9

Computational models in science

Introduction

In ordinary conversation and even in some scientific discourses, there often seems to be no difference between a formal model and a computational model. Unfortunately, conflating these notions or concepts is not without epistemological consequences. It is true that all computational models rest upon formal models. But the reverse is not to be taken for granted. It is not because one has a formal model that a computational model will be readily available.[1]

The following explanations may appear technical for semioticians, but they are essential for understanding the specific definitions and roles of computation in scientific enquiries. This is required for semiotics to adequately integrate computation into its practice. Semiotics cannot enter the digital world without understanding more precisely the notion of computation.

As we shall see, computational models represent a heavy burden in the encounter of semiotics with the digital. This means that if one applies computer technologies to the analysis of semiotic artefacts that cannot be modelled by formally calculable functions, then the results obtained will not be trustworthy. Therefore, if semiotics is to enter the digital world, it is essential to better understand what is computable per se and what is not computable. Otherwise, the adventure into the digital will rapidly meet a dead end.

Computational models in science: Definitions

Computational models are deeply related to formal models. They are a particular sort of formal model. Still, symbols and formulas used in computational models present some specific properties that give them their own signature.

Briefly defined, a computational model is one that constructs a special type of formal symbolic system. Its syntax restricts the type of admissible symbols, and it defines a particular type of rules for the manipulation of formulas. Its semantics refers to a particular type of mathematical structure: functional relations that can be 'calculated'. And its associated pragmatics allows for effective procedures that a computer can implement.

A computational model is therefore highly related to a formal model and ultimately to a computer model. With regard to the formal model, a computational model is a set of formal statements or rules that express how the functions of the formal models can be effectively calculated. And with regard to a computer model, the computational rules must be concretely implementable in a physical machine or computer.

These characteristics of computational models have an important and decisive epistemic effect on scientific enquiries. They restrict the choice of possible formal models that can be used if the enquiry is to use a computer. And finally, it is only if the formal model itself contains some computable functions that these may, in turn, be included in the computational model – it is only then that a computer model can offer a technology that can effectively compute them. Thus, even if computational models are ontologically deeply entangled with formal and computer models, both models are not identical. And they have specific properties and operations.

What mainly characterizes a computational type of model is that its core concept, *computation*, is about a formal property, not a technology. It appeared as an answer to questions raised by Hilbert: Is there a procedure by which it can be decided, just by manipulating the symbols, whether a specific mathematical formula or equation belongs or not to a formal system? In other words: what is the calculability or computationality of a mathematical formula or equation?

Among the many solutions offered, two became very important. One of the first propositions was the Church Thesis (Church 1936). For Church, calculability was to be understood through a type of formal language[2] that manipulates *recursiveness*: the λ-definable calculus. Even though Gödel himself had also worked on recursiveness, he regarded it as 'thoroughly unsatisfactory' (Sieg 2006). Practically at the same time, another solution was offered by Turing (1937): calculability would be equivalent (not synonymous) to '*computation*' if there existed an *effective* procedure for *systematically* and *productively* generating output symbols from some initial input symbols.

To demonstrate this thesis, he built two 'machines' (Figure 9.1). A first one (called an *abstract* machine or *Machine A*) was defined by a list of symbols q_i (0 & 1) and a transition state S_i. Machine A formally represents a function through a sequence of instructions expressed in formulas containing symbols of type q_i (0 & 1) and S_i.

The second machine (called a *physical automaton* or *Machine B*) was a physical machine made of mechanisms containing elements such as a paper roll, some wheels and reading and printing devices, and it was subsequently called a 'Turing machine' by Church (1937).

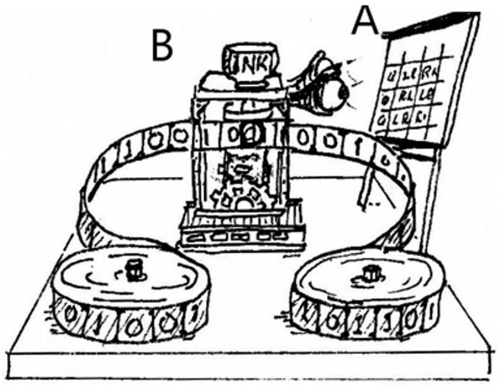

Figure 9.1 Turing Machines A & B.

Technically, this meant that if a formal function was *calculable* (Machine A), then it was *computable* by a physical automaton (Machine B). This claim was later called the

'Turing thesis'. It states that 'every intuitively computable function is computable by an abstract automaton' (Turing 1936).

Turing's proposition, often called the 'Strong Turing Thesis', establishes a relation between two structures: an abstract one and a physical one. The first structure can be called a *calculable* function if and only if it can be processed or 'computed' by an *effective procedure*, that is, a physical machine. Soare formulated the notion of computation in more contemporary terms:

> A computation is a process whereby we proceed from initially given objects, called inputs, according to a fixed set of rules, called a program, procedure, or algorithm, through a series of steps and arrive at the end of these steps with a final result, called the output. . . . The concept of computability concerns those objects which may be specified in principle by computations and includes relative computability. (Soare 1996: 286, emphasis removed)

This thesis will be very important for understanding the relationship between the formal expression of a calculable function given in a formal model and its translation into a computable model. It ultimately means that even if formal models can create formulas that represent complex functional relations for some phenomenon under study, nothing guarantees that they are *computable* and therefore accessible to computer processing. In other words, integrating computer processing in research requires that the formal models called upon must propose only formulas that contain computable functions if they are to be processed.

Both theses were later brought together to form an important integrated thesis called the 'Church-Turing' thesis by Kleene (1952: 300) who gave it its first formulations.

Gandy, a student of Turing, formulated it in the following manner: 'whatever can be calculated by a machine can be calculated by a Turing machine' (Gandy 1980). Rogers[3] (1987) showed that the same class of partial functions (and of total functions) could be obtained in each case.

This Church–Turing thesis[4] has been extended to many other types of formal systems and languages. Briefly summarized, it says that calculable functions are 'equivalent' if they can be computable by a Turing machine. This has been shown for Post's (1936) production rule, Curry and Feys's (1958) combinatorial logic, von Neumann's (1966) automata, the Gandy (1980) machine, Chomsky's (1957) automata grammar and many other types of formal symbolic systems.

One of the important translations that the notion of computation has received is that of *algorithm*. This notion is one by which the computer operations are indeed mostly defined. It has become the keyword for naming the set of rules or instructions that a computer follows to accomplish a task. But it was Markov[5] ([1954] 1960) who demonstrated the equivalence between algorithmic formalization, Church's recursive functions and Turing's machines. And in fact, the instruction part of a Turing machine is algorithmic.

Briefly defined, algorithms are effective procedures to achieve the computation of a function. They are presented and expressed in a language through a sequence of instructions applied to input so as to systematically produce some output. Many other formulations of the notion of *algorithm* exist.

For our research purposes here, there are three implicit distinctions in the definition of algorithms that must be emphasized, for they are often either ignored or conflated. They remain important in understanding the epistemic role of algorithms in computational models.

The first one is that a procedure is said to be an algorithm only if it is applied to a computable function. Algorithms are therefore finite and *effective* procedures. This means that if a problem presents high complexity (N.B.: not complication), its formal model may introduce some undecidable functions. This also means that the effective procedures involved may never stop and hence render algorithms useless.

The second one is that one must not confuse a computable function with an algorithm. This is because the same computable function can be expressed by several algorithms. For example, there are several different algorithms for calculating the arithmetic mean of a set of numbers, as it is a calculable function.

And a third distinction is that there exist many different types of programming languages to express the same computable function and its algorithms. Preferred ones have been von Neumann's 'flow chart' or Miller, Galanter and Pribam's TOTE. But there are other ones such as McCarthy's LISP machine, Anderson's production rules and Chomsky's automata rules. And today, there is a proliferation of high-level languages that can express algorithmic procedures.

This means that even if a programming language is elegant and well-formed, it does not follow that the algorithm and the computable function underlying it is transparent, known, understood and ultimately easily implemented in a computer.

To rapidly illustrate the notion of algorithm, let's take a simple example. Suppose that in our conceptual model, we assert in a natural language the following proposition: it is possible to find the sum of the first one hundred numbers, that is, 'the sum of 1 + 2 + 3 + 4 . . . 100'.

A formal model can be offered as the algebraic equation solution:

$$F(X) = \sum_{i=1}^{100} X_i + X_{i+1} \ldots X_n$$

One other interesting formal expression was given by mathematician F. Gauss:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

where $n = 100$.

These reformulations show that these mathematical functions can be expressed using different formulas or equations. Each one would give the same results.

But each one, in turn, can be translated into an algorithmic programming language such as the following pseudocode for the first equation:

Program to calculate the sum of n digits (Figure 9.2).

One must notice that each instruction given in a programming language and translated into another one requires creative intuitions and expert conceptualization. And it can sometimes require time and energy.[6]

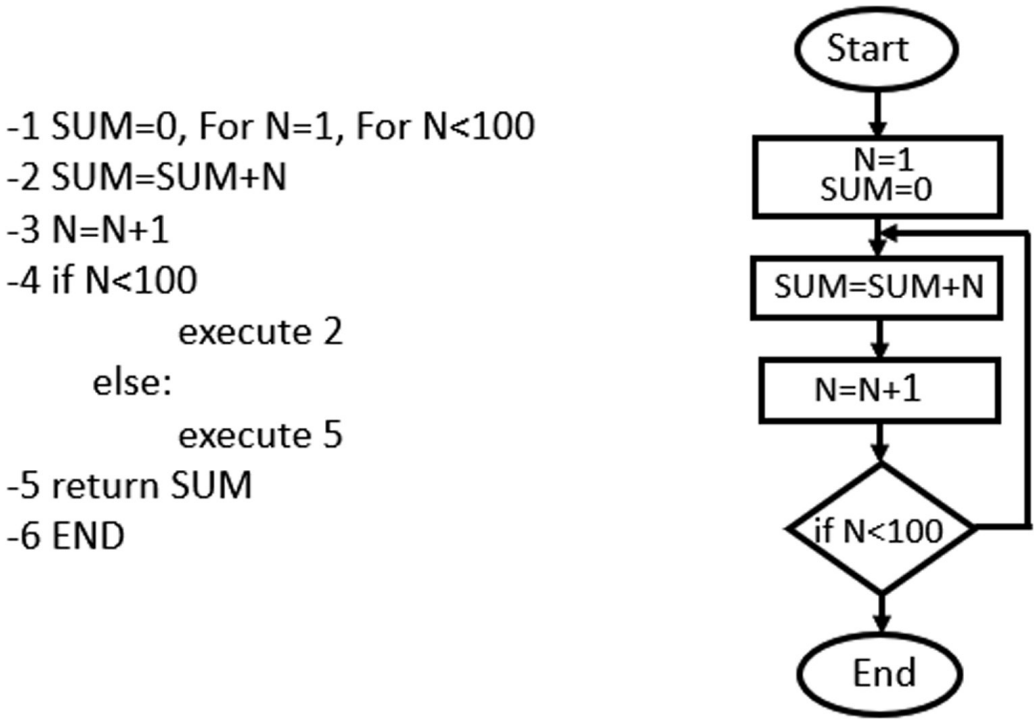

Figure 9.2 Pseudocode and flow chart.

This last algorithmic expression could have been presented in other more detailed programming languages. Each language would have to add more symbols and formulas, some of which pertain to instructions referring to the input data, others to many other types of entities such as variables, identifiers, control flows or external devices (printer, discs, memory, etc.).

Formally speaking, then: 'an algorithm is an intensional definition of a special kind of function – namely a computable function. *The intensional definition contrasts with the extensional definition of a computable function, which is just the set of the function's inputs and outputs*' (Dietrich 1999: 11, emphasis modified).

Finally, let's recall that functions are not always defined intensionally by some formula or algorithm; they can also be defined extensionally. This means that a functional relation can be defined through a data list or, more specifically, through its data structures. These are the arguments and values of the function. And, in fact, some see data structures as an extensional definition of computable functions and therefore as an extensional presentation of an algorithm and program.[7] This interpretation of data as possible arguments and values of computable functions, as we will see later, will be applied to 'big data'. It will give them a very important epistemic role in machine-learning algorithms, for they will be seen as implicitly being the expression of possible computable functions.

Fodor and Pylyshyn (1988) proposed a reformulation facilitating the comprehension of computability by cognitive science and by the digital humanities. According to this reformulation, a function will be recognized as computable if it has at least one of the following properties: atomicity, systematicity and productivity. For the particular

field of cognition to which they applied such a computable function, they added the property of interpretability.

The first property involves the admissibility of inputs (called *arguments*) with respect to a computable function. These inputs must be discrete or 'atomic' – that is, non-continuous. Usually, the digital encoding into 0s and 1s symbolically encodes this property. The second and third properties are the most important ones for our discussion. They specify the nature of the procedure or of the operations which will be applied to these inputs. Indeed, these must be *systematic* and *productive*, that is, the procedures must enable us to *systematically produce* increasingly complex sequences of symbols or to reduce them to simpler ones. These two properties of systematicity and productivity are simpler means of talking about *recursiveness*, *algorithms* and *combinatoriality*. If a system solely possesses these three properties, it is de facto a formal system in the sense of Hilbert. And as several researchers will emphasize, and as Searle (1980) and Harnad (1990a, 2017) will regularly reiterate, in the cognitive field, such a system is essentially 'syntactic'. It is indeed interpretable, but its semantics are external – hence the importance of cognitive science to add the property of interpretability.

Still, the reformulation by Fodor and Pylyshyn allows us to briefly present here two classical critiques that have been regularly addressed to computational models. Because it requires *atomicity*, *systematicity* and *productivity*, computationality cannot adequately model phenomena that are continuous and that present complex dynamicity. It was claimed that many dynamic systems may not be approached by classical atomic combinatorial sequential formal systems and Turing machines.

The length of this book does not allow us to enter such debates here. Let us simply recall that atomicity, systematicity and productivity are properties of models – and not realities as such. And these models are mediators for describing and explaining. On the basis of this, the question then arises as to whether Turing-type computation models can compute functions that contain Cantorian continuity and parallelism, as is the case for instance with formal dynamic systems. A negative answer to this question would forbid the application of computational models to dynamic and parallel systems.

It so happens that many mathematicians, such as Rice (1953), Lacombe (1955) and Grzegorczyk (1957), to name but a few, have proven that analogue machines could also *compute* functions applied to continuity and that their computation is equivalent to a Turing computation. A similar type of argument concerns the sequentiality of Turing computation. On this topic, Gandy (1988: 33) has shown that computation can be parallel – that is, that there are machines whose computation processes involve parallel changes in many overlapping parts.[8] Siegelman (1995) has claimed that the class of Turing's original physical machine 'TM' is just a subclass of the many other computing machines. In other words, formal symbolic systems that model complex dynamic and parallel systems can be computed and therefore presented in algorithms.

But it is the problems underlying the notion of productivity that raise the most serious problems for computational models. They raise the question of non-computable functions.

Non-computability

In mathematics, the notion of productivity once more encounters the problem of the decidability of theorems for formal systems: are theorems always provable in a formal system? Or, in terms of computation: are all functions calculable? Could it be the case that a formal model contains functions that would not be computable?

Here are two examples of such non-computable functions. Our first problem is one that was described by Penrose (1989). It is not presented in formal terms or using an equation. It is more of an illustration of a situation in which we would intuitively expect there to be some functional relations underlying the question asked and where there could exist a calculable equation or algorithm for finding the solution. Here is the problem: imagine that to cover the surface of a rectangle, we have to use atypically shaped tiles such as those shown in Figure 9.3. What is the effective formal procedure for ensuring a perfect fit of the tiles in the rectangle? (Figure 9.3).

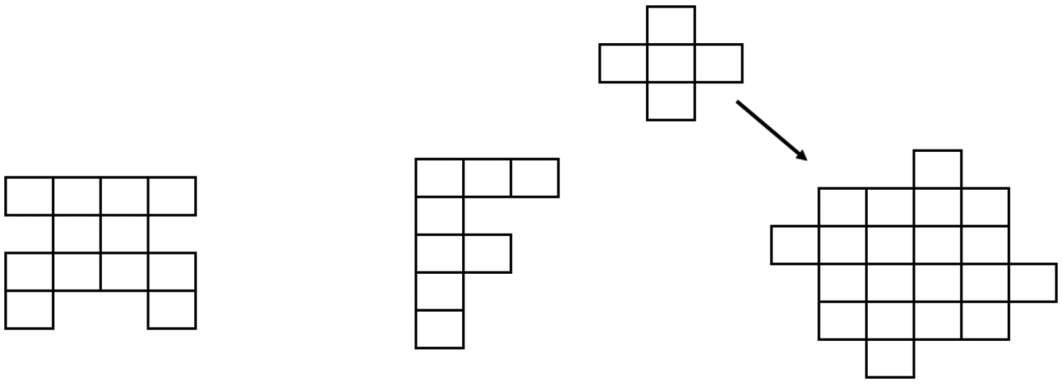

Figure 9.3 Penrose's Polyomino Tilings.

Penrose proved that no recursive function or algorithm exists to show if these shapes could cover an entire surface. In other words, it is not possible to find a calculable function of the type $f(x, y) = (x,y)$ that would enable us to compute the necessary number of tiles for covering such a surface.

The second example is a typical mathematical one. It presents itself in the form of a system of several equations all having the following form: $Ax + By = C$ where A, B and C are constants.

$3x + 7y = 1$

$x^2 - y^2 = z^3$, and x, y and z are integers

This type of system of equations is called 'diophantine'. Intuitively, just looking superficially at these algebraic equations, it appears to be easy to find a general algorithmic solution for them. But, on the contrary, Matiyasevich ([1970] 1993) and Davis, Matiyasevich and Robinson (1976) demonstrated that no such algorithm exists. In other words, although the values given to the function's variables may be atomic numerical values, such a type of system of equations is non-computable.

In mathematics, the existence of these types of non-computable functions is not a rare occurrence. 'Everyday mathematics leads us unavoidably to incomputable objects'

(Cooper 2004: 1). Many mathematicians and computer scientists have demonstrated that computable functions form only a very narrow subset of all mathematical functions. There is an infinite number of computable functions ($\aleph$), whereas non-computable functions are infinitely more numerous ($2^{\aleph}$). In other words, they are not countable.

In short, these two examples chosen among an *infinity* of other possible ones show that infinity of mathematical functions, though they may be well-formed as functions, are not calculable/computable.

Oracles

This problem of non-computability is one which Turing confronted in his doctoral thesis. He proposed an original solution for making non-computable arithmetic functions into computable ones. He appealed to 'oracles': 'With the help of the oracle we could form a new kind of machine (he called them o-machine), having as one of its fundamental processes that of solving a given number-theoretic problem' (Turing 1939: 161).

This specific solution, of the oracles, is highly interesting and important, for it allows the creation of programs that find answers to non-computable problems. Many such oracles or heuristics can be added and appended to form a complete computable program. And because of these oracles, the computation may proceed and eventually end. However, by the same token, the problem of non-computability becomes even more problematic than one would think. By appending oracles, more complexity is brought into the solution. As Chaitin, Doria and da Costa (2012: 36) put it: 'just add[ing] new axioms . . . increase[s] the complexity *H(A)* of your theory *A*!.' In other words, because oracles may entail redundancy, there is an increased risk of generating greater randomness which, in turn, may steer the systems towards an absolute probability called *Omega*, which, according to a theorem by Chaitin (1998), renders the system non-computable. Hence, although a complex static or dynamic system with discrete or continuous input may at first appear to be computable, it may paradoxically turn into a non-computable system. Consequently, the greater the complexity of that which needs to be functionally modelled, the higher the probability that it will be non-computable.

This non-computability problem is not rare in science. On the contrary, it is omnipresent. It is everywhere. 'Undecidability and incompleteness are everywhere, from mathematics to computer science, to physics, to mathematically-formulated portions of chemistry, biology, ecology, and economics' (Chaitin, Doria and da Costa 2012: xiii).

And for some contemporary computer scientists, non-computable functions are far more interesting from a theoretical standpoint than computable functions. 'The subject [of computation] is primarily about incomputable objects, not computable ones' (Soare 2009: 395).

Epistemic roles of computational models in science

Historically speaking, formal models have always been at the heart of scientific inquiries, but accompanying them with computational models is quite recent. Today, however, they

have become an important component of scientific theories. They have radically changed the way science is done. Some see their contributions by the huge number of data they are given and the speed at which they can be processed. But these features belong more to the computer technology that implements them than to the computational models themselves. But as more and more epistemologists have stressed, computational models have their specific role in science, be it epistemic, expressive or communicative.

The first role of a computational model is an epistemic one. As computational models are a particular sort of formal model, their epistemic status resembles the latter. Indeed, they have some similar cognitive *surrogate* role. Like formal models, they represent something in the sense that they are projected or mapped onto the objects to which they are applied. So, if, as computation theories have shown, algorithms are equivalent to effective procedures for calculable functions, then computational models contain algorithms for computing functions projected onto the objects studied by the sciences.

For example, in modelling a pendulum, there will be a mapping from a formal model FM that is equivalent to a computable model or algorithm AL, onto some object structure OS itself represented by some specific data. In other words, it is one of the epistemic roles of a computational model to offer a means for algorithmically expressing this functional mapping (Figure 9.4).

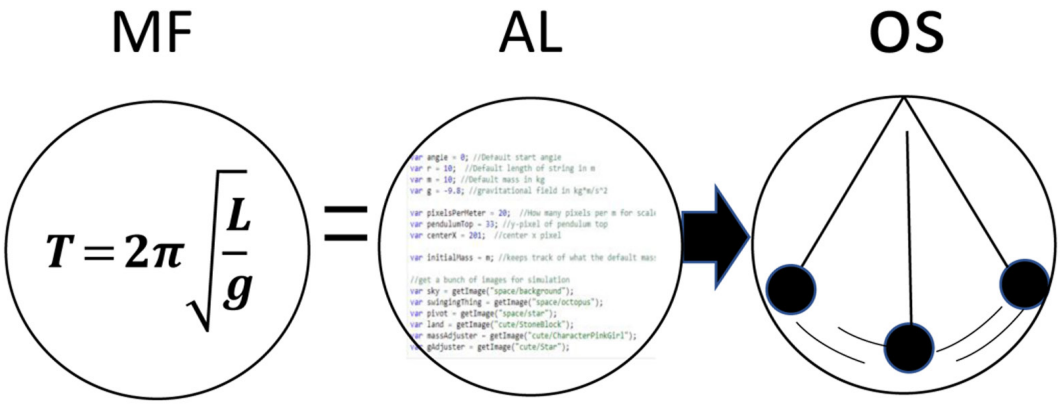

Figure 9.4 Functional mapping of a pendulum.

And as computational models use symbols and formula, they also construct their specific type of epistemic *categorization* and *reasoning* operations. We will explore these epistemic roles in view of their eventual application in semiotic enquiries that aim at integrating computation in the analysis of some semiotic artefacts.

Categorization role of a computational model

The epistemic categorization role of a computational model has a specific signature. It has its own way of constructing and expressing in a computational language (a) the object or entities that are to be submitted to and produced by a computable function, and (b) the calculable functions that underlie the algorithmic statements. That is its main purpose. In more concrete terms, it offers a model of the object and functional

relation in terms of algorithmic statements or 'programs'. A typical example would be that of a computational model which included an algorithm (of the Monte-Carlo type) that uses randomness for computing some function or algorithm to estimate variance or to decide the shortest path between many points, for instance.

But what is more interesting is that the categorization of its objects and algorithms can take two main forms. Just as a formal model may express its calculable functions in two ways, a computational model can express them intensionally or extensionally. Take the classical and simple mathematical function of squaring a number. This functional relation in a formal model is expressed intensionally by the following formula:

$$x = x \cdot x, y = x^2, \text{ or } F(x) : x^2$$

or extensionally by the list:

$$F(x, y) : (1, 1), (2, 4), (3, 9) \ldots (n, n).$$

Because this function is a computable one, by virtue of the extended Church-Turing thesis, a computational model can express it, under certain conditions, by means of different algorithmic statements or instruction. The first and most usual one is through classical intensional algorithmic formulas or through a program. The second one can express the inputs and outputs of the function through an extensional list in a database. Hence, there are two ways of expressing an identical computable function in a computational model.

But they each have a specific epistemic role. The intensional categorizing formulation of an algorithm expresses a function using abstract generalization formulas. It presents the instructions to be followed if the function is to be effectively computed by computers.

But for most humans, such formulas often appear only as sets of formal symbolic statements that are cognitively unreadable. More profoundly, it is often not sufficient for explaining or for understanding the object and the processes to which it is applied. In other words, it is not because a computational model correctly expresses computable functions that the expressed procedures are traceable (Gotel et al. 2012) and hence readable and understandable by humans. A simple proof of this is often seen in laboratories or industries, which, over the years, have developed algorithms but without providing the documentation that should have accompanied them. These programs may still be effective, but they may become opaque for later programmers and users. In such cases, the algorithms cannot easily be understood. Not every scientist or programmer is like von Neumann, who was said to have had the capacity to directly read algorithms written in a binary code.[9]

One reason for this is that intensional algorithmic expressions are expressed in a language that encodes its instructions using a vocabulary and formulas that are either close to machine coding or to some higher-level programming language. This latter language can be simple or complex in its expressive means. For instance, the same computable function can be expressed using various equivalent low-level assembly languages or in higher-level languages (Python, C++, etc.). But this does not guarantee that they will all have the same cognitive content and hence the same

level of understandability for humans. Depending on the language chosen, algorithms can either be understandable or completely opaque. And in effective and concrete processing, the instructions activated may be tractable or non-tractable. In other words, it is not because algorithms are well-formed syntactically or semantically that they are automatically cognitively accessible to humans.

It follows then that in a computational model, this type of intensional categorization may become a black box for many users. One may know what the inputs to give to an algorithm and what its expected outputs are; however, because one does not always have access to what it does or to what the computable function underlying it is, the understanding of it will be limited, if not absent.

A computational model may also categorize a computable function through some extensional presentation. This is achieved by listing the inputs and their corresponding outputs. But scientists and epistemologists do not tend to see a list as a mode by which a computable function is expressed. But as demonstrated by many computational theories, under certain strict conditions,[10] a list can indeed be an acceptable means for presenting a computable function.

Take again our squaring function $y=x^2$ or $F(x)$: x^2 where the inputs are the value of the argument x, with $f(x)$ or y representing the value of the function. It is the list of these paired values in a domain that can form an extensional presentation of the function *(1, 1) (2, 4) (3, 9). . . (n, n)*. And in computer science, this list of 'inputs' and 'outputs' can be included in a database and called a 'data set'. And hence, the list opens up a new way of understanding the categorization role of a computational model.

It follows then that computation models can express a calculable functional relation extensionally. The data are its input and outputs. In more concrete terms, it is possible to think of data sets as an extensional definition of an algorithm. In Table 9.1, our preceding example of the squaring function is expressed using a simple data frame. The following data set could be seen as hiding an underlying computable function.

Table 9.1 Extensional Definition of a Function in a Frame-like Presentation

Squaring function	**1**	**2**	**3**	
1	1	-	-	
2	-	4		
3	-	-	9	
…	-	-	-	
10	-	-		100

In science, the notion of data is not often seen through such a perspective. Data receive a variety of definitions. For instance, in computer science, the term 'data' may refer to the inputs and outputs given to or produced by a computer and inscribed in a database. In mathematics, the term refers more specifically to a set of numerical values of functions over mathematical entities. In statistics, it may refer to the dependent or independent values of a formal model applied to experiments. In philosophy, data are interpreted as being the result of some knowledge processes such as perception, observation and experimentation. In the theoretical literature, all three meanings are

often conflated. The data found in a database are often only seen as a set of 'numerical values' collected through some observation or situation. And they do not seem to be functionally related. What is important is that they are well stored, searchable, visualizable and managed through some relational database management system (RDBMS). It was the intuition of Codd (1983),[11] creator of the relational database, that data sets could be organized as relational sets. Hence, a data set could be an extensional definition of relations between objects and their attributes or properties. But which relations are to be implemented is determined by the creator of the database.

Some other researchers, mainly from the fields of machine learning and data mining, saw these data sets in a very heuristic way. For instance, T. M. Mitchell (1997) and many others saw data sets as possible instantiations of functional relations to be discovered. A data set would therefore be the result of underlying functional regularities that could be submitted to 'algorithms that can learn regularities in rich, mixed-media data' (Mitchell 1999: 35).

To illustrate this intuition, we may consider a simple experiment: the movement of various sorts of pendulums producing numerous data related to a multiplicity of different features. Some are important, other less so. For instance, the data set could be: the numerical value associated to the length of the rod, the oscillation period, the location, the weight of the ball, its gravity, its price, its level of noise, the temperature, the surrounding atmospheric pressure and so forth. All these data represent features of the pendulum. All may be put into relation, but only a few of them are related by a computable functional dependency. In a computational model, it could be the role of a machine-learning algorithm to discover or approximate which of these features bear computable functional relations. For example, the algorithm could discover that the length L, the gravity g and the time T are data that could be intensionally expressed by means of a classical algebraic equation:

$$T = 2\pi\sqrt{\frac{L}{g}}$$

In this example, we can see that in a computational model, data pertaining to some object can be presented either extensionally or intensionally. Extensionally, the data are presented through some ordered list. Intensionally, the data are implicitly embedded in some algorithm which is equivalent to a calculable function. This dual way of presenting data directly regulates the type of reasoning processes that can be applied to them. They could be either deductive or inductive, but abductive reasoning may also be called upon.

Reasoning role of computational models

Just as with the conceptual and formal models, reasoning also has an important epistemic role in computational models. But here, the reasoning process is deeply related to the form of the data on which it will be applied – that is, to their intensional form or extensional form. In both cases, reasoning is a rule-governed inferential process that is applied to the data.

From a cognitive standpoint, computational modelling will apply all three of the main types of reasoning processes we have seen in the conceptual and formal model: inductive, deductive and abductive. These three types of reasoning have become the core of reasoning strategies in various computer sciences that came to deal with artefacts that were carriers of meaning.

Top-down deductive reasoning

In early artificial intelligence, the preferred type of model was the computational deductive model. It was often said to represent a *top-down* manner of reasoning. Its general architecture was usually applied to some general knowledge data statements from which, by an inference based on existential and instantiation rules, it will deduce some particular knowledge.

In the knowledge-representation (KR) paradigms of AI systems, this top-down approach has been presented through various semi-formal or formal means, schemas and frames, which became knowledge levels, knowledge spaces, knowledge bases, folksonomies, ontologies, knowledge graphs, etc. These KRs are not conceptual models as such, but rather computational expressions of part of the *conceptualization of the domain under study*. In other words, in a computational knowledge-representation paradigm, knowledge is expressed through an abstract and formal but specific conceptualization of a domain. This is well expressed by Gruber's classical definition of ontologies: 'An ontology is an explicit specification of *a conceptualization*' (Gruber 1993b: 199, emphasis added).

Let's consider some classic examples of these computational models called 'frames'. These types of models are usually grounded in some natural language sentences of a conceptual model: for instance, *All the cities of England have a mayor*. The proposition underlying this English sentence could also have been expressed in another natural language, for instance, French: *Toutes les villes d'Angleterre ont un maire*. The formal model in turn translates not the sentences itself but the general propositions underlying them. So, it will be expressed directly through a logical formula that dissimulates the general quantification hidden in the English sentence while explicit such as ∀x *City (x)* ⇒ *HAS MAYOR (x)*. In a Minsky frame type KR, the computational model will embed these statements in a sort of database and it can express extensionally some of the knowledge pertaining to the mayorship of England's cities (Figure 9.5).

Mayored cities of England	**Mayor name**	**Mayor employee number**
London	Johnny Walker	WalkJ8594
Oxford	Elizabeth Arden	ArdenW3928
Cambridge	Granny Smith	SmithB3401
….		
City n	Name x	Number x

Figure 9.5 Examples of Minsky type frames.

The general statement of this frame does not take the form of a propositional logical formula but it implicitly asserts that each city in England has the feature '*has a mayor*' who, in turn, '*has a name*' and '*has an employee number*'. Naturally, many other features and sub-features could be added.

And onto this knowledge-representation format, some inferential types of algorithms can be applied (e.g. 'if, then'). So, when a new input is entered, such as *London is a city*, then the system will 'deduce' that if *this city is in England then it has a mayor* and also that *it has a mayor whose name is* '*Johnny Walker*'.

This type of top-down algorithmic reasoning process is at the core of knowledge-based systems[12] or rule-based systems.[13] The main architecture of these systems includes (a) a database where general or individual knowledge is stored and retrieved; and (b) a variety of inferential engines applied to the database.

In the earlier years of AI, many variants of these frames were wrapped up into computer applications. The prototypes of these systems took the form of expert systems. Their architecture included a knowledge base and an inference engine containing a set of top-down inference rules that implemented a forward-chaining process where the general antecedent allowed the deduction of the instantiated consequent. These systems often also included some other types of backward-reasoning processes that went from the consequent to the antecedent. But these belonged more to the inductive and abductive modes of reasoning. Later on, these frames were transformed into more sophisticated types of knowledge representation, with important types being ontologies and knowledge graphs.[14]

Today, the semantic web paradigm can be seen as a knowledge base system where the internet serves as a knowledge source. The difference in the processing is that the knowledge data (textual or iconic) must be transformed into a predicative general form called the Resource Description Framework (RDF) so that it may be used by inferential rules.

Bottom-up inductive reasoning processes

The top-down reasoning process has always been seen as the essential feature of an artificially intelligent process. The problem with this type of reasoning, as has repeatedly been stressed, lies in the acquisition of this knowledge. Where does it come from? Should it always be given by some external agent or could it not be learned by the system? This is where inductive and abductive reasoning steps in.

In computational models, this second type of reasoning process is often called a *bottom-up* algorithmic process. Its architecture is opposite to that of the top-down approach. It usually contains a set of data which can be extracted or discovered by some algorithmic procedures over some type or another of general knowledge.

Strictly speaking, such generalization reasoning is inductive if and only if the data are exhaustively given or asserted. For instance, only if there is an exhaustive data list in which each and every city of England is associated with an individual mayor may we conclude that '*All the cities of England have a mayor*'. This inductive example can be expressed using the following logical formula: (CITY (London) & has MAYOR (London)) & (CITY (Birmingham) & has MAYOR (Birmingham)) . . . (CITY (x_i) & MAYOR (x_i)) then if $\forall x$ (CITY (x) $\Rightarrow$ HAS MAYOR (x)).

The general knowledge that '*All the cities of England have a mayor*' has been produced by a generalization inference rule called *universal instantiation*. It allows a conclusion to be considered true until a counter example is given.

In computer science, these inductive reasoning procedures or bottom-up procedures are now often called *data-driven algorithms* or *data mining analytics*. This algorithmic architecture is one where the inputs are a set of structured data to which are applied various types of algorithms that transform, aggregate and classify these data to produce an output that is a sort of general formula, if not the best approximation of a universal formula.

In a conceptual model, this reasoning process is seen as identifying regularities, patterns, prototypes and even 'laws' governing the data that describe the features and relations of the state of affairs under study.

In a formal model, these general formulas are expressed using intensional synthetic formulas. Recall here the list of values of our squaring function. An inductive reasoning process would aim at synthesizing the following extensional list of functional relations: (1, 1), (2, 4), (3, 9), . . . (*n*, *n*) by the formula $F(x) = x^2$.

This inductive process is usually easy to achieve if the data are finite, not too sparse and do not contain too many different types of variables. But often, the object under study will be complex and present a myriad of micro-features and numerous types of static and dynamic relational structures. Finding the patterns that characterize data and expressing them through some sort of general formula is quite a challenge. It is one of the most important problems that contemporary computational science has to deal with. Still, most actual practices continue to use inductive reasoning processes, but as the object they study is often presented to them through a huge amount of data they are increasingly pressed to explore efficient inductive reasoning procedures supported by data-driven algorithms.

Many computational models have proposed algorithms to process these big data in this inductive way. The algorithms they offer are often metaphorically called 'machine learning',[15] 'deep learning'[16] or, prosaically, 'pattern recognition'[17] or 'inductive programming logic'.[18] These types of algorithms have become the contemporary stars of artificial intelligence research, where 'intelligence' is mainly understood in terms of inductive learning procedures.

These data are not processed in the form in which they come. They do not jump into algorithms for the simple reason that they are data. Whatever their source or their type, they have to be submitted to many complex operations, sometimes manual, sometimes algorithmic, in order to (a) become inputs for these machine-learning algorithms, (b) be effectively processed by these algorithms, and, finally, (c) be evaluated when applied to concrete cases.

In the first moment of this complex procedure, the data must be acquired – a task that is not without complex problems, for their sources are often numerous, anonymous, noisy, non-controlled and of various types. Afterwards, they must be prepared to be made into admissible arguments or inputs for these algorithms. They must hence be cleaned, selected, categorized, annotated, refined, standardized, validated, verified, updated and so on. Afterwards still, they have to be machine-encoded, formatted and normalized. In other words, data are not 'given'; they are worked on. Despite all these

heavy preprocesses and processes, machine-learning type of computational models have the wind in their sails. Some even believe that theory-driven research has been killed! As Anderson puts it, 'The data deluge makes the scientific method obsolete' (2008).[19]

In the second stage of these inductive reasoning procedures, the data have to be processed by some effective algorithms. But these algorithms come in many shades. They are not a 'one fit for all'. A relevance analysis must be made. Many are rooted in different mathematical models and expressed in different languages. So, we find underlying them various formal mathematical models belonging, for instance, to inductive probabilistic statistics, mathematical classification, statistical optimization, inductive logic, dynamic combinatorics, dissipative dynamics or formal systems. We also see that these use many formal languages such as predicate logic, directed graphs, differential geometry, linear and non-linear algebra, topology and so on. All these formal models and languages possess their particular conditions of use and parameters. They are not easily interchangeable. Choosing one or another is often quite a challenge in itself.

Finally, these algorithms have been inspired by various but specific scientific theories: cybernetics,[20] vision and behaviour psychology,[21] neurobiology,[22] insect or swarm intelligence (ants, termites), stigmergy,[23] genetics and evolution,[24] immunology,[25] artificial neural nets, connectionism,[26] neurophenomenology,[27] situated philosophy of mind,[28] catastrophe semiotics,[29] radical material epistemology[30] and so on. Rapidly summarized, these various inductive reasoning algorithms are understood as types of computable classification and optimization functions that have for arguments some data values and deliver as outputs some patterns in the data. They are often interpreted in cognitive terms and are said to *recognize*, *learn*, *adapt*, *discover* and *evolve*.

Many sub-procedures have been explored to create flexible but robust learning. Two important ones among these are the 'supervised' and 'non-supervised' ones. The supervised learning procedure is one that is helped by some training. It uses a subset of well-structured data that serves as a prototype. What is learned in this subset is then applied to the whole set in view of discovering the most similar patterns. The second procedure is called 'unsupervised' learning. Without the help of prototypes, the procedure clusters with the help of complex parameters the data that share some common features. These two strategies are often seen as 'superficial' learning processes. So, some *deeper* and more complex learning procedures were proposed. These deep learning procedures usually start by building some interrelated classes of elements that share either some common data features or some interrelated classes of features. Afterwards, these classes are themselves reclassified by means of multi-level strategies so as to build new levels of interrelated classes. And after some iterations and more detailed readjustments, they come to 'recognize' more distinguished, solid, optimized or most probable patterns in the data.

During the last stage in the application of these inductive algorithmic procedures, some evaluations and interpretations must be performed. Big data are not simple and unproblematic artefacts. They are not without their own epistemological problems. One typical problem is the adequacy of the algorithms with respect to the specific objects

applied. In laboratories, these algorithms will be tested against accepted benchmarks. But in many concrete applications, no such benchmarks exist, so great confidence must be placed in the algorithms themselves. Finally, as theories, data are often biased, noisy, situated and institutionalized, the results may also be oriented in different directions. This entails quite a challenge for their interpretation and validity. As Gould eloquently said: 'inanimate data can never speak for themselves, and we always bring to bear some conceptual framework, either intuitive and ill-formed, or tightly and formally structured, to the task of investigation, analysis, and interpretation' (Gould 1981: 166).

The third type of reasoning processes is, as expected, the abductive one. In computational models, this reasoning process will be most useful. For example, recalling our cookie example, it may happen that an AI system is given in its knowledge base that

All cookies in the box are chocolate cookies.

But it may also receive the data:

There are chocolate cookies.

If the system is to reason correctly, it should not infer from the first general statement *that*

These cookies are from the box.

Rather, correct reasoning should only produce the hypothesis that

Possibly *these cookies are from the box.*

This is because it could be the case that these cookies come from somewhere else.

Because of the heuristic role of this type of reasoning, computational models have been increasingly exploring it as it seems quite natural. It has become an important dimension of many computational models, mainly in artificial intelligence with regard to *algorithmic learning*. It may be used for problem-solving (Kakas et al.), discourse interpretation (Hobbs et al. 1993), knowledge discovery (Zhou 2019) and programming languages (Console and Saitta 2000). It is currently becoming more and more important in machine learning (Levesque 1989; Crowder, Carbone and Friess 2020; Zhou 2019; Dai et al. 2019).

Although abductive reasoning is rich in promise, its transformation into algorithms must still undergo more research. The challenge is in either making probabilistic adjustments, building adaptation strategies or calling upon structures of a priori knowledge bases for validating the hypothetical conclusions. There is an increasing number of such algorithms. Some are 'one-shot',[31] 'few shots',[32] 'prototypical',[33] 'hybrid neural' and 'symbolic' learning machines.[34]

We hope to have shown some of the complex epistemic roles of computation models. They are a subtle but sophisticated play between different types of algorithms

that categorize the object under study and algorithms that allow reasoning on what has been categorized. The three types of reasoning processes we have presented define the epistemic role of computational models. And they always activate a dialectic relation between general formula, rules or laws and individual data. In certain cases, if the pattern is known and well expressed, then the deductive process will search some data for a proof or disproof. In some circumscribed situations, if all the data are available, then an inductive or at least a best approximation of an inductive process will search namely for a generalization or for rules. Finally, in cases where just a few data are known, an abductive reasoning process will be called upon. It will be the most complex one.

The expressive and communicative role of computational models

Expressing and communicating the data, the algorithms and their results is achieved through specific notational forms in computational models.

First, as we know, computers do not manipulate the data directly in the form in which they are presented to humans (natural language expressions, scripts, databases, digits, etc.). They process encoded data.[35] So, whatever their source, they have to be digitized so that they may serve as inputs to algorithms. Second, the algorithms themselves are expressed in some formal computer language. These languages have specific syntaxes and semantics, and each programming language is specialized in generating instructions for the manipulation of specific types of data and operations in some domain. This gives them their expressive power. Some are very close to a basic, low-level computer language (assembler). Some others are geared towards databases (Cobol, Perl), towards the manipulation of mathematical functions (C++, LISP, MATLAB) and others towards natural language (Python, etc.).

All these types of languages and the programmes they express may be so well mastered by the researchers that they become the 'natural' way by which the various components and operations of a computational model are communicated. It is common among computer scientists to talk about an algorithm in terms of a 'formal model' whose packages of algorithms are named according to the actions they carry out. Here, one 'googlelizes'. There, one 'topic modelizes'. Others perform 'data mining' or 'deep learning'. Some even have a proper name: the Monte Carlo algorithm. They become the 'specialized communication languages'. A new modern *lingua franca characteristica* is being born!

For our research purposes here, one important point to emphasize is the expressive and communicative role that pertains to the presentation of the results of the algorithms. This is quite different from what may be encountered in conceptual and formal models where natural and formal languages dominate. Here, another type of language will often be called upon and is used as a bridge towards the conceptual model.

When various complex algorithms are applied to huge data, the presentation of the processing and of the results obtained receive special attention in computational models. Because they are not so transparent and evident and often require a heavy cognitive investment in order to be understood, some sophisticated visualization techniques offer more effective means for expressing and communicating their

processes and, most importantly, their results. These visualization techniques became rapidly welcomed adjuvants and mediators for understanding algorithmic data categorization reasoning processes and their results.

Here is a simple example of such a visualization technique. During the Covid-19 epidemic, various experts used many formal equations for describing and analysing several of its dynamics. The equations underlying the algorithm express some functional dependency relations between several variables such as *time*, *infected population*, *recovered population*, *death*, *age*, *co-morbidity*, *geographical location* and so on. Some of these equations and big data may be cognitively easy to understand for a trained mathematician. But even then, for many expert communities and the general public, due to the complexity of the functions, they are not immediately or intuitively understandable. And as more and more variables and parameters are added, the understanding of what is computed and of what the result is becomes increasingly difficult. So, visualization provides some assistance, leading to the generation of insight.

The simple visualization in Figure 9.6 – a classical Cartesian graph – presents the dynamic results which may be produced by the application of computational models to three equations presenting the variance over time of the percentage of recovered population *r(t)*, infected population *i(t)* and of susceptible individuals who could get Covid-19 *s(t)* (Figure 9.6).

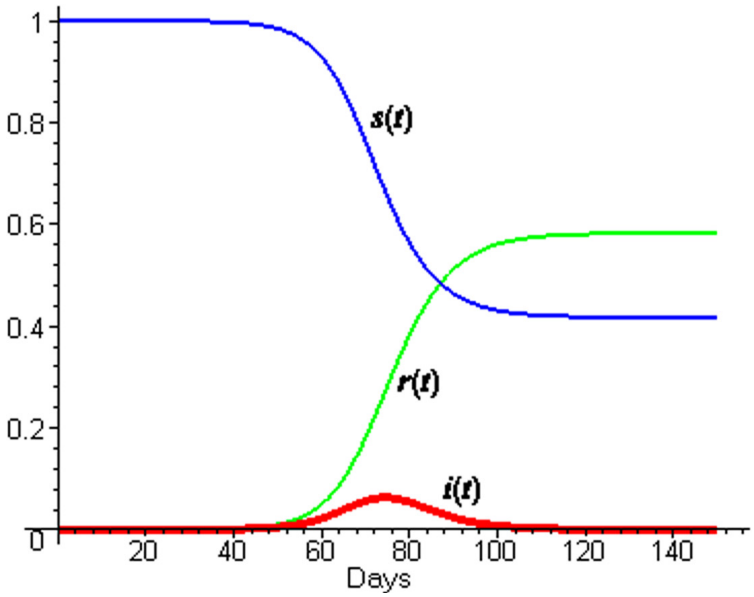

Figure 9.6 Visualization of three equations describing three dimensions of Covid-19.

Without knowing the exact equations and data that correspond to the visualization of the curves, it is possible to achieve some inductive, deductive and abductive reasoning from this graph. For instance, in this formal model (the SIR model), it can be seen by the *r(t)* curves compared to the other ones that 'a low peak level can lead to more than half the population getting sick' (Smith & Moore 2004). The conclusion is reached by the reader without a knowledge of the mathematical models that underlie the Cartesian graph and its curves.

At first sight, this iconic language seems simple. But technically, it is a translation of a specific set of results of computed equations into an iconic form. It is a sort of extensional presentation of the functions expressed through some equations and algorithms. The drawing corresponds to specific symbols of these equations. But one important point must be noted. The drawing is in juxtaposition to the curves representing the dynamics of three equations. So for the visualization to be understood by someone (an expert or a layperson), some parallel information processing is required. And that which is iconic becomes analogical. It is no longer digital. And it is the specific parallelism allowed by this iconic analogical language that gives the visualization its apparent simplicity and epistemic robustness, a rich cognitive value and a heuristic function. This is quite difficult to achieve with the traditional sequential reading of the equations.

Depending on the nature of the domain studied, on the formal models used and on the computational format taken, the visualization techniques may be static or dynamic, and they may take various forms: interactive tables, charts, graphs, maps and so on. They can use various metaphoric forms, such as buildings, games, roads, spaces, spots, pies, mountains and textures (see Chen 2013 for a rich set of examples; and Dondero and Fontanille 2014 for a detailed semiotic analysis).

The visualization techniques[36] enhance the epistemic role of both formal and computational models. They transform underlying formal and algorithmic models into various iconic forms (annotated with natural language). This is evident in Figure 9.7.

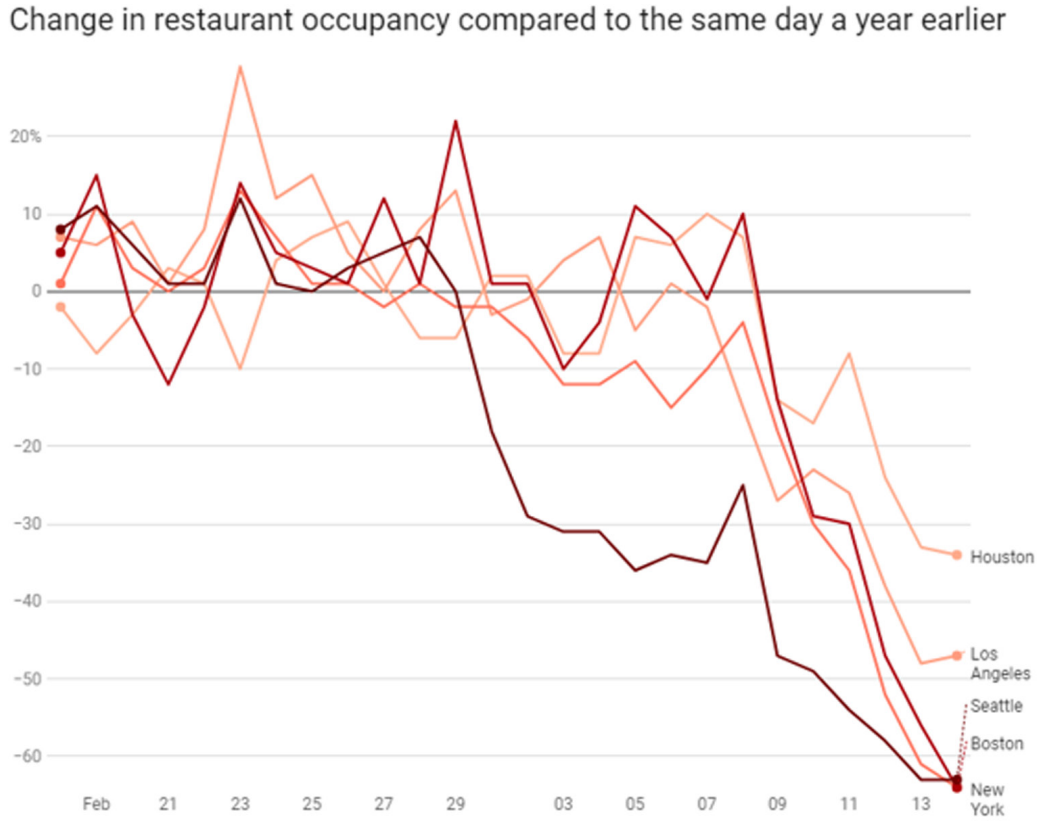

Figure 9.7 How Coronavirus is Devastating the Restaurant Business. Table cited in *Chart: How coronavirus is devastating the restaurant business*, by Rani Molla@ranimolla Mar 16, 2020, 12:20 p.m. EDT https://www.vox.com/recode/2020/3/16/21181556/coronavirus-chart-restaurant-business-local (open source).

Visualization techniques of the kind used in Figure 9.7 manipulate data not only to provide descriptive knowledge but also to create explanatory knowledge. In science, data visualization has become a full subdomain of computer science in itself. However, the challenge it entails is great, for it requires complex interpretative strategies which are often theory-laden or culturally grounded. And to be valid, it must be related to its underlying formal models; otherwise, they will be used as black boxes about which anything may be said.

Conclusion

Because computational models have a specific structure, they play a particular epistemic role in science. Indeed, they allow many sciences to transform the numerous complex calculations required by their formal models into effective procedures. And it is only if these effective procedures are formally formulated in an algorithm and eventually in a programming language that computer technology can effectively and concretely compute them. It then follows from these definitions that computation is per se different from computer technology. A computer is but a physical electronic means by which computation can be realized or *implemented*, and a computer can only compute electronically if it is given a set of *effective procedures*, one of which is algorithms or, in a more common manner of speech, *programs*.

The distinctions will become important in the encounter of semiotics with the digital: the real challenge of semiotics is not the *digital* (which is only a useful encoding procedure), but its embeddedness in computationality. This raises an important question: Can semiotics be computational? Or, can computational semiotics exist?