EMMC-CSA

European Materials Modelling Council

White paper

for standards of modelling software development

TABLE OF CONTENT

The EMMC-CSA project has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement No 723867

1. Executive summary

1.1 Description of the deliverable content and objectives

This white paper provides a basis for the standards of modelling software development and addresses areas such as method description, assumptions, accuracy and limitations; testing requirements; issue resolution; version control; user documentation and continuous support and resolution of issues.
The document is based on the work already carried out in the context of the EMMC to drive the adoption of software quality measures, and to ensure sustainable implementation of this EMMC initiative. Given the high level of sophistication of each of the developments which solve particular aspects of the multi-physics/chemistry spectrum of materials modelling, the industrial usefulness of individual achievements requires integration into larger software systems. Thus, guidelines and standards are needed, which will enable the exploitation of these codes.

1.2 Major outcome

This white paper provides guidelines for academic software developers creating materials modelling codes. In many cases, design decisions taken at an early stage have unforeseeable consequences for many years ahead. In this context, the white paper gives academic researchers a framework, which paves the way for successful integration and industrial deployment of materials modelling. This goal is achieved by addressing a range of topics including model descriptions and software architectures, implementation, programming languages and deployment, intellectual property and license considerations, verification, testing, validation, and robustness, organization of software development, metadata, user documentation, and support.

2. Progress report (main activities)

2.1 Introduction

The European Materials Modelling Council – Coordination and Support Action (EMMC-CSA) was installed in 2016 by the European Commission as a joint effort to foster materials research in the European industry. The goal of this CSA is to stimulate and enhance the use of materials modelling as an integral part of industrial research and development. These efforts intend to further increase the strength and competitiveness of the European manufacturing industry in a global market.
The EMMC-CSA combines the expertise and knowledge of partners from academia and industry to establish and strengthen the links between software developers (mainly research groups at universities), software distributors, and software users (mainly industrial R&D departments). To make closer connection between these groups, bridge the gap between the different paradigms underlying their operation methods, and establish a common language, it is suggested to create a platform for exchange. The present white paper is meant to serve this purpose with respect to software development and deployment. All steps of the life cycle ranging from the first ideas to support and documentation are addressed and put on a firm basis, which will serve as a framework for all parties involved in the process.
This white paper is put forward by EMMC members from Materials Design SARL (MDS) and QuantumWise A/S (QW) and is provided as a basis for the standards of modelling software development. The objective is to share some of the best practises in commercial software development to enhance the quality of academic software. Furthermore, one of the most important aspects of academic software projects is the training of students, and using a more advanced approach to software development will better position them for a job outside academia. Specifically, the white paper addresses areas including method description, software architecture, programming

languages, accuracy and limitations; testing requirements; issue resolution; version control; user documentation and continuous support and resolution of issues.
Of course, this white paper can only briefly touch several issues arising in the context of scientific software development. Readers interested in detail are referred to external sources of information including the references and links given at the end of this white paper.

2.2 Scope

Scientific computing traces back to the mid of the 20th century, when the ground-breaking technological discoveries as, e.g., the development of the transistor, were made and soon thereafter implemented in electronic devices with ever increasing integration and computational speed, which is best described by Moore's law. Major software developments aimed at making this computer power accessible via high-level programming languages, which were soon adopted by scientists to code their formulas and to get their problems solved; Fortran ("*for*mula *tran*slation") was one of these early languages (and still has its share in scientific computing).

Since these early days, an incredible amount of scientific software in applied mathematics, engineering, physics, chemistry, biology, geography, astronomy, and other disciplines has been developed. Most of this work was performed at universities and public and/or governmental laboratories, mainly by students and postdocs as part of their research. This had a strong impact on the way this software was designed, written, documented, and maintained. Since continuity quite often was hampered by the students and postdocs leaving from academic groups, writing of such code was often done with a rather short time horizon in mind, which resulted in "quick and dirty" coding. In addition, design, documentation, and maintenance in many cases were completely ignored. For similar reasons, distribution and licensing considerations in general also played no role at all. Therefore, many scientific codes were in a rather bad shape (or did not at all survive the time of a thesis or research paper) and still are, since many codes were passed down from one student generation to the next, which experienced similar research conditions as the generation before. This describes the status of most scientific software rather well with materials modelling software in general being no exception. Expecting a fundamental paradigmatic shift of this general situation probably is far too optimistic. Yet, it is fair to say that there is some variation in the maturity in materials modelling codes and some codes have made it already into commercial software products.

The past two or three decades have thus seen some development, which started with the commercialization of quantum chemistry codes for use by the chemical and pharmaceutical industry and soon after extended to codes for materials modelling. It should be noted here that the use of atomistic simulations in the chemical and pharmaceutical industry has already a long history with well-established and commercially supported software, the same is not the case for materials modelling software for the manufacturing industries, which are the focus of the present effort.

Software companies like Biosym (MSI, Accelrys, Biovia), Materials Design and Schrödinger spent much effort making originally academic materials modelling codes accessible to industrial application by adding functionality and/or graphical user interfaces, increasing the robustness, ensuring validation and verification, and providing documentation, maintenance, and long-term support. Earlier, also supercomputer companies such as Cray Research and Fujitsu have developed professional software systems for electronic and atomistic modelling. For example, in the late 1980's, Cray Research developed from scratch a density-functional code (DGauss). Combined with academic software and graphical user interfaces this became UniChem, which was used industrially in applications such as catalysis. Today we also see companies like QuantumWise (Synopsys) and SCM developing new simulation codes from scratch sometimes in close collaboration with academic groups or making extensive use of academic software libraries. All these companies thus serve the important purpose of

bridging the gap between the different perspectives and expectations of academic research in simulation, academic software developers and industrial end-users and therefore play a key role in the deployment process. They have thereby established a value-added chain reaching from academic research to (mostly) industrial end users.

Yet, still a lot can be done to ease the whole process and to make it more efficient. This White Paper addresses the software developers and provides them with basic guidelines for software development. It is the goal of this White Paper that these guidelines enable a more rapid integration of materials modelling software into industrial modelling software packages. For this reason, this White Paper focuses especially on those aspects of software development, which so far obtained only little attention as outlined above. Some of these were often implicitly included in the initial development process but received only little explicit consideration. We will thus put special emphasis on the following aspects:

- **Model description and software architecture**

  The first step underlying any software development is a clear definition of the model, which shall be described by the software and the construction of a software architecture which supports the model and is flexible towards future extensions.

- **Implementation, programming language and deployment**

  It is important to select a programming language which has the right balance between rapid development and performance. The choice may also impact whatever is possible to get contributions from the community and the ease of deployment on different operating systems and hardware platforms. A very important and often completely neglected feature of software developed in an academic environment regards code documentation, which strongly affects readability and enhancements of the code.

- **Intellectual property and license considerations**

  Intellectual properties considerations should accompany specifically academic software development, which as outlined above often is based on the work of generations of students and postdocs, who all take part in the value creation.

- **Verification, testing, validation and robustness**

  Code verification is often not done in a systematic manner. The same applies to testing procedures, which often are not established and performed, leading to reduced robustness of code. Formal verification of a materials modelling software must eventually be complemented by the validation of the initially defined underlying model. This step includes comparison of calculated results with experimental data or data created by other software.

- **Organisation of the software development**

  The software development may involve a group of researchers working at different location. This requires a strong control of different versions and merging into common branches. Furthermore, there must be strong leadership and planning to avoid branching of the development. Another aspect is bug tracking and feature requests. Industrial software development benefits a lot from standard tools of modern software engineering, which, however, are rarely used in academia.

- **Metadata**

  Another often overlooked feature in academic software development concerns writing of metadata to all output and internal files generated by the software, which would allow to reproduce and check calculated results.

- **User documentation**

  Of course, complex materials modelling software should come with an extended user's guide, which explains the intention of the code, its capabilities and limitations, as well as all input and output. Ideally, user documentation will also explain the functionality of the software at the hand of examples.

- **Support**

  In addition to robustness, long-term support is a key requirement of industrial end users. However, long-term support conflicts with a lack in knowledge dissemination in academia, which covers models and results but to a much lesser degree the software connecting these two.

2.3 Model description and software architecture

The first step in developing a new software project is to get an overview of the scope of the software and a vision for how the software could develop in the future. The implementation could have different stages, where in the first stage a simple model is developed and in later stages additional features are added to the model. It is important to select a software architecture for the initial stage which allows for extending the model without having to reprogram the entire core of the software. If the software is intended to grow too many thousands of lines, it is important to make an architecture where a new student or researchers can contribute without having to understand the entire software.

The most important aspect of a good software architecture is to make it modular with well-defined interfaces between the modules. A well-defined interface will mean that you can make local changes in one module without affecting the rest of the software. This will greatly increase the readability of the software and make it easier for new contributors to get started, since the new developer will initially just have to learn a single module and can make changes in this module without affecting the functionality of the rest of the software.

When modularising the software, it is also important to investigate if there are available software libraries which can be used to cover some aspects of the functionality. Using libraries can greatly reduce the implementation work and maintenance effort of the software. Furthermore, using well established libraries will make the software less error prone, since the library is shared among many applications and it is more likely that bugs will be discovered.

Some future extensions may require global changes in the software. An example could be an implementation of Density Functional Theory (DFT) which in the first version is implemented to only handle unpolarised systems while in later versions should handle spin-polarised or non-collinear systems. In this case the electron density will be a 1-component scalar field in the first version, but in later versions should be a 2-component or 4-component scalar field. In this case, one can consider using an object-oriented software architecture where such implementation details are hidden behind an electron density object such that the addition of spin-polarised and non-collinear systems will be a local modification in the object. Using object oriented programming also often increases the readability of the software and also adds an educational aspect to the training of students contributing to the project. However, using an object-oriented software architecture may require computer

science skills beyond the knowledge of some contributors and limit the number of contributors. Using object-oriented architectures may also require deep knowledge of programming to obtain good performance.

Thus, the most important aspect is to plan the software architecture beyond the first stage from the beginning and envision how future extension can be implemented.

2.4 Programming language and deployment

Once decisions about the software architecture have been made, the next step is to select a programming language. An important aspect of selecting the programming language is to ensure that the programming language is supported for the operating systems and hardware platforms where the application is supposed to be deployed. Below we discuss the pros and cons for different languages.

Fortran is probably the most popular language within the scientific community and in modern versions it is possible to use object-oriented frameworks. Fortran has the advantage of being able to interface with older numerical software from the scientific community and a large number of modules may be available. In particular, many researchers from the older generation know the language and it may be easier to build a community of established researchers. Fortran tends to give good performance and compilers exist for most supercomputer architectures. Fortran programs tend to be not so well structured and less readable, thus, using Fortran requires high discipline to get readable software. It is not commonly used in commercial software settings thus the training aspect of students is low, in particular if they work on a poorly structured Fortran program.

Python is becoming increasingly popular in the scientific community. It has the advantage of not being a compiled language and this makes it very fast for prototyping new functionality. It is also easy to deploy a python program on different operating systems. There is an increasing number of available scientific libraries and it is possible to make plots and scripting within python. Jupytor notebooks are becoming popular for documenting work done in Python. Python is also used by leading software companies, for instance Google, and it prepares students well for commercial software development. Since Python is an interpreted language, it can lack performance. This can be handled by moving some of the functionality to modules in a compiled language, however, then python should be linked to these modules and this increases the complexity of development and deployment of the software. Since programs are not compiled, many errors may only be found at run time and python programs requires an extensive testing environment.

c++ is one of the most popular programming languages for commercial software projects and is also increasingly used for scientific applications. It allows for developing well-structured and modular software and c++ software is therefore often easier to extend and maintain. Skilled c++ software developers are of great demand in the software industry. c++ programs can attain the same performance as Fortran programs, however, may require avoiding using some constructs in the language. Some supercomputer architectures may also lack good c++ compilers. In the older generation of researchers, there may be less knowledge about c++ programming and generally good c++ software design may require more knowledge than other programming languages. In many universities, students obtain formal training in object oriented programming and c++, while, it is rarer to see courses teaching Fortran.

Generally, selecting a programming language can boil down to: the programming skills of the community that should contribute to the software, ease of deployment on different hardware architectures and performance of the application. An additional aspect that should also be considered is the relevance of the training students obtain when working with the software.

Once the programming language has been selected, it is important to follow standard coding guidelines for this language. This may for instance be how to use indents, underscores, Capital or CamelCase name conventions, using standard conventions will make the software more readable for outsiders.

Needless to say, detailed source code documentation is a must for sustainable software development. Many programmers would agree that half a year after programming many pieces of their own source code look like having been written somebody else. Nevertheless, source code documentation usually has least priority to many programmers and often is omitted at all. This is a major obstacle to software maintenance and it makes enhancements of code rather difficult. Classically, a programmer would add documentation to each piece of software directly to the source code file in the form of commented lines, which are invisible to a compiler but may fully describe the implemented logic and algorithms, possibly also the limitations of a particular piece of code. This is even so for modern documentation tools such as Doxygen or sphinx, which generate nice and well-structured documentation from source code and therefore have found widespread use. However, eventually they all build on the information provided by the programmer inside the source code file. Since providing such information is not formally required by any software tool it is left to the self-discipline of the single programmer and/or the rules agreed on by a team of software developers to make software development a sustainable process. Again, needless to say, minutes spend in code documentation may save hours of analysing the code structure at a later stage.

2.5 Intellectual Property and License Considerations

Legal matters are not at the top of the list of most scientists and the same is true for writers of scientific software, be they physicists, chemists, or engineers. For this reason, they quite often end up opting for what they regard as the simplest solution, which in many cases means simply disregarding the matter at all or just opting for free dissemination of their software without enforcing any regulations. However, ignoring legal issues is the worst thing to be done as it essentially means giving away control. Hence, the present section serves the purpose of explaining the basic terms and giving a short overview of the most important options a scientist has to protect his or her intellectual property.

Of course, before any license considerations can be made, the intellectual property rights and the respective ownership need to be clarified. In this context, we must distinguish legal regulations in different countries. For instance, in the US, software written by employees is usually work for hire and thus the rights to the software belong completely to the employer. However, the distribution of the intellectual property rights may be different in other countries and it may depend on institution or be regulated by a specific agreement between employer and employee.

Another issue regards the extent of the intellectual property rights. At the international level the Agreement on Trade-Related Aspects of Intellectual Property Rights (TRIPS) between all members of the World Trade Organization (WTO) established minimal requirements binding further national regulations. TRIPS was negotiated in the context of the General Agreement on Tariffs and Trade (GATT) in 1994 and is administered by the WTO. Nevertheless, a controversial discussion concerning the property rights applicable to software is still ongoing. This is first related to the fact that there exists no legal definition of a “software patent”. In the US, patent law excludes “abstract ideas”. This point of view has been confirmed by a decision of the Supreme Court in 2014 and has been used to refuse patents involving software. According to the European Patent Office a program for a computer is not patentable unless it has the potential of causing a technical effect. To summarize, software in general cannot be patented but the intellectual property rights fall under the standard copyright laws.

Once ownership issues as mentioned above have been sorted out, decisions are to be made regarding a useful license model, which reflects the intentions of the software owner. The importance of license decisions is often underestimated among software developers especially from academia. However, the license model can have considerable impact on code development and thus license considerations should be included from the very beginning of a software development project. An additional advantage of choosing an appropriate licensing model already in a very early stage of a software project especially in academia is the fact that quite often many people, usually PhD students and postdocs, contribute. Getting their consent on a particular license model several years after the development started might be very difficult if not impossible and this fact may well spoil any sensible setup of a license strategy.

In general, two main types of software can be distinguished with respect to the underlying license model, namely, free or open-source software and proprietary software. In short, the former is characterized by the full transferral of all rights to a software from the original software developer and owner to the user. In this context, “free” is understood in terms of “liberty” rather than pricing. Free software is thus based on distribution of the source code and includes the right to fully control the software, i.e. to use, modify, and redistribute the software. In addition, the user himself may freely use the software for commercial purposes. Indeed, with this software model strict distinction between software developer and user melts away and eventually software development becomes an effort of many users or a whole community of developers. An often-quoted success story in this context is the development of the Linux kernel, which, once started by Linus Torvalds, soon triggered the excitement and work of many. Another well-known example is the GNU project started by the Free Software Foundation (FSF) in the mid-eighties, which likewise led to an enormous amount of software including, e.g., free compilers or free office software, as based on free-software principles and the combined efforts of numerous developers, which in combination with the Linux kernel led to the GNU/Linux operating system. Clearly, without the free-software model the creation of a full-blown free operating system would not have become possible. However, it should be noted that within such software development projects identification of the particular software owners can be cumbersome.

Specifically, free software is defined by the so-called four freedoms formulated by Richard Stallman, the founder of the FSF, in 1986 and published by the FSF as follows:

- Freedom 0: The freedom to **run** the program for any purpose
- Freedom 1: The freedom to **study** how the program works, and change it to make it do what you wish
- Freedom 2: The freedom to **redistribute** and make copies so you can help your neighbour.
- Freedom 3: The freedom to **improve** the program, and release your improvements (and modified versions in general) to the public, so that the whole community benefits.

Well known examples of free software other than the Linux kernel are the GNU Compiler Collection (gcc), the Apache web server, the Emacs text editor, the X Window graphical-display system, the LibreOffice office suite, and the TeX/LaTeX typesetting systems.

In 1998, the above definition of free software was complemented by the “Open Source Definition” published by the Open Source Initiative (OSI), which is based on the Debian Free Software Guidelines. Despite some differences, both definitions are basically equivalent as they both stand for the same qualities and values. This is even more so since, as stated by Richard Stallman, “nearly all free software is open source and nearly all open source software is free”. Software distributed under these licenses nowadays goes under the name “free and open-source software” (FOSS). Other license types following the FOSS scheme are the Berkeley Software

Distribution (BSD), MIT, Apache, and Educational Community License (ECL) licenses. These licenses are also called permissive as the guarantee the maximum of rights to the licensee.

Free or open-source software may still be combined with the so-called copyleft practice, which while granting the right to distribute copies and modified versions of the software complements the license with the restriction that these rights must be preserved in all copies and modified versions derived from the software. Consequently, every user benefiting from the principles of free software is forced to also let others benefit from the same principles. The most well-known copyleft-type licenses are the GNU General Public License (GPL), GNU Lesser General Public License (LGPL), and Mozilla Public License (MPL), which enforce this principle with different strength. Copyleft-type licenses thus stand between the permissive license models discussed before and all types of proprietary software licenses.

The latter deviate from the above software models by restricting the comprehensive rights of the software user, e.g. by not providing the source code, by restricting the rights to modify or redistribute the software, or by requesting payment of a license fee. Clearly, proprietary licenses can come in many different variants and their regulations may even depend on specific negotiations between the software owner and the user, i.e. between the licensor and the licensee.

Of course, the choice of license model also has considerable impact on the way a particular software package can be combined with other software. This is sketched in Fig. 1, which shows the different license models together with their combinations with other software following the same or another license model in both upstream and downstream direction. Of course, permissive software licenses as the BSD and MIT licenses give the licensee full freedom to include licensed software into other software package and even to distribute the bundled software under a copyleft or proprietary license. In contrast, copylefted software can only be distributed under a copyleft license. For this reason, the copyleft license model and specifically the GPL have been criticised for being viral.

Finally, we mention dual or hybrid license schemes meaning that the same software is distributed under different licenses. As an example, a software can be distributed to non-commercial users under a permissive or copyleft license, whereas commercial users may only purchase it under a proprietary license. Since such license scheme clearly imply a higher effort on the side of the software owner, this model is not often adopted.

As expected, in the academic world many software packages are distributed under FOSS licenses since academia is based on and used to free exchange of ideas and open discussion of scientific results. In this respect, free dissemination of scientific software is just a continuation of a long-standing and proven principle increasing the status of a scientist in the race for a professorship just as many publications in high-ranked journals do. Yet, increasing demands on scientific work to generate results with a high impact on applications and, hence, usefulness to the society are also causing increasing endeavour to offer software originally written for scientific research to industrial customers. This trend calls for both different requirements on the quality of scientific software as discussed in the accompanying sections of this document as well as for reconsiderations of the license models employed so far. For this reason, scientific software owners are strongly encouraged to rethink their license strategies. As mentioned above, due to the organisational structure of university research with scientific progress being based on the work of many students and postdocs changing the license model in a changing world may be difficult. However, simply staying with a license model, which might no longer adequately protect the intellectual property rights of the software owners, may cause even more difficulties on the long run. Reflecting the license scheme used so far thus means a valuable investment.

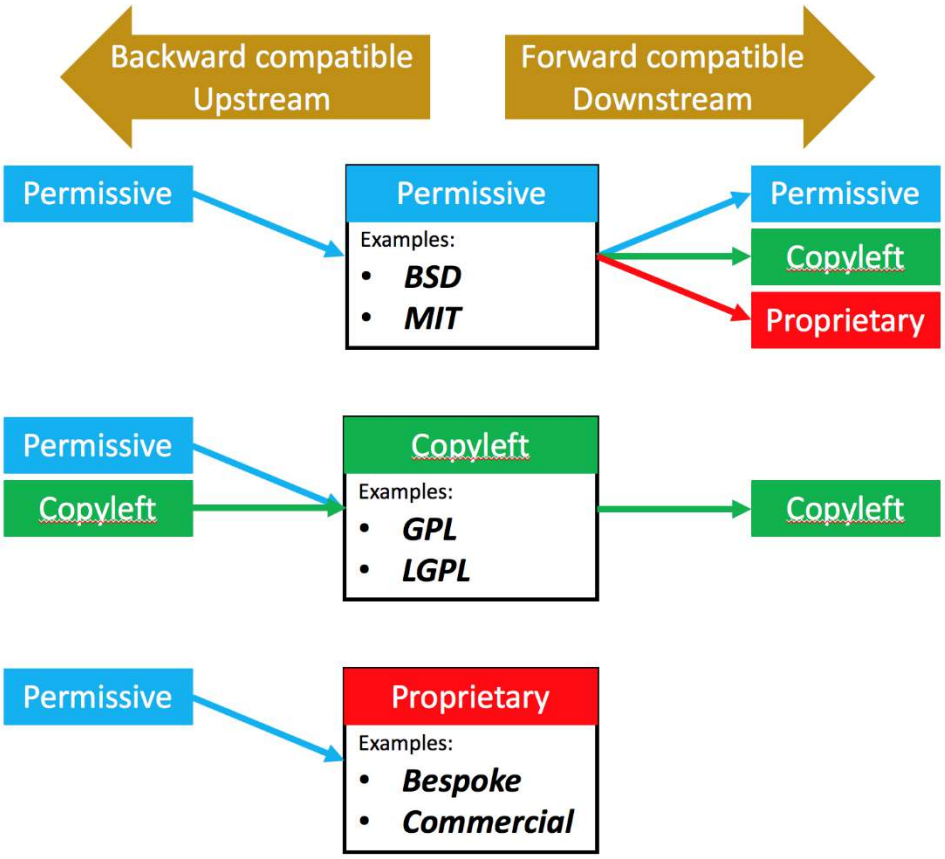

Figure 1. Schematic representation of license directionality. In general, permissively licensed code is forward compatible with any other license type. However, only permissive licenses, such as the BSD and MIT, can feed into other permissive licenses. Restrictive licenses like the GPL are backward compatible with themselves and permissive licenses, but must adopt the restrictive license from then on. Proprietary licenses can incorporate upstream permissively licensed code, but by definition are incompatible with any other downstream license. Further actions are not permitted without negotiating a separate license agreement with the rights owner. Adapted from: A. Morin, J. Urban, P. Sliz, *A Quick Guide to Software Licensing for the Scientist-Programmer*, PLoS Comp. Biol. **8**, e1002598 (2012).

2.6 Verification, testing, validation and robustness

Testing is an integral part of software development. In commercial software projects testing codes often have more lines than the software it tests. Testing must ensure that all parts of the software has been executed and function as planned. If the software is deployed on different platforms the test ensures that it behaves similar on different platforms and give consistent results when there are updates to operating systems or libraries. Testing also ensures consistent results between different releases of the software, thus, for simulation software in particular, testing frameworks are essential for obtaining trustworthy output of the simulations.

Testing is divided into unit tests and functional tests. Unit tests comprise testing the input and output of each function and module of the software. Such test should be part of the development of each new function to ensure that when all the pieces are put together the resulting software will work correctly. Functional tests comprise testing input and output of the entire software. They make sure that the individual pieces are working correctly together. There may be different hierarchies of functional tests, for instance, some running in serial on a daily basis, other running on all supported parallel architectures on a weekly basis, and large functional test suites running before making stable public releases. While academic software projects may find it overwhelming to implement professional level testing framework, at minimum large functional test suites testing all important aspects of the software should be performed before a public production release.

Formal verification of a materials modelling software must eventually be complemented by the validation of the initially defined underlying model. This step includes comparison of calculated results with experimental data or data created by other software. Such tests may after validation be included in the functional test suite.

2.7 Organization of the software development

If a group of researchers are contributing to the same software project, it is important to organise the development process. Often academic software projects branch into a number of different subprojects which diverge with time with no synergy between the subprojects. One important aspect of this is keeping track of different versions, the ability for working on local branches of the software and merging the changes into the main branch. This aspect can be handled by using a good version control system which is discussed in the next section.

Developing software often requires making choices, for instance, should the software be rewritten to increase the efficiency at the expense of clarity of the software? Which new developments are allowed into the main branch of the software? Should the software be restructured to allow for new developments? When libraries should be upgraded to the newest versions, etc. To decide on such questions, there need to be a decision structure. In many projects, it is the original developer that decides on these issues and this often work well. However, having more democratic bodies may increase the long-term scalability of the project and the quality of the decisions.

In publicly funded projects there may be a number of researchers working full time on software project with a common goal. In such projects, the efficiency of the work can often be increased by using methodologies from agile software development practices. A popular model is the scrum model. In this model, the development is done iteratively in short development cycles (sprints), typically 3-week cycles. Work is planned in all detail before each sprint and there is daily communication in the scrum team to ensure progress towards completion of all tasks in the sprint. At the end of the sprint the software is demoed and based on the progress and possible user feedback on the implementation the tasks for the next sprint is planned.

To work efficiently it is important to have access to efficient communication channels and shared software repositories. There exists a large number of software platforms which supports such developments. One of the most popular is GitHub which provides free access to open source projects.

2.8 Version Control

The development of complex software packages as they are routinely used in materials modelling requires a clear understanding of the changes made to the code and the respective changes of the results. Otherwise even small developer teams or single developers may soon lose track of the development process. In this regard, version control systems (VCS) as invented in the past decades constitute a big step forward. They go far beyond simply adding an increased version number to new code but add much more information including, e.g., the name of the developer and a time stamp. Overall, version control systems are characterized by five main tasks:

- Recording of all changes to the code. This allows to track in detail, which changes have been made by whom and when.
- Recovery of a previous state of the software. This allows to retract changes, which did not lead to a desired result or introduced errors.
- Archiving of all states of the software. This allows to restore previous versions of the software.
- Coordination of parallel editing of the same piece of code by several developers.

- Simultaneous work on different branches of the code.
- Merging of different branches of the code

From a software developer's perspective, working with version control systems is rather simple. The complete archive of the software is stored in a repository, from which it can be checked out, i.e. a copy of a particular status of the software is copied to the working directory of the developer. Once the developer has finished working on a piece of code the software is checked in, given a time step and user ID, and then synchronized with the software in the repository.

Depending on the physical location of the repository one distinguishes local, centralized, and distributed version control systems. While local VCS is used on single computers only, centralized and distributed VCS allow working with the system in distributed environments and differ only with respect to the storage of the repository. Examples of centralized VCS are the Source Code Control System (SCCS, invented 1972), the Revision Control System (RCS, 1982), the Concurrent Versions System (CVS, 1986), and Subversion (SVN, 2000). Examples of distributed VCS (DVCS) are ArX (2003), BitKeeper (2000), Codeville (2005), Darcs (2002), DCVS (2002), Fossil (2007), Git (2005), GNUarch (2001), GNU Bazaar (2005), Mercurial (2005), Monotone (2003), SVK (2003), and Veracity (2010). While all these systems are open source, there exist proprietary systems mainly in the field of centralized VCS. The large fraction of open source VCS is mainly because especially the open source community with its distributed software development processes has benefited a lot from such systems and still does.

Finally, one distinguishes different operation modes of version control systems. While some systems lock the respective part of the software in the repository once a developer checks out that part and allows other developers to access it only after the first developer has checked in his or her changes, other systems, provide a copy on check out and merge different copies in case several developers have worked on the same piece of code and had that checked in again. Of course, the former alternative may block code development to many users as it allows only single users to work on the code at a time, the latter alternative of course requires a much more complex version handling.

Modern distributed version control systems applying this copy-modify-merge approach are indispensable in nowadays distributed software development as they help simplifying the whole process and at the same time protect the main asset of any software developing group by archiving previous versions and keeping track of all changes.

2.9 Metadata

Providing information about a software together with data and results created by that software constitutes a much underrated and rarely considered but nevertheless very important aspect of any software including materials modelling software. In particular, it is a prerequisite of making software citable and the created results reproducible. Providing this information also goes far beyond mentioning, e.g., the applied exchange-correlation function or details of the basis set of an ab initio calculation. In contrast, to make results trackable and reproducible, detailed information about the code itself is needed.

This information usually goes under the name metadata. Metadata come together with the calculated results and should include the following items:

- Version/release number of the software, possibly also the release date
- Information about the software developer or the developing team
- Copyright and license information

- Information about the compiler used to create the respective executables and the corresponding compiler options
- Information about the hardware used to run the calculation
- Identity of the user running the calculation and the working directory
- Start and end time of the calculation
- A complete list of all input data used for the calculation

All this information is either available before the compilation process and may be coded into the software or else can be easily generated by modern programming languages.

Ideally, this information would also be given in all (intermediate or final output) files created by the software. This would serve the additional purpose of allowing for internal consistency of all data created by a calculation.

2.10 Documentation

In general, software documentation serves a large variety of purposes, which may be grouped along the following topics:

- The first step towards a comprehensive software documentation consists of the model description already dealt with above. It defines the model underlying the software, its purpose, the relation to other models and possibly the limitations of the model. This is the basic document of the whole project, from which everything else is derived.
- In a second step the main guidelines for the design of the software should be specified. This includes the code architecture, from which specifications of the algorithms to be used, the desired accuracy and again the limitations of the software can be derived. This issue has likewise been dealt with in a previous section.
- Technical documentation covers all questions connected with the implementation and may be covered by technical documentation as provided by doxygen or sphinx if appropriate comments are included in the software.
- Once the implementation is completed and the software is ready for application, the end user will require a user documentation or user's guide to be able to work with the software in an optimal and error-free way. We will come back to user documentation below.
- Finally, marketing of the modelling software requires yet another kind of documentation, which should come at a somewhat higher level and avoids detailed technical language to raise initial interest and to address decision makers.

While good user documentation is essential for the successful use and widespread dissemination of modelling software there obviously exists a large variety of ways to accomplish this goal. Clearly, a good user's guide should lay the foundation for a rather intuitive use of the software and allow for a very quick achievement of first calculated results as well as a rather steep learning curve. Enabling quick success stories will keep the user motivated and keen to reach "higher levels" of the software and to explore its full functionality.

Good software documentation includes the following key features:

- Description of the installation process and additional software to successfully install and/or run the software

  Since at the very beginning the end user knows only very little about the software and its structure, clear guidelines describing the installation process are mandatory for the acceptance of the software by the

user and a positive attitude. Note that this is the first contact the end user is making with the software and any problems, even little one, may demotivate him to move on. For this reason, the installation process should be very clearly outlined in the user documentation.

In addition, any other, auxiliary software like external libraries or other tools should be clearly mentioned and at best the end user should also be taught how to install that software in detail.

These guidelines apply both to the installation of executables and open-source software. However, in the latter case the user guide should also describe the availability and use of different compilers and related tools. In the optimal case, a Makefile should be provided and a simple "make" should generate the executables without further ado.

- Case studies and workable examples to teach users the basic functionality and to get them started

  Once the installation process has successfully finished, a good user documentation should guide the user through a set of examples ranging from rather simple cases to more complex ones, which allow the user to explore the functionality of the software and may even reach out to state-of-the-art scientific problems. Again, a precise description of all steps keeps the motivation of the user at a high level and allows him or her to create already at this early stage success stories.

  Optimally, the motivated end users will start to go beyond the examples discussed in the user's guide, play with input parameters to check their impact on the calculated results, and/or try out enhanced functionality of the software.

- Complete list of all input variables including default values and the respective physical units

  Clearly, any user's guide should include a complete list of all input parameters as well as of all input files necessary to run a calculation. Ideally, this list should also comprise a list of all intermediate and output files and their role in the process of a calculation. In addition, the size of such files should be mentioned in case it can become very large and thus critical for running an application.

  Of course, the list of all input parameters should mention the respective default values and ideally also describe how critical changes of a certain parameter are. Indeed, in the optimal case each input parameter might be classified according to the level an end user has already reached. Specifically, while the role of some input parameters is obvious already for beginners, modification of others might be recommended to expert users only.

  A much-overlooked issue regards the mention of the respective physical units for each of the input parameters (if applicable). E.g., in atomistic modelling software it should be clearly stated in the user's guide whether distances are expected to be given in nanometers, Angstroem, or Bohr radii.

- Screenshots of graphical user interfaces as well as of calculated graphical output

  In case a modelling software is coming with a graphical user interface display of screenshots can be very helpful and save a lot of detailed descriptions. In addition, display of calculated results allows the user to easily compare the results of his own calculation to those discussed in the user's guide and thereby check correct usage of the software.

- Background information about the underlying model(s)

  An extended user's guide will also include background information covering essential information about the model description, the basic architecture of the software, possible some of the algorithms used, and the limitations of the software. E.g., documentation of an ab initio code may comprise a chapter about the basic ideas of density functional theory as well as the most popular approximations nowadays coming with it. Such documentation going beyond describing the mere functionality of a code meets the needs of advanced users, which after having applied the software for some time may wish to learn more about the underlying theories and models.

2.11 Support

Long-term support is crucial for the acceptance of any materials modelling software by industrial end users. To pursue continued successful industrial research, users require long-term maintenance including updates and enhancements and a software owner, who can guarantee long-term help with the software in case of questions or problems. Yet, as outlined already above, this requirement is at variance with the paradigm of academic research with students and postdocs rather quickly moving on to new positions, research field, or to industry, such that their knowledge about the software is no longer available for direct access. This is where the industrial software owner come in since they can provide long-term stability in maintenance and support. Quite often, these industrial software owners work closely together with the academic groups, who developed and still develop the software.

3. Conclusions

This white paper addresses best practices of professional software development as guideline for academic software developers. It has been inspired by the observation,that in many cases decisions taken at an early stage have unforeseeable consequences for many years ahead but nevertheless are not well-thought-out. In addition, many academic software projects are characterized by uncontrolled growth starting from an initial small piece of code and thus lack systematic planning and management. In this context, the present white paper gives a framework for the development of materials modelling software to help especially academic software developers on their way towards successful applicability of their codes. While industrial application eventually needs commercial software tools, academic software development lays the foundation for a broader dissemination. By discussing a variety of topics including model descriptions and software architectures, implementation, programming languages and deployment, intellectual property and license considerations, verification, testing, validation, and robustness, organization of software development, metadata, user documentation, and support, the present white paper provides the information necessary for successful academic software development.

4. References

1. G. Wilson, D. A. Aruliah, C. T. Brown, N. P. C. Hong, M. Davis, R. T. G. Guy, S. H. D. Haddock, K. D. Huff, I. M. Mitchell, M. D. Plumbley, B. Waugh, E. P. White, and P. Wilson, *Best Practices for Scientific Computing*, PLoS Biol. **12**, e1001745 (2014).
2. A. Morin, J. Urban, P. D. Adams, I. Foster, A. Sali, D. Baker, and P. Sliz, *Shining Light into Black Boxes*, Science **336**, 159 (2012).
3. A. Morin, J. Urban, P. Sliz, *A Quick Guide to Software Licensing for the Scientist-Programmer*, PLoS

Comp. Biol. **8**, e1002598 (2012).

4. M. Christensen, V. Eyert, A. France-Lanord, C. Freeman, B. Leblanc, A. Mavromaras, S. J. Mumby, D. Reith, D. Rigby, X. Rozanska, H. Schweiger, T.-R. Shan, P. Ungerer, R. Windiks, W. Wolf, M. Yiannourakou, and E. Wimmer, *Software Platforms for Electronic/Atomistic/Mesoscopic Modeling: Status and Perspectives*, Integr. Mater. Manuf. Innov. **6**, 92 (2017).

5. Links

1. Software Development Best Practices for Computational Chemistry: https://github.com/choderalab/software-development
2. Quality of Materials Modelling Software: EMMC guidance on quality assurance for academic materials modelling software engineering, https://zenodo.org/record/192025
3. Software Sustainability Institute: http://www.software.ac.uk/, http://www.software.ac.uk/resources/top-tips
4. Software Engineering Support Centre (SESC): http://www.softeng-support.ac.uk
5. SESC Workshop 2014: http://www.softeng-support.ac.uk/news/2014/11/06/Report-SESC Workshop.html
6. Originally Canadian initiative called Software Carpentry: http://software-carpentry.org/
7. CECAM Workshops; e.g. 2014: http://www.cecam.org/workshop-0-198.html; 2015: http://www.cecam.org/workshop-1214.html
8. PSI-K workshops: http://www.psi-k.org/workshops.shtml
9. CINECA training workshops material: http://se4science.org/workshops/se4hpcs15/resources.htm
10. CCPForge – coding standard: http://ccpforge.cse.rl.ac.uk/gf/project/ccpforge/docman/Developer Documentation/
11. ABINIT software 'code of conduct' best practices guide for contributors: https://wiki.abinit.org/doku.php?id=developers:coding_rules
12. HPC Consortium Autumn Academy: http://www2.warwick.ac.uk/fac/cross_fac/hpc-sc/courses/academy/
13. UK HPC service: http://www.archer.ac.uk/documentation/best-practice-guide/
14. CCP5: http://www.ccp5.ac.uk/ (does not have a listing but from 2014 started to give two courses in SoftEng: 1. Fortran95 - day 1 and 2 for beginners, day 3 for advanced users 2. 'Hack days' for some of the codes displayed at http://www.ccp5.ac.uk/software/)

6. Appendix: Online resources to development of scientific software

In the white paper we discuss many aspects on proper software development methodologies. To learn more about these topics there are many online resources. In particular we can recommend the online academies https://www.udemy.com and https://www.coursera.org which have a number of courses related to the topics covered in the white paper. Forinstance to learn python we can highly recommend the course:
https://www.coursera.org/course/interactivepython1

Below you can find a selection of links to other web resources on selected topics covered in the white paper.

https://www.atlassian.com/agile:
Detailed description of agile Software development including scrum and kanban methods with tutorials and background information.

https://www.guru99.com/unit-testing-guide.html
This is a guide to test driven development with links to different unit testing software libraries

https://git-scm.com/docs/gittutorial
This is a tutorial on how to use git, which is the mostly used version control system.

https://www.amazon.com/Introduction-Performance-Computing-Scientists-Computational/dp/143981192X
This is a good book on scientific software development in general

https://bisqwit.iki.fi/story/howto/openmp/
http://mpitutorial.com/tutorials/
These are good tutorials on parallel computing using OpenMP and MPI.

Section 6 has been added on 14.09.2018

Authors	Volker Eyert (Materials Design S.A.R.L) Kurt Stokbro (Synopsys – former QuantumWise A/S)

Lead beneficiary	**MDS**
Contributing beneficiaries	MDS, Synopsys (QW)

EC-Grant Agreement	723867
Project acronym	EMMC-CSA
Project title	European Materials Modelling Council - Network to capitalize on strong European position in materials modelling and to allow industry to reap the benefits
Instrument	CSA
Programme	HORIZON 2020
Client	European Commission
Start date of project	01 September 2016
Duration	36 months

Consortium		
TU WIEN	Technische Universität Wien	Austria
FRAUNHOFER	Fraunhofer Gesellschaft	Germany
GCL	Goldbeck Consulting Limited	United Kingdom
POLITO	Politecnico di Torino	Italy
UU	Uppsala Universitet	Sweden
DOW	Dow Benelux B.V.	Netherlands
EPFL	Ecole Polytechnique Federale de Lausanne	Switzerland
DPI	Dutch Polymer Institute	Netherlands
SINTEF	SINTEF AS	Norway
ACCESS e.V.	ACCESS e.V.	Germany
HZG	Helmholtz-Zentrum Geesthacht Zentrum für Material- und Küstenforschung GMBH	Germany
MDS	Materials Design S.A.R.L	France
Synopsys / QW	Synopsys (former QuantumWise A/S)	Denmark
GRANTA	Granta Design LTD	United Kingdom
UOY	University of York	United Kingdom

Coordinator – Administrative information	
Project coordinator name	Nadja ADAMOVIC
Project coordinator organization name	TU WIEN
Address	TU WIEN I E366 ISAS I Gusshausstr. 27-29 I 1040 Vienna I Austria
Phone Numbers	+43 (0)699-1-923-4300
Email	nadja.adamovic@tuwien.ac.at
Project web-sites & other access points	https://emmc.info/