DOI: 10.3217/978-3-85125-668-0-21

Computational Science Studies | A Tool-Based Methodology for Studying Code

Lisa SCHÜTTLER, Dawid KASPROWICZ, G. GRAMELSBERGER

Theory of Science and Technology, RWTH Aachen University, Germany

Abstract

The practice of programming has become a key qualification for scientific research. However, from a STS point of view, general methodologies to access the program code and its semantics are still lacking. We present a new methodological approach called “Computational Science Studies” (CSS). Our main argument is that to get access to the program code on a semantic level, the code has to be considered as a research object. This can be done with software tools that help us to analyse the semantics in the program code. Therefore, we present a software tool, called the Isomorphic Comment Extractor (ICE),[1] to excavate the semantic content by analysing the comments of the code and its data structures. Taking the program code of a geological research project as an example, we argue that the programming practices shape research objects and embody the transition of empirical data into simulations of computational models. Hence, we first introduce the methodology of CSS and then discuss some methodological approaches from the field of Code Studies that have already dealt with computer code as a research object. Then we present the ICE in our geological case study and finally we argue why new software tools in STS are necessary to analyse the impact of computer-driven scientific knowledge.

1. We thank very much Lukas Böhres and Frederic Kerksieck for programming and co-conceptualizing the ICE-Tool.

1 Introduction: Computational Science Studies

In the last twenty years, the research on computer code witnessed an increasing interest from humanities' scholars. With the generally acclaimed importance of algorithms and the impact of computerized models and simulations on scientific expertise, code became an important research object regarding the digitalization of science. This also includes the digitalization of scientific objects in the humanities like manuscripts or fragments, leading to new methods and epistemologies that resulted in the foundation of new disciplines like the Digital Humanities (see for an overview of this process, Jones 2016). However, for sciences like Particle Physics or Climate Sciences, where scientific objects are no longer directly observable, programming turned out to become an essential part of scientific research and study (Gramelsberger 2010 & 2011). Surprisingly, a clear methodology how to approach code as a research object has not been worked out yet. We therefore introduce a new methodological approach called Computational Science Studies (CSS). CSS can be considered as a sub-discipline of Science Studies. It focuses on the computerization of scientific disciplines with an emphasis on the practices of programming. It is thus related to STS as well as to Philosophy of Science and to the Social Studies of Science. Our main argument is that with the growing importance of programming practices, instruments become necessary to examine this hidden level of scientific knowledge production. Since most of the program codes are opaque and written by more than one person, STS scholars without an education in computer science won't have the code literacy needed to understand them. Hence, one goal of CSS is to develop software tools that concede STS scholars and other scientists a semantic access to the code. This can be done either by focusing on the temporal dimensions in coding (e.g. syntactic orders like or by the structure and the content of the comments (semantic level of code). In what follows, we will concentrate on the written comments although we are well aware of the fact that programmers often do not comment at all (Gramelsberger 2010). However, since an external observer barely understands the written lines, it has been our goal to extract and visualize the comments to generate a first overview. Comments can be short annotations to explain something in the code or point to a bug but they can also reveal a deeper issue of the research project. That is why one leading question of the CSS and this article will be: How can we use tools that help us to understand the genealogy of research objects within the code?

Thus, we developed a software tool specialized on the extraction of comments and the visualization of data structures, the so called Isomorphic Comment Extractor (ICE). With this tool, STS scholars are able to trace content and meaning with the help of the comments within a code project. This simple procedure helps to analyse the composition of the code as well as to detect refinements, corrections or the integration of new libraries that would otherwise be ignored in the analysis of scientific knowledge production. We present the ICE in section 4 and exemplify it in our case study from the field of geology in section 5. Using the ICE, we show the diverse functions and semantics of comments and how a navigation through these comments can hint to some critical aspects of 3D-model-building in geology. But before we start with the case study, we have to elaborate further on the field of Code Studies. Since CSS is a methodological approach based on Code Studies, we will first explain the core idea of this approach on Computational Sciences from a Philosophy of Science perspective in section 2. In section 3, we discuss three variations of code studies (ethnographic codes studies, software studies and critical code studies) and compare them with our methodology on computer-driven sciences. In the conclusion, we discuss the results of our case study and point to restrictions and further research questions of our CSS methodology.

2 What are Code Studies?

In a general sense, Code Studies gather together different fields from (mostly) the humanities which explore the role of computer code for economic, social, political and scientific processes, the aesthetics of code as well as the different practices of coding (individual or collaborative coding, e.g., see Strathern 2005). Hence, Code Studies are not systematically coherent, which might be the reason for their lack of a common methodology. Instead, they focus on the practice of computer programming and its significance for the impact of software, which can be described as follows: “Programming languages are the medium of expression in the art of computer programming” (Mitchell 2002, 3). Letting aside the question whether programming computers is an art or a handicraft, programming languages as a medium entail a syntax and certain semantics that are written in the numerous lines of code. Scholars from Code Studies like David M. Berry see in the code the “literary side” of the programming practice that has a “real code” and an “absolute code”. While the first one is fragile and highly contingent, due to the particularity of single solutions, the second one aims at “thinking in terms of both the grammar of code itself, but also trying to capture how programmers think algorithmically, or better, computationally" (Berry 2015, 33). This relation between the "grammar of code" and the ways programmers are thinking through and with their programming languages is essential and we will come back to this later. Another definition of Code Studies highlights the aspect of performativity. Computational processes depend on a logic system with statements such as "*if* something, *then* something else". This turns the code to a two-fold object that describes future events and sets the instructions to execute a specific task (which is mainly the point where algorithms come into play) (Cox and McLean 2013, 41). In other words: The description of the algorithm and prediction of the event are both entangled in the lines of code so that "it [the code, L.S. & D.K.] can be history but intervening in the very process of history" (Cox and McLean 2013, 42). This two-fold logic of doing history and being part of a (research) history makes it hard to distinguish functional practices (to make the history work) from scientific practices in coding (writing the history of the research object).

What is so special for Code Studies is the claim that through code as a medium, the research object is generated and observed at the same time. This does not refer to a constructivist point of view. Moreover, it shows a new temporal regime in which the computational data is controlled by code and processed as scientific data in computational time. Hence, one can argue with the philosopher Sabina Leonelli that we have first to distinguish a time of data maintenance and construction and second a time of the "phenomena under investigation" (Leonelli 2018, 742-743). This distinction is important since it shows how any scientific value – either true/false or valid/non-valid – depends on this new temporality of data embedded in the medium of code. Parting from the idea of Code Studies, the CSS look for ways how to reflect these new temporal data regimes that highly depend on the media in which they are processed - whether this is a computational model of a physical phenomenon or a visualized model of an unknown, anticipated behaviour (e.g. the spread of an infection in a region over a certain time). Tools like the ICE help to understand the dynamics in these relational structures of data between code, database and the visualization on the interface. Of course, to extract comments out of the code is not new and it also might not always lead to new insights. Nevertheless, especially in collaborative coding we can use any programming editor to get an overview about the justifications why some programmer has chosen one kind of algorithm and not the other for a certain problem. Also, we find debugging processes where the computational model did not work or where an important decision has been made so that a documentation of the code modifications is needed.

Thus, taking this into account, Code Studies can be described as a loosely coupled field of disciplines that explore the temporal (syntax) and structural (comments, merging of versions in Version Control Systems[1]) dimensions of coding practices. We argue that these dimensions have become increasingly influential in the production of scientific knowledge. However, in all these cases, the reason why and how these "interventions in the very process of history" have taken place might be difficult to understand for a STS scholar who is neither a part of the scientific group nor one of the programmers. This makes the question of an adequate methodological approach so necessary but at the same time so demanding when it comes to the exploration of the computer code as a research object. Our tool is here just one first step based on the argument that we have to examine the dynamics of programming to know how computational sciences are shaping their research objects. In this emphasis on coding, CSS relates to preceding approaches of Code Studies that we would like to present now to highlight the similarities and differences between them and our methodology.

3 Approaches in the Field of Code Studies

3.1 Ethnographic Code Studies

In the last 15-20 years, several approaches have been developed that try to grasp the technical, ontological but also the ideological dimension of computer code (see for this three-fold problem Chun 2004 & 2011). For our question how computer code could be examined in the context of STS, we looked at three of them. The first, Ethnographic Code Studies (ECS), focuses on the modelling practices in scientific communities. ECS describe

1. "Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later." (Chacon and Straub, 2014, 9) From the developer's perspective, this feature is especially helpful for collaborations, as each collaborator can (simultaneously) work on their own local copy and they update their changes in the shared repository, which then distributes this changes to all the other local copies. From the Code Studies perspective, this presents us with the opportunity to access many projects and their developmental stages without the need for a personal connection to the developing team.

the methodological change in the sciences from the experiment to complex models – like in Climate Sciences – that are run via simulations. To do that, it is important to know how the scientific objects have been shaped during the modelling process. For example, the Social Studies of Science scholar Mikaela Sundberg described the development of a meteorological model and its genealogy, going from the field experiments to the modelling process that has taken place in the research group (Sundberg 2009).[1] To have better models means in this case to have a simulation running that produces more precise weather forecasts. This means that data from the field has to be integrated into the theoretical weather models, which means to decide what parameters are most important for the computational model. As Sundberg explains:

"Predictive construction involves the application of existing scientific knowledge to develop new parametrizations. At the weather service, there is a division of labor in this process between researchers and programmers. Researchers transform equations into code or more often make changes directly in the code. But this coding is generally preliminary. Programmers continue where the researchers leave off and develop the code in ways that enable the simulation model to use computer power more efficient but do not change the basic operations that the code executes" (2009, 169).

This cooperation of researchers and programmers for the development of simulation models is the main focus for ECS. On the one hand, this allows – like in the quote – to analyse the transformation of theories and models into the lines of code. On the other hand, Sundberg points out later in the text that it is never the code that becomes important for the scientists but the graphical and numerical outcome of the simulated models. She

1. Computational Models in Climate Sciences are peculiar because they are also widely discussed in the philosophy of science due to their impact on the relation of theory and experiment (Heymann, Gramelsberger and Mahony 2017; Winsberg and Goodwin 2016). These models have two special characteristics: First, the number of parameters (how many parameters, how long is the runtime of the simulation etc.) has to be scaled down to the very necessary factors for the simulation. This necessitates efficient use of the computer resources. The second point is that climate models are based on non-linear equations which cannot be solved analytically but have to be calculated discretely, from one time-point to another, which requires a huge amount of calculation power. Therefore, especially computer models in Climate Sciences are paradigmatic for a new balance between a theoretical claim to be proved and the practice of modelling. As Sundberg explains: "However, there is no algorithm for reading of models from theories. Therefore, theories function as constraints and not as determinants in the process of simulation model construction" (2009, 163).

also calls the code, in reference to Bruno Latour's "immutable mobiles", the "mutable mobile", a dynamic inscription hidden behind the visualized layers but indispensable for the identity of the simulation model and the reproducibility (and therefore mobility) of the scientific predictions (Sundberg 2009, 173 & 175). Hence, what Sundberg lays open are the fragilities of the simulated computational models due to the mutability and modifiability of computer code. But both in the work of the scientists as well as in the journal publications the code is considered as something "secondary" (Sundberg 2009, 172). Thus, it is still one of the open questions in ECR how to trace these changes of the model in the code and to analyse their impact *before* they become a condensed visual result of the simulation. Thus, the switches from researchers to programmers, as described in the quote, might not only have an organizational and technical impact but also a scientific one.[1] To analyse that, one would have to go into the structure of the computer code itself and look for syntactic and semantic indicators.

3.2 Software Studies

A wider perspective on the phenomenon of code has been taken by the interdisciplinary field of Software Studies. Software studies include scholars from media and cultural studies, sociology, computer science, and engineering. Since the 2000s their publications have dealt with the technical impact, the media history and the political power lying behind the everyday use of software (see as early examples Manovich 2001 and Fuller 2003). Thus, for Software Studies, programming languages are a key-element not only to trace the transformations of analog objects into the digital world, but also to take into account the moral and ideological implications that are associated with the use of software (Fuller 2008). In this sense, programming always entails a certain mode of seeing the world, a mode that anticipates its users. As the Software Studies scholar Adrian Mackenzie writes: "Knowledge, truth, speculation and prediction are practically re-configured through programming and coding" (Mackenzie 2013, 329). This view on software as a way of subjectivity became more and more dominant in the last years. The increase of platforms

1. In her ethnographic study about a group of meteorologists, Mikaela Sundberg describes how the code becomes part of a centre collective in the sense that the whoever enters the scientific organization also enters the code until he leaves and loses the access to work with the code (Sundberg 2010, 50). Such strict organizational models around code work are of great importance and they could be combined with tools like the ICE to examine the scientific work done in the code.

and online-connected things has led to another emphasis on the code as the element that always mediates but never appears on your bill or your medical health card – something always hidden behind “Coded Assemblages” (Kitchin and Dodge 2011, 7).[1] However, although most of Software Studies approaches emphasize the importance of the code as more than just another technological tool, they rarely go into the code itself to show how (and particularly *when*) the valuing aspects of programming come into play. There are two reasons for that. First, looking into the code of private companies is almost impossible (and they are often the target of critical Software Studies, see for example Rossiter 2016). Second, the code is not treated as a research object but as part of a bigger cultural impact which relates to our everyday usage of software – or, to quote from the title of another of Manovich’s books: “Software Takes Command” (Manovich 2013). Although it is important to emphasize the macrocultural context of software production and distribution, a deeper code analysis is still secondary for most of the contributions to software studies. However, scholars like Adrian Mackenzie have shown how the changing practices of coding (Mackenzie 2017), like the increase of Version Control Systems, libraries, and platforms for coding issues such as Stack Overflow[2], transformed significantly the relation of pure and applied sciences in the age of Machine Learning. Our methodology of CSS tries to develop tools to lay open these connections between changing coding practices and data visualizations, between the performance of the machine and the standardised procedures of knowledge production. Here also, a combination of software studies and the CSS methodology would be an option for the reflection of this double-sided nature of code as written history and writing practice that intervenes in the process of history.

1. Kitchin and Dodge refer especially to the complexification of code through developments like the Internet of Things and other so called smart applications. Moreover, the problem of an opaque computer code is also crucial for the analyses of machine learning as an instrument in science. Adrian Mackenzie even argues for a differentiation between “writing” and “growing” programming languages due to the increase of open source programming, data software packages and further attachments like libraries. Hence, although “machine learning utterly depends on code”, it would be insufficient just to focus on code to explain the use of machine learning in science (Mackenzie 2017, 22-23). Instead, Mackenzie calls for a hybrid methodology connecting code studies, ethnography, diagrammatic reasoning, media archaeology and history of statistics.

2. Stack Overflow is a web-based platform for requests about code writing for private and business programmers. It was developed in 2008 and runs since then under the Creative Common Licence.

3.3 Critical Code Studies

The third way of developing a method for the analysis of code has resulted out of the Software Studies movement in 2006 (Montfort et al. 2013, 6), the so called Critical Code Studies (CCS). In the words of one of their founders, the humanities scholar Mark C. Marino, Critical Code Studies "takes for its milieu the code layer of Software Studies […]. Rather than examining source code to the exclusion of these other aspects of technology […], CCS emphasizes the code layer in response to a lack of critical methodologies and vocabulary for exploring code through this cultural lens" (Marino 2014). One way of filling this methodological gap is to explore the "heteronormativity of code" in an experimental way (Blas 2008 after Marino 2014), which has often been done with methods from artistic research. The main goal here is to play with and through the practice of programming and to offer a new look at programming that is not uniquely bound to its functional involvement in software engineering. Thus, code is considered to be an aesthetic object that has also to be explored by artists. Although there has been an increase in the humanities' methods to read code closely, it also became clear that only focusing on code would be too restricted. Hence, Software Studies as well as Critical Code Studies do not represent fields with a certain restricted or fixed methodological approach, but they entangle the analysis of code with interviews, media histories, archive work and artworks (Montfort et al. 2013, 7). This could also be a model for the methodology of CSS since the scientific practices can't be reduced to programming. In contrast, the steps between data gathering, theory and model building, coding and simulation resemble more interwoven cascades than linear work-flows. To show how these dynamics are processed, revised and inscribed again to shape the research object, most analyses lack a new level of reflection. That is why we argue that the CSS methodology depends on software tools that help us to draw the connections between the interventions in the code and the dynamics of computational model-building and its simulations. It is also here where the CSS distincts strongly from the Critical Code Studies and other Software Studies in focusing on scientific processes.

4 What can Computational Science Studies Offer?

These three examples of Code Studies show how important the impact of programming has become – not only for programming software but also for the genealogy of scientific objects. If we take the phrase by Adrian Mackenzie, that "[k]nowledge, truth, speculation and prediction" are reconfigured through coding, then we have to ask again how to approach code more specifically. What remains problematic in all three examples is that the notion of code has been considered as a neutral tool - in other words, a tool that does not affect the practices of valuing and knowing which influence each other in scientific practice. So first, if we look at code as a research object for STS and Philosophy of Science, we have to make sure how the correlation of valuing and knowing as a scientific practice is expanded through programming. That means also that writing code is influenced by programming languages and the way these languages restrict or expand the scientific ideas that have to be implemented in the code. This can lead to the genealogy of scientific objects but also to tensions between the scientific model and its transformation into code. In-between model building and data visualisation, the practice of programming – and not the ontology of software, as so long claimed by Software Studies – becomes crucial. However, to look more closely at code as an empirical object without being an expert in programming can become difficult. That is why for our approach of CSS, the development of tools to do research in and with the code is fundamental for the study of computerized sciences.

In 2018, we founded the CSS Lab for this purpose where we intend to establish an open science infrastructure for Computational Science Studies in order to facilitate the scientific analysis of code. In the Lab, we are currently developing software tools designed to support STS scholars and other scientists in their case studies on code projects. The tools open up new elements of code analysis by helping to visualize software structures, supporting code genealogies and code comparisons, and allowing for an analysis of the scientific content of software projects (such as scientific models, scientific data analysis algorithms, and measurement procedures). The last of these, in particular, is an entirely new demand resulting from the requirements of conducting Computational Science Studies.

The Isomorphic Comment Extractor (ICE)[1] is the first of our CSS Tools and visualizes comments within the code for the most common programming languages such as Python, C/C++, Fortran, etc. The interface provides a special environment which is easy to use if

1. The ICE can be found on our website: https://www.css-lab.rwth-aachen.de/tools/overview [Accessed 16 September 2019].

you are not familiar with software projects (see Fig. 75). It is rooted in the isomorphic basis of all our tools, which represents and depicts the file structure of a software project in isomorphic form. In comments, there may be links to scientific papers on which the code is founded, documentation about ownership and other legal aspects, general hints on how to use the code, explanations about what the code should do, indications of problems that still need to be solved or other aspects related to the genesis of the code. So while not every software project is well documented through comments, it is still a reasonable endeavour to start any analysis by looking for the comments since that is the main location for background information on the code.

5 Case Study: Comments in GemPy's geophysics.py

5.1 The Project GemPy: Modelling of Geological Structures

One main aspect of geological research is "understanding and visualizing how rocks are organized below the Earth surface, both for practical reasons and out of scientific curiosity to understand the world below our feet." (Wellmann and Caumon 2019, 4) In recent years, the visualisation of geological structures is not only done in two-dimensional maps but has been extended to three-dimensional computer models of different types of geological layers. Since real-world measurements are sparse, one important aspect of geological modelling is to combine all existing data points as well as theories of geological events, changes, and concepts into one model.

Various tools, mostly commercial products, exist for the modelling of geological structures. One of them is the software GemPy (de la Varga and Wellmann 2019) which is an open-source project developed in Python by the Chair of Computational Geosciences and Reservoir Engineering at RWTH Aachen University, led by Prof. Florian Wellmann, who kindly allowed us to use their software as a case study for this paper.[1] The output of GemPy are malleable 3D models such as the one you can see in Fig. 74. The different colours represent different types of geological layers.

1. We are looking at GemPy version 1.16, which was released on May 1st, 2019. Since then, the project has been revised and completely redesigned for version 2.0. The version used for this article can be found at *https://github.com/cgre-aachen/gempy/releases/tag/v1.16* [Accessed 16 September 2019].

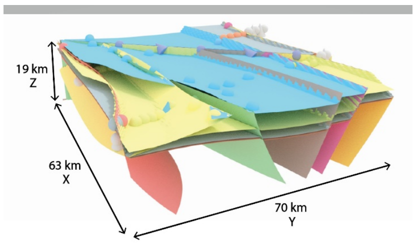


Fig. 74: *Example of a geological model generated with GemPy (de la Varga and Wellmann 2019).*

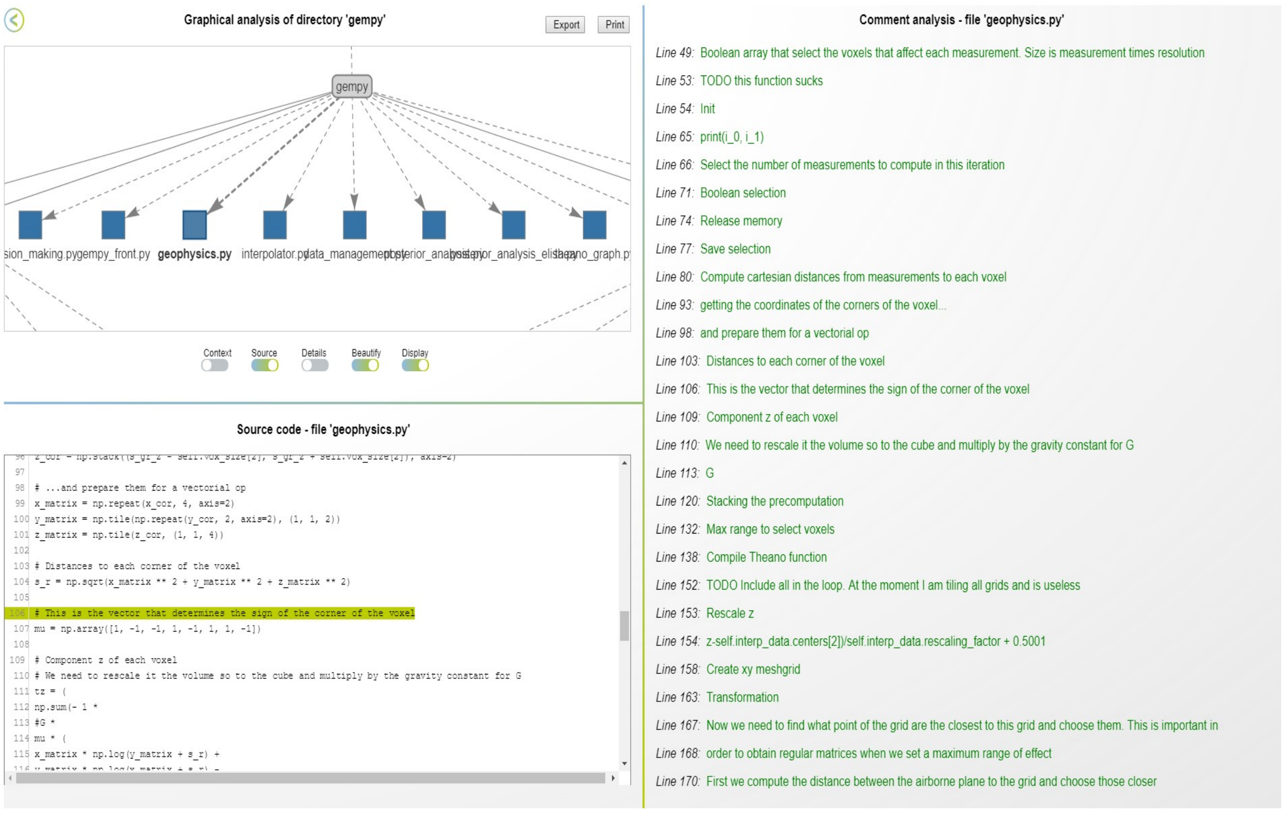
5.2 Studying Geological Modelling with the ICE

In the following, we will showcase the three main functions of the ICE, followed by an exemplary analysis of one part of GemPy. This analysis exemplifies the relation of model building and comment structure. In order to use the ICE, one needs to have a local copy of the project one is interested in. With GemPy, as with many others of today's software projects, this does not pose a problem, since the code is an open-source project and available for download on GitHub[1]. For starting the analysis, one has then simply to select the local directory of the code.

The main view in the ICE is divided into three parts that represent the main functions (see Fig. 75). The first aspect, which is directly visible after loading a project, is the graphical analysis of the data structure in the upper left corner. The tree structure shows grey boxes representing directories, whereas the coloured squares symbol single code files. The different programming languages are depicted in different colours. In well-maintained software projects, the file structure should give a first overview on the different functionalities of the project because files with similar functions are grouped together in a directory. In object-oriented[2] software projects, each file directly represents one object, i.e. one functionality, which should help even more for identifying the purposes of the different parts of a project. Therefore, a careful look at this structure can be beneficial for understanding the project. With the ICE, this is facilitated by the possibilities to zoom in on the tree and to showcase different branches.

1. GitHub is a host for Git repositories which are part of version control systems.
2. The concept of object-oriented programing "combines data abstraction and inheritance. The central feature is the object [… which] comprises a data structure definition and its defined procedures in a single structure" (Butterfield and Ngondi 2016, 377).

If one of the files in the data structure is selected by a click, the second section in the column on the right side will show all the comments from the selected file. If the project is well maintained, this can provide a good overview on what this file in particular is supposed to do. Especially helpful for this objective are the comments at the beginning of a file, which are usually more of a general nature. They describe the overall purpose of the file as well as general comments with a link to a reference, e.g. a scientific paper. But even if it is not well maintained, it is still possible to infer useful information from the comments: programmers often indicate aspects that do not (yet) work as they should or point out features that are still on the to-do list. If the project is not in a final version, these kinds of comments are vital tools for communication within the team of programmers (they are also fundamental for remembering different aspects if one programmer works alone) and the language used can sometimes be colloquial and/or explicit.



Source code - file 'geophysics.py'

```
 96 z_cor = np.stack((s_gr_z - self.vox_size[2], s_gr_z + self.vox_size[2]), axis=2)
 97
 98 # ...and prepare them for a vectorial op
 99 x_matrix = np.repeat(x_cor, 4, axis=2)
100 y_matrix = np.tile(np.repeat(y_cor, 2, axis=2), (1, 1, 2))
101 z_matrix = np.tile(z_cor, (1, 1, 4))
102
103 # Distances to each corner of the voxel
104 s_r = np.sqrt(x_matrix ** 2 + y_matrix ** 2 + z_matrix ** 2)
105
106 # This is the vector that determines the sign of the corner of the voxel
107 mu = np.array([1, -1, -1, 1, -1, 1, 1, -1])
108
109 # Component z of each voxel
110 # We need to rescale it the volume so to the cube and multiply by the gravity constant for G
111 tz = (
112 np.sum(- 1 *
113 #G *
114 mu * (
115 x_matrix * np.log(y_matrix + s_r) +
116 y_matrix * np.log(x_matrix + s_r) -
```

Comment analysis - file 'geophysics.py'

Line 49: Boolean array that select the voxels that affect each measurement. Size is measurement times resolution
Line 53: TODO this function sucks
Line 54: Init
Line 65: print(i_0, i_1)
Line 66: Select the number of measurements to compute in this iteration
Line 71: Boolean selection
Line 74: Release memory
Line 77: Save selection
Line 80: Compute cartesian distances from measurements to each voxel
Line 93: getting the coordinates of the corners of the voxel...
Line 98: and prepare them for a vectorial op
Line 103: Distances to each corner of the voxel
Line 106: This is the vector that determines the sign of the corner of the voxel
Line 109: Component z of each voxel
Line 110: We need to rescale it the volume so to the cube and multiply by the gravity constant for G
Line 113: G
Line 120: Stacking the precomputation
Line 132: Max range to select voxels
Line 138: Compile Theano function
Line 152: TODO Include all in the loop. At the moment I am tiling all grids and is useless
Line 153: Rescale z
Line 154: z-self.interp_data.centers[2])/self.interp_data.rescaling_factor + 0.5001
Line 158: Create xy meshgrid
Line 163: Transformation
Line 167: Now we need to find what point of the grid are the closest to this grid and choose them. This is important in
Line 168: order to obtain regular matrices when we set a maximum range of effect
Line 170: First we compute the distance between the airborne plane to the grid and choose those closer

Fig. 75: *Main view in the ICE, exemplified on the GemPy file geophysics.py (authors' illustration).*

The third aspect of the main view in the ICE is the section in the lower left corner. It shows the source code from your selected file. By clicking on one of the comments in the comment analysis section on the right side, one can then add another layer of information: the comment clicked on will be highlighted in the centre of the source code section. This is especially helpful if it is of interest what happens in the direct environment of a comment because comments often introduce the next few steps that will be executed in the code or explain the details of a specific variable. Or, as explained before, they indicate problematic features which can also indicate areas of interest for the STS researcher. It might be also useful in case that there is a paper on the conceptual ideas behind a project and one wants to compare the code with the paper. In this case, it is necessary to see not only the comments but also the specific equations in the code in order to compare them.

When using the ICE with GemPy, one can see a number of Python files (indicated in dark blue) in the project directory and some sub-directories that themselves bear more Python files. This indicates that the files of interest for understanding the geology behind the project are probably in the main directory, since all the names of the sub-directories suggest that they contain auxiliary functions (as they are called “plotting”, “add-ons”, and “utils”).

Looking at the files in the GemPy directory, the file name geophysics.py leaps to the eye, so we choose this file for the starting point of our analysis. The class name “GravityPreprocessing(object)” already hints at a possible functionality of this file, namely the calculation of the influence of gravity on some given object. By reading the comments, we can confirm that the influence of gravity is indeed calculated in this file, since there are comments indicating that a block of code is supposed to do exactly that. We can also infer that this file seems to be responsible for calculating distances and other relational aspects between the measurements and the voxels[1] of an object in the geological model. Since real-world measurements are sparse, the interpolation of those measurements is an important step in the model generation. It simulates the data for the areas in between the geological measurement points. Furthermore, the comments show some aspects that are work-in-progress, as there are comments including “TODO”, one of the main indicators for further work which automatically gets highlighted in many editors used for programming. These work-in-progress comments show a less formal writing style and are not as refined as some of the others. They seem to be meant as reminders for the programming team and not so much as a documentation for outsiders. For example, this difference can be seen in the comparison between line 152 “TODO Include all in the loop. At the moment I am tiling all grids and is useless” and line 106 “This is the vector that determines the sign

1. A voxel is the 3D equivalent of a 2D pixel, i.e. the representation of one value in a grid.

of the corner of the voxel". The first is an example for an informal comment that is not meant for the general public. It seems rashly written and has a slip in grammar. The second one on the other hand is a very descriptive comment on the calculation in the line following it. That is a very common type of comment used for the documentation of a project for people not involved in the programming.

Even though this was just a first glance at a complex project, this analysis of the comments already clarifies many things. The comments reveal the purpose of the file and they bring out (some of) the steps taken to achieve this aim. Furthermore, it is possible to infer the state of development in which the file currently is in. The comments open up a temporal dimension in the code, as they sometimes refer to the point of their creation, sometimes to the future in which a programmer is supposed to work on some aspect, and sometimes to the present of the reader understanding the sequence of the program.

5.3 Benefits from Using the ICE

In contrast to code analysis tools aimed at software programmers which also allow for highlighting of comments, the ICE is easy to navigate and understand even if one is unfamiliar with the peculiarities of programming languages or the structure of software projects. This is especially facilitated by the combination of file structure navigation and the highlighting of comments. The user becomes able to detect hierarchies and connections between the different parts of the project which in comparison to the other forms of Code Studies creates an additional value since it provides a deeper access to the code itself. Even though no contact to the developing team is needed, the researcher can still trace (some) of the interactions through comments (especially with comments that include a "to do"-memo or the authorship of a file). Furthermore, comments can also provide an indication for the intention behind the code (e.g. an explanation which specific problem a certain piece of code is supposed to solve). An analysis of these kinds of comments can make explicit where the programmer has interpreted a scientific idea. In some cases, this can even be extended by showing interconnections between scientific papers, theories, or ideas and their implementations in the code. While the ICE can be used as a stand-alone application as shown here, it is meant to be one of many aspects of a full analysis of a software project. Of course, any analysis that depends on the code as the central textual basis operates under the assumption that the project is well-maintained. Our tools as well as methods from other forms of code studies are envisioned to supplement many different angles from which the analysis of a software project may be fruitful. This also includes established visual forms of knowledge representation like graphs, flow charts, or diagrams. But they all should not supersede an intensive examination of the code by the STS researcher.

6 Discussion

In this paper, we presented a new perspective on Science Studies and Code Studies. We claim that with the increasing impact of computation in science, the practice of programming becomes crucial for the analyses and interpretation of Computational Sciences. Hence, our methodology called Computational Science Studies stresses the point that coding is opaque and mostly written in collaboration. This demands for a new approach that enables STS and other Studies of Science to examine the transformation of empirical objects and scientific models into code. In following positions from Code Studies, we distinguished the syntactical and semantic dimension of programming languages. We emphasized the two-fold logic of coding in doing research history and being part of a research history which is important for the constructive but not constructivist character of computational sciences. To confine our approach more precisely which stems from a philosophy of science perspective, we presented the basic ideas of Ethnographic Code Studies, Software Studies and Critical Codes Studies and compared them with our methodology. We concluded that it is urgent not only to determine the heterogeneous impacts of writing software or the ontology of the code but to develop software tools that concede an access to the functional, organizational and scientific levels of programming. One such tool has been presented in this paper – the Isomorphic Comment Extractor.

In our case study from computational model building in geology, we examined the multiple functions of comments in the practice of programming. Thus, the semantic level of comments embodied functional, organizational and scientific contents around the issue of calculating in-between spaces of geological models. These models are based on measurements of different geological layers, but what they lack are empirical data about the space in-between the measuring points. Consequently, it is this space that has to be simulated and written down in code. Analysing the comments and the data structure with the ICE revealed the points in the project in which the simulated data points were incorporated through the code.

Tools like the ICE offer another point of view to this hidden layer of scientific knowledge production. However, such tools are only one possible way of looking at techniques of knowledge production. Another would be to take the graphs, diagrams or flow-charts into account and to compare their function for the representation and construction of knowledge with the analysis from tools like ICE. Therefore, the ICE is only one possible way to approach the structure of code and it could be complemented with visual tools, leading to what Karin Knorr-Cetina called a "viscourse" (Knorr Cetina 2001, 307), a visualization of knowledge embedded in an ongoing scientific discourse. Hence, the ICE is just a beginning for a tool-based methodology of CSS and not every scientific code offers the quantity and diversity of comments that is adequate to use the ICE as an analytic tool. However, in further research, the comparison of program codes of scientific projects could provide new insights. In particular, a first scheme of semantic code layers with their specific comments would ease the way how to use code as research object in the philosophical and social studies of science.

References

Berry, David M. (2015): The Philosophy of Software. Code and Mediation in the Digital Age. New York.

Butterfield, A. and Ngondi, G. (2016): A Dictionary of Computer Science. 7th ed. Oxford.

Chacon, Scott; Straub, Ben (2014): Pro Git. 2nd ed. Berkeley, CA.

Chun, Wendy H.K. (2004): On Software, or the Persistence of Visual Knowledge. In Grey Room 18 (4), pp. 26-51.

Chun, Wendy H.K. (2011): Programmed Vision. Software and Memory. Cambridge, MA.

Cox, Geoff; McLean, Christopher A. (2013): Speaking Code. Coding as aesthetic and political expression. Cambridge, MA.

de la Varga, M. and Wellmann, F. (2019): cgre-aachen/gempy. [online] https://github.com/cgre-aachen/gempy [Accessed 22 May 2019].

Fuller, Matthew (2003): Behind the Blip. Essays on the Culture of Software. New York.

Fuller, Matthew (2008): Software Studies. A Lexicon. Cambridge, MA.

Gramelsberger, Gabriele (2010): Computerexperimente. Zum Wandel der Wissenschaft im Zeitalter des Computers. Bielefeld.

Gramelsberger, Gabriele (2011): What do numerical models really represent? In Studies in History and Philosophy of Science 42 (2), pp. 296-302. DOI: 10.1016/j.shpsa.2010.11.037

Heymann, Matthias; Gramelsberger, Gabriele; Mahony, Martin (2017): Key Characteristics of Cultures of Prediction. Heymann, Matthias; Gramelsberger, Gabriele; Mahony, Martin (Eds.): Cultures of Prediction in Atmospheric and Climate Science. London, pp. 18-42.

Jones, Steve E. (2016): Emergence of the Digital Humanities (as the Network Is Everything). Gold, Matthew K.; Klein, Lauren F. (Eds.): Debates in the Digital Humanities 2016. Minneapolis and London. [online] https://dhdebates.gc.cuny.edu/read/65be1a40-6473-4d9e-ba75-6380e5a72138/section/09efe573-98e0-4a10-aaa3-e4b222d018fe#ch01 [Accessed 12 September 2019],

Kitchin, Rob; Dodge, Martin (2011): Code/Space. Software Everyday Life. Cambridge, MA.

Knorr-Cetina, Karin (2001): “Viskurse” der Physik: Wie visuelle Darstellungen ein Wissenschaftsgebiet ordnen. Heintz, Bettina; Huber, Jörg (Eds.): Mit dem Auge denken. Strategien der Sichtbarmachung in wissenschaftlichen virtuellen Welten. Wien and New York, pp. 305-320.

Leonelli, Sabina (2018): The Time of Data: Timescales of Data Use in the Life Sciences. In Philosophy of Science 85, pp. 741-754.

Mackenzie, Adrian (2013): Programming Subjects in the Regime of Anticipation: Software Studies and Subjectivity. In Subjectivity 6, pp. 391-405.

Mackenzie, Adrian (2017): Machine Learners. Archaeology of a Data Practice. Cambridge, MA.

Manovich, Lev (2001): The Language of New Media. Cambridge, MA.

Manovich, Lev (2013): Software Takes Command. New York.

Marino, Mark C. (2014): Field Report for Critical Code Studies. In Computational Culture. A Journal for Software Studies. Published 9th November 2014. [online] http://computationalculture.net/field-report-for-critical-code-studies-2014%E2%80%A8/ [Accessed 23 May 2019].

Montfort, Nick et al. (2013): 10 Print CHR$(205.5+RND(1)); : GOTO 10. Cambridge, MA.

Mitchell, John C. (2002): Concepts in Programming Language. Cambridge, MA.

Rossiter, Ned (2016): Software, Infrastructure, Labor: A Media Theory of Logistical Nightmares. New York.

Strathern, Marilyn (2005): Imagined Collectivities and Multiple Authorship. Ghosh, Rishab Aiyer (Ed.): Collaborative Ownership and the Digital Economy. Cambridge, MA, pp. 13-28.

Sundberg, Mikaela (2009): The Everyday World of Simulation Modeling: The Development of Parametrizations in Meteorology. In Science, Technology, & Human Values 34 (2), pp. 162-181.

Sundberg, Mikaela (2010): Organizing Simulation Code Collectives. In Science Studies 23 (1), pp. 37-57.

Wellmann, Jan Florian; Caumon, Guillaume (2019): 3-D Structural geological models: Concepts, methods, and uncertainties. [online] http://publications.rwth-aachen.de/record/754773/files/754773.pdf [Accessed 12 September 2019].

Winsberg, Eric; Goodwin, William M. (2016): The adventure of climate science in the sweet land of idle arguments. In Studies in History and Philosophy of Modern Physics 54, pp. 9-17.