3. Understanding and Analysing Science's Algorithmic Regimes: A Primer in Computational Science Code Studies

Gabriele Gramelsberger, Daniel Wenz, and Dawid Kasprowicz

Abstract

Developing and using of software has become an increasing factor in the scientific production of knowledge and has become an indispensable skill for research scholars. To examine this algorithmic regime of science, new methodological approaches are needed. We present our method of computational science code studies (CSS), which focuses on the written code of software, and introduce two software tools we have developed to analyse data structures, code layers, and code genealogies. In a case study from computational astrophysics we demonstrate how the translation from mathematical to computational models in science influences the way research objects and concepts are conceived in the algorithmic regime of science. We understand CSS as a method for science studies in general.

Keywords: scientific programming; software; science studies; philosophy of science; code analysis

Introduction

Science has increasingly become an endeavour that takes place *in front of* and *in* computers. The development of computer-based simulations, the impact of software in science, big data analysis, and the arrival of machine learning (ML) methods have provided a new way of doing science and producing scientific knowledge that we call the "algorithmic regime of science." In disciplines like particle physics, geology, or molecular biology, the practice of scientific programming and in general the usage of

Jarke, J., B. Prietl, S. Egbert, Y. Boeva, H. Heuer, and M. Arnold (eds.), *Algorithmic Regimes: Methods, Interactions, and Politics*. Amsterdam: Amsterdam University Press, 2024
DOI 10.5117/9789463728485_CH03

computational methods has become an essential part of everyday work. With computational methods, we mean approaches that not only enhance computing power but generate both new theoretical and experimental knowledge. Herein, programming as a scientific practice represents the connecting link between data, models, and the results of computer-driven simulations as visualizations on the screen. For scholars from philosophy of science and science and technology studies (STS), this ongoing growth of an algorithmic regime of science poses methodological challenges. How can we describe the impact of computational methods in scientific disciplines? How do scientists change their understanding of theories and models due to new practices like scientific programming and data-driven methods? Are there tensions or transmissions between approved scientific practices and computational methods that demand new skills of the scientists?

However, in the philosophy of science most of the questions about the status of computational science deal with epistemological issues. There is a vibrant discussion about the ontological status, in particular, of computer-based simulation: Is simulation "experimenting with theories" or is it another and autonomous form of knowledge production (Dowling, 1999; Gramelsberger, 2010; Winsberg, 2010)? Is simulation- and ML-based knowledge production transparent and reproducible or is its epistemic status "opaque" (Humphreys, 2004; Lenhard, 2019)? The discussions around these epistemological issues barely reach a methodological dimension. We argue that a methodological reflection is necessary, not only for the philosophy of science but for science studies in general.

To do so, we will focus here on scientific code as our primary research object. We call our approach "computational science code studies" (CSS). Our central thesis is that scientific code is more than merely another scientific tool of knowledge production. We conceive programming in science as a complex translation from classical mathematical to computational models[1] that consist of two elements: the material basis of code and computational statements.[2] Understanding and analysing science's algorithmic regimes from the perspective of the philosophy of science as well as STS requires

1 With "classical mathematical models" we mean models that are based on differential equations, while "computational models" are based on numerical simulations. The transition from one to the other is initiated when classical models are applied to complex situations that result in equations that cannot be solved analytically. This problem is solved by doing numerical simulations of those equations. These simulations are then the only thing that remains visible in the code. For historical details of this development cf. Gramelsberger, 2010, pp. 33–36.

2 We call the code in general, including the comment lines, the material basis of algorithmic regimes in science. The specific portions of the code that function as statements can be called

new methods and practices to explore the material basis and the execution of code but also the practices and politics which come along with science's algorithmic regimes. While ML methods—expanding and transgressing big data analytics—are currently under exploration in science, computer-based simulations have become a well-established and standardized algorithmic regime for science and technology.

We begin with general reflections about the transformation of scientific concepts into the computational from the point of view of the philosophy of science. We continue this train of thought by conducting a review of past and current methods for studying code in science and cultural studies. This leads to the general idea of CSS: Reading the actual code of scientific projects to extrapolate its scientific content and prepare it for an analysis that is able to keep track of the interweaving of science and programming practices. In this context, we introduce the Isomorphic Comment Extractor (ICE) and the General Isomorphic Code Analysis Tool (GICAT), two code analysis tools currently in development at the CSS Lab of the Chair of Theory of Science and Technology at RWTH Aachen University in Germany. Both tools have been designed to analyse different layers of code (comments, hierarchies, imports, or dependencies) and different temporal stages in the evolution of scientific code.

We illustrate the range of application of these tools with a case study of computational astrophysics. This case study also functions as a primer for exploring the material basis of science's algorithmic regimes and thereby to further illustrate our approach, CSS: We demonstrate how shifting between layers and genealogies of code enables science studies scholars to examine how concepts, measurements, and parameters are transformed with regard to the computational model. As translation processes never copy a model but render it in a different way, we ask with the help of our tools for the reconfiguration of scientific concepts and computational statements in the diverse layers of code. We show that with CSS, a new way of accessing scientific programming as a research object is provided that has yet only been treated marginally. This method of analysing scientific code should be useful for other science studies scholars as well to everybody who has to deal with challenges posed by programming practices that are often hard to examine. We therefore understand our method as combinable and extensible with other approaches from science studies.

its ideal basis, as they set up the translation of the mathematical formulations for the *execution* of software code.

The Formation and Transformation of Scientific Concepts into the Computational

Computers developed from being merely auxiliary tools in scientific endeavours to being essential parts of the practice of scientific research itself. This has led to a transformation of classical scientific methods with their clear-cut distinction between theory and experiment into something that is governed to an increasing degree by algorithmic regimes. An important step in this process is the translation of classical mathematical models into computational models consisting of computable statements. This means that in many cases the mathematical modes of description employed in theories switch from more direct forms of representation like differential equations or statistical methods to numerical simulations. As most of the concepts in science are defined or at least strongly dependent on their articulation by mathematical means, it is hard to imagine that this transformation process does leave the underlying scientific concepts unchanged. Therefore, the following questions arise in the context of CSS: How can we identify existing scientific concepts in the web of statements? How can we track changes of scientific concepts that are due to modifications in the code? Do new scientific concepts arise out of the practice of scientific coding?

The transformation of a scientific concept can be understood as an answer to a specific "problem situation" (Nersessian, 2001). According to this idea, concepts "arise from attempts to solve specific problems, using the conceptual, analytical, and material resources provided by the cognitive–social–cultural context in which they are created" (Nersessian, 2008, p. ix). Such new concepts are in most cases not really new; they are transformations of existing concepts, whereby this transformation can be seen as the integration of existing conceptual mechanisms into a new problem situation. The transformation of scientific concepts in computational sciences can be seen as such a "problem situation." The problems to be mastered are not purely inner-theoretical (like problems of consistency) or primarily caused by empirical data; they are brought about by a change of the very medium in which science is conducted. To understand what is at stake here, let's look briefly at the development of the contemporary framework that determines what a scientific concept is.

According to a now classical point of view in the philosophy of science, the meaning of a scientific concept is defined by its role in a theory (Poincaré, 2017; Duhem, 1914; Feyerabend, 1962). This picture implies two main sources for the change of the content of scientific concepts: The first consists of permanent modifications of a theory, and the second of temporary

modifications of some aspects of the theory to make it applicable to a specific situation. The latter kind of modification concerns parametric modifications of parts of the theory in experiments and in real-world applications. Here, the concepts prove themselves by predicting or bringing about specific outcomes from a set of given starting conditions. However, the starting conditions and the outcomes are always interpreted and evaluated in the context of the respective theory. Three developments have undermined this classic perspective.

First, the clear distinction between theory and experiment according to which theory leads and the experiments follow (cf. Popper, 1959) became blurred. This was not (only) done by an intricate philosophical argument but by analysing actual scientific practice (Hacking, 1983, 149ff.). The second development was that the propositional or syntactic view of theories (viewing a theory as a set of axioms) (Carnap, 1937; Hempel, 1965) was gradually replaced by the semantic view. According to the latter, scientific theories are first and foremost models (Suppes, 1960; Van Fraassen, 1980). The idea is that instead of seeing a specific scientific concept determined by one specific theory (implicitly defined by a set of axioms), the content of such a concept can be grasped through the sum of the models it figures in (i.e., the "family resemblance" of the operators that represent it in the respective models) (cf. Van Fraassen, 1980). Based on this picture, scientific concepts, which at the beginning of the 20th century were conceived as paradigms of unambiguity and exactness, became to be seen as evolving entities that not only secure and handle accumulated knowledge, but through their flexibility open up the path for new investigations (Wilson, 2006; Brandom, 2011; Bloch-Mullins, 2020). Third, with the rise of the computer model in science, the content of scientific concepts is spread even further apart. One of the most pressing problems is the translation of mathematical models as used in the semantic view of theories into numerical (computable) models. In more complex cases it is not even clear if the numerical model really instantiates the mathematical model of the underlying theory (Gramelsberger, 2011).

All this can be expected to lead to repercussions on the level of the scientific concepts expressed by the theory. In extreme cases the development of the mathematical model and the development of the computer model can split up into different projects that only occasionally interact. The decoupled development of the computer model can rather be understood as an ongoing series of experiments in silico than as a case of classical model building. In this way, the technical aspects can come to the fore: Modifications that are motivated by purely application-oriented considerations can infiltrate tacitly the core of the model. Difficulties for the tracking of scientific concepts in

a web of statements range from unclean coding by the individual scientist to the modularity of modern-day programming and the traceability of the different layers of execution in the code. However, from a well-documented piece of software one can potentially reconstruct more references and cross-references than from a classical scientific paper.

Programming as a Research Object in Science and Software Studies

The rising significance of software in the 21st century resulted in new subfields like software studies (Manovich, 2001; Fuller & Goffey, 2016), leading also to an increased attention on algorithms in the last fifteen years (Kitchin & Dodge, 2011; Christin, 2020; Marino, 2020). Thus, scholars from software studies and STS have dealt with the question of how to access the practices of programming. One important and early claim by software studies was to make software visible and to detach it from the idea of a neutral and functional tool (Chun, 2004). Software—and therefore programming practices—had an impact on people, professions, and institutions (Mackenzie, 2006; Chun, 2011). But software has also been shaped by social relations, it was therefore more a socio-technical object than merely a technical tool. This necessity of making software visible became even more urgent with the technical problems of archiving since older software also needs a special hardware and an operating system that are not always archived as well (Chun, 2011, p. 3; Mahoney, 2008). While these cultural and historical approaches highlighted the impacts of software and algorithms (Seaver, 2017), recent STS works pay attention to the practices of programming and the "dulled and expanse fading of ever evolving bodies of code" (Cohn, 2019, p. 423). This shift of attention from the invisibility of software to the everyday actions of programming comes along with the use of ethnographic methods to follow the software. Following up on Ian Lowrie's statement that no one can directly observe an algorithm since it is always a by-product of multiple social actions and agents (Lowrie, 2017, p. 7), STS scholars use ethnographic methods to lay open not only the dynamics of programming but also the intentions and expectations that arise throughout the development of software. This shift is important with regard to scientific programming, since it raises the question how the practice of programming and the way scientists think of their own concepts and models reciprocally impact each other. As Adrian Mackenzie has shown for the field of machine learning software, the increasing use of statistical computer models in science leads

to a state of ongoing testing of predictions as statistical hypotheses—a mode of reasoning he referred to as a "regime of anticipation" (Mackenzie, 2013, p. 393). Further research would have to examine for different disciplines how such "regimes of anticipation" influence the scientific understandings of prediction and probability in the algorithmic regime of science.

Ethnographic methods with qualitative interviews have also been widely used in the social studies of science. Considering, as we argue, the shift to algorithmic regimes of science, a crucial question is the relation of developers and users of code since not every scientist who works with computational methods must be a programmer. As Kuksenok et al. have shown in a qualitative analysis of four oceanographic research groups, the relation of users and developers of scientific code can be summed up in three different groups: (1) Scientists who code, (2) computer scientists who develop code and tools for scientists, and (3) scientific programmers (Kuksenok et al., 2017, p. 665; see also Sundberg, 2010). A methodological challenge for the social studies of science as well as for CSS represents the possible blurring of these distinctions in each discipline (Kelly, 2015; Edwards, 2010). Scientists learn how to program, and they extend their programming skills due to new programming languages like Python, e.g., which has become a widely used language in the natural sciences (Storer, 2017). Additionally, cultures of scientific programming change as well. The availability of libraries in Python, but also the possibility for scientists to add new libraries, was one reason for the popularity of Python in natural sciences. However, functional programming, which has often been used in scientific programming languages like Fortran (Suzdalnitskiy, 2020), is not associated with Python in the first place, although it can be implemented. These developments in scientific programming cultures from functional statements to more and more library-oriented languages have yet to be investigated.

As we will see in the forthcoming sections, tools like GICAT offer here a kind of meta-perspective on scientific programming that enables us to analyse how the translation process of the scientific into the computational model has been exercised in the code. To do that, solid knowledge of the scientific project is needed, especially of the models and the data sets that are used.

Software Tool Development for CSS

Getting access to the *material basis and the execution* of science's algorithmic regimes (computer code of the computational model/scientific

computer program) is less an issue of code protection than of the complexity and magnitude of scientific computer programs. For example, an atmosphere model in climate science from 2003 consists of a web of statements of 15,891 declarative and 40,826 executable statements written down in 65,757 code lines of the programming language Fortran90 accompanied by 34,622 comment lines (Roeckner, 2003). The scientific computer program xgaltool, which we will take a closer look at in the next sections, consists of a web of statements of 46 classes and 313 definitions in 9,213 lines of the programming language Python, including comment lines (https://gitlab.obspm.fr/dmaschmann/xgaltool). Furthermore, philosophers as well as researchers from STS usually lack programming skills and expertise. Thus, conducting computational science code studies is not a simple task. How can we make the study of computational sciences more accessible? We argue that one necessary step to answer this question consists in programming software tools designed to facilitate case studies on computational sciences in the subfield of code studies (Schüttler, Kasprowicz, & Gramelsberger, 2019). Our aim is to develop a toolbox for scientific code study based on four rules:

1. File structure isomorphism; i.e., under all circumstances preserving the file structure of a scientific computer program while analysing it, because even in object-oriented programming languages the ordered structure of files is meaningful. Thus, such an isomorphism guarantees structural identity with the scientific program as intended by the scientific programmer.
2. Modularity; i.e., based on the file structure isomorphism we are building up a hierarchy of ever more complex tools. Each tool can be used separately (e.g., Isomorphic Comment Extractor, or ICE), but can also be combined to a CSS toolbox for computational science code study.
3. Visual depth; i.e., the ability to zoom in and out of the structural layers of a program. On the top level only the file structure becomes visible, while zooming in unveils the class structure, its functions, and finally the code and comment lines.
4. Analysis filters; i.e., depending on the specific aim of an analysis a toolbar of filters is increasingly developed, which can be turned on and off in order to analyse scientific computer programs like xgaltool.

While file structure isomorphism, modularity, and visual depth help to organize access to the complex and vast body of scientific code, the analysis filters are doing the job of code analysis from a philosophy of science and

STS perspective. It is obvious that conceiving and successfully implementing interesting analysis filters is basic and ongoing research in CSS.

Case Study of Computational Astrophysics

Computational astrophysics provides interesting examples for a study of a specific algorithmic regime. By the 1970s the use of computers had shifted astronomy from observing the sky by using telescopes (empirical regime) to data visualization analysing images of the sky (representational regime) (Daston & Galison, 1992). Since the 1990s the use of CCD (charge-coupled device) chips in telescopes has shifted astronomy into a data-driven science by generating masses of photometric data (algorithmic regime) (Hoeppe, 2014). CCD chips in cameras not only produce images of the sky, but act as sensors for specific wavelengths of light. Thus, instead of "subjectively" analysing the sky and images of the sky, respectively, analysing data sets with algorithms "objectively" has become central for today's astronomy. However, if the algorithms are as objective as scientists claim is one of the interesting research topics in CSS by analysing the interpretative concepts like threshold settings of a scientific computer program.

One of these computational astrophysicists is Daniel Maschmann, who worked for one year at our Computational Science Studies Lab (CSS Lab) in Aachen, Germany, before he moved in 2019 to the Observatoire de Paris and the Sorbonne Université to start his PhD project. Since 2017, the CSS Lab is located at the Chair for the Theory of Science and Technology at RWTH Aachen University (www.css-lab.rwth-aachen.de) and is devoted to developing concepts, methods, and software tools for studying science's algorithmic regimes, in particular, the material basis of computer code, for example, tools like the Isomorphic Comment Extractor (ICE) or the General Isomorphic Code Analysis Tool (GICAT). Daniel Maschmann used early versions of our CSS tools in order to improve his computer program xgaltool, which he had first programmed for his MA thesis (https://gitlab.obspm.fr/dmaschmann/xgaltool; Maschmann et al., 2020). Xgaltool is an open-source computer program developed on GitLab for detecting merging galaxies in the Reference Catalog of galaxy Spectral Energy Distributions (RCSED)—a huge database containing photometric data on energy distributions of 800,299 galaxies in 11 ultraviolet, optical, and near-infrared bands. These photometric data result from CCD camera-equipped telescopes. CCD telescopes were developed in the early 1990s to conduct the Sloan Digital Sky Survey (SDSS) at the Apache Point Observatory in New Mexico—a gigantic endeavour to

scan one-third of the sky. Thus, the RCSED selects data from the SDSS for the spectral energy distribution. Furthermore, the RCSED data decompose the measured light into two components: the light emitted by the stars and the light of the galaxies' gas content, which is described by emission lines.

So-called double peak (DP) emission line galaxies have been extensively explored, because this type of galaxy can be an indication of a galaxy merger. A galaxy merger can occur when galaxies collide. The galaxy merger is one of the states of the evolution of galaxies, as classified by Edwin Hubble in 1926. Astrophysicists are still trying to understand how galaxies and stars form. Today they use computer-based simulation as well as indirect evidence from photometric data. DP emission line galaxies are relevant to empirically inspired galaxy evolution theory as they mostly consist of star-forming galaxies and "the star formation rate (SFR) of galaxies is a well-suited diagnostic to characterize their evolutionary state" (Maschmann et al., 2020, p. 1). Thus, what Daniel Maschmann was seeking with his xgaltool were DP emission line galaxies, whose emission line displays in a characteristic shape in the RCSED data. However, these galaxies are rare and represent only 0.8% of the RCSED data (Maschmann et al., 2020, p. 1). Thus, Daniel Maschmann calibrated xgaltool to the specific emission lines as following:

> We developed an automated three-stage selection procedure to find DP galaxies. The first stage pre-selects galaxies with a threshold on the S/N, and performs successively the emission line stacking, line adjustments and empirical selection criteria. Some emission lines are individually fitted at the second stage to select first DP candidates. We also selected candidates showing no DP properties to be the control sample (CS).... At the third stage, we obtained the final DPS using the fit parameter of each line. (Maschmann et al., 2020, p. 2)

From this cryptic quote the computational model for his xgaltool algorithm can be inferred. S/N describes a ratio between S (signal) and N (noise), which enables a classification of galaxies. For $S/N < 10$, 276,239 galaxies were selected from the RCSED, for $S/N < 5$ only 189,152 galaxies. Within the latter data sample complicated filtering methods were applied in order to reduce the number of selected emission lines ≥ 3 for 89,412 galaxies for the control sample. Reducing the number of galaxies further led to 7,479 interesting DP candidates. Finally, stage three sorted the emission lines of the 7,479 interesting DP candidates depending on their S/N ratio into three classes: one DP line (175), two DP lines (269), more DP lines (5,219). "The automated selection procedure selected DP galaxies with an objective

algorithm. This means that we did not need any visual inspection, which would have been a subjective factor in the sample selection" (Maschmann et al., 2020, p. 6).

Based on the selected double peak (DP) emission line galaxies the scientifically interesting part of the work could start by exploiting the shape of the emission lines exhibited in BPT diagrams. BPT diagrams were developed in 1981 by John A. *B*aldwin, Mark M. *P*hillips and Roberto *T*erlevich to classify emission-line spectra (Baldwin et al., 1981).[3] In the case of DP emission line galaxies three types of BPT diagrams were explored, which were based on "the relative intensities of the strongest lines, into groups corresponding to the predominant excitation mechanisms" (Baldwin et al., 1981, p. 16). Thus, types of galaxies are classifiable; for instance, star-forming (SF) galaxies, active galactic nuclei (AGN) galaxies, and composite (COMP) galaxies. An important scientific result was that most DP galaxies are SF galaxies and thus intensively contribute to galaxy mergers. In this way, by analysing the data carefully some indirect evidence could be gained about the role of DP emission line galaxies in the process of galaxy formation (Maschmann et al., 2020). Using algorithms for automatically generated data samples of the rare DP emission line galaxies, the astrophysicists provide a software- and statistics-based method to detect galaxy mergers and to classify new morphological types of galaxy formations.

CSS Tools Applied: GICAT and ICE

The above case study provides an example of the algorithmic regime of computational astrophysics. Of course, the scientific concepts involved in xgaltool are quite advanced, combining data analysis methods, filter methods, with many other computationally interpretative methods. For philosophers of science as well as for researchers from STS, it is difficult to grasp how scientific research is conducted under algorithmic regimes. This is simply because observational access to code is difficult. Making such code accessible is an important part of CSS, and the tools we develop are an integral part of this endeavour.

3 The BPT diagrams are based on the fundamental 19th-century discovery that different chemical elements produce different types of spectra, e.g., celestial objects like galaxies emitting gas. Based on emission spectroscopy the wavelengths of photons emitted by excited atoms or molecules of a gas can be measured and classified. For instance, hydrogen is characterized by the Balmer lines (Balmer 1885).

The Isomorphic Comment Extractor (ICE)

Many details about a scientific program can be found in its comments. However, software documentation is more an art than a science. Software documentation in the code is laborious, time-consuming, effectless on the performance of the code itself, standards are missing, and so forth. Nevertheless, in computational sciences the software is the basis of research. Thus, a well-documented code is part of responsible science. In particular, in the course of the open science development the transparency of software has become a major topic (Aghajani et al., 2019).

In programming, comment analysers are known tools, but they are usually restricted to the programming language used by the programmer. Our ICE tool can extract comments from various programming languages such as C++, Python, Java Scrips, and Fortran. Extracting comments (if available) from scientific computer programs provides useful insights into the scientific process behind the coding. By "throwing" a scientific computer program in the ICE tool one can easily analyse it in an isomorphic mode the story unveiled by the comments of a well-documented software code.

The General Isomorphic Code Analysis Tool (GICAT)

It is a far more complex endeavour to analyse the execution of a scientific computer program exhibited in the web of statements. To give an example: xgaltool file analysis_tools.py alone consists of eight classes and each class consists of several definitions. For instance, the class EmissionLineTools contains 19 definitions, among these the following:

In Python a function is defined using the *def* keyword (Figure 3.1, line 43) followed by arguments and parameters inside the parentheses. Arguments and parameters pass information into a function. With *r* (line 44) and *return* (line 62), for instance, control flow is organized in Python, i.e., calculations are performed and results are returned. In this case lines 61 and 62 set up the calculation of gas metallicity based on the data called in lines 56 to 58. Python also accepts function recursion, i.e., a function calls itself usually structured by if, else, return loops. Different languages employ different concepts—from variations of the before mentioned to completely different programming paradigms. Based on such programming concepts a web of statements is designed by the scientific programmer forming up the intended behaviour of her/his scientific computer program. Each change in the functionality of the code that modifies its behaviour results in a slightly different computational result. If one is not able to grasp these complex

```
43    def get_gas_met_n2(self, line_shape='gauss'):
44        r"""
45        Calculate gas metallicity following using emission lines [NII] 6585 and Halpha 6565
46        We follow Pettini and Pagel (2004) doi:10.1111/j.1365-2966.2004.07591.x
47
48        :param line_shape: Specified the emission-line fit. For the ACSED catalogue (Chilingarian et al.2017) this
49        can be 'gauss' or 'nonpar' for a gaussian or a non-parametric line approximation respectively.
50        For Maschmann et al. (in prep.) this is dedicated to the double gaussian fit and denotes the blue-shifted
51        ('peak_1') or red-shifted component ('peak_2').
52        :type line_shape: str
53
54        :return metallicity in [Z/H] dex
55        """
56        # get emission line fluxes
57        flux_nii = self.get_emission_line_flux(line_wavelength=6585, line_shape=line_shape)
58        flux_h_alpha = self.get_emission_line_flux(line_wavelength=6565, line_shape=line_shape)
59
60        # Pettini and Pagel (2004) Eq. (2)
61        log_n2 = np.log10(flux_nii / flux_h_alpha )
62        return 9.37 + 2.03 * log_n2 + 1.26 * (log_n2**2) + 0.32 * (log_n2**3)
```

Figure 3.1. Lines 43 to 62 of the analysis_tools.py of xgaltool. Courtesy: Daniel Maschmann.

interactions in the code, one is not able to recognize how the concept of metallicity is articulated in it. Therefore, understanding the functionality and its execution over time is crucial for CSS. Analysing, but also following, the development of a scientific computer program, i.e., carrying out a code genealogy, provides insights into changing scientific concepts.

Following these considerations, we started to develop the General Isomorphic Code Analysis Tool (GICAT). GICAT visualizes different layers of execution of a given software project. From class and inheritance structures in object-oriented languages to complex functional interdependencies in functional programming, GICAT can help to identify and disentangle the scientifically significant layers and threads in the web of statements of a given code. GICAT is not limited to Python. Like and in accordance to ICE, it supports different languages under different programming paradigms. To visualize different layers of execution in a web of statements, GICAT works with a set of preconfigured as well as free-definable analysis filters. The preconfigured filters give the user the means to orient herself in the code and to identify the scientific relevant structures on different levels. Free-definable filters are powerful tools that enable the experienced user to make out where the relevant threads and layers of scientific code condense to a structure that encodes more specific points of interest (especially in the deeper analysis of scientific concepts).

The preconfigured filters are automatically adjusted to the programming language of the targeted software project. We can illustrate how they give a first overview by applying GICAT to xgaltool (Maschmann et al., 2020), which gives us a general idea of the structure of the program. Figure 3.2 depicts the global structure of xgaltool via its class relations, the standard filter set for Python projects. In this context we show the project at two different stages. Comparing the structure from 15.06.2021 to the structure of 23.02.2022, we see that a connection between two classes (EnvironmentTools and PlotBPT)

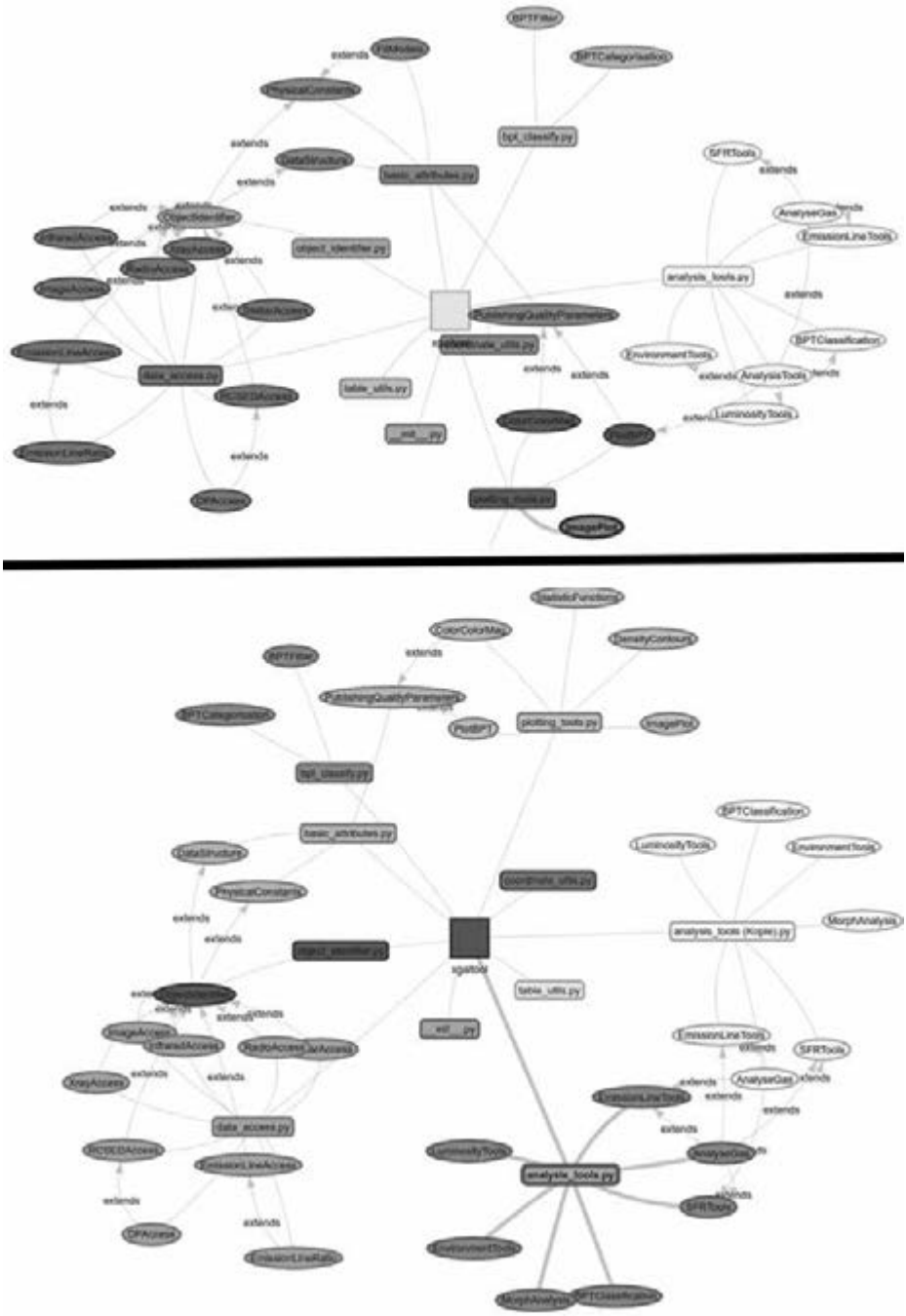

Figure 3.2. GICAT view on xgaltool visualized under the class filter of 15.06.2021 and 23.02.2022, in order to study class relations in two different software versions (code genealogy).

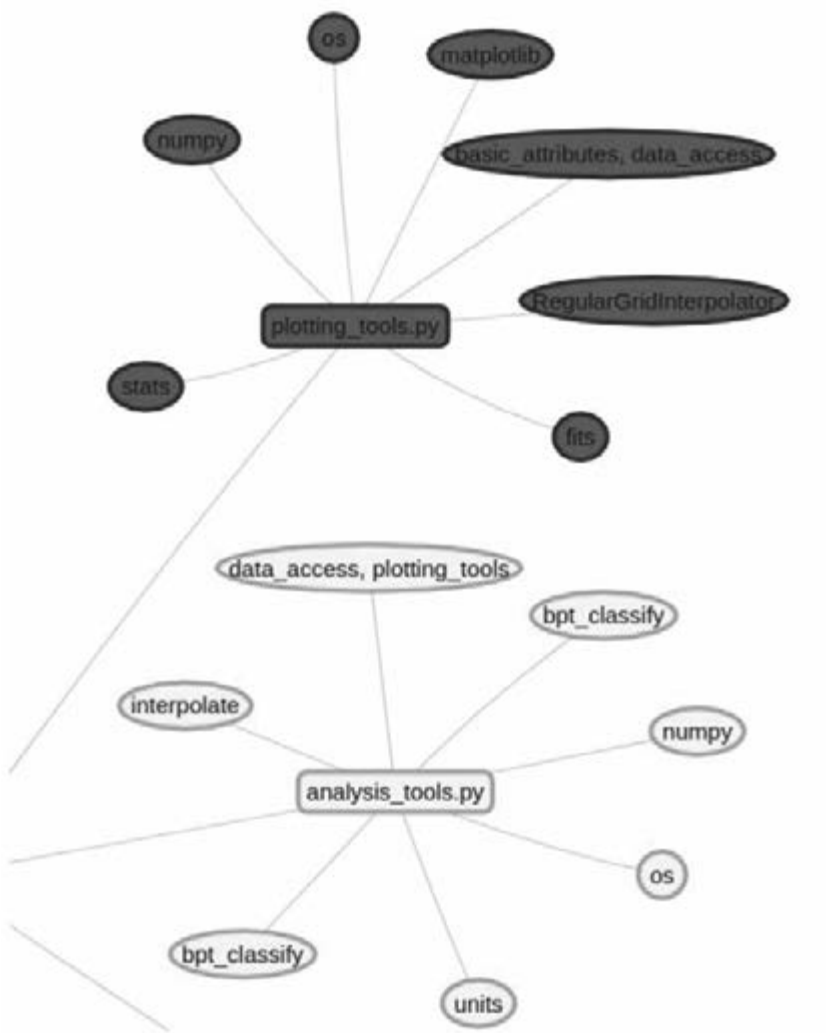

Figure 3.3. GICAT view on xgaltool detail under the library filter.

exists in the 2021 version that does not exist in the 2022 version anymore. This effectively cuts any direct connection between the groups plotting_tools and analysis_tools in the newer version of the program. This illustrates another important feature: GICAT enables the user to do a genealogical analysis, which makes it possible to track the development of different aspects of the scientific structures in the surveyed web of statements over time.

Making relations explicit while being able to place them into the greater picture of a given web of statements can yield important clues for the reconstruction of scientific concepts in a software project. Another example

of an advanced filter is shown in Figure 3.3. Here the imports of modules (libraries, in darker circles) and packages (of libraries, in lighter circles) is shown. This is important because as mentioned above one of the main hindrances of getting a clear picture of the structure of a web of statements is the modularity of contemporary programming. With the help of GICAT the user is able to keep track of the different dependencies and gets a synoptic overview of their overall structure.

Following our idea to create a modular toolbox for scientific code study, these features are complemented by ICE. The option to integrate this performant comment extractor and code viewer into the structure of GICAT gives the user direct access to the corresponding parts of the raw code of the visualized structures. This whole package should allow a smooth transition between the visualization of different layers of execution that are hidden in the web of statements as well as between these layers and the corresponding chunks of raw code. Adding the possibility of genealogical analysis, the user can track and reconstruct the evolution of the implementations of scientific models and concepts in the web of statements of a given software. This concerns the modifications that are consciously made by the developers in respect to the scientific content of their project as well as changes that are motivated by purely technical reasons.

Above we have seen that one of the central concepts in Maschmann et al. (2020) is "emission line." The emission line is what appears in a spectrum depending on what specific wavelengths of radiation a source emits. It is one of the primary sources for the astrophysicist to identify and classify galaxies. To reconstruct how this concept is articulated in a web of statements, we have to look at how it is entangled in its different layers of execution. In this regard, we use a GICAT visualization under the filter that depicts the structure of class inheritances (Figure 3.4). If a class inherits from another, this is represented in GICAT by an extension arrow. We see that the class AnalyseGas inherits from the two base classes EmissionLineTools and SFRTools. SFR stands for "star formation rate," which means the total mass of stars formed per year.

This piques our interest for further analysis, because the emission line is normally used to estimate the SFR, while the analysis of a gas is conducted through an analysis of its emission line. Therefore, although it seems natural that the class AnalyseGas inherits from the class EmissionLineTools (as we analyse gas through an analysis of its emission line), it is interesting that the class AnalyseGas inherits from the class SFRTools (as the SFR is estimated through the analysis of a gas via an analysis of its emission line). We have uncovered an important clue how the concept "emission line" is entangled in and articulated by the different layers of the given web of statements.

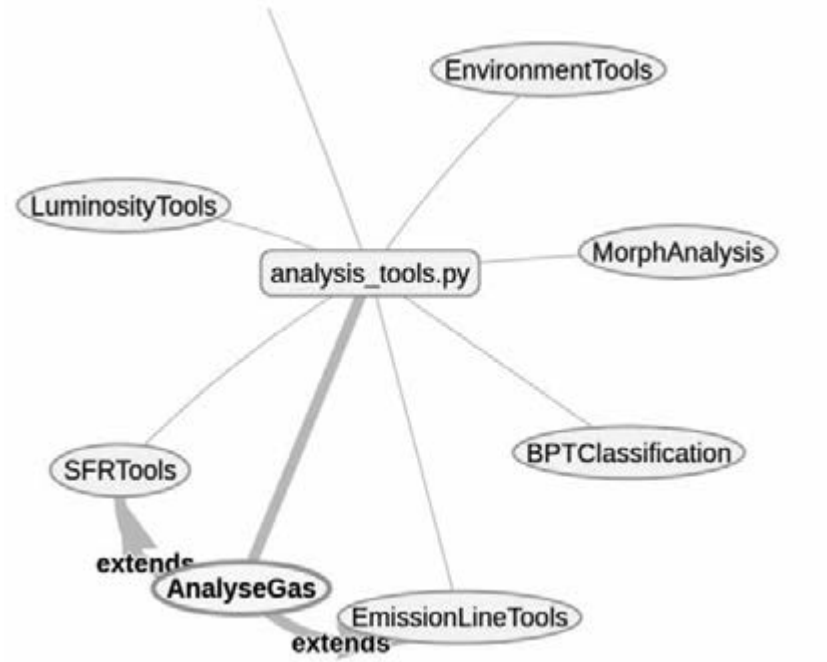

Figure 3.4. Visualization of xgaltool (23.02.2022) under the class-inheritance filter, close up "EmissionLineTools."

Guided by this, we can go on by looking into the relevant code, using ICE, the integrated code viewer, and the comment extractor. Alternatively, or complementary, we could dive deeper into the entangled layers of execution by using a filter that visualizes functional connections and dependencies or explore the structure of libraries our target draws on.

Discussion and Outlook

We have argued that the increasing use of computer simulations in science will reinforce the necessity for science studies to create new methodological approaches. As our case study from astrophysics illustrated, software like xgaltool needs to be analysed to explain the translation of a mathematical into a computational model and the decisions that have to be made during this programming process (as shown by the emission lines with the help of GICAT in 3.4). It should be emphasized that this kind of analysis is not limited to any specific programming paradigm and is also applicable to (seemingly) "indirect" approaches like ML. Machine learning—especially in

the context of scientific code—does not happen in a void. Its more specific procedures or preconfigured setups (i.e., a trained neural network) are always embedded in an encompassing architecture (which comprises things like an overarching program, a concrete experimental setup, etc). Reconstructing the scientific relevance of the ML-component consists then (as for any other component) primarily in reconstructing its role in this architecture. For a preconfigured ML setup it may be necessary to look at external sources. If the training is part of the running implementation (like in a program for speech recognition that adjusts itself to its user), then the learning algorithm and the path of the ongoing flow of data can be analysed directly. For such studies in the algorithmic regime of science, our tools offer modes of navigation through file structures and filters for code genealogies to make traceable what changes in the code have occurred, at what time of the project, and conducted by who. As shown in 3.2, this helps also to illustrate the modifications and decisions the scientists had to make during the programming process. These might be routine for scientists who program every day, but for CSS, STS, and also social studies of science, the different ways scientists are influenced recursively by their programming language and the standards how to use it post further research questions.[4] In this sense, how does the shift in scientific programming from functional statements to more and more library-oriented languages like Python influence the theoretical concepts and models of scientific projects? How do these programming practices change the expectations of scientists and their ways to make predictions and classify objects like galaxy mergers?

As mentioned before, our code-oriented approach and the tools we develop do not present the only way to explore algorithmic regimes in science. Additionally, to our perspective from the philosophy of science, methods from STS and the social studies of science can be complementary since both try to describe the role of software in knowledge production as well as the dynamics of scientific programming. Ethnographic methods and tool analysis can serve as in-depth and meta-perspectives, providing ways to zoom into the daily (and dull) work of coding and to zoom out to keep track of longer code genealogies. However, there are still some problems regarding the methodological solutions provided so far. First, it is difficult to generalize from single case studies since coding practices even in one and the same scientific discipline are not yet standardized. We

4 See also Kelly (2015) for a comparative approach from software engineering where the characteristics of scientific programmers are compared to guidelines of programming in software engineering.

lack categories and concepts to describe the dynamics in the translation process from knowledge-based scientific to computational models over different disciplines. Second, for science studies scholars not familiar with programming, it is difficult to see how efficiently or how messy the program has been written. What we have shown here is how we can access new artefacts of scientific programming via software tools for non-programming experienced science studies scholars. We have argued that theses artefacts (comment extractions, code genealogies, visualizations of inheritances) allow us to study the question how scientific concepts and models are integrated into computational models via the practices of scientific programming. In this sense, our tools enable the user to identify and study the scientific part of the code and therefore permit to examine the impact of software on the production of knowledge in the algorithmic regime of science.

References

Aghajani, E., Nagy, C., Lucero Vega-Márquez, O., et al. (2019). Software documentation issues unveiled. In *ICSE '19: Proceedings of the 41st International Conference on Software Engineering* (pp. 1199–1210). https://doi.org/10.1109/ICSE.2019.00122

Baldwin, P., Phillips, M. M., & Terlevich, R. (1981). Classification parameters for the emission-line spectra of extragalactic objects. *Publications of the Astronomical Society of the Pacific, 93*(551), 5–19. https://doi.org/10.1086/130766

Balmer, J. J. (1885). Notiz über die Spectrallinien des Wasserstoffs [Note on the spectral lines of hydrogen]. *Annalen der Physik und Chemie, 25*, 80–87.

Bloch-Mullins, C. L. (2020). Scientific concepts as forward-looking: How taxonomic structure facilitates conceptual development. *Journal of the Philosophy of History, 14*, 205–231. https://doi.org/10.1163/18722636-12341438

Brandom, R. B. (2011). Platforms, patchworks, and parking garages: Wilson's account of conceptual fine-structure in wandering significance. *Philosophy and Phenomenological Research, 82*, 183–201. https://doi.org/10.1111/j.1933-1592.2010.00485.x

Carnap, R. (1937). *The logical syntax of language*. Routledge.

Christin, A. (2020). The ethnographer and the algorithm: Beyond the black box. *Theory and Society, 49*, 897–918. https://doi.org/10.1007/s11186-020-09411-3

Chun, W. H. K. (2004). On software, or the persistence of visual knowledge. *Grey Room, 18*(4), 26–51.

Chun, W. H. K. (2011). *Programmed visions: Software and memory*. MIT Press.

Cohn, M. L. (2019). Keeping software present: Software as a timely object for STS studies of the digital. In J. Vertesi & D. Ribes (Eds.), *digitalSTS: A field guide for science & technology studies* (pp. 423–446). De Gruyter.

Daston, L., & Galison, P. (1992). The image of objectivity. *Representations, 40*, 81–128.

Dowling, D. C. (1999). Experimenting on theories. *Science in Context, 12*, 261–274.

Duhem, P. (1914). *The aim and structure of physical theory*. Atheneum.

Edwards, P. (2010). *A vast machine: Computer models, climate data and the politics of global warming.* MIT Press.

Feyerabend, P. (1962). Explanation, reduction, and empiricism. In H. Feigl & G. Maxwell (Eds.), *Minnesota studies in the philosophy of science, vol. 3* (pp. 28–97). University of Pittsburgh Press.

Fuller, M., & Goffey, A. (2016). The obscure objects of orientation. In M. Fuller, *How to be a geek: Essays on the culture of software* (pp. 15–36). Polity Press.

Gramelsberger, G. (2010). *Computerexperimente. Zum Wandel der Wissenschaft im Zeitalter des Computers* [Computer-based experiments: Science in the age of the computer]. Transcript.

Gramelsberger, G. (2011). What do numerical (climate) models really represent? *Studies in History and Philosophy of Science, 42*, 296–302.

Hacking, I. (1983). *Representing and intervening: Introductory topics in the philosophy of natural science*. Cambridge University Press.

Hempel, C. G. (1965). *Aspects of scientific explanation and other essays in the philosophy of science*. The Free Press.

Hoeppe, G. (2014). Working data together: The accountability and reflexivity of digital astronomical practice. *Social Studies of Science, 44*(2), 243–270.

Hubble, E. P. (1926). Extragalactic nebulae. *Astrophysical Journal, 64*, 321–369.

Humphreys, P. (2004). *Extending ourselves: Computational sciences, empiricism, and scientific method*. Oxford University Press.

Kelly, D. (2015). Scientific software development viewed as knowledge acquisition: Towards understanding the development of risk-aversive scientific software. *Journal of Systems and Software, 109*, 50–61. http://dx.doi.org/10.1016/j.jss.2015.07.027

Kitchin, R., & Dodge, M. (2011). *Code/space: Software and everyday life*. MIT Press.

Kuksenok, K., Aragon, C., Fogarty, J., Lee, C. P., & Neff, G. (2017). Deliberate individual change framework for understanding programming practices in four oceanographic groups. *Computer Supported Cooperative Work, 26*, 663–691. https://doi.org/10.1007/s10606-017-9285-x

Lenhard, J. (2019). *Calculated surprises: A philosophy of computer simulation*. Oxford Scholarship Online.

Lowrie, I. (2017). Algorithmic rationality: Epistemology and efficiency in the data sciences. *Big Data & Society, 4*(1). https://doi.org/10.1177/2053951717700925

Mackenzie, A. (2006). *Cutting code: Software and sociality.* Peter Lang.

Mackenzie, A. (2013). Programming subjects in the regime of anticipation: Software studies and subjectivity. *Subjectivity: International Journal of Critical Philosophy, 6*(4), 391–405.

Mahoney, M. S. (2008). What makes the history of software hard? *IEEE Annals of the History of Computing, 30*(3), 8–18. https//doi.org/10.1109/MAHC.2008.55

Manovich, L. (2001). *The language of new media.* MIT Press.

Marino, M. C. (2020). *Critical code studies*. MIT Press.

Maschmann, D., Melchior, A.-L., Mamon, G. A., et al. (2020). Double-peak emission line galaxies in the SDSS catalogue: A minor merger sequence. *Astronomy & Astrophysics, 641*, A171. https://doi.org/10.1051/0004-6361/202037868

Nersessian, N. J. (2001). Conceptual change and commensurability. In H. Sankey & P. Hoyningen-Huene, *Incommensurability and related matters* (pp. 275–301). Kluwer Academic.

Nersessian, N. J. (2008). *Creating scientific concepts*. MIT Press.

Poincaré, H. (2017). *Science and hypothesis*. Bloomsbury. (Original work published 1902).

Popper, K. (1959). *The logic of scientific discovery*. Routledge.

Roeckner, E., et al. (2003). *The atmospheric general circulation model ECHAM5: Model description* (report no. 349). Max-Planck-Institute for Meteorology.

Schüttler, L., Kasprowicz, D., & Gramelsberger, G. (2019). Computational Science Studies. A Tool-Based Methodology for Studying Code. In G. Getzinger (Ed.), *Critical issues in science, technology and society studies: Conference proceedings of the 17th STS Conference Graz 2018, 7th–8th May 2018* (pp. 385–401). Verlag der Technischen Universität Graz.

Storer, T. (2017). Bridging the chasm: A survey of software engineering practice in scientific programming. *ACM Computing Survey, 50*(4), article no. 47. https://doi.org/10.1145/3084225

Sundberg, M. (2010). Organizing simulation code collectives. *Science Studies, 23*(1), 37–57.

Suppes, P. (1960). A comparison of the meaning and uses of models in mathematics and the empirical sciences. *Synthese, 12*, 287–301. https://doi.org/10.1007/BF00485107

Suzdalnitskiy, I. (2020, December 7). These modern programming languages will make you suffer. *Better Programming*.

Van Fraassen, B. C. (1980). *The scientific image*. Clarendon Press.

Wilson, M. (2006). *Wandering significance: An essay on conceptual behavior*. Clarendon Press.

Winsberg, E. (2010). *Science in the age of computer simulation*. University of Chicago Press.

About the Authors

Prof. Dr. **Gabriele Gramelsberger** holds the Chair for the Theory of Science and Technology at RWTH Aachen University in Germany, and is co-director of the Käte Hamburger Kolleg "Cultures of Research." Her research interests include the digital transformation of science; reflective modelling and simulation; and AI in the research and theory of computational neuroscience.

Daniel Wenz is a research assistant at the Chair for Theory of Science and Technology and coordinator of the CSS Lab at RWTH Aachen University in Germany. His research interests include classic epistemology, the theory of science, and the philosophy of mathematics.

Dawid Kasprowicz is a research assistant and fellow coordinator at the Käte Hamburger Kolleg "Cultures of Research" at RWTH Aachen University in Germany. His research fields include philosophy of computational sciences, phenomenology, media theory, and human–robot interactions.