Chapter 6
An Innovative and Open Toolbox

6.1 Introduction

This chapter introduces the platform, OpenMOLE. It is a generic tool used to run the different methods, which is presented in detail in the previous chapters. To simplify the comprehension, we focus on a simple model, but which does not concern the city modelling. However the principles are the same.

The modelling process can be seen as an iterative process, in which specific knowledge is injected and series of issues has to be discussed. How does each input participate in the production of outputs? Are all inputs necessary to generate all the expected dynamics? What are the robustness intervals for the inputs? What are all the possible dynamics of the model? Answering these questions helps in getting a better understanding of the model under construction and a better idea of *what my model is*? OpenMOLE has been thought to answer these questions. It exposes a workflow formalism in which the model is the centre of attention. Numerical experiments can be designed from simple parameter exploration to high level methods dealing with calibration, sensitivity analysis, scenario reproduction.

This chapter presents the central concepts and the OpenMOLE formalism with the example of a simple but stochastic complex-system model. In the first part, we explain how to run a piece of program exposing this stochastic model with OpenMOLE, then we show how to do replications on it, how to explore the input space of parameters according to a Latin Hypercube Sampling (LHS). Finally, we expose three advanced methods: The first one is an evolutionary process, which aims at finding an optimal set of input parameters to simulate a given output (or reproducing a scenario). The second one provides with the validity of the input ranges in the context of the previous scenario reproduction. The third one produces a map of output diversity.

This Ant model has been chosen to serve as a didactic example. It is simple to explain its rules, yet it belongs to the category of complex systems. It is a real-world model getting a minimal set of inputs and outputs, so that the OpenMOLE methodology tools can be easily understood. However, in OpenMOLE, a model can

D. Pumain and R. Reuillon, *Urban Dynamics and Simulation Models*,
Lecture Notes in Morphogenesis, DOI 10.1007/978-3-319-46497-8_6

be viewed as a black box so that it is quite simple to transfer the following methods to an other model.

This chapter does not explain how to instal OpenMOLE, how to launch and how to handle the OpenMOLE application. The reason for this omission is that such instructions are provided and updated on the OpenMOLE website.[1]

6.2 The Ant Model

We propose to study a Netlogo model picked up from the Netlogo Library.[2] However, no skills in Netlogo programming are required. As embedded models in OpenMOLE are encapsulated and can be viewed as a black boxes, the following OpenMOLE scripts can be used for any other language.

The Ant model was created by (Ury Wilensky 1997 and 1999) (Fig. 6.1). The NetLogo's website describes this model as follows: *In this project, a colony of ants forages for food. Though each ant follows a set of simple rules, the colony as a whole acts in a sophisticated way. When an ant finds a piece of food, it carries the food back to the nest, dropping a chemical as it moves. When other ants sniff the chemical, they follow the chemical towards the food. As more ants carry food to the nest, they reinforce the chemical trail.*

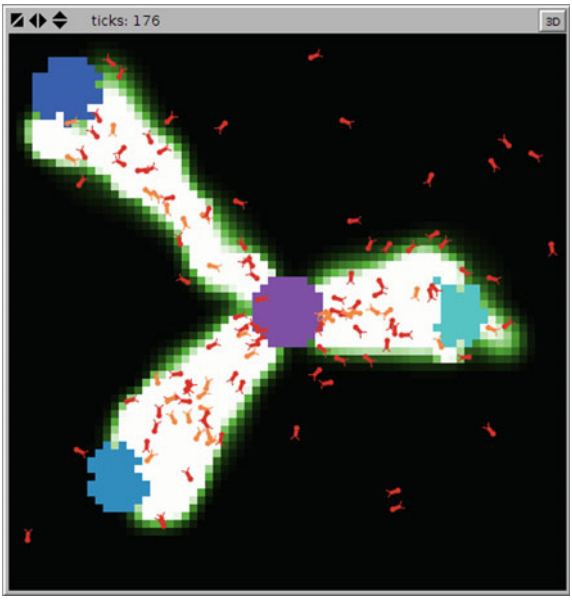


Fig. 6.1 The Netlogo ants model

[1] https://www.openmole.org/.

[2] http://ccl.northwestern.edu/netlogo/models/Ants.

In this experiment, three food spots are set in the ants living area. The experiment consists in testing the impact of the three model inputs on the time required by the ants to consume the three food spots.

The tree inputs of the model are

- the number of ants,
- the evaporation rate of the chemical,
- the diffusion rate of the chemical.

We modified the source code so that we can obtain the food extinction time for each spot

Listing 1 final-ticks-food1, 2, 3 represents the needed number of ticks, measured in simulation steps or ticks (final-ticks-food) to consume the spots 1, 2, 3

```
to compute-fitness
  if ((sum [food] of patches with [food-source-number = 1] = 0)
      and (final-ticks-food1 = 0)) [
    set final-ticks-food1 ticks ]
  if ((sum [food] of patches with [food-source-number = 2] = 0)
      and (final-ticks-food2 = 0)) [
    set final-ticks-food2 ticks ]
  if ((sum [food] of patches with [food-source-number = 3] = 0)
      and (final-ticks-food3 = 0)) [
    set final-ticks-food3 ticks ]
end
```

This model is stochastic. At each time step an ant, which is not sniffing the chemical, can go in any direction randomly. As a consequence, we need to repeat a given experiment (set with given input values) several times to ensure that any pattern generated is robust. Therefore we need to initialize a Random Number Generator by means of a *seed* value.

6.3 Embed the Model in OpenMOLE

The first operation is to run the Netlogo model on the OpenMOLE platform.

OpenMOLE can run executions on High Performance Computating environments. It implies that we need to ensure that any code embedded by the platform can be ported from one machine to another. This depends on the language with which the model is coded. In the case of the Netlogo language, it is straightforward, since Netlogo runs on the Java Virtual Machine, which has been designed to be portable. Otherwise a packaging operation, based of the Care software[3] would be necessary to ensure that all the required libraries at runtime are embedded. We chose not to expose this packaging operation here to focus on methods. However, this operation is simple and well supported in OpenMOLE.

[3] http://reproducible.io/.

An experiment is described in OpenMOLE as a workflow. A workflow is composed of Tasks, which can be chained and ordered by means of another concept: the Transitions. Let us introduce a couple of OpenMOLE concepts:

A Task is an atomic execution component, which can be run concurrently. They tasks have been designed so that they have no interfering side effects. Therefore they can be safely dispatched on several threads, processes or computers. A task can carry a programm, which will be executed at runtime. It receives values (Val) as inputs from the workflow and can produce other values (Val) as outputs.

A Val is a typed Value. It can represent a Double, an Integer, a String, a File (and even Java defined class).

A Transition defines a precedence link between two Tasks. It is always run locally, unlike the Tasks, which can be run on remote environments. It makes the Vals travel from one Task to another.

We first design a very simple workflow containing only one Task (carrying the Netlogo model). We also map the inputs and the outputs of the Netlogo model to some Vals set as inputs and outputs of the Task. Thus we can assign values to the Netlogo model inputs thanks to the mapped Val. In the general case, Task inputs are set with the values of Vals arriving from the workflow by means of a Transition (what we do later in the chapter). But, for now, we just build a very simple workflow composed of one single Task, so that no Transition can feed the Task with any Val. That is why, the input values of the Task are assigned manually.

So we first define seven Vals corresponding to four inputs: the population of ants, the evaporation rate, the diffusion rate, the seed for the RNG as well as maxsteps, which represents the maximum of steps in the Netlogo code. We also define three outputs: the extinction time for the resource spots 1, 2 and 3.

Listing 2 4 Vals for the inputs and 3 Vals for the outputs

```
// Define the input variables
val population = Val[Double]
val diffusion = Val[Double]
val evaporation = Val[Double]
val seed = Val[Int]
val maxsteps = Val[Int]

// Define the output variables
val food1 = Val[Double]
val food2 = Val[Double]
val food3 = Val[Double]
```

We define a NetlogoTask, containing the nlogo source, the launching instructions, the input/output mapping, as well as some manual initialization for the inputs.

Listing 3 Set of the task carrying the model

```
// Define the NetlogoTask
val cmds = Seq("random-seed ${seed}", "run-to-grid")
val ants =
  NetLogo5Task(workDirectory / "ants.nlogo", cmds) set (
```

```
    name := "ants",
    // Map the OpenMOLE variables to NetLogo variables
    netLogoInputs += (population, "gpopulation"),
    netLogoInputs += (diffusion, "gdiffusion-rate"),
    netLogoInputs += (evaporation, "gevaporation-rate"),
    netLogoInputs += (maxsteps, "gmax-steps"),
    netLogoOutputs += ("final-ticks-food1", food1),
    netLogoOutputs += ("final-ticks-food2", food2),
    netLogoOutputs += ("final-ticks-food3", food3),
    // The seed is used to control the initialisation of the
        random number generator of NetLogo
    inputs += seed,
    outputs += (population, diffusion, evaporation, maxsteps),
    // Define default values for inputs of the model
    //seed := 42,
    population := 125.0,
    maxsteps := 2000
)
```

Our first workflow is almost ready! We are just not able to visualize the produced outputs. Indeed, a Task has no side effect, so that it cannot display the value it produces. A Task can be viewed as a portable function, which maps an input value to an output value, nothing more. That is why, we introduce the following concept:

A Hook can be plugged on a Task to perform an action upon completion of the task it is attached to. The action is done locally, once the Task execution is back from an eventual remote host. There exists different kinds of Hooks, among which the AppendToCsvHook to append a Val value at the end of a given CSV file or a ToStringHook to display a Val value.

We need the latter to display the values of food1, food2 and food3. As these three Vals are provided as outputs, plugging a ToStringHook on the Task that produces them will result in their displaying when they are produced by the Task.

Listing 4 Hook plugging

```
//Define a workflow with one Task, hooked by the ToStringHook
ants hook ToStringHook()
```

With these final two lines, the workflow can be run and produces the following output:

Listing 5 Hook displaying th waiting times for the extinction of the 3 food spots

```
{food1=746.0, food2=1000.0, food3=2109.0}
```

6.4 Do Repetitions

An interesting thing is to replicate this stochastic model to get a mean value for the outputs. To do so, we introduce a new kind of Task, a new kind of Transition and a new concept.

A Sampling is a tool for exploring a space of parameters. The term parameter is understood in a very broad acceptation in OpenMOLE. It may deal with numbers, files, random streams, images, etc. There exists a lot of different ways to explore a space of parameters. An exhaustive list of the available Samplings in openmole is given on the https://www.openmole.org/ website.

An Exploration Task is a special Task, whose only setting is a Sampling. Its only goal is to compute the Sampling it carries and to generate all the parameter sets produced by the sampling. It is always followed by a special Transition:

An Exploration Transition links an ExplorationTask to another Task. It creates one new execution stream by sample point in the Sampling of the ExplorationTask. Exploration transitions are represented by the symbol $-<$ (Fig. 6.2).

To carry out the replications on our model, we want to pick up n values from a uniform distribution of integers. Let's admit, we just need 10 repetitions. Then the Exploration Task carrying this Sampling is defined by:

Listing 6 The definition of the Exploration Task and the new workflow statement

```
val replications =
  ExplorationTask (
  seed in UniformDistribution[Int]() take 10) set (
  name := "Replicate ants",
  (inputs, outputs) += (diffusion, evaporation),
  diffusion := 10.0,
  evaporation := 10.0
)

replications -< (ants hook ToStringHook())
```

Listing 7 Results for 10 repetitions

```
{food1=625.0, food2=1311.0, food3=1900.0}
{food1=546.0, food2=1109.0, food3=2574.0}
{food1=526.0, food2=1233.0, food3=2063.0}
{food1=790.0, food2=1214.0, food3=1901.0}
{food1=714.0, food2=1205.0, food3=2133.0}
{food1=534.0, food2=1067.0, food3=2035.0}
{food1=748.0, food2=1338.0, food3=2149.0}
{food1=908.0, food2=1148.0, food3=1821.0}
{food1=682.0, food2=1149.0, food3=1829.0}
{food1=905.0, food2=1315.0, food3=1771.0}
```

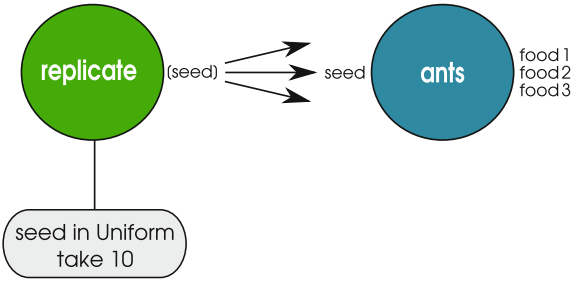


Fig. 6.2 10 different seeds are generated and given as input to 10 instances of the ants Task. Each of them provides food1, food2 and food3

6.5 Automatic Workload Distribution

In the previous section, we generated 10 computation streams. They are independent from one another since they do not require any information from another stream. So that we can easily take advantage of the parallelism with OpenMOLE.

OpenMOLE allows the distribution of computation on servers, on clusters (PBS, OAR, SGE, Slurm, Condor), or on the EGI grid. After having provided with your login/password or your ssh private key or your Grid certificate to the platform depending on what technology you use (see the application documentation for the details on https://www.openmole.org/), delegating the workload on these environments is straightforward. All we need to do is to create the required environment and to specify the Task you want to delegate on it.

Listing 8 Definition of a computational environment (PBS, local multi-core, EGI Grid, …) and assignment to the ants Task, so that the latter will be deported on the previously defined environment at runtime

```
val env = new PBSEnvironment("myLogin", "PBSmachineName")
// val env = LocalEnvironment(10) to take advantage of the
    cores of your own personal emachine
// val env = EGIEnvironment("vo.complex-systems.eu") for
    accessing the Grid VO vo.complex-systems.eu
// etc.

explore -< ants hook ToStringHook() on env
```

6.6 Expose the Variability of the Model

We can use the previous workflow to highlight the variability of the Ants model and to well understand why it is so important to do repetitions on such stochastic models. Let us set both *diffusion* and *evaporation* to 25.0. Then let us do 100,000 repetitions to have an idea of the variability of the model response. The following graphs show

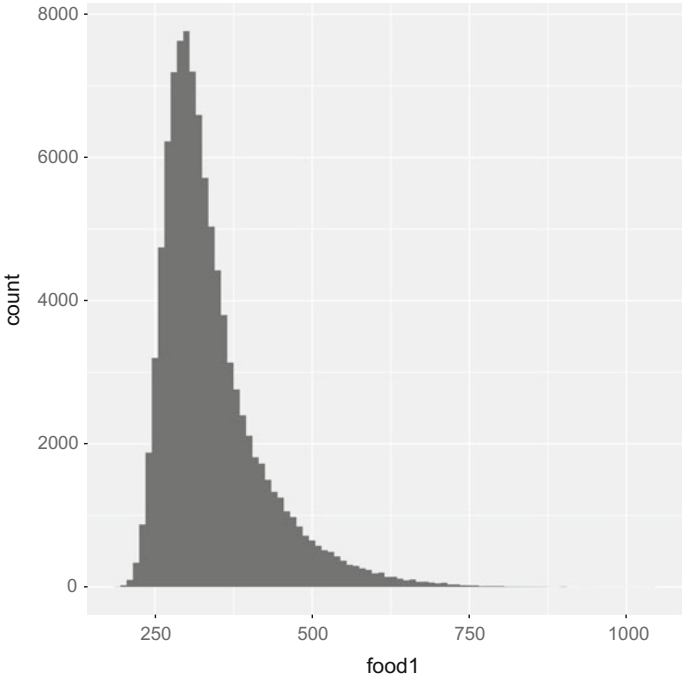


Fig. 6.3 Distribution for the spot food1 for *diffusion* = 25.0 and *evaporation* = 25.0

the required time to consume each food spot for the same input parameters (Figs. 6.3, 6.4 and 6.5).

6.7 Aggregate the Results

We now want to aggregate all the streams and compute a median value on them. To do so, we need a new kind of Transition, which is the counterpart of the Exploration one and merges all the streams generated by the Exploration into one array: **the Aggregation Transition** (represented by $>-$). We then plug another Task onto this transition to perform the median value from that array. For this, we use a Task called a ScalaTask, which can execute some Scala[4] code.

Listing 9 A Task for computing the median values

```
val medFood1 = Val[Double]
val medFood2 = Val[Double]
val medFood3 = Val[Double]
```

[4] http://www.scala-lang.org/.

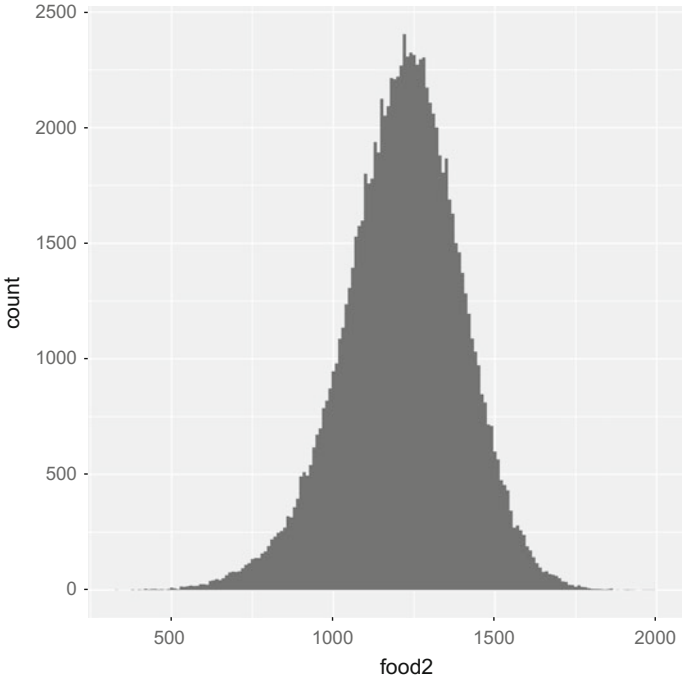


Fig. 6.4 Distribution for the spot food2 for *diffusion* = 25.0 and *evaporation* = 25.0

```
val medians =
  ScalaTask ("""
    import math.abs

    val medFood1 = food1.median
    val medFood2 = food2.median
    val medFood3 = food3.median""") set (
    name := "medians",
    inputs += (food1.array, food2.array, food3.array),
    outputs += (medFood1, medFood2, medFood3)
  )
```

The workflow becomes

Listing 10 A Task for computing the median values

```
replications -< ants >- (medians hook ToStringHook())
```

The resulting workflow can be represented by Fig. 6.6.
The output given by the Hook set on the median Task gives:

Listing 11 Median values for food1, food2 and food3 for 10 repetitions of ants

```
{avgFood1=649.5, avgFood2=1250.0, avgFood3=1979.0}
```

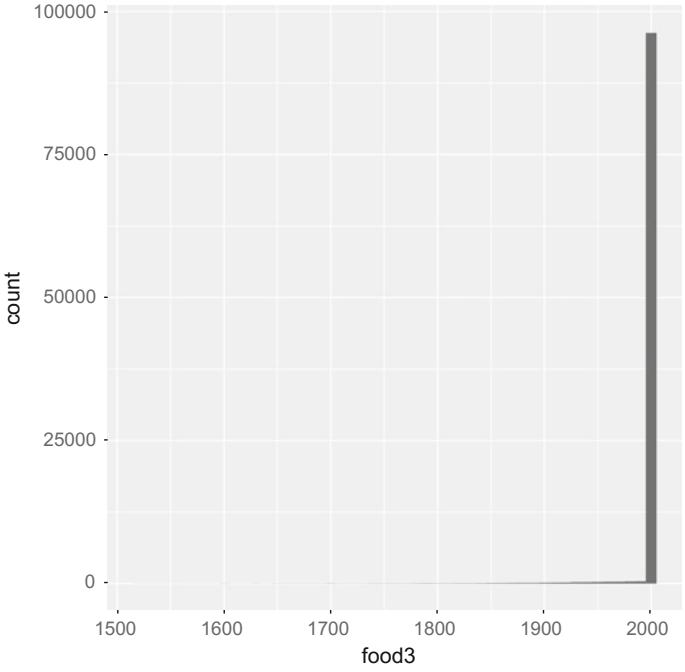


Fig. 6.5 Distribution for the spot food3 for *diffusion* = 25.0 and *evaporation* = 25.0

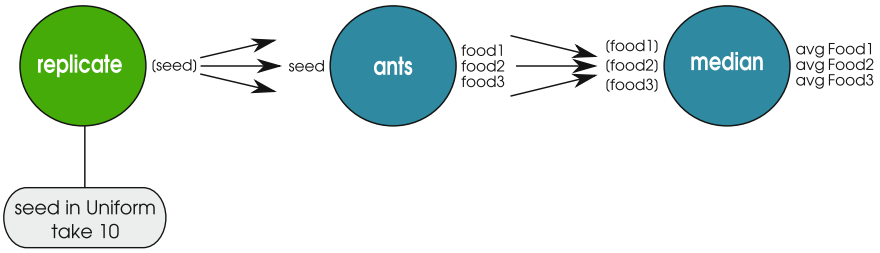


Fig. 6.6 Values generated for food1, food2 and food3 by each of the 10 ants instances are merged into 3 arrays ([food1], [food2] and [food3]) by means of an Aggregation Transition and are processed by the median Task, which provides median values for each array (avgFood1, avgFood2 and avgFood3)

6.8 Explore the Space of Parameters

We now explore the parameter space composed by the evaporation rate and the diffusion rate values to test their individual and combined effects on the time for consuming the food spots. We do not study the impact of the population size since it seems clear that the bigger the population, the faster the food spots will be eaten. The population is thus set arbitrarily to 125.0. To perform the sampling of parameter

values, we use a Latin Hypercube Sampling[5] of size 100. It means that a sampling of 100 couples (diffusion, evaporation) is generated. We evaluate each couple 100 times, leading to 10,000 executions of the model. It implies some modifications in the script.

First, we need to build a new ExplorationTask to carry out the LHS sampling. This exploration will be executed before the replication one, so that we can calculate a median value on 100 repetitions for each sample generated by the LHS.

Listing 12 A LHS sampling is carried out by a TaskExploration and build 100 couples (evaporation, diffusion)

```
val sampling =
  LHS(
    500,
    diffusion in (10.0, 100.0),
    evaporation in (10.0, 100.0)
  )

val exploration = ExplorationTask(sampling)
```

As shown in the Fig. 6.7, diffusion and evaporation are propagated in the workflow through the replicate Task and then directly to the median Task. Indeed, we need these values to be stored at the end of the workflow with medians of food extinction times. This way, we can pair the outputs to the inputs used to generate them.

To do so, we add these two parameters as input and as output of the replicate Task and we add a Transition between the replicate Task and the median Task (which takes also these two parameters as inputs). At this point, we need to introduce two new concepts.

The Capsule: carries a Task and several Slots.

A Slot is a synchronization point for all the Transitions arriving on it. It guarantees that all the Transition transmissions are completed before starting the Task carried by the Capsule. When a Task is created, a Capsule is automatically generated to carry it. Sometimes, we need to create it manually to keep a reference on it.

In our case, we need to create the Capsule of the replicate Task in order to build two Transitions: one to the ants Task and the other to the median Task. On the other hand, we need to create manualy the Slot of the median Task to make a synchronization point between the Transitions arriving from the ants Task and the replicate Task. The Fig. 6.7 and the Listing 13 give an overview of this technical rearrangement.

Listing 13 The full script of the experiment

```
val seed = Val[Int]
val population = Val[Double]
val diffusion = Val[Double]
val evaporation = Val[Double]
val maxsteps = Val[Int]

// Define the output variables
```

[5] https://en.wikipedia.org/wiki/Latin_hypercube_sampling.

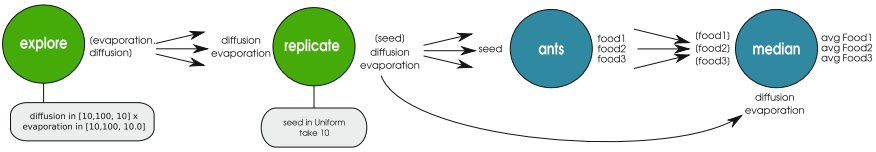


Fig. 6.7 An Exploration Task is designed to make vary diffusion and evaporation. It generates an array of couples (diffusion, evaporation), which combines all different possible combinations of the two variables. These values are transmitted to the replicate Task and then to the median Task, so that they can be stored in a file thanks to the Hook set on the median Task

```
val food1 = Val[Double]
val food2 = Val[Double]
val food3 = Val[Double]

val medFood1 = Val[Double]
val medFood2 = Val[Double]
val medFood3 = Val[Double]

// Define the NetlogoTask
val cmds = Seq("random-seed ${seed}", "run-to-grid")
val ants =
  NetLogo5Task(workDirectory / "ants.nlogo", cmds) set (
    name := "ants",
    // Map the OpenMOLE variables to NetLogo variables
    netLogoInputs += (population, "gpopulation"),
    netLogoInputs += (diffusion, "gdiffusion-rate"),
    netLogoInputs += (evaporation, "gevaporation-rate"),
    netLogoInputs += (maxsteps, "gmax-steps"),
    netLogoOutputs += ("final-ticks-food1", food1),
    netLogoOutputs += ("final-ticks-food2", food2),
    netLogoOutputs += ("final-ticks-food3", food3),
    // The seed is used to control the initialisation of the
        random
number generator of NetLogo
    inputs += seed,
    outputs += (population, diffusion, evaporation, maxsteps),
    // Define default values for inputs of the model
    //seed := 42,
    population := 125.0,
    maxsteps := 2000
 )

val replications =
   ExplorationTask (
    seed in UniformDistribution[Int]() take 100) set (
    name := "Replicate ants",
    inputs += (diffusion, evaporation),
    outputs += (diffusion, evaporation),
    diffusion := 10.0,
    evaporation := 10.0
  )
```

```
val medians =
  ScalaTask ("""
    import math.abs

    val medFood1 = food1.median
    val medFood2 = food2.median
    val medFood3 = food3.median""") set (
    name := "medians",
    inputs += (food1.array, food2.array, food3.array),
    outputs += (medFood1, medFood2, medFood3)
  )

val sampling =
  LHS(
    100,
    diffusion in (10.0, 100.0),
    evaporation in (10.0, 100.0)
  )

val exploration = ExplorationTask(sampling)

val storeHook = AppendToCSVFileHook(workDirectory /
    "result.csv")

exploration -< Strain(replications -< ants >- medians) hook
    storeHook
```

The output of this experiment, stored in the result/result.csv file gives an exploration of 100 different sets of parameters, each having been repeated 100 times. Using this method, we can find the best input couple, which leads to the scenario we aim at simulating. For instance, we may be interested in producing the following real-world experiment: the spots 1, 2 and 3 are emptied in respectively 250, 400 and 800 seconds. So, we are looking for the lowest distance between the simulated output and the expected output, which can for instance be expressed as the sum $|250 - \text{avgFood1}| + |400 - \text{avgFood2}| + |800 - \text{avgFood3}|$

The closest simulation to this target gives the minimal sum of 197. Of course the best score is reached if the experiment reproduces exactly the real case (meaning a sum of 0). The input values associated are presented in the following table:

diffusion	evaporation	Sum of differences
37.8	10.0	197.0

Well, we find *one* solution. 100 simulations (with 100 repetitions for each, i.e. 10,000 runs) might seem like a large-scale experiment but a continuous two-dimensional problem may produce a lot of heterogeneity in the output space. Is there

a better solution to this problem and to which extent is it better? What are the validity intervals for the inputs? What does the output space of parameter look like? So many questions we try to answer with evolutionary methods.

6.9 Optimization with Genetic Algorithms

In a genetic algorithm, an individual carries a genome, which is a set of genes (values for each input parameters). Evaluating an individual means executing a model simulation with the parameter values in the genome and performing the desired measures on the model output. The set of measured values constitutes what we will call here a pattern. Each simulation thus generates a pattern. When the model is stochastic, we can take the mean or median pattern of several simulation replications with the same parameter values. In the end, an individual is composed of the genome and the associated pattern.

Back to our ants optimization problem, the objective here is to find the closest pattern to a experimentally measured pattern (value 250, 400 and 800 for avgFood1, avgFood2, avgFood3 respectively). This problem is also called a calibration problem. To do so, we use the multi-criteria optimization genetic algorithm NSGA2 available in OpenMOLE and used for the calibration of SimpopLocal, cf. Chap. 3. It takes the following parameters as inputs:

- mu: the number of individuals to be randomly generated in order to initialize the population,
- objectives: the objectives to minimise,
- genome: the sequence of model input parameters on which the optimization is done, with the associated lower and upper bounds,
- replication: the repetition strategy

Listing 14 The NSGA2 settings in OpenMOLE

```
// Execute the workflow
// Define the population (10) and the number of generations
    (100).
// Define the inputs and their respective variation bounds.
// Define the objectives to minimize.
// Assign 1 percent of the computing time to reevaluating
// parameter settings to eliminate over-evaluated individuals.
val nsga2 =
  NSGA2(
    mu = 50,
    genome = Seq(
      diffusion in (0.0, 99.0),
      evaporation in (0.0, 99.0)),
    objectives = Seq(deltaFood),
    replication = Replication(seed = seed, aggregation =
        Seq(median))
  )
```

The variable foodTimesDifference is a new Val, representing the sum of absolute differences between the experimental time to reproduce and the simulated times.

We also need to cope with the distributed computation. OpenMOLE offers several approaches to tackle this problem. Among them, we are here interested in the steady-state approach. This algorithm begins with n individuals and launches a maximal number of evaluations as long as there are available computing units. When an evaluation is over, it is integrated in the population and a new individual is generated and evaluated on the computing unit that has just been freed. This method uses all computing units continuously and is recommended in a cluster environment.

Listing 15 Distribution in OpenMOLE with the steady approach

```
val evolution =
 SteadyStateEvolution(
   algorithm = nsga2,
   evaluation = ants -- objective,
   parallelism = 10,
   termination = 100
 )
```

We feed SteadyGA with the evolution method that was described above (*nsga*2) and the piece of workflow to be evaluated (*evaluation*). The *parallelism* parameter specifies how many evaluation are concurrently submitted to the execution environment and *termination* is the termination criterion; here it runs for 100 generations (note that this parameter can also be set as a duration (10 h for example)). SteadyGA launches new evaluations as long as current evaluations are below this value.

SteadyGA returns two variables called in our example puzzle and ga. The second contains information on the current evolution and allows to define hooks that save the current population into csv file or to print the current generation. The following code provides 2 Hooks to (i) save the population corresponding to each generation into a file results/population#.csv, where # is replaced by the number of the generation and (ii) to display in console the generation number:

```
// Define a hook to save the Pareto frontier
val savePopulationHook = SavePopulationHook(evolution,
    workDirectory / "results")
```

When we launch this OpenMOLE workflow, the evolution will progressively produce parameter values having the best fitness, i.e. for which the model is closest to experimental values. We show the evolution of the distance between simulation and experimental measures between successive evaluations in the following Fig. 6.8.

In this table is presented the best result at the end of 800 evaluations.

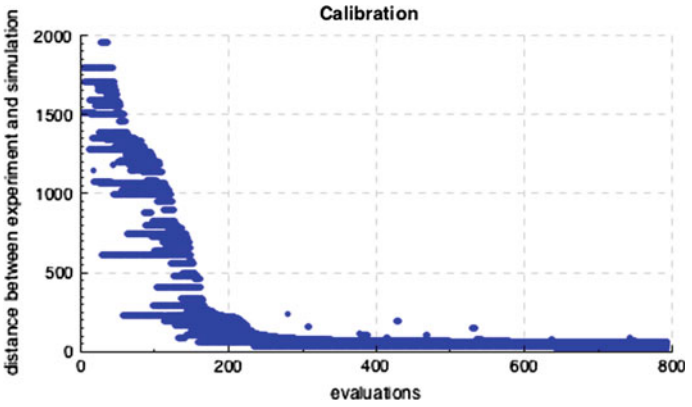


Fig. 6.8 Evolution of the distance between the experimental values and the model. It converges in less than 300 hundred evaluations

diffusion	evaporation	Sum of differences
71.17	5.61	15.5

This result is better than the one obtained with the LHS exploration method. The sum of differences is more than 6 times lower in almost less than half the evaluation time. It is also interesting to notice that the input value are in a completely different regions of space: (71.17, 5.61) versus (13.27, 10.18). It demonstrates how the Genetic Algorithm is faster and more efficient in this kind of optimization problem. The difference between the two methods would be even greater in higher dimensionality problems.

6.10 Sensitivity Analysis with the Profiles Method

The method we now present focuses on the impact of the different parameters in order to better understand how they contribute to the model overall. In our Ants example, we calibrated the model to reproduce a set of notional experimental measurements. We would like to know whether the model can reproduce this pattern for other parameter values. It may be that the model cannot reproduce the experimental measurements if a crucial parameter is set to a value other than the one found by the calibration process. On the other hand, another parameter may prove not to be essential at all; that is, the model may be able to reproduce the experimental measurements whatever its value. To establish the relevance of our model parameters, we will investigate the parameters' profiles for the model and for the targeted pattern.

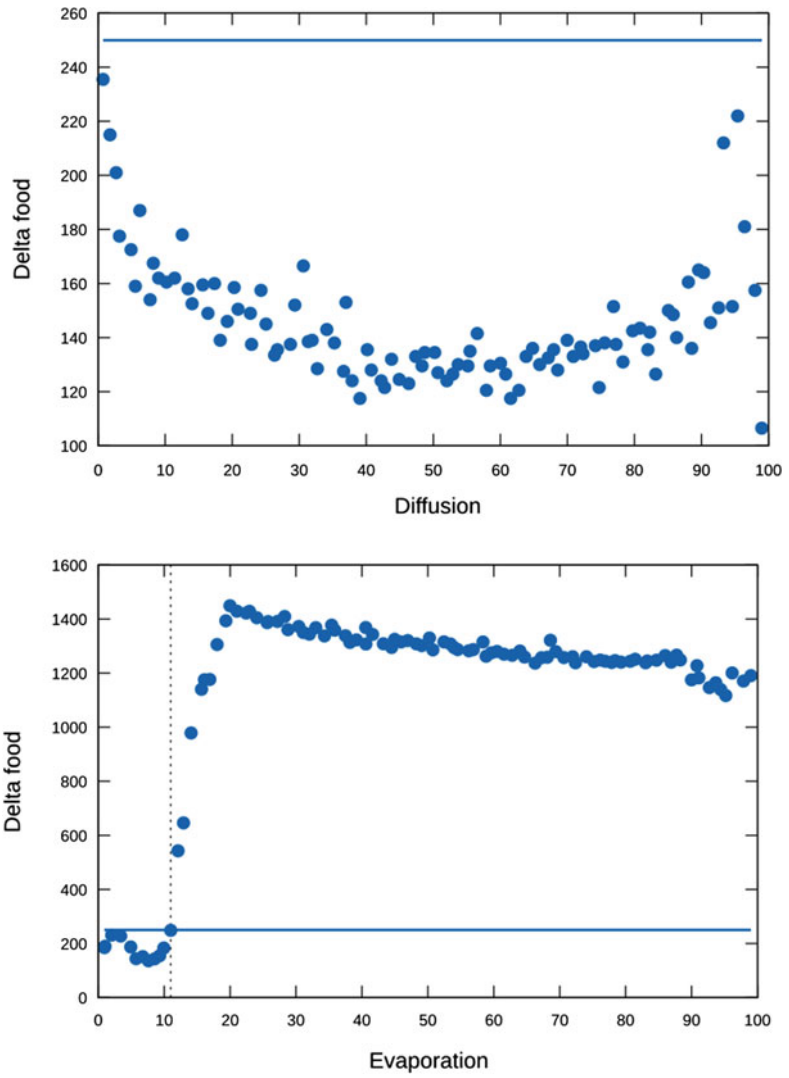
We first establish the calibration of the evaporation parameter. Specifically, we would like to know whether the model can reproduce the targeted pattern with different evaporation rates. We divide the parameter interval into *nX* intervals of the same size, and we apply a genetic algorithm to search for values for other the parameters (the ants model only takes two parameters, so that the dispersal parameter is the only one to be varied), which, as done previously in the calibration, minimise the distance between the measurements produced by the model and the ones observed experimentally. In the calibration case, we kept the best individuals of the population whatever their parameter values. This time, we still keep the best individuals, but we keep at least one individual for each interval division of the profiled parameter (in this case, the evaporation parameter). Then, we repeat the process with the dispersal parameter.

To set a profile for a given *variable* in OpenMOLE, the GenomeProfile evolutionary method is used:

```
def profile(variable: Val[Double]) = {
  val profile =
    GenomeProfile (
      x = variable,
      nX = 100,
      genome = Seq(
        diffusion in (0.0, 99.0),
        evaporation in (0.0, 99.0)),
      objective = deltaFood,
      replication = Replication(seed = seed)
    )

  // Calibration profile of 1000 points for the parameter
  val evolution = SteadyStateEvolution(
    algorithm = profile,
    evaluation = ants -- objective,
    termination = 20000
  )
```

The arguments *genome*, *termination*, *objective* have the same role as the calibration workflow. The argument *objective* is in this instance not a sequence but a single objective to minimise. The argument *x* specifies the index of the parameter to be profiled, i.e. its position within the inputs sequence, indexing starting at 0. *nX* is the size of the of the interval in the parameter range discretisation.



When the diffusion rate is set to any value above 10, the model is able to reproduce experimental measures rather accurately. A refined profile within the interval [0; 20] may be useful to give a more precise picture of the change in the influence of the parameter. Model performance is on the contrary strongly sensitive to the evaporation parameter, as values over 10 lead to a strong increase in minimal fit. When running the model with a diffusion rate of 21 and evaporation rate of 15, we observe that the ants are not able to build a sufficiently stable pheromone path between the nest and furthest food pile, which increases the time needed to exploit it in a considerable way.

6.11 Validation, Testing Output Diversity

Knowing that a model can reproduce an observed phenomenon does not ensure its validity. By validation, we mean that we can trust it to explain the phenomenon in other experimental conditions and that its predictions are valid with other parameter values. We have already established that one way to test a model is to search for the variety of behaviours it can exhibit. The discovery of unexpected behaviours, if they disagree with the experimental data or the direct observation of the system it represents, provides us with the opportunity to revise the assumptions of the model or to correct bugs in the code. This principle also holds for the absence of expected pattern discovery, which reveals the inability of the model to produce such patterns. As we test a model and as we revise it, we can move toward a model we can trust to explain and predict a phenomenon.

One might wonder, for instance, if in our ant colony model the closest food source is always exploited before the furthest. Accordingly, we decide to compare the different patterns that the model generates, looking specifically at the amount of time the model requires to drain the closest and the furthest food sources.

As in the previous experiment, we consider a task that runs 10 replications of the model with the same given parameter values and that provides, as its output, the median pattern described in two dimensions by the variables medFood1, the time in which the closest food source was exhausted, and medFood3, the time in which the furthest food source was exhausted.

To search for diversity, we use the PSE (Pattern Space Exploration) method (Chérel et al. 2015). As with all evolutionary algorithms, PSE generates new individuals through a combination of genetic inheritance from parent individuals and mutation. PSE (inspired by the novelty search method) selects the parents whose patterns are rare compared to the rest of the population and to the previous generations. In order to evaluate the rarity of a pattern, PSE discretises the pattern space, dividing this space into cells. Each time a simulation produces a pattern, a counter is incremented in the corresponding cell. PSE preferentially selects the parents whose associated cells have low counters. By selecting parents with rare patterns, we have a better chance to produce new individuals with previously unobserved behaviours.

In order to use PSE in OpenMOLE, the calibration utilized in the previous section is run with a different evolution method. We used to provide the following parameters:

- genome: the model parameters with their minimum and maximum bounds,
- objectives: the objectives measured for each simulation and within which we search for diversity,
- parallelism and termination have the same meaning as in the calibration example.

Here is the OpenMOLE code used for the PSE

```
val pse =
  PSE(
    genome = Seq(
      diffusion in (0.0, 99.0),
```

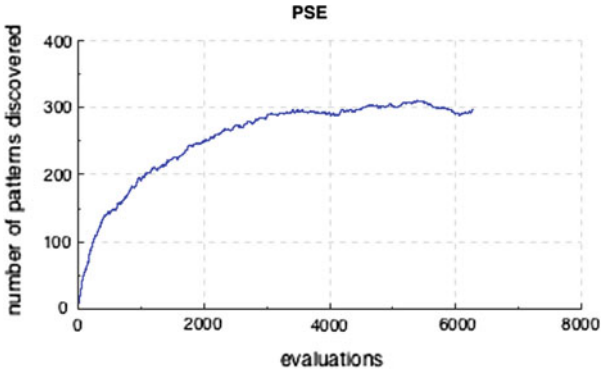
```
    evaporation in (0.0, 99.0)),
  objectives = Seq(
    food1 in (0.0 to 4000.0 by 50.0),
    food3 in (0.0 to 4000.0 by 50.0)),
  replication = Replication(seed = seed)
)

val evolution =
 SteadyStateEvolution(
  algorithm = pse,s
  evaluation = ants,
  parallelism = 10,
  termination = 1000000
)
```

As the exploration progresses, new patterns are discovered. The following figure gives the number of known patterns (the number of cells with a counter value greater than 0) with respect to the number of evaluations.

When this number stabilizes, PSE is no longer making new discoveries. One has to be careful when interpreting this stabilization. The absence of new discoveries can mean that all the patterns that the model can produce have been discovered, but it is possible that other patterns exist but that PSE could not reach them.

The following figure shows the set of patterns discovered by PSE when we interrupt the exploration after it stabilizes.



The first observation that can be made is that all patterns have indeed been discovered: in every pattern, the closest food source has been drained before the furthest one. Further, there seems to be minimum and maximum bounds on the time period during which the nearest food source is consumed.

These observations give us starting points for further reflections on the collective behaviour of the ants. For instance, is the exploration of the closest food source systematic? Could there be ant species that explore further food sources first? If we found such a species, we would have to wonder which mechanisms make it possible and revise the model to take them into account. This illustrates how the discovery

of the different behaviours the model is able to produce can lead us to formulate new hypotheses of the system under study, to test them and to revise the model, thus enhancing our understanding of the phenomenon.

References

Chérel, G., Cottineau, C., Reuillon, R.: Beyond corroboration: Strengthening model validation by looking for unexpected patterns. PLoS ONE **10**(9), 1–28 (2015)

OpenMOLE scientific workflow, distributed computing and parameter tuning. https://www.openmole.org/

Wilensky, U.: NetLogo Ants model. Center for connected learning and computer-based modeling, Northwestern institute on complex systems, Northwestern University, Evanston, IL (1997). http://ccl.northwestern.edu/netlogo/models/Ants

Wilensky, U.: NetLogo. Center for connected learning and computer-based modeling, Northwestern institute on complex systems, Northwestern University, Evanston, IL. (1999). http://ccl.northwestern.edu/netlogo/