Classifying simulation methods

Guillaume Chérel

2018-09-12

Contents

Introduction

A model on its own has no purpose. It requires a scientific question and a simulation experiment to produce an answer to that question: how shall one run simulations, with what input parameters, how should they vary... The process of modelling is that of finding a path that connects a question to an answer, with models and simulation experiments. To draw this path, how one constructs their models depends on the experiments they can design, thus on the *simulation methods* they know of and can use.

A simulation methods is an algorithm that takes a simulation model and runs it in a certain way to produce some data. For example, a very simple method is to run a simulation for a given list of parameter values and compute the mean and standard deviation of the model output. Many more methods are available, and a lot of questions can be answered with an appropriate combination of them.

In the team developping the OpenMOLE platform, we work with researchers of various fields and help them design simulation experiments. With time, we find

ourselves developping an increasing number of methods. We are looking for a way to organise them and communicate about them.

Simulation methods can be experessed as functions, and function types are a good way to describe functions, abstracting away the internal details. We propose to use types to describe and classify simulation methods.

The objective is that upon reading a method type, one should be able to guess the method purpose without having in depth knowledge of its mathematical or algorithmic details. Readers should know if the method is applicable do their case of study, including the structure of its model and the scientific question.

The following 2 sections will clarify the notion of function type and show where functions appear in simulation. Then, we will review some examples of simulation methods and their types.

Function type

In computer science, the notion of a function is close to its mathematical counterpart. It takes a value and returns another. We will add some information by using types: a function takes a value of some type a, and returns a value of another type b. Let's call such a function f. We will write:

$$f : a \to b$$

The part $a \to b$ is called the the *function type*. As we will see, it carries some meaningful information.

For example, we can define a function which takes a letter in the alphabet and returns a natural number from 1 to 26, such that "A" is associated to "1", "B" to "2", ..., "Z" to "26". Let's call l this function. To refer to the output value of the function, we usually write $l(\text{"}A\text{"}), l(\text{"}B\text{"}), \ldots$. For example, $l(\text{"}C\text{"}) = 3$.

This function takes a value of type *Letter of the alphabet*, and returns a value of type *Natural number*, or Nat for short. So we will write l : *Letter of the alphabet* $\to$ *Nat*, and *Letter of the alphabet* $\to$ *Nat* is l's type.

As another example, the function *abs* which gives the absolute value of an integer, such that $abs(-3) = 3$ and $abs(2) = 2$, has the type: $Int \to Nat$, where Int denote the types of integers.

A function can take multiple arguments. We will write their types with multiple arrows: *arg1 type* $\to$ *arg2 type* $\to$ *arg3 type* $\to \cdots \to$ *result type*. Let *add* be function that takes two integers and returns their sum. Its type is: $Int \to Int \to Int$. We get its result by writing $add(a)(b)$. For example, $add(1)(2) = 3$.

A function can also return a function. The type of a function that takes a value of type a and returns a function of type $b \to c$ is written $a \to (b \to c)$. In

fact, the type of the function *add* can also be read as $Int \to (Int \to Int)$. The function *add* can equivalently be seen as a function which takes two arguments and returns a value, or as a function which takes one argument and returns a function, itself taking one argument and returning a value. To emphasize the second interpretation, we can note that if we give a single argument to the function *add*, we get the function: $add(a) :: Int \to Int$. If we pass a value b to this function, we get the sum of a and b: $add(a)(b) = a + b$. This notation is the same as with multiple arguments, this is why the signatures $a \to b \to c$ and $a \to (b \to c)$ are equivalent.

More generally, a function taking n arguments can be seen as a function that takes 1 argument and returns a function taking $n - 1$ arguments, all the way down. So, the types

$$a_1 \to a_2 \to a_3 \to \cdots \to a_n \to b$$

and

$$a_1 \to (a_2 \to (a_3 \to (\cdots \to (a_n \to b) \ldots)))$$

are equivalent.

Just like a function can return a function, it can also take a function as argument. The type of a function taking a function of type $a \to b$ and returning a value of type c will simply be written $(a \to b) \to c$. Because we read function types from left to right, this time, the parentheses are mandatory. If we forget them, we are writing the type of a function taking two arguments as seen above:

$$a \to b \to c = a \to (b \to c) \neq (a \to b) \to c$$

The meaning of function type written with concrete types like $Int \to Nat$ is clear. A function type written with type variables like $a \to Int$ means that the function takes a value of any type and returns an integer. The type $a \to a$ is for a function that takes a value of any type and return a value of the same type. For example, let's consider a function *map* that takes a function $f : a \to b$, a list of values of type a, and applies f to all the elements of the list and returns the new list. For example:

$map(add(1))(List(1, 2, 3)) = List(2, 3, 4)$
$map(l)(List("A", "B")) = List(1, 2)$, where l is the function defined above.

The type of *map* is: $(a \to b) \to List(a) \to List(b)$. We start to get an idea of how function types carry information about what a function does.

Models and methods as functions

A simulation model usually takes some inputs that can be parameters, initial conditions, and returns some output. We propose to see a model as a function of

type $x \to y$. This is the most generic type for a simulation model. Let's consider a simple predator-prey model which simulates the population dynamics of sheeps and wolves, takes as input the initial number of both populations, a number of years, and returns the number of sheeps and wolves as many years later. This model can be represented by a function of type $(Nat, Nat, Nat) \to (Nat, Nat)$. Models inputs and outputs can be arbitrarily complex: they can be composed of arrays, matrices, geographical data, networks, time series, ...

A single type can be attributed to multiple functions. There can be different functions to model the dynamics of two populations of preys and predators. They may encode different dynamics and answer differently to the same input values, but as long a they take as input a number of prey and a number of predators, and give the same information back, they all share the same type.

A simulation method takes a model, runs simulations, and returns information based on the simulation output. A simulation method can be represented by a function of type: $(x \to y) \to r$, where r is the result type. Usually, simulation methols will take additional arguments. For example, model calibration aims at finding a model input value such that the model produces a desired output. It requires as additional argument the desired output. Such a method may have a type like:

$$y \to (x \to y) \to x$$

We propose to use types to classify simulation methods. In the following, we will see a list of generic methods which work on a wide class of models. We will see that we can guess from the method type alone the purpose of the method.

Methods and their types

The following are examples of simulation methods and an illustrations of how they can be described by types. The intention is to show how we can use types to classify methods rather than to propose a definitive classification. As we will see, there is more than one way to describe a single method.

Direct sampling

Direct sampling is straightforward: run a model on a sequence of input values, and get pairs of the inputs and associated outputs. A good implementation may take care of distributing the simulations in parallel to significantly reduce the overall simulation time. It does little work for the modeller beyond that: it is left to them to choose how to use the list of associated inputs and outputs. Different sampling methods can be used to construct the input list, with different properties: uniform sampling of the input space, grid sampling, one factor at a time, latin hypercube sampling, Sobol sequences, etc. Each sequence have

different properties which we will not detail here but can affect the usage of the result.

All direct samplings, however the sequence is generated, may have the following generic type:

$$(x \to y) \to List_n((x, y))$$

where $List_n(x)$ should be read as "a list of n elements of type x".

A particular direct sampling method will have a more specific type, reflecting the way the sequence is built. For example, a simple direct sampling can just require the users to build the sequence themselves. It may be represented as the function:

$$simpleDS : List_n(x) \to (x \to y) \to List_n((x, y))$$

If we give it a list l of type $List_n(Int)$, for example, it returns a function:

$$simpleDS(l) : (Int \to y) \to List_n((Int, y))$$

We say that this type is more specific than the type $(x \to y) \to List_n((x, y))$ above because we can turn the latter into the former by replacing x by Int.

We have our classification criterion, we say that a method represented by a function f is a direct sampling if it can return a function $f' : (x' \to y') \to List_n((x', y'))$ whose type is more specific than the generic direct sampling type.

As another example of direct sampling, we could sample the input values uniformly. We know how to sample a real value uniformly within a finite interval. We can repeat the process to generate input values for a model that takes as input k real values. This method would require the user to provide it with an interval for each real value of the model input, and a seed to initialise the random number generator by sampling. It could be represented by the function:

$$uniformRealDS : List_k(Interval) \to Seed \to (List_k(Real) \to y) \to List_n((List_k(Real), y))$$

With the type *Seed* appearing in the function type, we are making explicit that this method is stochastic, suggesting that replications may be necessary in order to draw robust conclusions. Because the sampling process used produces real values, this function restricts the type of the model to one that takes k real values as inputs, where k is the same as the number of intervals given as first argument.

If the model takes inputs within a finite number of values, it is also easy to sample uniformly, we just pick one at random n times. A function that would realise this could be:

$$uniformDiscreteDS : List_m(x) \to Seed \to (x \to y) \to List_n((x, y))$$

As a last example, let's use Sobol sequences to build the input list. A Sobol sequence can generate values between 0 and 1 in k dimension (i.e. points in

the k-dimensional unit hypercube), with good space filling properties. A direct sampling using a Sobol sequence can be represented by a function:

$$sobolDS : (Nat, Nat) \to (List_k([0,1]) \to y) \to List_n((List_k([0,1]), y))$$

where the first argument of type (Nat, Nat) is (n, k), the number of input values to generate and the dimension of the input. This method restricts the model type to input of k numbers between 0 and 1.

These examples are different direct sampling methods, using different algorithms, constraining differently the types of models they can work on, but they all fit into the class of direct sampling: they can all return functions that are more specific than the generic direct sampling type.

Calibration

We've already talked about the problem of calibration in the previous section. Its purpose is to find an input value such that the model reproduces some desired output. The result we require from a method of this type is just one input value, even if there are multiple solutions. There is no guarantee that there will be one for any given models. We need to represent that the method can return one solution or none. We will use result type is $Maybe(x)$. It represents either a value a of type x, written $Just(a)$, or a value which represents no value, simply written $Nothing$.

A calibration method should always require a desired output and a model and return a just one value or nothing.

$$y \to (x \to y) \to Maybe(x)$$

It may be difficult to implement algorithms that solve this problem exactly. For example, when the model outputs real numbers, due to the limited precision of floating-point numbers on a computer, it may be impossible for a model to output a certain value in simulation, even if it can in theory.

In OpenMOLE, we use genetic algorithms to give an approximate solution to this problem. A simple genetic algorithm would first creates randomly n candidates (input values) and computes their scores by running a simulation for each, and computing the euclidean distance between the simulation output and the desired output. Then it would select the m closest and recombine them together by pairs to recreate a new population of n candidates. After T generations, it would return the closest candidate. This genetic algorithm would require a number of candidates to generate, a number of best candidates to select, a number of generations to run, a function to recombine a pair of candidates and generate a new one $((x, x)rightarrowx)$ and a random seed. Since we use the euclidean distance to compare the simulation outputs to the desired outputs, the

output type needs to be a sequence of real numbers. The type of the method implemented with this genetic algorithm is:

$$calibGA : (Nat, Nat, Nat) \to ((x, x) \to x) \to Seed \to List_n(Real) \to (x \to List_n(Real)) \to Maybe(x)$$

This method is a calibration in the same sense than the method in the previous sections are direct samplings: it returns a function whose type is more specific than the generic calibration type.

The previous type is more difficult to read than the generic type above. Generally, more specific types require more work from the reader than more generic ones. It is important to give the specific type when designing a method, because it helps users know exactly in what context they can work, and if they are applicable to their case of study. But it is good to fit specific method into more generic types, because those are easier to understand. Intuitively, generic types give the purpose of a method: they tell us *why* one would want to use it. Specific types tell us *how* to use them.

Inverse

The method *inverse* extends calibration: instead of looking for one input value such that the model reproduces the desired output, we are looking for all of them. It takes a desired output, a model, and returns a set of all the input values such that the model reproduces the desired output. The generic type for inverse methods could be:

$$y \to (x \to y) \to Set(x)$$

where $Set(x)$ is the type for a set (in the mathematical sense) of elements of type x.

Interestingly, when we apply an inverse function to a model $m : x \to y$, we get a function that looks precisely like the inverse of the model:

$$inverse(m) : y \to Set(x)$$

that is, a function taking a value of type y and returning values of type x.

A candidate method called OSE (Origin Space Exploration) is currently being developed at ISCPIF.

Diversity

This problem aims at finding all the different outputs that a model can generate. Its type is simple: it is a function that takes a model and returns the set of all possible outputs. In mathematical terms, with the model represented by the function $m : x \to y$, we are looking for the image of m. The generic type is:

$$(x \to y) \to Set(y)$$

One use of this method is to test a model. If the among the model outputs we find unacceptable values, it may be the sign that there is an error in the model assumptions or in the implementation. This gives the modeller an opportunity to investigate and improve the model.

Another use is a kind of sensitivity analysis. Let's say we have a model that we want to use for prediction, and we have measured sensible parameters for the model, but our measurement is noisy. We may use this method by restricting the model parameter so they can only vary slightly around the measured value. By studying the result of the method, we can tell if the model always makes the same prediction when the parameters vary slightly, or if they can change radically. If so, we will probably want to be cautious about using it to make predictions.

Here again, this is the ideal formulation of the problem and we will most likely only design approximations. The difficulty is that, unless we can try exhaustively all the possible input values (which may only be possible for model for which the possible inputs are countable and few), there is usually no way to know if we have found all possible output values or not. It may be that the particular method that we use is unable to find other possible output values and stagnates. One candidate method for this problem is PSE (Pattern Space Exploration), developed at ISCPIF.

Conclusion

Methods can be represented as functions and described with function types. We have introduced a classification criteria that hierarchically organizes method typesm. At the top, the most generic type is $(x \to y) \to r$ in which all simulation method fit. We have seen a few other classes that populate the next level:

- direct sampling: $(x \to y) \to List_n((x, y))$
- calibration: $y \to (x \to y) \to Maybe(x)$
- inverse: $y \to (x \to y) \to Set(x)$
- diversity: $(x \to y) \to Set(y)$

And we have given examples of specific method types that are the leaves of the hierarchy. Those are the method that can actually be implemented and used in a simulation experiments.

Types of more generic classes give us some intuition about what a method does. They can be used to convey concisely the meaning and usage of a method without going very deep into the implementation details, saving researchers time and making them more accessible.

The types of specific method requires more work from the reader to be understood, but give precise information about what information they require from the user and what they give back.

The types of generic classes tell us *why* one would want to use a certain method. Specific types tell us exactly *how* to use them.