ISC 5228
Markov Chain Monte Carlo

Traffic Modeling

1 Introduction

In this lab, we will explore the Nagel-Schreckenberg model. It is a minimal model for simulating the spontaneous formation of traffic jams on freeways without a recognizable cause (ex. accidents, lane closures, traffic lights, road patterns etc.) The original paper describing the model "A cellular automaton model for freeway traffic" was published in 1992; it explains how road congestion can emerge as the density of cars on roads increases, and illuminates the role of randomness and individual action.

2 Model

The Nagel-Schreckenberg model (NSM) is a discrete model in which positions (x_i) and velocities (v_i) of N cars traveling on a (single lane) freeway of length M take integer values.

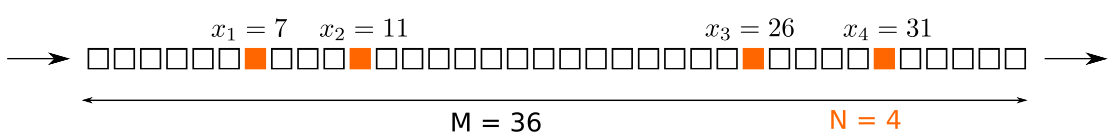


Figure 1: A example set up illustrating the different variables in the problem.

Periodic boundary conditions are applied on the freeway, so that a car leaving from one end, re-emerges from the other end. Thus, the freeway may be visualized as a circular track.

The speed limit is denoted by $v_{\max}$ (= 5 say). Time is also discretized into steps. At each step, the positions and velocities of all the cars are updated according to the following rules in order:

1. **Acceleration**: Unless the speed limit is violated, all cars increase their velocity by 1 unit.

$$u = \min(v_i(t) + 1, v_{\max}) \tag{1}$$

2. **Deceleration**: If the distance to car in front ($d = x_{i+1}(t) - x_i(t)$)[1] is smaller than u from eqn (1), the velocity is reduced to a safe level to avoid collision.

$$u = \min(u, d - 1) \tag{2}$$

[1] assuming the cars are indexed in sequence

3. **Randomization**: The speed of all moving cars ($v_i(t) > 0$) is reduced by 1 unit with a probability p.

$$u = \begin{cases} \max(0, u-1), & \text{if } p \geq U[0,1) \\ u, & \text{otherwise.} \end{cases} \tag{3}$$

4. **Update Position and Velocity**: All cars are moved forward by the number of cells equal to their velocity.

$$v_i(t+1) = u \tag{4}$$
$$x_i(t+1) = x_i(t) + v_i(t+1) \tag{5}$$

2.1 Periodic boundary conditions

Due to periodic boundaries we need to make two corrections:

1. if $x_i > M$, we set $x_i = x_i - M$.[2]
2. for the car with the largest x_i ($x_{\max}$), the distance to the car in front ($x_{\min}$) is computed according to $d = x_{\min} - x_{\max} + M$.

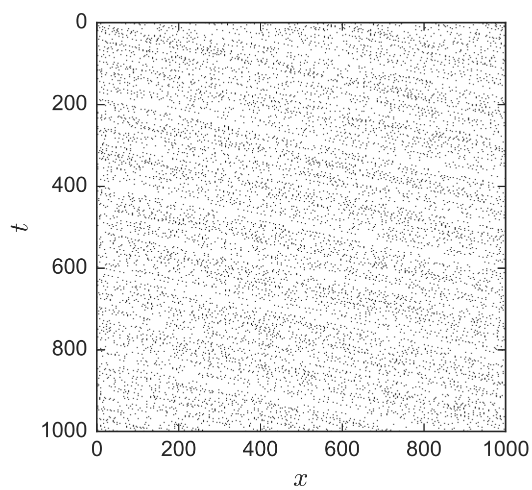
3 Exercises

As the base case, consider a road of length $M = 1000$, with $N = 50$ cars. All cars are initially stationary, i.e. $v_i(t = 0) = 0$. The maximum speed $v_{\max} = 5$ and $p = 1/3$ (for randomization).

For initial positions pick any valid starting configuration; two cars cannot occupy the same position, and for $i < M$, $x_{i+1} > x_i$.

Perform 1000 "burn-in" updates, to "forget" the initial configuration. Perform all tasks below starting with the configuration obtained after burn-in.

1. **Flow Trace**: Run the simulation for 1000 steps, and make a plot of the position and time such as the one below.



[2] numpy.remainder may be useful.

2. For $N = 50$, determine the average speed $\bar{v}$ of the cars over the entire run with three independent replicas. The average flow is defined as $f(N) = N\bar{v}$.

3. **Fundamental Diagram**: Repeat the step above by varying the number of cars between $N = 50$ and $N = 300$. Plot $\bar{v}(N)$ and $f(N)$ as a function of N. Report and explain your observations.

4. Estimate the (optimal) value of N that gives the best flow? Compare the flow traces at $N = 50$, the optimal N, and $N = 300$, and qualitatively describe the differences.

4 Programming Notes

The location and velocities of the cars may be stored in N dimensional (integer) arrays, say `carLoc` and `carVel`. Thus, the position and velocity of the i^{th} car is $x_i =$`carLoc[i]` and $v_i =$`carVel[i]`.

Initially, we may set `carVel = 0`, and set positions by sorting a random permutation.[3] The burn-in phase should remove any artifacts introduced by this initialization.

Updating car positions and velocities is the core subroutine. You may use the following:

```
def updatePosVel(carLoc, carVel, M, vmax, prand):
    """updates car positions and velocities over 1 step"""

    numCars = len(carLoc)

    # acceleration
    u = np.minimum(carVel + 1, vmax)

    # deceleration
    d = np.append(np.diff(carLoc), carLoc[0] - carLoc[-1]) # PBC
    d = np.remainder(d, M)    # PBC
    u = np.minimum(u, d-1)

    # randomization
    idx     = np.random.rand(numCars) - p < 0
    u[idx] = np.maximum(0, u[idx]-1)

    # update
    Loc    = np.remainder(carLoc + u, M)

    return Loc, u
```

A burn in routine that initializes, and returns `carLoc` and `carVel` at the end of the run.

```
def BurnInRun(N=200, M=1000, vmax=5, prand=0.3333, ncycles=M):
```

[3]Try `np.sort(np.random.permutation(M)[0:N])` in python, or `sort(randperm(M,N))` in Octave/Matlab.

```
    # initialize carVel and carLoc
    ...

    for t in range(1, ncycles):
        carLoc, carVel = updatePosVel(carLoc, carVel, M, vmax, prand)
    return carLoc, carVel
```

After the burn-in run, we carry out the production run, which looks a lot like the burn-in run, except that we track more variables. Two in particular are (i) the average velocity at any time, and (ii) an occupancy matrix (size $M\times$ `ncycles`) which is sparse, and has ones in positions occupied by cars and zero elsewhere.

```
def MainRun(ncycles, numCars, M, vmax5, p):
        """"You will have to modify this to keep track of flowtrace"""

    # burn-in
    carLoc, carVel = BurnInRun(numCars, M, vmax, p, ncycles=M)

        # track avg velocity
    vavg = np.zeros(ncycles)
    vavg[0] = np.mean(carVel)

    for t in range(1, ncycles):
        carLoc, carVel = updatePosVel(carLoc, carVel, M, vmax, p)
        vavg[t] = np.mean(carVel)

    return np.mean(vavg)
```

The flowtrace is a visualization of the occupancy matrix, say `matOcc`. You can image it using a command like the following in python:[4]

```
plt.imshow(matOcc, cmap="Greys", interpolation='nearest')
```

[4]In Matlab, try `imagesc`, or `spy`.