Cellular Automata Model of Traffic

1.1 A First CA Model

In this section, a CA model will be used to model simple traffic flow on a freeway. This model will consider a one-way lane, decomposed into cells of length 7.5m. Each cell will have a state of either "car" or "no car," depending on whether or not a car currently occupies the cell. Each car will have a velocity, in units of cells per timestep. A maximum velocity of 5 cells/timestep will be used. A timestep of 1 second will be used. This model will be based on the assumptions that there is an abscence of vehicle collisions and a conservation of vehicles. In order to enforce an absence of collisions, cars will be able to decelerate with no time lag. In order to enforce a constant number of cars, a periodic boundary condition will be used.
At each time step, every car will be updated in parallel using the following rules:

1. Accelerate: $v_i = min\{v_i + 1, v_{max}\}$
2. Decelerate: $v_i = d(i, i+1)$, if $v_i > d(i, i+1)$, where d(i, j) is the distance between vehicles i and j
3. Move: vehicle i moves forward v_i cells

Therefore, a vehicle will always attempt to accelerate to the maximum velocity, but will need to decelerate to avoid collisions.

The model will be initilized with 300 cells, corrresponding to a rode length of 2.25km. Vehicles will be initialized in a random location, with a random velocity between 0 and 5 cells/timestep.

Simulation Code

In [ ]:
```
import random
import copy
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import MultipleLocator

max_vel = 5
```

In [ ]:
```
class Cell:
  def __init__(self, state=0, velocity=-1):
    self.state = state
    self.velocity = velocity
  def __eq__(self, other):
    if isinstance(other, Cell):
      return self.state == other.state and self.velocity == other.velocity
    return False
```

In [ ]:
```
def initilize(num_cells, occupancy_percent):
  cells = [Cell() for _ in range(num_cells)]
  num_cars = (int)((occupancy_percent/100) * num_cells)
  indices = random.sample(range(num_cells), num_cars)
  for index in indices:
    cells[index].state = 1
    cells[index].velocity = random.randint(0, 5)
  return cells

def distance(cells, index, num_cells):
  for i in range(1, len(cells)):
      next_cell = cells[(i + index) % num_cells]
      if next_cell.state == 1:
        return i - 1

def accelerate(cell, max_vel):
  cell.velocity = min(cell.velocity + 1, max_vel)

def decelerate(cell, next_distance) :
  if cell.velocity > next_distance:
    cell.velocity = next_distance

def move(cells, num_cells):
  cells_copy = copy.deepcopy(cells)
  for index, cell in enumerate(cells_copy):
    if cell.state == 1:
      next_cell = (index + cell.velocity) % num_cells
      if cells[next_cell].state == 1 and index != next_cell:
        print(f"Collision at {next_cell}")
      cells[index].state = 0
```

```
      cells[index].velocity = -1
      cells[next_cell].state = 1
      cells[next_cell].velocity = cell.velocity

def update(cells, num_cells, max_vel, p=0):
  for index, cell in enumerate(cells):
    if cell.state == 1:
      next_distance = distance(cells, index, num_cells)
      accelerate(cell, max_vel)
      decelerate(cell, next_distance)
  move(cells, num_cells)

def add_impulse(cells, impulse):
  for index, cell in enumerate(cells):
    if cell.state == 1:
      cell.velocity = min(cell.velocity + impulse, max_vel)

def show(cells):
  cells_list = []
  velocities = []
  for cell in cells:
    velocities.append(cell.velocity)
    if cell.state == 1:
      cells_list.append("car")
    else:
      cells_list.append("empty")
  print(cells_list)
  print(velocities)
```

In [ ]:
```
def simulate(max_vel, occupancy_percent, num_cells, num_time_steps, p, impulse):
  cells = initilize(num_cells, occupancy_percent)
  velocities = np.zeros((num_time_steps, num_cells))
  for time in range(0, num_time_steps):
    update(cells, num_cells, max_vel, p)
    if impulse > 0 and time == 0:
      add_impulse(cells, impulse)
    for index, cell in enumerate(cells):
      velocities[time][index] = cell.velocity
  return velocities

def create_plot(max_vel, occupancy_percent, num_cells, num_time_steps, title, p, sim, gridlines, impulse = 0):
  velocities = simulate(max_vel, occupancy_percent, num_cells, num_time_steps, p, impulse)
  plt.figure(figsize=(12, 6))
  plt.imshow(velocities, cmap='Greys', aspect='auto', interpolation='nearest')
  plt.xlabel("Cell")
  plt.ylabel("Time")
  plt.title(title)
  plt.colorbar(label="Velocity")
  if gridlines:
    ax = plt.gca()
    ax.xaxis.set_major_locator(MultipleLocator(1))
    ax.grid(which='major', axis='x', linestyle='-', linewidth=0.5, color='k')
  if impulse > 0:
    plt.savefig(f'{occupancy_percent},{num_cells},{p},{sim},{impulse}.jpg')
  else:
    plt.savefig(f'{occupancy_percent},{num_cells},{p},{sim}.jpg')
  plt.close()
```

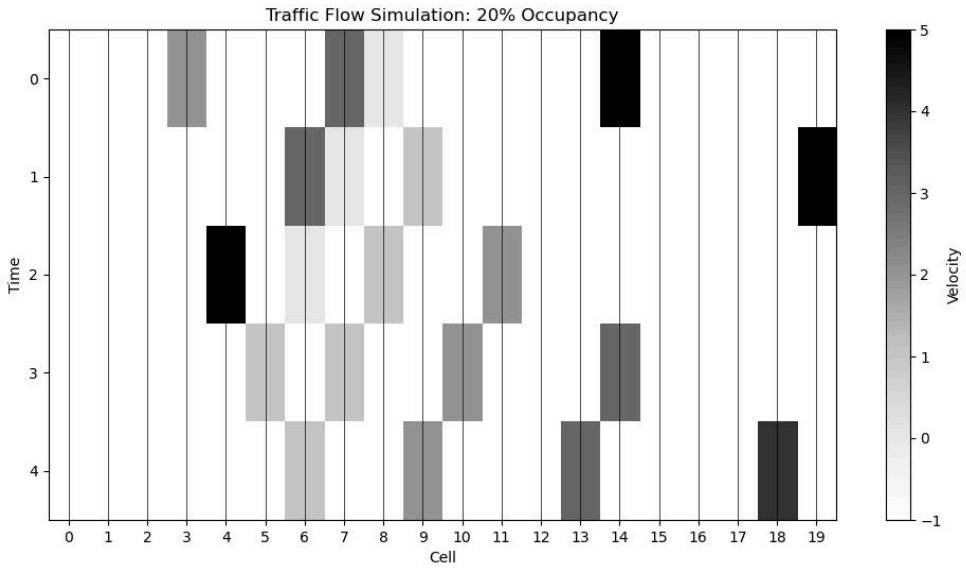
Note that an update step is performed before vehicle speeds are recorded, meaning the randomly initilized vehicles are not displayed. This is done because the random vehicles may exhibit inconsistent behavior, such as slower car directly behind a faster one. This behavior would become much more apparent in later sections (two lane road) so it was excluded.

1.1.1 Testing

First, to test that the model is exhibiting expected behavior, the following code will simulate a road with 20% initial occupancy, 20 cells, and for 5 time steps.

In [ ]:
```
sim = 1
occupancy_percent = 20
num_cells = 20
num_time_steps = 5

title = f"Traffic Flow Simulation: {occupancy_percent}% Occupancy"
create_plot(max_vel, occupancy_percent, num_cells, num_time_steps, title, 0, sim, True)
```



As can be seen, each vehicle accelerates when possible, and moves forward by an amount of cells equal to its velocity each time step. Additionally, each vehicle decelerates when necessary to avoid a collisions. As expected, both of the intital assumptions (that there are no collisions and there is a conservation of vehicles) hold for the duration of the simulation. Each vehicle also obeys the boundary conditions when necessary.
Note that a "velocity" of -1 is the default for a cell containing no car, and is signified by the color white in the above plot and all following plots.

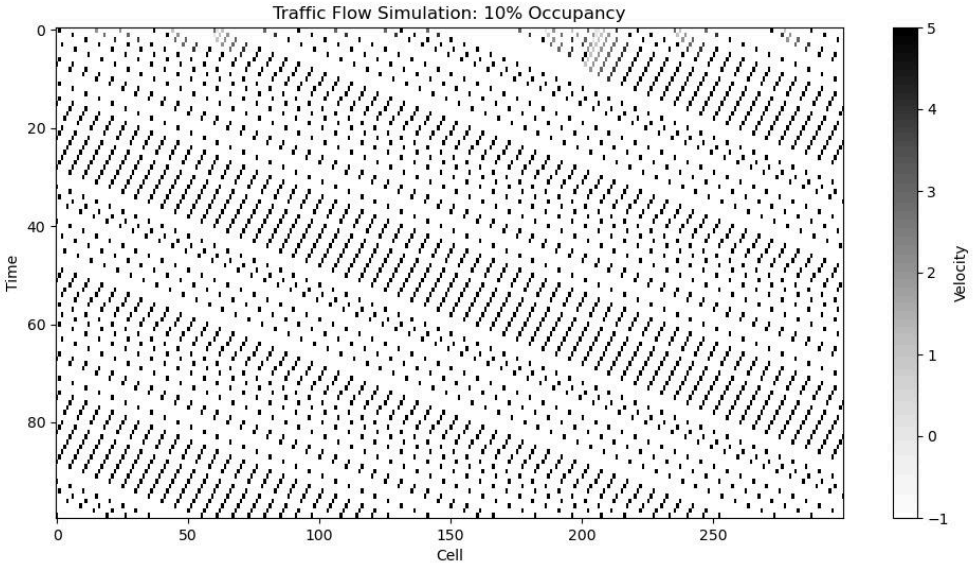
1.1.2 Experiments

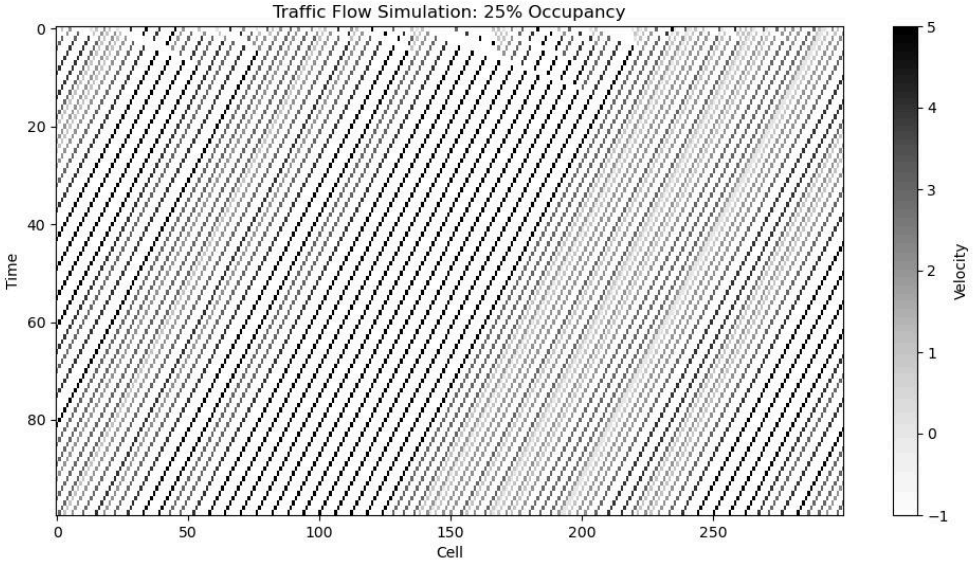
Now, the model will be used to perform a experiments. Traffic conditions will be observed at occupancy percents of 10%, 25%, 50%, and 80% for 100 time steps.

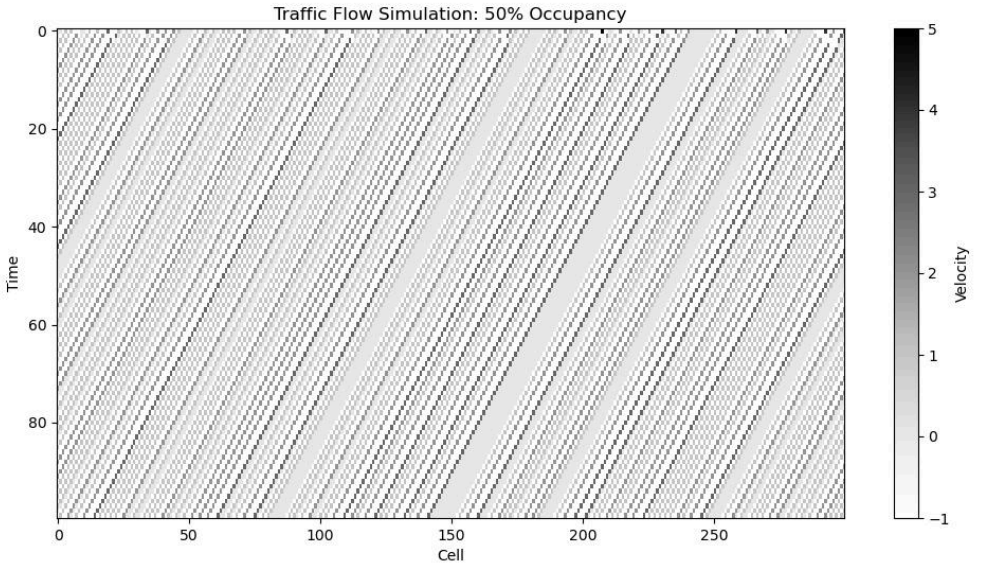
In [ ]:
```
def simulate_occupancy_percents(p, max_vel, num_cells, num_time_steps):
    occupancy_percents = [10, 25, 50, 80]
    for occupancy_percent in occupancy_percents:
        if p == 0:
            title = f"Traffic Flow Simulation: {occupancy_percent}% Occupancy"
        else:
            title = f"Traffic Flow Simulation: {occupancy_percent}% Occupancy, p = {p}"
        create_plot(max_vel, occupancy_percent, num_cells, num_time_steps, title, p, sim, False)
```

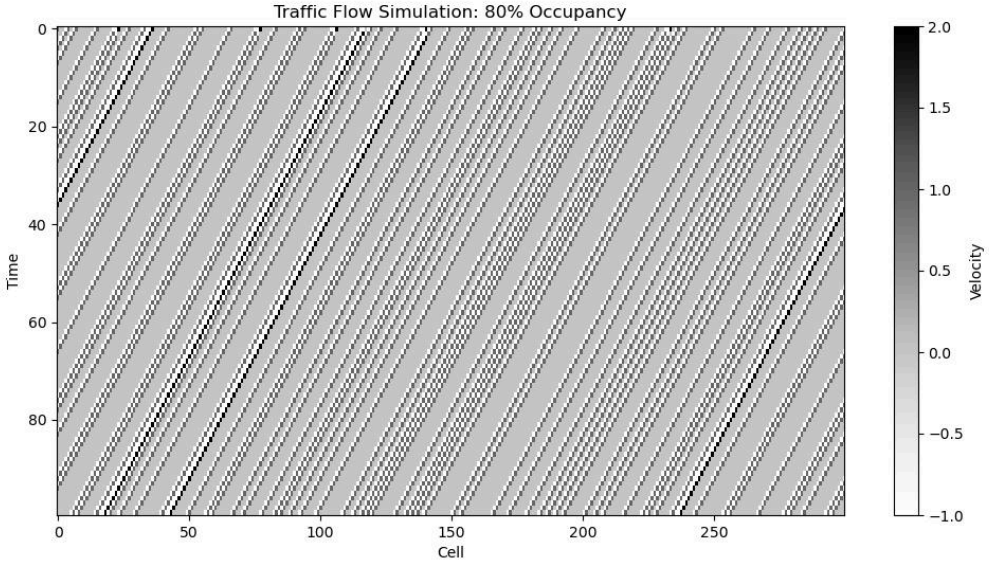
In [ ]:
```
num_cells = 300
num_time_steps = 100
occupancy_percents = [10, 25, 50, 80]

simulate_occupancy_percents(0, max_vel, num_cells, num_time_steps)
```


Traffic Flow Simulation: 10% Occupancy
Time
Cell
Velocity
0
20
40
60
80
0
50
100
150
200
250
5
4
3
2
1
0
−1


Traffic Flow Simulation: 25% Occupancy
Time
Cell
Velocity
0
20
40
60
80
0
50
100
150
200
250
5
4
3
2
1
0
−1





In these plots, each row represents a state of the CA model at a single time step. As expected based on the analysis in Bungartz et al. (2014), the plot of the model with an occupancy of 10% (which is less than the critical density of 22.2 veh/km or 16.7% on a 300 cell road with max velocity 5) begins with initial traffic conditions, due to the random initilization, which quickly resolve themselves. At all other occupancy percents above the critical occupancy for this road, no vehicles have the required 6 cells to reach the maximum velocity, so after the randomly initilized vehicles adjust, a steady state of traffic is quickly reached.
Additionally, the average velocity seen across the road at each time step decreses as the occupancy percent increases, as each vehicle has less room on average to accelerate.

1.2 Stochastic Behavior

Now, the model will be updated to reflect a driver's tendency to delay acceleration and overreact when decelerating (due to a variety of factors) with the use of a randomization parameter p.

The rules for each vehicle will now be:

1. Accelerate: $v_i = \min\{v_i + 1, v_{max}\}$
2. Decelerate: $v_i = d(i, i+1), \text{if } v_i > d(i, i+1)$, where d(i, j) is the distance between vehicles i and j
3. Randomize: $v_i = \max\{v_i - 1, 0\}$ with probability p
4. Move: vehicle i moves forward v_i cells

This will allow the model to represent the overreaction of driver's when decelerating and their late reaction when accelerating.

Simulation Code

In [ ]:
```
sim = 2
```

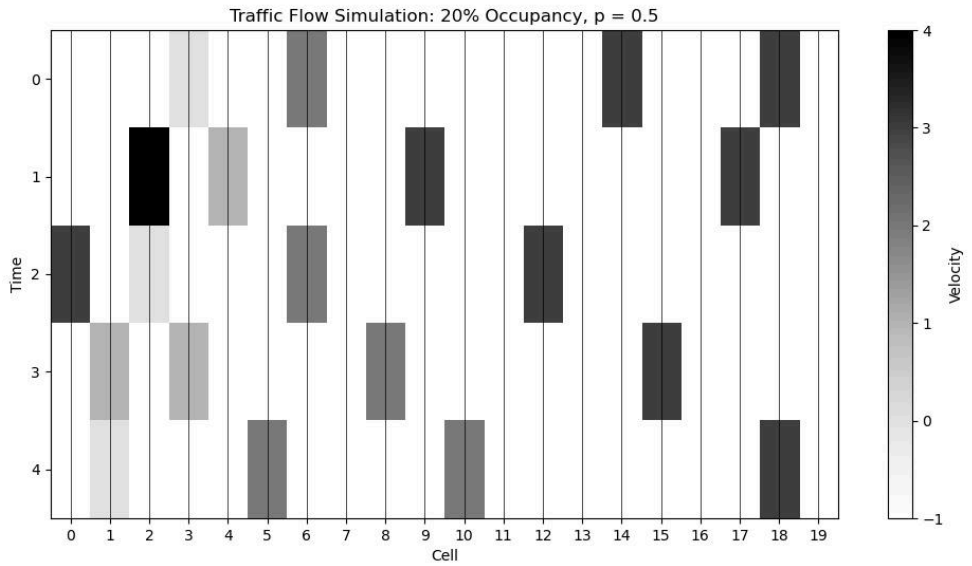
In [ ]:
```
def randomize(cell, p):
  if random.random() <= p:
    cell.velocity = max(cell.velocity - 1, 0)

def update(cells, num_cells, max_vel, p):
  for index, cell in enumerate(cells):
    if cell.state == 1:
      next_distance = distance(cells, index, num_cells)
      accelerate(cell, max_vel)
      decelerate(cell, next_distance)
      randomize(cell, p)
  move(cells, num_cells)
```

Testing

In [ ]:
```
p = 0.5
occupancy_percent = 20
num_cells = 20
num_time_steps = 5
title = f"Traffic Flow Simulation: {occupancy_percent}% Occupancy, p = {p}"

create_plot(max_vel, occupancy_percent, num_cells, num_time_steps, title, p, sim, True)
```



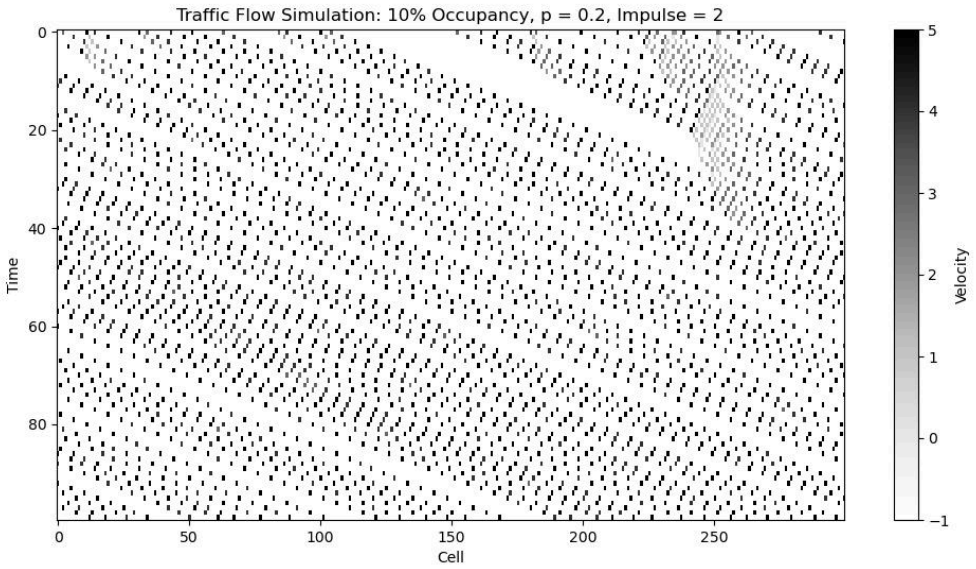
First, as with the non-stochastic model, a simple plot is displayed to show the expected behavior of each vehicle. The same simulation is used as in the first model, with the addition of the randomization step. As show in the plot, the same behavior as in the first model is exhibited, except for vehicles randomly decelerating more than they should.

1.2.1 Initial Impulse

In this section, the output of the stochastic model will be studied with a dally factor of 0.2, and with each vehicle given an initial impulse of 2.

In [ ]:
```
p = 0.2
occupancy_percent = 10
num_cells = 300
num_time_steps = 100
impulse = 2
title = f"Traffic Flow Simulation: {occupancy_percent}% Occupancy, p = {p}, Impulse = {impulse}"

create_plot(max_vel, occupancy_percent, num_cells, num_time_steps, title, p, sim, False, impulse)
```



The most notable difference between this plot and the correpsonding plot in Section 1.1 with a dally factor of 0 is the sudden traffic situation that occurs toward the beginning of the simulation between times 0 and 20. This situation resolves itself quickly, most likely due to both the low dally factor and the low occupancy percent, which is below the critical occupancy (see 1.1.2) necessary to gaurantee traffic conditions. Additionally, there appears to be less space between groups of cars in this simulation. This could be due to both the random intitlization of the cars and cars travelling at the speed limit dallying, allowing groups to catch each other. As can be seen at time 20, there is a large gap with no cars that is quickly filled once a traffic condition begins due to dallying at around cell 250.

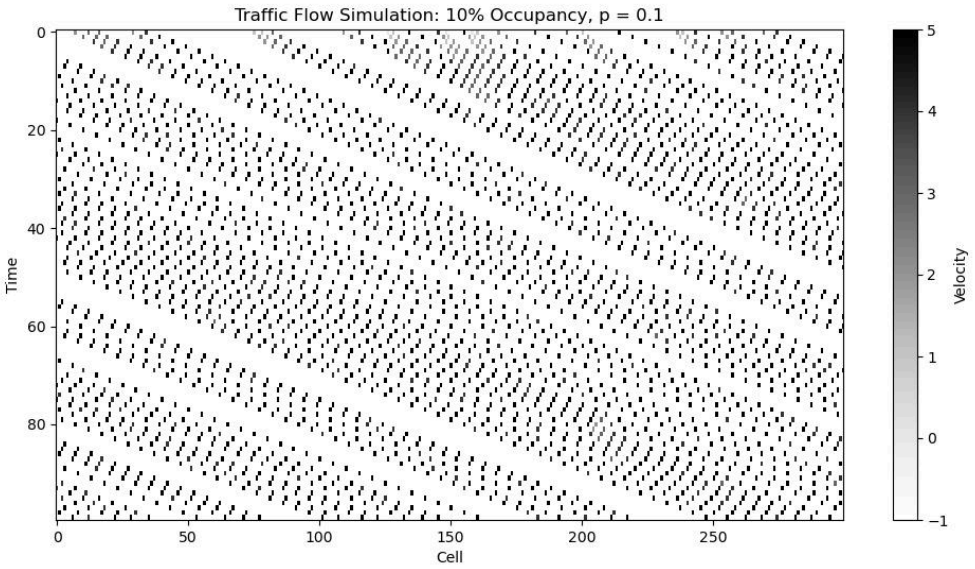
1.2.2 Experiments

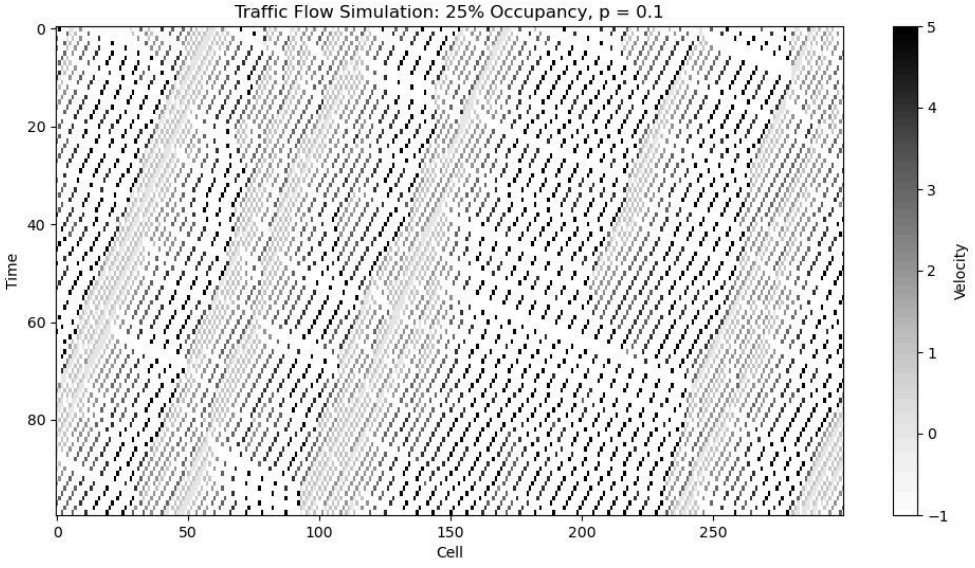
Now experiments will be run with a combination occupancy percents (10, 25, 50, 80) and dally factors (0.1, 0.2, 0.5)

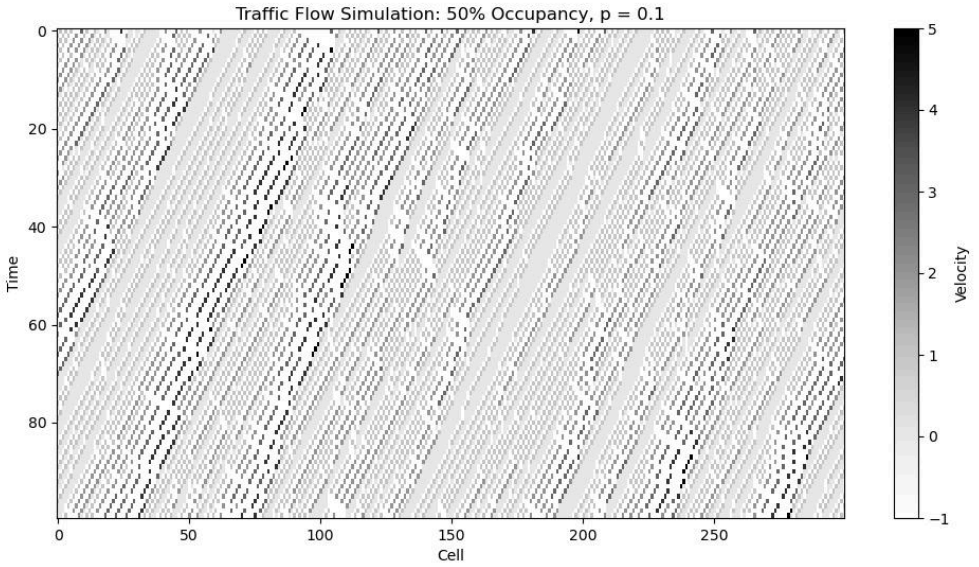
In [ ]:
```
num_cells = 300
num_time_steps = 100
```

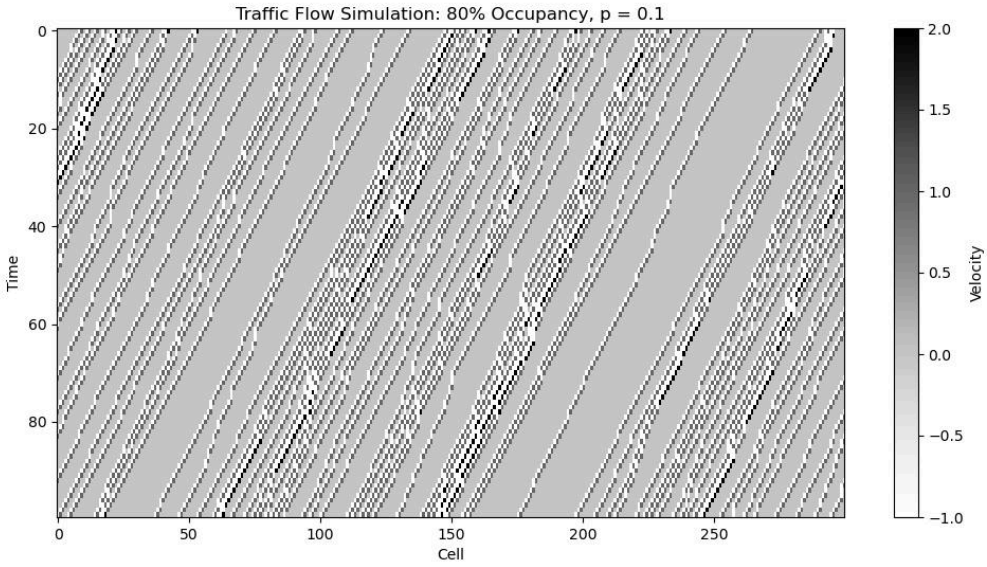
In [ ]:
```
p = 0.1

simulate_occupancy_percents(p, max_vel, num_cells, num_time_steps)
```


Traffic Flow Simulation: 10% Occupancy, p = 0.1
Time
Cell
Velocity
0
20
40
60
80
0
50
100
150
200
250
5
4
3
2
1
0
−1

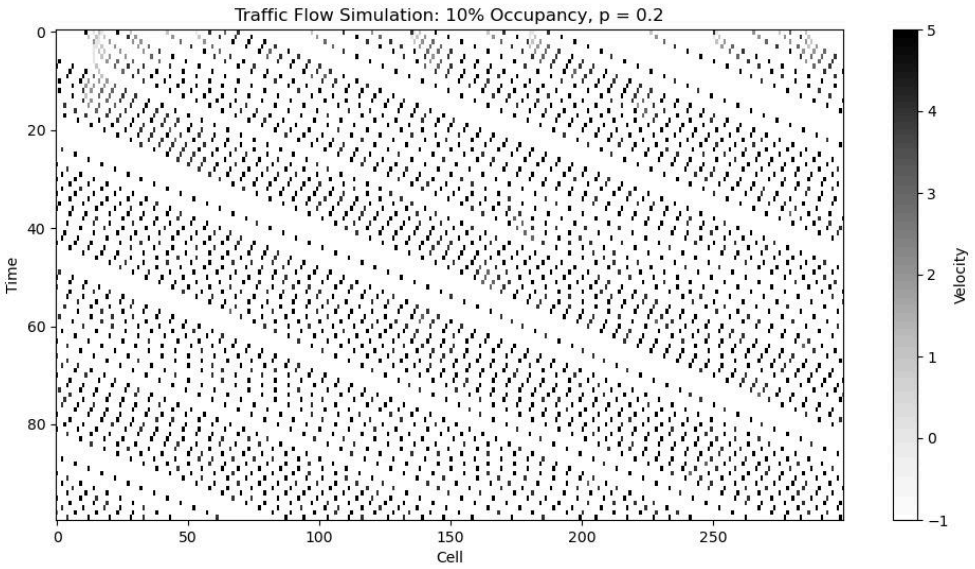

Traffic Flow Simulation: 25% Occupancy, p = 0.1
Time
Cell
Velocity
0
20
40
60
80
0
50
100
150
200
250
5
4
3
2
1
0
−1

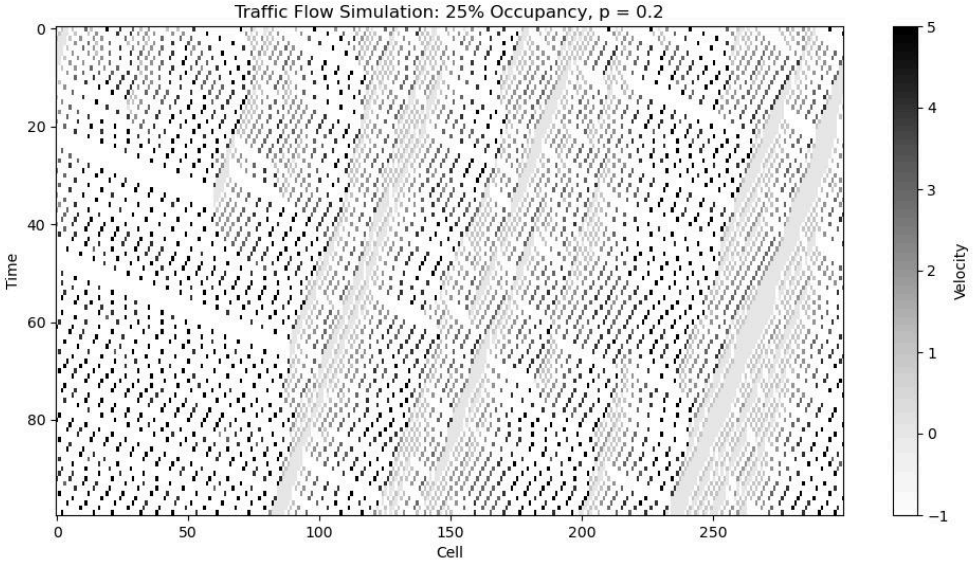


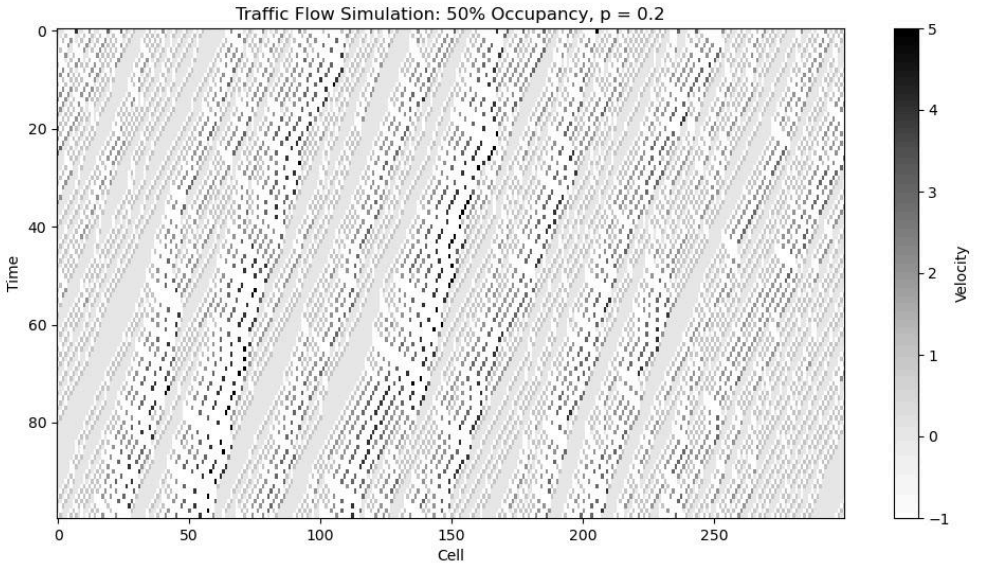


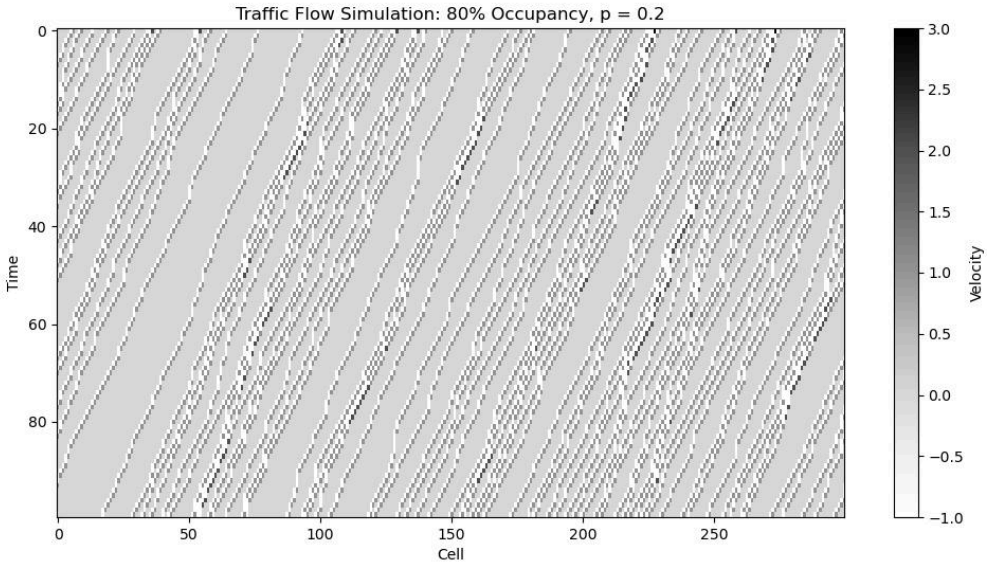
In [ ]:
```
p = 0.2

simulate_occupancy_percents(p, max_vel, num_cells, num_time_steps)
```


Traffic Flow Simulation: 10% Occupancy, p = 0.2
Time
Cell
Velocity
0
20
40
60
80
0
50
100
150
200
250
5
4
3
2
1
0
−1

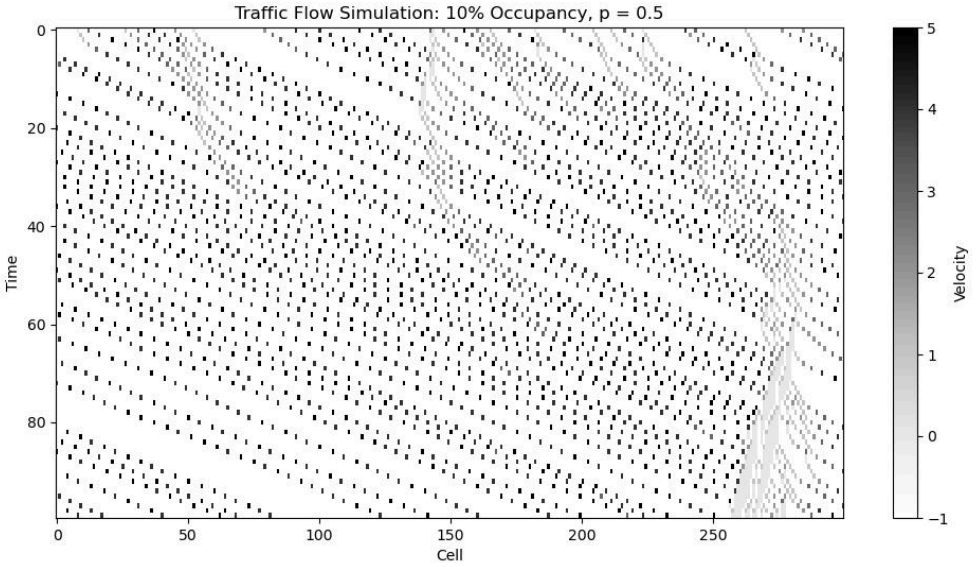

Traffic Flow Simulation: 25% Occupancy, p = 0.2
Time
Cell
Velocity
0
20
40
60
80
0
50
100
150
200
250
5
4
3
2
1
0
−1

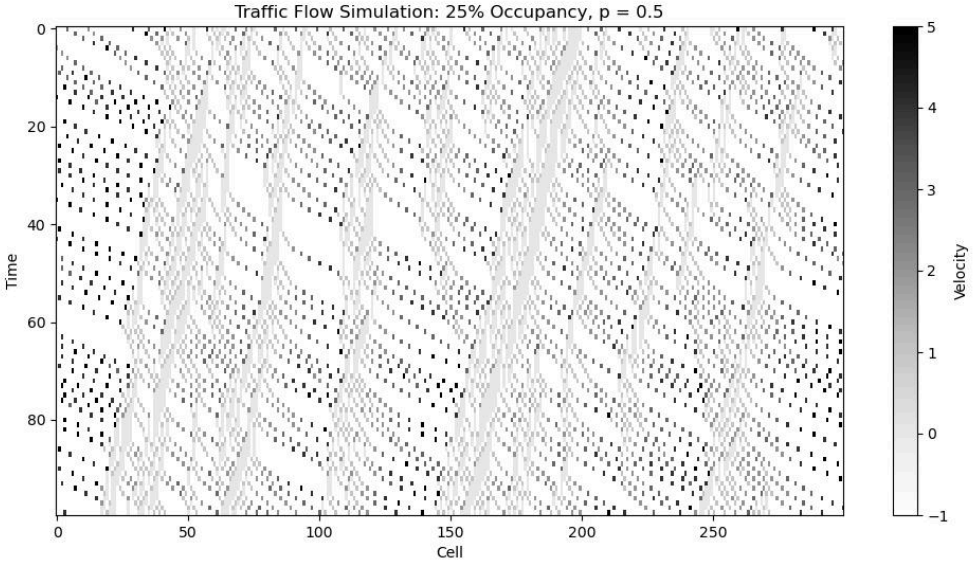


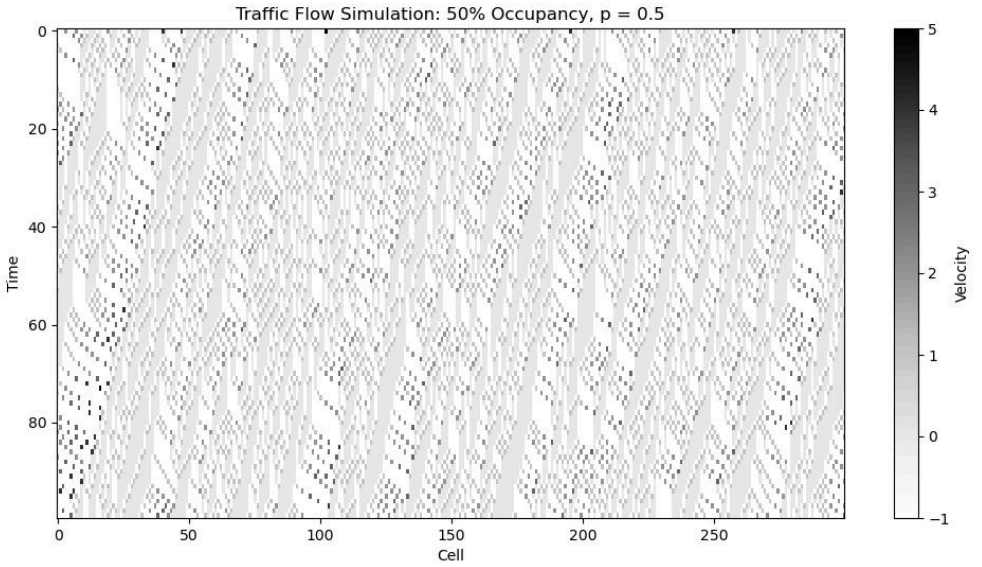


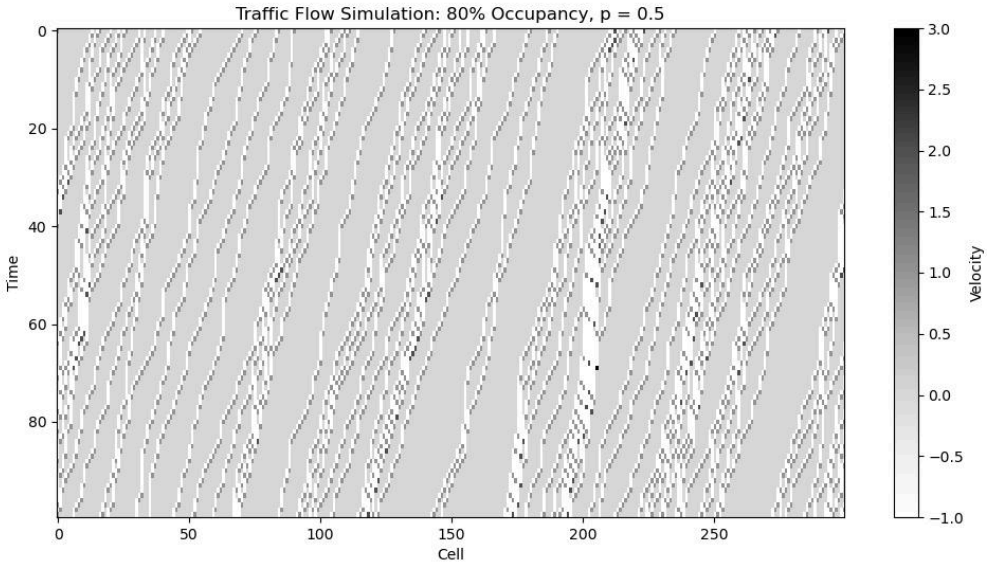
In [ ]:
```
p = 0.5

simulate_occupancy_percents(p, max_vel, num_cells, num_time_steps)
```


Traffic Flow Simulation: 10% Occupancy, p = 0.5
Time
Cell
Velocity
0
20
40
60
80
0
50
100
150
200
250
5
4
3
2
1
0
−1


Traffic Flow Simulation: 25% Occupancy, p = 0.5
Time
Cell
Velocity
0
20
40
60
80
0
50
100
150
200
250
5
4
3
2
1
0
−1





With the use of the dally factor the stochastic model now exhibits waves of traffic situations. These traffic jams appear for no apparent reason, and even occur spontaneously when the occupancy percent is non-critical. These traffic conditions occur when a driver dallys, and as a result the driver behind must decelerate. This dirver may overreact when decelerating (due to the dally factor), and the vehicle behind will decelerate. As a result, traffic occurs. Since these traffic conditions are a result of dallying, it would be expected that more traffic jams would occur with a higher dally factor, as well as with higher occupancy percent (since higher density results in a higher probability that a car will need to decelerate if another car dallys). This expected behavior is exhibited in the above plots, with more waves of traffic appearing with higher p and occupancy percent. Aside from the traffic conditions, the overall shape of these plots is similar to that of the first model, with any plots with an occupancy percent lower than the critical level displaying cars moving at the speed limit (aside from those stuck in traffic), and the other plots showing steady state traffic. These plots seem to match the analysis given in Bungartz et al. (2014), with the plots with a higher dally factor displaying traffic jams that do not resolve themselves and larger regions of slower traffic.

1.4 Two Lane Traffic

The stochastic model from 1.3 will now be extended to two lanes. There will be a left lane and a right lane, each with a maximum velocity. The vehicles in the model will behave similarly to the one lane model, but will now attempt to change lanes before decelerating. A vehicle will only change lanes if this action will allow it to keep its current speed rather than decelerating. Vehicles will only change lanes cautiously, meaning that they will change lanes only if there is no possibility of a collision, regardless of the actions

of the vehicles around them. Unlike in the one lane models, each vehicle will now consider both the distance to and speed of the vehicles around it.

1.4.1 - 1.4.2 Algorithm

To achieve the cautious land change behavior, each vehicle will consider the vehicle directly in front of it and behind it, and will change lanes only if these cars can both change their speeds or change lanes without a collision occuring. The current vehicle will also take into account the potential dallying of itself and other vehicles

This leads to four conditions that must be met before a vehicle changes lanes. Let i denote the current vehicle and let the subscript o denote the opposite lane.

1. $v_i < v_{i+1} - 1 + d(i, i+1) + 1$
2. $d(i, (i+1)_o) \geq v_i$
3. $v_{i-1} - (d(i, i-1) + 1) < v_i - 1$
4. $v_{(i-1)_o} - (d(i, (i-1)_o) + 1) < v_i - 1$

Condition 1 ensures that a vehicle will not collide with the vehicle in front of it if they both switch lanes. Recall that d(i, j) gives the distance between two cells, with adjacent cells having distance 0. Therefore, vehicle i will end up v_i cells ahead of its current postion, which must be behind the vehicle in front of it (if that vehicle switches lanes). Vehicle i + 1 will end up $v_{i+1} + d(i, i+1) + 1$ cells ahead of vehicle i, in the other lane (without dallying).

Condition 2 simply ensures that vehicle i will not collide with the vehicle that would be in front of it in the other lane.

Condition 3 ensures that a collision will not occur if vehicle i - 1 changes lanes. Vehicle i - 1 will end up $v_{i-1} - (d(i, i-1) + 1)$ cells ahead of the cell vehicle i is currently in. After changing lanes, vehicle i must be sure it ends up ahead of vehicle i - 1, otherwise there would be a collision. This condition is unneccesary to avoid collisions, since each vehicle checks that a collision will not occur with the vehicle in front of it. However, every driver is cautious when changing lanes, and does not asssume that every other driver will follow these rules.

Similarly, condition 4 makes sure that after changing lanes, the vehicle behind vehicle i in the other lane will not collide with it.

Note that these cases take into account any potential dallying, both by other vehicles and the vehicle attempting to change lanes. It is also assumed in these conditions that the current vehicle knows that the vehicle in front of it will not change lanes unless it can maintain its current speed (otherwise lane changes would never occur because of the possibility that the vehicle in front of the current vehicle could change lanes and decelerate).

There are some more implied conditions that will be checked before a lane change is attempted: the current vehicle needs to change lanes to avoid decelerating, there is not another vehicle directly adjacent to the current vehicle in the opposite lane, and the current vehicle will not break the speed limit of the opposite lane after changing lanes.
While it may be possible for a driver to change lanes if a vehicle is directly next to it (as long as they will not end up in the same cell), the decision to change lanes by a vehicle occurs every time step, and a driver would never decide to begin changing lanes while another driver is directly next to them. Therefore, this condition will be used in this model.

This gives the complete set of rules for each vehicle:

1. Accelerate: $v_i = \min\{v_i + 1, v_{max}\}$
2. Change Lanes: If all conditions for cautious lane change are met
3. Decelerate: $v_i = d(i, i+1), \text{if } v_i > d(i, i+1)$, where d(i, j) is the distance between vehicles i and j
4. Randomize: $v_i = \max\{v_i - 1, 0\}$ with probability p
5. Move: vehicle i moves forward v_i cells

1.4.3 Simulation Code

In [ ]:
```
def initilize(num_cells, occupancy_percents, max_vels):
  cells = []
  for i in range(2):
    occupancy_percent = occupancy_percents[i]
    max_vel = max_vels[i]
    curr_cells = [Cell() for _ in range(num_cells)]
    num_cars = (int)((occupancy_percent/100) * num_cells)
    indices = random.sample(range(num_cells), num_cars)
    for index in indices:
      curr_cells[index].state = 1
```

```
      curr_cells[index].velocity = random.randint(0, max_vel)
    cells.append(curr_cells)
  return cells

def distance(cells, curr_cells, index, num_cells, curr_lane, intended_lane, direction):
  if curr_lane == intended_lane:
    for i in range(1, len(curr_cells)):
      next_cell = curr_cells[((direction * i) + (index)) % num_cells]
      if next_cell.state == 1:
        return i - 1
  else:
    for i in range(1, len(curr_cells)):
      other_lane = cells[intended_lane]
      next_cell = other_lane[((direction * i) + (index)) % num_cells]
      if next_cell.state == 1:
        return i - 1
  return num_cells

def can_change_lanes(cells, cell, i, index, num_cells, max_vels):
  curr_lane = cells[i]
  other_lane = cells[(i+1)%2]
  dist_ahead = distance(cells, curr_lane, index, num_cells, i, i, 1)
  dist_behind = distance(cells, curr_lane, index, num_cells, i, i, -1)
  dist_ahead_other = distance(cells, curr_lane, index, num_cells, i, (i+1)%2, 1)
  dist_behind_other = distance(cells, curr_lane, index, num_cells, i, (i+1)%2, -1)
  cond1 = cell.velocity < curr_lane[(index + dist_ahead + 1)%num_cells].velocity - 1 + dist_ahead + 1
  cond2 = dist_ahead_other >= cell.velocity
  cond3 = curr_lane[(index - (dist_behind + 1))%num_cells].velocity - (dist_behind + 1) < cell.velocity - 1
  cond4 = other_lane[(index - (dist_behind_other + 1))%num_cells].velocity - (dist_behind_other + 1) < cell.veloc
  if (cell.velocity <= max_vels[(i+1)%2] and other_lane[index].state == 0):
    return cond1 and cond2 and cond3 and cond4
  else:
    return False

def change_lanes(cells, i, index):
  curr_lane = cells[i]
  other_lane = cells[(i+1)%2]
  other_lane[index].velocity = curr_lane[index].velocity
  other_lane[index].state = 1
  curr_lane[index].velocity = -1
  curr_lane[index].state = 0

def move(cells, num_cells):
  lane_change = False
  cells_copy = copy.deepcopy(cells)
  for index, cell in enumerate(cells_copy):
    if cell.state == 1:
      next_cell = (index + cell.velocity) % num_cells
      if lane_change:
        cells[next_cell].state = 1
        cells[next_cell].velocity = cell.velocity
      else:
        cells[index].state = 0
        cells[index].velocity = -1
        cells[next_cell].state = 1
        cells[next_cell].velocity = cell.velocity
      if cells_copy[next_cell].state == 1 and index != next_cell:
        lane_change = True
      else:
        lane_change = False

def update(cells, num_cells, max_vels, p):
  for i in range(2):
    curr_cells = cells[i]
    for index, cell in enumerate(curr_cells):
      if cell.state == 1:
        accelerate(cell, max_vels[i])
  cells_copy = copy.deepcopy(cells)
  for i in range(2):
    curr_cells = cells[i]
    for index, cell in enumerate(curr_cells):
      if cell.state == 1:
        next_distance = distance(cells_copy, cells_copy[i], index, num_cells, i, i, 1)
        if cell.velocity > next_distance and can_change_lanes(cells_copy, cells_copy[i][index], i, index, num_cel
          change_lanes(cells, i, index)
  for i in range(2):
    curr_cells = cells[i]
    for index, cell in enumerate(curr_cells):
        if cell.state == 1:
          next_distance = distance(cells_copy, cells_copy[i], index, num_cells, i, i, 1)
          decelerate(cell, next_distance)
          randomize(cell, p)
```

```
  for i in range(2):
    curr_cells = cells[i]
    move(curr_cells, num_cells)
```

In [ ]:
```
def simulate(max_vels, occupancy_percents, num_cells, num_time_steps, p):
    cells = initilize(num_cells, occupancy_percents, max_vels)
    velocities_both = np.zeros((3*num_time_steps + 2, num_cells))
    velocities_left = np.zeros((num_time_steps + 1, num_cells))
    velocities_right = np.zeros((num_time_steps + 1, num_cells))
    for time in range(0, num_time_steps + 1):
      update(cells, num_cells, max_vels, p)
      for i in range(2):
        for index, cell in enumerate(cells[i]):
          if i == 0:
             velocities_left[time][index] = cell.velocity
          else:
             velocities_right[time][index] = cell.velocity
          velocities_both[3*time + i][index] = cell.velocity
    return velocities_both, velocities_left, velocities_right

def create_plots(bothlanes, leftlane, rightlane, max_vels, occupancy_percents, num_cells, num_time_steps, p, sim,
  velocities = simulate(max_vels, occupancy_percents, num_cells, num_time_steps, p)
  if bothlanes:
    title = f'Both Lanes, p = {p}, Left: {occupancy_percents[0]}% Occupancy, Max Vel= {max_vels[0]}, Right: {occu
    create_plot_bothlanes(num_cells, num_time_steps, title, p, sim, gridlines, velocities[0])
  if leftlane:
    title = f'Left Lane: p = {p}, {occupancy_percents[0]}% Occupancy, Max Vel= {max_vels[0]}'
    create_plot_singlelane(num_cells, title, p, sim, gridlines, velocities[1], 0)
  if rightlane:
    title = f'Right Lane: p = {p}, {occupancy_percents[1]}% Occupancy, Max Vel= {max_vels[1]}'
    create_plot_singlelane(num_cells, title, p, sim, gridlines, velocities[2], 1)

def create_plot_bothlanes(num_cells, num_time_steps, title, p, sim, gridlines, velocities):
  time_steps = range(0, num_time_steps + 1)
  plt.figure(figsize=(12, 6))
  plt.imshow(velocities, cmap='Greys', aspect='auto', interpolation='nearest')
  plt.xlabel("Cell")
  plt.ylabel("Time")
  plt.title(title)
  plt.colorbar(label="Velocity")
  num_ticks = min(20, len(time_steps))
  tick_step = max(1, len(time_steps) // num_ticks)
  plt.yticks(np.arange(0, num_time_steps * 3 + 2, 3 * tick_step), time_steps[::tick_step])
  if gridlines:
    ax = plt.gca()
    ax.xaxis.set_major_locator(MultipleLocator(1))
    ax.grid(which='major', axis='x', linestyle='-', linewidth=0.5, color='k')
  plt.savefig(f'{num_cells},{p},{sim}.jpg')
  plt.close()

def create_plot_singlelane(num_cells, title, p, sim, gridlines, velocities, lane):
  plt.figure(figsize=(12, 6))
  plt.imshow(velocities, cmap='Greys', aspect='auto', interpolation='nearest')
  plt.xlabel("Cell")
  plt.ylabel("Time")
  plt.title(title)
  plt.colorbar(label="Velocity")
  if gridlines:
    ax = plt.gca()
    ax.xaxis.set_major_locator(MultipleLocator(1))
    ax.grid(which='major', axis='x', linestyle='-', linewidth=0.5, color='k')
  plt.savefig(f'{num_cells},{p},{sim},{lane}.jpg')
  plt.close()
```

1.4.4 Testing

distance

In [ ]:
```
test_cells = [[Cell(1, 0), Cell(0, -1), Cell(1, 0), Cell(0, -1), Cell(0, -1)],
              [Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1)]]

test_distance = distance(test_cells, test_cells[0], 2, 5, 0, 0, 1)
print(test_distance == 2)

test_cells = [[Cell(0, -1), Cell(0, -1), Cell(1, 0), Cell(0, -1), Cell(1, 0)],
              [Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1)]]

test_distance = distance(test_cells, test_cells[0], 2, 5, 0, 0, -1)
print(test_distance == 2)
```

```
test_cells = [[Cell(0, -1), Cell(0, -1), Cell(1, 0), Cell(0, -1), Cell(0, -1)],
              [Cell(1, 0), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1)]]

test_distance = distance(test_cells, test_cells[0], 2, 5, 0, 1, 1)
print(test_distance == 2)

test_cells = [[Cell(0, -1), Cell(0, -1), Cell(1, 0), Cell(0, -1), Cell(0, -1)],
              [Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(1, 0)]]

test_distance = distance(test_cells, test_cells[0], 2, 5, 0, 1, -1)
print(test_distance == 2)
```

```
True
True
True
True
```

can_change_lanes

Condition 1

In [ ]:
```
test_cells = [[Cell(1, 2), Cell(1, 2), Cell(0, -1), Cell(0, -1), Cell(0, -1)],
              [Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1)]]

test_can_change_lanes = can_change_lanes(test_cells, test_cells[0][0], 0, 0, 5, [5, 7])
print(test_can_change_lanes == False)

test_cells = [[Cell(1, 2), Cell(1, 0), Cell(0, -1), Cell(0, -1), Cell(0, -1)],
              [Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1)]]

test_can_change_lanes = can_change_lanes(test_cells, test_cells[0][0], 0, 0, 5, [5, 7])
print(test_can_change_lanes == False)

test_cells = [[Cell(1, 2), Cell(1, 3), Cell(0, -1), Cell(0, -1), Cell(0, -1)],
              [Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1)]]

test_can_change_lanes = can_change_lanes(test_cells, test_cells[0][0], 0, 0, 5, [5, 7])
print(test_can_change_lanes == True)
```

```
True
True
True
```

Condition 2

In [ ]:
```
test_cells = [[Cell(0, -1), Cell(1, 5), Cell(0, -1), Cell(0, -1), Cell(0, -1)],
              [Cell(1, 2), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1)]]

test_can_change_lanes = can_change_lanes(test_cells, test_cells[1][0], 1, 0, 5, [5, 7])
print(test_can_change_lanes == False)

test_cells = [[Cell(0, -1), Cell(0, -1), Cell(1, 2), Cell(0, -1), Cell(0, -1)],
              [Cell(1, 2), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1)]]

test_can_change_lanes = can_change_lanes(test_cells, test_cells[1][0], 1, 0, 5, [5, 7])
print(test_can_change_lanes == False)

test_cells = [[Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(1, 0), Cell(0, -1)],
              [Cell(1, 2), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1)]]

test_can_change_lanes = can_change_lanes(test_cells, test_cells[1][0], 1, 0, 5, [5, 7])
print(test_can_change_lanes == True)
```

```
True
True
True
```

Condition 3

In [ ]:
```
test_cells = [[Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1)],
              [Cell(1, 2), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(1, 4)]]

test_can_change_lanes = can_change_lanes(test_cells, test_cells[1][0], 1, 0, 5, [5, 7])
print(test_can_change_lanes == False)

test_cells = [[Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1)],
              [Cell(1, 2), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(1, 2)]]

test_can_change_lanes = can_change_lanes(test_cells, test_cells[1][0], 1, 0, 5, [5, 7])
print(test_can_change_lanes == False)
```

```
test_cells = [[Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1)],
              [Cell(1, 2), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(1, 1)]]

test_can_change_lanes = can_change_lanes(test_cells, test_cells[1][0], 1, 0, 5, [5, 7])
print(test_can_change_lanes == True)
```

```
True
True
True
```

Condition 4

In [ ]:
```
test_cells = [[Cell(1, 2), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1)],
              [Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(1, 3)]]

test_can_change_lanes = can_change_lanes(test_cells, test_cells[0][0], 0, 0, 5, [5, 7])
print(test_can_change_lanes == False)

test_cells = [[Cell(1, 2), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1)],
              [Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(1, 2)]]

test_can_change_lanes = can_change_lanes(test_cells, test_cells[0][0], 0, 0, 5, [5, 7])
print(test_can_change_lanes == False)

test_cells = [[Cell(1, 2), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1)],
              [Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(1, 1)]]

test_can_change_lanes = can_change_lanes(test_cells, test_cells[0][0], 0, 0, 5, [5, 7])
print(test_can_change_lanes == True)
```

```
True
True
True
```

v_i > Max velocity of other lane

Adjacent car in other lane

In [ ]:
```
test_cells = [[Cell(1, 4), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1)],
              [Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(1, 1)]]

test_can_change_lanes = can_change_lanes(test_cells, test_cells[0][0], 0, 0, 5, [7, 3])
print(test_can_change_lanes == False)

test_cells = [[Cell(1, 4), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1)],
              [Cell(0, 0), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(1, 1)]]

test_can_change_lanes = can_change_lanes(test_cells, test_cells[0][0], 0, 0, 5, [7, 7])
print(test_can_change_lanes == False)
```

```
True
True
```

update

In [ ]:
```
test_cells = [[Cell(1, 2), Cell(0, -1), Cell(1, 1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(1, 3
              [Cell(0, -1), Cell(0, -1), Cell(1, 3), Cell(1, 0), Cell(0, -1), Cell(1, 1), Cell(0, -1), Cell(0, -1

expected_cells = [[Cell(0, -1), Cell(1, 1), Cell(0, -1), Cell(0, -1), Cell(1, 2), Cell(0, -1), Cell(0, -1), Cell((
              [Cell(0, -1), Cell(1, 4), Cell(1, 0), Cell(0, -1), Cell(1, 1), Cell(0, -1), Cell(0, -1), Cell(1, 2)

update(test_cells, 10, [5, 7], 0)

print(test_cells == expected_cells)


test_cells = [[Cell(1, 1), Cell(0, -1), Cell(1, 2), Cell(1, 3), Cell(0, -1), Cell(1, 0), Cell(0, -1), Cell(0, -1)
              [Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0,

expected_cells = [[Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(1, 1), Cell(0, -1), Cell(1, 1), Cell((
              [Cell(1, 1), Cell(0, -1), Cell(1, 2), Cell(0, -1), Cell(0, -1), Cell(1, 3), Cell(0, -1), Cell(0, -1

update(test_cells, 10, [5, 6], 0)

print(test_cells == expected_cells)


test_cells = [[Cell(1, 1), Cell(0, -1), Cell(1, 5), Cell(1, 5), Cell(0, -1), Cell(1, 0), Cell(0, -1), Cell(0, -1)
              [Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0,

expected_cells = [[Cell(0, -1), Cell(0, -1), Cell(1, 0), Cell(0, -1), Cell(1, 1), Cell(0, -1), Cell(1, 1), Cell(0
              [Cell(1, 1), Cell(0, -1), Cell(1, 2), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -
```

```
update(test_cells, 10, [6, 5], 0)

print(test_cells == expected_cells)

test_cells = [[Cell(0, -1), Cell(1, 2), Cell(0, -1), Cell(0, -1), Cell(1, 2), Cell(0, -1), Cell(0, -1), Cell(0, -
              [Cell(1, 0), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(1, 3), Cell(0, -

expected_cells = [[Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cel
              [Cell(0, -1), Cell(1, 1), Cell(0, -1), Cell(0, -1), Cell(1, 3), Cell(0, -1), Cell(0, -1), Cell(0, -

update(test_cells, 10, [6, 5], 0)

print(test_cells == expected_cells)

test_cells = [[Cell(0, -1), Cell(1, 1), Cell(0, -1), Cell(1, 0), Cell(0, -1), Cell(1, 0), Cell(1, 2), Cell(1, 5),
              [Cell(1, 1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(0, -

expected_cells = [[Cell(1, 3), Cell(0, -1), Cell(1, 1), Cell(0, -1), Cell(1, 1), Cell(0, -1), Cell(0, -1), Cell(0
              [Cell(0, -1), Cell(0, -1), Cell(1, 2), Cell(0, -1), Cell(0, -1), Cell(0, -1), Cell(1, 1), Cell(0, -

update(test_cells, 10, [6, 5], 0)

print(test_cells == expected_cells)
```

```
True
True
True
True
True
```

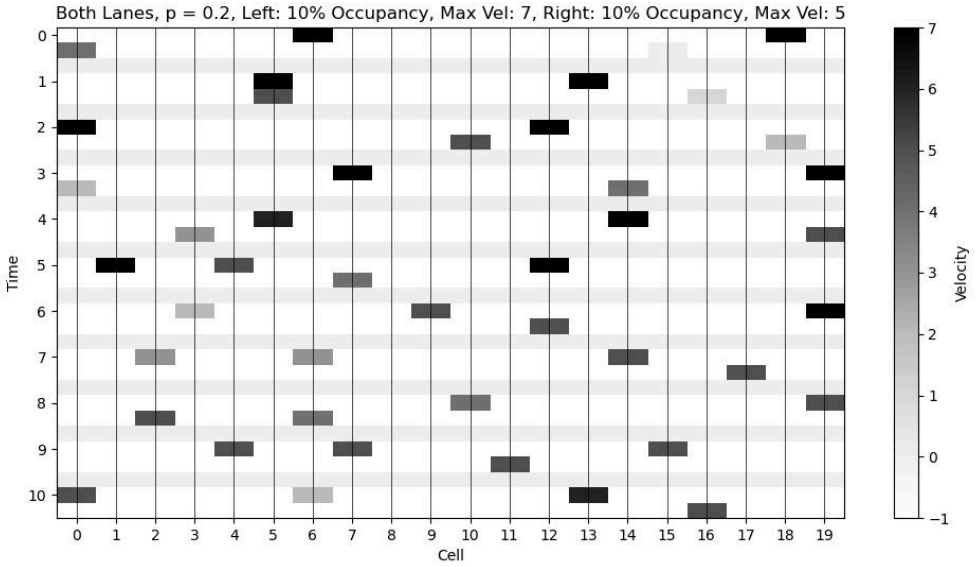
To test and verify the implementation of the algorithm described above, these test case were used. These cases give comprehensive tests of the methods that differ most between their implementation here and in the other models. For example, the distance function is used throughout the code in many important ways, such as for determining when a vehicle can switch lanes. In the other models, it simply needed to look ahead from the current vehicle to find the distance to the next vehicle. Here, a parameter is used to tell the model which lane to look in and in what direction. The four cases above test both lanes and both directions, as well ensure the correct distance is returned near the road boundaries. The other tests follow this same pattern to test the algorithm's functionality.

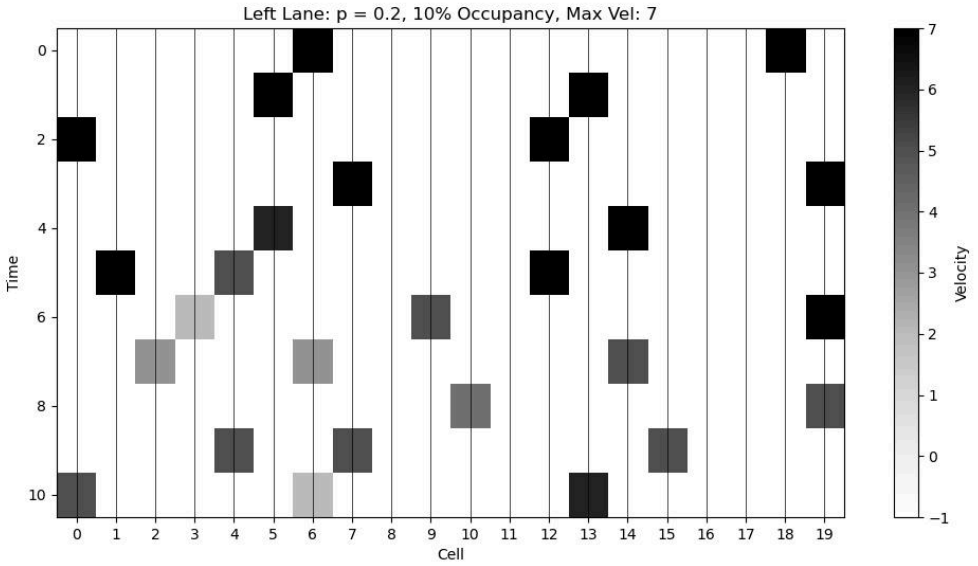
1.4.5 - 1.4.6 Experiments

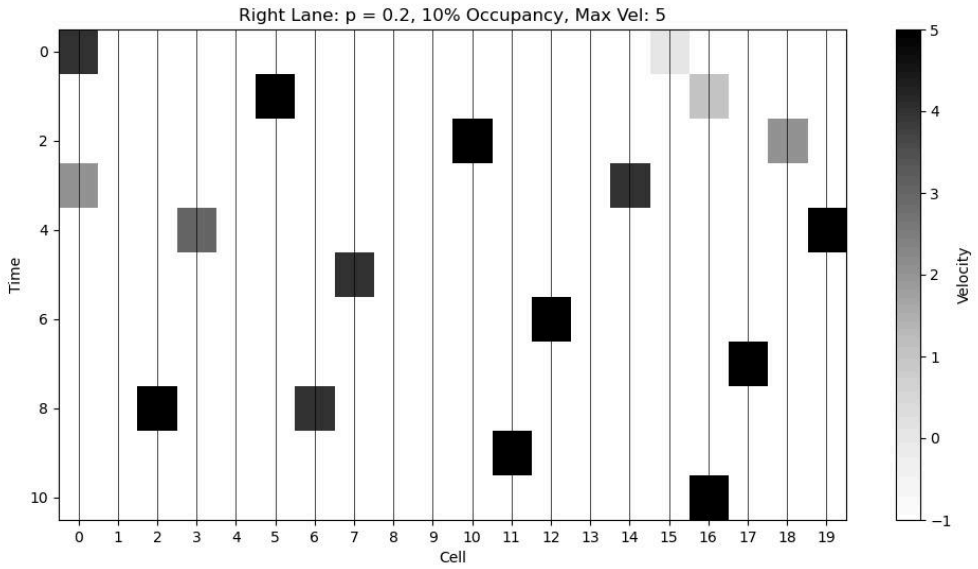
In this section, the model will be used to create several plots.

In [ ]:
```
sim = 3
```

In [ ]:
```
p = 0.2
occupancy_percents = [10, 10]
max_vels = [7, 5]
num_cells = 20
num_time_steps = 10
create_plots(True, True, True, max_vels, occupancy_percents, num_cells, num_time_steps, p, sim, True)
```


Both Lanes, p = 0.2, Left: 10% Occupancy, Max Vel: 7, Right: 10% Occupancy, Max Vel: 5
Time
Cell
Velocity

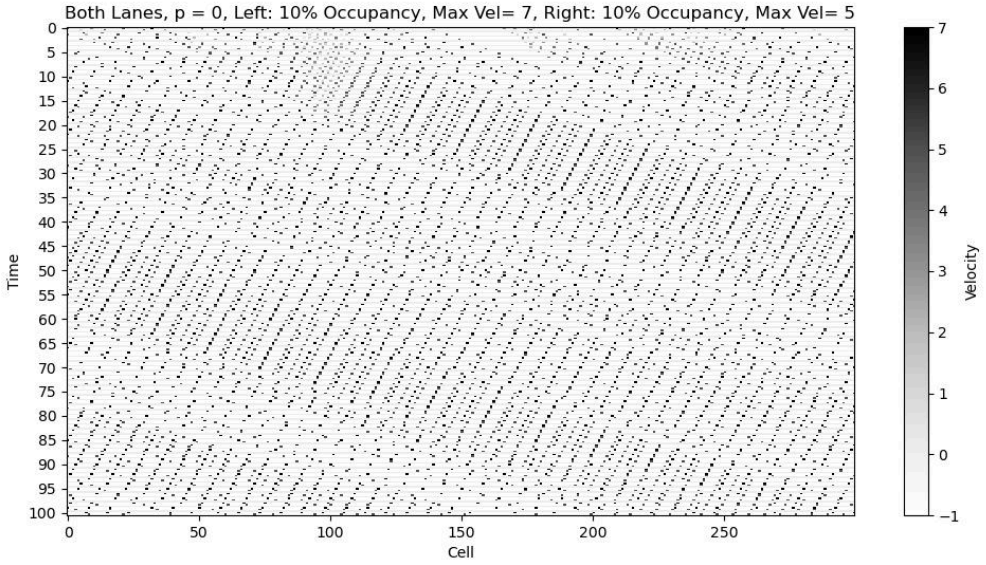

Left Lane: p = 0.2, 10% Occupancy, Max Vel: 7
Time
Cell
Velocity

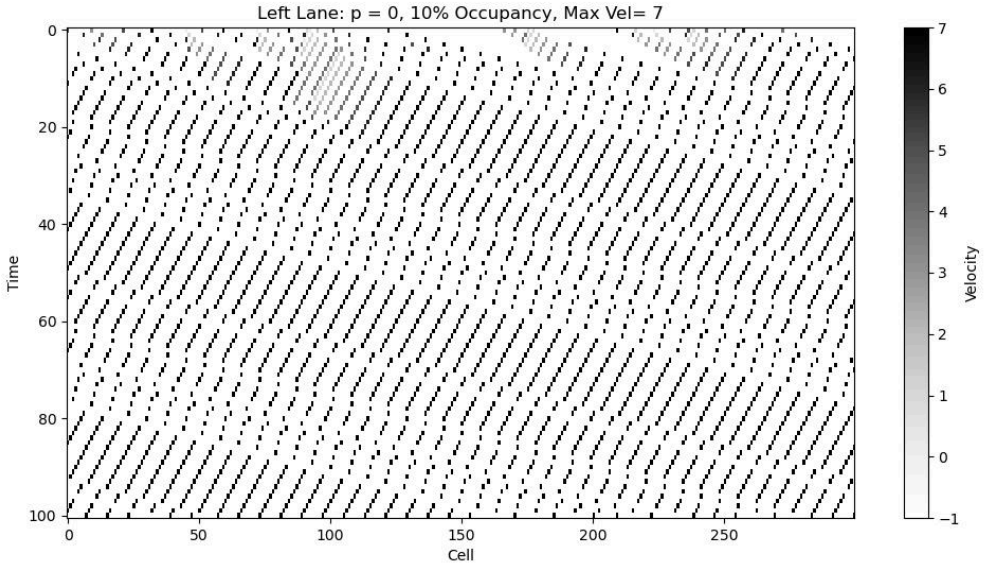


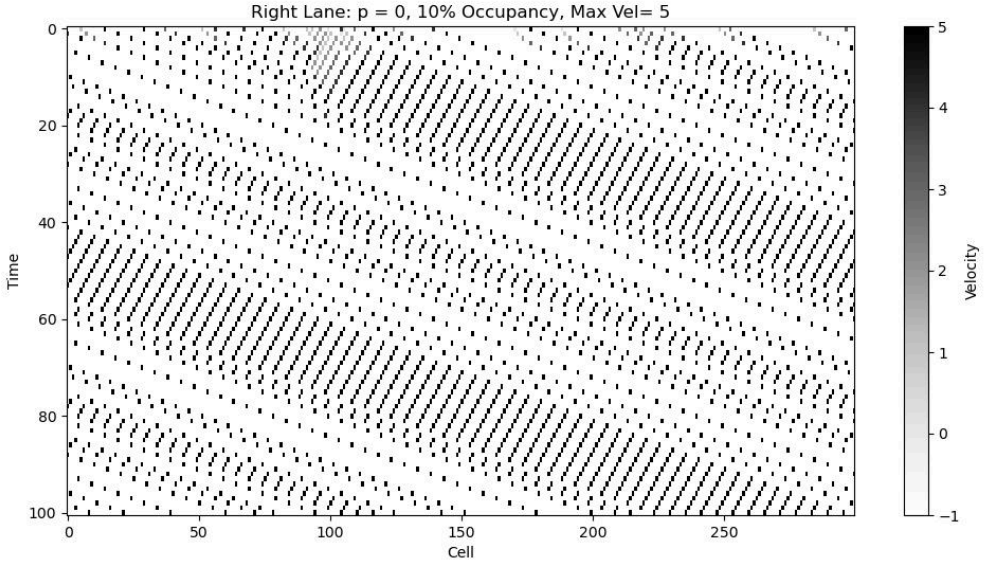
The plots will appear as above. The first plot displays both lanes, with the left lane on top and the right on bottom There is space between the road at different time steps, signified by a gray bar across the plot. The next plot is a plot of just the left lane, as it would appear if it were an entire road in any of the previous sections (without the assumption that vehicles are conserved). Similarly, the next plot displays the right lane.

The model will now be used to see results for occupancy percents 10 and 50 and dally factors of 0 and 0.5. These values will allow the observation of changes in occupancy percent, specifically between a non-critical and critical value, as well as a comparison between this model and both the non-stochastic and stochastic one lane models.

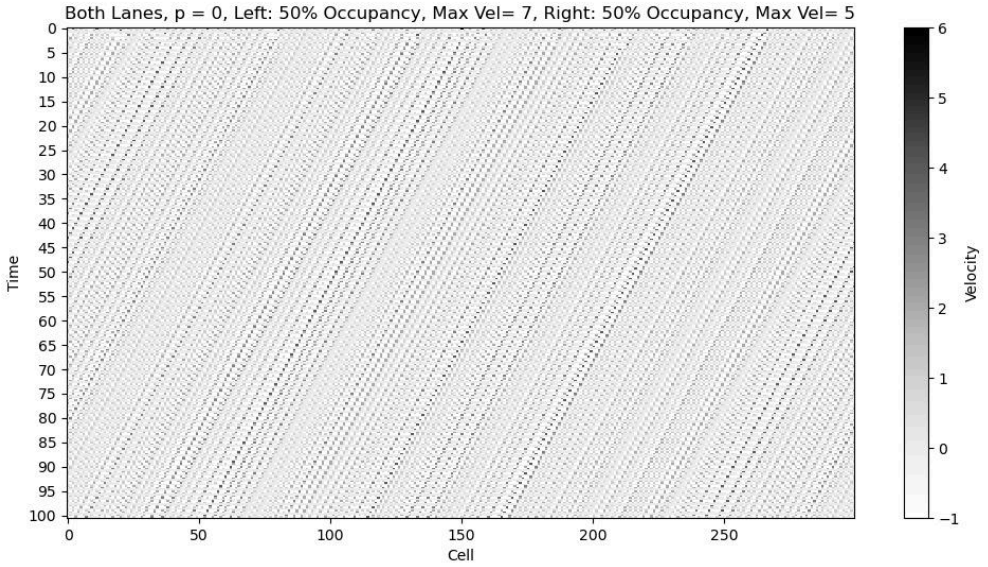
In [ ]:
```
sim = 3
p = 0
occupancy_percents = [10, 10]
max_vels = [7, 5]
num_cells = 300
num_time_steps = 100
create_plots(True, True, True, max_vels, occupancy_percents, num_cells, num_time_steps, p, sim, False)
```

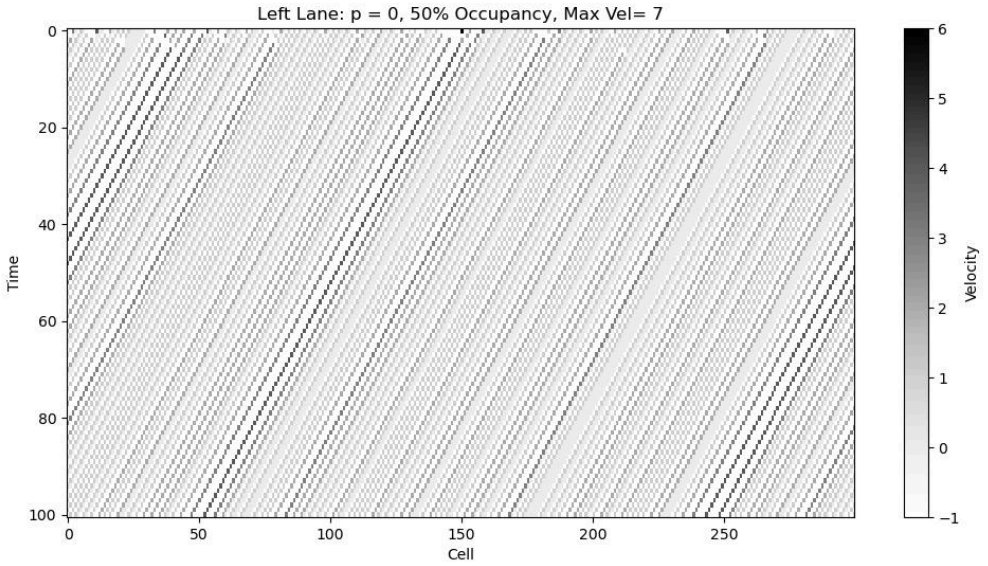


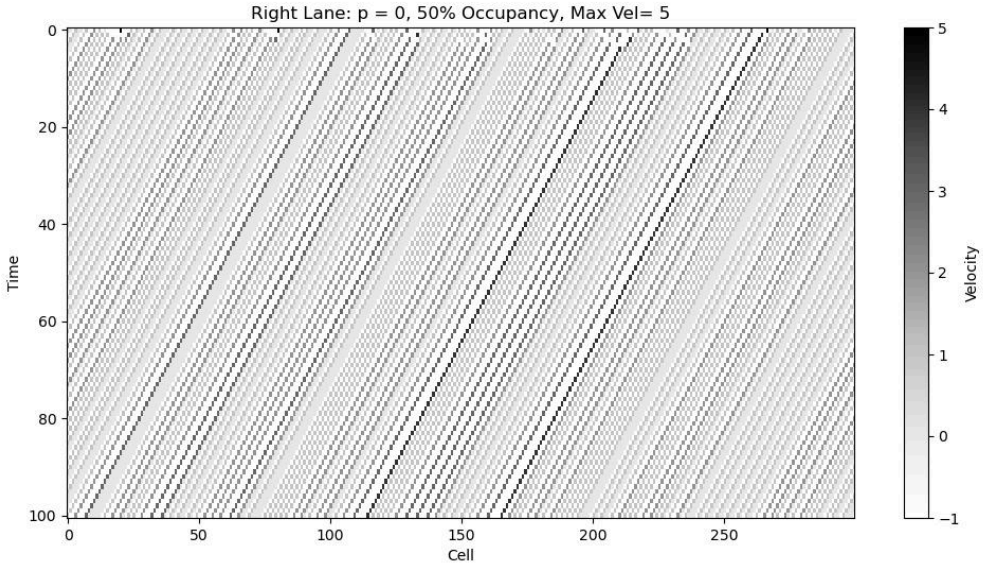




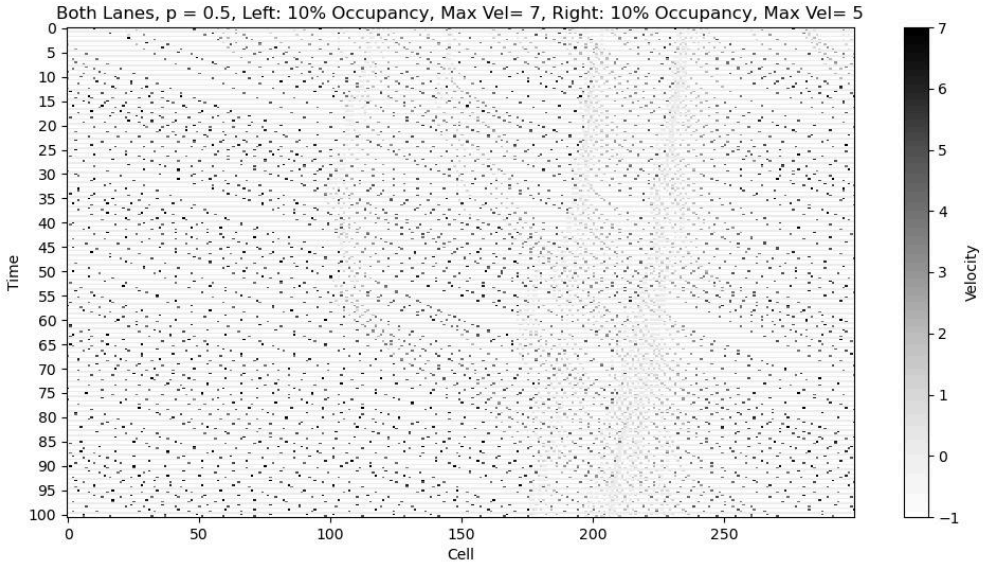
In [ ]:
```
sim = 4
p = 0
occupancy_percents = [50, 50]
max_vels = [7, 5]
num_cells = 300
num_time_steps = 100
create_plots(True, True, True, max_vels, occupancy_percents, num_cells, num_time_steps, p, sim, False)
```

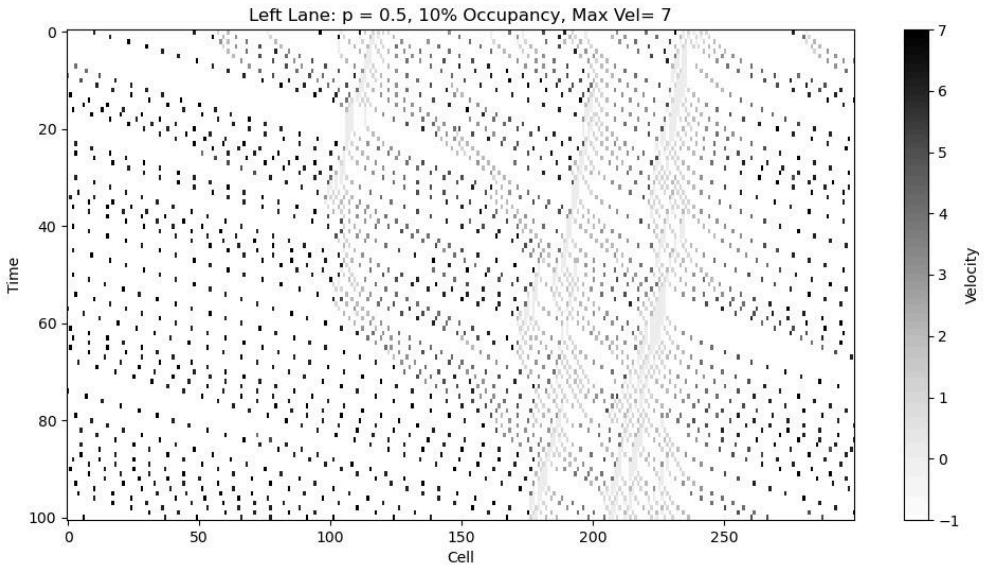

Both Lanes, p = 0, Left: 50% Occupancy, Max Vel= 7, Right: 50% Occupancy, Max Vel= 5
Time
Cell
Velocity

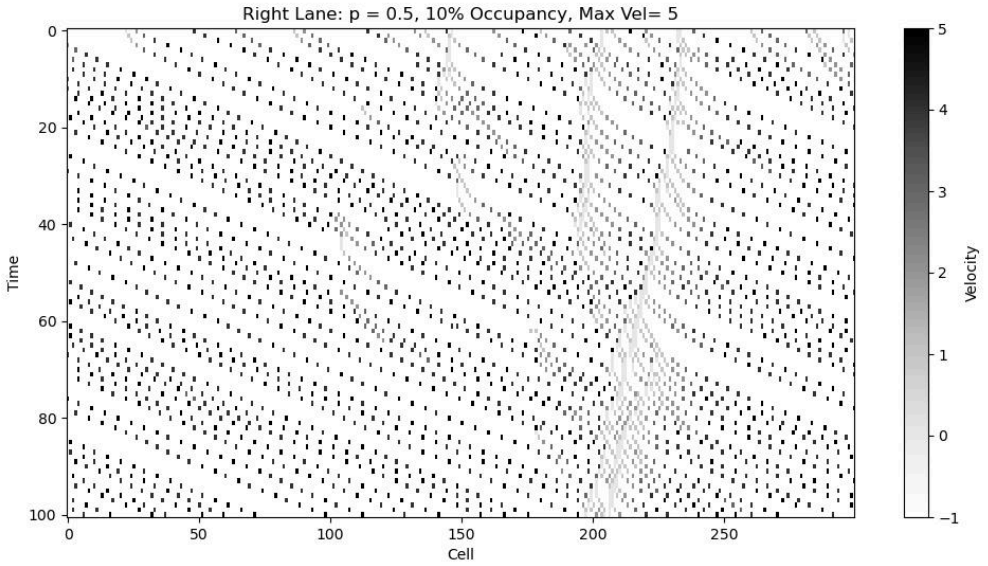

Left Lane: p = 0, 50% Occupancy, Max Vel= 7
Time
Cell
Velocity



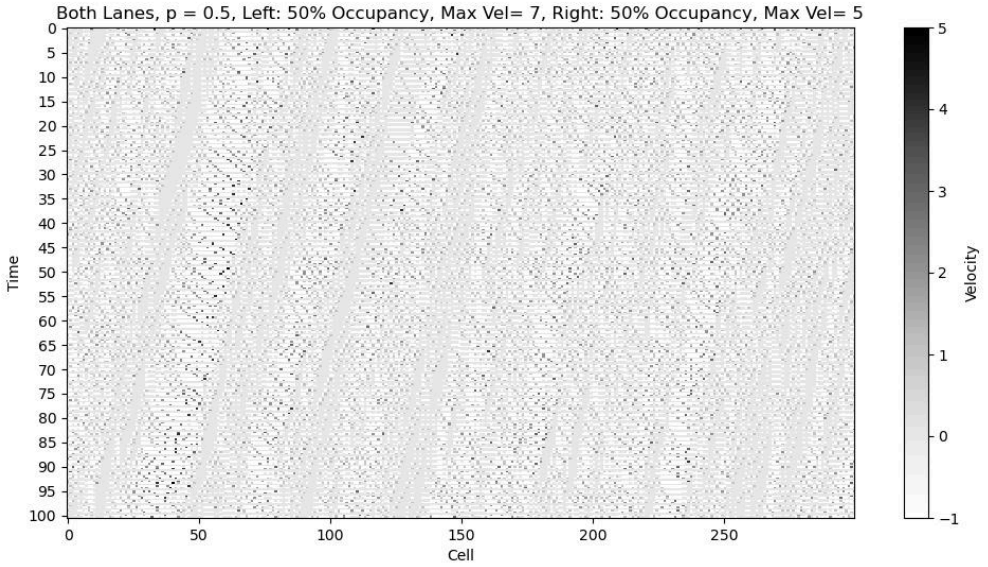
In [ ]:
```
sim = 5
p = 0.5
occupancy_percents = [10, 10]
max_vels = [7, 5]
num_cells = 300
num_time_steps = 100
create_plots(True, True, True, max_vels, occupancy_percents, num_cells, num_time_steps, p, sim, False)
```

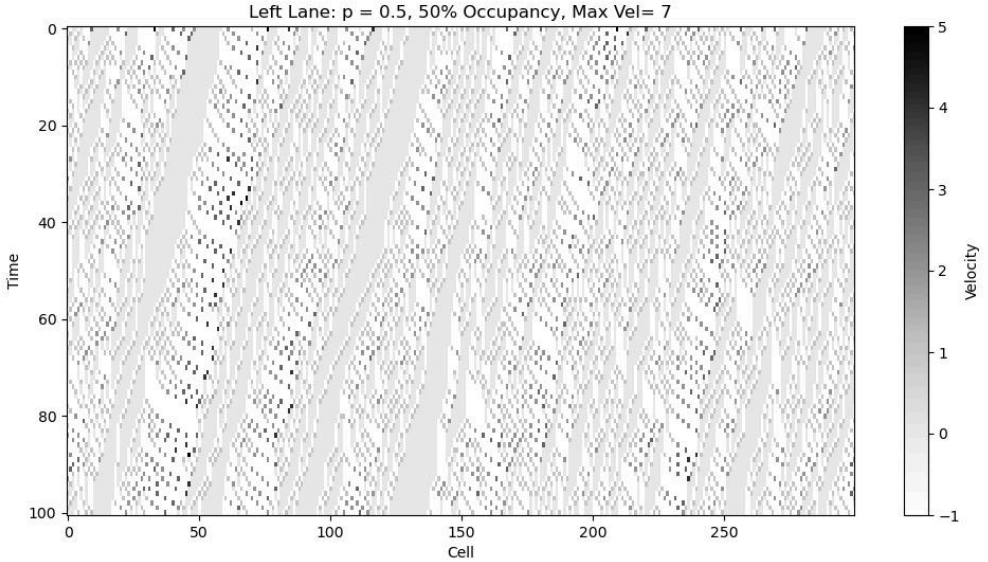


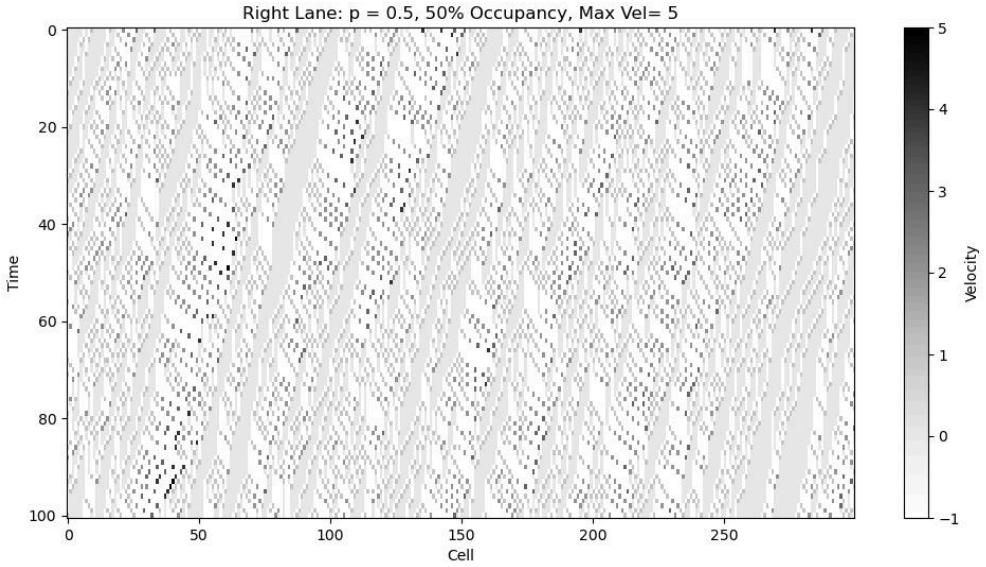




In [ ]:
```
sim = 6
p = 0.5
occupancy_percents = [50, 50]
max_vels = [7, 5]
num_cells = 300
num_time_steps = 100
create_plots(True, True, True, max_vels, occupancy_percents, num_cells, num_time_steps, p, sim, False)
```


Both Lanes, p = 0.5, Left: 50% Occupancy, Max Vel= 7, Right: 50% Occupancy, Max Vel= 5
Time
Cell
Velocity


Left Lane: p = 0.5, 50% Occupancy, Max Vel= 7
Time
Cell
Velocity



There are a few notable things about these plots. They are extremely similar to both of the other models, depending on the dally factor p. As can be seen in the test cases, lane changing is fairly rare due to how cautious each driver is. There does appear to be slightly more lane changing in the 10% occupancy case, which seems to result in a very small increase in traffic in the lane with a higher speed limit.
Traffic also appears to occur in both lanes at around the same time, due to the fact that if there is traffic in one lane those vehicles might chnages lanes, resulting in traffic in the other lane.
However, in the case of no dally factor the prevalence of lane changing seems to decrease, most likely because at a non-critical occupancy every car will eventually be able to travel at the speed limit.
At higher occupancies even less lane changing would be expected, as the density of cars on the road will make it less likely all of the conditions to change lanes are met. This is expected, as it can be quite difficult to change lanes while driving in dense traffic.
These effects of the dally factor and the occupancy percent seem to work together to result in few lane changes: when there is no traffic cars do not need to change lanes and when there is traffic they are not able to.

In this way the effects seen in the plot seem to match what might be observed in real life. However, the drivers in this simulation are more cautious than drivers in real life (for example, a driver in real life will pass a slow car ahead of them regardless of if this car changing lanes will result in a collision), and therefore more lane changes would be observed.