Eur. Phys. J. Plus (2023) 138:1094
https://doi.org/10.1140/epjp/s13360-023-04732-5

Tutorial

Eat, sleep, code, repeat: tips for early-career researchers in computational science

Idil Ismail[1,2,4,a], **Shayantan Chaudhuri**[2,3,6], **Dylan Morgan**[1,2,4], **Christopher D. Woodgate**[1,4], **Ziad Fakhoury**[1,2,4], **James M. Targett**[1,4], **Charlie Pilgrim**[5], **Carlo Maino**[1,4]

[1] EPSRC Centre for Doctoral Training in Modelling of Heterogeneous Systems, University of Warwick, CV4 7AL Coventry, UK
[2] Department of Chemistry, University of Warwick, CV4 7AL Coventry, UK
[3] EPSRC Centre for Doctoral Training in Diamond Science and Technology, University of Warwick, CV4 7AL Coventry, UK
[4] Department of Physics, University of Warwick, CV4 7AL Coventry, UK
[5] EPSRC Centre for Doctoral Training in Mathematics for Real-World Systems, University of Warwick, CV4 7AL Coventry, UK
[6] School of Chemistry, University of Nottingham, NG7 2RD Nottingham, UK

Received: 15 November 2023 / Accepted: 22 November 2023

Abstract This article is intended as a guide for new graduate students entering the field of computational science. With the increasing influx of students with diverse backgrounds joining the ever-popular field, the aim of this short guide is to help students navigate through the various computational techniques that they are likely to encounter during their studies. Here, we cover a broad spectrum of techniques, including Bash scripting, scientific programming, and machine learning, among other fields. This paper is structured into nine sections, each introducing a different computational method. To enhance readability, we have adopted a casual and instructive tone throughout and included relevant code snippets. Please note that due to the introductory nature of this article, it is not intended to be exhaustive; instead, we direct readers to a list of references to expand their knowledge of the techniques discussed within the paper. Finally, readers should note this article serves as an extension to our student-led seminar series, with additional resources and videos available at https://computationaltoolkit.github.io/ for reference.

1 Introduction

Computational science is a highly interdisciplinary field encompassing a wide array of subject areas, including physics, chemistry, engineering, mathematics, life sciences, and more. The objective of this article is to highlight several uniting themes, skills, and methods used by computational scientists within their respective fields. It serves primarily as an introductory guide for early-career researchers, representing a collaborative effort where each section is authored by a current graduate student, bringing forth their unique insights. Our goal is to both facilitate the learning process and assist researchers in effectively applying the discussed tools and methodologies. Finally, it is important that readers treat this guide as informed advice, drawn from the varied experiences and expertise of its contributors, rather than a definitive instructive manual.

2 Navigating the zoo of computational techniques

Computational science, regardless of the precise discipline, means running scientific software on a computer to obtain meaningful results. But what do we mean by 'software'? Simply put, scientific software is any program which takes in data and parameters, does something meaningful with it, and returns some new data.

Rather than author a piece in which I try to cover the whole of computational science in one go, I will focus on my specific sub-discipline of computational physics, electronic structure, and hope to draw out some lessons which might be applicable in other areas, too!

2.1 Where to start

On your research journey, you will inevitably have a number of academic mentors, be they your MSc/PhD project supervisor, a more senior student/researcher in your group, or an external collaborator. The intention is that these people are there to *help* you on your research journey, so don't be afraid to ask for help when you need it. Far too often, a student will struggle for days with a problem when another member of their research group would have known exactly what to do, having encountered that same problem before

[a] e-mail: Idil.Ismail@warwick.ac.uk (corresponding author)

themselves! So, while it is nice to be able to work independently, it is also important to recognize that your time is valuable and that it is worth asking for help with something that you are struggling with if you feel stuck.

With my PhD supervisor, I arrange a weekly one-on-one meeting in which I discuss what progress I have made and where I should go next, but I am also able to email her or drop by her office if I have a question between meetings. I like to aim to produce a meaningful graph or piece of data for each meeting. Over the course of four years of PhD study (allowing 8 weeks of the year off for holidays or illness) one graph per week works out at over 180 graphs in total, which hopefully makes thesis-writing a little less scary at the end! As a caveat, I will add that, while this arrangement has worked well for me, different research groups work differently, so you should discuss your own schedule with your supervisor.

When you start on a research project, your supervisor or academic mentor will almost certainly give you something from which to start, be it some papers to read, or maybe some test calculations/simulations to run. The idea here is for you to dip your toe into the water working on a problem for which we already know the answer, to make sure you understand what you're aiming for and what pitfalls there might be along the way. It might be that your research project is methods-based, meaning you are going to develop new algorithms, codes, or other tools to solve a problem that existing techniques can't solve or, at least, can't solve efficiently. Your project might have more of a results/ applications focus, meaning you will be applying existing techniques to new problems to see what can be learnt. Or it might be some combination of the two. Regardless, there will be some 'toy' problems with which you can get started, and some literature to review, before you can get your teeth stuck into a problem in earnest. Be sure to take these toy examples seriously and keep any code you write/use stored somewhere safely so you can refer back to it later; it will almost certainly come in handy at some point!

At this early stage, it is unlikely that you have had the ability to choose which codes or computational techniques you use, because you will have been working through some well-understood example problems. But this is reasonable; we always have to understand the state of the art before we aim to go further. For example, I work primarily in the areas of alloy theory and magnetism and, in particular, I study the nature of atomic arrangements in these systems and how those atomic arrangements can influence a system's magnetic properties. Some example problems which I would give someone starting in my area would be to compare the energy per atom of an ordered, intermetallic structure with a disordered structure. (A good example of such a system would be the Cu–Au alloy.) There are many suitable codes and techniques, but one could start with comparing the energy difference per atom obtained for a large supercell using a plane-wave density functional theory (DFT) code like CASTEP [1] with the same energy difference obtained using a code using an effective medium to represent the disorder, such as the coherent potential approximation as implemented in JuKKR [2]. I might also encourage someone new to the field to study the effect the magnetic state can have on some systems. For example, for an alloy containing Fe, care should be taken to treat the magnetic state (non-magnetic, paramagnetic, ferromagnetic) appropriately for the system being studied. A good example might be to compare the energy per atom for bcc Fe in each of those three states, and to verify that the ferromagnetic state is the one of lowest energy.

2.2 Where to go

After you have finished your literature review and had a go with some example problems, it will be time to begin working on genuine research problems. Inevitably, this will mean developing your own scripts/code along with using those written by other people. At this stage, it is wise to follow the advice of your supervisor; they have a lot more experience than you do and will have a good idea of how they would make progress on your research problem.

Follow their advice, and keep an eye on what is currently being done in the literature. Often, inspiration can be drawn from reading articles published by others working in your field. Perhaps they have developed a new technique which you think can be applied to your system, or perhaps they are using a different code/technique to you but on the same system. For example, in my own work, we recently performed a study for which one specific aim was to compare the results obtained using a variety of other techniques with that of our own [3]. Such comparisons are both necessary and informative; they tell the community what aspect of a problem one code/technique gets 'right,' and what aspects it gets 'wrong.'

2.3 Going it alone

Once you have a feeling for what you are doing, and the current state of the literature for your research area, it is not unwise to begin asking yourself where your research might go next. By the end of your PhD, you may well be more knowledgeable than your supervisor in the particular research sub-area in which you are working!

With this in mind, starting from your current area of research, ask yourself what the outstanding problems are, and then ask yourself how you might go about tackling them. Interdisciplinary and collaboration is key to solving a large number of scientific problems, so make sure to talk to those outside of your area, too, as they may have encountered the same problem in a different setting. Pursue interesting ideas and make progress where you can; very often seemingly small progressions can impact a field once their implications are understood.

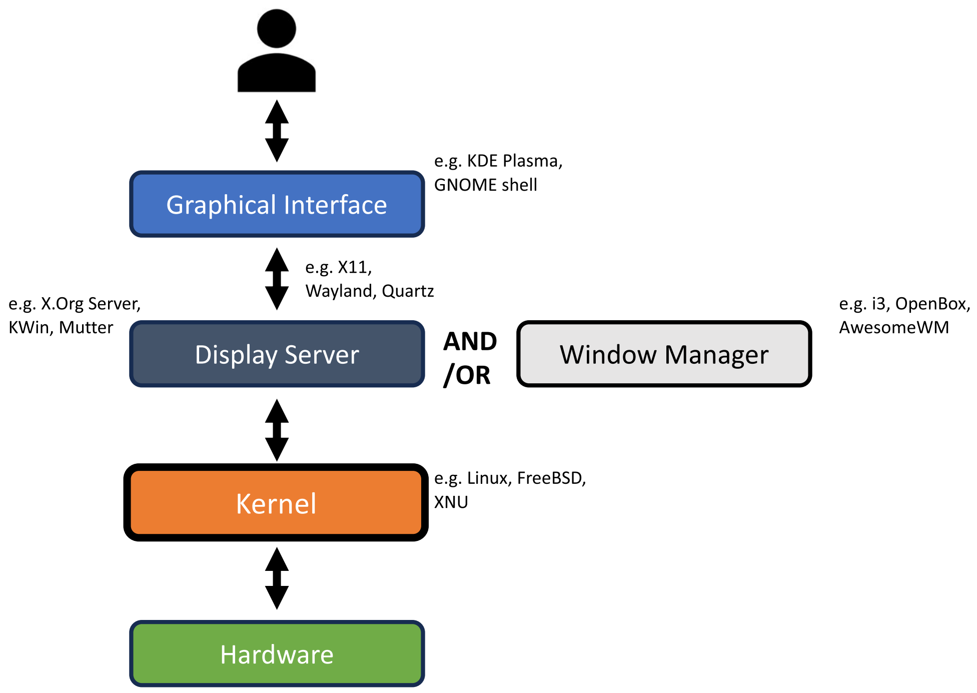
e.g. KDE Plasma, GNOME shell
Graphical Interface
e.g. X11, Wayland, Quartz
e.g. X.Org Server, KWin, Mutter
Display Server
AND /OR
Window Manager
e.g. i3, OpenBox, AwesomeWM
Kernel
e.g. Linux, FreeBSD, XNU
Hardware

Fig. 1 Diagram of the different layers of an operating system and how they interact with each other. Some commonly used examples operating systems can use for each layer are also shown

3 Introduction to Bash scripting

It is often necessary in computational science to automate mundane and repetitive tasks. At a somewhat low level, Bash scripting can be used to achieve this. However, briefly describing an operating system's architecture first will be helpful for understanding this. Beginning at the highest level, the user typically interacts with an application in the graphical interface. The graphical interface is supported by a display server, a window manager, or both, and it interfaces with the display server, for example, on Linux, through X11, or Wayland. Figure 1 shows a diagram of this. The display server is a front end to the kernel, and the kernel is essentially the operating system's source code and what fundamentally differentiates different operating systems from each other. Some well-known examples of kernels include the Linux kernel, FreeBSD, or XNU (for MacOS).

The command line, accessed through a Terminal emulator (although often incorrectly referred to as just a 'Terminal'), provides a direct interface between the user and the kernel and bypasses the graphical interface and display server. A group of languages can be used by a user in a terminal called Shell scripting languages, which use a particular syntax and have scripting capabilities.

3.1 What is Bash and why is it useful?

Using a Terminal to control a computer is often the fastest way of accomplishing tasks and, in many cases, especially those that are more complex, the only method. The Bourne Again SHell (Bash) is a Shell scripting language. Shell languages are a group of languages which have a specific syntax to allow the user to use the command line and write scripts. Bash, released in 1989, is arguably the most popular of these and has seen widespread usage across many operating systems, especially in Unix-like OSs. Bash was released to replace the Bourne Shell, which in turn was an improvement to the original sh language. Bash is still the most commonly used Shell today and is almost always the default on Unix-like operating systems (except MacOS as of 2019) and high-performance computing (HPC) systems. HPC clusters do not have a graphical interface, so everything must be done via a Terminal.

3.2 Useful features of Bash

For many individuals beginning a PhD or career in computational science, one of their first tasks is to learn how to navigate their file system. While the goal of this section is not to provide a full tutorial on using Bash (and for that, the reader should refer to Ryan's Tutorials [4, 5] on Linux and Bash for an excellent introduction), there are several powerful and valuable features I wish to highlight to the reader, as well as use cases and drawbacks in my personal opinion.

The first is command redirection, with a particular emphasis on piping. Generally, command redirection refers to altering the input and output of commands from the command line. In its most basic form, redirection can be used to write the output of a command to a file or to limit the output to only show error messages, for example. Piping is a slightly more advanced and potent tool for redirection, as it allows the output of one command to be directly used as the input for another using the | symbol. It is commonly employed with another powerful tool, `grep`, a command for searching for text in a larger body of text, like so: `ls -a | grep foo`. This command searches for all files in a directory with the string `foo` in its name. A more complicated example would be:

```
ls -a | grep foo | xargs cat | awk '{print $4}' | sed 's/pass/fail/g'
> foobar.txt
```

This finds all instances of files in a directory with `foo` in their name, takes the fourth word in each line, replaces that word with the word 'fail' if the fourth word is 'pass,' and outputs this to a new file called `foobar.txt`.

Command substitution is similar to redirection, and it involves substituting one bash command in another line of Bash, and the output of the substituted command is used. The following is an example of the use and syntax of this, which sets the variable `fftw_dir` to the location of a file in the `/usr/include/` directory that contains 'fftw.' The specific syntax for the command substitution is to place the command in `$()`; however, it is still common to see the now deprecated backtick convention using ``.

```
fftw_dir=$(find /usr/include/ -name "*fftw*")
```

At some point, a user will likely come across a case where they need to edit the content of a file. However, how is this possible when there is no graphical user interface (GUI) available? It would be possible to jointly link several commands directly from the command line; however, this is incredibly unwieldy and arduous. Instead, several applications are often installed on a system that enables editing and viewing files directly in a terminal. The most popular of these today include Nano, Vim, and Emacs. They use different keybindings to navigate through text in files, enabling navigation of the file without a mouse.

File permission and ownership is another area that often confuses relatively new users of the command line. It may be the case where a user will come across a file they need to read or write and will not have access to it, and Bash provides a particularly non-verbose message of the issue. Listing the directory's contents with `ls -l` will also output file/directory ownership and permissions information. The second and third columns of the output of `ls -l` describe its ownership, where the second column is the owner user, and the third column is the group. Ownership can be changed by `chown user_a:user_b path/to/file/or/directory` where `user_a:user_b` are the user and group owners, respectively. Permissions are given in the first column of the output of `ls -l`, where there are ten characters for each result. The first is either a '-' or 'd,' which outlines whether the result is a file or directory, respectively. The next nine characters are organized as three sets of three, each of which refers to the user, the owner, and all users. In this order, each set's character is one of '`r`,' '`w`,' or '`x`.' These letters indicate whether permissions are set to read, write, or execute the file or directory. If one of the letters has '-' in its place, then the file or directory does not have permission to perform this action. As these are given for the owner, group or all users, it describes which user has which permissions for the file. Ownership can be changed with `chmod ug+w path/to/file/or/directory`, giving the user and group write permissions for the file or directory. An example output using `ls -l` is shown below.

```
total 8
drwxr-xr-x@ 2 dylanmorgan  staff  64 20 Oct 11:15 bar
-rw-r--r--@ 1 dylanmorgan  staff   0 20 Oct 11:14 foo
lrwxr-xr-x@ 1 dylanmorgan  staff  27 20 Oct 11:16 foobar -> ../barfoo
-rwxr-xr-x@ 1 dylanmorgan  staff  53 20 Oct 11:15 tmp.sh
```

Finally, documentation is available for every Bash command accessible through the command line. These are called man (manual) pages and are accessed by typing `man`, followed by the command for which the user wishes to check the documentation. When a man page is open, the default vim keybindings (i.e., `h`, `j`, `k`, and `l`) are used to navigate the page, and '`q`' to exit and return to the command line.

3.3 Writing Bash scripts

Bash is Turing complete and has features also seen in many other languages, like `if` statements, `for` loops, functions, and argument parsing. Beyond just using Bash in a Terminal, Bash is also commonly used to write scripts. Scripts written in Bash have the same functionality and syntax as when using Bash in the Terminal. However, they are helpful when the user wishes to execute many bash commands in succession, schedule tasks, wrappers for other command-line applications, or even for writing command-line programs. The following snippet is an example of a short function taken from a bash script to clean binary files from a compilation of unit tests for a program.

```
unit_test_clean () {
    ### Remove compiled binaries from unit testing ###
    bins=("test/test_bin/*.mod" "test/test_bin/*.o" "test/test_bin/*test_spindec")

    # Check for binaries
    if [[ $(ls test/test_bin/* | grep -E 'mod|[.]o|test_spindec') == '' ]]; then
        echo "No binaries found"
        exit 1
    fi

    # Remove globs from bins if they aren't found
    for glob in ${bins[@]}; do
        echo "$glob"
        if [[ "$glob" == *'*'* ]]; then
            bins=("${bins[@]/$glob}")

          fi
      done

      echo -e "Removing the following files:\n"
      ls test/test_bin/* | grep -E 'mod|[.]o|test_spindec'
      echo

      for file in ${bins[@]}; do
          rm "$file"
      done
  }
```

Bash's widespread use over such a significant period - or at least a long time on a computer time scale - and use in systems administration is helpful for those starting in Bash. Answers to most problems can be found with a quick internet search, and sites such as StackOverflow (https://stackoverflow.com/) are beneficial for help on specific issues.

Bash is not without its faults, however, and there are many problems that other languages, such as Python or Julia, would be more applicable for. Floating-point variables are not available in standard Bash, so any calculations involving floating-point arithmetic should be avoided, even though they are technically possible to do by piping a string to the `bc` command. It is my opinion that Bash should not be used for any math operations whatsoever. Physically typing Bash scripts can also become a nuisance with the many special characters that must be included. There are also confusing syntax quirks and inconsistencies; for example, sometimes, single brackets should be used over double brackets in `if` statements. In summary, Bash's strengths lie in argument parsing, easy manipulation of files and directories, and automation of command-line-based commands.

4 Introduction to high-performance computing

HPC [6] involves utilizing many computing cores in parallel to augment computational performance to tackle complex, large-scale problems that would otherwise be very inefficient and unfeasible. Depending on your work, there is a good chance that, at some point, you will need to use HPC facilities. Such facilities will allow you to run calculations and processes in tandem rather than sequentially. HPC clusters can range in size from local services provided by universities to national and international supercomputers and typically comprise a login node that users can `ssh` into. This can then be used to submit jobs to a scheduler, which are then executed by compute nodes. Jobs can be run using multiple compute nodes, and each node will typically comprise multiple cores, with some facilities such as the ARCHER2 UK National Supercomputing Service (https://www.archer2.ac.uk/) providing 128 cores per node.

Here are the things, in my opinion, that you should do whenever you get access to a new HPC facility:

- **Set up basic commands:** This includes properly setting up your `$HOME/.bashrc` file (or equivalent), which can include useful `alias` commands. If you like having your own Conda environment,[1] also set this up. It is also good practice to add some `module load` commands to your `$HOME/.bashrc` file for basic modules that might be needed frequently, so these are automatically loaded whenever you are on the login node. This can include programming languages like Python, R, or CMake, often required for software compilation.
- **Familiarize yourself with the HPC:** Most HPC facilities have good online documentation, making it easy to look up anything you need. It is good to familiarize yourself with job resource limits, such as the maximum number of nodes per job/user and the maximum time limit (wall time) per job. It is also important to note that you won't always need the full wall time for every

[1] Conda is a widely used package-management environment that allows users to install specific software packages and dependencies. It facilitates the replication of software environments by creating isolated and self-contained spaces, preventing conflicts between distinct projects.

job. Setting the maximum wall time for every job will only increase the queue time for your job before it is executed! It is also worth looking up the different node types that the HPC cluster provides. Many clusters provide different node types, such as high-memory nodes for memory-intensive jobs and developmental nodes for testing, saving you from queuing for the standard compute nodes. Some facilities also include graphics processing unit (GPU) nodes, recommended for machine learning (ML)/ deep learning applications and accelerating software packages [7–12].

- **Software compilation:** When using software to run jobs on a HPC cluster, you must have a parallel binary or library of the software compiled. This will require you to be familiar with the compiler types available on the HPC cluster, e.g., Intel, GNU Compiler Collection (GCC), Cray, and these can typically be loaded as modules. But before compiling anything, always ask around! Do not waste time reinventing the wheel and struggling to compile. Compilations are cluster-specific, so asking your group members for instructions is always a good idea, as they may have previously compiled the software. Most HPC services have a help desk for compilation and other issues or questions, so don't be afraid to contact them.
- **Submitting jobs:** The most common job scheduler for HPC facilities is Slurm, though some use the Portable Batch System. Ensure you acquire an example job submission script, either from the HPC-specific documentation or from someone who has used the cluster, and submit a test job to ensure everything works. All commands within a job script must be written using the Bash language (or some other Shell variant), so make sure you are aware of simple scheduler commands such as `srun`/`qsub`. A good summary of useful Slurm commands can be found at https://slurm.schedmd.com/pdfs/summary.pdf, though reading through any cluster-specific documentation is still advisable.

5 Introduction to machine learning for computational science

In this short section, we aim to introduce the audience to some of the key software and algorithms that, in our opinion, are essential for anyone getting started in ML. Given the general scope of this article, we will aim to keep this section brief. Therefore, this will not be an exhaustive list but instead, we will provide references for further reading. The tips and suggestions offered in this section will be rather generic and only really intended for an uninitiated audience. They may not be of much benefit to more seasoned ML practitioners.

5.1 What is machine learning?

With the ever-increasing availability of larger datasets and computing power, a growing number of researchers are embracing ML as an integral part of their computational toolkit, a trend that is further demonstrated by recent reports of over 70,000 ML-related papers being published in 2022 alone (Fig. 2). This is not least because of its success in offering significant improvements in speed over more computationally demanding first-principles calculations, helping to further streamline and accelerate the discovery pipeline.

At this stage, it may be beneficial to explain what ML is and to give a short overview of its main principles. Simply put, ML is a sub-field of artificial intelligence concerned with modeling data using algorithms capable of learning and extracting meaningful patterns and relationships between sets of input variables. In each case, the computer learns from previous examples and optimizes its performance on a given task accordingly. A key feature of ML is its ability to make predictions on new data without being explicitly programmed. Broadly speaking, ML can be further subdivided into three main categories; supervised learning, unsupervised learning, and reinforcement learning [13]. Of the three, supervised learning models are the most popular and relevant type of ML in the natural sciences, and will therefore be the focus of the next few sections.

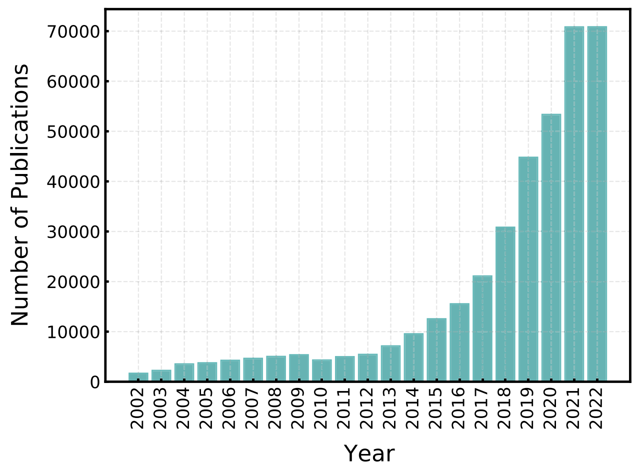

Fig. 2 Number of papers published in the field of machine learning (ML) over the past two decades (Web of Science (2023). Retrieved from https://www.webofscience.com)

A key distinguishing feature of supervised ML is that the model is trained on a labeled dataset, where there is a direct and explicit mapping of the input (predictor) and output (response) variables [13]. The goal here is for the model to learn a particular relationship that exists between those two variables. These ML models can be used in either regression or classification tasks, where the former requires continuous data as input and the latter requires a model to be trained on categorical labeled data. Examples of supervised learning models include linear regression, decision trees, and artificial neural networks (ANNs). As the name suggests, linear regression algorithms are used to model linear relationships between variables. However, in cases where the data is nonlinear, one might opt for alternative ML models, like decision trees or ANNs, which are capable of modeling complex nonlinear relationships between variables.

5.2 How to get started

There is a general and accepted approach to training and deploying a ML model, which can be summarized in the following steps (also exemplified in this short Jupyter notebook tutorial).

Data Acquisition. In the first instance, you will need a large enough dataset generated using a consistent methodology, whether that be experimental or simulated. Note that the reliability of your ML model will largely depend on the quality of your data, so it is important to evaluate and perhaps even perform some basic exploratory analysis on your dataset before you get started. For more tips and suggestions on how to select a suitable dataset, see [14].

Feature Selection. When developing any predictive model, it is important to consider the functional relationship between the input feature and the target value you are trying to predict. The procedure by which this is modeled is known as feature selection. There are generally two key points to keep in mind when deciding on which features you want to encode into you ML model.

- Though this may sound obvious, your chosen feature must be correlated to the target property and should be relevant to the desired prediction.
- Beware of choosing redundant features. Often, it is tempting to pack your model with as many features as possible, but you should restrict your chosen features to only those that improve the predictive capabilities of the model. Features that do not contribute to the accuracy of the model, often do nothing more, other than increase the overall computational cost, but could also contribute to worsening the overall predictive performance by learning redundant information.

Featurization. Before we can progress with training a ML model, we must first consider how we will represent and ultimately transform our data into a format that a computer can interpret, e.g., a vector representation. Effectively, we need a unique and non-trivial descriptor that can capture all the relevant features of the input data. The requirements for your descriptor will depend on the nature and complexity of your data type; however, generally, for molecular descriptors in chemistry and physics, one might consider the following key requirements:

1. **Unique**: such that structurally different molecules are mapped to distinct vector representations. ML models cannot distinguish between identical features ascribed to different molecules, and will likely give poor or inconsistent predictions in such cases.
2. **General**: where the descriptor should be applicable to all atomistic systems. The representations should remain faithful irrespective of whether they are used to encode isotopes, charged, or neutral molecules.
3. **Efficient**: with respect to the computational cost associated with training a ML model using geometric descriptors. Given that ML is primarily used to speed up traditional numerical calculations, it is imperative that the ML predictor be orders of magnitude faster than traditional simulations.

Moreover, additional descriptor requirements may involve properties such as rotational, translational, and permutational invariance, to ensure faithfulness of the ascribed feature.

Model Selection. The next step involves selecting a suitable ML algorithm that can map given input features onto each other. The model you end up choosing will depend on a number of factors, including but not limited to:

1. Whether your task requires a clustering, classification, or a regression algorithm.
2. The complexity of your data: Linear regression models will most likely be unsuitable to modeling high-dimensional and complex data, which will instead benefit more from ML models like neural networks.
3. Whether your data is labeled or unlabeled will determine whether you will require a supervised learning or unsupervised learning model.
4. The size of your dataset: Certain models are better suited for large datasets, such as support vector machines (SVM), random forests (RF), and deep learning models like convolutional neural networks (CNN). In contrast, Gaussian process regression is often restricted to smaller datasets due to its heavy computational costs [15, 16].

Data Splitting. To ensure that you've built a reliable, and crucially, testable model, you will want to set aside a portion of your dataset for testing, typically somewhere around 20%. The data points selected for testing should be equally distributed and preferably randomly selected. There are a number of techniques that can be used to achieve this, e.g., K-fold cross-validation [17]. Finally, you should be able to train your ML model using the remaining 80% of your dataset.

Hyperparameter Optimization. You can enhance the performance of your ML model through a process known as hyperparameter optimization. Hyperparameters are simply a set of external configurational settings which are pre-determined, and not learned from the data, influencing how a ML model learns. These settings dictate aspects such as the model's learning speed (e.g., the learning rate in gradient descent), the number of hidden layers of a neural network, or even the depth of search in a decision tree. Tuning or optimizing hyperparameters is a vital step, playing a major role in determining the performance of a trained ML model. Fortunately, there are a myriad of existing software libraries that contain ready-made functions you can use to accomplish this. Examples include brute-force algorithms such as grid-search methods (GridSearchCV) offered in Scikit-learn [18] or alternatively, Bayesian optimization methods.

Model Training & Evaluation. Once your model parameters and hyperparameters are defined, you can start training your model. To determine the success and accuracy of your model, you can then test it on the previously saved dataset and evaluate the accuracy of the predictions made using a loss function (a measure of how well a model's predictions ($\hat{y}_i$) match the ground truth (y_i) for a given number of data points N). Popular ones include the mean absolute error (MAE) and root mean squared error (RMSE), as shown in equations 1 and 2.

$$\text{MAE} = \sum_{i=1}^{N} |y_i - \hat{y}_i| \tag{1}$$

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{N}} \tag{2}$$

For regression tasks, you may also take note of the coefficient of determination (R^2), which gives a measure of 'best-fit.'

Model Deployment. Finally, once you're confident and happy with your testing errors, you can deploy your ML model on new and unseen data. This is the ultimate test of the accuracy of your newly trained ML model. Of course, it would help to benchmark your results against experimentally/theoretically verified findings for further confirmation.

5.2.1 Essential preparations before training your first machine learning model

Now that you've acquainted yourself with the general workflow of a typical ML task, you might be wondering how these steps can be applied in practice. However, as with learning any new skill, it is essential to have certain prerequisite knowledge. This is important not only to develop your ML model, but also to optimize and interpret its results. Listed below are three of what we deem are the most important prerequisites you should have before venturing to train your first ML model.

- **Mathematics:** First, it is most helpful to have a solid background in at least the basics of linear algebra and calculus. This will enable you to understand the algorithms you are working with, evaluate your dataset, and of course, test your model.
- **Domain Knowledge:** Similarly, having prior knowledge of the domain on which you intend to apply your ML model is arguably just as important as understanding the workings of your model, as this could highlight whether or not ML may even be suitable for the problem you are tackling.
- **Programming:** Finally, you must have a good grasp of at least one suitable programming language, and preferably Python. Other semi-popular programming languages for ML include Julia, R, and MATLAB. However, a general familiarity with data structures and algorithms would also be helpful.

5.2.2 Exploring popular machine learning packages

Given the current excitement and widespread interest in ML, we are fortunate to have access to a variety of open-source software packages that can be used to train and deploy a ML model. Due to the popularity of Python in the ML community; which is due in large part to the language's ease and flexibility, many of these software packages are also Python-based. Examples include Scikit-learn [18], TensorFlow [19], and PyTorch [20] (*though the core implementation of the last two is technically written in C++*). All three of these packages are well documented, each with their own set of strengths and weaknesses. The package you choose to start you ML journey with will largely depend on the particular requirements of your project. The latter two examples, TensorFlow and PyTorch, are more suited to deep learning tasks, while Scikit-learn offers many standard out-of-the-box supervised and unsupervised ML algorithms, while also complementing other popular libraries including Matplotlib [21], NumPy [22], and SciPy [23].

5.3 Should I *really* be using machine learning for this problem?

With all the excitement and hype surrounding ML, it's easy to get caught up in the buzz. However, as computational scientists, we must tread carefully and first consider whether our research problem is worthy of employing ML techniques at all; as the overall accuracy and generalizability of your results rest on this decision. To help you assess the suitability of ML for your research question, here's a checklist of statements to consider before jumping into the use (and abuse) of ML.

1. **Do I have enough quality data to train my ML model?** ML models are typically data-hungry. Therefore, the first point of inquiry should be whether there is sufficient and relevant data available to train your ML model. Here are some additional points to consider regarding your data requirements:

- Do I need to curate my own dataset, and is it within the computational budget to be considered practical?
- Is the data consistent and of high enough quality to ensure reliability?
- Is the data already in a format ready to use, or will I need to spend additional time pre-processing the data i.e., cleaning and feature engineering? Again, is it expense practical?

2. **Will using ML significantly speed up your calculations?** This question requires one to first assess the general speed of their calculations using traditional (*possibly first-principles*) simulations. ML is not a silver bullet, and the process of training and deploying an ML model can take some time itself. Therefore, one must consider factors that may help them decide and evaluate the potential of using ML to accelerate their calculations, and these include:

   - Is the time-complexity of your problem high enough to benefit from the computational speed offered by ML?
   - Does your research question require the identification of patterns and trends in large datasets?
   - Does your research question generate clear and measurable outputs?

3. **Is my research problem too complex to be modeled using ML?** To answer this question, first, you need to consider the possible reasons why you believe that traditional numerical simulations or statistical methods cannot adequately address your research aims. These may include questions such as:

   - How much domain knowledge do you require to train your ML model? To answer this, you will need to determine whether your investigation requires a direct or inverse approach. The latter likely requires more domain knowledge than the former where the scientific or physical principles are typically already known. An inverse problem research approach involves determining causal factors on the basis of observed simulation output, and therefore inherently requires expert knowledge to explain the physics or chemistry behind the observations, thus making it less suitable for ML investigation.
   - How complex or intricate are the features in your study? If your problem involves a high-dimensional feature space that is too complex to model via other methods, then it's likely that ML may not perform well either, as it's known to perform better with fewer and more relevant features. Again, ML is not a cure-all solution.

Finally, really ask yourself if you have trialed and exhausted all other possible statistical and simulation methodologies before turning to ML-based approaches, as these will likely offer far better and more reliable solutions to your problem.

6 Popular tools for scientific programming in Python

6.1 Do not reinvent the wheel

The computational sciences can be applied to a wide range of disciplines, where many of these fields share common tools and techniques, for example, visualization software, as we all need to plot graphs! ML, as mentioned earlier, is another versatile tool applied across varying disciplines and problems.

In most cases, existing software solutions can be found for your specific problem case, so 'don't reinvent the wheel' (*if you don't have to*). In fact, it can be quite detrimental at times to rewrite tools that have already been written from an efficiency standpoint. Most of these tools have had teams of engineers working on them, resulting in highly optimized libraries, surpassing any implementation that you or I could quickly develop. It is therefore quite vital that before you embark on any code sprints, to check what tools you have at your disposal, and how easy or straightforward it would be to incorporate them in your project.

6.2 Why Python?

Python is a high-level, general-purpose, programming interpretative language. This means it is more understandable and flexible. It boasts readability and interoperability with low-level compiled languages (mainly C/C++). It's highly popular among different software engineering areas outside of computational science, such as web development, game development, and data analysis.

The popularity of Python within the scientific computing community mainly stems from the fact there is an abundance of easy-to-use, well-documented, and optimized software packages written exclusively in Python. The availability of these packages, alongside interoperability and quick development, makes Python an easy choice for most computational scientists. Therefore, this section will primarily focus on suggesting Python packages that prove to be useful for computational science tasks. Other high-level options do exist and are used by many research groups, including, Julia and R. However, Python remains the clear 'go-to' language in this field.

6.3 Popular general scientific Python packages

6.3.1 NumPy/SciPy

One of the trade-offs for Python being such a high-level interpretative language, is its slower line-by-line interpretation. This is especially problematic when it comes to dealing with large matrices or multi-dimensional arrays which may require nested loops.

However, there are Python libraries designed to efficiently address these issues, namely, Numerical Python, or NumPy for short [22]. At the core of the NumPy package lies the array object, which efficiently stores and manipulates these multi-dimensional arrays. It provides a wide range of mathematical functions and operators optimized for numerical operations on arrays, enabling fast and memory-efficient computation through the use of compiled C/Fortran code. Its intuitive syntax and extensive documentation make it easy to use, even for those new to scientific computing. In addition, SciPy [23], or the Scientific Python Library, builds upon NumPy and provides a vast array of advanced algorithms and functions for scientific computing tasks. It covers areas such as optimization, interpolation, signal and image processing, statistics, and more. SciPy's comprehensive collection of sub-modules includes `scipy.optimize` for optimization problems, `scipy.interpolate` for interpolation and smoothing techniques, `scipy.signal` for signal processing, and `scipy.stats` for statistical analysis. These functionalities make SciPy a versatile and indispensable tool for solving complex scientific and mathematical problems efficiently. It cannot be overstated how crucial these two libraries are to the field of computational science. The ease of incorporating these advanced and efficient algorithms into a workflow allows you to focus on the bigger picture and avoid spending time dealing with nasty bugs on a sub-par implementation of these tools. These two libraries are a must have for any computational scientist.

6.3.2 Matplotlib and Seaborn

Matplotlib [21] is an essential tool for computational scientists as it provides a powerful and versatile framework for creating high-quality plots and figures. Its flexibility, integration with other scientific libraries, and extensive customization options makes it invaluable for tasks such as data exploration, analysis, and communication in computational science research.

While Matplotlib is considered quite a low-level visualization package in Python, it is highly versatile and allows users to easily customize their plots, it often requires quite a bit of effort to make presentable or publication-quality figures, even for standard plots.

Similarly, Seaborn [24] is a package built on top of Matplotlib that enables quick high quality figure generation for several standard plots. Also, since it's built on top of Matplotlib, many of the customizations you can make through Matplotlib are accessible with Seaborn-based plots too.

6.3.3 Pandas

Pandas [25] is a Python package that I like to describe as the Python extension of Microsoft Excel, except it is much quicker, more versatile, and more functional than Excel (and in my opinion, easier to use!). It's a crucial library for data manipulation and analysis. It provides efficient data structures and analysis tools that are essential for processing and exploring large datasets; something you will undoubtedly encounter during your PhD. Pandas also allows you to read data from a large host of formats and stores it in a DataFrame object, storing your data in a tabular format, not very dissimilar from a Excel spreadsheet. From this, you have a list of tools for cleaning, reprocessing, querying, quick statistical analysis, and visualization.

Additionally, Pandas makes heavy use of NumPy and is seamlessly integrated with it. This allows us to quickly store the results of complex numerical operations in simple DataFrame objects for quick analysis, providing a smooth and streamlined workflow.

6.3.4 Jupyter Notebook

Python's line-by-line interpretation allows for an interactive and iterative workflow. Being able to run just a few commands, examining their outputs, maybe making a few more adjustments before continuing with another set of commands. This can be quite useful for early exploratory experiments, as we discuss in Sect. 2.

Jupyter Notebooks [26] are actually a web application that can be installed like any other Python package. It's set up on a local server that can run interactive blocks of code, such as Python but also Julia and R, and are output straight to the same page. It takes advantage of the iterative interactive nature of Python (again, Julia and R as well) code, to be able to build annotated iterative workflows to be shared with others.

Jupyter Notebooks are valued highly among computational scientists, who are constantly trying to share and reproduce scientific work. It is not uncommon to see many presentations and tutorials using Jupyter Notebooks in this area. In some cases, Notebooks have been submitted along published work as supporting material, like the famous LIGO detection of gravitational waves experiment! [27].

6.3.5 Scikit-learn

Scikit-learn [18] is a popular open-source ML library in Python. It is also built on top of NumPy and SciPy and provides a wide range of algorithms and tools for ML, statistical modeling, and data analysis. Scikit-learn offers popular tools for building classical well established ML models for classification, regression, clustering, and dimensionality reduction. These are all integral tasks for working in predictive modeling and as such, Scikit-learn will probably be your first point of entry when working on any predictive modeling project.

6.3.6 PyTorch/TensorFlow/Jax

PyTorch [20], TensorFlow [19], and Jax [28] are regarded as deep learning packages, although they can be used more generally. Deep learning is a subset of ML that utilizes large neural networks, sometimes with billions of parameters to preform predictive tasks. The computational requirements for optimizing such models are very different to classical ML methods that can be found in Scikit-learn. Typically, it requires running on specialized hardware like graphical processing units (GPUs) or massive computer clusters. The optimization methods also tend to be numerical iterative algorithms rather than analytic solutions that can be found in most classical ML methods. These two computational requirements inspired the development of these deep learning packages. They consist of merging an Autograd framework [29] with a framework for operating on multi-dimensional arrays on GPUs and accelerated hardware. These requirements mean that PyTorch/TensorFlow/Jax have their own array object, separate to the NumPy array, that have the Autograd capabilities and hardware accelerated compatibility. This allows for efficient implementation of these large-scale neural networks using Python. Additionally, these libraries typically offer pre-built neural network architectures. In truth, you can perform any sort of computation using these libraries if the acceleration and Autograd capabilities are useful.

7 Markup languages for scientific writing

Perhaps you've seen your colleagues produce stunning web pages, documents, or Jupyter notebooks, leaving you curious about what software they used to achieve these results. The secret to their success lies in the efficient use of different markup languages, with Markdown and LaTeX being the most common choices for young researchers. In this section, we will approach the subject from a beginner's perspective and assume that you have little to no prior knowledge of Markdown or LaTeX . We will share some useful tips and tricks that we have used ourselves to learn markup languages, and we hope that you too can benefit from them. Whether you work in experimental chemistry or computational physics, we believe that mastering these markup languages will help you communicate your scientific findings more effectively, and produce more polished and professional documents.

7.1 Markdown

Markdown is a lightweight markup language that allows you to format text using an easy-to-read/write plain text format. A Markdown document can contain embedded graphics, source codes, and formulas. You may even have come across Markdown in several scenarios without realizing it. Two of the most frequent encounters in our experience are Jupyter Notebook Markdown cells and GitHub README files. Markdown is also commonly used for academic blogs, online forums, collaborative software, and documentation pages. The entire history and usage of Markdown can be found on Wikipedia, so you don't really have to look too far! Plenty of Markdown cheat sheets can also be found via a quick online search.

Beginners often encounter challenges when attempting to include bullet points, equations, images or links within the default Markdown environment. However, as an alternative, one could use third-party websites or software for mastering the basics of Markdown. One of the best 'free' third-party websites for Markdown is called HackMD. HackMD offers many features, such as real-time collaboration, integration with other services such as Google Drive and GitHub, and notably, it also has a way of keeping track of any changes made. This is especially helpful where multiple people are working together on a single document, thus allowing users to revert to previous versions if needed. However, its most valuable feature is the ability to render Markdown files while typing so that you can see exactly what's going on. Also, if perhaps you were previously formatting Markdown in a Jupyter Notebook, you can simply copy and paste everything from HackMD into the Jupyter Notebook Markdown cell. HackMD will allow you to sync contents to your GitHub repository if you are setting up your first repository. If you want to better visualize changes made to the README file, as you edit it, you can do so in HackMD and push these changes to your repository.

Alternatively, you can download one of the best Markdown software options (*in my opinion*), such as Visual Studio (VS) Code. With the help of some extensions, everything you were able to do using HackMD, can be accomplished using this platform. Although, for a beginner, we would advise sticking with HackMD as it is much easier to set up, and you can practice Markdown from pretty much anywhere. Thereafter, once you have gained more experience and feel more confident in switching between platforms, transitioning to VS Code will definitely benefit you in the long run.

There are numerous reasons to use Markdown, here are a few:

- Relatively short learning curve.
- As a lightweight markup language, its texts are almost human-readable.
- Markdown is the best choice for a document to be published both on the Web and as printed text, as it can be easily converted into HTML.

7.2 LaTeX

LaTeX is a popular markup language for creating scientific documents/books/articles containing complex typographic elements such as Greek letters and mathematical expressions, and so you can appreciate their relevance in STEM.

To begin with, let's discuss some of the mistakes we all made as beginners when we first got started with LaTeX.

Mistake #1: wasting time trying to pick the best LaTeX editor. While there are a number of wonderful options to choose from, including Overleaf, TeXmaker, TeXStudio, and many others, each with their own merits, we would advise against going down the, albeit very tempting, rabbit hole of trying to master all of these editors before ever learning how to use LaTeX. Instead, opt for a single editor, and stick with it until you are comfortable using LaTeX. For beginners, we highly recommend Overleaf (*the editor used to write this article*), available under https://www.overleaf.com/. A quick Google search reveals its extensive advantages. In fact, if you simply Google 'How to use LaTeX' (or a variation thereof), you will find tons of tutorials, documents and videos covering the subject. Overleaf has become one of the most widely accessed online LaTeX editors, offering a tutorial titled 'Learn LaTeX in 30 min.' This detailed tutorial covers everything you need to know, from 'What is LaTeX?' to 'Basic Formatting' to 'Downloading your finished documents.' We highly recommend you check this out.

Mistake #2: creating your own template from scratch. For experienced users, creating their own templates may still present some issues, thus it's unlikely that a beginner will succeed right away, and doing so only adds to frustration, which eventually demotivates you from using LaTeX. This is why we always recommend users use established templates as a starting point. Luckily, Overleaf is teeming with template options to help you get started, from curriculum vitaes (CVs), thesis templates, and even research paper templates that are specific to the journal you are submitting to. The Overleaf team have kindly organized everything into groups for you, depending on your needs, providing a user-friendly experience. Moreover, you have the convenience of searching for templates based on popular tags, recent uploads or even their selected, or, 'featured' templates. If there is ever something specific that you would like to change in your LaTeX document, you can always refer to the Overleaf Help Library for more information. For example, if you would like to know how to insert hyperlinks in LaTeX, just search 'Hyperlink.' Suppose you are unsure about how to structure a complicated mathematical equation in LaTeX, the online LaTeX equation editor offers a user-friendly platform for composing and rendering your equations.

Outside of writing reports, you may most likely come across LaTeX used in academic presentations (*probably why they look so much alike*). However, there's a reason why LaTeX is so popular among academics for making presentations, despite their slightly 'outdated' look. LaTeX presentations, or, Beamer presentations offer a number of advantages over more traditional Microsoft PowerPoint, Keynote, or Google Slides, and that is, the seamless incorporation of mathematical equations and code snippets, all using LaTeX's powerful typesetting capabilities. This makes all the difference for a computational scientist. Although, we note that one could also use equation tools (outside of Microsoft PowerPoint), and still get LaTeX - quality formatting. Therefore, for those more comfortable with Keynote presentations, one can still apply LaTeX equation formatting and convert it to a high-quality png-rendered image using online LaTeX equation editors such as 'LaTeX to png,' or versions thereof.

However, we would like to highlight that LaTeX can also be a valuable tool for non-computational scientists. In fact, we encourage experimentalists and other professionals to use LaTeX for collaborative document writing. In fact, many lecturers and educators outside of STEM have also been slowly embracing the benefits of LaTeX.

Looking for more reasons to use LaTeX? Here are five of my favorite reasons:

1. Although the learning curve can be steep, once you become proficient, it's a more powerful tool than most other document preparation software tools out there.
2. It is the right choice for complex and high-quality texts, especially for including mathematical or chemical equations.
3. It is the gold standard in STEM regarding publishing scientific documents. Many journals accept raw LaTeX documents for submission.
4. There are tons of LaTeX templates available online. Really! You could scroll through them for hours.
5. LaTeX Editors such as Overleaf offer a cloud-based collaborative writing platform, ideal for group projects, which makes it easy for users to track changes and manage the contributions of others.

8 Publishing (Python) packages

So you finally got your code working – why not share it with the world? In this section I (Charlie Pilgrim) will discuss what I learnt from publishing a Python package [30] as a PhD student. Hopefully this will help you decide whether you want to publish your own package, and give you an idea of the benefits and challenges involved in doing so. While I will focus on Python packages, much of the advice here is general and will apply to any programming language.

8.1 Benefits

The main benefits of publishing a package:

- **Other people can benefit from your code!** This may not be a direct benefit to you, but it feels great to get an email from someone thanking you for your package which is saving them a lot of time and effort.
- **It looks great on your CV**. Having a published package is a strong signal that you are a competent programmer. This can be very helpful if you are going for any kind of position that requires coding experience, whether that be in industry or academia.

- **Citations**. Often you can link a paper to the package, and ask people to cite the paper if they use the code. This could be a paper you have already written related to some research code (e.g., a new algorithm) or you can publish a short paper about the package itself. For example, the *Journal of Open Source Software* is a venue for short articles that allow researchers to get cited for software.
- **Get your name out there**. If you have developed some niche software then releasing the code is a good way to become known to researchers in your field. It is also perfectly reasonable to email other researchers in your area to ask them what they think of your software package - this can even lead to collaborations or job offers.
- **Publishing code is a good practice to get into**. After publishing a package I found that I naturally wrote code that was tidier, better structured and generally closer to publishing. This kind of code is much easier to work with, especially when you come back to a project after a few months. And of course writing code to a high standard to begin with makes if easier to then publish the code in a public repository, which is becoming more important in research [31].
- **You will learn a lot**. As we will see in the next section, there are lots of challenges to publishing packages. While overcoming challenges can feel difficult at the time, it is a great way to learn.

8.2 Challenges

Mostly the challenges of publishing a package are around making your code *production ready*. Software developers distinguish development code, which is essentially for your own use and as such can be messy and incomplete, and production code, which is for others to use and is of a generally higher quality.

Getting your code production ready and published will involve:

- **Tidying up**. As you develop code there is a tendency for things to get messy and confusing. Before publishing any code you will probably want to spend some time and effort tidying things up.
- **Writing Documentation**. You will need to tell users how to use your code. This includes the README file and other written documentation, as well as elements within the code itself such as function definitions and code comments.
- **Testing**. While writing your software you probably did at least some manual testing to check a few things work as expected. In order to be fully confident that the package works as expected you might want to write some more standardized tests such as *unit* or *functional* tests.
- **Covering edge cases**. What happens if a user does something unexpected? This may or may not be important to cover, depending on who you expect to use the package.
- **Extra features**. You will feel pressure to add extra functionality. Try and resist this! Doing one thing very well is a great aim for a package.
- **Actually publishing the package**. This is actually one of the simpler steps! For example, there are a number of tutorials to publish code to the Python package index which take less than an hour.
- **Ongoing maintenance**. You may get requests for new features, or people might tell you about bugs. And you might need to update your package to work with new versions of software dependencies as they are released, e.g., new Python versions. Of course, you can decide whether to address these issues when they arise.
- **Intellectual property and licensing**. When you publish code you will also need to provide a license that sets out the rules for how other people can use the code. A common choice is permissive open-source licenses such as the MIT license which allow anyone to re-use the code. If you are using or adapting someone else's code [32] then it is good practice to make sure that you follow the rules set out in any existing licenses.

8.3 Conclusion

Only you can decide whether to publish a package. Is it worth it? If you want to publish a package for general consumption then there is a lot of work involved in getting the code *production ready*. This work may be worth it if it will really help your career or you have a burning desire to publish the code. However in academia you might want to publish a small niche package that does just one thing very well. This could be extremely helpful for a small number of researchers. Often this kind of package does not need to be quite as polished as a more general package, as the users will be willing to work through any issues.

Hopefully this section has made the whole process of publishing packages a little more transparent. Even if you don't personally publish a package, it is generally useful to have an idea of what is involved in the process.

9 How do I make my work reproducible?

Science, in general, has been plagued by results that are not easily reproducible - and perhaps to some extent, this is to be expected; cutting-edge research is inherently difficult; if it weren't, it probably wouldn't be cutting edge. However, we should also expect it to be easier on the second or third attempt.

Reproducibility is an essential component of scientific research. It is what ensures the credibility of results discussed in research journals. What good is it if a research group at one institution claims the synthesis of a previously undiscovered material, but another research group, at another university cannot follow their experimental methodology to produce the same ‘novel’ material. Well, the same is true of computational code. In the ensuing discussion, we highlight seven key points that we deem most important to keep in mind when publishing code, so that one might be able to replicate the results provided.

1. **Documentation:** Perhaps an obvious point, but detailed documentation goes a long way. This must include a clear description of how your code is organized, what each script does, and how to execute the code. Popular tools for this include Sphinx [33] and Doxygen [34].
2. **Version control:** In the previous section we discussed how one might publish a Python package. This will have likely involved keeping track of previous versions of code using version control systems like Git. Platforms like Gitlab [35] and GitHub [36] are primed for this sort of thing, and are indispensable tools for keeping up to date with the latest versions of code. There are a ton of tutorials on how to use Git on the internet (*which is why we won’t get into it here*), so please do look into these yourself if you are not already familiar. For a gentle introduction, check out this video by Programming with Mosh on YouTube on how to get started with Git [37].
3. **Code dependencies:** Declare all code dependencies and versions of software and libraries used from the outset. You might typically include this in the README.md file as part of your code package. Also, the use of virtual environments can help you to avoid any clashes. If you have used random number seeds anywhere in your code, you should clearly document that, as this is critical for reproducibility.
4. **Tutorials:** Again, as with the documentation, having a full tutorial of the code, e.g., in the form of a Jupyter Notebook, complete with visualization of expected results, will be extremely helpful to users, so that they know what to expect.
5. **Open source:** Perhaps another obvious point, but any data used to produce your results should be made publicly available to the users of your code. This can be easily done through platform like Figshare [38] and Zenodo [39]. All it takes is citing this link in your research paper.
6. **Unit testing:** As discussed in the previous section, it might help if you include some sort of unit testing infrastructure in your published code e.g., `Pytest`. This will help ensure the accuracy of your code. A great tutorial on unit testing Python code is offered by Corey Schafer on YouTube [40], so do check out his video, and others like it.
7. **General good coding practices:** By sticking to traditional coding standards you reduce the risk of users misinterpreting your code, and are thereby more likely to have success in replicating your results. This includes things like how you structure your code, e.g., keeping the `src`, `examples`, and `docs` directories separate. Obvious! but necessary if you want to avoid confusion. There are a number of websites and resources you can consult to learn about the principles and practices of software design. For your convenience, we have provided links to five websites [41–45].

10 Recommended software and resources

For an impressive list of software programs for computational chemistry and physics, check out silicostudio by David Abassi [46]. Additionally, below are a few other websites, papers, and YouTube channels that you may find helpful.

Websites

- MolSSI best practices for software development [47]—a teaching resource by the molecular sciences software institute.
- The missing semester of your computer science degree [48]—a seminar series by graduate students at MIT.
- Introduction to Deep Learning [49]—a lecture course by graduate students at MIT.

YouTube channels

- TMP Chem [50]—introductory videos on computational chemistry.
- Virtual Simulation Lab [51]—computational tools seminar series introducing version control, visualization, uncertainty quantification, and much more.
- StatQuest [52]—videos on statistics, machine learning, and data science.
- The Computational Toolkit [53]—a virtual seminar series offering a practical introduction to tools, tips, and tricks in computational science.

Papers and other resources

- Cumby et al. [54]—a collection of Jupyter Notebooks to guide students with data-driven chemistry, and introduces Python programming along with its usage in data analysis, typically required for a chemistry degree.
- French [55]—a sequential learning system designed to guide the reader on how to use R programming to analyze data.
- Storopoli et al. [56]—an open-source and open-access book on how to do data science using the Julia programming language.
- Pilgrim et al. [32]—ten simple rules for working with other people’s code.
- White [57]—an eBook on deep learning for molecules and materials.

- Solomon et al. [58]—tips on writing your first journal paper.
- Shi et al. [59]—review on machine learning in chemistry.
- Rodrigues [60]—review on machine learning in physics.

11 Conclusions

In this paper, we have identified and discussed various topics that we believe will be useful and relevant to academic researchers starting a career in computational science. Here, we have advised where new researchers can look to get started with their research, covered useful programming languages that will likely be required at some point, as well as other key topics that are frequently used such as machine learning and high-performance computing. We also discuss some pastoral aspects associated with computational research and encourage students to discuss matters with other academics, both within and outside their research group. This paper should act as a useful resource to new computational scientists and aid them as they start their research careers, as well as learn about important concepts encountered by previous computational science PhD students.

Acknowledgements I.I., D.M., J.M.T., Z.F., C.M., and C.D.W. acknowledge funding from the Engineering and Physical Sciences Research Council (EPSRC) Centre for Doctoral Training in Modelling of Heterogeneous Systems [EP/S022848/1]. S.C. acknowledges funding from the EPSRC Centre for Doctoral Training in Diamond Science and Technology [EP/L015315/1] and the Research Development Fund of the University of Warwick. C.P. acknowledges funding from the EPSRC Mathematics for Real-World Systems Centre for Doctoral Training [EP/S022244/1]. In addition, we would like to acknowledge the valuable contributions of several other colleagues for their early efforts and input in the computational toolkit seminar series, on which this paper is based, and they are Peter Lewin-Jones, Kyle Fogarty, Lakshmi Shenoy, Matthew Harrison, and Charlotte Rogerson. Finally, we would like to thank both Professors James Kermode and Julie Staunton (University of Warwick) for their time in reading our drafts and offering valuable advice and comments on our manuscript.

Author contributions All authors have contributed equally to this manuscript.

Funding This study was funded by Engineering and Physical Sciences Research Council (EPSRC) [EP/S022848/1].

Data availability Not applicable.

Code availability Not applicable.

Declaration

Conflict of interest The authors declare that they have no conflict of interest.

Ethical approval Not applicable.

Consent to participate Not applicable.

Consent for publication Not applicable.

References

1. S.J. Clark, M.D. Segall, C.J. Pickard, P.J. Hasnip, M.J. Probert, K. Refson, M.C. Payne, First principles methods using CASTEP. Z. Kristall. **220**, 567–570 (2005)
2. P. Rüßmann, P. Mavropoulos, R. Zeller, J. Bouaziz, M. Dos Santos Dias, S. Blügel, D.S.G. Bauer, P.F. Baumeister, M. Bornemann, S. Brinker, P.H. Dederichs, B.H. Drittler, N. Essing, G. Géranton, N.H. Long, S. Lounis, E. Mendive Tapia, E. Rabel, F. Dos Santos, B. Schweflinghaus, D. Antognini Silva, A.R. Thiess, B. Zimmermann, The JuKKR code (2022). https://doi.org/10.5281/zenodo.7284738
3. C.D. Woodgate, D. Hedlund, L.H. Lewis, J.B. Staunton, Interplay between magnetism and short-range order in medium- and high-entropy alloys: Crconi, crfeconi, and crmnfeconi. Phys. Rev. Mater. **7**, 053801 (2023). https://doi.org/10.1103/PhysRevMaterials.7.053801
4. R. Chadwick, Linux Tutorial for Beginners - Learn Linux and the Bash Command Line. [Online; accessed 20. Oct. 2023] (2023). https://ryanstutorials.net/linuxtutorial
5. R. Chadwick, Bash Scripting Tutorial - Ryans Tutorials. [Online; accessed 20. Oct. 2023] (2023). https://ryanstutorials.net/bash-scripting-tutorial
6. K. Dowd, C.R. Severance, *High performance computing*, 1st edn. (O'Reilly & Associates, Cambridge, 1998)
7. W.P. Huhn, B. Lange, V.W.-Z. Yu, M. Yoon, V. Blum, GPU acceleration of all-electron electronic structure theory using localized numeric atom-centered basis functions. Comput. Phys. Commun. **254**, 107314 (2020). https://doi.org/10.1016/j.cpc.2020.107314
8. F. Spiga, I. Girotto, phiGEMM: a CPU-GPU library for porting quantum ESPRESSO on hybrid systems. In: 2012 20th Euromicro international conference on parallel, distributed and network-based processing, pp. 368–375 (2012). https://doi.org/10.1109/PDP.2012.72 . ISSN: 2377-5750
9. L. Vogt, R. Olivares-Amaya, S. Kermes, Y. Shao, C. Amador-Bedolla, A. Aspuru-Guzik, Accelerating resolution-of-the-identity second-order Møller-Plesset quantum chemistry calculations with graphical processing units. J. Phys. Chem. A **112**(10), 2049–2057 (2008). https://doi.org/10.1021/jp0776762. (**Accessed 2023-06-29**)
10. K. Wilkinson, C.-K. Skylaris, Porting ONETEP to graphical processing unit-based coprocessors. 1. FFT box operations. J. Comp. Chem. **34**(28), 2446–2459 (2013). https://doi.org/10.1002/jcc.23410

11. J. Yan, L. Li, C. O'Grady, Graphics processing unit acceleration of the random phase approximation in the projector augmented wave method. Comput. Phys. Commun. **184**(12), 2728–2733 (2013). https://doi.org/10.1016/j.cpc.2013.07.014. (**Accessed 2023-06-29**)
12. L. Genovese, M. Ospici, T. Deutsch, J.-F. Méhaut, A. Neelov, S. Goedecker, Density functional theory calculation on many-cores hybrid central processing unit-graphic processing unit architectures. J. Chem. Phys. **131**(3), 034103 (2009). https://doi.org/10.1063/1.3166140. (**Accessed 2023-06-29**)
13. C. Bishop, Pattern recognition and machine learning. J. Electron. Imaging **16**(4), 140–155 (2006). https://doi.org/10.1117/1.2819119
14. M.A. Lones, How to avoid machine learning pitfalls: a guide for academic researchers. arXiv (2021) https://doi.org/10.48550/arXiv.2108.02497, arXiv:2108.02497
15. M. Belyaev, E. Burnaev, Y. Kapushev, Exact inference for gaussian process regression in case of big data with the cartesian product structure. arXiv (2014) https://doi.org/10.48550/arXiv.1403.6573, arXiv:1403.6573
16. H. Liu, Y.-S. Ong, X. Shen, J. Cai, When gaussian process meets big data: a review of scalable GPs. arXiv (2018) https://doi.org/10.48550/arXiv.1807.01065, arXiv:1807.01065
17. L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, G. Varoquaux, API design for machine learning software: experiences from the scikit-learn project. In: ECML PKDD workshop: languages for data mining and machine learning, pp. 108–122 (2013)
18. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, É. Duchesnay, Scikit-learn: machine learning in python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
19. M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G.S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, TensorFlow: large-scale machine learning on heterogeneous systems. Software available from tensorflow.org (2015). https://www.tensorflow.org/
20. A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala, PyTorch: An Imperative Style, High-Performance Deep Learning Library. Curran Associates, Inc. (2019). http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf
21. J.D. Hunter, Matplotlib: a 2D graphics environment. Comput. Sci. Eng. **3**, 90–95 (2007)
22. C.R. Harris, K. Jarrod Millman, S.J. Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N.J. Smith, R. Kern, M. Picus, S. Hoyer, M.H. Kerkwijk, M. Brett, A. Haldane, J. Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, T.E. Oliphant, Array programming with NumPy. Nature **585**, 357–362 (2007)
23. P. Virtanen, R. Gommers, T.E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S.J. Walt, M. Brett, J. Wilson, K. Jarod Millman, N. Mayorov, A.R.J. Nelson, E. Jones, R. Kern, R. Larson, C.J. Carey, İ Polat, Y. Feng, E.W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Heriksen, E.A. Quintero, C.R. Harris, A.M. Archibald, A.H. Ribeiro, F. Pedregosa, P. Mulbregt, SciPy 1.0 Contributors: SciPy 1.0: fundamental algorithms for scientific computing in python. Nat. Methods **17**, 261–272 (2020)
24. M.L. Waskom 2021 seaborn: statistical data visualization. J. Open Source Softw. 6(60), 3021 https://doi.org/10.21105/joss.03021
25. T. Team, pandas-dev/pandas: pandas. Zenodo (2020). https://doi.org/10.5281/zenodo.3509134
26. T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, Jupyter notebooks – a publishing format for reproducible computational workflows. In: Loizides, F., Schmidt, B. (eds.) Positioning and power in academic publishing: players, Agents and Agendas, pp. 87–90 (2016). IOS Press
27. Ligo: tutorials. [Online; accessed 13. Nov. 2023] (2023). https://gwosc.org/tutorials
28. J. Bradbury, R. Frostig, P. Hawkins, M.J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, Q. Zhang, JAX: composable transformations of Python+NumPy programs (2018). http://github.com/google/jax
29. D. Maclaurin, D. Duvenaud, R.P. Adams, Autograd: effortless gradients in numpy. In: ICML 2015 AutoML Workshop, vol. 238, p. 5 (2015)
30. C. Pilgrim piecewise-regression (aka segmented regression) in Python. J Open Source Softw, 6(68):3859 (2021) https://doi.org/10.21105/joss.0385
31. R.D. Peng, Reproducible research in computational science. Science **334**(6060), 1226–1227 (2011). https://doi.org/10.1126/science.1213847
32. C. Pilgrim, P. Kent, K. Hosseini, E. Chalstrey, Ten simple rules for working with other people's code. PLoS Comput. Biol. **19**(4), 1011031 (2023). https://doi.org/10.1371/journal.pcbi.1011031
33. Sphinx: Sphinx. [Online; accessed 6. Nov. 2023] (2023). https://www.sphinx-doc.org/en/master
34. Doxygen: Doxygen: Doxygen. [Online; accessed 19. Oct. 2023] (2023). https://www.doxygen.nl/index.html
35. Gitlab: The DevSecOps Platform. [Online; accessed 19. Oct. 2023] (2023). https://about.gitlab.com
36. Github: Build software better, together. [Online; accessed 19. Oct. 2023] (2023). https://github.com
37. P.w. Mosh, Git tutorial for beginners: learn Git in 1 hour. Youtube. [Online; accessed 12. Nov. 2023] (2020). https://www.youtube.com/watch?v=8JJ101D3knE
38. Figshare: figshare - credit for all your research. [Online; accessed 12. Nov. 2023] (2023). https://figshare.com
39. Zenodo: Zenodo. [Online; accessed 12. Nov. 2023] (2023). https://zenodo.org
40. C. Schafer, Python tutorial: unit testing your code with the unittest Module. Youtube. [Online; accessed 12. Nov. 2023] (2017). https://www.youtube.com/watch?v=6tNS–WetLI
41. GeeksforGeeks: principles of software design. GeeksforGeeks. [Online; accessed 12. Nov. 2023] (2022). https://www.geeksforgeeks.org/principles-of-software-design
42. K. Chris, SOLID Design principles in software development. FreeCodeCamp (2023)
43. B.W. Boehm, Seven basic principles of software engineering. J. Syst. Softw. **3**(1), 3–24 (1983). https://doi.org/10.1016/0164-1212(83)90003-1
44. Archiveddocs: chapter 16: quality attributes. [Online; accessed 12. Nov. 2023] (2023). https://learn.microsoft.com/en-us/previous-versions/msp-n-p/ee658094(v=pandp.10)?redirectedfrom=MSDN
45. Molssi: MolSSI's Best Practices – MolSSI. [Online; accessed 12. Nov. 2023] (2023). https://molssi.org/molssis-best-practices
46. D. Abbasi, Cutting-edge free tools to unlock the power of computational chemistry - Silico Studio. Silico Studio (2023)
47. Molssi: MolSSI's Best Practices – MolSSI. [Online; accessed 19. Oct. 2023] (2023). https://molssi.org/molssis-best-practices
48. A. Athalye, The missing semester of your CS education. [Online; accessed 19. Oct. 2023] (2023). https://missing.csail.mit.edu
49. M.D. Learning, MIT deep learning 6.S191. [Online; accessed 19. Oct. 2023] (2023). http://introtodeeplearning.com
50. T. Chem, Computational Chemistry 0.1 - Introduction. Youtube. [Online; accessed 19. Oct. 2023] (2017). https://www.youtube.com/watch?v=YF-amZgE2h4 &list=PLm8ZSArAXicIWTHEWgHG5mDr8YbrdcN1K
51. Virtual Simulation Lab. Youtube. [Online; accessed 19. Oct. 2023] (2023). https://www.youtube.com/@VirtualSimulationLab/videos
52. StatQuest with Josh Starmer. Youtube. [Online; accessed 19. Oct. 2023] (2023). https://www.youtube.com/@statquest
53. The Computational Toolkit. Youtube. [Online; accessed 19. Oct. 2023] (2023). https://www.youtube.com/@thecomputationaltoolkit2890/videos

54. J. Cumby, M. Degiacomi, V. Erastova, J. Güven, C. Hobday, A. Mey, H. Pollak, R. Szabla, Course materials for an introduction to data-driven chemistry. J. Open Source Educ. 6(63), 192 (2023) https://doi.org/10.21105/jose.00192
55. T. French R for data analysis: an open-source resource for teaching and learning analytics with r. J. Open Source Educ. 6(63), 202 (2023) https://doi.org/10.21105/jose.00202
56. J. Storopoli, R. Huijzer, L. Alonso, Julia Data Science, (2021). https://juliadatascience.io
57. A.D. White, Deep learning for molecules and materials. Living J. Comput. Molecul. Sci, 3(1), 1499 (2021) https://doi.org/10.33011/livecoms.3.1.1499
58. G.C. Solomon, J.Z. Zhang, T. Cuk, An open letter to aspiring authors. ACS Phys. Chem. Au **2**(2), 68–69 (2022). https://doi.org/10.1021/acsphyschemau.2c00011
59. Y.-F. Shi, Z.-X. Yang, S. Ma, P.-L. Kang, C. Shang, P. Hu, Z.-P. Liu, Machine learning for chemistry: basics and applications. Engineering (2023). https://doi.org/10.1016/j.eng.2023.04.013
60. F.A. Rodrigues, Machine learning in physics: a short guide. Europhys. Lett. **144**(2), 22001 (2023). https://doi.org/10.1209/0295-5075/ad0575