The Open Handbook of Linguistic Data Management

Edited By: Andrea L. Berez-Kroeker, Bradley McDonnell, Eve Koller, Lauren B. Collister
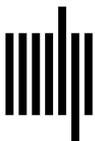
The MIT Press

29 Managing Computational Data for Models of Language Acquisition and Change

Matthew Lou-Magnuson and Luca Onnis

1 Introduction: Outline of work presented

The goal of this data management use case is two-fold: first, to illuminate how computational modeling works to augment traditional scientific methods and, second, to suggest ways in which computational models exist as a kind of data in and of themselves, and as such need to be properly archived.

The chapter begins with some background information on two specific models that form the object of the discussion. Next a brief overview of the process of computational modeling is presented and followed by some theoretical perspective on how models present unique challenges as data for archiving. In the longest portion of the chapter, these points are examined stage-by-stage in a case study, highlighting the decision process and ideals for data management involved. Finally, a unifying ideal for archiving computational models that generalizes the details of the case study is given.

1.1 Case studies presented

In this chapter, two computational models of language evolution are discussed. The term language evolution is used generally to refer to the biological evolution of the faculties necessary for human language, as well as traditional work on language change taken at a broader, systemic level; the models presented here are of the latter category. In particular, the models we discuss attempt to capture the interaction between the development of morphological complexity[1] and the structure of human social networks, but differ in the level of granularity at which they formalize the problem. Lou-Magnuson and Onnis (2018) developed a so-called *high-level model* of the fundamental conditions for morphological complexity to emerge; it seeks to answer the question: does a particular social network structure support the capacity

for sustained development of morphological complexity? The second, *low-level model*, presented in Lou-Magnuson (2018), builds on the results of the prior high-level model, but at a more fine-grained level of computational detail. It seeks to answer the question of observable language change: given the capacity for morphological complexity, as predicted by the low-level model, does more complex morphology actually emerge? It is beyond the scope of this chapter to explain in detail both models and the phenomena they capture; however, some further explanation in broad strokes is provided.

Both computational models share an identical framework for studying language evolution: computational agents are created and embedded in a social network that constrains who is allowed to communicate with whom. Each agent is able to produce expressions that carry some phonological and semantic form, as well as learn how to produce these expressions from usage examples. The agents are allowed to communicate with each other for a set period of time, after which each is replaced by a new agent that learns de novo how to produce expressions solely from the examples heard by the agent it replaces. This process of intergenerational transfer is allowed to repeat over and over again, and changes in the structure of expressions produced by the agents are analyzed.

The distinctions between the high-level and low-level models are in the level of detail in the expressions produced by the agents and the learning mechanisms involved. In the high-level model, the expressions are holistic collections of values (one representing a meaning, and few others the phonological content) that the agents pass between each other. Learning amounts to choosing a set of expressions (one for each meaning) to use in the next generation, potentially with some minor alterations to the values. The details are presented in Lou-Magnuson and Onnis (2018), but, conceptually,

language for the agents in the high-level model is like a deck of playing cards. To communicate, they exchange these cards among themselves, and each generation selects a playing card for each meaning, occasionally making a small change, such as turning a nine of clubs into a ten of clubs.

In contrast, each linguistic expression in the low-level model is a pair of two compositionally complex representations: a tree structure encoding the meaning, and a string of phonological symbols encoding the utterance that represents this meaning. In this model, the agents use an information theoretic method to learn a context-free rewrite system (such as a context-free grammar, but defined over these meaning and phonological structures) that they use to produce their own expressions and parse the expressions of other agents. Similarly, language in the low-level model is conceptually a modern natural language processing system on a smaller scale (see details in Lou-Magnuson 2018:chapter 3).

To investigate whether social structure potentially affects language change, the two models discussed above were run over thousands of generations on different social network topologies. Each topology captured a different social dynamic, for example, a society of intimates or a society with deep hierarchical leveling, that has been suggested in the literature to affect the morphological evolution of the language used in such communities over time. The high-level model provided insight into how these different structures affected the potential for morphological change in a given network; however, as the expressions produced and received by the agents were indivisible with respect to traditional linguistic units of analysis (i.e., sememes and morphemes), the model was only suggestive of the relationship between social structure and morphological structure. In the low-level model, though, the expressions exchanged by agents were at a level of detail such that actual grammars could be learned by the agents and the typology of the language quantitatively assessed.

The models represent computer simulations at two contrasting scales and scopes of development, each highlighting different challenges for data management. The high-level model is relatively lightweight in computational assumptions, implementation in code, and size of data outputs. In contrast, by attempting a more granular examination, the low-level model requires many more assumptions to be made, traditional software engineering

practices to manage the code base, and data outputs that are difficult to store and analyze on a single, conventional workstation. These differences of scale allow for a comparison of data management principles that may not generally arise in lab-based human studies or the collection of language data in the field.

2 Modeling social science

When compared to work done with human participants, computational modeling not only shares a number of data management concerns, but also presents several additional challenges stemming from the implementation of the model in code. These more modeling-specific challenges will be the primary focus. For example, the general principles of the data life cycle as discussed in Mattern (chapter 5, this volume) will not be presented again, but rather we will discuss how those concerns translate to the management of a code base as opposed to collections of language use. Again, the data being managed here is not the output of the models, but the models and code in which the models are written.

Of particular importance for data management in the context of computational modeling is understanding how computational work relates to the traditional scientific method as practiced in the social sciences (see figure 29.1[2]). Before we examine this process with respect to the high- and low-level models, we first provide a brief summary of each stage. After this process is presented, we also touch on how it creates new data management concerns in terms of preserving the model code, as well as documenting the stages and rationale of model creation.

2.1 Model
In this stage the task is to translate the complexity of the real world, into a set of simplifying assumptions. For example, the model of Newtonian kinematics takes the complexity of physical objects and motion and abstracts the world into Euclidean space—just three dimensions and time. This is not a statement that properties such as shape or volume are not important aspects of physical reality, but that for determining where objects are and where they are going, these details are less relevant, and add more difficulty than clarity. The goal of modeling is to replace a system that is too complex to readily understand with a simpler one that is able to be analyzed and yield insights. The goal of modeling is not to replicate
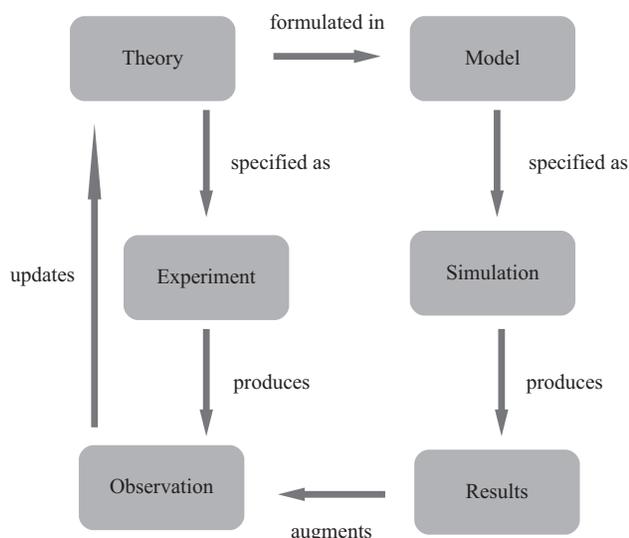
Figure 29.1
Parallels between computational modeling and the scientific method. A simplified version of the traditional empirical research cycle (theory, experiment, observation) is presented on the loop to the left, while its complementary stages of computational modeling (model, simulation, results) are presented on the outer loop to the right.

reality closely, for if such a feat is even possible, then one complex system has just replaced another.

In the realm of the social sciences, including linguistics, the models built must not only consider how to model the environment, but also abstract human beings into artificial agents. This often takes the form of reducing human behavior down to the cognitive processes relevant for the tasks of study. For example, Lou-Magnuson (2018) modeled language acquisition in agents as the ability to perform pattern abstraction and categorization. In addition to the interactions between environment and agent, models of language evolution must also consider a third artifact, language itself. There is no one best answer to this or any of the modeling questions, save perhaps simply the heuristic of how the modeling choices trim reality down to the core elements relevant for study.

2.2 Simulation

Having formulated a mathematical model, in other words, the equations and algorithms, the next step is to translate it into an explicit computable form, a piece of software written in one or more computer programming languages. At this stage, two competing aspects are in play: engineering and design. Typically, the process of

formulating a model allows for the design of experiments that would not be possible in reality. For example, having replaced living human beings with computational agents, one can simulate hundreds or thousands of generations, recording exactly the languages and learning processes of each. However, even if such a scenario is definable using the model, there are limits on what can be actually run on modern computer hardware. Thus, there is a trade-off between the limits of the engineering ingenuity in designing the software and the limits of what can be realistically explored in the possible space of the model. A practical aspect to consider is whether the desktop or laptop computer available to the researcher is sufficiently powerful, or whether a dedicated machine needs to be purchased or high-performance computing services rented on the cloud.

2.3 Results

The data recorded from simulations are generally analogous to those recorded in conventional experimental settings, and likewise, modern statistical techniques are used to analyze digital experiments. However, with simulations involving agents, and especially language, it is usually not clear how the effects of the model map the effects in real life. For example, is the amount of language change seen in a generation of digital agents comparable to the linguistic change seen in a generation of human beings, or is the space of meanings expressible by the language model in the simulation of the same scope as that of a natural human language?

These questions are, more likely than not, empirically unanswerable. Yet, in much the same way that the goal of defining a model was not to reproduce the complexities and nuances of reality, the goal of social science simulation is not to reproduce the same metrics as reality. The ideal result of traditional experiment analysis is to discover whether a given manipulation produced a statistically significant difference with meaningful effect size. The ideal result in the analysis of a simulation is to discover whether a statistically significant relationship exists between variables in the model, and perhaps even more important, new patterns of emergent behavior. These two things provide insight into the general mechanisms that drive the observations of reality and past experiments and suggest future areas of exploration that otherwise may have been unclear or inconceivable.

3   Implications for data management

In discussing lab-based psycholinguistics research versus computational modeling work, Dijkstra and De Smedt (1996) summarize the differences in terms of information specification. They invoke the notion of information processing at three levels of detail, as defined by Marr (1982) in organizing his pioneering work on the human visual system: *computation*, *algorithm*, and *implementation*. Briefly, the computational level is the specification of the inputs and outputs of a system to be studied—it defines the representations and transformations. The algorithmic level defines the procedures involved in making those transformations, and finally, the implementation level defines the physical system in which these algorithms are performed.

To better understand these levels, let us imagine a hypothetical understanding of human object detection. At the computational level, we might define the inputs as being visual information originating in the eye and the outputs being mental representations of the object. The transformation process is one of refinement from a set of physical stimuli to one or more internal, cognitively useful representations.

At the algorithmic level of understanding, let us posit two possibilities. First, that visual information is passed through sequential layers of feature detectors (as happens in actual human vision) eventually leading to the production of object-level representations. Second, we could imagine a vast store of previous visual experiences and associated objects, and that a distance is computed between new inputs and past experiences, and the closest memory pair is chosen. That is, we posit either that visual inputs are progressively refined until the final mental representation has been built from these refinements, or alternatively, that visual information is compared directly against past experiences and the closest mental representation is retrieved from memory in a one-off match.

At the implementation level, either algorithm could be posed as a task to a feed-forward neural network architecture (mimicking human biology). The first, progressive algorithm could be implemented as a so-called *deep neural network*, where layers of computation are stacked one after the other until the final level produces the desired representation. The second, one-shot algorithm could be a *shallow network*, wherein a direct mapping from visual input to mental representation is learned. Alternatively, the first algorithm could be more efficiently computed in less biologically plausible, convolutional neural network architecture, as is commonly done in computer vision and natural language processing work (Bengio et al. 2003; Krizhevsky, Sutskever, & Hinton 2012). Likewise, the second could be computed by storing the images and labels pairs, use Euclidean distance to find the five closest pairs in memory, and let the majority label be the chosen output—a procedure called *k*-nearest neighbors in the machine learning literature (Bishop 2006). There are multiple alternatives for specifying the mechanism by which these algorithms are executed that appeal to different practical concerns, such as biological mimicry and computational efficiency, and the designer of the model may have specific assumptions to prefer one or the other: the goals of a biologist doing theoretical research versus those of an applied computer vision engineer.

Dijkstra and De Smedt (1996) argue that, in general, traditional social scientists design human experiments working almost exclusively at the highest, computational level. They observe and reason about stimuli and responses and suggest mappings that exist between them. Furthermore, Dijkstra and De Smedt argue that this has been such a successful endeavor because the lower levels of analysis are essentially fixed: the algorithmic and implementation details are supplied naturally by the human brain and its neural architecture. In so far as the brains of participants are comparable to one another, these details need not be understood fully by social scientists before devising and testing hypotheses defined at the computational level alone.

It is here that we see that stark contrast and need for additional data management by those doing computational modeling. To produce a runnable simulation, one cannot neglect any of Marr's levels of understanding. A computational modeler must explicitly set forth the process that will transform the inputs into the outputs and physically implement it in some concrete system. As the example illustrated, there may be more than one viable process at the algorithmic level and usually many more options at the implementation level. The motivations behind these choices are far from arbitrary and usually are borne out from the researcher's assumptions and goals.

In terms of the model as an object of data management, computational modeling poses at least two novel

requirements on the researcher beyond the need to redundantly and safely store the simulation outputs. First, they must track the software that was engineered to implement the model and run the simulations. That is, a computational model requires preserving the technical means by which to rerun and verify the software itself. Second, they must record the decisions made to define the algorithmic and implementation levels of the model. The level of detail, both conceptual and technological is non-trivial and can even affect the simulation outcomes in unforeseeable ways. Revisiting the object-detection example, while the shallow neural network and *k*-means implementations can ostensibly execute the same algorithm, they interject different biases: the neural network is heavily dependent on the data used to train it, whereas the *k*-means results can be drastically altered by initial random conditions (Bishop 2006).

The necessity to account for these two additional factors largely stems from the increasing need for reproducibility in the social sciences, as discussed by Gawne and Styles (chapter 2, this volume) in regard to experimental work and more generally in linguistics by Berez-Kroeker et al. (chapter 1, this volume). For simulations to provide valuable insights, it is not enough that the output results are properly archived so that the findings can be verified. As discussed in section 2, the metrics are generally not directly comparable to human measures, and with the ability to arbitrarily increase sample sizes with more simulation runs, eventually some significant findings will emerge. What is important are the patterns of behavior that emerge in simulations and their impact on interpreting traditional lab-based work. In the absence of the human brain providing a common ground, the decisions involved in specifying Marr's levels of the model and in engineering the software itself become crucial elements that must be documented and preserved.

4 Case studies stage by stage

In this section the stages of computational modeling identified in the previous sections will be revisited. At each step an example of the decision *process* for the high-level and low-level models will be presented, followed by a discussion of the *data management* issues and how they were (or should have been) addressed. Although conceptually the scientific method and computational modeling follow a sequence of stages, the conceptualization and actual work done typically jumps around. For example, one may try to implement a particular algorithm only to discover that it cannot efficiently be done and be forced to reconceptualize the scope of a simulation. It is an increasingly desirable practice that the researcher explicitly documents this iterative decision-making process itself, as it increases reproducibility (Freedman et al. 2018).

For the case study in this chapter, both the high- and low-level model codes and supporting documentation can be found in the *main repository* (https://github.com /skagit/ntu_thesis.git). Any and all materials referenced in the following section can be found at this location. The main repository was initially used for Lou-Magnuson (2018), but an additional folder has been added—*data_ management*—for this chapter. It is expected that the reader consults these materials actively while reading the following sections.

4.1 Model: Process

At this stage, the task is to abstract the core elements of theory into a mathematical form, omitting as many extraneous details as possible. The phenomenon to be modeled was the development of grammatical complexity and, further, the relationship to the social network of language users. The research goal was ultimately to construct a model with a sufficient level of detail to measure synthesis (the morpheme-to-word ratio) directly. However, such computational operations on hierarchical data structures are expensive in terms of computation time, and many simulations would need to be run to explore the effects of varied social network parameters.

Thus, before work on the simulation coding began, we decided to create two related models: the high-level one that would be relatively light computationally but provide indirect support for our primary hypothesis, and the low-level model that would be computationally expensive but provide direct support for the hypothesis. In effect, the high-level model would be used to explore the space of model parameters and social network configurations, while the low-level model would be invoked to dive deeply into specific scenarios of interest.

Besides the decision to split the model for computational considerations, at this stage the relevance for the results to human studies was also decided. Within the field of language change, there exists a relatively recent, lab-based paradigm known as iterated learning used to simulate intergenerational transfer of knowledge (Kirby,

Cornish, & Smith 2008). At the time of our work, a number of small studies had been conducted looking at the emergence of compositionality in human communication (see Smith & Kirby 2012, chapter on compositionality, for comprehensive overview). However, the existing methods were unable to incorporate social network structures and were unable to operate at scales. To make sure the behavior of our models could be comparable with this growing body of work, the scope of what the model language could express was designed to match as closely as possible the stimuli used in the human experiments. Similarly, the procedure used for communication was designed to match that used in the human studies.

Beyond the desire to make the simulation results applicable to a set of existing studies, there is a strong hypothesis in the greater field of cognitive science that much of what the human mind does is learn using a simplicity bias (Chater & Vitányi 2003). There exists a well-defined mathematical principle that models this kind of simplicity-based learning and even specific algorithms that use the principle to learn context-free grammars from unstructured data, and model processes of language acquisition and evolution (Onnis, Roberts, & Chater 2002; Roberts, Onnis, & Chater 2005). To place the simulation results in this emerging, unifying context of simplicity-based learning, the decision was made to adapt the existing grammar learning algorithm into the model structure and use this principle as the basis for the agent's actions.

4.1.1 Model: Data management   This is the stage most often underrepresented in the archiving of simulation data. While the design decisions are explained in part in the research products for the two models, there is no direct indication in our collection of source files and simulation data that makes clear that certain assumptions of the model relate to specific studies already conducted or to broader scientific principles. As suggested in Dijkstra and De Smedt (1996), the decisions at this stage correlate with Marr's (1982) levels that squarely distinguish computational studies from human-based ones.

In future projects, we suggest that a document be added to the archive with sections corresponding to Marr's levels of analysis: computation, algorithm, and implementation. Inside the data management folder of the main repository, we have added a brief example of such a document for the high-level model—*model_exposition*. To keep the example reasonably short, we have omitted the details of the social network simulation,

which were also omitted in the preceding presentation of the case studies.

This document has a section for each of Marr's levels. In the computation section the overall objective of the model is given, as well as the inputs, outputs, and transformation process. The algorithmic section provides a broad overview of how the inputs are transformed to the outputs and, importantly, provides the rationale for the various steps involved. The mathematical details have not been included, nor would we recommend they be. The purpose is not to provide a document that enables reproducibility, but one that explains the design decisions involved. For example, it mentions the kinds of computations, such as probabilistic process, that are used, but rather than specify probabilities or parameters, it mentions why the process was designed the way it was. Crucially, it links the rationale of the algorithm design to the computation goals.

4.2   Simulation: Process

The simulation stage is largely the engineering of the software that runs the model. One of the first issues to be settled is the programming language used to code the model and specific scenarios to be run. For these projects, we considered different programming languages. Throughout the social sciences, Python has become the lingua franca for computational work for three reasons, it has a large ecosystem of available tools for almost every common task, performant libraries for linear algebra and matrix-bases computations (a staple representation used in the hard sciences), and a shallow learning curve with code that reads very similar to natural language. However, in contrast to the physical sciences, much of linguistic theory is based on symbolic manipulations (cf. phonological substitution rules) that are not easy to express as matrix operations. With these kinds of computations, Python is one of the least performant languages and a poor choice for constructing flexible, scalable models. In contrast, Julia is every bit as easy to learn and as clear as Python, but with execution speeds that are among the fastest available. Thus, our decision was set on the emerging language Julia.

4.2.1 Simulation: Data management   Perhaps more than any other stage, data management is most pervasive in the preparation of the source code files used in the simulation. While one does not need to be a professional software engineer to program simulations, a

number of best practices are crucial for maintaining the relevance of the code throughout its life cycle.

The main point to keep in mind is that unlike with professional software projects, one is not producing a final software product to be consumed by users. Instead, the researcher produces a computational model, and the effort should be placed on making that model as transparent as possible for other researchers to read, understand, and potentially reuse. Thus, organizational and code decisions should be guided first and foremost by clarity of the model.

Organization of the code should be kept as modular as possible, and minimally, the code that deals with the main steps defined at the algorithmic level should be kept apart from one another. For a large software project, such as the low-level model code in the main repository, we can see that there are a number of source code files defining natural concepts of the algorithm: the model deals with agents and networks, grammars and rules, and each of these is compartmentalized in a single place. In contrast, for small software projects such as that of the high-level model, there is but a single source file. For the high-level model, despite having the same fundamental components, the entirety of its source code is smaller than most individual pieces of the low-level model. Rather than spread the code over dozens of files, each just a few lines in length, it was kept together in one place. Additionally, it was written more linearly, whereby the smallest pieces of the model are presented first, followed by the code that uses them. This makes the source more like a narrative, introducing new concepts in a bottom-up fashion, from least to most complex.

Another aid to clarity is to write your code to be read and not just run. While it is easier to simply write code using abbreviated names and without comments, it will be extremely difficult to decipher yourself several weeks or months later, let alone for readers who are trying to understand it for the first time. Take for example the code that defines a linguistic construction and a language, located at the top of the main source file for the high-level model, and partially reproduced in figure 29.2. Even if you have never seen a single line of code in the programming language Julia before, there are plain English sentences explaining what each section does and the intent behind them. Additionally, items in the code are given long and so-called self-documenting names:

```
### DATA TYPES
mutable struct Construction
# a Construction has a meaning, some original source
signal pattern,
# and a level of reanalysis
meaning::Int64
origin::Int64
level::Int64
end
mutable struct Language
# a Language is a collection of meanings and
constructions
# its primary importance is not to model human
language, but to keep global variables such as id
numbers coordinated over the agents, but local to the
simulation in multiprocessing
meanings::Int64
origins::Int64
construction_i::Int64
constructions::Array{Construction,1}
end
```

Figure 29.2
Write to read.

again, not knowing anything about Julia, you could certainly direct your attention to the part dealing with a *construction* versus the part dealing with a *language*, and the comments clue you in on the details.

A common pitfall, especially when eager to get running code for some initial results, is to leave such comments out and use abbreviations to save yourself typing out long names. If I was being lazy I could have just named the first object *con* and given its components names with their first letters, *m*, *o*, and *l*, respectively. While writing the code, I would surely remember that *con* was short for construction and that *m* was short for meaning. However, how could an external reader of the code be expected to know what those symbols meant?

A further blow to clarity (as well as engineering) is not dividing one's code into small units of logical work. For example, on lines 325–329 (reproduced in figure 29.3) of the high-level model code there is a small function called *select_events*. First, note that thanks to the naming, it is clear that this piece of code is responsible for selecting some events, and thanks to the comments, we know that those events are the conversations that happen among the agents. The meat of the code is a single line (327) that

```
#select the conversations to happen this round of
communication
function select_events()
return shuffle(network.connections)
end
```

Figure 29.3
Small logical units.

simply implements the command "randomize a list of values" (which again, thanks to good naming, you know are the connections in a network). This single line would be trivial to write out on its own in the few places it occurs in the code. However, it captures a fundamental idea, how the agents' exchanges are chosen.

We could easily imagine a future in which, rather than at random, the agents chose their partners in some other manner. If we had not broken this piece of code out into a single reusable function, it would be much more error-prone to change: we would have to hunt down every location this happens and copy and paste the new selection code in its place. The more that code was used, the more likely it is that we will miss a piece or make a mistake in replacing it. In terms of clarity, the reader might not know what this code does each place we find it. We could add a comment explaining it, but we again run the risk of forgetting to comment it every place it occurs, or worse, forgetting to update a comment when changing it again. By keeping it in a single place, we make sure that it is easy to change (just this one place) and signal to the reader of the code that (however trivial and small) this code is important.

There are many more software engineering tips that could be given here, but these were selected primarily on the grounds of making the code a readable object for those wishing to engage the model in detail. To reiterate, beyond the engineering principles involved, strive to make the model clear to the reader of the code, including yourself.

4.3   Results: Process and data management

At this stage, the main concerns are relating the data to existing work and organizing the archive for easy consumption. The process of relating data to existing work is typically handled in the research output, such as a journal article or white paper, where one cites relevant studies and explains how they relate; for computational modeling this is no different than for traditional academic work. For that reason, we will focus on aspects of preparing the model source code for consumption.

In section 4.1.1 we introduced an exposition file and emphasized the importance it plays in linking the rational between the computational and algorithmic levels. Similarly, it provides a chance to link the source itself to those algorithmic steps. In the implementation section of that file, we make direct references to key parts of the algorithmic steps and how those steps can be found in greater detail in pseudo-code files.

Pseudo-code, as the name suggests, is a computational description of your model's code that one could implement in any given programming language but is itself not code in some particular language. It has all the details required to be translated fully into an implementation of your algorithms, but is written as closely as possible to plain English (or any other natural language), so that the reader can understand what your model code does without having to understand the programming language you chose to code in. The virtue of pseudo-code is that it exposes the lowest-level details of the model, that is, how your algorithmic steps were turned into and executable implementation, but without placing technical burden on your reader.

In the algorithmic section of the exposition file, two main steps of the model are clarified: an exchange stage where the agents communicate, and a transfer stage where new agents learn the language and replace the old. In the final, implementation section of the exposition file, these stages are each linked to individual pseudo-code files, contained in a pseudo-code directory alongside the model code. For example, looking at the *diffuse* file (a portion of which is reproduced in figure 29.4) you can see the details of the computations that happen during the exchange stage. Importantly, the exchange and transfer stages correspond to the two main theoretically driven concepts in language change, and keeping these stages separate and identifiable within the computer code helps any researcher to relate the algorithmic level with the computational level. In other words, organizing and labeling the code in ways that mirror the conceptual assumptions and theoretical choices of the researcher can promote reproducibility, as well as further expansions, and modifications of the model in follow-up studies.

For instance, in the exposition file we mentioned that agents attempt to decode expressions (called signals

```
Exchange (speaker, hearer)
pick a random meaning
pick a random signal from speaker's actives for the
meaning
IF hearer Can Understand OR speaker Can Repair
add speaker id and signal to hearer history
add signal to passive if not already present
IF signal already in passive AND chance <= 0.25
add signal to hearer active
END
ELSE
create new zero level signal
add signal to each agent's active and passive
add each agent's id and signal to each other's history
END
END

Can Understand (speaker signal, hearer)
FOR EACH passive signal hearer has for meaning
IF meanings and origins match
IF reanalysis levels within 1 OR speaker signal level
is zero
RETURN true
    END
  END
  RETURN false
END
```

Figure 29.4
Exchange and *Can Understand*: pseudo-code.

```
function canunderstand(target, hearer)
knowns=hearer.passive[target.meaning]

if target.level==0 #paraphrastic phrases are always
understood
return true
end
for known in [network.language.constructions[index]
for index in knowns]
#phonetics share same origin, and changes not great,
then understood
if known.origin==target.origin && abs(known.level-
target.level) <= 1
return true
end
end

return false
end
```

Figure 29.5
Can Understand: Julia code.

in the code) of others, and do so if a given signal they encounter is close enough to a signal they already know. Here we can see exactly how "close enough" is defined in terms of the integer values used in the simulation. It may seem unnecessary to specify such a simple computation like this in pseudo-code but compare it to the actual Julia implementation in lines 252–264 of the high-level model source code (reproduced in figure 29.5).

The intent of this code, despite the comments, is obfuscated by the mechanics of accessing the required variables and syntax unique to Julia: in the pseudo-code the plain English *reanalysis levels are within one* corresponds to the Julia *abs(known.level − target.level) <= 1* where the reader must know the naming conventions and data layout of the *known* and *target* signals, that the *abs* call is to a Julia library function that computes the absolute value, and must finally reason that testing that

absolute value of a difference of two integers is less than or equal to another integer captures mathematically the idea of being within a range. The important concept for the model is what was stated in the pseudo-code, and everything else is just circumstantial to Julia and the software engineering decisions made by the coder. Another advantage of detailing pseudo-code next to the code is for debugging. Anyone knowledgeable of the specific programming language inspecting the code can verify whether the actual code does implement what the pseudo-code intends to code, or whether a bug or a typo may need correcting (for instance, something as trivial as the use of a +sign instead of −in the *abs* example above).

Finally, something that may strike a reader familiar with archives of other social science studies is that the raw data themselves are missing. Simulations, often more so than human experiments, generate a tremendous volume of data. For example, in our low-level model, a single run of a single condition easily produces five gigabytes of data, which is already large for many consumer laptops to hold in memory. Even more problematic, there were dozens of conditions to be run, where each condition was replicated fifty to one hundred times. Thus, the raw data for any one condition would require keeping hundreds of gigabytes of data in memory and storing

many terabytes of data in the archive in all. This is not practical.

However, where differences between people generate differences in behavior in human studies, computational simulations make use of pseudo-random number generators to generate randomness. While there are many algorithms in use, a crucial commonality is that they require some initial input (usually an integer) to start the process, called the seed. Such generators are not truly random, as even though the sequences they generate are statistically random, they are totally determined by the seed—the same seed value will produce the same sequence every time. To make a computational model truly reproducible, the seed values used to generate the data must be clearly documented. When done so, only the code is required to be stored, as any portion of a massive data collection can be reproduced on demand.

5 Conclusion and suggestions for best practices

To meet greater demands of open science in the social sciences and linguistics, there is a much greater burden on the computational modeler to bridge the gap between how theories are typically presented in plain language, and their expression as mathematical objects that are amenable to implementation as a computer program. Indeed, this decision-making process is non-arbitrary and explicitly aligns the model with particular existing experiments and theoretical camps within the discipline. As such these decisions need to be a focus of documenting and archiving models in a manner that makes them reproducible and multipurpose.

Ideally, a model archive should contain an exposition document that outlines the assumptions made while developing it. As suggested in Dijkstra and De Smedt (1996), the three conceptual levels outlined in Marr (1982)—computation, algorithm, and implementation—provide a principled organizational base. There should also be pseudo-code available for the algorithmic steps, and the exposition document should make clear which pseudo-code corresponds with which step. In this way, not only are the assumptions documented, but interested readers can understand the computations without having to know the particular programming languages used to code the model. Especially as programming languages come and go, the model itself does not become irrelevant as the computations are preserved in a form that is universal.

For computational modeling to advance as a relevant and complementary method in linguistics, both models themselves must evolve beyond incidental artifacts of research. That is, it is no longer sufficient to just publish the source code, or even worse, just a binary executable file in an archive. Instead, any model (code and process) must become reproducible and informative in its own right. While there is no gold standard for how this should be done, this chapter has outlined the abstract differences between computational models and traditional experimental studies and, moreover, provided a framework for data management and preservation of computation models as data.

Notes

1. There does not exist any singular measure of the complexity of a language, nor any singular measure of the complexity of a language's morphology. In the two works presented in this chapter, morphological complexity was defined as the measure of synthesis (Greenberg 1960), which is average number of morphemes per word over a corpus of language use. Again, this is not the only measure of morphological complexity, and it is not without problems; for a discussion of alternatives and rationale for usage, the reader is directed to Lou-Magnuson (2018). Henceforth, the term *complexity*, unless otherwise specified refers to morphological complexity quantified as synthesis.

2. This figure is inspired by figure 1.1 in Dijkstra and De Smedt (1996), in which they discuss computational modeling in the context of psycholinguistics. In the original, the traditional scientific method is not shown in full.

References

Bengio, Y., R. Ducharme, P. Vincent, and C. Jauvin. 2003. A neural probabilistic language model. *Journal of Machine Learning Research* 3:137–1155.

Bishop, C. M. 2006. *Pattern Recognition and Machine Learning*. New York: Springer.

Chater, N., and P. Vitányi. 2003. Simplicity: A unifying principle in cognitive science? *Trends in Cognitive Sciences* 7 (1): 19–22.

Dijkstra, T., and K. De Smedt, eds. 1996. *Computational Psycholinguistics: AI and Connectionist Models of Human Language Processing*. London: Taylor and Francis.

Freedman, G., M. Seidman, M. Flanagan, M. C. Green, and G. Kaufman. 2018. Updating a classic: A new generation of vignette experiments involving iterative decision making. *Advances in Methods and Practices in Psychological Science* 1 (1): 43–59.

Greenberg, J. H. 1960. A quantitative approach to the morphological typology of language. *International Journal of American Linguistics* 26 (3): 178–194.

Kirby, S., H. Cornish, and K. Smith. 2008. Cumulative cultural evolution in the laboratory: An experimental approach to the origins of structure in human language. *Proceedings of the National Academy of Sciences* 105 (31): 10681–10686.

Krizhevsky, A., I. Sutskever, and G. E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems* 25 (2): 1097–1105.

Lou-Magnuson, M. E. 2018. Intimate connections: An agent-based model of the relationship between social network structure and language typology. PhD dissertation, Nanyang Technological University, Singapore.

Lou-Magnuson, M., and L. Onnis. 2018. Social network limits language complexity. *Cognitive Science* 42 (8): 2790–2817.

Marr, D. 1982. *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*. Cambridge: MIT Press.

Onnis, L., M. Roberts, and N. Chater. 2002. Simplicity: A cure for overregularizations in language acquisition? In *Proceedings of the 24th Conference of the Cognitive Science Society*, 720–725. Mahwah, NJ: Lawrence Erlbaum.

Roberts, M., L. Onnis, and N. Chater. 2005. Acquisition and evolution of quasi-regular languages: Two puzzles for the price of one. In *Language Origins: Perspectives on Evolution*, ed. Maggie Tallerman, 334–356. Oxford: Oxford University Press.

Smith, K., and S. Kirby. 2012. Compositionality and linguistic evolution. In *The Oxford Handbook of Compositionality*, ed. Wolfram Hinzen, Edouard Machery, and Markus Werning, chapter 24. Oxford: Oxford University Press. doi:10.1093/oxfordhb/9780199541072.013.0024.