Research Ideas and Outcomes 3: e13236
doi: 10.3897/rio.3.e13236

Workshop Report

Identifying the challenges of code/theory translation: report from the Code/Theory 2017 workshop

Caroline Jay[‡], Robert Haines[‡], Markel Vigo[‡], Nicolas Matentzoglu[‡], Robert Stevens[‡], Jonathan Boyle[§], Alan Davies[‡], Chiara Del Vescovo[|], Nicolas Gruel[‡], Anja Le Blanc[‡], David Mawdsley[‡], Dale Mellor[‡], Eleni Mikroyannidi[|], Richard Peter Rollins[‡], Andrew Rowley[‡], Julio Vega[‡]

‡ University of Manchester, Manchester, United Kingdom
§ Numerical Algorithms Group, Manchester, United Kingdom
| BBC, Salford, United Kingdom

Corresponding author: Caroline Jay (caroline.jay@manchester.ac.uk)

Reviewable v1

Received: 13 Apr 2017 | Published: 13 Apr 2017

Citation: Jay C, Haines R, Vigo M, Matentzoglu N, Stevens R, Boyle J, Davies A, Del Vescovo C, Gruel N, Le Blanc A, Mawdsley D, Mellor D, Mikroyannidi E, Rollins R, Rowley A, Vega J (2017) Identifying the challenges of code/theory translation: report from the Code/Theory 2017 workshop. Research Ideas and Outcomes 3: e13236. https://doi.org/10.3897/rio.3.e13236

Abstract

The Code/Theory workshop explored the process of translating between theory and code, from the perspective of those who do this work on a day to day basis. This report contains individual contributions from participants reflecting on their own experiences, along with summaries of their lightning talks and outputs from the discussion sessions. We conclude that translating between theory and code successfully requires a diversity of roles, all of which are central to the process of research.

Keywords

Software Engineering; Reproducible Research; Modelling; Ontologies; Scientific Theory; Scientific Computing; Scientific Software; Research Software Engineering; Linked Data

Introduction

On Monday 16th January 2017 we held the inaugural Code/Theory Workshop at the University of Manchester. The event brought together programmers and architects from academia and industry to discuss the challenges of translating between 'theory' - scientific or logical concepts - and 'code' - software, ontologies or linked data.

The aim was to capture the views and experiences of people dealing with these issues in practical terms, to understand where the challenges lie, and how we can address them. The discussion covered a broad range of topics, from low-level technical issues to high-level communication. An overwhelming message was the importance of teamwork, and a recognition that for the process to work well, the environment must support parity of esteem between programmers, researchers and domain experts.

The workshop started with each participant giving a lightning talk outlining their experiences in this area. This was followed by two discussion sessions in small groups: "Why is code/theory translation challenging?" and "How can we improve code/theory translation?". This report starts with personal contributions from all workshop participants, describing their own view of code/theory translation. It then summarises the outputs of two discussion sessions, focusing on why code/theory is challenging, and potential solutions to these challenges. It concludes with a recommendation that to support the difficult process of code/theory translation, promoting a culture where the contribution of all specialists are recognised and valued is key.

Participant contributions

Each participant was asked to present a lightning talk about their experiences in this area, which is summarised here. Participants were also invited to submit an abstract to give further details of their thoughts on this topic; where this was provided, it is included below.

Jonathan Boyle

Jonathan is currently an HPC Application Analyst at The Numerical Algorithms Group and has previously worked as a Research Software Engineer at the University of Manchester. He has a PhD in experimental physics.

In his lightning talk, Jonathan described theory as collective knowledge, often contained within papers, that is converted into personal knowledge, and particularly into understanding, and then into code design (Suppl. material 1). This process can be hard work, and the way in which it happens is not always obvious, and not well understood; it occurs at both a conscious and unconscious level, involving creativity and reflection. Translating theory into design is hard, while actually writing the code is more straightforward. The challenge involved in translating theory to code also varies considerably, with different levels of complexity, and some implementations being much

more straightforward than others. There are also social aspects to consider, as the process requires teamwork and different sets of skills, as well as potentially having a variety of goals and constraints. It is important to consider the context in which it is happening.

Abstract

In my experience the challenges associated with code/theory are similar in other domains, for example, the challenges associated with good project management, team working, and software engineering. Specific challenges involving theory tend to require converting theory to design and data to theory via personal understanding, and in my experience acquiring and applying understanding is a rather mysterious skill involving various processes within the human mind e.g. learning, idea generation, reflecting, intuition and planning.

Despite these similarities, there are large variabilities within theories, codes and contexts, where context includes the people and the project goals and constraints. Perhaps code/theory is the wrong level of consideration, and the scope needs to be increased or reduced accordingly to provide something more amenable to analysis.

Alan Davies

Alan Davies is a PhD candidate in Computer Science and a Research Assistant at the University of Manchester. He has previously worked as a Software Engineer in industry, and as a Cardiac Nurse.

Alan's talk highlighted the difficulty of 'data wrangling' - while writing code itself may be straightforward, working out how to subset, view and potentially reorganise the data, or deciding how to deal with missing values, can be very difficult, and may affect the validity of results. Automating this process can also take a lot of time, but is important for ensuring reproducibility of results. Mapping the results from computational analysis back to theory is particularly challenging. How can we determine how the results relate to the real world? What can we learn from them? Finding a narrative that fits with established theory can be difficult.

Chiara Del Vescovo

Chiara is currently a Data Analyst at the BBC, where her job is to manage the semantics of programme metadata across the organisation. She also has industry experience as a Data Architect and has a PhD in description logic and ontologies.

Chiara discussed the challenges of working with PIPS, a complex, centralised system that connects to and stores data from other systems regarding BBC programmes and resources, and which both software processes and humans interact with (Suppl. material 2). Her work involves translating from PIPS (the code) to data semantics (the theory): PIPS is considered a source of truth and influences how people understand the information that it contains. Although data would have semantics associated with it when it was initially created in the system (theory to code), this often changes over time due to data being used

in a different way to that originally intended, and the current state of the system informs or provides a new view of how it should actually be considered (code to theory). Depending on who is using PIPS, the semantics may change. A focus is abstracting and formalising the semantics of the data and producing a representation that is common to everyone so that all stakeholders talk the same language.

Nicolas Gruel

Nicolas has worked as a Research Software Engineer at the University of Sheffield and the University of Manchester (his current role). He has a PhD in Astrophysics and has previously worked as both a theoretical and observational Astrophysicist and in the medical domain.

Nicolas emphasised that when translating between theory and code, experimentation and observation to test any resulting theory were paramount (Suppl. material 3). Validating *in silico* models is difficult: nature is complex, and models are often too reductionist and do not work in every situation. People sometimes have such faith in computational models that they refuse to accept they are inaccurate, even when they are not supported by the data (the implicit assumption is that the empirical data is wrong). Experimentation is expensive, however, and there is optimism that *in silico* experimentation can replace *in vitro* or *in vivo* work, even when the evidence suggests this is not the case.

An additional issue is that a great deal of research code (and in Nicolas' experience, in medicine) is of poor quality, and very inefficient. To improve this, we need to improve adoption of version control, documentation, QA and maintenance.

Abstract

Increase in computing power has made it much easier to implement theory in code. *In silico* science has now been introduced and is used in a wide range of disciplines, where models are implemented through the use of software, itself often considered as a "black box". The results of those models are then considered as truth. In this rapid adoption of the *in silico* approach, a significant (and expensive) step is too often skipped: the validation of the models in the real world. Models are a simplification of a real process and their numerical implementation can only introduce more approximation. There is a need to reintroduce critical experimentation to validate models and their implementations.

Anja Le Blanc

Anja is a Research Software Engineer and Research Applications Manager who has worked across many domains, from modern dance to economics and heliophysics.

Anja described her experience working on two successful research software projects. The first was a software tool using Access Grid technology to combine multimedia data (images, movies, 3D projections) and modern choreography. This involved engaging with dance students and dancers through workshops, in an iterative process, and ultimately

publishing a paper about the work. The second project was an EU-funded project about introducing workflow systems to heliophysics, which involved heliophysicists from across Europe. Students were ultimately able to use the software to explore data in completely new ways, enabling new science to be conducted.

Abstract

The choice of software engineering tools is very domain dependent and not necessarily suitable for the task. When performing theory to code translation, the primary issues are that boundary conditions are badly coded, testing is non-existent, and there is not much software engineering. When translating code to theory, it is important to check whether code does what it says it does in all circumstances and to watch for assumptions (within the code) that are not clearly stated.

David Mawdsley

David is currently a Research Software Engineer and Data Scientist at the University of Manchester, where he is working on the automated behavioural coding of video data. He has previously worked as an analyst at HEFCE, where he gained experience of translating funding models to code. More recently he was a postdoc at the University of Bristol, working on Bayesian models for including dose-response and longitudinal information into network meta-analyses, drawing on his background in medical statistics. This involved translating the statistical theory and model into code for the MCMC samplers JAGS and Stan. David has a PhD in Physics.

David discussed how dissemination of work to collaborators can be challenging, and how technologies and libraries such as R and Knitr (R+LaTeX) can be used to provide a clear demonstration of what is happening in the code and support reproducible research (Suppl. material 4). This approach offers a way of packaging theories for sharing, providing ‘reusable’ theories that are easy to reimplement and test.

Abstract

One of the challenges of translating theory into code (and vice versa) is the barrier between the users of the code (who may well drive the theoretical basis of the models forward) and the producers of the code. One way of reducing this barrier is to provide the users of the research with easy to use and extensible versions of the code implementing the theory. The R "ecosystem" can facilitate this (and the production of reproducible research outputs) via the use of knitr, custom R packages and Shiny.

Dale Mellor

Dale is currently a Scientific Software Developer in the Cosmology Group of Astrophysics, University of Manchester. He has previously worked on hydrometeorological (stochastic) and hydrological (finite element) modelling. He has a PhD in maths.

At present, Dale is writing code to implement cosmological theory. He talked about the importance of understanding what the code is doing, both for testing and the removal of bugs, but also in order to interpret the outcomes of research (Suppl. material 5). To assist with this, it is important to display as much insightful detail as possible.

Eleni Mikroyannidi

Eleni works as a Data Architect at the BBC. Her responsibilities include the design of back-end data architecture, domain modelling, establishing data workflows, data maintenance and data quality assurance, and defining metadata strategy. Communicating strategies/solutions to various stakeholders, both technical and non-technical, is also a key responsibility of her role. She has a PhD in ontology engineering.

Eleni's talk covered her experience working at the BBC, managing data related to their sports, education and children's sections. This brought multiple levels of translating code into theory, with the challenges depending on the task at hand. Within agile development, there are workflows and schemas that support communication with others and help solve problems. For example, you move from code to theory when investigating an error within a system, conducting root cause analysis to come with a solution (which often feels like searching for a needle in a haystack due to legacy nature of the systems, although new systems can cause problems too).

Abstract

It is important to understand the nature of the underlying theory you are trying to embody in code, and why you are doing it. Systems are often heterogeneous, with different code, languages, data, teams and stakeholders. To understand theories from people with different expertise, you need to define a terminology that can be included in a data model. This process involves cycles of investigation, where communication can be supported by diagrams. It sounds chaotic, but it can also be fun.

Important aspects of code to theory translation:

- Abstracting away at the right level; architectural diagrams can do that well.
- Choosing the right vocabulary for communicating ideas to different stakeholders.
- Using the right examples for explaining to stakeholders.

Important aspects of theory to code translation:

- Having a clear understanding of requirements.
- Separating functional and non-functional requirements and highlighting features on MVP.
- Breaking down big tasks (epic work) into small implementations.

- Having a realistic timescale for the implementation of theory into code.
- Legacy systems can ruin your theory.
- Keeping realistic goals within your theory.

Richard Rollins

Richard is a Scientific Software Developer at the Jodrell Bank Centre for Astrophysics, where he works with the cosmology research group to create galaxy catalogues for the Dark Energy Survey. He has a PhD in astrophysics and previously worked as a Software Consultant at the DiRAC National Supercomputing Facility.

Richard uses image processing techniques to measure the shapes of galaxies from noisy optical images which can be used to help constrain theories for how our universe formed. In his experience working with research physicists, limited compute resources are often cited as a bottleneck in their ability to do science (Suppl. material 6). In fact, code quality and efficiency is a common and significant issue as people rarely know how to optimise appropriately for given hardware and fail to apply software engineering best practices. There are also open questions around the best statistical representations for large datasets when confronting them with theoretical models; it is not uncommon to end up with multiple different software pipelines which require a rigorous framework in order to be compared. Understanding the relationship between code, infrastructure and user, building around robust frameworks for analytics (data structures, parallelism, etc.) and providing clean APIs are key in allowing researchers to effectively and efficiently constrain theory from data.

Abstract

Cosmology is a field with much expertise in algorithms but far less experience in other areas of computer science including hardware, high-performance computing and software engineering. This presents an ongoing challenge as new code is needed to effectively leverage new hardware for ever more ambitious projects. Robust APIs lead to robust, reusable and extensible software that allow scientists to spend more time doing science. Promoting the value added by software developers and engineers in research is important to break through perceived ceilings in code quality among researchers.

Andrew Rowley

Andrew is a Senior Research Software Engineer, currently working as the Lead Software Engineer for the SpiNNaker tool chain, as part of the Human Brain Project. He was previously a Senior Research Software Engineer at the National Centre For Text Mining, Manchester and a Senior / Research Software Engineer at Research Computing.

In the Human Brain Project, the SpiNNaker processor is being used to build a system capable of simulating 1% of the human brain. Andrew's work involves moving from theory to software, running neural networks that are built in Python, and theoretical models built

manually, which then create data (Suppl. material 7). When translating from data to theory, it is often not understood in advance what the aim is. Goals are to create models that can run in real-time and to provide an API framework that neuroscientists can use. The latter is challenging, as researchers continually have new requirements that cannot be supported by the current version of the API, and need further development.

Abstract

One of the biggest challenges in translating theory into code is understanding; that is the understanding of the theory by the software engineers and the understanding of software engineering by researchers. This becomes more apparent as larger and more complex software tools are required to get the research results, and so the software starts to require a level of engineering to ensure that it is correct and producing correct results, while at the same time allowing new extensions to allow new research ideas to be incorporated. It is unlikely that all researchers can be expected to become software engineers themselves, and equally research software engineers can't be expected to gain full in-depth knowledge of all areas of research. Thus there must be a level of communication between these two parties to ensure that the final software actually implements the theory, as well as an understanding of the importance of getting the code correct.

Julio Vega

Julio is a PhD candidate in the School of Computer Science at the University of Manchester. His research examines how routines that can be monitored via mobile phone affect Parkinson disease progression.

Like Alan, he stressed the difficulties caused by data ‘wrangling’, and how filtering data and interpreting the results of analysis can be challenging. It can be daunting trying to integrate different data sources and transform disparate formats into one suitable for analysis. He also described the difficulties making comparisons across models.

Why is code/theory translation challenging?

In the first breakout session, participants were asked to identify the key challenges of translating between code and theory. The question was deliberately high-level, to give participants the freedom to discuss the issue in whatever terms they thought relevant.

Participants spoke about their own experience, and therefore covered a variety of contexts, including writing scripts for data analysis, creating platforms for workflows, dealing with databases with complex semantics, and large-scale computational models and simulations. The common thread running throughout was that the software was ultimately used, at least in part, for creating new knowledge or understanding.

A number of interrelated themes emerged from the discussion, covering both the technical difficulties of writing software and the difference in both the knowledge and values of those who wrote software for research and those who did not. Throughout the discussion, participants highlighted the importance of communication and teamwork. The themes are discussed in more detail in the subsections below, with an explanation and examples for each.

Designing software

As scientific theory is continually changing, software applications that represent it are often highly bespoke, and difficult to design: ‘Given there are a number of unknowns, how do you design a plan?’

Constraints and abstractions help to narrow the options and allow you to focus down on the problem: ‘by necessity, you have to concretise things. Computation doesn’t do undefined variables well.’ This has both positive and negative aspects. Programming necessarily requires precision, which can mean that the code is at too great a level of abstraction, and may not adequately represent the theory: ‘you need to be narrow to code things, but too narrow might be lossy where the theory is concerned. Going too narrow too early can be problematic.’

There is a tradeoff between the quick return offered by hard-coding values when they are already known and the fact that in practice there often turn out to be later changes to a model or unanticipated uses for the software. In astronomy, for example, values are often hard coded, as software is used for the analysis of a particular galaxy. Code would then be tweaked as required, for the analysis of further galaxies. Although this approach often does the required job, coding more flexibly would widen the cases that the software could cater for much earlier on, potentially saving time in the longer term. Developers were not against hard coding *per se* - sometimes, when you are trialling an approach or an idea it is a sensible time-saving measure - but there comes a time when ‘hard coded values are problematic’, and parameterisation is necessary. If you have a model which is flexible, it is easier to adapt it as you acquire new understanding.

Sustaining software

The bespoke nature of scientific software also causes problems in terms of its sustainability. RSEs recognise that for many aspects of novel software development they may be ‘reinventing the wheel’, but it is often quicker to write something from scratch than fix something that has been badly designed.

There is a related tension between deliberately developing software that is reusable, and ‘one-shot’ software that is intended for a single purpose, and often a single context or instance of a problem. There is pressure to produce software as quickly as possible: making it flexible is secondary, with the primary target being to meet the goals of the project.

Certain practices were identified as helping with sustainability: making sure code is documented; ensuring that more than one person is familiar with the codebase (‘code rarely survives beyond six months of somebody leaving’); using source control; using virtual machines and containers such as Docker.

It was often the case that people adopted version control out of necessity: ‘initially I had my scripts, but as things got complex, I started using Git and Jupyter. Git was a game changer’; ‘when people start having problems (losing code) is when they start adopting version control.’

Using containers was recognised as a step forward for reproducibility: ‘Docker is imperfect but is a step ahead in that direction. It’s a good way of dealing with different versions.’

Alongside the emphasis on documentation, was a desire for improved communication between developers, particularly when it comes to understanding someone else’s software, and whether it is suitable for a particular purpose: ‘are there ways in which we can try code faster? Sometimes you really want to ask someone, “can I do this with your code”?’.

There were also positive examples of design for sustainability paying off: a platform for data collection started to be used more widely, beyond its original remit. Although it had to adapt to the needs of new customers, ‘thanks to Git and similar,’ it was possible to scale it.

Misunderstanding of the domain

One of the key challenges faced by RSEs was getting to grips with new and diverse domains. In some projects, RSEs were not invited to the meetings where ‘the science’ was discussed, as their expertise was not viewed as relevant. As a result, it was often a struggle to understand the nuances of the science they were trying to implement in code, arising from a gap in understanding between domain experts, and RSEs: ‘you make many assumptions of what is expected from you and changes or new requirements are given halfway - you are expected to know everything beforehand.’

As the aim of any software developed was to encapsulate some form of theory or idea, the person with the expertise in this area should drive the functionality. As a software engineer, ‘sometimes you lose the feeling of what you are doing.’ To help ensure the software was on track, it was viewed as important to have the theory person in the testing loop. Ideally, they would ‘go away and think about results and see if they seem as they would expect.’

Misunderstanding of the software engineering process

It is common that the non-developers within a project (who may be researchers, academics, or other users), do not have a good understanding of the software engineering process. This is not a problem in itself, but it starts to cause difficulties when it leads to unrealistic expectations of what can be delivered: ‘in architecture, big delays in delivering are taken for granted while in software engineering a minimal delay is outrageous.’

Participants reported ‘fighting against PIs of the project who have preconceived ideas and want fast outcomes, which may ultimately cause problems’, and the fact that ‘building software is not regarded as valuable’ was also a common complaint: ‘people spend ages learning how to use equipment, but not code - there is less love for code than lab equipment.’ Significant amounts of money are still used to develop facilities, but less thought is given to software. There was a perception that: ‘researchers typically want something that works quickly, independent of its quality. We need to explain to people that software quality matters.’ There was a lack of appreciation for the skill that effectively engineering software required, and that there would be a balance between ‘the quality of the code, and the quality of the results.’

It was not just the process of building software, but also software itself that was perceived as undervalued: ‘papers are the currency for scientists. What is the role of software in terms of scientific outputs?’

The uncertainty of the scientific process meant that writing software properly was a gamble - ‘is it worth spending time writing good software if it’s going to get thrown away?’ It is still important to be able to express/explore early ideas in software prototypes, but these are entirely different products to the software that is used to actually do research. It is essential to recognise the transition, however: ‘It’s important to realise where you are in that situation.’ Doing this is tricky, and was viewed as a continuum, rather than binary.

There was an interesting paradox, where people failed to publish their code because they were worried about it being incorrect - even if it underpinned an important scientific paper - as there is a risk in sharing code: ‘what if someone finds a bug? The stakes are too high.’

Misunderstanding of computation

The fact that many researchers (including some of those who write code), lack an understanding of the software engineering process is well-documented. Jay et al. (In press) The present discussions revealed that this issue also ran deeper, however, manifesting itself not just as a lack of understanding of software engineering, but of computation itself: ‘not all scientists understand what software means - it is beyond their interest.’

There was a concern that within the scientific community at large, there is ‘a lack of understanding about how computing processes work.’ Researchers tended to view software as a ‘black box, which is trusted’, and this perspective leads to ‘blaming bad data or bad data collection when results are not expected.’ Treating software as a black box was viewed as reasonable for proprietary software - where there wasn’t an alternative - but it should be recognised that software produced for research is not necessarily perfect. There was a concern that researchers ‘not only do not understand the complexity of software, but they don’t know what coding is or what they want from it.’

One of the key issues raised was the importance of understanding that the implementation of theory, or the output of the results, will depend on the software being used. A researcher used to traditional statistical solutions wanted to view results as concrete or absolute,

whereas in reality, the results will depend on the implementation: ‘the system doesn’t have a floating point, this doesn’t mean this is wrong as floating point does not guarantee precision’; ‘it’s not wrong, it’s a different way of doing it.’ For example, ‘you can solve differential equations in different ways. Reproducibility is dependent on the system being used and the outcomes are different.’

In some cases, the implementation disagreed with models generated in the lab and elsewhere. Evaluation of software representing theory was critical, but also very difficult, as instances were cited of people choosing to ‘believe’ the software outputs over the data for real experimentation.

A final issue noted was that while not understanding software was clearly an issue for non-technical people, it was also a problem for software engineers themselves: software created to model theory or advanced science is often highly complex, and even those who create it can struggle to understand its intricacies and behaviour.

How can we improve code/theory translation?

In the second breakout, participants were asked to consider potential solutions to the challenges that emerged in the previous discussion. Again, there was freedom to discuss anything that people felt to be of relevance, rather than stick rigidly to addressing issues point by point.

The themes identified in this discussion relate to improving communication between different roles, increasing training, and changing research culture, such that the value of software within research is properly understood. The overriding message was that code/theory translation was now central to the scientific endeavour, and the most important way of supporting it is to ensure it is viewed as a whole team responsibility.

Communication

Everyone agreed that the best science happened in cohesive teams and that the ‘human element’ was an essential but elusive ingredient for success. Co-locating people in different roles - such as domain experts, software engineers and statisticians - such that they could talk regularly, and get used to each other’s technical language was viewed as essential, particularly in interdisciplinary research: ‘planning and designing with a diverse team is important’; ‘no one person knows everything - you need good communication skills for this, but that can be an overhead’; ‘lots of bottlenecks happen… but lots of creativity happens at the interface between two people; ‘code to theory is often one person, but there are lots of benefits of using 2+ people with different/same/overlapping skills’.

Understanding theory, and then trying to implement it in code, is an extremely complex process; domain experts often need time to understand the theory, so for software engineers outside the domain, it is particularly difficult. It tends to be impossible to specify requirements in full upfront; even specifying them partially can be a challenge: ‘the goal in

research is often a moving target, therefore communication becomes essential.' An area where it was viewed as vital to have significant involvement from domain experts and theoreticians was in the evaluation of the computational model (or other virtual representation of theory): 'it's important to validate as we're going along to show the theory is being followed'.

Interpersonal skills are essential too. Sometimes the issue 'may not be a software engineering issue, but a human factor.'

Tools and training

Training in software engineering skills early in the research process was viewed as essential: 'can we train researchers early enough in basic software tasks, (testing, continuous integration, version control) that they value it naturally? Even in the Arts, these days, data sizes are increasing to the point that software skills are needed to manipulate them. It might take a generation to bubble through.' At present there is a 'perceived ceiling that by the end of PhD people will have enough computational knowledge to do high-level research without software engineering experts,' but it was felt that in reality, many researchers' skills remain woefully inadequate.

From the perspective of software engineers, it is helpful to manage expectations, in terms of explaining what is possible technically ('making people understand that software is not easy to get right is key'), and the implications of particular approaches: 'the theoretician says to do something; the software engineer implements. It's not that simple. There is a dialogue. We need to find a sweet spot, probably through tools that enable quick feedback, and are able to increase understanding'; 'sometimes there does not appear to be any need for communication as only ONE person is working on a project. Then somebody else inherits the project, but nobody has documented the work before handing it over.'

Tools such as Jupyter and electronic notebooks were thought to be very valuable in terms of both retaining knowledge, and as a training tool: 'so working software can be seen next to theory.'

The value of software in research

Software has the potential to be both a tool for, and an outcome of, research. But while the written expression of theory (words or formalisms) is constructed for human understanding, and lab equipment has an obvious physical presence, software is both difficult for humans to read and understand, and intangible - essentially, unseeable, and unknowable: 'software may not be hidden in the sense that it is openly available, but people don't appreciate what it is doing, how much it is doing, how important it is.' Formally recognising the role of software engineering - and software engineers - was viewed as very helpful in this regard: 'if you've only ever seen what can be achieved by a 'code-savvy' PhD student then seeing what a room full of RSEs can do will be revelatory.'

Interesting questions that arose here included: how do we raise the profile of such software?; to what extent do people need to understand the software to understand the science?; how do we balance opening code and the need to commercialise? There was a view that publishers and funding agencies should push for code to be open, but there was also a concern that current publication models ‘are sometimes effectively a thin paper wrapper around software, and simply credit academic staff, not software engineers.’ Academia should not only recognise papers but should also acknowledge the value in software: ‘having software as a second class citizen is dangerous given that it is so often core to the research process. By not crediting software properly it stalls scientific progress: ‘there is no time to write it properly, no way to extend a career off the back of it.’

If we are to consider software as an output of research, this raises the question of how we disseminate and evaluate it: ‘it’d be great to have a platform where code was within the workflow of writing papers, as it is difficult to use and modify the source of others (when it is available).’ A scale of research software 'robustness', from ‘exploratory/throwaway’ to ‘publishable’ was viewed as useful, possibly modelled on the TriBITS Lifecycle Model.*1 The peer review of code must be addressed: ‘reviewing software is a large burden and shouldn’t be underestimated. Is there a way to automate reviewing software at all?’ The view that ‘we need to be careful about shaming people for exploratory code’ was also overwhelmingly expressed.

Conclusions

The overwhelming message from this first Code/Theory workshop was that science was a team effort, and cohesive working across all the people involved in a research project was essential to its success. While this in itself is not a surprising conclusion, it is probably true to say that from the perspective of many of the RSEs, it was not always reflected in the reality of their experience. This may be an artefact of the academic environment. Software is not currently valued in the same way as more tangible expressions of theory, such as words, or mathematics, and as such, software engineering is not valued in the same way as better understood parts of the research process, despite the fact that it is integral to the production of results.

RSEs do not write code in a vacuum, but play a key role in research activities: hypothesis generation; study design; data analysis; interpretation of results. Everyone in a research team is essential to its success, and while there is a natural division of labour between RSEs - who spend more time writing code - and other researchers - who spend more time writing papers - all of this work is core to the scientific endeavour. As illustrated in Fig. 1, while some of us work on the software necessary to produce results, and others work on the surrounding story necessary to explain them, in team-based research we are all *scientists*. Recognising this fact, and embedding this message in academic discourse, is key to securing the future of research.

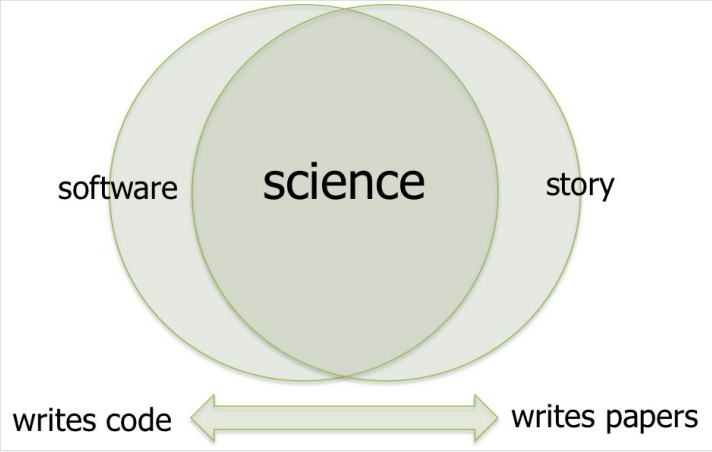

Figure 1.

In many areas of research, science is now produced at the intersection of the 'software', contributed by programmers, and the 'story', or theoretical narrative, contributed by domain experts. In team-based research, everyone is a scientist.

Acknowledgements

The workshop organisers would like to thank all the participants for their time, experience, expertise and enthusiasm.

Thank you to the Software Sustainability Institute for funding this workshop through Caroline Jay's Fellowship. The work carried out by the Software Sustainability Institute is supported by the UK Engineering and Physical Sciences Research Council (EPSRC) through grant EP/H043160/1 and EPSRC, BBSRC and ESRC Grant EP/N006410/1.

The workshop was organised by Caroline Jay, Robert Haines, Markel Vigo, Nico Matentzoglu and Robert Stevens, and hosted at the School of Computer Science, University of Manchester, UK

References

- Jay C, Sanyour R, Haines R (In press) “Not everyone can use Git”: Research Software Engineers’ recommendations for scientist-centred software support (and what researchers really think of them). Journal of Open Research Software URL: https://www.research.manchester.ac.uk/portal/files/53599032/JayCaroline_2.pdf

Supplementary materials

Suppl. material 1: Lightning Talk: Jonathan Boyle

Authors: Jonathan Boyle
Data type: multimedia
Filename: Boyle.pdf - Download file (1.66 MB)

Suppl. material 2: Lightning Talk: Chiara Del Vescovo

Authors: Chiara Del Vescovo
Data type: multimedia
Filename: DelVescovo.pdf - Download file (101.13 kb)

Suppl. material 3: Lightning Talk: Nicolas Gruel

Authors: Nicolas Gruel
Data type: multimedia
Filename: Gruel.pdf - Download file (124.79 kb)

Suppl. material 4: Lightning Talk: David Mawdsley

Authors: David Mawdsley
Data type: multimedia
Filename: Mawdsley.pptx - Download file (849.91 kb)

Suppl. material 5: Lightning Talk: Dale Mellor

Authors: Dale Mellor
Data type: multimedia
Filename: Mellor.pdf - Download file (3.38 MB)

Suppl. material 6: Lightning Talk: Richard Rollins

Authors: Richard Rollins
Data type: multimedia
Filename: Rollins.pdf - Download file (4.20 MB)

Suppl. material 7: Lightning Talk: Andrew Rowley

Authors: Andrew Rowley
Data type: multimedia
Filename: Rowley.pptx - Download file (4.67 MB)

Endnotes

[*1] https://trilinos.org/trac/trilinos/wiki/TribitsLifecycleModelOverview