Velib Simulation (with code)

January 14, 2019

1 MPRO - FAT - Velib System Simulation

Guillaume Dalle & Julien Khamphousone

```
In [1]: # Package installation, if necessary
        using Pkg
        Pkg.update()
        # Pkg.add("Random")
        # Pkg.add("StatsBase")
        # Pkg.add("DataFrames")
        # Pkg.add("Statistics")
        # Pkg.add("ProgressMeter");
```

```
  Updating registry at `~/.julia/registries/General`
  Updating git-repo `https://github.com/JuliaRegistries/General.git`
 Resolving package versions...
  Updating `~/.julia/environments/v1.0/Project.toml`
 [no changes]
  Updating `~/.julia/environments/v1.0/Manifest.toml`
 [no changes]
```

```
In [2]: # Imports
        using Random
        using StatsBase
        using DataFrames
        using Statistics
        using ProgressMeter;
```

```
In [3]: # Set a random seed to ensure reproducible results
        Random.seed!(63);
```

The code for the simulation is contained in the following cells, and available at https://gitlab.com/gdalle/fat-velib.

Useful data

```
In [4]: tmi = typemax(Int64)
```

```
nb_stations = 5

## TRANSITION RATES

# Average trip duration from i to j [min]
tau = [
    [tmi  3    5    7    7];
    [2    tmi  2    5    5];
    [4    2    tmi  3    3];
    [8    6    4    tmi  2];
    [7    7    5    2    tmi]]

# Rate of transition from t_ij to j [min^-1]
lambda_trip_station = 1 ./ tau

# Frequency of bike departures for each station [min^-1]
departures = [2.8,  3.7,  5.5,  3.5,  4.6] ./ 60

# Probability that a bike will leave from station i to station j
routing = [
    [0      0.22         0.32         0.2            0.26];
    [0.17         0       0.34         0.21         0.28];
    [0.19         0.26         0       0.24         0.31];
    [0.17         0.22         0.33         0       0.28];
    [0.18         0.24         0.35         0.23    0]]

# Rate of transition from i to t_ij [min^-1]
lambda_station_trip = departures .* routing

## INITIAL VALUES

# Number of bikes parked in station i
nb_bikes_station = [20,  16,  17,  13,  18]

# Number of bikes on the trip t_ij
nb_bikes_trip = [
    [0  1  0  0  0];
    [1  0  1  0  0];
    [0  1  0  1  0];
    [0  0  1  0  1];
    [0  0  0  1  0];
    ]

function define_random_initial_conditions()

    # Here we simulate random initial conditions

    number_of_bikes = 20+16+17+13+18+8
```

```julia
    nb_bikes_station = Array{Int64, 1}(undef, 0)
    nb_bikes_trip = [rand(0:2) for _ in 1:nb_stations, _ in 1:nb_stations]
    for i in 1:nb_stations
        nb_bikes_trip[i, i] = 0
        push!(nb_bikes_station, rand(1:number_of_bikes))
    end
    while sum(nb_bikes_station) != number_of_bikes-sum(nb_bikes_trip)
        station = rand(1:nb_stations)
        if nb_bikes_station[station] > 0
            nb_bikes_station[station] -= 1
        end
    end
    # Number of bikes on the trip t_ij

    return nb_bikes_station, nb_bikes_trip
end;
```

Colony model

```julia
In [5]: mutable struct Colonies
            lambda_station_trip::Array{Float64, 2}
            lambda_trip_station::Array{Float64, 2}
            nb_stations::Int # Number of stations in the bike-sharing system
            current_time::Float64 # Current time of the colonies process
            nb_bikes_station::Array{Int64, 1} # Number of bikes parked in station i
            nb_bikes_trip::Array{Int64, 2} # Number of bikes on the trip t_ij
        end

        Colonies(
            lambda_station_trip::Array{Float64, 2},
            lambda_trip_station::Array{Float64, 2}
        ) = Colonies(
            lambda_station_trip,
            lambda_trip_station,
            size(lambda_station_trip)[1],
            -Inf,
            Array{Int64, 1}(undef, 0),
            Array{Int64, 2}(undef, 0, 0)
        )

        function initialize!(
            col::Colonies,
            nb_bikes_station::Array{Int64, 1},
            nb_bikes_trip::Array{Int64, 2}
        )
            col.nb_bikes_station::Array{Int64, 1} = deepcopy(nb_bikes_station)
            col.nb_bikes_trip::Array{Int64, 2} = deepcopy(nb_bikes_trip)
            col.current_time = 0.
```

```
end

function transition_station_trip!(col::Colonies, i::Int, j::Int)
    col.nb_bikes_station[i] -= 1
    col.nb_bikes_trip[i, j] += 1
end

function transition_trip_station!(col::Colonies, i::Int, j::Int)
    col.nb_bikes_trip[i, j] -= 1
    col.nb_bikes_station[j] += 1
end;
```

Markov process simulation

In [6]:
```
"""
    Simulate the next transition of the Markov process.
"""
function simulate_next_transition(col::Colonies)
    # Compute current transition rates
    q_trip_station = col.lambda_trip_station .* col.nb_bikes_trip
    q_station_trip = col.lambda_station_trip .* convert(
        Array{Float64}, col.nb_bikes_station .> 0)

    # The sum of all transition rates defines the parameter of
    # the exponential distribution giving the time to the next transition
    total_transition_rate = sum(q_station_trip) + sum(q_trip_station)
    time_to_next_transition = -(1 / total_transition_rate) * log(rand())

    # Each transition has a likelihood that is proportional to its transition rate
    transitions = vcat(
        vec([
            (transition_type="station_trip", i=i, j=j)
            for i in 1:col.nb_stations, j in 1:col.nb_stations
        ]),
        vec([
            (transition_type="trip_station", i=i, j=j)
            for i in 1:col.nb_stations, j in 1:col.nb_stations
        ]),
    )
    transition_probas = vcat(
        vec([
            q_station_trip[i, j] / total_transition_rate
            for i in 1:col.nb_stations, j in 1:col.nb_stations
        ]),
        vec([
            q_trip_station[i, j] / total_transition_rate
            for i in 1:col.nb_stations, j in 1:col.nb_stations
        ])
```

```julia
    )

    # Sample a transition with the weights given above
    next_transition = sample(transitions, Weights(transition_probas))

    return next_transition, time_to_next_transition
end

"""
    Simulate a trajectory of the Markov process with a given initial repartition.
"""
function simulate(
        col::Colonies,
        max_time::Float64,
        nb_bikes_station::Array{Int64, 1},
        nb_bikes_trip::Array{Int64, 2}
    )

    col::Colonies = deepcopy(col)
    initialize!(col, nb_bikes_station, nb_bikes_trip)

    transitions_history = DataFrame(
        time = Float64[],
        transition_type = String[],
        i = Int64[],
        j = Int64[]
    )

    empty_station_duration = zeros(Float64, col.nb_stations)

    while col.current_time < max_time

        # Find the next transition
        next_transition, time_to_next_transition = simulate_next_transition(col)

        # If any station is empty, record the time it spent that way
        interval_duration = min(time_to_next_transition, max_time - col.current_time)
        for i in 1:col.nb_stations
            if col.nb_bikes_station[i] == 0
                empty_station_duration[i] += interval_duration
            end
        end

        if col.current_time + time_to_next_transition > max_time
            # The next transition will not have time to happen
            break
        end
```

```
        # Store the transition in the history log
        push!(transitions_history, (
            time = col.current_time + time_to_next_transition,
            transition_type = next_transition.transition_type,
            i = next_transition.i,
            j = next_transition.j
        ))

        # Perform the transition by modifying the Colonies object
        col.current_time += time_to_next_transition
        if next_transition.transition_type == "station_trip"
            transition_station_trip!(col, next_transition.i, next_transition.j)
        elseif next_transition.transition_type == "trip_station"
            transition_trip_station!(col, next_transition.i, next_transition.j)
        end

    end

    empty_station_duration ./= max_time

    return col, transitions_history, empty_station_duration
end

"""
    For each station, compute:
    - the frequency of emptiness at the end time
    - the mean duration spent empty
    over several trajectory simulated from the same initial state.
"""
function estimate_emptiness(
        col::Colonies,
        max_time::Float64,
        nb_simulations::Int64,
        nb_bikes_station::Array{Int64, 1},
        nb_bikes_trip::Array{Int64, 2}
    )

    empty_ends = zeros(nb_simulations, col.nb_stations)
    empty_durations = zeros(nb_simulations, col.nb_stations)

    @showprogress "Simulating trajectories " for sim in 1:nb_simulations

        new_col, transitions_history, empty_station_duration = simulate(
            col, max_time, nb_bikes_station, nb_bikes_trip)

        empty_ends[sim, :] = (new_col.nb_bikes_station .== 0)
        empty_durations[sim, :] = empty_station_duration
```

```
    end

    empty_end_freq = mean(empty_ends, dims=1)
    empty_end_freq_uncertainty = 1.96 * (
        std(empty_ends, dims=1) / sqrt(nb_simulations))

    empty_duration_mean = mean(empty_durations, dims=1)
    empty_duration_mean_uncertainty = 1.96 * (
        std(empty_durations, dims=1) / sqrt(nb_simulations))

    emptiness = DataFrame(
        empty_end_freq=empty_end_freq[:],
        empty_end_freq_uncertainty=empty_end_freq_uncertainty[:],
        empty_duration_mean=empty_duration_mean[:],
        empty_duration_mean_uncertainty=empty_duration_mean_uncertainty[:],
    )

    return emptiness
end

# Initial repartition of one bike, put arbitrarily in the first station
monobike_nb_bikes_station = zeros(Int64, 5)
monobike_nb_bikes_station[1] = 1
monobike_nb_bikes_trip = zeros(Int64, 5, 5)

"""
    For each station, compute:
    - the frequency of emptiness at the end time
    - the mean duration spent empty
    over several trajectores simulated from the monobike initial state.
"""
function estimate_emptiness_monobike(
        col::Colonies,
        max_time::Float64,
        nb_simulations::Int64,
    )
    return estimate_emptiness(
        col,
        max_time,
        nb_simulations,
        monobike_nb_bikes_station,
        monobike_nb_bikes_trip
    )
end;
```

Stationary distribution computations

```
In [7]: """
            Solve the traffic equations for the alpha_i with a linear system.
```

```julia
"""
function compute_alpha_station(col::Colonies)
    M = zeros(col.nb_stations, col.nb_stations)
    M[1, :] = ones(col.nb_stations)
    for i in 2:col.nb_stations, j in 1:col.nb_stations
        if i != j
            M[i, i] += col.lambda_station_trip[i, j]
            M[i, j] = - col.lambda_station_trip[j, i]
        end
    end

    b = zeros(col.nb_stations)
    b[1] = 1

    alpha_station = M \ b
    return alpha_station
end

"""
    Solve the traffic equations for the alpha_t_ij based on the alpha_i.
"""
function compute_alpha_trip(col::Colonies, alpha_station::Array{Float64, 1})
    alpha_trip = zeros(Float64, col.nb_stations, col.nb_stations)
    for i in 1:col.nb_stations, j in 1:col.nb_stations
        if i == j
            alpha_trip[i, i] = 0
        else
            alpha_trip[i, j] = alpha_station[i] * (
                col.lambda_station_trip[i, j] / col.lambda_trip_station[i, j]
            )
        end
    end
    return alpha_trip
end

"""
    Solve all the traffic equations.
"""
function compute_alpha(col::Colonies)
    alpha_station = compute_alpha_station(col)
    alpha_trip = compute_alpha_trip(col, alpha_station)
    return alpha_station, alpha_trip
end

"""
    Compute the stationary probability of emptiness with one bike.
"""
function emptiness_proba_monobike(
```

```
        alpha_station::Array{Float64, 1},
        alpha_bike::Array{Float64, 2}
)
        G_1 = sum(alpha_station) + sum(alpha_trip)
        emptiness_probability = 1 .- alpha_station / G_1
        return emptiness_probability
end;
```

1.1 3. Calibration

The numerical parameters are computed in and imported from the `data.jl` file.

```
In [8]: lambda_station_trip
```

```
Out[8]: 5Œ5 Array{Float64,2}:
 0.0         0.0102667  0.0149333  0.00933333  0.0121333
 0.0104833   0.0        0.0209667  0.01295     0.0172667
 0.0174167   0.0238333  0.0        0.022       0.0284167
 0.00991667  0.0128333  0.01925    0.0         0.0163333
 0.0138      0.0184     0.0268333  0.0176333   0.0
```

```
In [9]: lambda_trip_station
```

```
Out[9]: 5Œ5 Array{Float64,2}:
 1.0842e-19  0.333333    0.2         0.142857    0.142857
 0.5         1.0842e-19  0.5         0.2         0.2
 0.25        0.5         1.0842e-19  0.333333    0.333333
 0.125       0.166667    0.25        1.0842e-19  0.5
 0.142857    0.142857    0.2         0.5         1.0842e-19
```

1.2 4. Simulation of a trajectory

```
In [10]: nb_bikes_station
```

```
Out[10]: 5-element Array{Int64,1}:
 20
 16
 17
 13
 18
```

```
In [11]: nb_bikes_trip
```

```
Out[11]: 5Œ5 Array{Int64,2}:
 0  1  0  0  0
 1  0  1  0  0
 0  1  0  1  0
 0  0  1  0  1
 0  0  0  1  0
```

```
In [12]: max_time = 150 * 60. # we count the time in minutes
         nb_simulations = 1000
```

```
Out[12]: 1000
```

```
In [13]: col = Colonies(lambda_station_trip, lambda_trip_station);
```

```
In [14]: new_col, transitions_history, empty_station_duration = simulate(
             col, max_time, nb_bikes_station, nb_bikes_trip
         )
         head(transitions_history)
```

```
 Warning: `head(df::AbstractDataFrame)` is deprecated, use `first(df, 6)` instead.
   caller = top-level scope at In[14]:4
 @ Core In[14]:4
```

Out[14]:

	time	transition_type	i	j
1	0.433587	trip_station	4	3
2	0.8457	trip_station	5	4
3	0.881857	trip_station	3	2
4	1.10485	trip_station	3	4
5	1.79372	trip_station	2	1
6	3.06025	trip_station	1	2

We displayed the history of transitions for one simulated trajectory

1.3 5, 6, 8. Probability of emptiness and confidence intervals

```
In [15]: estimate_emptiness(col, max_time, nb_simulations, nb_bikes_station, nb_bikes_trip)
```

```
Simulating trajectories 100%|| Time: 0:00:21
```

Out[15]:

	empty_end_freq	empty_end_freq_uncertainty	empty_duration_mean	empty_duration_mean_unc
1	0.009	0.00585641	0.00784925	0.00118344
2	0.032	0.0109141	0.0226715	0.00198819
3	0.151	0.0222032	0.122304	0.00285211
4	0.043	0.0125795	0.0286295	0.0022124
5	0.098	0.018437	0.0780364	0.00272854

Here we displayed, in order of columns: - The frequency of emptiness at the end time (150h) for every station - The uncertainty on that value - The mean duration each station spends empty - The uncertainty on that value

For instance, the probability for station 3 of finishing the simulation with no bike is estimated at 0.151 ± 0.022, while its expected empty duration is estimated at $(0.122 \pm 0.003) \times 150h$.

1.4 7. Influence of initial conditions

We first simulated our model with the initial conditions provided in the data.

For this setting, we obtained the following percentage of time when each station is empty:

$$\begin{bmatrix} \textit{station} & \textit{mean emptiness duration } (\%) \\ 1 & 0.780.12 \\ 2 & 2.270.20 \\ 3 & 12.230.29 \\ 4 & 2.860.22 \\ 5 & 7.800.27 \end{bmatrix}$$

In [16]:
```
random_nb_bikes_station, random_nb_bikes_trip = define_random_initial_conditions();
```

In [17]:
```
random_nb_bikes_station
```

Out[17]:
```
5-element Array{Int64,1}:
  4
  9
  0
 18
 39
```

In [18]:
```
random_nb_bikes_trip
```

Out[18]:
```
5×5 Array{Int64,2}:
 0  1  1  1  2
 0  0  2  2  0
 0  0  0  1  1
 2  1  1  0  2
 2  1  0  2  0
```

In [19]:
```
estimate_emptiness(col, max_time, nb_simulations, nb_bikes_station, nb_bikes_trip)
```

```
Simulating trajectories 100%|| Time: 0:00:17
```

Out[19]:

	empty_end_freq	empty_end_freq_uncertainty	empty_duration_mean	empty_duration_mean_unc
1	0.006	0.00478897	0.00824964	0.00120917
2	0.044	0.0127183	0.0244554	0.00201993
3	0.14	0.0215172	0.125485	0.00278177
4	0.034	0.0112383	0.0266072	0.00213945
5	0.112	0.0195564	0.079548	0.00295704

We then tried randomly-defined intial conditions (printed above).

For this new setting, we obtained the following percentage of time when each station is empty:

$$\begin{bmatrix} \textit{station} & \textit{mean emptiness duration } (\%) \\ 1 & 0.820.12 \\ 2 & 2.450.20 \\ 3 & 12.540.28 \\ 4 & 2.660.21 \\ 5 & 7.950.30 \end{bmatrix}$$

The initial conditions will always have an influence on the result, because unless the starting point is already the stationary distribution the process will never reach it exactly. However we can suspect this influence is negligible, since the confidence intervals on the estimated probabilities overlap.

1.5 9. Stationary state approximation

If we consider that after 150 hours, the chain has already mixed more than enough, then the result of the question 8 may be better to approximate the stationary probability than the result of question 5.

Indeed, the percentage of time when the station is empty (Q8) may include lots of observations of the (near-)stationary probability (eg. the last 100 hours out of 150), and so the mean emptiness duration can exploit a large part of the trajectory. On the other hand, the end time emptiness frequency only considers one observation per trajectory.

Another way to answer is to note that the confidence intervals are much tighter with the "empty duration" method than with the "emptiness frequency".

1.6 10. Better precision

In line with the previous question, it would be better to perform more simulations to increase the precision, because 150h seems to be a well-chosen duration to ensure near-stationarity.

1.7 11. Traffic equations

The traffic equations in this closed migration process are given by:

$$\forall i, \quad \alpha_i \sum_{j \neq i} \lambda_{it_{ij}} = \sum_{j \neq i} \alpha_{t_{ji}} \lambda_{t_{ji}i} \tag{1}$$

$$\forall i \neq j, \quad \alpha_{t_{ij}} \lambda_{t_{ij}j} = \alpha_i \lambda_{it_{ij}} \tag{2}$$

Combining both equations, we find that the α_i are the solution of a linear system given by

$$\forall i, \quad \alpha_i \left(\sum_{j \neq i} \lambda_{it_{ij}} \right) - \sum_{j \neq i} \alpha_j \lambda_{jt_{ji}} = 0$$

Replacing the first of those constraints (which is redundent) by

$$\sum_i \alpha_i = 1$$

allows us to solve the system without getting the trivial solution $\forall i, \alpha_i = 0$.

The $\alpha_{t_{ij}}$ are then obtained from the α_i with the second traffic equation. This two-step method is useful because we only have to solve a system in N_s variables, and not N_s^2.

```
In [20]: alpha_station, alpha_trip = compute_alpha(col);
```

```
In [21]: alpha_station
```

```
Out[21]: 5-element Array{Float64,1}:
          0.21452558088665818
          0.2059815546761553
          0.1814915242044868
          0.20641426572971042
          0.19158707450298923
```

```
In [22]: alpha_trip
```

```
Out[22]: 5×5 Array{Float64,2}:
          0.0          0.00660739  0.0160179   0.0140157   0.0182204
          0.00431875   0.0         0.00863749  0.0133373   0.0177831
          0.0126439    0.0086511   0.0         0.0119784   0.0154722
          0.0163755    0.0158939   0.0158939   0.0         0.00674287
          0.0185073    0.0246764   0.0257046   0.00675664  0.0
```

1.8 12. One-bike state space

In the one-bike case, the state space is

$$E = \left\{ \mathbf{n} = \left((n_i)_i, (n_{t_{ij}})_{(i,j),i \neq j} \right) \quad | \quad \sum_i n_i + \sum_{(i,j),i \neq j} n_{t_{ij}} = 1 \right\}$$

In other words, there is one state per station and one per trip.

1.9 13. One-bike emptiness probabilities

The normalization factor of the stationary distribution is given by

$$G_1 = \sum_i \alpha_i + \sum_{i \neq j} \alpha_{t_{ij}}$$

And the probability of a station being empty is simply:

$$\mathbb{P}(n_i = 0) = 1 - \frac{\alpha_i}{G_1}$$

```
In [23]: emptiness_proba_monobike(alpha_station, alpha_trip)
```

```
Out[23]: 5-element Array{Float64,1}:
          0.8321704317214969
          0.8388546706096625
          0.8580139299585932
          0.8385161482338817
          0.8501158888898838
```

All stations have more or less the same stationary probability of being empty, between 83% and 85%.

1.10 14. Comparison with one-bike simulations

```
In [24]: estimate_emptiness_monobike(col, max_time, nb_simulations)
```

```
Simulating trajectories 100%|| Time: 0:00:03
```

Out[24]:

	empty_end_freq	empty_end_freq_uncertainty	empty_duration_mean	empty_duration_mean_unc
1	0.849	0.0222032	0.830747	0.00137401
2	0.842	0.0226182	0.838869	0.00123474
3	0.849	0.0222032	0.858803	0.00092348
4	0.839	0.0227912	0.838149	0.00126756
5	0.838	0.0228482	0.850638	0.00108675

Fortunately, the theoretical values computed above mostly fall within the confidence intervals of the simulation.