THE ACT OF COMPUTER PROGRAMMING IN SCIENCE

JAVIER BURRONI

College of Information and Computer Sciences
University of Massachusetts Amherst
Amherst, MA 01003

Abstract. Classically, computers have been used as *knowledge discovery* tools insofar as the result of executing a program provides useful insight. For instance, the solution of a differential equation may help us understand the natural world, the value of a parameter of a statistical model may help us understand the probabilistic structure of a domain, the variable assignment maximising an objective function may help to further business goals. A secondary class of *knowledge discovery* stems from the act of using a programming language. By modeling a domain computationally, the developer can discover new and interesting properties of that domain, and better convey those insights to others. The purpose of this work is twofold: First, we want to show that programming languages can help their users achieve *knowledge discovery moments* and, secondly, that this property is the least exploited feature of programming languages in the general science community. We want to outline a research program with the objective of making scientific programming more efficient in its ultimate goal of *knowledge discovery*.

"[. . .] for Vannevar Bush and for many others, analog machines had a wonderfully evocative quality. They didn't just calculate an answer; they invited you to go in and make a tangible model of the world with your own hands, and then they acted out the unfolding reality right before your eyes." [Waldrop, 2002]

1. Introduction

To computer scientists, the act of transforming an idea into working code is an act of understanding. We assume that this is not because of a particular characteristic of the scientists in comparison to those from other disciplines, but because of the strong relation between our object of study (our domain) and the tools we use, i.e., programming languages. For instance, looking into the code of an algorithm, a scientist may have a good intuition about the time complexity, memory complexity, and even about the correctness of the algorithm. A relevant characteristic is that we get this understanding *in addition* to the execution of the program. It is the programming language itself that makes explicit some characteristic of the original problem and allows us to reason about it in a different way.

E-mail address: `jburroni@cs.umass.edu`.

[1]Petricek follows Foucault's concept of *episteme*. "An episteme defines the assumptions that make human knowledge possible in a particular epoch. It provides the apparatus for separating what may from what may not be considered as scientific." [Petricek, 2016]

Petricek [2016] discusses the “episteme[1], paradigms and research programmes” of programming language (PL) investigation, opening the door for new ways of research in the programming language discipline. The analysis of signs and resemblances in PL appears among the proposed topics in that work, as one that was hidden in the current episteme. “If resemblances and metaphors played fundamental role in our scientific thinking, we would not just gain interesting insights from them, but we would also ask different questions” [Petricek, 2016]. The present work is developed under the research program proposed by Petricek, but instead of looking at the episteme on which programming language research is inscribed, we focus on the epistemic[2] component of the actual programming languages: how programming languages aid *knowledge discovery*. In scientific programming, where knowledge has central relevance, we hypothesise that the capacity of programming as a device for *knowledge discovery* is under-used. One important feature of programming languages that facilities *knowledge discovery* is their formal nature, but we leave this property aside to focus on the less explored properties. Our interest lies in *knowledge discovery* as a result of the act of programming. To this end, we investigate the use of programming languages in science (section 2) examining three different patterns of usage (subsection 2.1, subsection 2.2 and subsection 2.3). Each usage pattern will be depicted with small case studies, and a discussion about names will follow (section 3). Finally, the need for a theoretical framework to improve the use of programming languages in science will be discussed in section 4.

We had two objectives when selecting case studies. Some cases were chosen because of their relevance to our hypothesis, but others for their relevance to a particular field. In all cases we hope that they aid in understanding our exposition.

2. Programming Languages in Science

One scientist is working with data and performs a linear regression. Her results show that the null hypothesis can be rejected, thus implying the statistical relation between two variables. Another scientist is modeling a social-network effect, and she finds that the model resembles a preferential attachment process. These are two different examples of what we call a *knowledge discovery moment*, a situation that appears several times during the course of a research based on programming. The first example represents the most common situation that usually happens at the conclusion of an experiment, when observing the results. Alternatively, the second example represents a more subtle *knowledge discovery moment*, a moment that may appear through reflection while creating a computational model. Notwithstanding their differences, both examples are situations where new insights are mediated by the use of programming languages. Usually, programming languages are used for calculation —the first example—, but as they are *languages*, they have features to facilitate reflection and deep thought —the second example. In particular, we have the following underlying working hypothesis: an important activity of science, and *knowledge discovery* in general, is the creation of concepts, and concepts can be thought as elements of a lower (abstract) level under the cover of a *name*. It happens that this activity is also important in programming through programming languages. The relation between both instances of this activity is not fully exploited nor understood. In this section we will explore the different uses of PL in science and how they relate to the exploitation of the different *knowledge discovery moments*. We identify at least three very different approaches to PL for use in science: a calculation-based approach, an approach based on domain specific languages (DSLs), and a simulation based approach. This distinction is fuzzy, but it will help us to expose different aspects of programming.

[2]In this work, we follow Turkle and Papert [1990] with regard the use of the word *epistemology*. Instead of having a single form of knowledge, the propositional, we build on top of the idea of “different approaches to knowledge”.

2.1. **Programming Languages as Calculation Devices.**

"In effect, [J. C. R. Licklider] explained to them, everyone at the Pentagon was still thinking of computers as giant calculators and data processors." [Waldrop, 2002]

Computers have been used as scientific calculation devices since their creation. Fortran established a way to use the computer that consisted of writing a model, compiling a file and finally executing it. The main *knowledge discovery moment* stemming from PL designed to do computation is the moment when the result is available, after the execution. In this case, the knowledge is *crystallised* in the result. A particular *use* of these PL that moves away from this paradigm is the exploratory analysis possible when the PL provides a **read-eval-print-loop**: a REPL. Contemporary examples of this include Mathematica, Jupyter Notebook, RStudio and Matlab. These tools are useful for both large and small problems. Researchers can readily probe small aspects of the system and adapt an experimental plan based on the result of the probes. This is not exploiting a particular feature of a PL (aside from the REPL)—instead it is exploiting the interactive computing paradigm, as devised in the 1950s. The role of interactive computing in the programming language community has changed over time. Smalltalk provides an approach to interactivity that is completely different to the REPL, as can be seen for instance in Goldberg [1984], but this idea is not used in mainstream scientific research. Also, Mathematica introduced a way to program in an environment that resembles an interactive document, and this idea was followed by Python with its Jupyter Notebook and RStudio. The level of interactivity added to these environments—for instance, with Bokeh [Bokeh Development Team, 2014]—and the enormous progress on performance for numerical computing put these tools among the favourite options for scientific programming [Shen, 2014]. It is worth mentioning that this idea can be traced back to the WEB environment developed by Knuth [1984], where LaTeX and Pascal code were integrated in a single document. An important difference between the WEB environment and Jupyter is that the former was an *exposition of code*, a tool to facilitate understanding of the code.

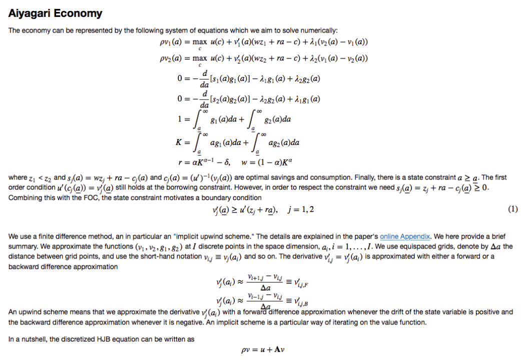
2.1.1. *Case Study: Quant**Econ**.*

"Quant**Econ** is a NumFOCUS fiscally sponsored project dedicated to development and documentation of modern open source computational tools for economics, econometrics, and decision making."[3]

This project made an important step forward in the use of programming for economics. It gathered disparate research in economics, then organized and presented this research to the community. In addition to its quality, the participation of relevant figures as a Nobel laureate attracted the attention of many researchers of the field. Among other things, it created a collection of Jupyter notebooks with a large part of current economic and econometric ideas. A typical notebook from Quant**Econ** has the content of a *lecture* where a specific topic is analysed (see Figure 1). These notebooks usually include very detailed descriptions of a model, consistent with a published article, as in Figure 1a, followed by (or interleaved with) an implementation of the model, as show in Figure 1b. We have reproduced a small functional unit in Figure 2 and a code snippet of this function is shown in Figure 1c. This code demonstrates a problem that we wish to expose with this work: the task of instructing the *computer* what to do accounts for the majority of the identifiers. These are strictly *mechanical operations*. In this case, the program is a calculation device, translated from mathematics, and the additions are nuisances required for it to work: identifiers like `self`, reshape or slice objects. Our hypothesis is that a theoretical reader will understand this program, in the

[3]`http://quantecon.org`

sense that the knowledge crystallised by the programmer will be recovered[4]. However, it is unlikely that new concepts will emerge from this exposition, leading to a *knowledge discovery moment*. We believe that new concepts will not emerge because it is hard to relate this code to other concepts creating metaphors and abstractions. We could create suggestions to improve the implementation and even to improve the language. However, we think that it is better to first acknowledge the existence of a particular problem, then understand its causes and finally propose solutions to it.



Aiyagari Economy

The economy can be represented by the following system of equations which we aim to solve numerically:

$$\rho v_1(a) = \max_c \; u(c) + v_1'(a)(wz_1 + ra - c) + \lambda_1(v_2(a) - v_1(a))$$
$$\rho v_2(a) = \max_c \; u(c) + v_2'(a)(wz_2 + ra - c) + \lambda_2(v_1(a) - v_2(a))$$
$$0 = -\frac{d}{da}[s_1(a)g_1(a)] - \lambda_1 g_1(a) + \lambda_2 g_2(a)$$
$$0 = -\frac{d}{da}[s_2(a)g_2(a)] - \lambda_2 g_2(a) + \lambda_1 g_1(a)$$
$$1 = \int_{\underline{a}}^{\infty} g_1(a)da + \int_{\underline{a}}^{\infty} g_2(a)da$$
$$K = \int_{\underline{a}}^{\infty} ag_1(a)da + \int_{\underline{a}}^{\infty} ag_2(a)da$$
$$r = \alpha K^{\alpha-1} - \delta, \quad w = (1-\alpha)K^{\alpha}$$

where $z_1 < z_2$ and $s_j(a) = wz_j + ra - c_j(a)$ and $c_j(a) = (u')^{-1}(v_j(a))$ are optimal savings and consumption. Finally, there is a state constraint $a \geq \underline{a}$. The first order condition $u'(c_j(\underline{a})) = v_j'(\underline{a})$ still holds at the borrowing constraint. However, in order to respect the constraint we need $s_j(\underline{a}) = z_j + ra - c_j(\underline{a}) \geq 0$. Combining this with the FOC, the state constraint motivates a boundary condition

$$v_j'(\underline{a}) \geq u'(z_j + r\underline{a}), \quad j = 1, 2 \tag{1}$$

We use a finite difference method, an in particular an "implicit upwind scheme." The details are explained in the paper's online Appendix. We here provide a brief summary. We approximate the functions (v_1, v_2, g_1, g_2) at I discrete points in the space dimension, $a_i, i = 1, \ldots, I$. We use equispaced grids, denote by Δa the distance between grid points, and use the short-hand notation $v_{i,j} \equiv v_j(a_i)$ and so on. The derivative $v_{i,j}' = v_j'(a_i)$ is approximated with either a forward or a backward difference approximation

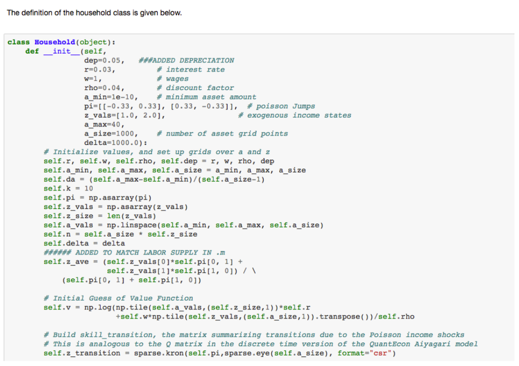
$$v_j'(a_i) \approx \frac{v_{i+1,j} - v_{i,j}}{\Delta a} \equiv v_{i,j,F}'$$
$$v_j'(a_i) \approx \frac{v_{i-1,j} - v_{i,j}}{\Delta a} \equiv v_{i,j,B}'$$

An upwind scheme means that we approximate the derivative $v_j'(a_i)$ with a forward difference approximation whenever the drift of the state variable is positive and the backward difference approximation whenever it is negative. An implicit scheme is a particular way of iterating on the value function.

In a nutshell, the discretized HJB equation can be written as

$$\rho v = u + \mathbf{A}v$$

(A) Screenshot of Aiyagari continuous time's model description.



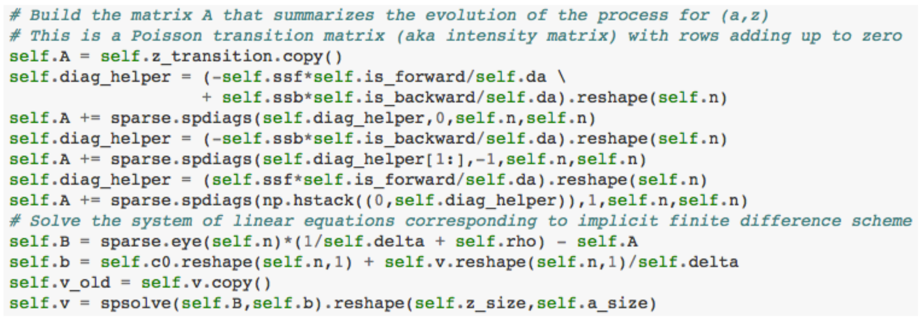
The definition of the household class is given below.

```
class Household(object):
    def __init__(self,
                dep=0.05,   ###ADDED DEPRECIATION
                r=0.03,         # interest rate
                w=1,            # wages
                rho=0.04,       # discount factor
                a_min=1e-10,    # minimum asset amount
                pi=[[-0.33, 0.33], [0.33, -0.33]],  # poisson Jumps
                z_vals=[1.0, 2.0],                # exogenous income states
                a_max=40,
                a_size=1000,    # number of asset grid points
                delta=1000.0):
        # Initialize values, and set up grids over a and z
        self.r, self.w, self.rho, self.dep = r, w, rho, dep
        self.a_min, self.a_max, self.a_size = a_min, a_max, a_size
        self.da = (self.a_max-self.a_min)/(self.a_size-1)
        self.k = 10
        self.pi = np.asarray(pi)
        self.z_vals = np.asarray(z_vals)
        self.z_size = len(z_vals)
        self.a_vals = np.linspace(self.a_min, self.a_max, self.a_size)
        self.n = self.a_size * self.z_size
        self.delta = delta
        ###### ADDED TO MATCH LABOR SUPPLY IN .m
        self.z_ave = (self.z_vals[0]*self.pi[0, 1] +
                      self.z_vals[1]*self.pi[1, 0]) / \
            (self.pi[0, 1] + self.pi[1, 0])

        # Initial Guess of Value Function
        self.v = np.log(np.tile(self.a_vals,(self.z_size,1))*self.r
                        +self.w*np.tile(self.z_vals,(self.a_size,1)).transpose())/self.rho

        # Build skill_transition, the matrix summarizing transitions due to the Poisson income shocks
        # This is analogous to the Q matrix in the discrete time version of the QuantEcon Aiyagari model
        self.z_transition = sparse.kron(self.pi,sparse.eye(self.a_size), format="csr")
```

(B) Extract of the Household class.



```
# Build the matrix A that summarizes the evolution of the process for (a,z)
# This is a Poisson transition matrix (aka intensity matrix) with rows adding up to zero
self.A = self.z_transition.copy()
self.diag_helper = (-self.ssf*self.is_forward/self.da \
                    + self.ssb*self.is_backward/self.da).reshape(self.n)
self.A += sparse.spdiags(self.diag_helper,0,self.n,self.n)
self.diag_helper = (-self.ssb*self.is_backward/self.da).reshape(self.n)
self.A += sparse.spdiags(self.diag_helper[1:],-1,self.n,self.n)
self.diag_helper = (self.ssf*self.is_forward/self.da).reshape(self.n)
self.A += sparse.spdiags(np.hstack((0,self.diag_helper)),1,self.n,self.n)
# Solve the system of linear equations corresponding to implicit finite difference scheme
self.B = sparse.eye(self.n)*(1/self.delta + self.rho) - self.A
self.b = self.c0.reshape(self.n,1) + self.v.reshape(self.n,1)/self.delta
self.v_old = self.v.copy()
self.v = spsolve(self.B,self.b).reshape(self.z_size,self.a_size)
```

(C) Code snippet of method solve_bellman.

FIGURE 1. Screenshots of a Quant**Econ** notebook[5]. This notebook is based on the model of Achdou et al. [2014].

2.2. Domain Specific Languages.

Domain specific languages (DSL), as the name states, contain elements proper of their domain while reducing nuisance added by constructs that are only used for general purpose computation. Our main focus are those DSL for which the domain is close to the scientist's domain. In the previous sections we analysed PL that were meant for computations, and

[4]We use the *crystallised information* created by Hidalgo [2015]: the code is a crystallised version of the writer's knowledge.

[5]`http://nbviewer.jupyter.org/github/QuantEcon/QuantEcon.notebooks/blob/master/aiyagari_continuous_time.ipynb`

some of them are proper DSLs for mathematical calculations. For our purposes, those languages are not considered DSLs because they are more relevant as calculation tools.

The use of domain specific languages yields different *knowledge discovery moments*. When a DSL is used, we have a *knowledge discovery moment* which can be related to Kuhn's view of *normal science*: the concepts are already defined in the language and the researcher build on top of these concepts. Note that this language may encode an existing paradigm or exhibit an entirely new way of thinking. However, there is a *knowledge discovery moment* prior to this, and this happens in the act of designing a DSL. The challenge is to model the domain's concept—the basic axioms—in term of the metalanguage, and while doing this task a *knowledge discovery moments* may emerge. Therefore, the design of DSLs is an interesting moment, and some process like *semantic-driven design* increase their usefulness for the creation of knowledge:

> "The semantic-driven design process consists of two major parts. **The first part is concerned with the modeling of the semantic domain, which is based on the identification of basic semantic objects and their relationships**. The second part consists of the design of the language's syntax, which is about finding good ways of constructing and combining elements of the semantic domain." [Erwig and Walkingshaw, 2014]

It is clear that finding *the basic semantic objects and their relationships* are fundamental tasks of science and while doing this, a *knowledge discovery moment* emerges. As a downside, this moment is only *offered* to the designer of the DSL, not to the users. If the designers are not specialists in the domain of the language (e.g., a computer scientist working on a DSL for biochemistry), the possibility of discovery may not be great. A corollary of this fact is that creation of DSLs should be encouraged to the sciences, in collaboration with computer scientists, as a way to make discoveries.

2.2.1. *Case Study: MathMorph.* The first case study is MathMorph [MathMorph development team, 2000], a DSL for working with mathematical objects, developed in the late nineties using Squeak, a Smalltalk dialect. This is an example of the idea from Priestley [2011] that "[for the Smalltalk community, programming was thought of] as a process of working interactively with the semantic representation of the program, using text simply as one possible interface"[6].The conclusion of Notarfrancesco and Caniglia [2000], where the project is described, shows why it is relevant to our research:

> "While object orientation is normally an abstraction task, where real things have to be represented in a virtual space, the same practice has the inverse result when mathematical notions are modeled. The model of a mathematical concept is more tangible than the concept itself. Instead of *abstracting*, one experiences the rather unusual feeling of *concreting*.[7]
>
> Along these few years we have also noticed many interesting facts regarding pedagogy. A few of them are:
>
> - The students learn Squeak as a natural consequence of thinking about mathematical ideas.
> - Well known mathematical notions suddenly show unsuspected properties.
> - Some theorems are naturally generalised in uncommon ways. As a result, deeper than normal understanding is achieved.

[6]Cited in [Petricek, 2016]

[7]Turkle and Papert [1990] also noted this property. To them, "the computer has a theoretical vocation: it can make the abstract concrete; it can bring formality down-to-earth."

- Living examples that naively begin as simple forms of code testing, quickly become rich sources of new questions and problems.
- The classical barriers between formal definitions and intuitive ideas are changed into useful and precise specifications on how to move from the paper or blackboard to the Squeak world in a straightforward way.
- Algorithmic thinking and geometry, usually absent in the conventional approach, get included in the whole subject of study." [Notarfrancesco and Caniglia, 2000]

For the development team, the act of programming is also an act of *knowledge discovery*. Even though the project was discontinued, there are still efforts to learn about abstract mathematics using Smalltalk programming. For instance, the work on homological algebra presented in Caniglia [2015], where Smalltalk was used to understand a mistake in an homological algebra proof. Unlike automated theorem proving, this is an example of *knowledge discovery* through code.

2.2.2. *Case Study: Probabilistic Programming.* Probabilistic programming languages (PPL) are designed for probabilistic modelling through the addition of two constructs: one that allows the sampling of random variables, and other that allows for conditioning on data [Gordon et al., 2014]. The development of these languages is currently very active. A large amount of knowledge is created as a consequence of the language and not of computation. An interesting feature of some PPLs is that Bayesian networks can easily be represented as probabilistic programs [Gordon et al., 2014, section 3.1]. In that way, reasoning about programs is equivalent to reasoning about joint distributions. From this relation, opportunities for *knowledge discovery* are possible by considering inference techniques driven by the structure of those programs.

In the following we present two examples of inference techniques that were discovered because of the representation as code. First, Sankaranarayanan et al. [2013] explored the idea of analysing execution paths (symbolic execution) of probabilistic programs by finding the probability of satisfaction for a system of constraints. Second, Ritchie et al. [2015] used the continuation-passing style representation to get a faster Metropolis-Hasting proposal. Again, the code representation of the program is fundamental for the insight. On this line of thought, the "Design and Implementation of Probabilistic Programming Languages" [Goodman and Stuhlmüller, 2014] was created as an interactive book on probabilistic programming languages, centered around code. In these cases, the *knowledge discovery moments* are *offered* to the designers of the DSL, as suggested above. This happens because they are forced to think about the domain in the metalanguage.

Separately, PPL descendants of WinBUGS [Lunn et al., 2000] such as JAGS [Plummer and others, 2003] and STAN [Gelman et al., 2015] can be seen as DSL for statistical modelling in the sense that models resemble classical whiteboard statistics. A hierarchical Bayesian model can be understood as such looking at the code. In this way, the tool used to reason about the model, and the one used to create inference are unified.

Bayesian networks themselves deserves further examination. BN are directed acyclic graphs where each node represents a random variable and is associated with a conditional probability distribution. They are interesting for our investigation because they emerged as a device useful for both representing joint probability distributions and computing probabilistic queries on that representation. In its origin, they were a tool to encode probabilities in a computer, but Bayesian networks turn out to be more than simple data structures, and led to opportunities for *knowledge discovery.*

> "[Bayesian networks should be seen] not merely as a passive parsimonious code for storing factual knowledge but also as a computational architecture for reasoning about that knowledge" [Pearl, 1985]

The impacts of this new abstraction could be thought of as the impact of the Nucleic acid double helix idea. This is not a perfect example of our hypothesis as this did not emerge from observation of code. However, it has a similar characteristic: it is a computational abstraction with impact on the original domain.

2.3. **PL as a Modelling Tool: Simulations.** Leaving out the use of statistical inference —a calculation based approach—, one the most widespread ways in which programming languages are used in the social sciences is through simulations. Simulations rely on program execution for the *knowledge discovery moment*, but unlike the calculation device, the domain is modelled in the program. Additionally, they allow for the presence of emergent phenomena. Scientists model the micro-behaviors and observe the macro-behavior. This is particularly true in the so-called agent-based models (ABM) [Macy and Willer, 2002]. In these frameworks, the scientist must model two basic things: the agents and their relations. Additionally, a global constraint may be modeled in the form of an environment. The expected *knowledge discovery moments* of these models is the *emergent behavior* and the consequences of perturbations to the model, i.e., interventions. Here too, the modeling itself is also an opportunity for learning. The assumptions about the agents and relations put, either explicitly or implicitly, in the model become relevant as the result depends on them. A good programming language should be able to expose these assumptions as fundamental variables.

In simulation frameworks, as with any DSL, the level of generality in the language imposes constraints on program design. In some simulations, the domain is already crystallised in the environment, and the *knowledge discovery moments* are restricted to its theoretical framework. This is another instance of Kuhn's *normal science* (see above). Other frameworks allow for more general modeling, at the cost of making the modelling task harder. A similar effect appears at model's level:

> "There are some issues related to the application of ABM [agent-based models] to the social, political, and economic sciences. One issue is common to all modeling techniques: a model has to serve a purpose; a general-purpose model cannot work. The model has to be built at the right level of description, with just the right amount of detail to serve its purpose; this remains an art more than a science". [Bonabeau, 2002]

It is interesting that the ABM community developed a standard to describe the simulations: The Overview, Design concepts and Details (ODD) standard protocol [Grimm et al., 2006]. The elements of ODD can be seen in Table 1. This protocol established the creation of a document that describes the models and should allow for a complete re-implementation of the models using only this document —which is a text document, separated from the code. We would like to highlight that some aspects of the ODD protocol (necessarily) refer to elements of a program. Thus, it is possible that this protocol was created to solve a failure of programming languages as a communication device. In fact, the authors of the standard noted their intentions clearly:

> "There are two main and interrelated problems with descriptions of IBMs [individual based-models]: (1) there is no standard protocol for describing them and (2) IBMs are often described verbally without a clear indication of the equations, rules, and schedules that are used in the model." [Grimm et al., 2006]

Overview	Purpose State variables and scales Process overview and scheduling
Design concepts	Design concepts
Details	Initialization Input Submodels

Table 1. Description of elements from the ODD standard. Table taken from Grimm et al. [2006]

2.3.1. *Case Study: NetLogo.* One of the programming languages most used in the social sciences is NetLogo [Railsback and Grimm, 2011]. It is important to know which features of this language attracted many scientists from disparate disciplines such as economy, ecology, sociology and political sciences. NetLogo is a multi-agent version of Logo. Logo was designed by Feurzeig and Papert in 1967 as a tool to make programming accessible to children. The principal object of it is the turtle: it was "an on-screen 'cursor' that showed output from commands for movement and small retractable pen, together producing line graphics"[8]. For many computer scientists, the popularity of NetLogo in the social sciences may be odd (as it was designed for children), which is an indication that further research may be necessary to understand the advantages of this language for social science. There are two elements that may explain this phenomenon. First, it is a multi-agent system, thus making it suitable for agent-based modelling. Also, Logo featured the idea of 'body syntonic' reasoning:

> "The Logo turtle was designed to be 'body syntonic', to allow users to put themselves in its place. When children learn to program in Logo, they are encourage to work out their programs by 'playing turtle'. The classic example of this is developing the Logo program for drawing a circle. This is difficult if you search for it by analytic means (you will need to find a differential equation), but easy if you put yourself in the turtle's place and pace it out. (The turtle makes a circle by going forward a little and turning a little, going forward a little and turning a little, etc.)." [Turkle and Papert, 1990] (see also Papert [1980])

Then, it is possible that the popularity of NetLogo[9] is a validation of Papert's ideas. Consequently, for simple models NetLogo code reads as a model of the domain: the percentage of identifiers related to the domain are high relative to the *system* identifiers. An example of NetLogo code is shown in Listing 1: a code snippet for a predator-prey model[10].

Listing 1. Predator-Prey Model

```
;; Predator-Prey model from http://modelingcommons.org
to go
  ;; listen to the hubnet client
  every 0.1
  [
    listen-clients
    display
  ]

  ;; if wander? is true then the androids wander around the landscape
```

[8]https://en.wikipedia.org/wiki/Logo_(programming_language)

[9]Interestingly, the name Logo was "derived from the Greek word for 'word' or 'thought' [λόγος]." Goldenberg [1982] This relation between 'words', 'thought' and programming languages is the basic element of our research.

[10]http://modelingcommons.org/browse/one_model/2401#model_tabs_browse_info

```
  if wander?
    [ androids-wander ]

  ;; the delay below keep plants from growing too fast
  ;; and predator/prey from losing points too fast
  every 3
  [
    if any? turtles
    [
      plants-regrow
      ask students
      [
        set energy energy - 0.5
        if energy <= 0
        [ student-die ]
        update-energy-monitor
      ]
      do-plot
    ]
    tick
  ]
end
```

3. The final act: Naming

The literature on naming and programming languages is ample and the advantages of a good naming practice are very well known. There is a current effort on software engineering trying to identify linguistic patterns in source code that are recognisable as bad practices [Arnaoudova et al., 2016] and the relation of names with the code's overall quality [Butler et al., 2009]. Our perspective is not identical to that of the software engineering community, but shares some elements with it. In particular, we consider naming a very important part of programming, but specifically we conjecture that naming allows for a very relevant *knowledge discovery moment*. The act of naming is equivalent to the act of making an abstraction, and abstractions are important outputs of science.

In the early years of software development, users of Fortran employed single-character variable names. This was partly due to technical limitations, but sometimes this was thought of as an imitation of mathematics. While it is true that some mathematical variables are named with a single character, those variables are also typically referred to more expressively in natural language in terms of their properties. Also, naming mathematical objects is an activity continuously performed, as the names become a means for abstracting useful and common mathematical objects. Lastly, mathematicians usually discuss objects in the context of a larger piece of writing, where single-character abbreviations are defined before they are referenced. The *reader* should be able to recognise all the involved elements in the operations for it to be valid. When writing code, there is a well-known difference in relevance between the code and the accompanying text (documentation). The eye of the reader will go first to the code and later, in case of doubt, her eye will move to the documentation. This difference in relevance becomes more important when *knowledge discovery* is part of the objectives of coding.

The most well-known programming language which features a notation inspired by the notation of mathematics is APL—A Programming Language [Iverson, 2007]. The stated properties of notation that guided APL design were: "ease of expressing constructs arising in problems, suggestivity, ability to subordinate detail, economy, and its amenability to formal proof". We can argue about whether or not APL fulfilled its goals, but what is relevant to us is that the language was meant to be a tool of thought: a *knowledge discovery* tool.

Another exploration of the relation between mathematical language, programming languages and natural languages is the classical work of Naur [1975]. In that work, Naur notes the "awkwardness at the use of the word language in the context 'programming language"'. He argues that the two main differences between both are: 1) the form (written vs spoken), and 2) the way of understanding the words (fuzzy vs “perfectly well-defined”). Interestingly, he acknowledge that some uses of natural language aren’t fuzzy, in particular the use in science, and this is the use of interest for the present work. Even thought the use in computing is well-defined, we note that there exists a level of freedom in programming languages: the freedom of naming.

Since the creation of LISP [McCarthy, 1960], there were no strong technical limitations on identifier naming, and naming became an important part of programming. The Smalltalk community has a long tradition of giving a central relevance to names. Beck devoted a significant amount of his work [Beck, 1997] to the use of names as a communication device. For example, the “Intention Revealing Messages” are thought of as “the most extreme case of writing for readers instead of the computers”. Interestingly, this tradition is also related to *knowledge discovery moments.* In Wilkinson [2014], the author explicitly states that “putting a name is the most difficult part [of programming] as it is creating knowledge”. This idea of writing for readers was also investigated by Knuth. Working in a way to relate the code and its accompanying text, he developed the concept of *literate programming* [Knuth, 1984]. In that work, he stated that:

> “Instead of imaging that our main task is to instruct a *computer* what to do, let us concentrate rather on explaining to *human beings* what we want a computer to do.”

We could rephrase scientific programming as “explaining to human beings, *in a way that new things can be learned*, what we want a computer to do.”

In the aforementioned work, Knuth made also a reflection about the level of verbosity of WEB’s sections, that can be translated to others code units:

> “The name of a section (enclosed in angle brackets) should be long enough to encapsulate the essential characteristics of the code in that section, but it should not be too verbose. I found very early that it would be a mistake to include all of the assumptions about local and global variables in the name of each section, even though such information would strictly be necessary to isolate that section as an independent module. The trick is to find a balance between formal and informal exposition so that a reader can grasp what is happening without being overwhelmed with details.” [Knuth, 1984]

4. Past and Future Work

There are at least two dimensions of work in this area: a quantification and a philosophical dimension.

4.1. **Quantification.**

Some of the ideas presented on this work could be quantified. It would be very informative to know the distribution of sizes of variable names, the size of functional units and the amount of code duplication both in scientific code and in *non-scientific* code. These variables will show how different scientific code is from *non-scientific* code, but the cause of the difference will still be unknown. Ideally, we would like to have a causal study relating programming features with scientific *knowledge discoveries.*

4.2. **Philosophical dimension.**

Additionally, it is good to have a clear understanding of what to measure. To this end, it is necessary to build a theory; the epistemic component of the act of programming should be studied. The work of philosophers could be of help. In the nineteenth century, Frege et al. [1951] described the relation between *objects*, *concepts* and *references.* More ideas were developed in the twentieth century, beginning with the work of De Saussure [1916], and Peirce [1931], but including Foucault [1966] and Barthes [1977]. There is a complete field of science devoted to the research of the use of words, but what is relevant to us is how the words in code help to understand and to create new *knowledge discovery moments*. This enterprise already started with Noble et al. [2006], where design patterns were analysed from a semiotic perspective (the study of sign processing and meaning-making).

In the work of Tanaka-Ishii [2010], programming itself was analysed semiotically. This field created (at least) two models of signs: a two-element model proposed by de Saussure and composed of the *signifier* and *signified* (the dyadic model), and a three-element model proposed by Peirce and composed of the *object*, *label representamen*, and *interpretant* (the triadic model). In the dyadic model, the two sides of a sign corresponds to the *signifier* (a label) and the *signified* (the concept). For instance, if we analyse the term 'tree', the term itself is the *signifier* which invokes the concept of a tree, the *signified*. By contrast, the triadic model adds a third element. It is "the *representamen* (label) that evokes the *interpretant* (idea or sense) defining the *object* (referent). In the example of the tree, the label 'tree' (*representamen*) evokes the idea of the tree (*interpretant*), which designates the referent tree (*object*)"[11]. It's worth mention that different sign models expose different aspects of the sign.

Tanaka-Ishii studied, among other things, the relation between semiotic models and programming paradigms: the relation between the dyadic model and the functional paradigm, and the triadic model and the object oriented paradigm:

> "In functional programs all identifiers are dyadic, whereas in object-oriented programs dyadic and triadic identifiers are both seen. [...] In the dyadic model, different uses attribute additional meanings to dyadic identifiers. In contrast, in the object-oriented paradigm such meanings should be incorporated within the identifier definition from the beginning. Everything that adds meaning to an identifier must form part of its definition; therefore if two sets of data are to be use differently, they must appear as two different structures.[...] That is, a triadic identifier has its meaning described within its class. If two triadic identifiers' meanings differ in some aspect, then the difference must be visible within the identifiers' classes." [Tanaka-Ishii, 2010]

This work forms a framework on which we can build a complete theory that relates the identifiers on code with the creation of knowledge, but still there are many dots that need to be connected.

5. Conclusions

In linguistic theory, the Sapir-Whorf hypothesis states that language determines thought, or at least that its usage and categories influence thought.[12] It is clear that the language used for modelling a scientific object will have an influence regarding the discoveries on this object. As Diaconis put it when discussing reproducing kernel Hilbert spaces:

> "Like all transform theories (think Fourier), problems in one space may become transparent in the other, and optimal solutions in one space are often usefully optimal in the other." [Berlinet and Thomas-Agnan, 2011, Preface]

To this end, it is beneficial to do more than translate from one language to another. The problem must be thought of in the second language. In our case, the problems must be thought of in the

[11]Examples taken from [Tanaka-Ishii, 2010].

[12]`https://en.wikipedia.org/wiki/Linguistic_relativity`

context of programming languages. Information flows between a program, its writer and its reader, but once the program is written, the information is crystallised in its code. When the information that goes from the code to the reader and that is more than the information crystallised on it, we have a *knowledge discovery moment*. In this work we made the first exposition of a problem regarding the use of PL in science, and its relation to the moment of the *knowledge discovery*. Future works are required to quantify this hypothesis, and most importantly, to determine the ultimate cause of this phenomenon: Is it because of a lack of *natural expressiveness* of the programming languages, a lack in the capacity to express thoughts in a way that are easily communicable to humans, or is it because a failure in the communication of the power of PL to users outside the computer science community?

The main contribution of this work is to make explicit the need for a theoretical framework that helps to make programming language better tools for understanding in scientific programming—a need that has been present since the dawn of computer science. When Dijkstra [1972] encouraged all the programmers to "forget that FORTRAN ever existed [...] for as vehicle of thought it is no longer adequate", he also proposed "the analysis of the influence that programming languages have on the thinking habits of their users". This is still a valid research program and becomes more relevant in the domain of scientific programming. We should embrace that research program.

Acknowledgements

I would like to thank to David Jensen and Dan Garant for their thoughtful comments and suggestions; to Hernán Wilkinson for helpful discussions and references; and both to the Smalltalk User Group Argentina (SUGAR) and FAST organisation for the inspiration.

References

Y. Achdou, J. Han, J.-M. Lasry, P.-L. Lions, and B. Moll. Heterogeneous agent models in continuous time. *Preprint*, 2014. URL `http://www.princeton.edu/~moll/HACT.pdf`.

V. Arnaoudova, M. Di Penta, and G. Antoniol. Linguistic antipatterns: What they are and how developers perceive them. *Empirical Software Engineering*, 21(1):104–158, 2016. URL `http://link.springer.com/article/10.1007/s10664-014-9350-8`.

R. Barthes. *Elements of Semiology*. Hill and Wang, reissue edition, Apr. 1977. ISBN 0-374-52146-8.

K. Beck. *Smalltalk Best Practice Patterns. Volume 1: Coding*. Prentice Hall, Englewood Cliffs, NJ, 1997. URL `https://raw.githubusercontent.com/anapata/reading/master/chapter14/Draft-Smalltalk%20Best%20Practice%20Patterns%20Kent%20Beck.pdf`.

A. Berlinet and C. Thomas-Agnan. *Reproducing kernel Hilbert spaces in probability and statistics*. Springer Science & Business Media, 2011. URL `https://books.google.com/books?hl=en&lr=&id=bX3TBwAAQBAJ&oi=fnd&pg=PP11&dq=Reproducing+Kernel+Hilbert+Spaces+in+Probability+and+Statistics&ots=jU3hW31AC8&sig=Ta5-9bNtMXx-0-jCoN5DfXSeqOE`.

Bokeh Development Team. *Bokeh: Python library for interactive visualization*, 2014. URL `http://www.bokeh.pydata.org`.

E. Bonabeau. Agent-based modeling: Methods and techniques for simulating human systems. *Proceedings of the National Academy of Sciences*, 99(suppl 3):7280–7287, 2002. URL `http://www.pnas.org/content/99/suppl_3/7280.short`.

S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Relating identifier naming flaws and code quality: An empirical study. In *2009 16th Working Conference on Reverse Engineering*, pages 31–35. IEEE, 2009. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5328661`.

L. Caniglia. Homological Smalltalk Algebra. In *ESUG*, Brescia at the Università Cattolica, Italy, July 2015.

F. De Saussure. Nature of the linguistic sign. *Course in general linguistics*, pages 65–70, 1916. URL `https://www.homeworkmarket.com/sites/default/files/q2/02/04/making_sense_of_language_ch_2.pdf`.

E. W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972. URL `http://dl.acm.org/citation.cfm?id=361591`.

M. Erwig and E. Walkingshaw. Semantics-driven DSL design. *Computational Linguistics: Concepts, Methodologies, Tools, and Applications*, page 251, 2014. URL `https://books.google.com/books?hl=en&lr=&id=EhOXBQAAQBAJ&oi=fnd&pg=PA251&dq=Semantics-Driven+DSL+Design*&ots=vMe53Xh3D6&sig=2U-UCfRgSwbLAWA5w9CSnu2h5r0`.

M. Foucault. *Les mots et les choses: une archéologie des sciences humaines.* Gallimard, 1966.

G. Frege, P. T. Geach, and M. Black. On Concept and Object. *Mind*, 60(238):168–180, 1951. ISSN 0026-4423. URL `http://www.jstor.org/stable/2251430`.

A. Gelman, D. Lee, and J. Guo. Stan a probabilistic programming language for Bayesian inference and optimization. *Journal of Educational and Behavioral Statistics*, page 1076998615606113, 2015. URL `http://jeb.sagepub.com/content/early/2015/10/09/1076998615606113.abstract`.

A. J. Goldberg. SMALLTALK-80: the interactive programming environment. 1984. URL `http://cumincad.scix.net/cgi-bin/works/Show?61be`.

E. P. Goldenberg. Logo-A cultural glossary. *Byte*, 7(8):210, 1982.

N. D. Goodman and A. Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. `http://dippl.org`, 2014. Accessed: 2017-1-16.

A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani. Probabilistic programming. In *Proceedings of the on Future of Software Engineering*, pages 167–181. ACM, 2014. URL `http://dl.acm.org/citation.cfm?id=2593900`.

V. Grimm, U. Berger, F. Bastiansen, S. Eliassen, V. Ginot, J. Giske, J. Goss-Custard, T. Grand, S. K. Heinz, G. Huse, and others. A standard protocol for describing individual-based and agent-based models. *Ecological modelling*, 198(1):115–126, 2006. URL `http://www.sciencedirect.com/science/article/pii/S0304380006002043`.

C. Hidalgo. *Why information grows: The evolution of order, from atoms to economies.* Basic Books, 2015. URL `https://books.google.com/books?hl=en&lr=&id=xpPSDQAAQBAJ&oi=fnd&pg=PT9&dq=why+information+grows&ots=7eJ7aS5Jbd&sig=FanhpuDQ97SODzU0xbO87sWUy3c`.

K. E. Iverson. Notation as a tool of thought. *ACM SIGAPL APL Quote Quad*, 35(1-2):2–31, 2007. URL `http://dl.acm.org/citation.cfm?id=1234322`.

D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984. URL `http://comjnl.oxfordjournals.org/content/27/2/97.short`.

D. J. Lunn, A. Thomas, N. Best, and D. Spiegelhalter. WinBUGS-a Bayesian modelling framework: concepts, structure, and extensibility. *Statistics and computing*, 10(4):325–337, 2000. URL `http://link.springer.com/article/10.1023/A:1008929526011`.

M. W. Macy and R. Willer. From factors to factors: computational sociology and agent-based modeling. *Annual review of sociology*, 28(1):143–166, 2002. URL `http://annualreviews.org/doi/abs/10.1146/annurev.soc.28.110601.141117`.

MathMorph development team. MathMorph, 2000. URL `http://www.dm.uba.ar/MathMorphs/`.

J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960. URL `http://dl.acm.org/citation.cfm?`

id=367199.

P. Naur. Programming Languages, Natural Languages, and Mathematics. *Commun. ACM*, 18(12): 676–683, Dec. 1975. ISSN 0001-0782. doi: 10.1145/361227.361229. URL http://doi.acm.org/10.1145/361227.361229.

J. Noble, R. Biddle, and E. Tempero. Patterns as Signs: A Semiotics of Object-Oriented Design Patterns. *Systems, Signs and Actions*, 2(1):3–40, 2006. URL http://www.sysiac.org/uploads/2-2-Noble.pdf.

L. Notarfrancesco and L. Caniglia. MathMorphs: An Environment for Learning and Doing Math. Technical report, University of Buenos Aires, Argentina, 2000.

S. Papert. *Mindstorms: Children, computers, and powerful ideas.* Basic Books, Inc., 1980. URL http://dl.acm.org/citation.cfm?id=1095592.

J. Pearl. *Bayesian networks: A model of self-activated memory for evidential reasoning.* University of California (Los Angeles). Computer Science Department, 1985.

C. S. Peirce. *Collected Papers.* Cambridge: Harvard University Press, 1931.

T. Petricek. Programming language theory: Thinking the unthinkable. *PPIG 2016*, Sept. 2016. URL http://tomasp.net/academic/drafts/unthinkable/unthinkable-ppig.pdf.

M. Plummer and others. JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling. In *Proceedings of the 3rd international workshop on distributed statistical computing*, volume 124, page 125. Vienna, 2003. URL http://www.ci.tuwien.ac.at/Conferences/DSC-2003/Drafts/Plummer.pdf.

M. Priestley. *A science of operations: machines, logic and the invention of programming.* Springer Science & Business Media, 2011. URL https://books.google.com/books?hl=en&lr=&id=uflV0_q-FEUC&oi=fnd&pg=PR3&dq=A+science+of+operations:+machines,+logic+and+the+invention+of+programming.&ots=5JkZzOLwWI&sig=zOnLW0dwBgn2LRQmy76kVIl25vU.

S. F. Railsback and V. Grimm. *Agent-based and individual-based modeling: a practical introduction.* Princeton university press, 2011. URL https://books.google.com/books?hl=en&lr=&id=tSI2DkMtoWQC&oi=fnd&pg=PP2&dq=Agent-Based+and+Individual-Based+Modeling+:+A+Practical+Introduction&ots=dQ7ZYA4NYF&sig=A3hphAvIIfq67h5NwSosQV-xgrI.

D. Ritchie, A. Stuhlmüller, and N. D. Goodman. C3: Lightweight Incrementalized MCMC for Probabilistic Programs using Continuations and Callsite Caching. *arXiv:1509.02151 [cs]*, Sept. 2015. URL http://arxiv.org/abs/1509.02151. arXiv: 1509.02151.

S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In *ACM SIGPLAN Notices*, volume 48, pages 447–458. ACM, 2013. URL http://dl.acm.org/citation.cfm?id=2462179.

H. Shen. Interactive notebooks: Sharing the code. *Nature News*, 515(7525): 151, Nov. 2014. doi: 10.1038/515151a. URL http://www.nature.com/news/interactive-notebooks-sharing-the-code-1.16261.

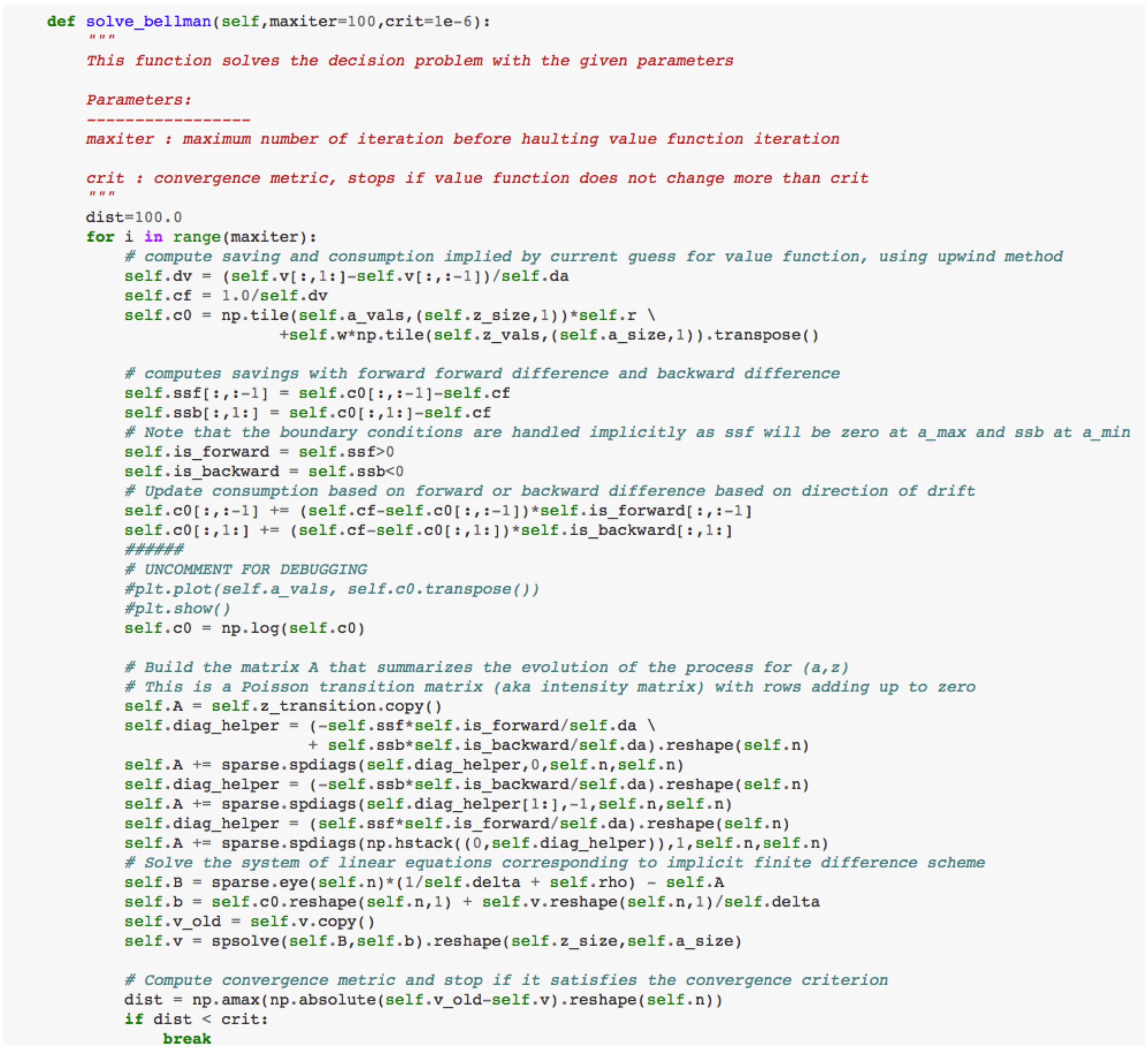
K. Tanaka-Ishii. *Semiotics of programming.* Cambridge University Press, 2010. URL https://books.google.com/books?hl=en&lr=&id=irizHa1MXJoC&oi=fnd&pg=PR5&dq=Semiotics+of+Programming&ots=qM4TCdH_we&sig=9eEiXuNNPGzYnMFDl33cXS_Uddk.

S. Turkle and S. Papert. Epistemological pluralism: Styles and voices within the computer culture. *Signs: Journal of women in culture and society*, 16(1):128–157, 1990. URL http://www.journals.uchicago.edu/doi/pdfplus/10.1086/494648.

M. M. Waldrop. *The Dream Machine: J.C.R. Licklider and the Revolution That Made Computing Personal.* Penguin USA (P) C/O Penguin Putnam Incorporated, 2002. ISBN 978-0-14-200135-6. Google-Books-ID: 7HpQAAAAMAAJ.

H. Wilkinson. How Smalltalk affects Thinking. In *Smalltalks 2014*, Cordoba, Argentina, 2014. URL `https://www.youtube.com/watch?time_continue=15&v=brdx8YAVZag`.

Appendix A. Complete solve_bellman from Quant**Econ**



```
def solve_bellman(self,maxiter=100,crit=1e-6):
    """
    This function solves the decision problem with the given parameters

    Parameters:
    -----------------
    maxiter : maximum number of iteration before haulting value function iteration

    crit : convergence metric, stops if value function does not change more than crit
    """
    dist=100.0
    for i in range(maxiter):
        # compute saving and consumption implied by current guess for value function, using upwind method
        self.dv = (self.v[:,1:]-self.v[:,:-1])/self.da
        self.cf = 1.0/self.dv
        self.c0 = np.tile(self.a_vals,(self.z_size,1))*self.r \
                        +self.w*np.tile(self.z_vals,(self.a_size,1)).transpose()

        # computes savings with forward forward difference and backward difference
        self.ssf[:,:-1] = self.c0[:,:-1]-self.cf
        self.ssb[:,1:] = self.c0[:,1:]-self.cf
        # Note that the boundary conditions are handled implicitly as ssf will be zero at a_max and ssb at a_min
        self.is_forward = self.ssf>0
        self.is_backward = self.ssb<0
        # Update consumption based on forward or backward difference based on direction of drift
        self.c0[:,:-1] += (self.cf-self.c0[:,:-1])*self.is_forward[:,:-1]
        self.c0[:,1:] += (self.cf-self.c0[:,1:])*self.is_backward[:,1:]
        ######
        # UNCOMMENT FOR DEBUGGING
        #plt.plot(self.a_vals, self.c0.transpose())
        #plt.show()
        self.c0 = np.log(self.c0)

        # Build the matrix A that summarizes the evolution of the process for (a,z)
        # This is a Poisson transition matrix (aka intensity matrix) with rows adding up to zero
        self.A = self.z_transition.copy()
        self.diag_helper = (-self.ssf*self.is_forward/self.da \
                            + self.ssb*self.is_backward/self.da).reshape(self.n)
        self.A += sparse.spdiags(self.diag_helper,0,self.n,self.n)
        self.diag_helper = (-self.ssb*self.is_backward/self.da).reshape(self.n)
        self.A += sparse.spdiags(self.diag_helper[1:],-1,self.n,self.n)
        self.diag_helper = (self.ssf*self.is_forward/self.da).reshape(self.n)
        self.A += sparse.spdiags(np.hstack((0,self.diag_helper)),1,self.n,self.n)
        # Solve the system of linear equations corresponding to implicit finite difference scheme
        self.B = sparse.eye(self.n)*(1/self.delta + self.rho) - self.A
        self.b = self.c0.reshape(self.n,1) + self.v.reshape(self.n,1)/self.delta
        self.v_old = self.v.copy()
        self.v = spsolve(self.B,self.b).reshape(self.z_size,self.a_size)

        # Compute convergence metric and stop if it satisfies the convergence criterion
        dist = np.amax(np.absolute(self.v_old-self.v).reshape(self.n))
        if dist < crit:
            break
```

Figure 2. Complete implementation of the method solve_bellman based on Achdou et al. [2014].