a free spot.
        '''
        donor_phase_car_id, donor_phase_num_cars = jax.lax.cond(car_id[0] ==␣
↪-1, #
            lambda _: (jnp.array([-1]), num_cars - 1),
            lambda _: jax.lax.cond(jnp.logical_and(car_cell_id[0] == cell.id,␣
↪car.active_state),
                lambda _: (car_id, num_cars),
                lambda _: (jnp.array([-1]), num_cars - 1),
                None),
            None)

        # Phase 2: Recipient
        '''
        Look at the (max) 8 Cells surrounding this Cell:
        Relative coords -> Absolute coords -> Cell ids -> Car ids in Cell ->␣
↪Car ids that want to come to this Cell.
        If at any point the value is no longer interesting, it will be␣
↪filtered out:
            If a coordinate is outside of the road the Cell id will be -1.
            If a Cell has no Cars the id given will be -1.
            If a Car has requested a different cell it will give a 0, whereas␣
↪other Cars get 1.

        The Cell takes the array of 8 boolean values of whether a Car wants␣
↪to come to it and multiply it by the priority_mask and the wait_times
```

```
        to get a preference array, of which it takes the highest value index,␣
↪and then the Cell takes that index from the array of Car ids to get the␣
↪favorite.

        Then the Cell checks whether it has an open slot and it succesfully␣
↪found a Car id.
        If both these conditions are True, the Cell takes the preffered Car␣
↪and increases the num_cars by one.
        If not, The old car will remain in the Cell and the num_cars will␣
↪stay the same as before.
        '''
        # Relative positions to absolute positions to cell ids around Cell.
        cells_dXY = jnp.array([[[-1], [-1]], [[0], [-1]], [[1], [-1]], [[1],␣
↪[0]], [[1], [1]], [[0], [1]], [[-1], [1]], [[-1], [0]]])
        cells_XY_around_me = jnp.array([cell.params.content['X'], cell.params.
↪content['Y']]) + cells_dXY
        cells_id_around_me = jax.vmap(jit_XY_to_cell_id, in_axes=(0, 0, None,␣
↪None, None))(cells_XY_around_me[:, 0], cells_XY_around_me[:, 1], jnp.
↪array([X_max]), jnp.array([Y_max]), jnp.array([moves_vertical]))

        # Convert cell_ids to car_ids
        def get_car_ids(cell_id):
            # If the cell id is valid (not -1), take its car_id. If not␣
↪return an invalid value: jnp.array([-1])
            cell_id = cell_id[0]
            return jax.lax.cond(cell_id >= 0,
                                lambda _: cells.state.content['car_id'][cell_id],
                                lambda _: jnp.array([-1]),
                                None)
        car_ids_around_me = jax.vmap(get_car_ids)(cells_id_around_me)

        # Get requested_cell_id and wait_time from car_ids
        def get_car_requested_cell_ids_wait_times(car_id):
            # If the Car id is valid (not -1), return its requested_cell_id␣
↪and wait_time. If not return invalid values: (jnp.array([-1]) jnp.array([-1.
↪0]))
            car_indx = jnp.argmax(jnp.where(car_id[0] == cars.id, 1, 0))
            return jax.lax.cond(car_id[0] >= 0,
                                    lambda _: (cars.state.
↪content['requested_cell_id'][car_indx], cars.state.
↪content['wait_time'][car_indx]), # get the request cell id and wait time
                                    lambda _: (jnp.array([-1]), jnp.array([-1.0])), #␣
↪if the car id is -1, then return -1, -1.0
                                    None)
        car_requested_cell_ids, car_wait_times = jax.
↪vmap(get_car_requested_cell_ids_wait_times)(car_ids_around_me)

        # Picking a favorite Car
        car_options = jnp.where(car_requested_cell_ids == cell.id, 1, 0)  #␣
↪Array of indices of cars that want to come to this Cell
```

```
        cars_squeezed = jnp.squeeze(car_options, axis=1) # Flatten [[x] [x]␣
↪[x] [x] [x] [x] [x] [x]] into [x x x x x x x x]
        relevant_cars = jnp.multiply(cars_squeezed, priority_mask) # Cars␣
↪that are legal to come to this Cell, with built-in lane preference
        preference = jnp.multiply(relevant_cars, jnp.squeeze(car_wait_times,␣
↪axis=1))  # Array of preference across cars

        # Getting the preferred car out
        ## Need to build in a check that the chosen car is not 0 or less,␣
↪because that would by-default make it take the first 0 value, which is just␣
↪the first index
        max_value = jnp.max(preference)
        highest_value_index = jnp.where(max_value <= 0, -1, jnp.
↪argmax(preference))

        ## Converting index to car_id
        preferred_car_id = car_ids_around_me[highest_value_index]

        # Setting new car id and number of cars based on succes in finding a␣
↪new car and capacity in Cell
        new_car_id, new_num_cars = jax.lax.cond(jnp.
↪logical_and(donor_phase_num_cars[0] < 1, preferred_car_id[0] >= 0),# If␣
↪less than max capacity and preferred_car is not -1
                                                lambda _:␣
↪(preferred_car_id, donor_phase_num_cars+1), # Pull car in
                                                lambda _:␣
↪(donor_phase_car_id, donor_phase_num_cars), # Remain with old car
                                                None)

        # Updating state content and replacing agent
        new_state_content = {'car_id': new_car_id, 'num_cars': new_num_cars}
        new_state = State(content=new_state_content)
        return cell.replace(state=new_state)
```

6.3.2 Road simulation

[5]:
```
# Helper functions
def car_chosen(cars: Set, cells: Set):
    '''
    Function that checks for each car whether it was chosen by the Cell it␣
↪requested, V-mapped across cars.
    args:
        cars: Set of Car agents.
        cells: Set of Cell agents.
    returns:
        chosen: jnp.array of length cars filled with True/False values.
    '''
    def single_car_chosen(car: Agent, cells: Set):
        requested_id = car.state.content['requested_cell_id']
        chosen_id = cells.state.content['car_id'][requested_id][0]
        chosen = jax.lax.cond(requested_id[0] >= 0,
                              lambda _: car.id == chosen_id[0], # Only to␣
↪move if the cell chose the car id
                              lambda _: False, # If no cell has been␣
↪requested yet, car is not to move.
                              None)
        return chosen

    chosen_arr = jax.vmap(single_car_chosen, in_axes=(0, None))(cars, cells)
    return chosen_arr

jit_car_chosen = jax.jit(car_chosen)

def select_finished_cars(cars: Set, select_params: None):
    '''
    Function that checks for each Car whether they have reached their␣
↪destination.
    args:
        cars: Set of Car agents.
        select_params: Unused argument that is required by jit.
    returns:
        arrived: jnp.array of length cars filled with True/False values.
    '''
    def check_for_one_car(car: Agent):
        current_id = car.state.content['current_cell_id']
        destination_id = car.params.content['destination_cell_id']

        arrived = jnp.logical_and(current_id == destination_id, car.
↪active_state)
        return arrived

    return jax.vmap(check_for_one_car, in_axes=(0))(cars)


def create_cell_and_car_set(key_seed = 8, road_shape = (3, 7), num_cars = 10,␣
↪num_active_cars = 0, direction = 0):
    '''
    Creates the Cell and Car agents and sets as specified in the arguments
    args:
        key_seed: int on which the random pseudorandom key is based. This␣
↪function call is likely the first in the simulation process, so it makes␣
↪sense to create the key from here.
        road_shape: shape in which the Cell integrate into a larger structure.
↪ A tuple of (X, Y), also referred to as X_max and Y_max.
        num_cars: the maximum number of Cars that can be active at once in␣
↪the simulation
        num_active_cars: the number of Cars that are active from the start.␣
↪This is usually zero, because the simulation takes care of adding Cars to␣
↪entry Cells.
```

```
    '''
    key = jax.random.PRNGKey(key_seed)

    # Creating Cars
    car_key, cell_key = random.split(key)
    car_set = create_agents(agent = Car, params = None, num_agents =␣
↪num_cars, num_active_agents = num_active_cars, agent_type = 3, key =␣
↪car_key)
    car_set = Set(agents=car_set, num_agents=num_cars,␣
↪num_active_agents=num_active_cars, state=None, params=None, policy=None,␣
↪id=0, set_type=0, key=None)

    # Setting cell_create params variables
    x, y = road_shape
    num_cells = x * y

    X = jnp.array([x])
    X_max_arr = jnp.tile(X, (num_cells, 1))
    Y = jnp.array([y])
    Y_max_arr = jnp.tile(Y, (num_cells, 1))
    direction = jnp.array([direction])
    direction_arr = jnp.tile(direction, (num_cells, 1))

    # Creating Cells
    cell_create_params = Params(content={'X_max': X_max_arr, 'Y_max':␣
↪Y_max_arr, 'direction': direction_arr})
    cell_set = create_agents(Cell, cell_create_params, num_cells, num_cells,␣
↪2, cell_key)
    cell_set = Set(agents=cell_set, num_agents=num_cells,␣
↪num_active_agents=num_cells, state=None, params=None, policy=None, id=0,␣
↪set_type=1, key=None)
    return cell_set, car_set, key
```

6.3.3 Parallel simulation

With Simulation agent. working in dimensions (n, c) = (7, 3) and with Cars travelling up. Because Simulation agent must be usable in parallel, direction and dimensions cannot be modified from the outside, because the spawning function has to work with static sizes. The variables can be modified from the code however.

[6]:
```
X_MAX = jnp.array([3]) # 3 lanes
Y_MAX = jnp.array([7]) # 7 length of the road
NUM_CARS = 10

direction = 0
Dt = 1.0  # time step

CAR_AGENT_TYPE = 0
CELL_AGENT_TYPE = 1

SIM_CREATE_PARAMS = Params(content={
    "X_max": X_MAX,
    "Y_max": Y_MAX,
    "num_cars": NUM_CARS,
    "car_agent_type": CAR_AGENT_TYPE,
    "cell_agent_type": CELL_AGENT_TYPE
})
```

[7]:
```
num_cars = 100
car_create_params = None
cars = create_agents(Car, params=None, num_agents=num_cars,␣
 ↪num_active_agents=0, agent_type=0, key=random.PRNGKey(0))
car_Set = Set(num_agents=num_cars, num_active_agents=0, agents=cars, id =0,␣
 ↪set_type=0, params=None, key=None, state=None, policy=None)

print("Initial Car Set:")
print("cars active state:", car_Set.agents.active_state.reshape(-1))
print("cars current cell id:", car_Set.agents.state.
 ↪content['current_cell_id'].reshape(-1))
print("cars requested cell id:", car_Set.agents.state.
 ↪content['requested_cell_id'].reshape(-1))
print("cars wait time:", car_Set.agents.state.content['wait_time'].
 ↪reshape(-1))

car_add_params = Params(content={'num_active_agents': car_Set.
 ↪num_active_agents,
                                 'current_cell_id': jnp.array([[0], [1], [2],␣
 ↪[3], [4]]), # Example current cell ids for new cars
                                 'destination_cell_id': jnp.array([[5], [6],␣
 ↪[7], [8], [9]])}) # Example destination cell ids for new cars
num_cars_to_add = 3
car_Set = jit_add_agents(Car.add_agent, car_add_params, num_cars_to_add,␣
 ↪car_Set)

print("\nCar Set after adding new cars:")
print("cars active state:", car_Set.agents.active_state.reshape(-1))
print("cars current cell id:", car_Set.agents.state.
 ↪content['current_cell_id'].reshape(-1))
print("cars requested cell id:", car_Set.agents.state.
 ↪content['requested_cell_id'].reshape(-1))
print("cars wait time:", car_Set.agents.state.content['wait_time'].
 ↪reshape(-1))

cars_remove_params = Params(content={'remove_indx': jnp.array([0, 2])}) #␣
 ↪Indices of cars to remove
num_cars_to_remove = len(cars_remove_params.content['remove_indx'])
car_Set, _ = jit_remove_agents(Car.remove_agent, cars_remove_params,␣
 ↪num_cars_to_remove, car_Set)
print("\nCar Set after removing some cars:")
print("cars active state:", car_Set.agents.active_state.reshape(-1))
```

```python
print("cars current cell id:", car_Set.agents.state.
 ↪content['current_cell_id'].reshape(-1))
print("cars requested cell id:", car_Set.agents.state.
 ↪content['requested_cell_id'].reshape(-1))
print("cars wait time:", car_Set.agents.state.content['wait_time'].
 ↪reshape(-1))
```

```
Initial Car Set:
cars active state: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
cars current cell id: [-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1]
cars requested cell id: [-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1]
cars wait time: [-1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
-1. -1.
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]

Car Set after adding new cars:
cars active state: [1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
cars current cell id: [ 0  1  2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1]
cars requested cell id: [-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1]
cars wait time: [ 0.  0.  0. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
-1. -1.
```

```
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]

Car Set after removing some cars:
cars active state: [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
cars current cell id: [ 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1]
cars requested cell id: [-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1]
cars wait time: [ 0. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
-1. -1.
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
```

[8]:
```python
def spawn_cars_in_entry_cells(key, cells, cars, num_active_cars):
    """
    this function generates data that will be used to update cars and cells␣
 ↪such that new cars are spawned in the entry cells
    we will take advantage of the fact that the for_loops in jit_add_agent␣
 ↪for cars and jit_set_agents for cells will only go to the number of cars we␣
 ↪want to add
    thus a natual constraint is that num cars to add = num entry cells to␣
 ↪update = between 1 and total number of entry cells=3
    args:
    key: random key
    cells: cells agent
    num_active_cars: number of active cars in the simulation
    #
    Update check how the car ids and indexes are disentangled
    """
    entry_cell_ids = jnp.array([0, 1, 2]) # entry cell ids
    car_indx = jnp.array([0, 1, 2]) + num_active_cars # car ids at most 3␣
↪cars can be spawned, num_active_cars is the number of cars already in the␣
↪simulation, thus taking adv of the fact that agents are always appended to␣
↪the end of the list
    key, *spawn_car_keys = random.split(key, 4)

    #first shuffle the entry cells
    entry_cell_ids = jax.random.permutation(spawn_car_keys[0], entry_cell_ids)

    #next take the num_cars from the shuffled entry cells
    num_cars_entry_cells = jnp.take(cells.state.content['num_cars'],␣
↪entry_cell_ids)
    is_cell_available = jnp.where(num_cars_entry_cells == 0, 1, 0) # check if␣
↪the entry cell is available 1-> available, 0-> available
    current_cell_idx = jnp.array([0, 1, 2])
    current_cell_idx = jnp.argsort(-1*is_cell_available) # sort in descending␣
↪order, so that the available cells are at the beginning

    # now sort the entry cells based on the availability
    entry_cell_ids = jnp.take(entry_cell_ids, current_cell_idx)
    # now sort the car ids based on the availability
    #car_ids = jnp.take(car_ids, jnp.take(current_cell_idx,␣
↪current_cell_idx)) # don't know why this works but it does
    idx= jnp.argsort(entry_cell_ids)
    car_indx = jnp.take(car_indx, idx)
    car_ids = jnp.take(cars.id, car_indx) # car ids are different from␣
↪indexes, so we need to take the car ids from the cars agent

    # now lets determine the exit cell ids, just randomly choose 3 exit cell␣
↪ids
    exit_cell_ids = jax.random.randint(spawn_car_keys[1], (3,), minval=18,␣
↪maxval=21) # exit cell ids

    car_add_params = Params(content={'current_cell_id': entry_cell_ids,␣
↪'destination_cell_id': exit_cell_ids, 'num_active_agents': num_active_cars})
    cell_set_params = Params(content={'set_indx': entry_cell_ids, 'car_id':␣
↪car_ids})

    num_cars_to_add = jax.random.randint(spawn_car_keys[2], (1,), minval=1,␣
↪maxval=4) # number of cars to add, everything heavily relies on the fact␣
↪that the for-loops will just go to this number
    num_cars_to_add = jnp.minimum(num_cars_to_add[0], jnp.
↪sum(is_cell_available)) # make sure that the number of cars to add is less␣
↪than the number of available cells

    return car_add_params, cell_set_params, num_cars_to_add, key

jit_spawn_cars_in_entry_cells = jax.jit(spawn_cars_in_entry_cells)
```

```
[9]: @struct.dataclass
class Simulation:
    cars: Set
    cells: Set
    key: jax.random.PRNGKey

    @staticmethod
    def create_simulation(params, key):
        num_cars = params.content['num_cars']
        X_max = params.content['X_max']
        Y_max = params.content['Y_max']

        cars_params = None
        key, car_key = random.split(key)
        car_agent_type = params.content['car_agent_type']
        cars = create_agents(Car, params=cars_params, num_agents=num_cars,␣
↪num_active_agents=0, agent_type=car_agent_type, key=car_key)
        car_Set = Set(num_agents=num_cars, num_active_agents=0, agents=cars,␣
↪id=0, set_type=car_agent_type, params=None, key=None, state=None,␣
↪policy=None)

        key, cell_key = random.split(key)
        cell_agent_type = params.content['cell_agent_type']
        num_cells = X_max[0] * Y_max[0]
        X_max_arr = jnp.tile(X_max, (num_cells, 1)) # 3 lanes
        Y_max_arr = jnp.tile(Y_max, (num_cells, 1)) # length of road 7
        direction_arr = jnp.tile(0, (num_cells, 1)) # all Cells point upward
        cell_params = Params(content={'X_max': X_max_arr, 'Y_max': Y_max_arr,␣
↪'direction': direction_arr})
        cells = create_agents(Cell, params=cell_params, num_agents=num_cells,␣
↪num_active_agents=num_cells, agent_type=cell_agent_type, key=cell_key)
        cell_Set = Set(num_agents=num_cells, num_active_agents=num_cells,␣
↪agents=cells, id=0, set_type=cell_agent_type, params=cell_params, key=None,␣
↪state=None, policy=None)
        return Simulation(cars=car_Set, cells=cell_Set, key=key)

    @staticmethod
    @jax.jit
    def step(sim: Agent):
        # step 1: spawn new cars in the entry cells
        cars_add_params, cells_set_params, num_cars_to_add, key =␣
↪jit_spawn_cars_in_entry_cells(sim.key, sim.cells.agents, sim.cars.agents,␣
↪sim.cars.num_active_agents)

        cars = jit_add_agents(Car.add_agent, cars_add_params,␣
↪num_cars_to_add, sim.cars)
        cells = jit_set_agents(Cell.set_entry_cell, cells_set_params,␣
↪num_cars_to_add, sim.cells)
```

```
        # step 2: get_which car has moved
        chosen_arr = jit_car_chosen(cars.agents, cells.agents)
        car_step_inputs = Signal(content={'car_chosen': chosen_arr}) #␣
↪Creating a Signal to be used in the Car step function.

        # step 3: step cars
        car_step_params = Params(content={'X_max': X_MAX,
                                          'Y_max': Y_MAX,
                                          'cells': cells.agents,
                                          'dt': Dt})
        cars = jit_step_agents(Car.step_agent, car_step_params, ␣
↪car_step_inputs, cars)

        # step 4: remove cars
        num_agents_selected, selected_indx = jit_select_agents(
            select_func=select_finished_cars,
            select_params=None,
            set=cars
        )
        car_remove_params = Params(content={'remove_indx': selected_indx})
        car_set, sorted_indx = jit_remove_agents(
            remove_func=Car.remove_agent,
            remove_params=car_remove_params,
            num_agents_remove=num_agents_selected,
            set=cars
        )

        # step 5: step cells
        cell_step_params = Params(content={'cars': cars.agents,
                                           'cells': cells.agents,
                                           'X_max': X_MAX[0],
                                           'Y_max': Y_MAX[0]})
        cell_step_input = None
        cells = jit_step_agents(Cell.step_agent, cell_step_params,␣
↪cell_step_input, cells)


        return Simulation(cars=cars, cells=cells, key=key), cells_set_params
```

[10]:
```
#test the simulation creation
sim = Simulation.create_simulation(SIM_CREATE_PARAMS, random.PRNGKey(0))
print("Initial Simulation:")
print("Cars active state:", sim.cars.agents.active_state.reshape(-1))
print("Cars current cell id:", sim.cars.agents.state.
↪content['current_cell_id'].reshape(-1))
print("Cars requested cell id:", sim.cars.agents.state.
↪content['requested_cell_id'].reshape(-1))
print("Cars wait time:", sim.cars.agents.state.content['wait_time'].
↪reshape(-1))
print("Cells active state:", sim.cells.agents.active_state.reshape(-1))
print("Cells car id:", sim.cells.agents.state.content['car_id'].reshape(-1))
print("Cells num cars:", sim.cells.agents.state.content['num_cars'].
 ↪reshape(-1))

sim.cells.agents.state.content['car_id'].reshape(-1)
```

```
Initial Simulation:
Cars active state: [0 0 0 0 0 0 0 0 0 0]
Cars current cell id: [-1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
Cars requested cell id: [-1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
Cars wait time: [-1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
Cells active state: [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
Cells car id: [-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
Cells num cars: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

[10]:
```
Array([-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
       -1, -1, -1, -1], dtype=int32)
```

[11]:
```
sim, cell_set_params = Simulation.step(sim)
```

[12]:
```
# test step
print("\nSimulation after one step:")
print("Cars active state:", sim.cars.agents.active_state.reshape(-1))
print("Cars current cell id:", sim.cars.agents.state.
 ↪content['current_cell_id'].reshape(-1))
print("Cars requested cell id:", sim.cars.agents.state.
 ↪content['requested_cell_id'].reshape(-1))
print("Cars wait time:", sim.cars.agents.state.content['wait_time'].
 ↪reshape(-1))
print("Cells car id:", sim.cells.agents.state.content['car_id'].reshape(-1))
print("Cells num cars:", sim.cells.agents.state.content['num_cars'].
 ↪reshape(-1))
print("cell ids:", sim.cells.agents.id.reshape(-1))
print("\nCell Set Parameters after one step:")
print("Set Indices:", cell_set_params.content['set_indx'].reshape(-1))
print("Car IDs:", cell_set_params.content['car_id'].reshape(-1))
print(cell_set_params)
```

```
Simulation after one step:
Cars active state: [1 0 0 0 0 0 0 0 0 0]
Cars current cell id: [ 1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
Cars requested cell id: [ 3 -1 -1 -1 -1 -1 -1 -1 -1 -1]
Cars wait time: [ 1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
Cells car id: [-1  0  0  0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
Cells num cars: [-1  1  0  0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
cell ids: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20]

Cell Set Parameters after one step:
Set Indices: [1 0 2]
Car IDs: [1 0 2]
Params(content={'car_id': Array([1, 0, 2], dtype=int32), 'set_indx': Array([1,
0, 2], dtype=int32)})
```

6.3.4 Simulate dynamic road size and direction

Without using Simulation agent -> only one simulation can take place at once. Simulating for multiple steps and displaying road step by step.

[13]:
```
'''
The global step is in order:
1. Spawn Cars in start Cells.
2. Check if Cars were accepted by their requested Cells.
3. Step Cars to move and/or request Cells.
4. Remove Cars that have reached their destination.
5. Step Cells to update the Cars that were previously in them, and accept new␣
↪Cars that will come to them
'''
def cars_in_entry_cells(key: jnp.array, cell_set: Set, car_set: Set, X_max:␣
↪jnp.array, Y_max: jnp.array, num_entry_exit_cells: int):
    '''
    This function generates data that will be used to update cars and cells␣
↪such that new cars are spawned in the entry cells.
    We will take advantage of the fact that the for_loops in jit_add_agent␣
↪for cars and jit_set_agents for cells will only go to the number of cars we␣
↪want to add.
    Thus a natural constraint is that num cars to add = num entry cells to␣
↪update = between 1 and total number of entry cells=3.
    args:
        key -> Random key.
        cell_set -> Set of Cell agents.
        car_set -> Set of Car agents.
        X_max -> X dimension of road structure: number of lanes.
        Y_max -> Y dimension of road structure: number of rows.
    Returns:
        car_add_params: Params content for the Cars that will be added.
        cell_set_params: Params content for the Cells that will house the␣
↪Cars about to be added.
        num_cars_to_add: Integer that represents how many Cars will be added␣
↪in the next timestep.
        key: Split original random key.
    '''
    # Getting variables out of the args.
    cells = cell_set.agents
    cars = car_set.agents
    num_active_cars = car_set.num_active_agents # The number of active Cars␣
↪in the simulation.

    # To make road size flexible and scalable. (cars added based on number of␣
↪lanes, which is based on x_max/y_max depending on heading dimension)
    num_lanes_aranged = jnp.arange(num_entry_exit_cells) # Used for ids. #␣
↪Replace X_max_value with X_max[0]
```

```
    # Make use of entry and exit variables, instead of hardcoding the bottom␣
↪three as entry Cells and the top three as exits.
    ## We actually only have to do this vmap once for every update to the␣
↪road structure, TODO

    entries = (2 * cells.params.content['entry'].flatten() - 1) * cells.id #␣
↪Convert entry [0,1] to [-1, 1] and multiply by id * [0, 20]
    exits = (2 * cells.params.content['exit'].flatten() - 1) * cells.id #␣
↪Convert exit [0,1] to [-1, 1] and multiply by id * [0, 20]

    ## Includes a lot of -1's throughout, we sort the values and then take␣
↪the number of entry and exit Cells we know there are
    sort_idx = jnp.argsort((entries <= 0))
    sorted_ids = entries[sort_idx]
    entry_cell_ids = sorted_ids[:num_entry_exit_cells]

    sort_idx = jnp.argsort((exits <= 0))
    sorted_ids = exits[sort_idx]
    exit_cell_ids = sorted_ids[:num_entry_exit_cells]


    # Prepare car indices and keys
    car_indx = num_lanes_aranged.copy() + num_active_cars # At most num_lanes␣
↪cars can be spawned, num_active_cars is the number of Cars already in the␣
↪simulation, thus taking advantage of the fact that agents are always␣
↪appended to the end of the list.
    key, *spawn_car_keys = random.split(key, 4)

    # Shuffling the entry Cells.
    entry_cell_ids = jax.random.permutation(spawn_car_keys[0], entry_cell_ids)

    # Taking the num_cars from the shuffled entry Cells.
    num_cars_entry_cells = jnp.take(cells.state.content['num_cars'],␣
↪entry_cell_ids)
    is_cell_available = jnp.where(num_cars_entry_cells == 0, 1, 0) # Check if␣
↪the entry Cell is available: 1-> available, 0-> not available.
    current_cell_idx = entry_cell_ids.copy()
    current_cell_idx = jnp.argsort(-1*is_cell_available) # Sort in descending␣
↪order, so that the available Cells are at the beginning.

    # Sorting the entry Cells based on whether they have free spots.
    entry_cell_ids = jnp.take(entry_cell_ids, current_cell_idx)

    # Sorting the car_ids based on the availability of cells.
    idx = jnp.argsort(entry_cell_ids)
    car_indx = jnp.take(car_indx, idx)
    car_ids = jnp.take(cars.id, car_indx) # car_ids are different from␣
↪indices, so we need to take the car_ids from the cars agent.
```

```
    # Determining the exit Cell ids for the Cars that will be added, just␣
↪randomly choose num_lanes exit Cell ids.
    car_exit_cell_ids = jax.random.choice(key=spawn_car_keys[1],
                                          a=exit_cell_ids,
                                          shape=(num_entry_exit_cells,),
                                          replace=True)

    # Determining how many cars to add: 1 to number of lanes and never more␣
↪than total number of available Cell entries.
    num_cars_to_add = jax.random.randint(spawn_car_keys[2], (1,), minval=1,␣
↪maxval=num_entry_exit_cells+1) # Number of cars to add, everything heavily␣
↪relies on the fact that the for-loops will only go to this number, not␣
↪further.
    num_cars_to_add = jnp.minimum(num_cars_to_add[0], jnp.
↪sum(is_cell_available)) # Make sure that the number of cars to add is less␣
↪than the number of available cells.

    # Package and return
    car_add_params = Params(content={'current_cell_id': entry_cell_ids,␣
↪'destination_cell_id': car_exit_cell_ids, 'num_active_agents':␣
↪num_active_cars})
    cell_set_params = Params(content={'set_indx': entry_cell_ids, 'car_id':␣
↪car_ids})

    return car_add_params, cell_set_params, num_cars_to_add, key
def simulate(cell_set: Set, car_set: Set, key: int, shape: tuple = (3, 7),␣
↪direction: int = 0, lane_dim: int = 0, num_iter: int = 16):
    # Both X_max_val and X_max (and their Y counterpart) need to be used in␣
↪jax.lax.scan step function, because you can not concretize variables while␣
↪scanning.
    X_max = jnp.array([shape[0]])
    Y_max = jnp.array([shape[1]])

    num_entry_exit_cells = shape[lane_dim]
    dt = 1.0

    # Define step function.
    def step(carry: tuple, iteration: int):
        '''
        Represents a single step in the simulation, behaving as listed above.
        args:
            carry: tuple of cell_set: Set of Cells, car_set: Set of Cars and␣
↪key: a (pseudo)random int.
            iteration: int, represents what number of step this run is in the␣
↪simulation.
        returns:
        (cell_set, car_set, and key), the updated versions of the carry␣
↪variables cell_set, car_set, and key.
```

```
        car_positions_over_time: a 1D jnp.array of length num_cars, each␣
↪index is a Car id and the value in this array the Cell index or -1 for␣
↪inactive cars.
            Will later be used for visually representing each state
        '''
        # 0. Take values from previous timestep (or initial conditions).
        cell_set, car_set, key = carry

        # 1. Spawn Cars in start Cells.
        car_add_params, cell_set_params, num_cars_to_add, key =␣
↪cars_in_entry_cells(key, cell_set, car_set, X_max, Y_max,␣
↪num_entry_exit_cells)
        car_set = jit_add_agents(add_func=Car.add_agent,␣
↪add_params=car_add_params, num_agents_add=num_cars_to_add, set=car_set)
        cell_set = jit_set_agents(set_func=Cell.set_entry_cell,␣
↪set_params=cell_set_params, num_agents_set=num_cars_to_add, set=cell_set)


        jax.debug.print("Simulation at t={}\n\t\t-Added {} car(s)",␣
↪iteration, num_cars_to_add)

        # 2. Find chosen Cars.
        car_chosen = jit_car_chosen(car_set.agents, cell_set.agents)

        # 3. Step Cars.
        car_step_input = Signal(content={'car_chosen': car_chosen})
        car_step_params = Params(content={'dt': dt, 'X_max': X_max, 'Y_max':␣
↪Y_max, 'cells': cell_set.agents})
        car_set = jit_step_agents(step_func=Car.step_agent,␣
↪step_params=car_step_params, input=car_step_input, set=car_set)

        jax.debug.print("\t\t-Active state: {}\n\t\t-Cell ids: {}␣
↪\n\t\t-Requested cell ids: {} \n\t\t-Destination cell ids: {}", car_set.
↪agents.active_state, car_set.agents.state.content['current_cell_id'].
↪reshape(-1), car_set.agents.state.content['requested_cell_id'].reshape(-1),␣
↪car_set.agents.params.content['destination_cell_id'].reshape(-1))

        # 4. Remove Cars at destination.
        num_agents_selected, selected_indx = jit_select_agents(
            select_func=select_finished_cars,
            select_params=None,
            set=car_set
        )
        car_remove_params = Params(content={'remove_indx': selected_indx})
        car_set, sorted_indx = jit_remove_agents(
            remove_func=Car.remove_agent,
            remove_params=car_remove_params,
            num_agents_remove=num_agents_selected,
            set=car_set
        )
        jax.debug.print("\t\t-{} cars arrived at their destination.
↪\n\t\t-Sorted indices after removal: {}", num_agents_selected, sorted_indx)

        # 5. Step cells.
        cell_step_params = Params(content={
            'cars': car_set.agents,
            'cells': cell_set.agents,
            'Y_max': shape[1],
            'X_max': shape[0]
        })
        cell_set = jit_step_agents(step_func=Cell.step_agent, input=None,␣
↪step_params=cell_step_params, set=cell_set)

        jax.debug.print("\t\t-Cell X coord: \t{}\n\t\t-Cell Y coord:␣
↪\t{}\n\t\t-Cell ids: \t{}\n\t\t-Chosen cars: \t{}", cell_set.agents.params.
↪content['X'].reshape(-1), cell_set.agents.params.content['Y'].reshape(-1),␣
↪cell_set.agents.id, cell_set.agents.state.content['car_id'].reshape(-1))
        #'''

        car_current_cell_ids = car_set.agents.state.
↪content['current_cell_id'].reshape(-1)
        return (cell_set, car_set, key), car_current_cell_ids

    # Setup initial conditions.
    initial_conditions = (cell_set, car_set, key)

    # Perform the simulation.
    carry_final, car_positions_over_time = jax.lax.scan(step,␣
↪initial_conditions, jnp.arange(num_iter))
    return carry_final, car_positions_over_time
```

[14]:
```
# Setup
direction = 0
lane_dim = jnp.where(direction % 2 == 0, 0, 1)
shape = (7, 3) if lane_dim else (3, 7)
cell_set, car_set, key = create_cell_and_car_set(key_seed = 12, road_shape =␣
↪shape, num_cars = 10, num_active_cars = 0, direction = direction)

# Sim & print
_, car_positions_over_time = simulate(cell_set, car_set, key, shape = shape,␣
↪direction = direction, lane_dim = lane_dim, num_iter = 16)
print_car_positions_sequence(car_positions_over_time, shape[0], shape[1], not␣
↪lane_dim)
```

```
Simulation at t=0
                -Added 1 car(s)
Road at timestep t=0
                -Active state: [1 0 0 0 0 0 0 0 0 0]
                -Cell ids: [ 2 -1 -1 -1 -1 -1 -1 -1 -1 -1]
                -Requested cell ids: [ 4 -1 -1 -1 -1 -1 -1 -1 -1 -1]
                -Destination cell ids: [18 -1 -1 -1 -1 -1 -1 -1 -1 -1]
                -0 cars arrived at their destination.
                -Sorted indices after removal: [0 1 2 3 4 5 6 7 8 9]
                -Cell X coord:  [0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2]
                -Cell Y coord:  [0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6]
                -Cell ids:      [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
16 17 18 19 20]
                -Chosen cars:   [-1 -1  0 -1  0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1]
Simulation at t=1
                -Added 0 car(s)
                -Active state: [1 0 0 0 0 0 0 0 0 0]
                -Cell ids: [ 4 -1 -1 -1 -1 -1 -1 -1 -1 -1]
                -Requested cell ids: [ 8 -1 -1 -1 -1 -1 -1 -1 -1 -1]
                -Destination cell ids: [18 -1 -1 -1 -1 -1 -1 -1 -1 -1]
                -0 cars arrived at their destination.
                -Sorted indices after removal: [0 1 2 3 4 5 6 7 8 9]
                -Cell X coord:  [0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2]
                -Cell Y coord:  [0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6]
                -Cell ids:      [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
16 17 18 19 20]
                -Chosen cars:   [-1 -1 -1 -1  0  0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1]
Simulation at t=2
                -Added 1 car(s)
                -Active state: [1 1 0 0 0 0 0 0 0 0]
                -Cell ids: [ 4  2 -1 -1 -1 -1 -1 -1 -1 -1]
                -Requested cell ids: [ 7  4 -1 -1 -1 -1 -1 -1 -1 -1]
                -Destination cell ids: [18 20 -1 -1 -1 -1 -1 -1 -1 -1]
                -0 cars arrived at their destination.
                -Sorted indices after removal: [0 1 2 3 4 5 6 7 8 9]
                -Cell X coord:  [0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2]
                -Cell Y coord:  [0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6]
                -Cell ids:      [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
16 17 18 19 20]
                -Chosen cars:   [-1 -1  1 -1  1  0 -1  0 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1]
Simulation at t=3
                -Added 0 car(s)
                -Active state: [1 1 0 0 0 0 0 0 0 0]
                -Cell ids: [ 7  4 -1 -1 -1 -1 -1 -1 -1 -1]
                -Requested cell ids: [11  6 -1 -1 -1 -1 -1 -1 -1 -1]
                -Destination cell ids: [18 20 -1 -1 -1 -1 -1 -1 -1 -1]
                -0 cars arrived at their destination.
                -Sorted indices after removal: [0 1 2 3 4 5 6 7 8 9]
                -Cell X coord:  [0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2]
                -Cell Y coord:  [0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6]
                -Cell ids:      [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
16 17 18 19 20]
                -Chosen cars:   [-1 -1 -1 -1  1  1 -1  0  0 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1]
```

```
Simulation at t=4
                -Added 1 car(s)
                -Active state: [1 1 1 0 0 0 0 0 0 0]
                -Cell ids: [ 7  4  2 -1 -1 -1 -1 -1 -1 -1]
                -Requested cell ids: [ 9  6  4 -1 -1 -1 -1 -1 -1 -1]
                -Destination cell ids: [18 20 18 -1 -1 -1 -1 -1 -1 -1]
                -0 cars arrived at their destination.
                -Sorted indices after removal: [0 1 2 3 4 5 6 7 8 9]
                -Cell X coord:  [0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2]
                -Cell Y coord:  [0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6]
                -Cell ids:      [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
16 17 18 19 20]
                -Chosen cars:   [-1 -1  2 -1  1  1  1  0  0  0 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1]
Simulation at t=5
                -Added 0 car(s)
                -Active state: [1 1 1 0 0 0 0 0 0 0]
                -Cell ids: [ 9  6  2 -1 -1 -1 -1 -1 -1 -1]
                -Requested cell ids: [12  9  4 -1 -1 -1 -1 -1 -1 -1]
                -Destination cell ids: [18 20 18 -1 -1 -1 -1 -1 -1 -1]
                -0 cars arrived at their destination.
                -Sorted indices after removal: [0 1 2 3 4 5 6 7 8 9]
                -Cell X coord:  [0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2]
                -Cell Y coord:  [0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6]
                -Cell ids:      [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
16 17 18 19 20]
                -Chosen cars:   [-1 -1  2 -1  2  1  1  1  0  0  0 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1]
Simulation at t=6
                -Added 0 car(s)
                -Active state: [1 1 1 0 0 0 0 0 0 0]
                -Cell ids: [ 9  6  4 -1 -1 -1 -1 -1 -1 -1]
                -Requested cell ids: [13  9  7 -1 -1 -1 -1 -1 -1 -1]
                -Destination cell ids: [18 20 18 -1 -1 -1 -1 -1 -1 -1]
                -0 cars arrived at their destination.
                -Sorted indices after removal: [0 1 2 3 4 5 6 7 8 9]
                -Cell X coord:  [0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2]
                -Cell Y coord:  [0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6]
                -Cell ids:      [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
16 17 18 19 20]
                -Chosen cars:   [-1 -1 -1 -1  2  2  1  1  1  1  0  0 -1  0 -1 -1
-1 -1 -1 -1 -1]
Simulation at t=7
                -Added 1 car(s)
                -Active state: [1 1 1 1 0 0 0 0 0 0]
                -Cell ids: [13  9  4  2 -1 -1 -1 -1 -1 -1]
                -Requested cell ids: [16 12  8  4 -1 -1 -1 -1 -1 -1]
                -Destination cell ids: [18 20 18 18 -1 -1 -1 -1 -1 -1]
                -0 cars arrived at their destination.
                -Sorted indices after removal: [0 1 2 3 4 5 6 7 8 9]
                -Cell X coord:  [0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2]
                -Cell Y coord:  [0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6]
                -Cell ids:      [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
16 17 18 19 20]
                -Chosen cars:   [-1 -1  3 -1  2  2 -1  1  2  1  1  0 -1  0  0 -1
-1 -1 -1 -1 -1]
Simulation at t=8
                -Added 0 car(s)
                -Active state: [1 1 1 1 0 0 0 0 0 0]
                -Cell ids: [13  9  8  2 -1 -1 -1 -1 -1 -1]
                -Requested cell ids: [16 13 10  4 -1 -1 -1 -1 -1 -1]
                -Destination cell ids: [18 20 18 18 -1 -1 -1 -1 -1 -1]
                -0 cars arrived at their destination.
                -Sorted indices after removal: [0 1 2 3 4 5 6 7 8 9]
                -Cell X coord:  [0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2]
                -Cell Y coord:  [0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6]
                -Cell ids:      [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
16 17 18 19 20]
                -Chosen cars:   [-1 -1  3 -1  3  2 -1 -1  1  1  1  1 -1  1  0 -1
0 -1 -1 -1 -1]
Simulation at t=9
                -Added 0 car(s)
                -Active state: [1 1 1 1 0 0 0 0 0 0]
                -Cell ids: [16 13  8  4 -1 -1 -1 -1 -1 -1]
                -Requested cell ids: [18 16 11  7 -1 -1 -1 -1 -1 -1]
                -Destination cell ids: [18 20 18 18 -1 -1 -1 -1 -1 -1]
                -0 cars arrived at their destination.
                -Sorted indices after removal: [0 1 2 3 4 5 6 7 8 9]
                -Cell X coord:  [0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2]
                -Cell Y coord:  [0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6]
                -Cell ids:      [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
16 17 18 19 20]
                -Chosen cars:   [-1 -1 -1 -1  3  3 -1 -1 -1 -1  1  1 -1  1  1 -1
0  0 -1 -1 -1]
Simulation at t=10
                -Added 1 car(s)
                -Active state: [1 1 1 1 1 0 0 0 0 0]
                -Cell ids: [16 13  8  4  2 -1 -1 -1 -1 -1]
                -Requested cell ids: [18 16 11  7  4 -1 -1 -1 -1 -1]
                -Destination cell ids: [18 20 18 18 19 -1 -1 -1 -1 -1]
                -0 cars arrived at their destination.
                -Sorted indices after removal: [0 1 2 3 4 5 6 7 8 9]
                -Cell X coord:  [0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2]
                -Cell Y coord:  [0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6]
                -Cell ids:      [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
16 17 18 19 20]
                -Chosen cars:   [-1 -1  4 -1  3  3 -1  3 -1 -1 -1  1 -1  1  1 -1
1  0  0 -1 -1]
Simulation at t=11
                -Added 0 car(s)
                -Active state: [1 1 1 1 1 0 0 0 0 0]
                -Cell ids: [18 16  8  7  2 -1 -1 -1 -1 -1]
                -Requested cell ids: [-1 20 11 10  4 -1 -1 -1 -1 -1]
                -Destination cell ids: [18 20 18 18 19 -1 -1 -1 -1 -1]
                -1 cars arrived at their destination.
                -Sorted indices after removal: [1 2 3 4 0 5 6 7 8 9]
                -Cell X coord:  [0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2]
                -Cell Y coord:  [0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6]
                -Cell ids:      [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
16 17 18 19 20]
                -Chosen cars:   [-1 -1  4 -1  4  3 -1  3  3 -1 -1 -1 -1 -1  1 -1
1  1 -1  0 -1]
Simulation at t=12
                -Added 0 car(s)
                -Active state: [1 1 1 1 0 0 0 0 0 0]
                -Cell ids: [16  8  7  4 -1 -1 -1 -1 -1 -1]
                -Requested cell ids: [20 11 11  6 -1 -1 -1 -1 -1 -1]
                -Destination cell ids: [20 18 18 19 -1 -1 -1 -1 -1 -1]
                -0 cars arrived at their destination.
                -Sorted indices after removal: [0 1 2 3 4 5 6 7 8 9]
                -Cell X coord:  [0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2]
                -Cell Y coord:  [0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6]
                -Cell ids:      [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
16 17 18 19 20]
                -Chosen cars:   [-1 -1 -1 -1  4  4 -1  3  3 -1 -1  3 -1 -1 -1 -1
1  1 -1 -1  1]
Simulation at t=13
                -Added 1 car(s)
                -Active state: [1 1 1 1 1 0 0 0 0 0]
                -Cell ids: [20  8 11  4  2 -1 -1 -1 -1 -1]
                -Requested cell ids: [-1 11 13  8  5 -1 -1 -1 -1 -1]
                -Destination cell ids: [20 18 18 19 18 -1 -1 -1 -1 -1]
                -1 cars arrived at their destination.
                -Sorted indices after removal: [1 2 3 4 0 5 6 7 8 9]
                -Cell X coord:  [0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2]
                -Cell Y coord:  [0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6]
                -Cell ids:      [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
16 17 18 19 20]
                -Chosen cars:   [-1 -1  0 -1  4  0 -1 -1  4 -1 -1  3 -1 -1 -1 -1
-1  1 -1 -1 -1]
Simulation at t=14
                -Added 0 car(s)
                -Active state: [1 1 1 1 0 0 0 0 0 0]
                -Cell ids: [ 8 11  8  5 -1 -1 -1 -1 -1 -1]
                -Requested cell ids: [10 13 11  7 -1 -1 -1 -1 -1 -1]
                -Destination cell ids: [18 18 19 18 -1 -1 -1 -1 -1 -1]
                -0 cars arrived at their destination.
                -Sorted indices after removal: [0 1 2 3 4 5 6 7 8 9]
                -Cell X coord:  [0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2]
                -Cell Y coord:  [0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6]
                -Cell ids:      [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
16 17 18 19 20]
                -Chosen cars:   [-1 -1 -1 -1 -1  4 -1 -1  4 -1 -1  3 -1  3 -1 -1
-1 -1 -1 -1 -1]
Simulation at t=15
                -Added 1 car(s)
                -Active state: [1 1 1 1 1 0 0 0 0 0]
                -Cell ids: [ 8 13  8  5  2 -1 -1 -1 -1 -1]
                -Requested cell ids: [11 15 11  7  4 -1 -1 -1 -1 -1]
                -Destination cell ids: [18 18 19 18 20 -1 -1 -1 -1 -1]
                -0 cars arrived at their destination.
                -Sorted indices after removal: [0 1 2 3 4 5 6 7 8 9]
                -Cell X coord:  [0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2]
                -Cell Y coord:  [0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6]
                -Cell ids:      [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
16 17 18 19 20]
                -Chosen cars:   [-1 -1  1 -1  1 -1 -1 -1  4 -1 -1  4 -1  3  3 -1
-1 -1 -1 -1 -1]
         .  .  .
         .  .  .
         .  .  .
         .  .  .
         .  .  .
         .  .  .
         .  .  0
Road at timestep t=1
         .  .  .
         .  .  .
         .  .  .
         .  .  .
         .  .  .
         .  0  .
         .  .  .
Road at timestep t=2
         .  .  .
         .  .  .
         .  .  .
         .  .  .
         .  .  .
         .  0  .
         .  .  1
Road at timestep t=3
         .  .  .
         .  .  .
         .  .  .
         .  .  .
         .  0  .
         .  1  .
         .  .  .
Road at timestep t=4
         .  .  .
         .  .  .
         .  .  .
         .  .  .
         .  0  .
         .  1  .
         .  .  2
Road at timestep t=5
         .  .  .
         .  .  .
         .  .  .
         0  .  .
         1  .  .
         .  .  .
         .  .  2
Road at timestep t=6
         .  .  .
         .  .  .
         .  .  .
         0  .  .
         1  .  .
         .  2  .
         .  .  .
Road at timestep t=7
         .  .  .
         .  .  .
         .  0  .
         1  .  .
         .  .  .
         .  2  .
         .  .  3
Road at timestep t=8
         .  .  .
         .  .  .
         .  0  .
         1  .  .
         .  .  2
         .  .  .
         .  .  3
Road at timestep t=9
         .  .  .
         .  0  .
         .  1  .
         .  .  .
         .  .  2
         .  3  .
         .  .  .
Road at timestep t=10
         .  .  .
         .  0  .
         .  1  .
         .  .  .
         .  .  2
         .  3  .
         .  .  4
```

```
Road at timestep t=11
        .  .  .
        .  0  .
        .  .  .
        .  .  .
        .  2  1
        .  .  .
        .  .  3
Road at timestep t=12
        .  .  .
        .  0  .
        .  .  .
        .  .  .
        .  2  1
        .  3  .
        .  .  .
Road at timestep t=13
        .  .  .
        .  .  .
        .  .  .
        .  .  1
        .  .  0
        .  2  .
        .  .  3
Road at timestep t=14
        .  .  .
        .  .  .
        .  .  .
        .  .  1
        .  .  2
        .  .  3
        .  .  .
Road at timestep t=15
        .  .  .
        .  .  .
        .  1  .
        .  .  .
        .  .  2
        .  .  3
        .  .  4
```

6.3.5 Intersection simulation (WIP)

[15]:
```python
# Intersection helper functions
def print_intersection_car_positions(road_cars: jnp.array, X: int, Y: int):
    """
    Print the positions of cars on an 8-road intersection in a cross shape.
    Args:
        road_cars: jnp.array of shape (8, T), each row a road of car cell IDs.
        X: int, short road dim (3)
        Y: int, long road dim (7)
    """
    moves_vertical_flags = [True, False, False, True, True, False, False,␣
↪True]
    road_grids = []

    for road_id in range(8):
        road = road_cars[road_id]
        mv = moves_vertical_flags[road_id]

        width = X if mv else Y
        height = Y if mv else X
        grid = [" ."] * (width * height)

        for car_id, cell_id in enumerate(road):
            if cell_id != -1:
                grid[int(cell_id)] = f"{car_id:2}"

        road_grid = []
        for row in reversed(range(height)):
            row_cells = []
            for col in range(width):
                idx = row * width + col if mv else col * height + row
                row_cells.append(grid[idx])
            road_grid.append(row_cells)
        road_grids.append(road_grid)

    # Centered padding for vertical roads (4, 3, 7, 0)
    side_pad = ["  "] * (Y)
    # Top verticals: roads 4 and 3
    for r in range(Y):
        row = (
            side_pad +
            road_grids[4][r] +
            ["  "] +
            road_grids[3][r] +
            side_pad
        )
        print("".join(row))

    # Top horizontal: road 5 and 2
    for r in range(X):
        row = (
            road_grids[5][r] +
            ["  "] * 7 +
            road_grids[2][r]
        )
        print("".join(row))

    print("")  # visual spacer between horizontal bands

    # Bottom horizontal: road 6 and 1
    for r in range(X):
        row = (
            road_grids[6][r] +
            ["  "] * 7 +
            road_grids[1][r]
        )
        print("".join(row))

    # Bottom verticals: roads 7 and 0
    for r in range(Y):
        row = (
            side_pad +
            road_grids[7][r] +
            ["  "] +
            road_grids[0][r] +
            side_pad
        )
        print("".join(row))
def intersection_movement(road_cars: jnp.array, X: int, Y: int):
    """
    Print the positions of cars on an 8-road intersection in a cross shape,␣
 ↪across time.

    Args:
        road_cars: jnp.array of shape (8, num_cars, T), each␣
 ↪[road][car][time] has a cell ID or -1.
        X: int, short dimension of the road (3).
        Y: int, long dimension of the road (7).
    """
    moves_vertical_flags = [True, False, False, True, True, False, False,␣
 ↪True]
    _, num_cars, T = road_cars.shape

    for t in range(T):
        print(f"\nIntersection at time t={t}\n")

        road_grids = []

        for road_id in range(8):
            mv = moves_vertical_flags[road_id]
            width = X if mv else Y
            height = Y if mv else X
            grid = [" ."] * (width * height)

            for car_id in range(num_cars):
                cell_id = road_cars[road_id, car_id, t]
                if cell_id != -1:
                    grid[int(cell_id)] = f"{car_id:2}"

            road_grid = []
            for row in reversed(range(height)):
                row_cells = []
                for col in range(width):
                    idx = row * width + col if mv else col * height + row
                    row_cells.append(grid[idx])
                road_grid.append(row_cells)
            road_grids.append(road_grid)

        # Side padding to center vertical roads
        side_pad = ["  "] * (Y)

        # Top vertical roads: 4 and 3
        for r in range(Y):
            row = (
                side_pad +
                road_grids[4][r] +
                ["  "] +
                road_grids[3][r] +
                side_pad
            )
            print("".join(row))

        # Top horizontal roads: 5 and 2
        for r in range(X):
            row = (
                road_grids[5][r] +
                ["  "] * 7 +
                road_grids[2][r]
            )
            print("".join(row))

        print("")

        # Bottom horizontal roads: 6 and 1
        for r in range(X):
            row = (
                road_grids[6][r] +
                ["  "] * 7 +
                road_grids[1][r]
            )
            print("".join(row))

        # Bottom vertical roads: 7 and 0
        for r in range(Y):
            row = (
                side_pad +
                road_grids[7][r] +
                ["  "] +
                road_grids[0][r] +
                side_pad
            )
```

```
            print("".join(row))

# Setup
x, y = (3, 7)
road_car_positions = jnp.tile(jnp.array([2, 6, 7, 18, 1, 11, 15, 9, 16, 5]),␣
 ↪8).reshape(8, 10)
big_stack = jnp.tile(car_positions_over_time, 8).reshape(8, 10, -1)
print_intersection_car_positions(road_car_positions, x, y)
intersection_movement(big_stack, x, y)
```

```
               3 . .   3 . .
               6 8 .   6 8 .
               . . .   . . .
               7 . 5   7 . 5
               1 2 .   1 2 .
               . . 9   . . 9
               . 4 0   . 4 0
 0 9 . 5 . . .               0 9 . 5 . . .
 4 . 2 . . 8 .               4 . 2 . . 8 .
 . . 1 7 . 6 3               . . 1 7 . 6 3

 0 9 . 5 . . .               0 9 . 5 . . .
 4 . 2 . . 8 .               4 . 2 . . 8 .
 . . 1 7 . 6 3               . . 1 7 . 6 3
               3 . .   3 . .
               6 8 .   6 8 .
               . . .   . . .
               7 . 5   7 . 5
               1 2 .   1 2 .
               . . 9   . . 9
               . 4 0   . 4 0

Intersection at time t=0

               . . .   . . .
               . 5 .   . . .
               . 0 .   . 5 .
               . . .   0 . .
               . . 7   . . .
               . . .   . 7 .
               . . .   . . .
 4 . 2 . . . .               7 . . . . . .
 . . 7 . . 5 .               . . 0 . . . .
 . . . . . . .               . . . 5 . . .

 . . 5 . . . .               . . . . . . .
 . 7 2 . . 0 .               . 0 5 . . . .
 . . . . . . .               . . . . . . .
               . . .   . . .
               . . .   . . .
               . . .   . . .
               . . .   . . .
               . . 7   . . .
               . . .   . 5 .
               . . 9   . . 0
```

Intersection at time t=1

```
               . . .   . . .
               . . .   . . .
               . 5 .   . . .
               0 . .   5 . .
               . . .   0 . .
               . 7 .   . . .
               . . 2   . . 7
 7 . 5 . . . .               . . . . . . .
 . 2 . . 0 . .               . 0 . . . . .
 . . . . . . .               . . 5 . . . .

 7 . 0 5 . . .               0 . . . . . .
 . 2 . . . . .               . 5 . . . . .
 . . . . . . .               . . . . . . .
               . . .   . . .
               . . .   . . .
               . 5 .   . . .
               . . 0   . . .
               . . .   . . .
               . . 7   . . .
               . . .   . . .
```

Intersection at time t=2

```
               . . .   . . .
               . 8 .   . . .
               . 3 .   . 8 .
               . . .   3 . .
               . . 5   . . .
               . . .   . 5 .
               . . .   . . .
 2 . 0 . . . .               5 . . . . . .
 . . 5 . . 8 .               . . 3 . . . .
 . . . . . . .               . . . 8 . . .

 . . 8 . . . .               . . . . . . .
 . 5 0 . . 3 .               . 3 8 . . . .
 . . . . . . .               . . . . . . .
               . . .   . . .
               . . .   . . .
               . . .   . . .
               . . .   . . .
               . . 8   . . .
               . . .   . 8 .
               . . 7   . . 3

Intersection at time t=3

               . . .   . . .
               . . .   . . .
               . 8 .   . . .
               3 . .   8 . .
               . . .   3 . .
               . 5 .   . . .
               . . 0   . . 5
 5 . 8 . . . .               . . . . . . .
 . 0 . . 3 . .               . 3 . . . . .
 . . . . . . .               . . 8 . . . .

 5 . 3 8 . . .               3 . . . . . .
 . 0 . . . . .               . 8 . . . . .
 . . . . . . .               . . . . . . .
               . . .   . . .
               . . .   . . .
               . 8 .   . . .
               . . 3   . . .
               . . .   . . .
               . . 5   . . .
               . . .   . . .

Intersection at time t=4

               . . .   . . .
               . 6 .   . . .
               . 1 .   . 6 .
               . . .   1 . .
               . . 8   . . .
               . . .   . 8 .
               . . .   . . .
 0 . 3 . . . .               8 . . . . . .
 . . 8 . . 6 .               . . 1 . . . .
 . . . . . . .               . . . 6 . . .

 . . 6 . . . .               . . . . . . .
 . 8 3 . . 1 .               . 1 6 . . . .
 . . . . . . .               . . . . . . .
               . . .   . . .
               . . .   . . .
               . . .   . . .
               . . .   . . .
               . . 8   . . .
               . . .   . 6 .
               . . 5   . . 1

Intersection at time t=5
```

```
               . . .   . . .
               . . .   . . .
               . 6 .   . . .
               1 . .   6 . .
               . . .   1 . .
               . 8 .   . . .
               . . 3   . . 8
 8 . 6 . . . .               . . . . . . .
 . 3 . . 1 . .               . 1 . . . . .
 . . . . . . .               . . 6 . . . .

 8 . 1 6 . . .               1 . . . . . .
 . 3 . . . . .               . 6 . . . . .
 . . . . . . .               . . . . . . .
               . . .   . . .
               . . .   . . .
               . 6 .   . . .
               . . 1   . . .
               . . .   . . .
               . . 8   . . .
               . . .   . . .

Intersection at time t=6

               . . .   . . .
               . 9 .   . . .
               . 4 .   . 9 .
               . . .   4 . .
               . . 6   . . .
               . . .   . 6 .
               . . .   . . .
 3 . 1 . . . .               6 . . . . . .
 . . 6 . . 9 .               . . 4 . . . .
 . . . . . . .               . . . 9 . . .

 . . 9 . . . .               . . . . . . .
 . 6 1 . . 4 .               . 4 9 . . . .
 . . . . . . .               . . . . . . .
               . . .   . . .
               . . .   . . .
               . . .   . . .
               . . .   . . .
               . . 9   . . .
               . . .   . 9 .
               . . 8   . . 4

Intersection at time t=7

               . . .   . . .
               . . .   . . .
```

```
              . 9 .   . . .
              4 . .   9 . .
              . . .   4 . .
              . 6 .   . . .
              . . 1   . . 6
 6 . 9 . . . .              . . . . . . .
 . 1 . . 4 . .              . 4 . . . . .
 . . . . . . .              . . 9 . . . .

 6 . 4 9 . . .              4 . . . . . .
 . 1 . . . . .              . 9 . . . . .
 . . . . . . .              . . . . . . .
              . . .   . . .
              . . .   . . .
              . 9 .   . . .
              . . 4   . . .
              . . .   . . .
              . . 6   . . .
              . . .   . . .
```

```
Intersection at time t=8
```

```
              . . .   . . .
              . 7 .   . . .
              . 2 .   . 7 .
              . . .   2 . .
              . . 9   . . .
              . . .   . 9 .
              . . .   . . .
 1 . 4 . . . .              9 . . . . . .
 . . 9 . . 7 .              . . 2 . . . .
 . . . . . . .              . . . 7 . . .

 . . 7 . . . .              . . . . . . .
 . 9 4 . . 2 .              . 2 7 . . . .
 . . . . . . .              . . . . . . .
              . . .   . . .
              . . .   . . .
              . . .   . . .
              . . .   . . .
              . . 9   . . .
              . . .   . 7 .
              . . 6   . . 2
```

```
Intersection at time t=9
```

```
              . . .   . . .
              . . .   . . .
              . 7 .   . . .
              2 . .   7 . .
              . . .   2 . .
```

```
               . 9 .   . . .
               . . 4   . . 9
 9 . 7 . . . .               . . . . . . .
 . 4 . . 2 . .               . 2 . . . . .
 . . . . . . .               . . 7 . . . .

 9 . 2 7 . . .               2 . . . . . .
 . 4 . . . . .               . 7 . . . . .
 . . . . . . .               . . . . . . .
               . . .   . . .
               . . .   . . .
               . 7 .   . . .
               . . 2   . . .
               . . .   . . .
               . . 9   . . .
               . . .   . . .

Intersection at time t=10

               . . .   . . .
               . 5 .   . . .
               . 0 .   . 5 .
               . . .   0 . .
               . . 7   . . .
               . . .   . 7 .
               . . .   . . .
 4 . 2 . . . .               7 . . . . . .
 . . 7 . . 5 .               . . 0 . . . .
 . . . . . . .               . . . 5 . . .

 . . 5 . . . .               . . . . . . .
 . 7 2 . . 0 .               . 0 5 . . . .
 . . . . . . .               . . . . . . .
               . . .   . . .
               . . .   . . .
               . . .   . . .
               . . .   . . .
               . . 7   . . .
               . . .   . 5 .
               . . 9   . . 0

Intersection at time t=11

               . . .   . . .
               . . .   . . .
               . 5 .   . . .
               0 . .   5 . .
               . . .   0 . .
               . 7 .   . . .
               . . 2   . . 7
 7 . 5 . . . .               . . . . . . .
 . 2 . . 0 . .               . 0 . . . . .
 . . . . . . .               . . 5 . . . .

 7 . 0 5 . . .               0 . . . . . .
 . 2 . . . . .               . 5 . . . . .
 . . . . . . .               . . . . . . .
               . . .   . . .
               . . .   . . .
               . 5 .   . . .
               . . 0   . . .
               . . .   . . .
               . . 7   . . .
               . . .   . . .

Intersection at time t=12

               . . .   . . .
               . 8 .   . . .
               . 3 .   . 8 .
               . . .   3 . .
               . . 5   . . .
               . . .   . 5 .
               . . .   . . .
 2 . 0 . . . .               5 . . . . . .
 . . 5 . . 8 .               . . 3 . . . .
 . . . . . . .               . . . 8 . . .

 . . 8 . . . .               . . . . . . .
 . 5 0 . . 3 .               . 3 8 . . . .
 . . . . . . .               . . . . . . .
               . . .   . . .
               . . .   . . .
               . . .   . . .
               . . .   . . .
               . . 8   . . .
               . . .   . 8 .
               . . 7   . . 3

Intersection at time t=13

               . . .   . . .
               . . .   . . .
               . 8 .   . . .
               3 . .   8 . .
               . . .   3 . .
               . 5 .   . . .
               . . 0   . . 5
 5 . 8 . . . .               . . . . . . .
 . 0 . . 3 . .               . 3 . . . . .
 . . . . . . .               . . 8 . . . .
 5 . 3 8 . . .               3 . . . . . .
 . 0 . . . . .               . 8 . . . . .
 . . . . . . .               . . . . . . .
               . . .   . . .
               . . .   . . .
               . 8 .   . . .
               . . 3   . . .
               . . .   . . .
               . . 5   . . .
               . . .   . . .

Intersection at time t=14

               . . .   . . .
               . 6 .   . . .
               . 1 .   . 6 .
               . . .   1 . .
               . . 8   . . .
               . . .   . 8 .
               . . .   . . .
 0 . 3 . . . .               8 . . . . . .
 . . 8 . . 6 .               . . 1 . . . .
 . . . . . . .               . . . 6 . . .

 . . 6 . . . .               . . . . . . .
 . 8 3 . . 1 .               . 1 6 . . . .
 . . . . . . .               . . . . . . .
               . . .   . . .
               . . .   . . .
               . . .   . . .
               . . .   . . .
               . . 8   . . .
               . . .   . 6 .
               . . 5   . . 1

Intersection at time t=15

               . . .   . . .
               . . .   . . .
               . 6 .   . . .
               1 . .   6 . .
               . . .   1 . .
               . 8 .   . . .
               . . 3   . . 8
 8 . 6 . . . .               . . . . . . .
 . 3 . . 1 . .               . 1 . . . . .
 . . . . . . .               . . 6 . . . .

 8 . 1 6 . . .               1 . . . . . .
 . 3 . . . . .               . 6 . . . . .
 . . . . . . .               . . . . . . .
. . .   . . .
. . .   . . .
. 6 .   . . .
. . 1   . . .
. . .   . . .
. . 8   . . .
. . .   . . .
```

[16]:
```
'''
roads = jnp.zeros(8, Set)
cars_on_roads = jnp.zeros(8, Set)

directions = jnp.zeros(8, jnp.int32)
'''

def intersection():
    num_lanes = 3
    road_length = 7
    road_key_bases = random.randint(key=jax.random.PRNGKey(4), shape=(8,),␣
 ↪minval=0, maxval=1024)

    directions = jnp.arange(4)
    vertical_shape = (num_lanes, road_length)
    horizontal_shape = (road_length, num_lanes)

    # Create all sets
    in_up_cell_set, in_up_car_set, iu_key = create_cell_and_car_set(key_seed␣
 ↪= road_key_bases[0], road_shape = vertical_shape, num_cars = 10,␣
 ↪num_active_cars = 0, direction = directions[0]) # up
    out_right_cell_set, out_right_car_set, or_key =␣
 ↪create_cell_and_car_set(key_seed = road_key_bases[1], road_shape =␣
 ↪horizontal_shape, num_cars = 10, num_active_cars = 0, direction =␣
 ↪directions[3]) # right
    in_left_cell_set, in_left_car_set, il_key =␣
 ↪create_cell_and_car_set(key_seed = road_key_bases[2], road_shape =␣
 ↪horizontal_shape, num_cars = 10, num_active_cars = 0, direction =␣
 ↪directions[1]) # left
    out_up_cell_set, out_up_car_set, ou_key =␣
 ↪create_cell_and_car_set(key_seed = road_key_bases[3], road_shape =␣
 ↪vertical_shape, num_cars = 10, num_active_cars = 0, direction =␣
 ↪directions[0]) # up
    in_down_cell_set, in_down_car_set, id_key =␣
 ↪create_cell_and_car_set(key_seed = road_key_bases[4], road_shape =␣
 ↪vertical_shape, num_cars = 10, num_active_cars = 0, direction =␣
 ↪directions[2]) # down
    out_left_cell_set, out_left_car_set, ol_key =␣
 ↪create_cell_and_car_set(key_seed = road_key_bases[5], road_shape =␣
 ↪horizontal_shape, num_cars = 10, num_active_cars = 0, direction =␣
 ↪directions[1]) # left
```

```
    in_right_cell_set, in_right_car_set, ir_key =␣
↪create_cell_and_car_set(key_seed = road_key_bases[6], road_shape =␣
↪horizontal_shape, num_cars = 10, num_active_cars = 0, direction =␣
↪directions[3]) # right
    out_down_cell_set, out_down_car_set, od_key =␣
↪create_cell_and_car_set(key_seed = road_key_bases[7], road_shape =␣
↪vertical_shape, num_cars = 10, num_active_cars = 0, direction =␣
↪directions[2]) # down

    road_keys = jnp.array([iu_key, or_key, il_key, ou_key, id_key, ol_key,␣
↪ir_key, od_key]) # As keys are ints, they can be in an array

    # Generate car_ids, entry road, spawn cell, exit road, destination
    num_entry_exit_cells = num_lanes * 4 # We have one entry in each lane,␣
↪and four entry roads

    def cars_in_entries():
        num_lanes_aranged = jnp.arange(num_entry_exit_cells) # Used for ids.␣
↪# Replace X_max_value with X_max[0]

        # Find 4 sets of entries (ind 0, 2, 4, 6) and exits (ind 1, 3, 5, 7)␣
↪shapes of both arrays: (4, 21)
        entry_cell_ids = jnp.array([(2 * in_up_cell_set.agents.params.
↪content['entry'].flatten() - 1) * in_up_cell_set.agents.id, (2 *␣
↪in_left_cell_set.agents.params.content['entry'].flatten() - 1) *␣
↪in_left_cell_set.agents.id, (2 * in_down_cell_set.agents.params.
↪content['entry'].flatten() - 1) * in_down_cell_set.agents.id, (2 *␣
↪in_right_cell_set.agents.params.content['entry'].flatten() - 1) *␣
↪in_right_cell_set.agents.id])
        exit_cell_ids = jnp.array([(2 * out_right_cell_set.agents.params.
↪content['exit'].flatten() - 1) * out_right_cell_set.agents.id, (2 *␣
↪out_up_cell_set.agents.params.content['exit'].flatten() - 1) *␣
↪out_up_cell_set.agents.id, (2 * out_left_cell_set.agents.params.
↪content['exit'].flatten() - 1) * out_left_cell_set.agents.id, (2 *␣
↪out_down_cell_set.agents.params.content['exit'].flatten() - 1) *␣
↪out_down_cell_set.agents.id])

        entry_cell_ids = jnp.sort(entry_cell_ids, axis=0)[:, :
↪num_entry_exit_cells]
        exit_cell_ids = jnp.sort(exit_cell_ids, axis=0)[:, :
↪num_entry_exit_cells]

        #car_indx = num_lanes_aranged.copy() + num_active_cars #TODO Figure␣
↪out what to do with car id/car indx
        key, *spawn_car_keys = random.split(key, 9)

        # Shuffling the entry Cells.
        entry_cell_ids = jax.vmap(jax.random.permutation, in_axes=(0, 0))(jnp.
↪stack(spawn_car_keys[:4]), entry_cell_ids) # Generating independently␣
↪random permutations for entry cell ids
```

```
        '''
        The rest of the spawn_cars_in_entry_cells function

        # Taking the num_cars from the shuffled entry Cells.
        num_cars_entry_cells = jnp.take(cells.state.content['num_cars'],␣
↪entry_cell_ids)
        is_cell_available = jnp.where(num_cars_entry_cells == 0, 1, 0) #␣
↪Check if the entry Cell is available: 1-> available, 0-> not available.
        current_cell_idx = entry_cell_ids.copy()
        current_cell_idx = jnp.argsort(-1*is_cell_available) # Sort in␣
↪descending order, so that the available Cells are at the beginning.

        # Sorting the entry Cells based on whether they have free spots.
        entry_cell_ids = jnp.take(entry_cell_ids, current_cell_idx)

        # Sorting the car_ids based on the availability of cells.
        idx = jnp.argsort(entry_cell_ids)
        car_indx = jnp.take(car_indx, idx)
        car_ids = jnp.take(cars.id, car_indx) # car_ids are different from␣
↪indexes, so we need to take the car_ids from the cars agent.

        # Determining the exit Cell ids for the Cars that will be added, just␣
↪randomly choose num_lanes exit Cell ids.
        num_cells = X_max * Y_max
        car_exit_cell_ids = jax.random.choice(key=spawn_car_keys[1],
                                             a=exit_cell_ids,
                                             shape=(num_entry_exit_cells,),
                                             replace=True)

        # Determining how many cars to add: 1 to number of lanes and never␣
↪more than total number of available Cell entries.
        num_cars_to_add = jax.random.randint(spawn_car_keys[2], (1,),␣
↪minval=1, maxval=num_entry_exit_cells+1) # Number of cars to add,␣
↪everything heavily relies on the fact that the for-loops will only go to␣
↪this number, not further.
        num_cars_to_add = jnp.minimum(num_cars_to_add[0], jnp.
↪sum(is_cell_available)) # Make sure that the number of cars to add is less␣
↪than the number of available cells.

        # Package and return
        car_add_params = Params(content={'current_cell_id': entry_cell_ids,␣
↪'destination_cell_id': car_exit_cell_ids, 'num_active_agents':␣
↪num_active_cars, 'dt': jnp.array([1.0])})
        cell_set_params = Params(content={'set_indx': entry_cell_ids,␣
↪'car_id': car_ids})

        return car_add_params, cell_set_params, num_cars_to_add, key
        '''
```

```


    # Determine number of Cars added to simulation
    #car_set.agents.state.content['current_cell_id'].reshape(-1)


intersection()
```