Managing Academic Software Development

Dr Sam Mangham

Research Software Group

ORCID:0000-0001-7511-5652

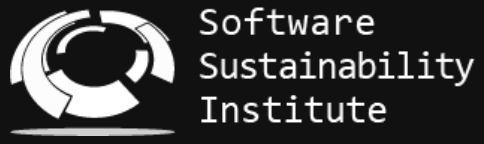
University of Southampton

Speaker notes

Hi everyone, I'm Sam Mangham, and I'm here to deliver this talk on Managing Academic Software Development.

Who Am I

- Senior RSE @ University of Southampton
- Trustee @ Society of Research Software Engineering
- RSE @ Software Sustainability Institute
- Generalist, interdisciplinary RSE, training, community

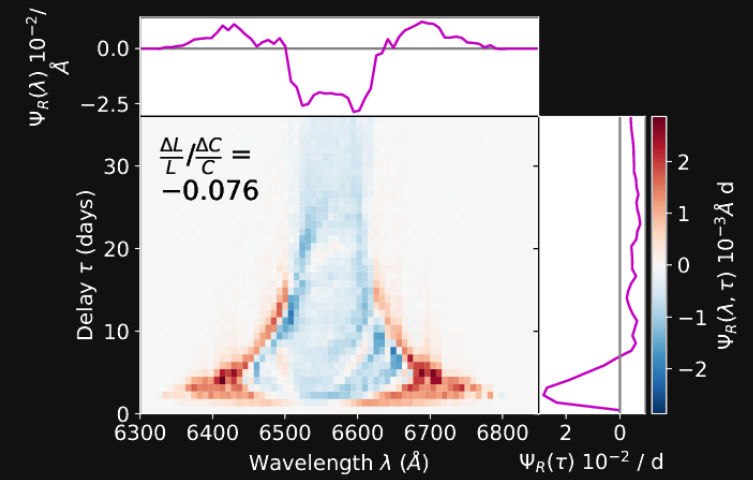


Speaker notes

So a quick bit of background - I'm a senior research software engineer at the university of Southampton, as well as being one of the RSEs working for the Software Sustainability Institute, and I'm a trustee of the UK's Society for Research Software Engineering.
As a software engineer I'm an interdisciplinary generalist, and as an RSE I've worked on everything from web-based machine learning platforms to HPC codes for astrophysics. I'm also involved in training and community work - if you're in a GMT timezone, ask me about the Society of RSE's mentoring program later on!

Background

- PhD in Astrophysics
  - HPC monte carlo radiation transfer code for supermassive black holes
- Neutronics @ Culham Centre for Fusion Energy
  - HPC monte carlo radiation transfer code for fusion
- Both large legacy HPC codes!



Mangham et al, 2019, ESO/M. Kornmesser

Speaker notes

My background, though, is solidly in HPC. I originally worked as a radiation physicist in the UK national fusion labs, running monte carlo simulations of the radiation levels in fusion reactors, and tracking the neutron-induced activation in the materials. In addition to expanding the modelling software, I also worked on developing tools and workflow improvements for our research group.

Then I left to get a PhD at Southampton, working again on monte carlo models of radiation, this time around super-massive black holes to determine how light echoes can be used to probe their outflows. As well, because I'm a born RSE, I put a lot of effort into trying to get the code documented and tidied. I also, along with all the other people working on the code, tried to persuade the professor who wrote it that originally it really, really shouldn't be named "Python" any more. We were unsuccessful.

But anyway both of these projects were old, large HPC codes - FORTRAN77 and inexplicably pre-ANSI C - so I have some relevant experience for this group!

Why?

Speaker notes

But to begin - why am I giving this talk? What do I even mean about how to manage software development, and what in particular is special about it being *academic* software?

Enterprise

- Often large teams
- Formal training
- Formal project management frameworks & staff
- Software is the product

Academic

- Small/single teams
- Large numbers of loose collaborators
- Limited training
- Ad-hoc management (by other researchers) or self-management
- Papers are the product

Research Institutes

- Somewhere in-between
- Vary with scale, focus, discipline

Speaker notes

There's a lot of advice on the internet about how to manage a software project - we're probably all familiar with terms like Agile or Waterfall. But most of it comes from the perspective of industry, developing enterprise software. It's usually calibrated for large teams, working within a formal project management framework, usually with specifically-employed project managers, and focuses on how to produce a product to a tight deadline. In academia, however, things are pretty different. There are far more solo developers or small teams, most with limited or no formal training in software development, who have to self-manage their time, and who aren't really employed to produce software - it's papers that are the product. Software is simply a by-product. As a result, a lot of advice from industry is just a bit much, and it doesn't help that there are entire industries based on overcomplicating it so they can deliver expensive training courses.

Of course, research institutes and national labs lie somewhere between the two extremes. Those with large, well-funded teams might have more of the formal, industrial-style project management, whilst smaller groups often more closely resemble the loosely-structured environment of academic software development.

Regardless, the sheer volume of material generated on project management suggests it's an important task. Academics typically work on many different projects, all cutting-edge and complex, with a wide range of collaborators, but with very little training or support in how to manage them. This talk is intended to provide an introduction to relatively simple tools and practises that can help you structure your development a little better, hopefully producing more reliable, sustainable software. It's not intended to be exhaustive, or proscriptive though - take what works for you and makes your life easier.

Outline

- Development
- Usage
- Publication

Speaker notes

I'm going to cover three main components of academic software.
Firstly, how to keep its development clear and on-track, and make sure the code you're writing will enable easy development in future.
Then I'll quickly cover how to manage its ongoing use to maximise usability, and then put a bit more depth into how you can manage the publication of your software. You might be familiar with some, or even large parts of this - but hopefully everyone will find at least something useful to take away.

Managing Development

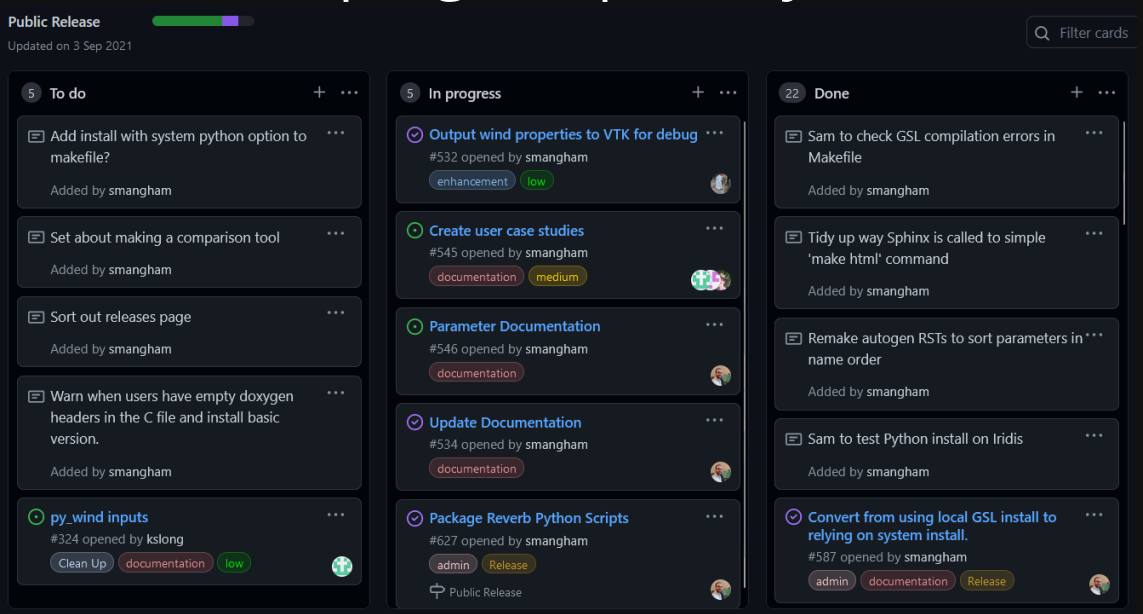
Speaker notes

I'll kick off with some best practise on how to develop your software.

Project Boards

"Programmers tend to start coding right away. Sometimes this works." **- Eric Larsen, 2018**

- Break a project into components
- Subdivide as you go!
- Track progress publicly



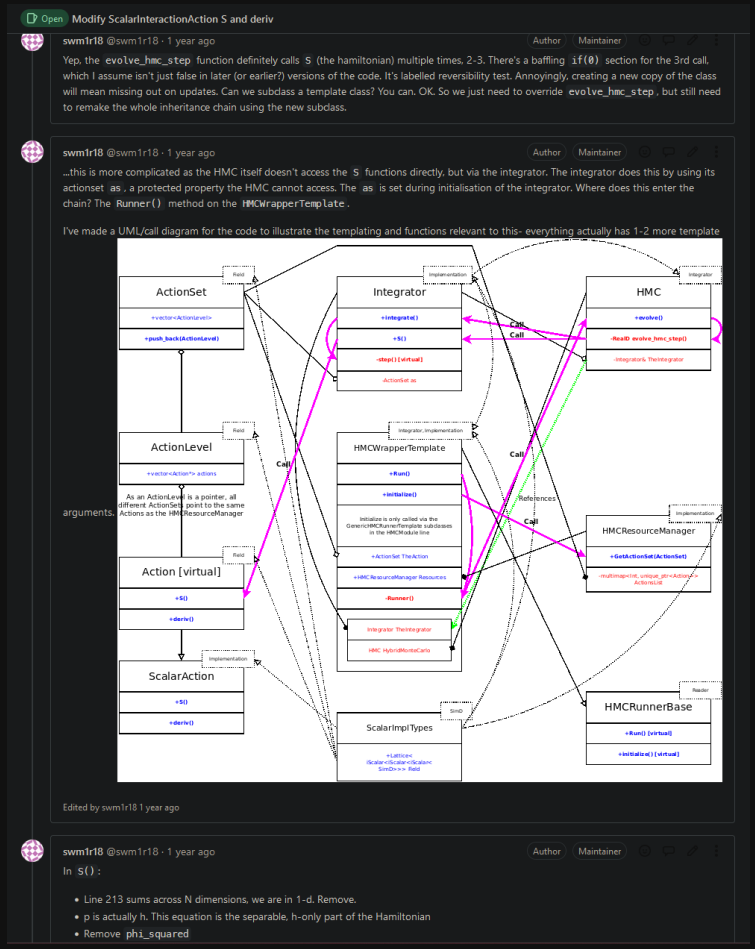
Speaker notes

A common approach to academic software development is to just... start coding. We know what we need to do, we know how to do it, let's just get on with it. This works, relatively often for small, simple projects, but for anything substantial you quickly hit the limits of it. You end up with projects that hit unexpected roadblocks, spiral in size as every meeting and new collaborator generates a new avenue of research, and quickly become unmanageable messes where nobody really knows what anyone else is doing, or even how far they are through what they agreed to do in the first place. In order to make our development process more sustainable and accessible, we need a way to describe it and structure it.

Fundamentally, any project, software or not, can be broken down into a series of discrete tasks. Breaking a project down into a series of small, descriptive and self-contained tasks forces us to think through what we're actually doing. A common tool for this is the Kanban board, or project board. They're intended to provide a visual depiction of your development process and are available integrated into repository hosting sites like GitHub and GitLab or independently on sites like Trello. They allow us to display and track those tasks as cards on a board, moving them between To Do, In Progress and Done columns - and potentially breaking tasks down into smaller subtasks too where they turn out to be too large. These give us clear illustrations of where we are in the project, and what's left to be done.

Project Boards

- Document process on tasks
  - GitHub/GitLab etc. let you turn issues into lab books
- BUS FACTOR
  - Collaboration
  - Future You is a collaborator
  - Knowledge decays quickly



Speaker notes

Most sites that manage project boards will also allow you to comment on tasks, attaching links, pictures and text. This allows a project board to function as something analogous to an experimentalists' lab book, allowing us to keep running notes on how we're accomplishing the task, what problems we've run into along the way and the steps we took to resolve them.

It's important to keep this clear documentation of your development process. There's the concept of the Bus factor in coding - how many developers on a project would need to be hit by a bus in order for it to become unrecoverable? For most academic software, the bus factor is one. If that one developer becomes unavailable due to illness, or leaves academia, what remains is an incomprehensible mess with an undefined amount of work left to do. Without clear task management, it's also hard to collaborate with others - bringing a new PhD student onto the project involves having to dump a vast amount of context and information from your head into theirs, and keeping track of progress and problems requires endless meetings. It's also a problem if you step away from the project, as in many ways Future You is just another collaborator. A huge amount of knowledge can decay in a short time whilst you work on another project; documenting the state of development makes jumping back in much easier.

Prioritisation

- Time estimates
- MoSCoW

- Consider and revise!

Speaker notes

One of the main problems with using project boards in academia is the initial scope of our projects is often poorly-defined, and academics are if nothing else generators of ideas - any attempt to do something will inevitably turn out to be far harder than is expected, because this is cutting-edge research, whilst spawning several ideas for new papers. It's much too easy to end up overcommitted, with too many unfinished features and a project board that accrues tasks faster than we complete them, with everything trapped forever in the "In progress" column.

The best way to avoid this is to prioritise and scope your project, which we can do more easily by borrowing a few techniques from the Agile methodology used in industry. There's plenty of guides for full implementations of the Agile workflow, but as mentioned before they generally rely on a much more structured work environment than we have, often with formal project managers and roles like 'product owner' that don't really exist in academia - so it's a lot easier to pare it back to just what fits our needs.

If we're using project boards to track the individual tasks that make up our project, we want to try and come up with rough estimates of how long each task will take. Once we have that, then for a given block of time, which in industry they'd call a sprint but could potentially be a month or two in academia where our schedules are messier and clogged up with teaching and conferences, we want to fill that with tasks and *prioritise* them. A common strategy for this is MoSCoW - Must, Should, Could, Won't, where only 60% of our development time in a block should go on tasks we think are essential for our project. A good 40% should be on things that are nice-to-haves but not essential. That way, if (or when!) our time estimates turn out to be wrong, we've already decided what to drop. Instead of dragging the task out forever, we decide we'll only compare 3 algorithms rather than 4, or we'll drop support for a particular input format or physical regime.

Crucially, though, we acknowledge up front what we *won't* do - what's just out-of-scope if we want to have a hope of finishing our code before the conference, or getting a paper out in the next 6 months. But we keep it around to come back to later.

Then, after a month or two of work, as tasks break down into subtasks and new ideas bubble up, we re-prioritise and consider the work we'd like to get done in the next couple of months. This might not work for your project - it may be that you genuinely *can't* think of any parts that aren't completely essential to your project, and with no deadlines it's genuinely a matter of 'it's done when it's done'. But even then it'd still be a good idea to take a leaf from MoSCoW and increase your estimate of how long it'll take by 2/3!

Prioritisation

- Won'ts aren't forever
- Typical won'ts
  - Future research avenues
  - Features you don't need right now
  - Bugs that don't stop work
- Acknowledge them publicly
  - Help others plan around you
- Leave time for testing & documentation!

Speaker notes

As mentioned, when setting up a project, it can be hard to decide what *is* a won't, though. Generally we still want to accomplish all of our ideas!

Won'ts aren't forever - they're just for the current chunk of development. So for a typical academic project, won'ts are likely to include potential avenues of research you open up during the project, but didn't budget time for. If you don't prioritise by explicitly allocating these things for the next paper, then you'll often try to cram them into the current one, turning it into a sprawling mess that often ends up having to be split *anyway*.

Other typical won'ts come from collaboration - it's a common problem to want to satisfy *all* the feature requests and bug fixes from your collaborators and colleagues. This spreads your time much too thin, and ends up turning a project into a game of whack-a-mole where you're just responding to the last email you got. If you don't have the time then it's bette to be up-front about that - let people know a certain mode just isn't coming for a few months, or a that particular bug in the regime they want to explore is too gnarly to fix right now. This lets people plan their own projects better instead of having PhD students spinning their wheels waiting for fixes that won't be coming any time soon, and pestering you for updates with emails.

Poor prioritisation and overloading the project with new features also often results in important parts of the project being neglected. It's common to end up publishing academic code packed full of interesting and powerful functionality, but without enough tests to prove it actually works or without any of the documentation required for anyone other than the developer to use it! Nobody would suggest, up-front, that there should be no documentation - so good prioritisation ensures that it doesn't fall by the wayside.

Version Control

- Protection against disaster
- Test and verify changes are *intended*
- Avoid having to rerun entire papers' worth of analysis to avoid version mismatches

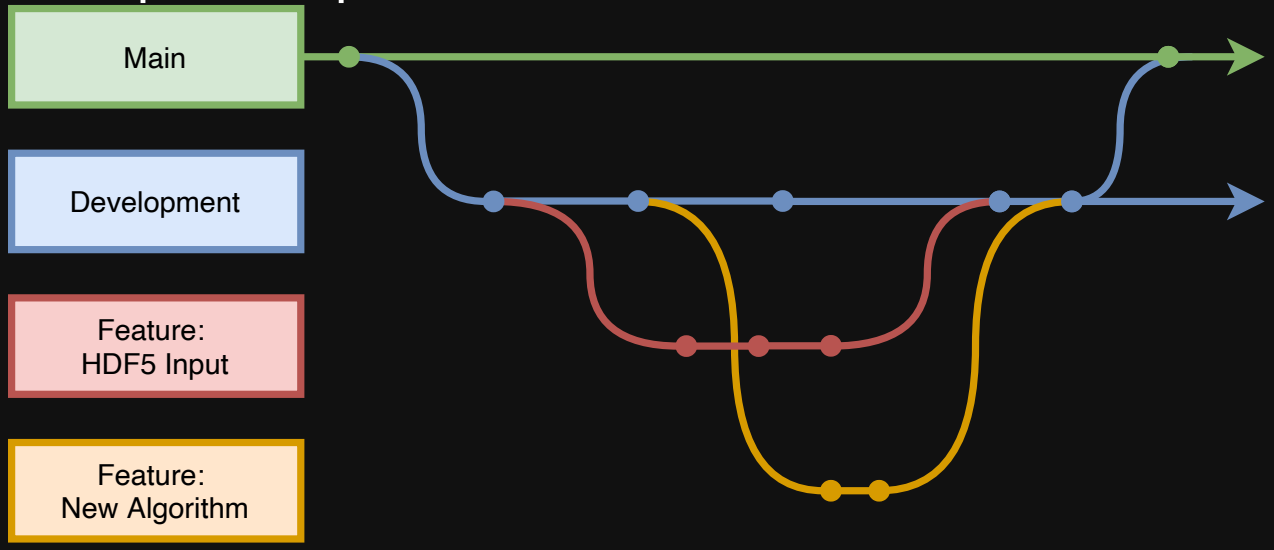
Speaker notes

I'm assuming everybody's code is already on version control - if your code isn't, please stop watching right now and get it set up. Version control and remote backups are absolutely critical to good project management, not least because if your building burns down as Southampton's computer science department did in 2005 you will continue to have a project and not just a pile of ash and tears.

In addition, having a continuous record of the state of your code is vital, allowing you to easily test the differences between different versions of code and identify where unintended changes to behaviour or code have snuck in, as well as being key for publication and reproducibility. Being able to roll back to the version of the code you used when generating a paper to respond to referee requests for revision is key to avoid ending up with papers containing a mishmash of results generated by different versions of the software, or wasting huge amounts of time having to redo everything in the paper!

Branching Workflows

- New branches for new features
  - Link branches to tasks
  - Easy to parallelise work
  - Easy to switch to working on another feature
- Regularly merge branches back to development!
  - Otherwise each developer ends up with a divergent version
- Review pull requests



Speaker notes

One of the key features of version control is the ability to have multiple parallel 'branches' of code. Common branching version control workflows are to have a main branch, containing the stable version of your code you'd share with users, and a development branch for work-in-progress versions. This model works particularly well with the kind of task-based, board-based project management styles discussed earlier.

Instead of trying to implement every single task on the development branch, you can easily parallelise. Each task that would take more than about one commit to do can be implemented as part of a new branch, isolating it from the others, making it far easier to implement and test without the risk of accidental bleed-over from other changes to the code. If one task proves more complicated than expected, or stalls due to a lack of data from collaborators, or the tests to validate it take a very long time, then meanwhile you can easily switch over to working on a different one. Better, you can easily collaborate without stepping on each other's toes. Ideally, these branches would then be merged into a development branch once they've been fairly rigorously tested, and that branch then into your main branch when you finish a version of the code that's being used in a paper.

It's important to keep the scope of branches contained, and to regularly merge them back to the development branch - or at the very least, if testing is taking a long time, routinely pull the development branch into them to keep them up-to-date. It's common for individual researchers to end up making all their changes to one branch they 'own', and end up with it so far behind the rest of the code that they just don't have the time required to merge it back. This can lead to entire projects worth of work becoming marooned in these divergent branches.

A common practise in industry, and one that you can even adopt in single-developer academic projects, is to make these merges as formal 'pull requests' - and sites like GitHub provide interfaces for these. You can set procedures that have to be completed before a branch is merged - ideally this would include getting collaborators on your project to peer-review it, but even solo developers can use this as an opportunity to take stock, run automated tests, make sure the documentation is up to date and so on.

Write Sustainable Code

- Proactively avoid technical debt
- Share and collaborate more easily
  - No code worth writing is disposable!
- Write for collaborators and community
- Can't reproduce results if the code isn't sustainable
  - HPC-BP talk on this

Speaker notes

We also need to consider how the code we write factors into the long-term management of our project. Poor or idiosyncratic code is hard to maintain, hard to share, and tends to lead to a buildup of technical debt that's hard to address. Decisions become opaque, harder to understand and harder still to refactor out. You might consider it not worth the effort if your code is only going to be a 'disposable' single-writer, single-user project, but code always tends to be more reusable than you think. One of the main problems with academic projects is how 'simple' codes slapped together as a proof of concept inevitably become part of the foundations of larger ones, in a way that undermines their long-term functionality. It takes far more effort to correct the problems once the code has grown, and if you don't your project will eventually reach a point where through years of PhD and postdoc churn nobody really understands how it works any more and the only practical solution is to rewrite it from scratch.

Plus, the idea that any code you write is a single-user project that nobody else will ever read or run is essentially assuming your research is irrelevant - if it generates important results, people will want to be able to understand and replicate them. Even thinking selfishly and in the short term, nobody has a perfect memory, and unless you can guarantee you can remember every single quirk and decision you make when writing then it's worth putting a little extra time in to write readable code to save you days of effort six months down the line when Reviewer 2 asks an awkward question and the information has leaked from your brain.

The seminar series has previously covered advanced techniques for reducing technical debt and improving sustainability like containerisation, but in this brief section I'll focus on the code itself.

Write Readable Code

- Easier onboarding
- Follow community standards
  - E.g. PEP 8 for Python
    - pylint, flake8
  - E.g. C++ Core Guidelines, LLVM for C++
    - clang-tidy
  - Pick a style and stick to it!

Speaker notes

Fundamentally, for a properly-managed, sustainable project you want your code to be as readable as possible. The more human-readable code is, the easier it is for you to expand your project by bringing in additional collaborators - and the more sustainable it is. Academic software will often pass through a chain of new PhD students, and onboarding them is always a time-consuming and complex process. If your code is full of shortcuts, quirks and idiosyncracies this also encourages the students to develop and add idiosyncracies of their own to it... and future development becomes bogged down by the complexities of understanding what, exactly, the last person to have their hands on the code actually did and if it's what they meant it to do.

The best way to avoid this, and to keep your project clear and comprehensible, is to stick to clear coding style. Most languages will have a couple of different style guides, that define conventions of spacing, braces, name formats and the like. The specific choice is usually less important than picking one and sticking to it. Whilst no existing style might be entirely to your taste, and there's always a temptation to just declare your own coding style, using one that's publicly defined and widely adopted reduces the mental load on people joining your collaboration and working on and using your code. It thus gets more science done, more quickly.

Checking style compliance is a common feature of linters, static code analysis tools available for most languages. In Python, PEP8 is the main community standard that's checked by a variety of linters including pylint and flake8, whilst C++ has things like the LLVM styleset you can check against using clang-tidy. There's even more aggressive code formatting tools like black for python or clang-format for C++, that automatically restructure your code to fit a specific style.

Write Readable Code

- Descriptive variable names
  - Minimise potential for collision!
  - Not 'c', 'e', 'hb'
- Code completion & IDEs
  - CLion, PyCharm, Visual Studio Code
- Modular code
  - You **will have to** refactor!
  - You can't predict your code's future

Speaker notes

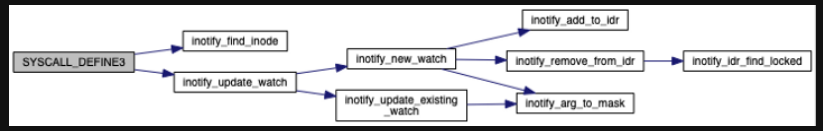
One major readability boost that can't be automated is choosing clear, descriptive variable names instead of the classic single-letters like 'a'. Whilst they might make it quicker to write your code, and arguably more closely resemble the equations you're modelling, you quickly run into problems with collisions as your code grows and adds libraries. Short names become ambiguous and confusing at best, and actively clash with other variables and functions at worst. Refactoring code to avoid these is a pain. And, again, they make the code more dependent on human memory - as it grows, you have an increasingly large list of things a developer needs to remember in order to be able to contribute.

Fortunately, most modern integrated development environments like CLion or Visual Studio Code support automatic linting, pointing out violations of code standards, as well as offering autocompletion making long, descriptive variable names just as easy to use as short ones. In addition, most have the functionality to mount remote filesystems, simplifying development on HPC.

Code is also more sustainable and readable when modular. Breaking code down as far as possible into well-defined functions with clear inputs and outputs makes for easier development. They fit more naturally into task-based and feature-branch workflows, are easier to work in in parallel, and are generally easier to refactor when you need to modify the structure or functionality of your code to enable future expansion. This last one is a key point - no code will ever survive a long time *without* needing refactoring effort at some point, as your initial design can't possibly take into account all the possible future uses you might want to put the code to. Clear, descriptive names in modular functions make it possible to restructure a codebase without the need for extensive edits to the code itself.

Document Your Code

- Bus factor again
- Optionally: Document then design
  - Test-driven development
- Automated tools
  - Sphinx
  - Doxygen
- Automatic hosting
  - ReadTheDocs for Sphinx
  - CodeDocs.xyz for Doxygen
- Call graph generation
- docs-like-code



Call graph, Christina Jacob, 2020

Speaker notes

However even the most human readable code is still not completely obvious to the reader, no matter how hard you try - and that means again most projects fall foul of the bus factor. We need to ensure that the context and justification of our code lives somewhere outside our head, again to enable us to bring new collaborators on-board and protect ourselves from the inevitability of knowledge decay.

If you've written your code in simple, descriptive human-readable code in modular blocks this should actually be fairly straightforward. We just need to document what we're doing and why, but without delving deeply into the mechanics of the implementation - because they should be clear from the code itself. In fact, you can even document *before* you write the code - defining what a function does, what it takes and returns, before you make the implementation. This can be taken even further to test-driven development, where you define the tests that your code would need to pass in order to function before you write it. Written in this way, it's much easier to expand and extend your software, as the functionality is clearly exposed to you and not buried in dozens of lines of code.

There are even community standards for writing these comments that allow them to be parsed by automatic documentation tools like Sphinx or Doxygen. These extract comments from the code and compile them into API documentation intended for developer use - easily searchable and a major timesaver. Then, websites like ReadTheDocs or CodeDocs.xyz allow you to link up GitHub repositories, automatically generating and hosting webpages. Once you have these, they make your life much easier - even experienced developers on a project will often find it easier to just search a website than rack their brains for the location of a function in a sprawling codebase. They can even generate call graphs, illustrating which functions call others allowing you to easily understand the structure of your code - particularly important when other collaborators begin to add to it. I've included a simple example here from a non-scientific project, as unfortunately the graph generator for code I wanted to demo has broken in my absence!

This also illustrates one of the common problems with documentation - it can go stale. Particularly with large,

complex HPC codes, outdated documentation can be a huge problem. You can end up wasting days attempting to use features or compilers that are *documented* as being supported, but have long since been deprecated - certain, the whole time, that they're supposed to be working so you must just be making a mistake. A development intended to counter this is the docs-like-code workflow, where the documentation for features is stored as close to the code as possible, and considered to have *parity* with it - when you edit the code, you edit the documentation to update it at the same time. You don't merge branches with undocumented changes that you'll 'write up later'. Of course, this is a lot easier to do in an environment where you have someone else reviewing your pull requests to keep you honest...

Test-Driven Development

- Continuous Integration
- Many more detailed talks on this!

Speaker notes

I briefly mentioned test-driven development on the last slide and I'll just skim through it as I can't do it justice in this talk - but there's a variety of tools designed to make writing code based around tests first easier. One common setup is continuous integration, using test frameworks to running automated tests on code before you do things like merge branches. Most repository hosting sites like GitHub and GitLab support continuous integration pipelines, and even allow you to hook up remote compute resources to them for tests you can't feasibly run on the free resources they provide.

Questions?

Speaker notes

So before we move on to the next section, does anyone have any questions?

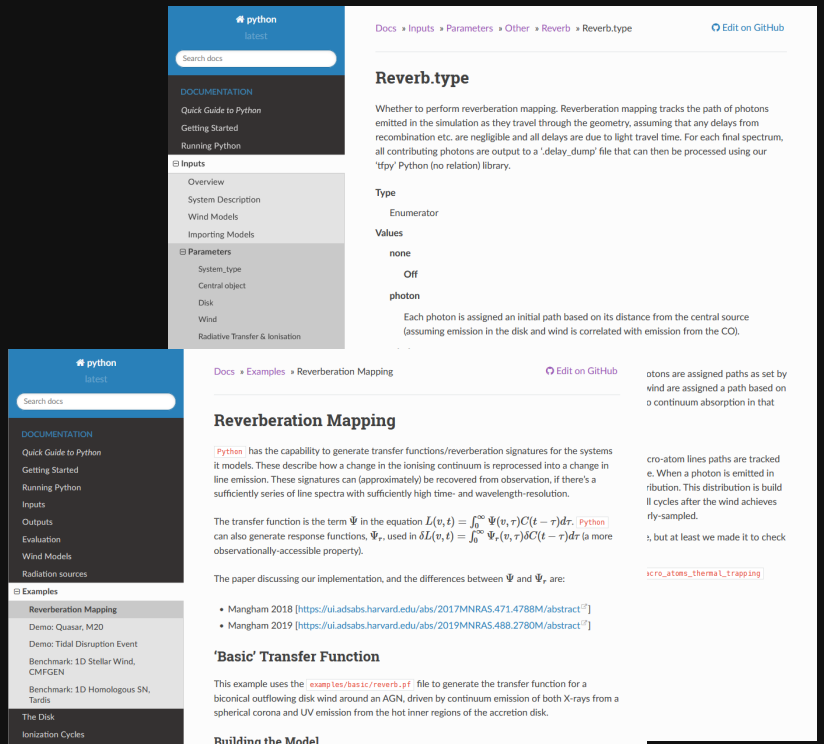
Managing Usage

Speaker notes

That generally covers how we can manage the development of our code to make it as easy as possible for us to keep our development process on track, producing code that's easy to work on, how do we ensure it's easy to work *with*? How do we best manage the ongoing usage of the code in a way that maximises the amount of actual results our code can generate?

Public Documentation

- Easy onboarding
- Quick reference for yourself
- Online documentation platforms
  - ReadTheDocs again
  - GitHub Pages
  - GitHub wikis



Speaker notes

The first and biggest point is to head right back to documentation. We've discussed developer documentation of how the components of the code work, but equally important is user documentation that describes how to actually install, run and analyse the outputs of the code. A lot of academic codes lack this - information on how to actually use them just lives within the brains of a research group or a small community, and getting a new user like a new PhD student involves personally walking them through how to set up and use it. For anyone outside the group hoping to download and use the code, and collaborate with you, they have a swathe of unknown unknowns - which physical regimes are the code's outputs meaningful in? How do you set the input parameters properly? Getting this information from the code itself requires knowing where to look in the first place.

If we want to maximise the use of our code, and our contributions to the field, we need to ensure it's as clear as possible how to use it. Fortunately, we've already introduced documentation platforms like ReadTheDocs. These can include not just autogenerated API documentation, but hand-written text like descriptions of input file parameter, images, equations typeset in LaTeX and even Jupyter notebooks containing examples of how the data is analysed. Whilst it might seem a large investment of time to write this originally, it then saves a huge amount of time when actually introducing people to the code, and on top of that involves documenting things about the code you're likely to need to include in papers anyway, so serves as a resource for later.

On the right there's couple of examples using Sphinx and ReadTheDocs to render a Jupyter notebook with a walkthrough of how to set up and run a particular type of analysis, and documentation on the parameters used, but you can also use GitHub pages, just open a wiki on your GitHub repository, or even set up an entire seperate website. Generally, though, the easier the process of writing documentation is and the closer your documentation is to your code, the better. The important thing is reducing friction and increasing usability.

Public Issues

- Facilitate problem solving
  - Searchable if possible!
- Own up to the code's limitations
  - Benefits far outweigh embarassment!
- Issues are a dialogue with your users
  - Even non-issues!
  - Structure it with issue templates

Speaker notes

Another major help for users is having a public record of issues with your code, ideally on whatever repository hosting service you're using. Fundamentally, there's only a limited number of things that can go wrong with a code - though it often feels unlimited. A public, searchable record allows users to solve their own problems by comparing them to previous ones. Sites like GitHub and GitLab enable this by allowing you to link together issues, making this process easer, and the easier it is for someone to get around problems the more likely they are to use your code and cite you. Crucially, though, they let you keep an accessible record of the things your code *cannot* do- the unfixed bugs it has, what it fails to accomplish. By keeping this open, you can save a huge amount of time spent by users trying to use your code for things that others have established are impossible. People can be a bit iffy about what they might think of as showing their code's dirty laundry, but the benefits far outweigh the cost. This is especially true if the issues are accessible from search sites, so the traditional "Paste the error message into google" programming technique works. You can't necessarily rely on the people using your software knowing where its help site is - it might become part of a larger pipeline.

One of the main advantages of issues is that they open up a *dialogue* with the users of your code. This is valuable even when there are very few users, or when the issues they raise aren't really genuine problems. Understanding how people are trying to use your code, and what they want it to do, gives you information on what would be useful for the field going forwards, and what about your code is unintuitive or inconvenient. Addressing these will generally improve your software. That said, it's a lot easier when the issues are communicated in a coherent way, and people have a bad habit of not describing their problem in detail, but proposing a change that they think will address the symptoms. Most repository hosting sites allow you to set up issue templates, that encourage users to report their problems in a structured way and attach sufficient information to help you debug them. Often, simply forcing people to go through this extra step results in them solving their *own* problems, by making them think more systematically about what they're trying to do.

Questions?

Speaker notes

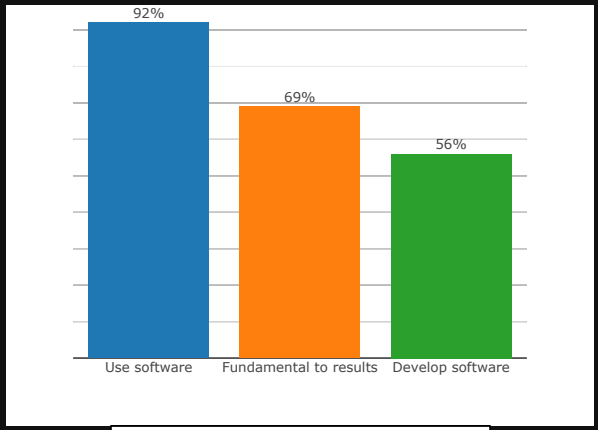
And again, though this section is pretty short, does anyone have any questions about code usage?

Managing Release

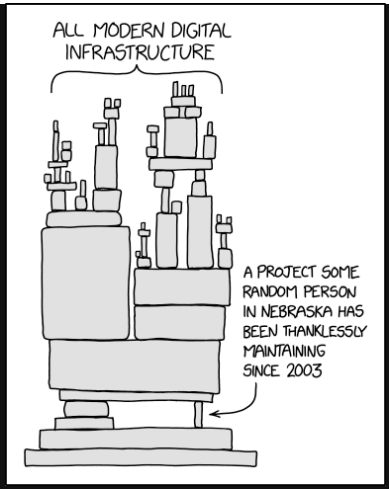
Speaker notes

Finally, one of the most important parts of developing a code is *releasing* it. Whilst it's common to keep the code itself internal and only share its outputs, many journals are moving to require the public release of code, and it's in both our and the field's interests to do so - so I'll introduce some ways to manage this process.

Release Your Software



- Majority of research relies on software
- Much is paperware
- Public release is required for reproducibility!



S.J.Hettrick et al, Software in Research Survey, 2014; DOI:10.5281/zenodo.1183562

Randall Munroe, XKCD - Dependency

Speaker notes

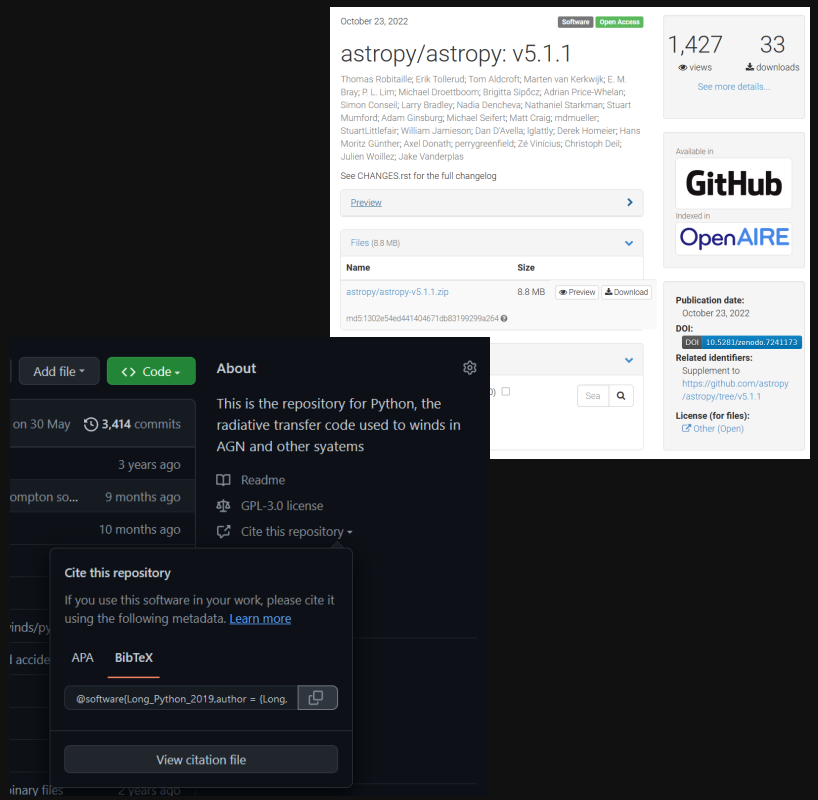
So by this point, the vast majority of research depends on software - surveys by the Software Sustainability Institute showed it's fundamental to at least two thirds of research, and obviously in this field it's 100%. However, the actual *acknowledgement* of software as a research product itself, separate from the papers produced by it, is often lacking. This lack of acknowledgement then leads to funding difficulties - code may underpin an entire field but go completely unfunded, relying on maintenance squeezed in around the developers' day jobs, as in the classic XKCD comic we've probably all seen before.

This has two common outcomes. The first is 'paperware', poorly-documented software riddled with technical debt that's just not maintainable. Rather than serving as a platform for future development, paperware is frequently thrown away and rewritten when a new PhD student comes along. This is a massive waste of time and effort, and a real hit to research outputs. The other outcome is postdocs who've spent years maintaining vital software finding themselves pushed out of academia by a lack of a publication record. The Research Software Engineering movement is improving this, by pushing for greater support for software development and providing roles for dedicated software developers, but it can only be successful if we publicly acknowledge the value of software.

If you release your software, and cite it in your papers, you acknowledge the effort that's gone into it. More than that, you make it clear to others where they can use your software, and how they can cite it. This is essential from the perspective of reproducibility, as the software is a fundamental component of your paper and required to reproduce your results. Further, though, releasing your code for others to use also drives collaboration, drives up citations for your work, and helps the field accomplish more. Some researchers shy away from publicly releasing their software as they're worried about being 'scooped' - but pragmatically, encouraging others to use your code means you already have a head start over them! There's a reason that tech companies obsess over creating standards and platforms they can then force others to use.

Structured Releases

- GitHub Releases
- Citation.CFF
- Zenodo
  - Provides citeable DOIs
- Include *all* info
  - Library versions
  - Compiler versions
  - Compiler flags



Speaker notes

So let's cover tools and practises for managing your release. Just citing your software by its name runs into some issues with reproducibility, though. As code evolves over time, the behaviour can change dramatically. Citing just by software name is equivalent to referencing a "Methods" section from another paper that's constantly being rewritten.

Fortunately, there's infrastructure in place to manage this problem too. Giving your code formal, versioned releases associated with particular papers and outputs allows you to specify particular versions of your code. In a feature-branch workflow like I mentioned in the section on managing development, these would typically be associated with a commit to the main branch, tagged with a particular version number that provides an easy reference. Sites like GitHub support this by listing releases with documentation, as well as letting you add a Citation.CFF file. This is a community-developed file format that provides instructions on how you'd like your software cited, including things like linking to your ORCID and requesting multiple citations, to for example a release paper and to a DOI for your specific version. Whilst your University library can usually give you a DOI for a piece of software, another really useful service is Zenodo, a DOI-generating site that links to GitHub, and can automatically provide you with DOIs for commits to specific branches. There's an annoying chicken-and-egg situation where you can't include the Zenodo DOI for a commit in the commit itself, though.

When creating a release, for HPC codes it's especially important to document all the requirements to run the code, including things like libraries and their versions, compilers and their versions. For heavily-optimised code it's not uncommon to end up bound to specific compiler versions, and if this isn't documented this is a massive pain for a user to figure out. The easier you make it for others to run your code from a release, the more likely they are to use it to write papers, produce scientific outputs, and cite you.

Software Licenses

- Previous HPC-BP talk
- No License
  - Automatically copyrighted
  - No rights for others to do *anything*
- Open-Source
  - Copyleft (e.g. GPL3)
  - Permissive (e.g. MIT)
- Proprietary License
  - Lawyers are expensive
- choosealicense

Speaker notes

Finally, when you're making a release you need to consider the license your code is under - this can have a major impact on how your project is used and developed in future. I'll only give a quick overview - the HPC Best Practises series has an entire previous talk on the topic!

Without a license, your code is just under whatever copyright laws your country has. In the US, UK and EU that means nobody else can use, copy, modify or distribute your code, even if it's publicly available. A lot of projects kind of get away like this, but it's a bit like playing chicken. Without a license, then usually it'll be your employer who owns the code you created, which can represent years of your time - and there's always the possibility that if you move institutions, or fall out with the wrong person, you find yourself permanently locked out of it. Pre-emptively selecting an open-source license can be your only protection against this. Having no license also pretty much prevents your code from being adopted by industry and having impact that way, as nobody's going to risk the legal fallout of integrating a complex and valuable piece of code into their business only to get a knock on the door from a hungry University demanding money.

The other end of the spectrum are open-source licences that give others the right to copy, modify and distribute your code, but without any assumption of liability. These come in two flavours - copyleft, that require that any future modifications to the code are also open-source, and permissive, that don't. Generally, copyleft also deters industrial collaborators, as modifying your code to work with their internal pipelines and data sources can then leak commercially-sensitive information.

Technically, depending on your contract you might not own copyright to your code and not be able to open source it without working with your University IP management team. However, at least in the UK, many haven't caught up with the realities of academic software development - when we asked our University team for advice on a project, the query got passed around for a month until someone who didn't realise we were the ones who made the query approached us to ask for our advice on it! So unless you're looking for a proprietary license, custom-written so you

can charge industrial users, it's easier to just go MIT or GPL without asking them.

However, realistically complex and burdensome proprietary licenses involving months of expensive legal fees also deter collaboration. If you want your project to have long-term support from industry, it can be better to just open-source it and encourage external contributors. I'd recommend using the website choosealicense.com for a straightforward and comprehensible guide to license selection.

Thank you for your time

⭐ s.mangham@soton.ac.uk / rsg.soton.ac.uk ⭐
s.mangham@society-rse.org / society-rse.org
s.mangham@software.ac.uk / software.ac.uk

Advertisements

bit.ly/SocRSE-Mentoring-2022

rsg.southampton.ac.uk/jobs

Any questions?

Speaker notes

So that's been my quick summary of how you can manage your academic software development in a way that will hopefully result in releasing more organised code that's easier to use and get credit for.

Thanks for your time and I hope there's been something useful in this talk! If you'd like to contact me, I wear a lot of hats but my Southampton email address s.mangham@soton.ac.uk is the best bet.

I'd also like to quickly plug a few things:
If you're on or near GMT time, and a member of the Society of Research Software Engineering, which anyone around the world can join, not just people in the UK, our mentoring scheme is open to applications - for the next week or so!
If you're in the UK, the Southampton Research Software Group is recruiting - we have posts open for 2 regular RSEs, 2 Senior RSEs and a Senior HPC specialist! It's a great group to work in, so please come join us.

So again, thanks for your time and I'm happy to answer any questions about code release, or other topics mentioned in the presentation!