Panu Somervuo, University of Helsinki, Finland, May 2023

Obsimulator 1.0

Contents

Description

obsimulator is an extension of point process simulator 'ppsimulator' (https://github.com/psomervuo/ppsimulator). It allows to simulate observation processes and behaviors of two types of observers: explorers and followers.

How to get started

After uncompressing the file ‘obsimulator.zip’, there will be three directories: ‘docs’, ‘code’, and ‘examples’. In the directory 'examples', there are eight subdirectories each containing an example demonstrating different features of the software. Each subdirectory contains a readme file containing all commands to run the example. R is needed for preparing input files and visualizing the outputs.

It is assumed that the user is familiar with the terminology and concepts of ppsimulator. If not, it is advisable to read the manual of that software package first. In order to compile obsimulator, type the following command in directory ‘code’ (note ‘$’ is terminal prompt, i.e. don’t type it):

```
$ gcc obsimulator.c -lm -O2 -o obsimulator
```

When running the simulator without arguments, it will output its expected input:

```
$ ./obsimulator
Obsimulator version 1.0
required arguments:
 -p filename process definitions
 -m filename model components
 -i filename initial configuration
 -g filename road network ('none' for no input)
 -o filename basename of output
 -U integer/float space length
 -T integer max time
 -rs record_species (delimited by ',')
 -ek explorer kernels (delimited by ',')
 -fk follower kernels (delimited by ',')

optional arguments:
 -dT integer/float interval for saving the state of configuration, negative
value means saving only final output (default -1)
 -osep filename basename for separate point coordinate files (default none, if
defined, this will set -outc 0)
 -outc 1/0 output merged point coordinate file '.points', 1: yes, 0: no (default
1)
 -outn 1/0 output point count file '.counts', 1: yes, 0: no (default 1)
 -oute 1/0 output event count file '.events', 1: yes, 0: no (default 1)
 -trL float truncation limit for Gaussian kernel in units of sd (default 3)
 -w integer/float cell width (default min kernel radius), note: w will be
rounded up so that U is its multiple
 -r integer seed value for random number generator (default 1)
 -E integer max number of events (default 2147483647)
 -s 1/0 stop if extinction of points, 1: yes, 0: no (default 1)
 -H integer number of hierarchical levels to speedup computation for large U/w
(default 0)
 -exn integer number of candidate targets for explorer (default 100)
 -obstacle filename obstacle input (default none)
 -obserror filename observation error input (default error-free observations)
 -obrecords filename output observation record book
 -obsummary filename output summary, counts of observations
 -obsout filename observation events
 -info integer (default 0)
```

When running the simulator with input points and model definition, it will write to standard output the names of all processes that have been defined in the model file and how many times they have been occurred during the simulation.

Scope and background

In the present modeling scenario, there are two kinds of observers: explorers who go to unexplored regions and followers who go after already found targets. They both have the same information available: map of already observed targets. In the simulation world there are also unobserved targets, they become observed when observer walks by and detects them. All observation journeys start and end in the city. Travel from the city to the target is done by first using road and then continuing off-road at the point where the distance between the road and target is the smallest. Off-road moves need not be directly towards the target and they can include random-walk behavior, but eventually they will lead to the target location after which the observer returns back to the city. All events (moves, detections, etc.) are simulated from stochastic processes in continuous time and space.

Before going into details of modeling explorers and followers, let's start with how the movement is modeled in the obsimulator. The key element is that traveller points can utilize a road network when movíng towards their targets. Road is represented as a graph where movements take place between neighboring nodes by means of jump process. In addition, it is also possible to move without the road towards the target (offroad move). Each traveller point has a memory which enables it to make consistent movements towards its target location within multiple simulation events. This is different in ppsimulator where points do not have any memory which means that if the point moves towards one target location in the present event, in the next event the movement can be towards another target location if there are multiple points representing the same target species. Another difficulty to model chains of directed movements in ppsimulator is due to the fact that all kernels are spherically symmetric.

It is important to note that while ppsimulator has been written to go hand-in-hand with the analytical tool supporting the fairly general catalyst-reactant-product models without memory, obsimulator has six processes which are outside of this framework. These "hard-wired" processes which utilize and modify the memory of the traveller point are: PlanToMoveFromStartOfRoad, PlanToMove, MoveOnRoad, MoveOffRoad, ExplorerSelectTarget, and FollowerSelectTarget. Other processes, e.g. ChangeInType, only copy the memory/plan of the traveller, but they do not modify the travel plan. In addition to these six processes, there are two processes whose definitions should also not be changed. They are MakeObservation1 and MakeObservation2. Some features like modeling observation error assume their presence. The implementations of these eight special functions are fixed in the C code and their definitions should not be edited in the process description input file. Other processes can be freely created and modifield in the same way as in ppsimulator where the only requirement for the "freely defined" process is that it must have a central point to which other members of the process are connected and the same species cannot be present multiple times as an input in the process defnition (with the expection that the species of the central point is allowed to be twice in the input).

A road network is a graph, i.e., a collection of nodes and their connections. Each road node has both species attribute and location and it can be involved as a catalyst in any user-defined process. However, in the present version of the simulator, it should not be used as a product or reactant of any process. In the present implementation, road network must be pre-defined and it is given as a separate input file for the simulator. Although it would be possible to define a road network generating processes, it is not supported in the present version. Another speciality regarding the road nodes is that when outputting snapshots of the simulation, road nodes are not included in the output point coordinate file. This avoids reading twice the same road node if the simulation is initialized based on the output of the earlier simulation. In all cases the road network must be given as an input in a separate file. As a small benefit, this also saves some disk space since road information is stored only once.
The format of the road network file is the following: after an optional header line, there is one line for each graph node:

```
nodeindex species x-coordinate y-coordinate list-of-neighbor-nodes
```

First line of the file is considered as a header line if it does not start with number and in that case the content of the first line is ignored. This means that user can include e.g. column names in the first line although the software does not utilize them. For the remaining part of the file, each node has a running index, the start index can be arbitrarily chosen (e.g. 0 or 1) but after that the nodes must be listed in an ordered manner so that the index of the next node is always the increment by one of the previous node. All columns must be separated by a single space or tabular character. The list of neighbor nodes consist of node indices where each element is separated by a single space, tabular character, or comma. The number of columns need not be fixed (since different nodes may have different number of connections), but if the user prefers to have a fixed number of columns in each line, the empty neighbor indices must be NA (without quotes) in which case they are ignored. End of each line is detected by newline character "\n", but carriage return character "\r" is allowed to be present in the file as well, so the software should be able to read the road network text file regardless whether it has been prepared using Linux, Mac, or Windows.

All road nodes need not be of the same species type. Different parts of the road may have different characteristics which are represented by the corresponding species type. There are no restrictions to the topology of the road network either, e.g. roads may have loops and junctions, however, in the present version of the simulator, in order to reach all the nodes, the network graph must be singly connected. This is because in the present implementation all road moves start from a root node (node with the smallest index). It would be straightforward to make this more general (by changing a few lines in a C code), but the present version suffices for the case where a point representing a city is located next to the road node with the smallest index and all movements along the road are for the people living in that city and their onroad travels start from this particular node. The road user points start their journey from the city and then come back there after visiting their targets. The benefit of this is that it suffices to run Dijkstra's algorithm only once in the beginning of each simulation and after that, giving any node of the network, the shortest path from the root to that node is readily available without any further calculations.

Traveller point has memory. It is used to store the target of the movement. The selection of the target is based on distance kernel between the location of traveller s0 and target location t1. During the travel initialization process, the traveller species changes into s1. Movement process can be defined for species s1 and in all movements, the target remains in the memory of s1 until the traveller reaches the target t1 in which case the traveller species changes again. There are two travel initialization processes: PlanToMove which does not utilize road and PlanToMoveFromStartOfRoad which utilizes both road and offroad. In the latter function it is possible to give separate traveller species changes depending on whether the traveller uses road or offroad.
In the definitions of these two processes (see Appendix1), some information of multiple change-in-type species are a bit misleadingly stored as catalysts although they act as delayed products. Although this may obscure the clarity of the process definition, it has been done simply due to technical reasons (this way the information was able to get easily stored without major changes in C-code). Nevertheless, it is important to note that these processes are specially hard-wired in the present C-code and they are not general to be used e.g. with analytical tools or ppsimulator software.

1. ‘PlanToMove’ is used for initializing offroad movements without road network.
2. ‘PlanToMoveFromStartOfRoad’ is used for initializing movements utilizing road network.
3. ‘MoveOnRoad’ specifies the rate of jumps to the next road node.
4. ‘MoveOffRoad’ specifies the rate and distribution of jumps towards the target outside the road.
5. ‘ExplorerSelectTarget’ is used for initializing explorer observations
6. ‘FollowerSelectTarget’ is used for initializing follower observations
7. ‘MakeObservation1’ is used for detecting a target by observer
8. ‘MakeObservation2’ is used for detecting a target species by observer

Example 1. Offroad movement

This demonstrates directed movement towards the target. Let's specify five types of species:

species	description
1	start species
2	target species
3	traveller at origin
4	traveller after movement initialization
5	traveller arriving at target

For simplicity, initial configuration file 'xin' contains only three points:

```
1 5.0 5.0
2 10.0 5.0
3 5.0 5.0
```

There is one traveller point (species 3) at location (x=5.0, y=5.0). Species 1 is only used for marking the start location of the traveller and there is one target species (2) at location (x=10.0, y=5.0). Let's define two different models:

File ‘modelA’:

```
PlanToMove[3,4,2, tophat[400,global],5]
MoveOffRoad[4, tophat[1,0.5]]
```

File ‘modelB’:

```
PlanToMove[3,4,2, tophat[400,global],5]
MoveOffRoad[4, tophat[1,0.5]]
Jump[4, tophat[1,0.5]]
```

PlanToMove[3,4,2, tophat[400,global], 5] means that once the travel plan is made for a specific point representing species 3, it changes its type into species 4. The target of movement is randomized according to the tophat-kernel between the location of species 3 and the species 2. Once the target is chosen, it remains specific for the particular traveller point until it reaches its target. Here 'global' is used for radius parameter which means that all points of type 2 regardless their distance from the species 3 are equally likely to be selected as targets. The last argument '5' is the species type into which the traveller point 4 changes when it has reached the target.
The movement takes place according to component MoveOffRoad[4, tophat[1,0.5]]. Here the rate of jump is 1 and the length of jump is between 0 and 0.5 from uniform distribution. All jumps are directed towards the location of the specific target selected for the traveller point during the execution of process component PlanToMove. However, different travellers may have different targets. In modelB, there is an additional movement component Jump[4, tophat[1,0.5]] which results that there are also jumps to random directions with the same rate and length distribution as in the target-directed jumps.

Simulator is run using the command (here three lines, each starting with prompt $):

```
$ P=~/Work/obsimulator/code
$ EXTRA="-rs 0 -ek NONE,1,NONE,1,NONE,1,NONE,1 -fk NONE,1,NONE,1,NONE,1,NONE,1"
$ $P/obsimulator -p $P/obsprocesses.txt -m modelA -g none -i xin -o outA -U 20 -
T 30 -dT .1 -info 1 $EXTRA
```

The first one sets the variable P to point to the location of the code. In the present example, obsimulator.zip was uncompressed in directory ~/Work.
The second line is needed for the information regarding observers and followers. Although in this example we are not interested in observers and followers, we need to spesify the arguments (-rs, -ek, -fk). The main command to run the simulator is in the third line. Argument "-g none" means that no road network file is used. The results of ten replicates are shown in Fig 1.

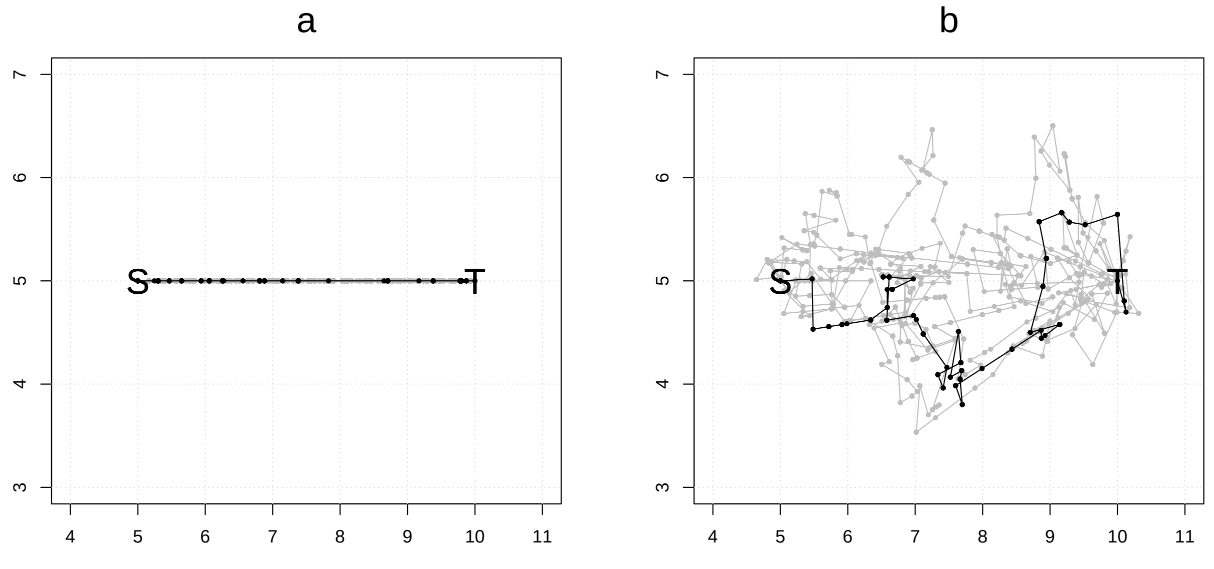


Fig 1. Offroad movements between source S and target T. Lines represent jumps and dots are the landing points between the jumps. One of the trajectory

realizations is shown in black. With modelA all jumps are towards the target and with modelB the movement process consists of two components: jumps directed towards the target and jumps to random directions.

Example 2. Movement along a road

Our example road network is in file 'roadfile'. It contains 17 road nodes which represent three different types of species. The description of 14 different species in the simulation are:

species	description
1	fast road
2	medium-speed road
3	slow road
4	city
5	city person
6	fast onroad traveller
7	medium-paced onroad traveller
8	slow onroad traveller
9	offroad traveller towards target
10	offroad traveller back from target
11	observer
12	unobserved plant
13	observed plant
14	travel target

The initial configuration 'xin' consists of species types 4,5,12, and 14. It is defined using R:

```
x4=matrix(c(1,2),nr=1,nc=2)
x5=matrix(c(1,2),nr=4,nc=2,byrow=T)
x12=matrix(runif(2*5000,min=0,max=10),nc=2)
# road end nodes
x14=matrix(c(8.0,3.0, 7.5,7.5, 1.5,8.0),nc=2,byrow=T)
x=rbind(cbind(4,x4),cbind(5,x5),cbind(12,x12),cbind(14,x14))
write.table(x,"xin",col.names=F,row.names=F)
```

Road file 'roadfile' speficies the locations and connections of road nodes and assignes the nodes into species types 1,2,3.

```
1 1 1.0 2.0 2
2 1 2.0 2.5 1,3
3 1 3.0 2.8 2,4
4 1 4.0 3.5 3,5,9
5 1 5.0 3.2 4,6
6 1 6.0 2.5 5,7
7 1 7.0 2.5 6,8
8 1 8.0 3.0 7
9 2 4.5 4.5 4,10
10 2 5.0 5.5 9,11,14
11 2 6.0 6.0 10,12
12 2 7.0 6.5 11,13
13 2 7.5 7.5 12
14 3 3.9 5.8 10,15
15 3 3.5 7.0 14,16
16 3 2.5 7.5 15,17
17 3 1.5 8.0 16
```

The file ‘model1’ is

```
PlanToMoveFromStartOfRoad[5,6,14, truncatedGaussian[3000,4], 9,11,10,6,5]
MoveOnRoad[6,1.0]
```

```
MoveOnRoad[7,1.0]
MoveOnRoad[8,1.0]
ChangeInTypeByFacilitation[6, 7, 1, tophat[1,.01]]
ChangeInTypeByFacilitation[6, 8, 1, tophat[1,.01]]
ChangeInTypeByFacilitation[7, 6, 2, tophat[1,.01]]
ChangeInTypeByFacilitation[7, 8, 2, tophat[1,.01]]
ChangeInTypeByFacilitation[8, 6, 3, tophat[1,.01]]
ChangeInTypeByFacilitation[8, 7, 3, tophat[1,.01]]
MoveOffRoad[9, tophat[1,1]]
MoveOffRoad[10,tophat[1,1]]
ChangeInType[10,11,10]
ChangeInTypeByFacilitation[13, 12, 11, tophat[1,1]]
ChangeInTypeByFacilitation[13, 12, 6, tophat[1,1]]
ChangeInTypeByFacilitation[13, 12, 7, tophat[1,1]]
ChangeInTypeByFacilitation[13, 12, 8, tophat[1,1]]
```

The first model component PlantToMoveFromStartOfRoad specifies that when initializing the travel, species 5 changes into species 6. The target location of the travel is selected according to truncatedGaussian between the location of species 5 and target species 14. The five last arguments are for change-in-types of the traveller points. After species 5 has changed into species 6, it is ready to start the travel using the road network. The movement jumps take place when the event MoveOnRoad occurs. Traveller 6 continues using the road until it arrives the the closest road node to the target location (location of the selected point of type 14). Then the traveller changes from species type 6 to 9 and continues the travel using offroad moves until it arrives to the target. Once in target, the traveller changes from type 9 to 11. It continues to be species 11 until some outside event turns it into species 10. From that on, the traveller goes back to same road node it left earlier using offroad moves. Once it reaches the road, it turnes into speices 6 and then continues to move back to the original road start node staying on the road. After it has reached the start of the road node, the traveller changes to species 5. MoveOnRoad process is defined for all three traveller types 6,7, and 8. In this model they all have the same jump rate 1.0. The following six ChangeInTypeByFacilitation processes change the type of traveller based on the type of road node. E.g. ChangeInTypeByFacilitation[6, 7, 1, tophat[1,.01]] means that if species 7 (middle-paced traveller) is on road node of type 1 (fast road), the traveller changes the type from 7 to six (fast traveller). The connectivity kernel is narrow since once the traveller is on road its location will be the same as the location of road node. However, due to numerical reasons the radius should not be defined to be too small. MoveOffRoad is defined for two types of offroad travellers 9 and 10. ChangeInType[10,11,10] defines that traveller 11 becomes 10 with rate 10. The last four ChangeInTypeByFacilitation processes model how the observations are made, unobserved plant (12) becomes observed (13) and the observation can be made all four traveller species 6,7,8, and 11.

File ‘model2’ is otherwise the same as ‘model1’, but the jump rates are different for travellers 6,7,and 8:

```
MoveOnRoad[6,100.0]
MoveOnRoad[7,30.0]
MoveOnRoad[8,1.0]
```

Simulation is run with the command

```
$ P=~/Work/obsimulator/code
$ $P/obsimulator -p $P/obsprocesses.txt -m model -g roadfile -i xin -o out -U 20
-w 1 -T 50 -osep foo $EXTRA
```

Here we specify U=20 although all point coordinates are x=0..10, y=0..10. This is because the simulation space has torus topology where the "corner" points (0,0) and (10,10) are the closest points but here we want that the corner points are the most distanct ontes.

Notice that it is important to define the grid size by -w because the default value is the length of the smallest kernel and in this model the smallest kernel is narrow (tophat[10,.001]). Actually we would like to have dirac's delta type of connectivity kernel and tophat kernel with small radius is just a crude approximation for that. Since the number of the grids to be allocated for simulation is (U/w)^2, this number would become overwhelmingy large with small w. The consequence would be that the simulation would require lots of memory and also become very slow. If simulation seems to stuck right from the beginning, it is good to check whether the command line option -w has been used or not.

The simulation results using model1 and model2 are shown in Fig 2. With equal movement rates using model1, the neighborhood of all road nodes are equally covered by observed plants whereas using model2, the number of observed plants is largest on the road where the jump rates are lowest. Since rate parameter for ChangInType process from species 11 back to onroad traveller is the same at all three ends of the road, there is similar concentration of observed plants at the end of the roads.

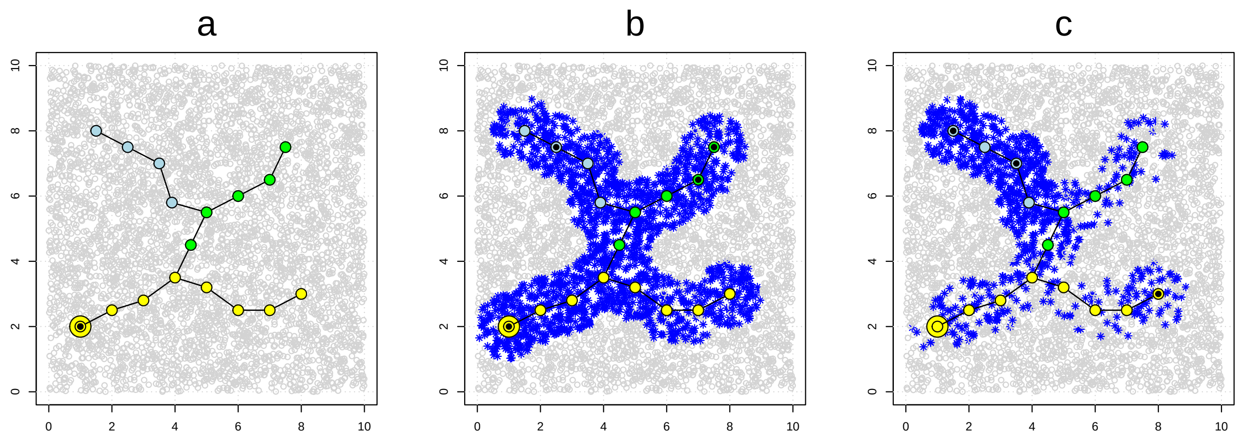


Fig 2. Road movements and observed plants. Initial configuration (a), road movements with equal jump rates in model1 (b), road movements with different jump rates in model2 (c). Unobserved plants are in gray and observed plants are in blue. Large circle at the bottom left is the city and road start is in the middle of it. Species type 1 in yellow (fast road at bottom), type 2 in green (middle-paced road going towards top-right corner) and type 3 in light blue (slow road going towards top-left corner). There are more observations along that road where traveling is slow since that allows more time for making observations.

Example 3. Observers using both road and offroad movements

This is similar to the previous example but now there are two types of targets (e.g. plants) with different probabilities to be observed. Target is selected according to truncatedGaussian between the city and the location of the target.

species	description
1	fast road
2	medium-speed road
3	slow road
4	city
5	city person
6	fast onroad traveller
7	medium-paced onroad traveller
8	slow onroad traveller
9	offroad traveller towards target
10	offroad traveller back from target
11	observer

12   unobserved easy-to-find target
13   observed easy-to-find target
14   unobserved difficult-to-find target
15   observed difficult-to-find target

File ‘model’:

```
PlanToMoveFromStartOfRoad[5,6,12, truncatedGaussian[3000,4], 9,11,10,6,5]
PlanToMoveFromStartOfRoad[5,6,14, truncatedGaussian[3000,4], 9,11,10,6,5]
MoveOnRoad[6,3.0]
MoveOnRoad[7,2.0]
MoveOnRoad[8,1.0]
ChangeInTypeByFacilitation[6, 7, 1, tophat[1,.01]]
ChangeInTypeByFacilitation[6, 8, 1, tophat[1,.01]]
ChangeInTypeByFacilitation[7, 6, 2, tophat[1,.01]]
ChangeInTypeByFacilitation[7, 8, 2, tophat[1,.01]]
ChangeInTypeByFacilitation[8, 6, 3, tophat[1,.01]]
ChangeInTypeByFacilitation[8, 7, 3, tophat[1,.01]]
MoveOffRoad[9, tophat[1,.2]]
MoveOffRoad[10,tophat[1,.2]]
ChangeInTypeByFacilitation[13, 12, 11, tophat[0.03,0.01]]
ChangeInTypeByFacilitation[15, 14, 11, tophat[0.0003,0.01]]
ChangeInType[10,11,1]
```

Initial configuration is given by

```
x4=matrix(c(1,2),nc=2)
x5=matrix(c(1,2),nr=10,nc=2,byrow=T)
x12=matrix(runif(2*2500,min=0,max=10),nc=2)
x14=matrix(runif(2*2500,min=0,max=10),nc=2)
x=rbind(cbind(4,x4),cbind(5,x5),cbind(12,x12),cbind(14,x14))
write.table(x,"xin",col.names=F,row.names=F)
```

This will produce 10 observers. Their initial location is in the city (x=1.0, y=2.0) which is also the location of the first node of the road network.

Simulation is run with the command:

```
$ P=~/Work/obsimulator/code
$ $P/obsimulator -p $P/obsprocesses.txt -m model -g roadfile -i xin -o out2 -U
20 -w 1 -T 5000 -osep foo2 $EXTRA

gillespie!!! domain size U=20.000000 (dimension 2), cell_width=1.000000
Number of events per process:
1)    PlanToMoveFromStartOfRoad[5,6,12,truncatedGaussian[3000,5],9,11,10,6,5]
      821
2)    PlanToMoveFromStartOfRoad[5,6,14,truncatedGaussian[3000,5],9,11,10,6,5]
      923
3)    MoveOnRoad[6,100.0]     11270
4)    MoveOnRoad[7,30.0]      3385
5)    MoveOnRoad[8,1.0] 1816
6)    ChangeInTypeByFacilitation[6,7,1,tophat[1,.01]] 712
7)    ChangeInTypeByFacilitation[6,8,1,tophat[1,.01]] 0
8)    ChangeInTypeByFacilitation[7,6,2,tophat[1,.01]] 967
9)    ChangeInTypeByFacilitation[7,8,2,tophat[1,.01]] 410
10)   ChangeInTypeByFacilitation[8,6,3,tophat[1,.01]] 410
11)   ChangeInTypeByFacilitation[8,7,3,tophat[1,.01]] 342
12)   MoveOffRoad[9,tophat[1,.2]]   23158
13)   MoveOffRoad[10,tophat[1,.2]]  23013
14)   ChangeInTypeByFacilitation[13,12,11,tophat[0.0006,0.01]]   545
15)   ChangeInTypeByFacilitation[15,14,11,tophat[0.00006,0.01]]  140
16)   ChangeInType[10,11,1]   1736
```

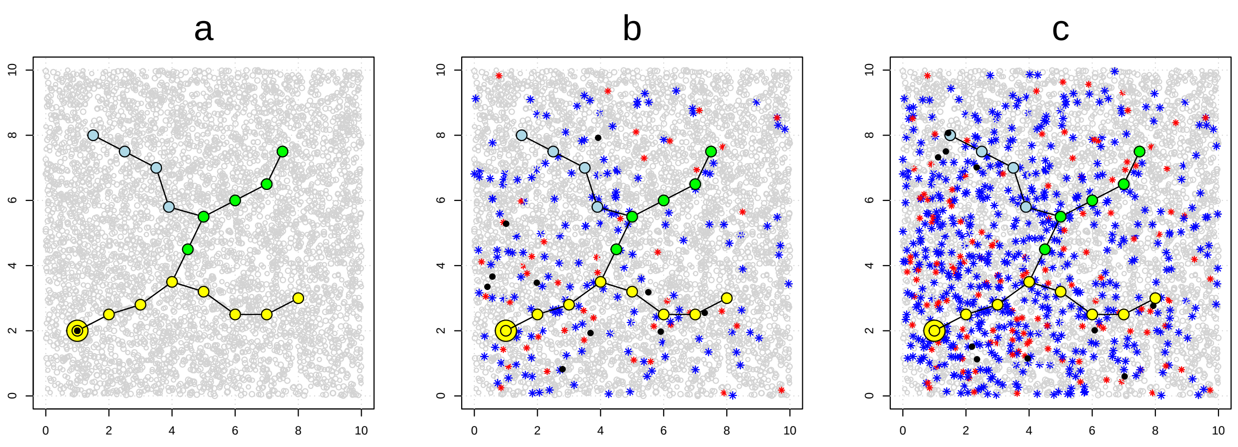


Fig 3. Unobserved targets are shown in gray and observed targets are in blue (easy-to-find) and red (difficult-to-find) at different time points T=0 (a), T=1500 (b), and T=5000 (c). Observers are shown as black filled dots.

Making observations with explorers and followers

After becoming familiar with simulating movements both along the road and off the road, we can now go to the more details of simulating observer behavior. In order to simulate explorers and followers, we need the following items:

1) Keeping up the target records

Each target contains time information: the first time it has been found and the most recent time it has been detected. Also the number of detections of each target can be recorded. The same record information is available in the city for both explorers and followers.
When observer finds a target (either previously observed or unobserved), the observation time of the target is updated. This can be done either when observer comes back to city where his observation journey started or immediately (we can imagine that everyone uses cell phones so information goes without any delays). Although time stamp of the record is an important piece of information, maybe the delay between the detection of target and putting it into the records in the city is not that important. Slightly easier implementation is to update the time information at the same moment when the target is detected. In the previous example, the process ChangeInTypeByFacilitation was used to make observations, but in order to model observation errors and keeping the records, two processes MakeObservation1 and MakeObservation2 should be used.

2) Process for selecting a target for explorer:

Target z for explorer is a plain coordinate in the simulation world, not the location of any object like observed or unobserved target.
Since explorer should go to those regions where there are not yet any observed targets and none of the unobserved targets are by definition not yet known, there are no existing objects to be used as target points. Instead of exactly finding the target location with largest minimum distance to any observed targets (which could be done) or calculating a score for "unobservedness" using a grid (which also could be done but it would introduce a problem of how fine grid should be used), the present implementation is that the selection is done based on N candidate points (e.g. N=100) which are sampled from entire simulation space (x,y)~(U(0,L),U(0,L)). A value/score is calculated to each of the candidate points that combines: 1) distance to city, 2) distance to road, 3) distance to observed plants. E.g. good target would minimize 1 and 2 and maximize 3. This can be implemented by sampling a target location z among N candidates based on p1(c,z)p2(r,z)p3(z,o), where

- p1(c,z) is kernel for distance between city and target location (could be also between city and nearest road point to target)
- p2(r,z) is kernel for distance between target location and road
- p3(z,o) is the kernel for distance between candidate location

p1 and p2 can be e.g. Gaussian or tophat, but p3 must be something which increases as a function of distance (e.g. k(z,z)-k(z,o) where k is a Gaussian). Explorer then goes to the selected target z and makes observations along the way (observation period can be restricted to take place only during the travel from road to target location). It might happen that there are no targets (not even unobserved ones) in the neighborhood of z or there are some but explorer is not able to find them (detection process is stochastic) in which case the explorer comes back to city empty-handed. Otherwise time information of all targets that have been detected are updated.

3) Process for selecting a target for follower:

Target t for follower is one of the already observed targets. It can be selected/sampled based on the product of connectivity kernels p1(c,t)p2(r,t)p3(t), where
- p1(c,t) is kernel for distance between city and target (could be also between city and nearest road point to target)
- p2(r,t) is kernel for distance between target and road
- p3(t) is the kernel for distance in time (e.g. difference between current time and the time event when target was observed last time)

Follower will travel to the coordinate location of t and since the observation process is stochastic, he/she either detects the target or not. In addition, follower may find additional targets during the travel (observation period can be restricted to occur only between the road and target).

Point types may have multiple attributes. Traveller point has 4 levels of attributes:

1: original point type
2: direction of travel: F (forward) or B (backward)
3: target road node location: node index or its xy coordinate
4: (optional) target out-of-road location: xy coordinate

Moving from city to target location X and returning back to city consists of the following steps:

1) calculate nearest road node R from target location X
2) traveller point P changes type:
   - 2nd attribute F (forward traveller)
   - 3rd attribute R
   - 4th attribute X
3) move from city to R along road
4) move from R to target location X
5) traveller observes target
6) move from X to R
7) traveller point P changes type:
   - 2nd attribute B (backward traveller)
8) move from R to city along road
9) report observations

Moving along road from start node Rs to target node Rt:

Road is a directed network of nodes where traveller moves from one node to the next with Jump process.
Traveller moves either forward or backward based on its attribute.
When coming to junction node, traveller chooses the shortest path based on target coordinate.

Moving without road from start location Xs to target location Xt:

Offroad moves are jumps directly towards the location of the targets. Jump lengths are defined by the movement kernel.

Example 4. Planning of a trip for explorer and follower

Observer starts the observation trip from city and returns to city. During the trip, the observation species may change to six different species types and for each species, a different action/actions can be specified with other process/model components. E.g. for certain observer species type, only movements may be allowed, and for certain observer species type, only target detections may be allowed (but it is equally possible to define several processes for one observer species). If the interest is just to use any observer to make detections, a model component 'PlanToMoveFromStartOfRoad' can be used. However, if the interest is to make the difference between explorers and followers, there are two specific process definitions for this, 'ExplorerSelectTarget' and 'FollowerSelectTarget'. In c-code they are implemented in functions explorer_select_target and follower_select_target.

In the process definition, the two components 'ExplorerSelectTarget' and 'FollowerSelectTarget' include the definition of observer species in the city and then 6 other species specified for change in type of observer in different parts of the observation trip. In addition, one target species must be defined and one rate parameter. All these parameters are defined in a model file. Additional parameters are defined from command line using argument -ek for explorer and -fk for follower. These command line parameters are strings of 8 entries separated by ',' e.g. -ek k1,p1,k1,p2,k3,p3,k4,p4. This string defines 4 kernels and their parameters:

k1: kernel for distance between city and target
k2: kernel for distance between road and target
k3: kernel for distance between already observed location and new target (this only affects explorer)
k4: kernel for time difference between current time and last time of observation (this only affects follower)

Each kernel type must be one of the following: NONE, LINEAR, TOPHAT, GAUSSIAN, or EXPONENTIAL.
Parameter is the length scale, i.e. in tophat the radius.

Let's have following species in simulation: (1) city, (2) road, (3) unobserved plant, (4) observed plant, (5-8) explorer, and (9-12) explorer.
Note that although 7 observer species types must defined in process 'ExplorerSelectTarget' (the same for 'FollowerSelectTarget'), they don't have to be different species, e.g., we may specify that the observer type in forward trip from city to target is the same type as in the backward trip from target to city. For details, here are the three processes related to planning of the trip for explorer and follower:

```
ExplorerSelectTarget[s0, s1, t1, r, s2, s3, s4, s5, s6]
FollowerSelectTarget[s0, s1, t1, r, s2, s3, s4, s5, s6]
```

These processes define that species s0 becomes traveller s1 with rate r and plans 2-way trip from the beginning of the road to target t1 using road to nearest location of t1 and then continuing offroad, s1-s6 are point type changes for traveller during the journey: s1 forward on road, s2 forward offroad, s3 at target, s4 back offroad, s5 back onroad, s6 back in start point. The differences between these two processes are how the targets are selected. For explorer, only unobserved targets are selected and for follower, only observed targets are selected. Since there is no information for unobserved targets, explorer may select a target location which does not contain any species within its detection radius. Target species t1 for explorer is the species to mark the selected target location, it should not be any other species involved in the model. Command line arguments -ek and -fk give parameters for target selection.

In this model, we have 4 different types for explorer and 4 different types for follower: i) in-the-city-species, ii) on-the-road-species, iii) on-the-offroad-species (when walking from road to target), iv) at-target-species (when doing detections). We define the movements, detections, and other actions for these species in the model file.

Initial configuration 'xin1' will consist of city (1), road (2), and unbserved plants (3). There will be one explorer (5) and follower (9) initially located in the city as well. Here the first node of the road is located in the city.

File 'xin1' (species x-coord y-coord):

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

File 'road1' (node species x-coord y-coord neighbor-list):

```
1 2 2 2 2
2 2 3 2 1,3
3 2 4 2 2,4
4 2 5 3 3,5
5 2 6 4 4,6
6 2 6 5 5,7
7 2 6 6 6,8
8 2 5 7 7
```

File 'model1':

```
ExplorerSelectTarget[5, 6, 13, 1.0, 7, 8, 7, 6, 5]
MoveOnRoad[6, 1.0]
MoveOffRoad[7, tophat[0.6,1.0]]
MakeObservation1[4, 3, 8, tophat[3,1.0]]
ChangeInType[7, 8, 0.1]
FollowerSelectTarget[9, 10, 4, 1.0, 11, 12, 11, 10, 9]
MoveOnRoad[10, 0.8]
MoveOffRoad[11, tophat[0.6,1.0]]
MakeObservation1[4, 3, 12, tophat[3,1.0]]
ChangeInType[11, 12, 0.2]
```

Here we have used species 13 to denote target species for explorer. It is used internally in the software to mark the target location of explorer but it will not be visible when outputting point configurations or record files (but if needed, it can be easily included in those). In the model1, we defined that only observer at target site (explorer 8 and follower 12) are making observations, i.e. are able to change unobserved plant (3) into observed one (4). Here we define that explorer picks targets everywhere and follower picks targets within distance<2 from road. Notice that in the model we have to specify rate with observer at site turns into observer at road in order to start the journey back from target site to city. The ratio between the rates to go back and change the unobserved plant into observed one determines the detection rate.

```
$ P=~/Work/obsimulator/code
$ $P/obsimulator -p $P/obsprocesses.txt -m model1 -i xin1 -g road1 -o out -U 10
-T 1000 -rs 4 -ek NONE,1,NONE,1,NONE,1,NONE,1 -fk NONE,1,TOPHAT,2,NONE,1,NONE,1
-osep xout

gillespie!!! domain size U=10.000000 (dimension 2), cell_width=1.000000
Number of events per process:
1)    ExplorerSelectTarget[5,6,13,1.0,7,8,7,6,5]      34
2)    MoveOnRoad[6,1.0] 251
3)    MoveOffRoad[7,tophat[0.6,1.0]]      266
4)    MakeObservation1[4,3,8,tophat[3,1.0]]     6
5)    ChangeInType[7,8,0.1]   34
6)    FollowerSelectTarget[9,10,4,1.0,11,12,11,10,9]  82
7)    MoveOnRoad[10,0.8]      309
8)    MoveOffRoad[11,tophat[0.6,1.0]]     234
9)    MakeObservation1[4,3,12,tophat[3,1.0]]    0
10)   ChangeInType[11,12,0.2] 33
```

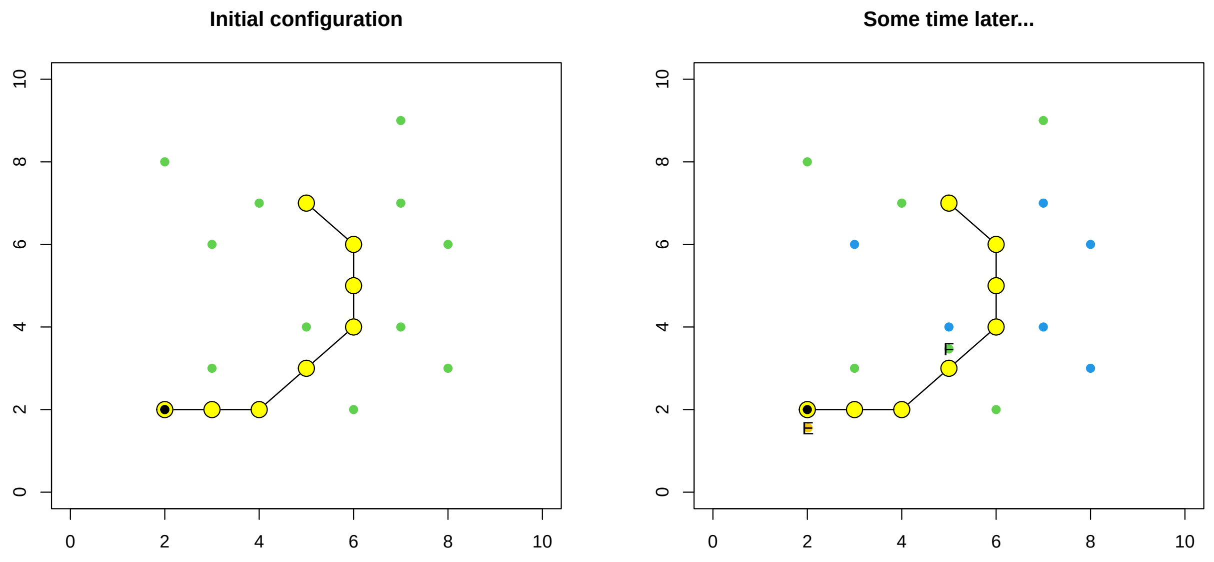


Fig 4. Unobserved plants (green dots) and observed plants (blue dots). E denotes explorer and F denotes follower.

Example 5. Obstacles

Obstacles are points that affect the acceptance probability of a process. They are defined in a file where four columns are: obstacle_target, obstacle_source, acceptance_probability, and radius. Here is an example.

File 'obstaclefile':

```
2 3 0.0 1
2 4 0.2 1
```

The effect of this is:
-line1: if species 2 appears within radius 1 of species 3, it will be accepted with probablity 0.0 (i.e. always rejected)
-line2: if species 2 appears within radius 1 of species 4, it will be accepted with probablity 0.2

When running simulator, obstacle file is specified by argument -obstacle.
Let's have small example to demonstrate the effect of obstacles. For simplicity, there are no road/movements here.

File 'xin2' (species x-coord y-coord):

```
1 5 5
3 6 5
4 4 5
4 5 4
3 5 6
```

File 'model2':

```
BirthToAnotherType[2, 1, tophat[1,2]]
```

Let’s first show how species 1 produces offspring species 2 around it:

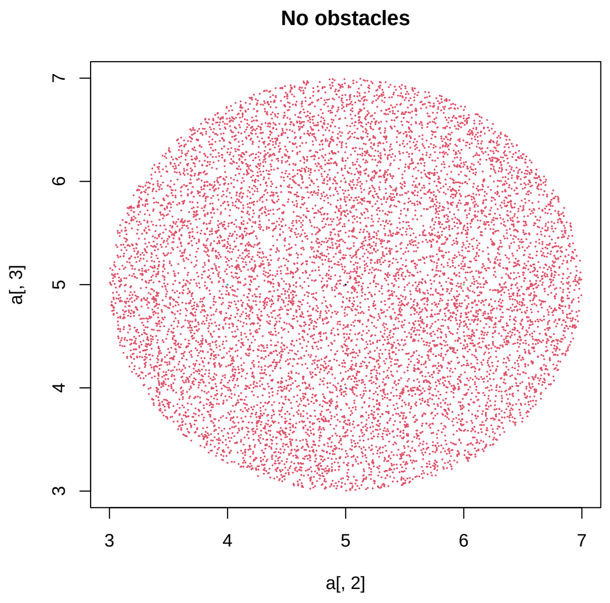
```
$ P=~/Work/obsimulator/code
$ $P/obsimulator -p $P/obsprocesses.txt -m model2 -i xin2 -g none -o out2 -U 10
-T 10000 -rs 1,2 -ek NONE,1,NONE,1,NONE,1,NONE,1 -fk NONE,1,NONE,1,NONE,1,NONE,1
-osep xout2A

gillespie!!! domain size U=10.000000 (dimension 2), cell_width=10.000000
Number of events per process:
1)    BirthToAnotherType[2,1,tophat[1,2]] 10010
```

Let’s now add obstacles. Species 2 will appear based on conditional probability defined by obstacle species 3 and 4 if species 2 appears within defined radius from them:

```
$P/obsimulator -p $P/obsprocesses.txt -m model2 -i xin2 -g none -o out2 -U 10 -T
10000 -rs 1,2 -ek NONE,1,NONE,1,NONE,1,NONE,1 -fk NONE,1,NONE,1,NONE,1,NONE,1 -
osep xout2B -obstacle obstaclefile

gillespie!!! domain size U=10.000000 (dimension 2), cell_width=10.000000
Number of events per process:
1)    BirthToAnotherType[2,1,tophat[1,2]] 9969
```



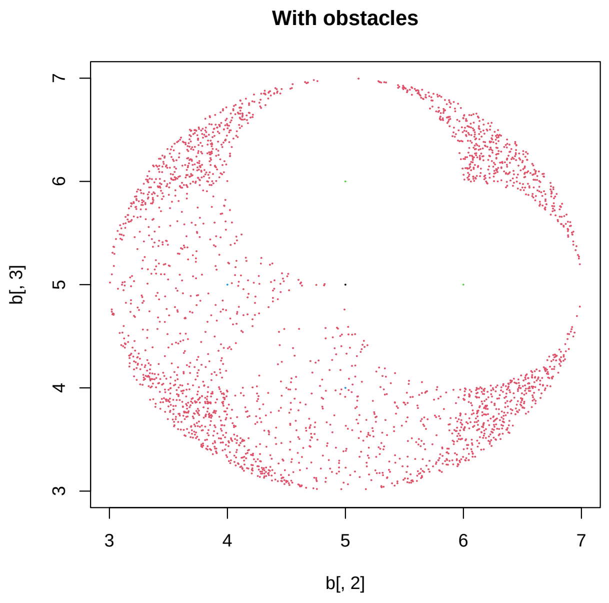


Fig 5. An illustrative example showing how obstacles work. Here a point generator seed (species 1) is located in the middle (x=5,y=5) and it generates red points (species 2) around it with a radius of 2. There are 4 points (species 3 and 4) that create obstacles within a specified radius (1) around them. They affect the acceptance probability of producing red points (species 2).

Example 6. Observation error

Simulator allows to mimick cases where observer identifies the species of a target incorrectly. User needs to define the list of species against which observer might confuse the species identification. File 'obsprobfile' gives an example of this. This file is given to simulator by using an argument -obserror. In the file it is specified for each observer species, how they confuse the true observed species into some other species. If file has line 'A B C p', after observer_species A has detected true_species B, it will report having seen species C with probability p. Note that the sum of probabilities p for pair A,B must sum to one. But the number of lines (different reported species C) may vary different pairs A,B. And if there are no lines in the file for pair A,B, then the observations of species B by observer A are error-free.

Let's have a simple model, again species 1 will produce offspring 2 and then observer 3 detects them. Species 2 will be unbserved and species 4 will be observed species so 2 changes into 4 if observer 3 detects them.
Note: although observer 3 may report having seen species 4,5,or 6 (after seen 2), unobserved species 2 will change into species 4 in all of these cases, otherwise things would become quite messy since some other observer might detect the same target point so it must have the species attribute corresponding to true species. The observation error affects only the record keeping, i.e. which species have been reported. When planning the observation trip, it is possible to do so that the target is selected based on reported species, so in this way the erroneous reports could affect the follower behavior. Not everything has been implemented, but this is possible to do. Instead of trying to guess what is needed, is better to have your feedback so that I will implement those features in the simulator which are relevant.

All records can be outputted to file with argument -obrecords outfile1, and the summary is outputted to file with argument -obsummary outfile2. The summary is basically just the confusion matrix of observed species.

file 'xin3' (species x-coord y-coord):

```
1 5 5
3 5.1 5.1
```

file 'model3':

```
BirthToAnotherType[2, 1, tophat[1,2]]
MakeObservation1[4, 2, 3, tophat[1,2]]
```

file 'obsprobfile' (four columnns are: observer_species true_observed_species: reported_observed_species probability):

```
3 4 4 0.7
3 4 5 0.2
3 4 6 0.1
```

Now have to list species 4,5,6 as record species (-rs) in order to include them in output (summary and/or record) file.

```
$ P=~/Work/obsimulator/code
$ $P/obsimulator -p $P/obsprocesses.txt -m model3 -i xin3 -g none -o out3 -U 10
-T 1000 -rs 4,5,6 -ek NONE,1,NONE,1,NONE,1,NONE,1 -fk
```

```
NONE,1,NONE,1,NONE,1,NONE,1 -osep yout -obserror obsprobfile -obsummary
yobsummary

gillespie!!! domain size U=10.000000 (dimension 2), cell_width=2.000000
Number of events per process:
1)    BirthToAnotherType[2,1,tophat[1,2]] 1011
2)    MakeObservation1[4,2,3,tophat[1,2]] 963
```

The first row of 'yobsummary' contains the counts of species 4 and how they have been reported as different species. Other lines are for 5 and 6 (but now containing zeros, since they have not been detected as true species). The proportions in the first line follow the probabilities defined in file ‘obsprobfile’.

```
      reportedS4  reportedS5  reportedS6
trueS4      684   193   86
trueS5      0     0     0
trueS6      0     0     0
```

Example 7. Getting information of observations

In the following, we want to write the observation history of each target into outputfile 'obshistory':

```
$ P=~/Work/obsimulator/code
$ $P/obsimulator -p $P/obsprocesses.txt -m model4 -i xin1 -g road1 -o out -U 10
-T 1000 -rs 4,14 -ek NONE,1,NONE,1,NONE,1,NONE,1 -fk NONE,1,NONE,1,NONE,1,NONE,1
-osep xout -obserror obsprobfile4 -obrecords obshistory
```

Simulator produces the following output to the screen:

```
gillespie!!! domain size U=10.000000 (dimension 2), cell_width=1.000000
Number of events per process:
1)    ExplorerSelectTarget[5,6,13,1.0,7,8,7,6,5]     27
2)    MoveOnRoad[6,1.0] 198
3)    MoveOffRoad[7,tophat[0.6,1.0]]     326
4)    MakeObservation1[4,3,8,tophat[3,1.0]]    7
5)    ChangeInType[7,8,0.1]   27
6)    FollowerSelectTarget[9,10,14,1.0,11,12,11,10,9] 249
7)    MoveOnRoad[10,0.8]     174
8)    MoveOffRoad[11,tophat[0.6,1.0]]    214
9)    MakeObservation1[4,4,12,tophat[3,1.0]]   134
10)   ChangeInType[11,12,0.2] 25
```

File 'obshistory' contains (can be different based on your random number seed):

```
species     x     y     num_detections    first_time  last_time
      distance_to_road  reportedS4  reportedS14
4     7.000000    7.000000    1     48.643558   48.643558   1.414214    1     0
4     8.000000    3.000000    63    226.945615  805.342469  2.236068    62    1
4     7.000000    4.000000    1     227.222560  227.222560  1.000000    1     0
4     6.000000    2.000000    72    268.289945  896.080904  1.414214    71    1
4     3.000000    3.000000    1     376.338108  376.338108  1.000000    1     0
4     5.000000    4.000000    1     596.444524  596.444524  1.000000    1     0
4     2.000000    8.000000    2     827.752717  936.358102  3.162278    1     1
```

There are 7 rows since there were 7 events for explorer at site to observe the unobserved plant (MakeObservation1[4,3,8,tophat[3,1.0]]).
Then there are 134 events where follower (12) has observed the already observed plant. These correspond to 3 plants which explorer has recognized incorrectly (column reportedS14 is nonzero). However since follower did not make any

observation errors, for the corresponding 3 plants, the reportedS4 counts 62+71+1=134 correspond to the events (MakeObservation1[4,4,12,tophat[3,1.0]]) since explorer in this model did not made observations for the already observed plants (only unobserved species 3).

In general, in obshistory file, the first column is true species, and then the last columns (reportedS...) contain the counts for correct and incorrect observations.

Example 8. Getting coordinates and time stamps of observations

In the previous example, the output file contained the summary of the detections of each target. Sometimes we might be interested also to get the entire time series of detections. When giving argument -obsout filename, all observation events are written to file 'filename'. The format is one line per observation with following 6 columns:

```
true_observed_species x-coord y-coord time reported_observed_species observer
```

Let's use the same model as in Example 7 but now add -obsout obslog

```
$ P=~/Work/obsimulator/code
$ $P/obsimulator -p $P/obsprocesses.txt -m model4 -i xin1 -g road1 -o out -U 10
-T 1000 -rs 4,14 -ek NONE,1,NONE,1,NONE,1,NONE,1 -fk NONE,1,NONE,1,NONE,1,NONE,1
-osep xout -obsout obslog1
```

Simulator produces the output:

```
gillespie!!! domain size U=10.000000 (dimension 2), cell_width=1.000000
Number of events per process:
1)     ExplorerSelectTarget[5,6,13,1.0,7,8,7,6,5]      29
2)     MoveOnRoad[6,1.0] 259
3)     MoveOffRoad[7,tophat[0.6,1.0]]      299
4)     MakeObservation1[4,3,8,tophat[3,1.0]]     7
5)     ChangeInType[7,8,0.1]   29
6)     FollowerSelectTarget[9,10,14,1.0,11,12,11,10,9]1003
7)     MoveOnRoad[10,0.8]      0
8)     MoveOffRoad[11,tophat[0.6,1.0]]     0
9)     MakeObservation1[4,4,12,tophat[3,1.0]]    0
10)    ChangeInType[11,12,0.2] 0
```

file 'obslog' is:

```
4 7.000000 7.000000 48.643558 4 8
4 2.000000 8.000000 81.708136 4 8
4 8.000000 6.000000 136.421872 4 8
4 4.000000 7.000000 324.775820 4 8
4 8.000000 3.000000 493.896271 4 8
4 5.000000 4.000000 690.790600 4 8
4 7.000000 9.000000 800.278879 4 8
```

Let's now include the possibility for observation error. File ‘obsprobfile4’ contains two lines:

```
8 4 4 0.5
8 4 14 0.5
```

and the component FollowerSelectTarget[9,10,14,1.0,11,12,11,10,9] defined that follower goes to target only when species 14 has been reported

```
$ $P/obsimulator -p $P/obsprocesses.txt -m model4 -i xin1 -g road1 -o out -U 10
-T 1000 -rs 4,14 -ek NONE,1,NONE,1,NONE,1,NONE,1 -fk NONE,1,NONE,1,NONE,1,NONE,1
-obserror obsprobfile4 -obsout obslog2

gillespie!!! domain size U=10.000000 (dimension 2), cell_width=1.000000
Number of events per process:
1)    ExplorerSelectTarget[5,6,13,1.0,7,8,7,6,5]      27
2)    MoveOnRoad[6,1.0] 198
3)    MoveOffRoad[7,tophat[0.6,1.0]]      326
4)    MakeObservation1[4,3,8,tophat[3,1.0]]     7
5)    ChangeInType[7,8,0.1]   27
6)    FollowerSelectTarget[9,10,14,1.0,11,12,11,10,9]249
7)    MoveOnRoad[10,0.8]      174
8)    MoveOffRoad[11,tophat[0.6,1.0]]     214
9)    MakeObservation1[4,4,12,tophat[3,1.0]]    134
10)   ChangeInType[11,12,0.2] 25

there are 141 lines in file 'obslog2', beginning looks like:

4 7.000000 7.000000 48.643558 4 8
4 8.000000 3.000000 226.945615 14 8
4 7.000000 4.000000 227.222560 4 8
4 8.000000 3.000000 249.783211 4 12
4 8.000000 3.000000 250.458485 4 12
4 8.000000 3.000000 250.846749 4 12
4 8.000000 3.000000 251.376268 4 12
4 6.000000 2.000000 268.289945 14 8
4 8.000000 3.000000 277.983926 4 12
4 8.000000 3.000000 279.107680 4 12
4 8.000000 3.000000 279.997911 4 12
4 6.000000 2.000000 308.145353 4 12
...
```

Now because follower can observe the same already observed target, there are multiple observations and they can even happen consecutively, i.e. the follower is in the target site making multiple observations from the same target before starting journey back to home. This might cause strange results when calculating summaries unless these replicate observations are somehow cleaned from the output.

Maybe it is not very good idea to have process like MakeObservation1[4,4,12,tophat[3,1.0]], where detection does not change the target type. One could define in the model that all observations change the type of observed species, i.e. are only once observed by explorer (target changes from 3 to 4) and only once observed by follower (in which case species 4 changes into something else), i.e. changing MakeObservation1[4,4,12,tophat[3,1.0]] to MakeObservation1[x,4,12,tophat[3,1.0]], where x is some species.

Other option would be that observer changes its type to back-to-home traveller after detecting the target, i.e. changing MakeObservation1[4,4,12,tophat[3,1.0]] to MakeObservation2[11,12,4,tophat[3,1.0]] in the model.

File ‘model5’:

```
ExplorerSelectTarget[5, 6, 13, 1.0, 7, 8, 7, 6, 5]
MoveOnRoad[6, 1.0]
MoveOffRoad[7, tophat[0.6,1.0]]
MakeObservation1[4, 3, 8, tophat[3,1.0]]
ChangeInType[7, 8, 0.1]
FollowerSelectTarget[9, 10, 14, 1.0, 11, 12, 11, 10, 9]
MoveOnRoad[10, 0.8]
MoveOffRoad[11, tophat[0.6,1.0]]
MakeObservation2[11, 12, 4, tophat[3,1.0]]
ChangeInType[11, 12, 0.2]
```

```
$P/obsimulator -p $P/obsprocesses.txt -m model5 -i xin1 -g road1 -o out -U 10 -T
1000 -rs 4,14 -ek NONE,1,NONE,1,NONE,1,NONE,1 -fk NONE,1,NONE,1,NONE,1,NONE,1 -
obserror obsprobfile4 -obsout obslog3
```

```
gillespie!!! domain size U=10.000000 (dimension 2), cell_width=1.000000
Number of events per process:
1)    ExplorerSelectTarget[5,6,13,1.0,7,8,7,6,5]     28
2)    MoveOnRoad[6,1.0] 199
3)    MoveOffRoad[7,tophat[0.6,1.0]]     247
4)    MakeObservation1[4,3,8,tophat[3,1.0]]    6
5)    ChangeInType[7,8,0.1]   28
6)    FollowerSelectTarget[9,10,14,1.0,11,12,11,10,9]249
7)    MoveOnRoad[10,0.8]     247
8)    MoveOffRoad[11,tophat[0.6,1.0]]    258
9)    MakeObservation2[11,12,4,tophat[3,1.0]]  21
10)   ChangeInType[11,12,0.2] 4
```

now obslog3 has 27 lines, there are still consecutive observations from the same target, but follower has gone back to city between these observations (this can be checked when running simulator with -info 1 which lists all events).

Appendix 1. Process definitions of special functions in file obsprocesses.txt

```
(* species s0 becomes traveller s1 and plans 2-way trip from the beginning of
the road to target t1 using road to nearest location of t1 and then continuing
offroad, kernel a is for s0-t1 pair, s1-s6 are point type changes for traveller
during the journey: s1 forward on road, s2 forward offroad, s3 at target, s4
back offroad, s5 back onroad, s6 back in start point *)
PlanToMoveFromStartOfRoad[s0_, s1_, t1_, a_, s2_, s3_, s4_, s5_, s6_] :=
Module[{Products = {{s1, x1}}, Reactants = {{s0, x1}}, Catalysts = {{t1, y1},
{s2, x2},{s3, x3},{s4, x4},{s5, x5},{s6, x6}}, listAll, function},
listAll = {Products, Reactants, Catalysts}; function[x1_, y1_] := a[x1 - y1];
{listAll, function}];

(* species s0 becomes traveller s1 and plans the offroad trip from its current
location to target t1 with kernel a, once at target location, traveller s1
changes to s2 *)
PlanToMove[s0_, s1_, t1_, a_, s2_] :=
Module[{Products = {{s1, x1}}, Reactants = {{s0, x1}}, Catalysts = {{t1, y1},
{s2, x2}}, listAll, function},
listAll = {Products, Reactants, Catalysts}; function[x1_, y1_] := a[x1 - y1];
{listAll, function}];

(* traveller s1 moves with rate r, note: travel must be first planned using
PlanToMoveFromStartOfRoad *)
MoveOnRoad[s1_, r_] :=
Module[{Products = {{s1, x2}}, Reactants = {{s1, x1}}, Catalysts = {}, listAll,
function},
listAll = {Products, Reactants, Catalysts}; function[x1_] := r; {listAll,
function}];

(* traveller s1 moves with kernel a, note: travel must be first planned using
either PlanToMove or PlanToMoveFromStartOfRoad *)
MoveOffRoad[s1_, a_] :=
Module[{Products = {{s1, x2}}, Reactants = {{s1, x1}}, Catalysts = {}, listAll,
function},
listAll = {Products, Reactants, Catalysts}; function[x1_, x2_] := a[x1 - x2];
{listAll, function}];
```

```
ExplorerSelectTarget[s0_, s1_, t1_, r_, s2_, s3_, s4_, s5_, s6_] :=
Module[{Products = {{s1, x1}}, Reactants = {{s0, x1}}, Catalysts = {{t1, y1},
{s2, x2},{s3, x3},{s4, x4},{s5, x5},{s6, x6}}, listAll, function},
listAll = {Products, Reactants, Catalysts}; function[x1_] := r; {listAll,
function}];

FollowerSelectTarget[s0_, s1_, t1_, r_, s2_, s3_, s4_, s5_, s6_] :=
Module[{Products = {{s1, x1}}, Reactants = {{s0, x1}}, Catalysts = {{t1, y1},
{s2, x2},{s3, x3},{s4, x4},{s5, x5},{s6, x6}}, listAll, function},
listAll = {Products, Reactants, Catalysts}; function[x1_] := r; {listAll,
function}];

(* observer s3 changes s2 into s1 with kernel a (between s2 and s3) *)
MakeObservation1[s1_, s2_, s3_, a_] :=
Module[{Products = {{s1, x2}}, Reactants = {{s2, x2}}, Catalysts = {{s3, x3}},
listAll, function}, listAll = {Products, Reactants, Catalysts};
function[x2_, x3_] := a[x2 - x3]; {listAll, function}];

(* target s3 changes observer s2 into s1 with kernel a (between s2 and s3) *)
MakeObservation2[s1_, s2_, s3_, a_] :=
Module[{Products = {{s1, x2}}, Reactants = {{s2, x2}}, Catalysts = {{s3, x3}},
listAll, function}, listAll = {Products, Reactants, Catalysts};
function[x2_, x3_] := a[x2 - x3]; {listAll, function}];
```

Appendix 2. Example how to plot road network in R

```
plot.graph = function(g,scol,cex=1) {
 for (i in 1:nrow(g)) {
  v=as.numeric(strsplit(as.character(g[i,5]),",")[[1]])
  for (j in v) {
   lines(g[c(i,j),3], g[c(i,j),4])
  }
 }
 points(g[,3],g[,4],pch=19,col=scol[g[,2]],cex=cex)
 points(g[,3],g[,4],pch=1,col="black",cex=cex)
}

g=read.table("roadfile",header=F)
plot.graph(g,c("yellow","green","lightblue"),cex=2)
```