Understanding Input Data Requirements and Quantifying Uncertainty for Successfully Modelling Smart Cities

**Technical Summary / Overview**

*This document reports on the aims, methods, and findings of the “Understanding Input Data Requirements and Quantifying Uncertainty for Successfully Modelling Smart Cities” project. It was a 6-month collaboration between the University of Leeds (Mike Adcock, Nick Malleson, Jon Ward and Alison Heppenstall) and Improbable (Dan Tang and Jon Coello).*

*The project code and associated documentation are in a (currently private) GitHub repository:* <https://github.com/nickmalleson/keanu-post-hackathon>. The source code is in the directory: keanu-post-hackathon/keanu-examples/stationSim/src/main/java/StationSim.

# Initial Project Aims

The project was part of wider efforts to develop methods that can be used to better understand uncertainty in individual-level models and, ultimately, to develop urban simulations that help us to better understand cities and assist policy makers with managing risk and uncertainty.

The aim of this project was to explore the uncertainty associated with an individual-level model of pedestrian movements in a city. Specifically, the project began to experiment with the volumes of hypothetical sensed data (e.g. those produced by individual people or autonomous footfall sensors) that are required to reduce the uncertainty in a simulation to acceptable levels.

A simple pedestrian model was created (see Figure 1) to represent individual agents who enter through three entrances, move across a space, and enter through one of two exits. This could represent, e.g., a train arriving at a train station and passengers leaving through the concourse. The model is first executed to generate a hypothetical reality (a ‘pseudo-truth’[[1]](#footnote-1)). Following this, the pseudo-truth data are sampled at varying intervals (both spatial and temporal) and have noise added. This will simulate the data generation processes that occur in real cities. The *keanu* software library, that is currently under development by Improbable, is used to explore the volume and quality of data that are required for the individual-level model to be able to simulate the truth data to an acceptable level of uncertainty.

# Methods

## Agent Based Model - *StationSim*

An agent based model (*StationSim*) of a hypothetical train station was created to model crowding (Figure 1). While simple in nature, this model was designed to include stochastic elements and so produced differing results on subsequent runs.

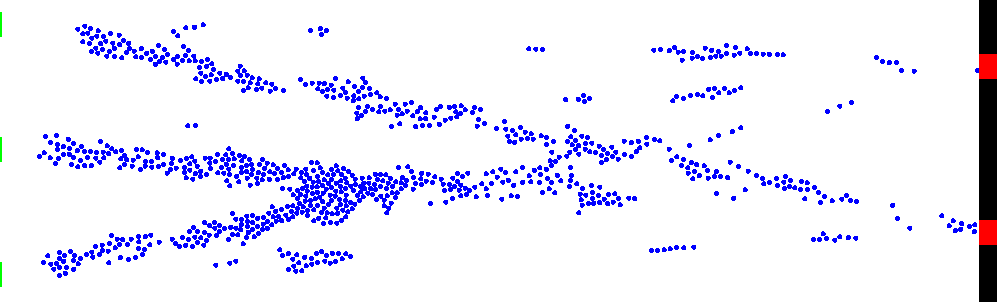
**

Figure 1. A snapshot of the simple, hypothetical model that is used here. Agents arrive from the green entrances on the left and move towards the red exits on the right. During the course of the runtime crowding occurs in different areas.

*StationSim* allows for the many parameters to be set including number of entrances and exits, the interval at which people can pass through entrances/exits and the probability that a person from a given entrance will choose a given exit. The parameter used for *StationSim* in this work is shown in Table 1.

Table 1. Parameters used for StationSim for this work

|  |  |
| --- | --- |
| Parameter | Value |
| areaWidth | 200.0 |
| areaHeight | 100.0 |
| numPeople | 700 |
| numEntrance | 3 |
| numExits | 2 |
| exitProbs | {{0.2, 0.8}, {0.3, 0.7}, {0.9, 0.1}} |
| exitInterval | 30 |
| entranceInterval | 2 |
| entranceSize | 10 |
| exitSize | 10 |

When StationSim is run agents are generated at a given interval and are assigned an exit to move towards. This exit assignment uses probabilities defined in the parameters for each entrance and exit. The maximum speed for the person is chosen from a random distribution (this random number is provided by *keanu*, as discussed later). When a person starts to approach another person in front of them they slow down. When they collide with a person in front they attempt to move around them by moving on the y axis. When people reach the exit they can pass through at set intervals. The number of people that can pass through an entrance or exit in a given step is also set in the model parameters. An explanation of how StationSim functions is presented as pseudocode in Figure 2.

## Random Variables in *StationSim*

There are three areas of StationSim with random decisions that give its stochastic nature. These are:

1. the choice of exits for each person;
2. the choice for maximum speed of a person; and
3. the choice of direction and speed a person travels on the y axis when the encounter a person ahead of them.

entranceInterval = How often an entrance will allow people to enter simulation

exitInterval = How often an exit will allow people to pass through an Exit simulation

entranceSize = The number of people who may pass through an entrance in a single step

exitSize = The number of people who may pass through an exit in a single step

toatoalnumPeople = Total number of people that can enter throughout the run of the simulation

equally space entrances along y axis at x

equally space exits along y axis at x

for each entrance:

peopleRemainigForEntrance = totalNumPeople / number of entrances (integer is division used)

for each simulation step:

if step % exitInterval == 0:

for each exit:

allow number of people through exit equal to exitSize

sequencer = All people sorted ascendingly by euclidean distance to their target exit

for each person in sequence:

for i in 0 to slowingDistance:

testPosition = new coordinates calculated by linear interpolation toward exit

if testPosition collides with another person:

reduce speedFactor

break

newPosition = new coordinantes calculated by linear interpolation towards exit

if newPosition does not collide with another person:

current postion = new position

else:

new position = randomly choose direction on y axis and calculate distance

if newPostition does not collide with other people:

currentPosition = newPosition

else:

newPostion = chosse other diection on y axis an calculate distance

if newPosition does not collide with other people:

currentPosition = new Position

if step % entrance interval == 0:

for each entrance:

create number of people equal to entrance size

peopleRemainingForEntrance -= number of people created

for each person:

choose target exit using probailities fom exitChoices

choose maxSpeedFactor from uniform distribution

Figure. 2: Pseudocode representing simplified version of StationSim

## Probabilistic Modelling with Keanu

The model itself was wrapped in a probabilistic programming library, under development by Improbable, called ‘Keanu’. Keanu allow the user to build Bayesian networks which can be used to make probabilistic predictions about complex problems. The code that utilised Keanu in this work is contained in the ‘Wrapper’[[2]](#footnote-2) class (discussed in the following section).

Figure 2 illustrates the process of generating the pseudo-truth data and demonstrates how it informs the probabilistic model. Recall that the input to *StationSim* is a random number generator (i.e. a list of random numbers that define the execution path of the model). The probabilistic model (implemented using keanu) *observes* some features of the pseudo-truth data (such as the number of agents in the simulation at each iteration) and uses these observations to create a posterior distribution over all of the model parameters. This posterior distribution is then sampled using MCMC (Metropolis-Hastings). If the posterior is more tightly constrained to the pseudo-truth data than the prior (i.e. the posterior without any observations) then the probabilistic model is successfully finding solutions that fit the observations (to within the noisiness of the observations). I.e. it is successfully finding the “true posterior”1.



Figure 2. An illustration of the modelling process and probabilistic model (from ABMUS 2018 paper/presentation)

## The *Wrapper* class

When the *main* function in wrapper is executed the pseudo-truth data are generated by running StationSim once. StationSim is wrapped by a function that takes a random number generator as an argument and returns model outputs defined by which option is selected in the static class variables. This effectively creates a black box of StationSim as this is the only interaction wrapper may have with the ABM. The random number generator that is passed as the sole input to StationSim effectively controls the execution of the model.

The probabilistic programming part of this work is contained in a function called keanu (named after the library it is using). This can be called with different observation intervals with a value of 0 meaning no observations of the pseudo-truth data are made (i.e. the prior). Running this function with no observation was used as a control for our experiments.

When Keanu is called, a Bayesian network is constructed of the probabilistic variables. This network can be visualised using the GraphViz (.dot) file that is written out during the run of this function. The truth data are observed with added noise. The posterior is then sampled using Metropolis Hastings. The aim is to essentially estimate the random numbers (or a distribution to draw them from) that were used as the input for StationSim. From the samples drawn, a certain number (set in one of the Wrapper class variables) are dropped from the start of sampling and they are then down-sampled (also set in class variables).

## Using *Wrapper* and *Analysis* classes

It is possible to choose what to *observe* (i.e. what information to give to keanu about the ‘real’ system) and what information that the model should output. Interesting experiments can be conducted when the observed data are not the same as those output by the model. *Options* are used in the code (see the *createOptions()* function of the *Wrapper* class) to chose what is observed and what should be output. The *createOptions()* function creates a map with the key as the option and the value as the length of array that is outputted per step from StationSim. In this output array all elements except the last one per step are observed.

The variables we get out of StationSim are controlled by which method we use from the *Analysis* class. This is found in the switch statement in *Wrapper.run()* where each case corresponds to an option. To add a new option add a new case and in that case call a method from analysis (which you will write yourself to give you desired output). Then add the option to createOptions() using the format to describe above (remember the last element is not observed but instead written out in our samples/truth). Finally set your new option in the static class variable ‘option’. The last element is not observed but is use to assess constraint of the model. If you want to observe and assess the same variable (as in option one) then use an array of size 2 with the same value in both positions. *(Note that this could be re-written more cleanly e.g. by using inheritance or enumerations, but is satisfactory for now).*

The random number generator that is the only input to StationSim can be limited to creating a finite number of random numbers, *n*. If *n* is reached and the model needs further random draws, then the finite list of numbers is recycled. This is analogous to passing *n* parameters to StationSim and asking the probabilistic model find values of those parameters that adequately constrain the posterior to the observations. The number *n* is set in *Wrapper.numRandomDoubles*, which for this work it is set at 10. We found that as numRandomDoubles is increase Wrapper takes a lot longer to run (?).

The number of samples to be taken from the posterior is set with *Wrapper.numSamples*. Dropping samples and down-sampling rates are set with *dropSamples* and *downSample* respectively. The *sigmaNoise* variable sets how noisy the observed values are after adding noise. We found that if this is set too high then the model is not properly constrained. Future work will explore the impacts of changing the amount of noise in the observations.

In the main function an array called *obIntervals* sets which observation intervals will be used (0 is no observations). The same truth value is used each time along with the same timestamp so we can see the separate files produced are related (and unique form other runs).

The write function is called to write out samples. As discussed previously it is the last value in the output array per step that is written out for both truth and samples. This means that it may or may not be the metric we used for observing depending on the option we have used. The truth data are written out for each observation Interval but it is the same data for each one.

# Results

Through the use of Improbable’s library, Bayesian inference was used to constrain the model output. This was achieved through estimation of the random numbers used within the model when the truth data were generated.

The ABM was run to produce truth data from the total number of people in the simulation at each step. This truth data was then observed in the Bayesian network and the posterior was sampled from with Metropolis Hastings. When compared to results generated using the same steps without observation taking place it was found that the model has been successfully constrained (Fig 3).

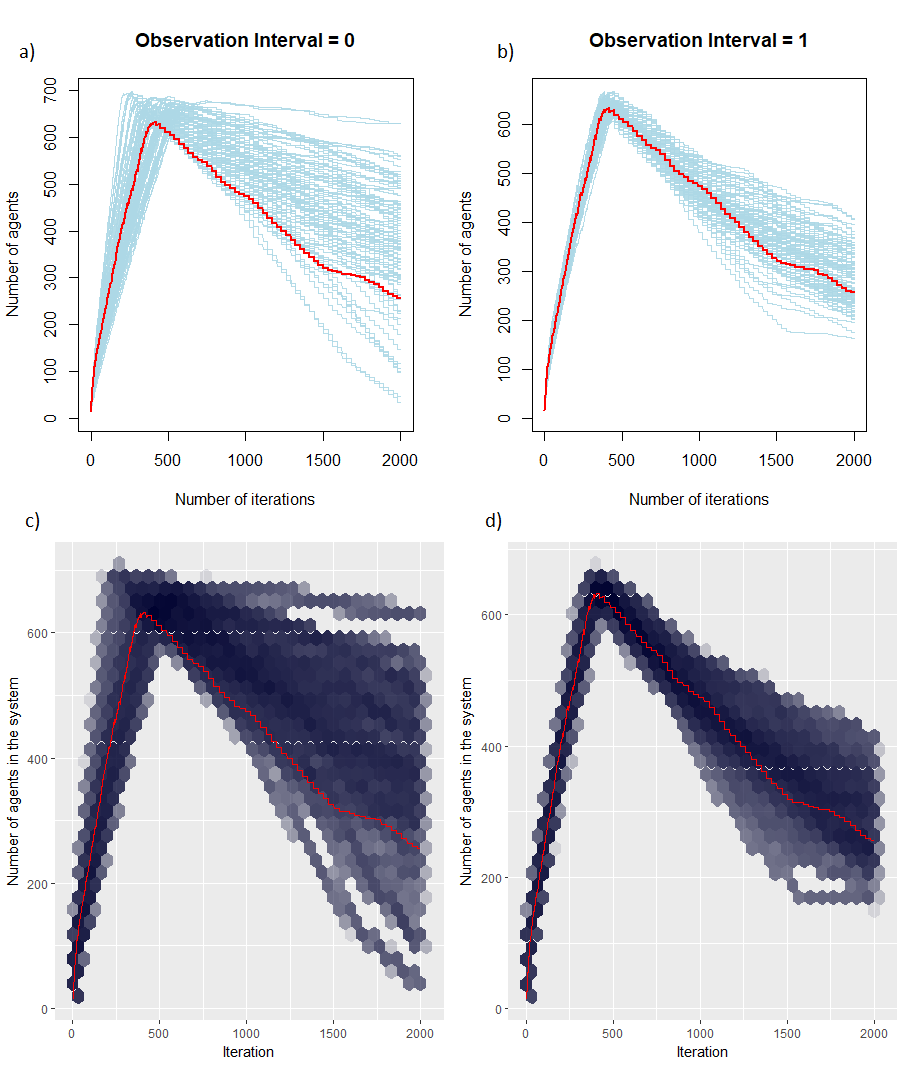
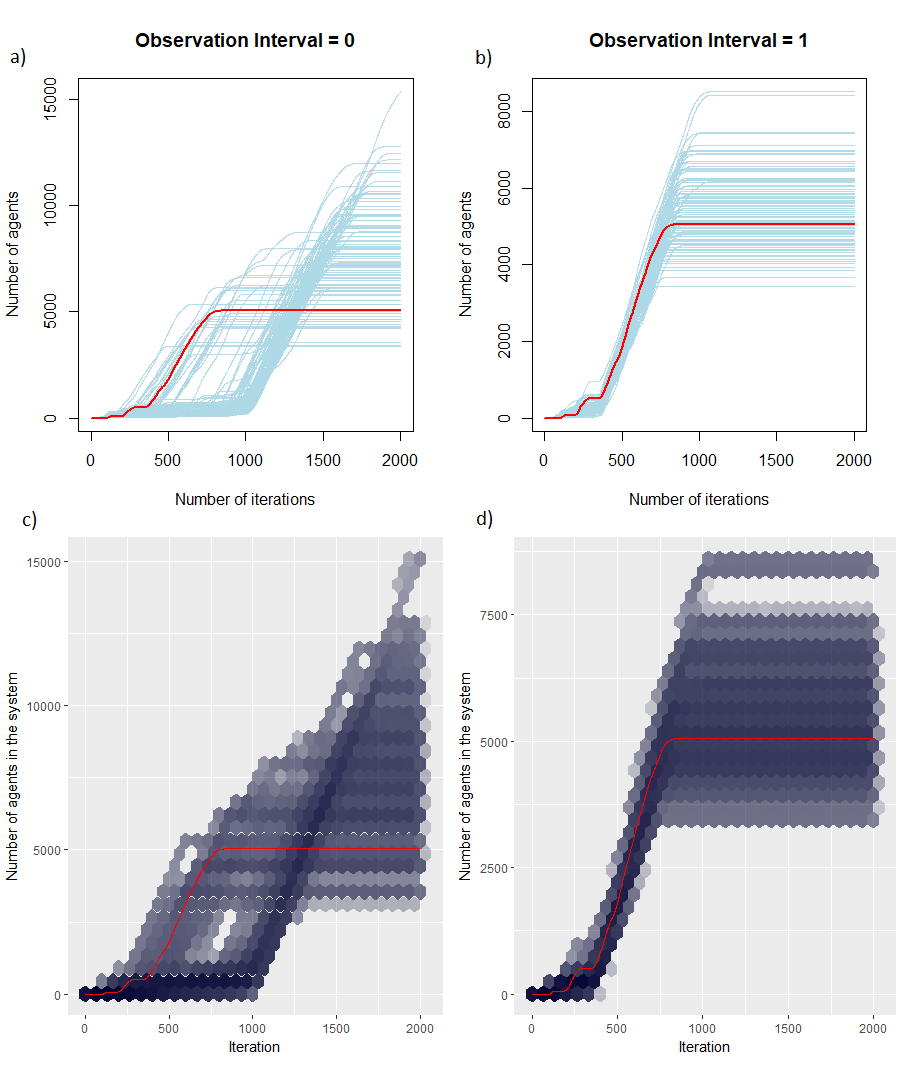


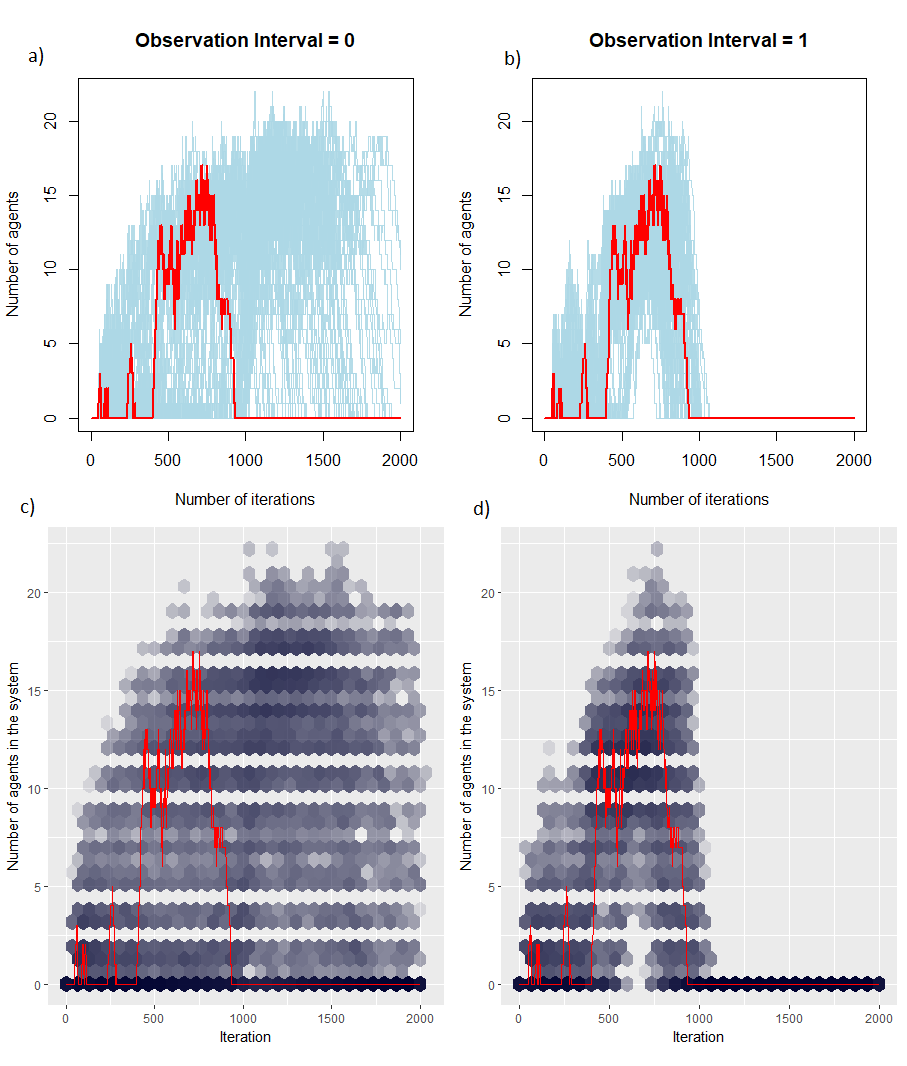
Fig. 3: Results of sampling the posterior without (a and c) and with (b and d) observations. When the ‘truth’ data for the total number of people in the simulation per iteration are used to constrain the posterior distribution, the sampling routine is much better able to estimate the input model state, so the outcomes of the samples are much closed to the ‘truth’ data as measured by the same metric as was observed. Plots a and b show each individual sample. Plots c and d show the density of samples in hex bins.

As well as assessing the constraint of the model using the same outputs that were observed in the Bayesian network, we also assessed model constraint using an alternate output to find how the model was constrained elsewhere. To achieve this, the truth data were generated for the cumulative number of people that had passed through each entrance and exit per step. This truth data was then observed and the posterior was sampled from again using Metropolis Hastings. Rather than containing the same type of output as the truth data the samples were of the number of people in a particular 10 by 10 grid space of the simulation at each step. This was assessed both by looking at the number of people in a given step in the grid square and by using a cumulative total. Again we found that when compared results generated without observing the truth data the model had been successfully constrained (Figs 4 and 5).

The interval at which the observation were made from the truth data was carried out at various intervals. However no pattern was found between constraint and observation intervals except of course when no observations were made.

****

*Fig. 4: Results of sampling the posterior without (a and c) and with (b and d) observations. When the ‘truth’ data of number of people passing through each entrance and exit per iteration are used to constrain the posterior distribution, the sampling routine is much better able to estimate the input model state, this is shown by the model being constrained for the cumulative number of people in a 10 by 10 grid space in the simulation. Plots a and b show each individual sample. Plots c and d show the density of samples in hex bins.*

****

*Fig. 5: Results of sampling the posterior without (a and c) and with (b and d) observations. When the ‘truth’ data of number of people passing through each entrance and exit per iteration are used to constrain the posterior distribution, the sampling routine is much better able to estimate the input model state, this is shown by the model being constrained for the number of people in a 10 by 10 grid space in the simulation per iteration. Plots a and b show each individual sample. Plots c and d show the density of samples in hex bins.*

1. Grazzini, Jakob, Matteo G. Richiardi, and Mike Tsionas. 2017. “Bayesian Estimation of Agent-Based Models.” *Journal of Economic Dynamics and Control* 77 (April): 26–47. <https://doi.org/10.1016/j.jedc.2017.01.014>. [↑](#footnote-ref-1)
2. keanu-post-hackathon/keanu-examples/stationSim/src/main/java/StationSim/Wrapper.java [↑](#footnote-ref-2)