On Holden's Seven Guidelines for Scientific Computing and Development of Open-Source Community Software

Knut-Andreas Lie

SINTEF Digital, Mathematics and Cybernetics, Oslo, Norway
Also: Department of Mathematical Sciences, NTNU, Trondheim.

Abstract

Two decades ago, Helge Holden proposed seven guidelines to improve the way new achievements and results in scientific computing were presented, evaluated, and compared in contemporary scientific literature. In this essay, written as a tribute to Helge on his 60th birthday, I revisit the guidelines and point out why they are still valid today seen from my perspective, working as a contract researcher at the interface between mathematics and applications in industry.

Developing new computational methods usually involves a lot of experimental programming. Over the past decade, my research group has developed an open-source community code that today has hundreds of users worldwide. I discuss some considerations that have gone into this development and present a few lessons learned. Moreover, based on this experience, as well as from development of professional software for our clients, I present advice on how you can be more productive in your experimental programming and increase the impact of your scientific results.

Science is what we understand well enough to explain to a computer. Art is everything else we do.
– Donald Knuth, Foreword to the book A=B (1996)

1. Introduction

Throughout the 1980 and 90s, numerical computation established itself as a third way to science in complement to the classical duality of experiments and theoretical models. These were vigorous times for scientific computing. Major advancements in numerical discretization methods and iterative linear solvers, combined with a continuous and rapid growth in computing power, enabled highly resolved numerical simulations to be adopted in many new scientific disciplines. Growing maturity of third-generation programming languages like FORTRAN, C, and C++ enabled scientists to write simulation programs of unprecedented complexity, and color monitors with powerful computer graphics spawned the development of advanced and powerful visualization techniques that increased our ability to visually interpret and understand the results of advanced simulations. During the same period, LaTeX became widely adopted among scientists, which together with the emergence of the world wide web in the early 1990s, dramatically changed the way science was communicated. All of a sudden, it was quite simple to include both vector and raster graphics in your scientific papers, make very impressive presentations, and quickly share these with your colleagues around the world. Altogether, this presented unparalleled opportunities for members of the relatively young scientific-computing community, which grew rapidly in numbers.

Email address: knut-andreas.lie@sintef.no (Knut-Andreas Lie)

Being part of a revolution, it is easy to become too eager in your quest for progress and forget or disregard wisdom and well-established practices developed by previous generations. Helge Holden was among those who saw this, and during my first years as his student, he wrote a paper [11] in an attempt to influence the way computer simulations were performed and presented to the scientific community: "*[. . .] some words of warning may be appropriate at this moment as we are easily becoming victims of ever more impressive presentations. It is easy both as spectators and performers of the art of scientific computing to forget the critical eye of the scholar and the rigorous requirements of modern science. It is becoming all too common to present results of simulations lacking sufficient documentation to allow the repetition or reproduction of the results.*" To amend what he perceived as a serious deficiency, Helge proposed seven guidelines[1]:

1. Your results should always be reproducible.
2. Test the stability of your method with respect to variation of parameters.
3. Compare your method to other methods.
4. Report cases where your method fails.
5. When possible, compare the computer simulations to real experiments.
6. Establish standard test cases in your field.
7. Make your own code available to your colleagues.

Some of the observations presented in the manuscript were quite controversial at that time and the paper was never accepted for publication. However, being controversial does not mean you are wrong, and in this essay, written on the occasion of Helge's 60th birthday, I try to give him the credit he deserves by providing a complementary discussion of the ideas put forth in his original paper. In particular, I will try to relate part of the discussion to research activity over the past two decades, involving joint supervision of a number of master and doctoral students. A main achievement of this research is the development of MRST, a comprehensive toolbox for rapid prototyping of new computational methods for subsurface flow modelling (`http://www.sintef.no/mrst`), and OPM, an open innovation platform for industry-grade simulations (`http://opm-project.org`). In the last part of the essay, I discuss some of the considerations that have gone into the development of these softwares and summarize some lessons learned. Like many other scientists who spend a major portion of their time writing software, members of my research team are self-taught. However, we have been exposed to best practices for professional software development as part of our contract research. I summarize what I have observed to be good practices if you want to be a productive developer of computational methods, write reliable codes, and increase the impact of your work.

2. Reproducible research for scientific computing

Replication of experiments is usually considered the golden standard in science and should not be confused with the principle of reproducibility, which Helge suggested as a necessity if scientific computing was to be considered as a serious science. I like the explanation of the difference between the two concepts given by the editor of the *Biostatistics* journal [28]:

> *The replication of scientific findings using independent investigators, methods, data, equipment, and protocols has long been, and will continue to be, the standard by which scientific claims are evaluated. However, in many fields of study there are examples of scientific investigations that cannot be fully replicated because of a lack of time or resources. In such a situation, there is a need for a minimum standard that can fill the void between full replication and nothing. One candidate for this minimum standard is*

[1]The paper actually started out as "Ten commandments on scientific computing", but was toned down during the process towards potential publication.

reproducible research, which requires that data sets and computer code be made available to others for verifying published results and conducting alternative analyses.

Although offered in a different scientific field, it applies equally well to scientific computing.

Reproducible research in scientific computing is usually attributed to Jon Claerbout [9, 10]. In 1990, he set a goal of reproducibility for his research group at Stanford University. The goal was that not only should anybody be able to recompute the group's research results on any computer, but they should also be able to reproduce documents the group had published to present this research. At that time, this was a quite ambitious undertaking. Today, it somewhat simpler if you use notebook facilities in a scripted language. One good example is the Jupyter Notebook for Python (`http://jupyter.org/`), which enables you to mix computer code with rich text, mathematical formulas, plots and rich media. Similar functionality was recently introduced through so-called Live Scripts in MATLAB, which in many aspects supersedes the useful, albeit less powerful `publish` function. Likewise, use of virtual machines or container systems like Docker can be good ways to disseminate research on computational methods. A virtual machine emulates a computer system and enables others to rerun the software and data used by the authors of a scientific paper, without having to download and set up the necessary software libraries used for the simulations. Software containers are more lightweight systems that only bundle the libraries and settings necessary to make your code run on any system. Notice, however, that virtual machines and container systems only ensure a very limited type of reproducibility since all you can do is to rerun numerical experiments and change input parameters. Without access to source code, you really cannot dig into the code to understand how it works and verify that it indeed implements what is written in the scientific paper. Access to complete source code, as suggested in Holden's last guideline, is therefore an important ingredient to reproducibility and the higher goal of replicability. We will come back to this later.

During the last two decades, the idea of reproducible research in computational science has picked up significant momentum and has been voiced by a large group of well-respected and influential computational scientists, see e.g., [19]. However, if one chooses to look critically at scientific publishing, it has largely remained in the same sorry state as observed by Helge Holden in 1994 [11]. Here, I have chosen to include two quotations by other scientists. The first is from 1995 by Buckheit and Donoho [6]:

An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment and the complete set of instructions which generated the figures.

One decade later, the unfortunate situation was expressed even more pointedly by LeVeque [17]:

Within the world of science, computation is now rightly seen as a third vertex of a triangle complementing experiment and theory. However, as it is now often practiced, one can make a good case that computing is the last refuge of the scientific scoundrel [...] Where else in science can one get away with publishing observations that are claimed to prove a theory or illustrate the success of a technique without having to give a careful description of the methods used, in sufficient detail that others can attempt to repeat the experiment? [...] Scientific and mathematical journals are filled with pretty pictures these days of computational experiments that the reader has no hope of repeating. Even brilliant and well intentioned computational scientists often do a poor job of presenting their work in a reproducible manner. The methods are often very vaguely defined, and even if they are carefully defined, they would normally have to be implemented from scratch by the reader in order to test them.

Even now, ten years after, much of the same observations hold true. As referee and editor, I have never to date been offered the possibility to look at any authors' source code or use their software to rerun and verify numerical experiments reported in the paper. There are journals that require software to be published alongside papers, but these are few. Fortunately, there are indications that for many scientific journals it is more a question of when and how the requirement for reproducibility will be mandated. An increasing number of journals are offering authors the possibility to upload their computer code and input data, so that others can download and experiment with these on their own computer. Nevertheless, even though readers tend to access scientific publications electronically, the standard is still a static text document in most journals, and review of software and interactive, notebook-type presentation of numerical experiments have yet to permeate scholarly publishing. In the future, one can only hope that the growing demand for open-access publishing, and general competition within scientific publishing will induce a much needed innovation toward more interactive formats that better support the principle of reproducibility.

3. From proof-of-concept towards widespread adoption

Many researchers develop new methods to satisfy their own curiosity, or because it is great fun, but I still believe that most of us do it because we want to make something useful and have a lasting impact on the scientific community and/or society. In this, academia and contract research organizations, like the one I have worked in for the past two decades, are not very different. The difference lies more in how we measure impact and success. In academia, the apparent success criteria are theories and scientific papers, whereas impact can be measured in terms of citations, invitations to conferences, etc., which are superficial indications of the more vague concept *scientific quality*. Publications and citations are also important in contract research organizations, but creating values for your clients and acquiring new research contracts generally rank higher.

In this section, I discuss how Holden's Guidelines 2 to 6 can be used to help you to success and ensure that the methods you develop have an impact, regardless of whether you work in academia or closer to industry and commerce. My focus will primarily be on the experimental process leading up to new computer codes whose aim are verify that new computational methods work as claimed, verify and validate physical models, and/or provide proof-of-concepts for new computational workflows. High scientific quality in this process is utterly important if the computational methods developed should later enter large-scale community codes used for scientific discoveries in other parts or science, or professional production codes developed to support (critical) decisions in the private and public sector.

3.1. Verification and comparison with other methods

To justify the development of a new computational method and entice the interest of others, two approaches are common to use, possibly in combination:

- You can either demonstrate that your method solves a new problem not yet solved by other methods, or that is solves a class of problems that so far has only been partially solved; or
- You can demonstrate that your method solves a known class of problems better than existing methods, e.g., by comparing with these methods and/or pointing out deficiencies that your method does not have (**Guideline #3**).

Providing honest and fair comparisons with existing methods is more difficult than it may sound. In well-established applications of various forms of fluid or solid dynamics, there is usually a plenitude of computational methods to compare with. It is therefore tempting to pick a standard textbook

method, which is simple to implement and whose limitations and deficiencies are widely accepted. While such comparisons can be informative to a certain point, they do not carry the same value as comparisons with a state-of-the-art method. The best is, of course, to collaborate directly with the developers of the method you wish to compare with, since they have intimate knowledge of the inner workings of the method and know how to tune (undocumented) features to insure optimal performance. Such collaborations are sometimes out of the question if commercial interests are involved or the goal of your research is to defeat the other method. Unless implemented as open source, it is therefore often difficult to get your hands on a functional and efficient implementation of the methods you should compare with, especially if they are from recent literature; I will get back to this later when discussing **Guideline #7** in Section 4. Your only option is then to implement the methods yourself. This is in many cases a significant undertaking and you easily end up spending a considerable time reinventing or reverse-engineering crucial algorithmic features that are not well documented for the reasons discussed in the previous section. Let me take one of my own papers [14], which compared and contrasted various upscaling and multiscale methods for simulating two-phase flow in porous media, as an example. Writing this paper required almost a half-year, concentrated effort to bring our implementation of methods not developed by ourselves to a maturity level where we could trust them to provide fair and unbiased comparisons with our own multiscale method. This, despite the fact that the first author is an unusually smart and capable programmer. In our case, this exercise proved to be worthwhile since it gave us a lot of insight that could be used in subsequent research. Around that period, we had what I would describe more as a friendly competition than a direct cooperation with the developers of one of the contending multiscale method. Whenever they published a refined version of their multiscale method, we tried to come up with test cases that showed deficiencies in their method and rendered ours in a good light, and vice versa. My impression is that the overall development of multiscale methods benefited from such a healthy competition, and I would generally recommend it as a means to bring your research rapidly forward.

This brings me to the choice of the cases you use to verify, validate, and assess the performance of your method. If your method solves partial or ordinary differential equations, the first thing to do, is to verify that it is able to reproduce analytical solutions on simplified problems. If possible, these solutions should verify correct behavior of as many as possible of the terms entering your model equations. Secondly, you should verify that your method converges and/or scales as anticipated. Once this is done, you should look at the robustness and versatility of your method. Slightly paraphrasing **Guideline #2**, this means that you should stress test your method with respect to assumptions and variation of parameters so that you know how robust the method is, what the limitations are, and so on. Looking at **Guideline #4**, the results of your tests should be reported irregardless of whether they are positive or not. This will, in the long run, give your more scientific credibility. Looking at it from a purely selfish perspective, it is better that you discover and disseminate weaknesses in your own method, rather than having others pointing it out in subsequent publications.

Unfortunately, in performing extensive and objective testing of your method, you have several mechanisms working against you. First of all, humans are inherently lazy, and if we can get away with only investigating and presenting a restricted range of numerical tests, we will almost inevitably do so. In particular, the *publish or perish* syndrome tends to leave us all with little time to conduct thorough tests of new methods. Once we have found a small series of cases showing the superiority of our new method, we seek to publish. In doing so, it is very easy to bring the competition to your own home ground and design biased test cases focusing on the aspects for which your method is particularly good. Likewise, when running large series of test cases, it is very easy – despite our best intentions – to subconsciously only pick those instances that show our method in a good light.

This is a known fallacy in experimental science, but best practices that address this are, to the best of my knowledge, seldom discussed when teaching courses in computational science.

The mechanisms discussed above are strengthened by contemporary publishing culture, which tends to focus on success rather than failure and seldom allows you to publish research on methods that fail to work. This is a pity, since it may often be more interesting for others to learn about well-conceived approaches, or approaches that suggest themselves naturally based on current knowledge, that turn out to not work as anticipated. Publishing negative results will not only prevent others from wasting precious time chasing dead ends, but if you also provide an explanation why your method did not work, others may gain significant new insight that can help them to come up with alternative solutions.

3.2. Benchmark problems and standard test cases

Holden's **Guideline #6** rightfully suggests that you should establish test cases that can work as a standard in your field. Setting up good test cases is not a simple undertaking, but usually has great value for the scientific community, provided that the test case is well designed. (It is also smart from a bibliometric point of view, since papers introducing standard test cases typically generate a large number of citations.) Test cases come in many variants, from standard benchmark problems that are run to measure the computational performance of processors, (iterative) linear solvers, nonlinear solvers, etc., to more open-ended setups, where the challenge is to compute as accurate or optimal solution as possible. Benchmarks can also pose problems that do not yet have any known or well-established solution. Test cases that contain observed behavior of a physical system (as emphasized in **Guideline #5**) are particularly useful, since the ultimate goal of many computer simulations is to predict the results of actual physical processes.

For most researchers, the use of standard test cases is a simple, yet effective way to compare different computational methods. Given that the test case is utilized for the same purpose, and results are reported in a consistent manner, it is in principle easy to compare results reported by different researchers. To a certain extent, this alleviates the need to compare with other state-of-the-art methods as long as these have been validated on the same test cases. Oftentimes, researchers will also modify standard test cases and (ab)use them for different purposes than what they were designed for. This can be quite useful, since the research community may have developed a familiarity with the original setup that enables your peers to quickly interpret and assess results also on a modified setup, as long the original case is not obfuscated beyond recognition.

In his paper [11], Helge Holden pointed out that test cases should be set up based on a consensus process in the scientific community, not as a static decision, but in accordance to scientific progress and development of computers. Carefully designed test problems have the power to drive research in certain directions, and can be a useful way to align activity in a research community with certain business interests and societal needs. The danger is, of course, that test cases may have a too strong influence on the focus of a scientific community. This was also noted by Helge, who pointed out the danger that researchers may be tempted to tailor-make their methods to benchmark tests. I have seen this tendency in my own field of research: computational methods for subsurface multiphase flow. Simulation models of real petroleum reservoirs take a long time to make and contain a lot of information about company assets. These models are hence considered business critical and are seldom shared openly with the research community. For many researchers, the most obvious alternative when looking for realistic data has been the (in)famous SPE 10 test case [8]. This synthetic model was originally posed as a hard test case for numerical homogenization methods (referred to as upscaling methods in petroleum engineering) and has an exaggerated variation in petrophysical parameters, but a much simpler grid geometry and fluid model than what is common in models of real assets. This is often overlooked, and there are many examples of over-fitted

methods that show excellent performance on SPE 10 and similar cases, but fail to provide solutions on problems of practical value. Once such a model comes into widespread use, it has a self-reinforcing effect. Even though you realize that it not necessarily is a representative test case, you *have* to use it because this is what your peers expect you to do.

When posing a test case, there are many practical issues to consider, and these will obviously vary from one field of science to another. As a minimum requirement, the purpose of the test case should be clearly specified along with the set of assumptions that restrict the problem and/or leave room for the user to make his/her own choices. These should be stated in a document that can be referred to by a persistent and unique identifier such as a Digital Object Identifier (DOI) or alike. In many cases it is natural to identify output parameters to be measured, or offer a set of reference and/or user-generated solutions or output parameters for comparison. If the test case involves input (or output) data, one should make sure that these are published along with clear specification of legal rights, preferably under a permissive license that enables users to freely interact with the data. For test cases involving several subproblems, it is also important that each subproblem is clearly labeled so that users later can refer to it in a unique manner.

Within subsurface flow modelling, a common approach to compare methods or modelling approaches is to invite participants to make their best attempt to reproduce a certain physical scenario. After a certain period, results are collected, compared and contrasted, and reported in a publication. (SPE 10 mentioned above was one such benchmark). In several cases, the data have only been made available to registered participants, and after the study has finished, the data are no longer available. This is a short-sighted practice that should be avoided. Not only should the data offered as part of the original setup remain available, but results reported by different participants should be made openly available so that researchers later can use them to make independent comparisons. The same goes for any truth model involved in the study.

Last, but not least, let me point out that standard test cases can easily be created somewhat unintentionally. Numerical examples reported in the first papers discussing a certain class of methods have a tendency of later becoming de facto test cases. This means that rather than reporting somewhat haphazardly generated examples highlighting salient features of your method in a graceful view, you should always consider to what extent these examples can be reused by others and put extra effort into designing test cases that can be used to stress-test not only your method, but a wide class of methods designed for similar purpose.

3.3. Evolution of computational methods viewed using Gartner's Hype Cycles

Over the years, I have watched the evolution of several computational technologies; some quite close, like GPU computing and multiscale methods, and others more from afar; some have become widely adopted, whereas others have dwindled into obscurity. If we disregard the rare and ingenious ideas that get widely adopted in almost no time, evolution of computational methods follow a very similar pattern, shown in Figure 1. This curve, called Gartner's Hype Cycles, was developed to interpret technology hypes and enable industries to assess their risk when investing in emerging technologies. Let us see how it can be applied to describe research and dissemination of new methods in scientific computing.

In part, this curve is the result of a divide in focus among most mathematicians and researchers in the applied sciences. Mathematicians tend to develop advanced theories and rather sophisticated methods for idealized problems, whereas researchers in applied sciences and industry tend to work on problems that are outside the bounds of contemporary theories using somewhat less sophisticated methods. Let me exemplify: Whereas a lot of theory for nonlinear PDEs is developed in unbounded domains, models for real physical processes are usually posed on bounded domains and are to a large degree determined by their boundary conditions. Likewise, mathematicians tend express their

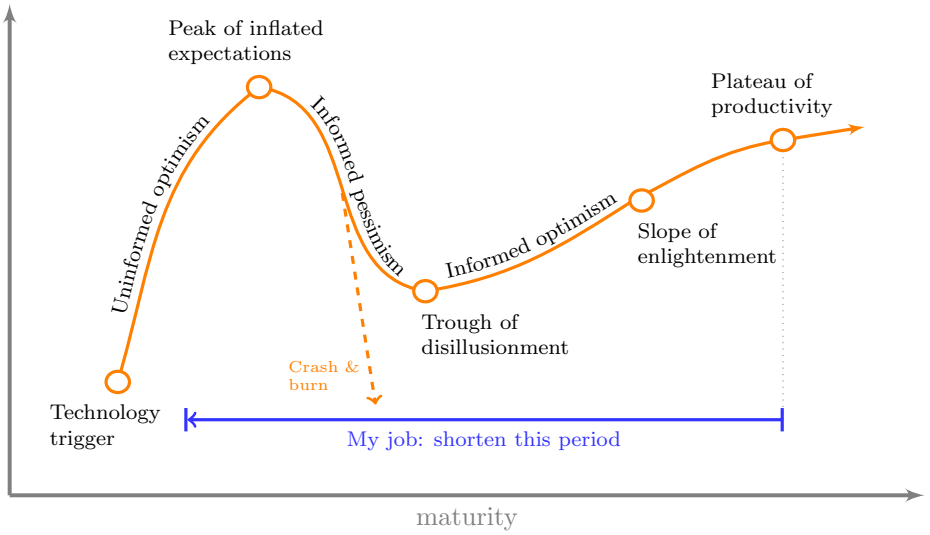


Figure 1: Development of computational methods following the Gartner Hype Cycle. An important aspect of my job as contract researcher is to identify promising technologies and make the transition from uniformed optimism to the plateau of productivity as short as possible.

results using functional spaces and study PDEs of a certain type (hyperbolic, parabolic, elliptic), whereas models of many physical processes often exhibit mixed characteristics or involve a mixture of differential equations, empirical laws, and tabulated relationships for which it is often not clear what the appropriate functional spaces are.

Early reports on new methods arising from mathematics or computer science have a certain tendency of being overly optimistic with regard to generality and application potential. There are at least two reasons for this. First of all, you need to have a certain missionary streak, or alternatively be a salesman to entice the interest of others. Secondly, new methods are generally not as well tested as one might hope because of the mechanisms discussed in the previous section, or because testing on state-of-the-art descriptions of physical phenomena is way too complicated to be contemplated within a realistic time-frame by anybody without expert application knowledge. However, once a new idea picks up momentum, it is bound to be tested for a wide variety of models and parameters as researchers try to adapt the ideas to their own problems. This will generate many success stories, but also a lot of failures when the new method is applied outside its scope or range of validity. Sooner or later, the initial interest starts to wane when it becomes clear that the method is not as suitable as initially suggested or will require significant more research to introduce sufficient improvements or do the necessary adaptions. Continuing to research a method down the *slope of informed pessimism* and through the *trough of disillusionment* can be a very frustrating exercise. However, as pointed out above, a lot of new insight can often be gained through failures, and if you manage to grit your teeth and stay focused, you may eventually be able to start climbing the *slope of enlightenment* and push your method up to the *plateau of productivity*, where widespread adoption takes place.

How do you accelerate the time to informed optimism? Here, good test problems play an essential part. These should not only be the kind that stress-test your method within its current scope, but it is also important to have a succession of increasingly challenging test cases that can be used as milestones to continuously drive your methods towards a wider scope. Equally important, you need to have flexible prototyping tools to support the necessary experimental programming. I will get back to this in the next section.

A personal story. A few years after the turn of the century, my research group started to develop multiscale methods for reservoir simulation. From my perspective, this process has followed the Gartner Hype Cycle. Helge has not been directly involved as a publishing author in this research, but has acted as co-supervisor for many master and doctoral students.

I had first encountered the idea in 1997, when Helge and I met Tom Hou – who that year published the first paper on multiscale methods [12] – during a sabbatical at the Mittag–Leffler Institute in Stockholm. I remember that Helge asked me whether we should start working on this, but I found the idea to be somewhat contrived and doubted it would find widespread application. Quite ironic, since I later have spent more than a decade developing multiscale methods towards industry adoption. For completeness, let me briefly describe the multiscale idea. Assume that you have a variable-coefficient Poisson equation, in which the coefficient exhibits variations over many orders of magnitude and that the spatial variation of the coefficient takes place over a broad range of length scales with no clear scale separation. (Poisson's equation arises in porous media if we combine mass conservation $\nabla \cdot \vec{v} = q$ for an incompressible fluid with Darcy's law $\vec{v} = -\mathbf{K}\nabla p$.) Discretizing the equation, e.g., with a standard first-order finite-volume method, we get the following

$$-\nabla \cdot \big(\mathbf{K}\nabla p\big) = q \qquad \longrightarrow \qquad \mathbf{A}\mathbf{p} = \mathbf{q}. \tag{1}$$

Here, p is the pressure (or more generally the potential) of a single-phase fluid, $\mathbf{K}$ is the permeability of the rock (i.e., the ability to transmit fluids), and q is a volumetric source term. The key idea of multiscale methods is to partition the fine grid used to discretize the Poisson equation into a coarse grid, to which we associate a vector of unknowns $\mathbf{p}_c$. For each coarse grid block, we define and solve a variable-coefficient Poisson equation with zero right-hand side on the fine grid. The fine-scale solution is restricted so that the resulting solution is one at the center of the block and zero at the centers of all the other blocks. By specifying appropriate boundary conditions, the local solution, which we will refer to as a multiscale basis function, will have compact support restricted inside the nearest neighbors of our block. Collecting these basis functions as columns in a matrix $\mathbf{P}$, we have derived a prolongation operator that maps unknowns on the coarse grid to unknowns on the fine grid. If we also define a restriction operator $\mathbf{R}$ that sums entities defined over all cells inside each block, we have a systematic method for forming a reduced flow problem on the coarse grid that is consistent with the differential operator $\nabla \cdot \mathbf{K}\nabla$ on the fine grid,

$$\mathbf{RAP}\mathbf{p}_c = \mathbf{R}\mathbf{q} \qquad \longrightarrow \qquad \mathbf{A}_{ms}\mathbf{p}_c = \mathbf{q}_c. \tag{2}$$

What I have described is an algebraic formulation of the multiscale finite-volume method [13]. When I started working on these methods, there had already been made some bold claims that multiscale methods would give three orders of magnitude computational speedup. Over the succeeding years, we used an alternative mixed formulation [7] to extend multiscale methods to the complex grid formats used in industry. Grids of real assets have unstructured topology and polyhedral cell geometries with bilinear non-matching faces and up to three orders-of-magnitude aspect ratios. By and large, we succeeded in adapting the method to these grids and developing automated coarsening methods that robustly could handle the many intricate special cases arising for such grids [3, 4]. However, our somewhat cyclopic path of development reached its peak of inflated expectations when we tried to extend the method from slightly compressible flow to models with the full complexity seen in industry-standard applications. It turned out that the mixed formulation of complex flow models was not as robust as existing literature had seemed to indicate. After several futile attempts, we abandoned the multiscale mixed finite-element method and let it slide down the slope of informed pessimism towards obscurity.

During the same period, the development of the multiscale finite-volume method had followed an equally cyclopic path towards realistic flow physics. Useful developments included a fully algebraic

formulation and reformulation of the method as an iterative method. In the iterative formulation, the multiscale matrix $\mathbf{A}_{ms}$ is used as a global preconditioner to eliminate low-frequency error components in combination with a standard local smoother that effectively eliminates high-frequency error components. This makes the method quite similar to a multigrid method, but has the advantage of exposing parallelism and enabling users to stop the iteration at any tolerance and still obtain mass-conservative fluxes. On the other hand, the development had been quite unsuccessful in extending the method to unstructured grids and the special grid formats uses in the petroleum industry. Being able to handle such grids is a prerequisite for industrial adoption. On an offhand chance, I suggested to Olav Møyner, one of Helge and my students, that he could write his master thesis on this problem. This turned out to be a stroke of luck. Over the past 4–5 years, we have managed to develop a new and very robust formulation for fully unstructured and stratigraphic grids [24, 25], by combining original ideas from Olav with insight obtained working on the mixed method. Our new method has been implemented by Schlumberger and is a cornerstone of what today is considered as next-generation technology for reservoir simulation. The interested reader can find a more thorough discussion of this method, and the various technical developments that lead up to it over the past decade, in Lie et al. [22].

The research described above has had two unintended side effects. First of all, it has inevitably given members of my research group a lot of insight into multiphase flow in porous rocks and induced a shift in our research focus from mathematics towards reservoir engineering. More important, it has lead to the development of an open-source community code for rapid prototyping of new computational methods for subsurface flow simulation that currently is used by many hundred researchers, students, and engineers all over the world. More details about this software are given in Section 5, whereas a summary of the lessons learned during its development will be presented in the next section.

4. Development of open-source community code

In his seventh and last guideline, Helge recommended that academic computer codes used in simple numerical experiments should be made available to others. This is probably the guideline I have personally taken most to heart. Today, the arguments for **Guideline # 7** probably seem overly cautious. In particular, I tend to differ on Helge's observation that public release should be restricted to codes of little commercial value. As you will see later in this section, it is possible to publicly release codes with significant commercial value, you only have to use a different business model than licensing if you want to use the code to earn money. However, at that time, the idea of giving away your code for free was, as far as I understand from Helge, considered by many to be an almost ridiculous idea, which explains why the accompanying arguments were toned down during the unsuccessful review process. Almost twenty years later, LeVeque [18] presented very compelling arguments for why you *should* release your code publicly. The short and humorous article describes an alternative universe in which mathematical proofs are not required and presents ten reasons why papers should not contain proofs. Through this simple thought experiment, LeVeque shows how absurd it is that computational sciences does not live by the same standard as mathematics. If his arguments do not convince you, I do not have much new to add, except to say that it has worked marvels for my research group. If you are already convinced, I urge you to set a good example in your own research, and request others to follow your lead when you act as supervisor, as reviewer or editor for scientific journals and conferences, or as evaluator on grant proposals.

My aim herein is to explain how you can bring methods from the peak of inflated expectations and onto the slope of enlightenment. My focus will thus be on somewhat larger codes than what Helge originally suggested to release. Based on our experience in developing and maintaining what

has become two community codes, I can make a few simple observations of why this may be highly useful for a research group:

- Publishing and maintaining an open-source code is an efficient (albeit not always simple) way of coordinating activities within a research group or among cooperating scientists.
- Releasing your code to the public does not jeopardize your intellectual property rights as long as you are careful in your choice of licensing policy and combine it with scientific publications.
- Release of open-source code is an efficient means of attracting interest to your research and getting in connection with potential collaborators.
- When governed by a firm but kind hand, the development of a common code base will enforce internal collaboration across different research activities and insure that results from one activity can be leveraged in another.

However, I think that some of these observations also hold for less comprehensive codes, and if this is what is most relevant in your case, you may still gain useful insight from the following discussion.

4.1. Choice of language for experimental programming

Unlike professional programmers, who typically have a detailed specification of software requirements from end users, developers of new numerical methods and computational algorithms seldom know exactly what their programs should do. Obviously, you will know what problem you aim to solve and have an idea of how to do it, but generally you will not know whether your approach will work until you have tested. The first attempt seldom does, and getting a working algorithm usually involves a lot of test and trials. Hence, the computer is your laboratory and should be treated as one, meaning that you should try to make your *experimental programming* as productive and reliable as possible, and that numerical experiments should be subject to the same standard as physical experiments (as discussed above).

One important choice you have to make is what language to use. This choice will obviously depend on what part of computational science you work in, what computer languages you have been exposed to, and the level of your programming skills. However, for the type of work I am doing (developing simulation technology to describe physical processes), my recommendation is crystal clear: Unless you are *really* fluent in a compiled language like C, C++, or FORTRAN, as much as possible of your initial experimental programming should be done in a scripting language like Python, Julia, or MATLAB/Octave that offers extensive support for numerical algorithms. The resulting code may not be as efficient as in a compiled language, but the development process is so much simpler. Once you get your ideas to work, you can always replace parts of your code by a compiled back-end code or rewrite everything from scratch in a compiled language. In my research group, we primarily use MATLAB for prototyping and C/C++ when developing production codes for our clients. Even with several very capable programmers in the group, it is my consistent experience that developing ideas in MATLAB and later reimplementing in C/C++ is more efficient than doing everything in C/C++ from the start.

Table 1 lists a number of factors I believe contribute to slow down experimental programming in a compiled language compared with a scripting language. The basic (imperative) syntax in a scripting language like MATLAB and its open-source clone Octave is fairly simple and will generally be intuitive to any mathematician with a basic course or two in programming. The language has many built-in mathematical abstractions, which together with numerical functions and routines for data analysis and visualization enable you to write quite compact programs that are close to the underlying mathematics. This is, of course, also possible in C++, provided that you have the

Table 1: List of factors that contribute to slow down the development cycle of experimental programming in third-generation compiled language compared to a fourth-generation, scripted language.

	3rd generation Fortran, C, C++	4th generation MATLAB, Python
Syntax	complicated	intuitive
Cross-platform	challenging	✓
Build process	✓	✗
Linking of external libraries	✓	✗
Type checking	static	dynamic
Mathematical abstractions	user-defined	built-in
Numerical computations	libraries	built-in
Data analysis and visualization	libraries/external	built-in
Debugger, profiling, etc	external/IDE	built-in
Traversing data structures	loops, iterators	vectorization†

† also: indirection maps and logical indices

right user-defined abstractions and suitable libraries for numerical computations, data analysis, and visualization. C++ is a multi-paradigm programming language that gives you the choice between procedural (imperative), object-oriented, generative, and template meta programming. Not only is the language wordier and less expressive than MATLAB/Octave, but it is easy to write quite obscure programs by utilizing features of the language that are alien to those who are less diligent in their search for sublime computer codes.

The build process and linking of external libraries are two other factors that not only contribute to slow down the development cycle, but also severely challenge the portability of your code. For small, standalone codes, this is not a big problem, but for medium to large-scale codes, it can be a very time-consuming task[2] to set up an appropriate build system, make sure that all necessary libraries and software modules are in sync, and insure cross-platform compatibility. These issues are largely non-existent in MATLAB since it is an interpreted and not a compiled language. There can obviously be some problems with backward and forward compatibility as a result of existing functionality being improved and new functionality being introduced, but by and large this has not been an issue for us. (The only exception might be the 3D graphics, which not only is surprisingly slow in MATLAB, but also has issues with cross-platform compatibility.)

Altogether, the development process tends to be quite different in MATLAB/Octave than in C++. Experimental programming is at its best when you can gradually make small changes to an existing and functional code. By using the built-in debugger, you can prototype while testing an existing program. As in any debugger, you can run code line by line, and stop and inspect variables at any point. However, since MATLAB/Octave is interpreted and has dynamic type checking, you can at any point not only change the content of your variables and data structures, but modify them completely by changing their type, introducing new data members, etc. You can also introduce new variables, data structures, and (anonymous) functions, or go back and rerun parts of the code with changed parameters. This way, you can try out each operation and build your program as you go. In my opinion, this is one of the primary reasons why prototyping in MATLAB/Octave is so efficient. On the other hand, static typing insures a certain consistency and can be very helpful in catching errors.

Let me end the section with a few words about homespun versus commercial or community codes. This question is particularly relevant when working at the interface between mathematics

[2]In the OPM project, the development team spent more than a year to get to a satisfactory solution.

and an applied science; in my case, simulation of CO_2 storage and hydrocarbon recovery. It is very hard to make any general recommendation, but let me observe the following: To avoid the danger of spending a lot of time reinventing the wheel, you should know what is current state-of-the-art and know both the capabilities and limitations of contemporary commercial and community codes. This is generally no little undertaking, but should at least be attempted to prevent you from falling prey to the infamous not-invented-here syndrome. On the other hand, reimplementing well-established functionality is a good way to increase your expertise and understand tacit assumptions and limitations in exiting software. As computational software matures, they tend to include an increasing number of (undocumented) safeguards ensuring robust behavior even when the software is used with inconsistent input and outside its normal range of validity. Simulators of physical processes should *never* be used as black boxes and it is of uttermost importance to have a number of experts that understand their inner workings.

4.2. Advice for good development practices

At the end, I will share some advice based on my experience as manager for the development of MRST [20, 26] as well as various research projects that have involved a significant amount of software development. My advice are not necessarily particularly original (see e.g., [30]), nor as focused on the actual coding and software tools as die-hard programmers would have liked, but hopefully they may still be useful:

- Learn standard tools for efficient software development like (distributed) version-control systems, issue trackers, unit testing, and task automation (i.e., systems that enable automatic acceptance and regression testing of your code) and use them to your advantage.
- Write for humans, not computers. In experimental programming, researchers spend the majority of their time reading/writing code and not waiting for computer runs to finish. Using consistent style and formatting makes it easier for others to read and understand your code.
- Write your software incrementally, using an *agile approach*. Get a first (simplified) version to work as early as possible, test it, and use the results to improve and expand your implementation. Be prepared to make (substantial) changes as you gain more insight into the problem.
- Break your code up into easy understandable functions, document what the functions do, their input and output arguments, and include, if possible, small examples of their typical and intended use.
- Be lazy! If the computer can do a task for you, let it do it. (One example: use automatic differentiation as discussed on page 20 to avoid the error-prone process of deriving and implementing derivatives of functions).
- Document functionality, not semantics and mechanics. If nothing else, insure that your future self is able to understand the code.
- Avoid premature optimization. Once the code is working as anticipated, you can always profile it and try to remove bottlenecks. If this obfuscates the code, you should consider keeping the original version as part of the documentation.

4.3. Maintaining integrity of your code

It is challenging to maintain integrity of your software under the (frequent) restructuring of code that inevitably follows from an agile approach; in particular for complex code features that are not well covered by e.g., unit tests. My best advice to maintain integrity is: *by thinking about it all the time*[3]. That said, you can make life easier for yourself if changes are committed to the version-control system and tested as frequently as possible. Best practices for multiple developers working on distributed software repositories suggest that commits should be merged into baseline every day. My experience is that this rule is difficult to enforce strictly in research projects, but it is seldom a good idea to work for more than a day without testing your code or let your private development branches deviate too far from baseline.

I also recommend that you use a tool for software self-testing like Jenkins, which automates the task of pulling code from your repository and running a set of predefined tests. Ideally, automated tests should incorporate as many of the cases you use for validation and verification as possible and should not only check that the code runs through without errors, but also verify that results are correct and monitor performance measures such as iteration counts, convergence rates, computational time, etc. To design meaningful tests, you should keep in mind that computed results are seldom bitwise identical so that suitable mathematical norms should be used when comparing against analytical solutions and previously stored computations. In OPM, for instance, we distinguish between *acceptance tests*, which check that you are within a prescribed tolerance of an analytical or numerical reference solution, and *regression tests*, which check that you reproduce previous results within zero or a tiny tolerance. Often, results of these tests need to be manually interpreted, since it is generally quite difficult to design fully automated tests of results that keep changing as your algorithms and methods get better and better. To avoid becoming a drag, the self tests should neither be too extensive nor run too frequently (e.g., once per day), and possibly be split up into multiple levels that are run at different time intervals.

Code review, or peer review of source code, is another recommended best practice to maintain integrity and insure correctness of new code. We have used this with some degree of success in OPM. However, formal code reviews can easily degenerate to counterproductive discussions about semantics and mechanics more than assumptions and functionality, and I am personally more fond of the informal peer review that arises naturally when multiple developers collaborate to test and maintain the same code.

4.4. Choosing the right development model

When setting up a new open-source project, there are several choices you need to make. First of all, you need to decide where to host your software repository and how you wish to distribute your code. Should you place your code in a *public repository* on a centralized service like GitHub and Bitbucket so that anybody who wants can have access at any time, or should you place it in a private repository and only provide periodic releases or only release it when you have finished the development? The fact that our work can/will be viewed and evaluated by our peers tends to keep most of us on our toes, and any of the first two alternatives is thus preferable. The third alternative is an ensnaring invitation to procrastinate important activities such as cleaning up code, documenting it, writing examples/tutorials, and so, and should thus be avoided. Whether you should choose the first or the second alternative depends on the commercial setting; how your research is funded; licensing and copyright questions; how you, your organization, or your project

[3]Supposedly, this is what Sir Isaac Newton answered when asked how he discovered the law of gravity. I have not been able to verify the truth of it, but I still think it is a good explanation that characterizes a lot of scientific work.

wish to collaborate or cooperate with external contributors to the software; and to what extent you want to retain control of future developments.

As explained already, my research group has been involved in two larger open-source projects over several years. The Open Porous Media (OPM) initiative follows an open and fully transparent innovation model. The code, which is hosted on a public repository on GitHub, has been jointly developed by researchers from several different organizations, funded through contract research. Copyright is held jointly by the developers' organizations and commercial companies funding the development. Contributions from third-party developers who wish to retain copyright are welcome and encouraged, which is a major advantage and incentive to contribute. However, contributions will be reviewed by a meritocratic group of official maintainers, who are appointed partly by affiliation, and contributions are requested to follow the GNU Public License (GPL), which is a copyleft license. The overall model insures a certain negative control for the funding partners, and is designed to prevent undue commercial utilization or hijacking of the project. The disadvantage is that this limits the incentives of non-funding partners to develop and maintain the code on their own. In my experience, it is also relatively costly to provide strategic direction and ensure that necessary consensus is reach among the developers. Several large open-source projects, like FEniCS, use an alternative model in which the project is managed by a non-profit foundation.

Some open-source projects are characterized by the fact that full copyright is owned exclusively by a single entity. This model is partially used for MRST [20, 26], which I will discuss in more detail in Section 5. MRST is an important part of our research infrastructure, but is developed as a shared public utility and not as a product to be sold. The central software modules that make up the biannual releases are owned by SINTEF (which is a non-profit research foundation) and are kept in private repositories on Bitbucket. Code contributions to the fundamental parts of the software are only accepted if the contributor transfers copyright of the code to SINTEF. On the other hand, we both encourage and help our collaborators and third-party developers to create and release add-on modules to the software as long as these follow the GPL license used for the rest of the software. In principle, these modules can modify or replace any part of the basic functionality in MRST. The reason we have chosen this particular model is that it gives us strategic control of the basic functionality, freedom to use the software we developed as we wish in contract research and if necessary release it to our clients under a different license, as well as clear incentives to maintain and improve the software.

4.5. Maintaining a lab journal

The discussion in the previous sections focused mainly on the software engineering, which for many scientific purposes is secondary to the development of algorithms and methods. In this latter process, the computer is essentially your laboratory. Within experimental sciences, it is basic knowledge that all experiments should be documented in a lab journal. The same should go without saying for numerical experiments:

- State hypotheses and cases you want to test, report your results, and discuss how you interpret them, potential causes for incorrect behavior, ideas for future improvements, etc.
- If possible, use a notebook format (like in Jupyter or Live Scripts in MATLAB) to set up your test cases, in which you can mix text, plots, and computer code.
- Save the exact input parameters and your results; disk space is much less expensive than the time you spend, should you later need to go back and recreate your results. Use a version-control system for all input data that have been manually created.

- Mark entries in the lab journal with a unique label from your version control system identifying the *exact* code and inputs you used to run the tests. With a version-control system like Git or alike, this would be the hexadecimal commit number.

This will save you a lot of time when (and not if) you have to go back and redo some of your experiments (e.g., when getting a paper back from review). It will also make it much simpler to communicate your preliminary results to colleagues or your supervisor/students.

4.6. Personal attitude

Last, but not least, I would like to point out that open source projects have a huge affinity on software nerds/geeks. To avoid becoming one, I suggest:

- Don't use your programming skills to show off! Try to aid others rather than alienating them.
- Keep it simple, stupid! If you feel pride in having managed to condense a complicated computational construct to a few code lines, chances are pretty high that you will be the only one understanding it.
- Try not to become a religious fanatic who quarrels or fights turf wars over standardization and whose opinions get stronger the less important the issue discussed is.
- If using object orientation, avoid becoming an *onion producer* who makes layers upon layers of abstractions (with no core) in an attempt to be generic.

With this, I wish you good luck in your experimental programming. I look forward to see your source code on the net as an integer part of your next paper.

5. The MATLAB Reservoir Simulation Toolbox

In the last section of this essay, I try to make the ideas discussed above more concrete and demonstrate that it is indeed possible to also publish codes that have a significant commercial potential. To this end, I will describe a comprehensive open-source software developed by my research group over the past decade. The discussion is admittedly detailed at times, but I still hope that readers outside of the reservoir simulation community may find it inspiring and possibly learn something from our use of the MATLAB (or Octave) language. I believe, in particular, that our close relation between mathematical operators and their numerical implementation can be useful for others working with low-order finite-volume discretizations of flow equations within other fields of science and engineering.

5.1. A brief history of the software and why it was developed

What is today the MATLAB Reservoir Simulation Toolbox (MRST) [21, 20, 15, 5, 26] grew out of research on mimetic discretizations and multiscale methods for reservoir simulation on complex grids, as outlined in Section 3.3. It was early decided to use MATLAB as our primary development platform, in part because of an idea that a scripting language would be more efficient, inspired by the late Hans Petter Langtangen's pioneering work [16], and in part because we happened to know MATLAB quite well. At first, our development was poorly coordinated. As an example, writing one of our earliest papers [1], involved three different codes, each written by only one of the authors. Obviously, this was no viable path, and hence MRST was born. A few years earlier, we had published an educational paper that essentially explained how to implement a simple reservoir simulator in less than 50 lines of MATLAB code [2]. This paper and the accompanying code had

attracted much more interest than anticipated. Inspired by this, and with Helge's last guideline at the back of my mind, I decided we should release our new code under a free software license. We chose to use the GNU General Public license, since this would prevent others from simply picking up our software and use it in commercial products.

Pushing multiscale and mimetic discretization methods toward realistic applications meant that we had to develop a lot of general infrastructure for multiphase flow simulations on unstructured grids [21]. This made the new software quite attractive also in other projects, and MRST grew into a general prototyping framework that was used in more or less all of our research. All the way, our development policy has been that generic ideas from contract research is put into MRST and released publicly. Code which is decided to have business-critical value to our clients or ourselves, is isolated in separate branches or modules and is never published. Since the software serves many different purposes, we have, after a bit back and forth, come to the conclusion that it is best to organize it so that it consists of a small core module offering basic functionality, and a large set of add-on modules that each implements specific computational methods or mathematical models. Many of these modules can be combined to support more comprehensive workflows, but there are also cases in which two modules offer functionality that makes them mutually exclusive.

Continuing to release research results in the form of open-source software was not uncontroversial within my organization, but somehow I managed to convince my superiors that the marketing effect would far out-weight the loss of potential license fees. My winning argument was that when your marked is monopolized by a few software providers, you need to use another mechanism to attract potential clients. With our industry clients, my argument is that allowing us to release generic parts of new functionality we develop for them, is the price they pay for being able to leverage functionality developed for other clients.

Initially, MRST was written using a procedural (imperative) programming paradigm and focused almost exclusively on incompressible flow [21]. We made a few attempts at extending the capabilities to contemporary flow physics, but were not really successful until one of my colleagues, Stein Krogstad, decided to implement *automatic differentiation* [27]. (I will come back to this in more detail below.) This opened unparalleled capabilities for rapid prototyping and within a few weeks, we had developed our first compressible, three-phase solver and verified that it gave satisfactory match with the market-leading commercial simulator simplified test cases. Ensuring robust and accurate simulations on models of real hydrocarbon assets is far more challenging, and it took us several months to figure out the correct way to interpolate tabulated fluid data[4], reverse-engineer undocumented features in models of near-horizontal wells, etc. This is generally where the science stops and the art or tricks-of-the-trade starts. A full-fledged reservoir simulator contains a lot of intricate functionality, like well modelling, nonlinear solvers with time-step control, (multilevel) iterative linear solvers with appropriate preconditioning methods, and so on. This means that codes written with a procedural approach gradually become quite unwieldy, unless these are meticulously designed, which seldom is the case in experimental programming. To amend this, Olav Møyner (who was doctoral student of Helge and me at the time) developed a new object-oriented framework. Combined with automatic differentiation, this framework offers very powerful support for rapid prototyping [15, 5, 23].

At this point, you may ask how efficient MATLAB is for reservoir simulation. The incompressible simulators written using a procedural approach are quite efficient, typically a factor 3–5 times slower than commercial solvers, and we have been able to simulate two-phase flow on models having

[4] As an example: If two parameters μ and B that enter your flow equations as $1/\mu B$, should you interpolate μ and B, μB, $1/\mu$ and $1/B$, or $1/\mu B$? It turned out that the latter choice was the correct.

up to 60 million grid cells on a standard workstation. Industry-standard models for three-phase compressible flow are significantly more computationally demanding than incompressible, two-phase flow. Moreover, our automatic differentiation approach has primarily been written to be as flexible as possible and incurs a certain overhead, but we believe that this can be significantly reduced through a more careful implementation of our library for automatic differentiation. In sum, I currently would not recommend simulation of models containing more than a few hundred thousand cells, which in most cases should be more than sufficient when developing proof-of-concept simulators and workflows on models with realistic complexity.

Looking at the large user community that the software has attracted, it seems that a somewhat suboptimal computational performance is by far out-weighted by the flexibility, free access, and full source code that MRST offers. Each of the past eight biannual releases have been downloaded from 1000 to 2000 unique computers (according to Google Analytics), and at the time of writing, the software has been used in 90 master and doctoral theses, and in more than 110 journal and proceedings papers by authors not affiliated with SINTEF.

What are the points that make the software attractive to such a large audience? First of all, it is because the software is free, in a high-level language like MATLAB, and offers full access to source code. However, I also believe that the fact that we have been quite diligent in documenting the code and developing tutorials and examples that highlight the salient features of various functionality has contributed to make it more attractive. Last, but not least, we have put significant effort into developing routines for reading and processing input data on industry-standard format, which significantly simplifies the process of testing new methods on realistic scenarios.

Let me also add that major parts of MRST can also be run in the latest version of Octave, as a completely free alternative to MATLAB, provided a number of changes are made to account for minor differences between Octave and MATLAB. The main exception is graphical user interfaces, which are written quite differently in the two languages. Originally, OPM Flow (http://opm-project.org) was developed as a C++-cousin of MRST, intended for full-scale commercial simulations. The two have many similarities in the underlying design, which simplifies the process of moving methods prototyped in MRST into industrial adoption. Lately, however, the OPM project has focused more on optimizing computational performance and this has resulted in larger and increasing differences in the two codes.

5.2. Key ideas for rapid prototyping

In this section, I will try to briefly explain some of the principles we have used in MRST to support rapid prototyping. Our choices are admittedly strongly influenced by the type of problems we study and the low-order finite-volume methods we use to this end. Still, there might be some insight here that also applies to other types of problems and numerical methods.

In developing the toolbox, we have tried to make functionality that enables clean and simple implementation of flow equations, as close to the underlying mathematical models as possible. This way, we seek to insure less error-prone coding and create quite compact codes that are relatively simple to maintain and extend. Key ideas to this end, include

- Hide specific details of grid, discretizations, constitute laws, and parameters describing geologic and petrophysical properties.
- Always use a fully unstructured grid format to represent all types of grids so that algorithms can be implemented without knowing the specifics of the grid.
- Define abstract discretization and averaging/mapping operators that are not tied to specific flow equations and can be precomputed independently.

- Use vectorization to ensure an almost 1-to-1 correspondence between continuous and discrete variables to avoid visible loops and use as few indices as possible.
- Use automatic differentiation to avoid having to explicitly linearize flow equations, analytically compute and implement derivatives, gradients and Jacobians, which generally is a time-consuming and error-prone process.

Vectorization, logical indexing, and summation techniques. MATLAB/Octave is a quite expressive language and has many different constructions that help to make your code shorter and hence easier to read and maintain. Two relative simple techniques are commonly used to avoid looping through arrays, as one typically would do in C++ and similar compiled languages. Vectorization lets you operate directly on the matrix level and write code almost as if you were working with scalar variables

```
% Vectorization
f = sin(y).*exp(-x.^2/2);
```

```
% For-loop
f = zeros(size(x));
for i=1:numel(x)
  f(i) = sin(y(i))*exp(-x(i)^2/2);
end
```

Because of MATLAB's Just-in-time (JIT) compiler, the vectorized code can be slower than the `for` loop when the arrays `x` and `y` have few elements. On the other hand, the vectorized code is much closer to the mathematics and significantly more efficient on large arrays. This was a trivial example, but the principle applies to more complex cases. Another nice feature is logical indexing. To exemplify, we can set all negative elements of a vector to zero

```
v(v<0) = 0;
```

or compute the average of negative and positive values

```
i = v>=0;
avg = [sum(v(~i)) sum(v(i))]./[sum(~i) sum(i)];
```

Another useful construct, is the `accumarray(p,v)` function, which collects all elements of `v` that have identical subscripts in `p`, sums them, and stores in the location given by `p`. As an example, let `p` be a partition vector defining a coarse grid so that `p(i)=j` if cell `i` belongs to block `j`. The average of a scalar quantity `v` defined in each cell can be computed as

```
avg = accumarray(p,v)./accumarray(p,1);
```

To compute the average over a vector quantity with m elements per cell, defined as an $n \times m$ array `v`, we can use a sparse matrix to sum the elements and `bsxfun` for element-by-element division,

```
tmp = sparse(p, n, 1)* [v, ones(n,1)];
avg = bsxfun(@rdivide,tmp(:,1:end-1),tmp(:,end))
```

The last two constructs are powerful, although not as neat as logical indexing, and are used a lot in MRST for computational efficiency and to generate compact codes devoid of `for` loops.

Grids and discretizations. The most fundamental quantity in MRST is the grid, which generally will be a collection of 3D polyhedral cells having an unstructured topology. To ensure maximum flexibility in developing new computational algorithm, all grids are represented in a relatively verbose format containing geometric properties such as vertices; face centroids, normals, and areas; and cell centroids and volumes. The grid topology is described in terms of mappings between cells and faces, and between faces and the cells they separate, as shown in Figure 2. Using these mappings, we can define discrete divergence and gradient operators. The `div` operator is a linear mapping from faces

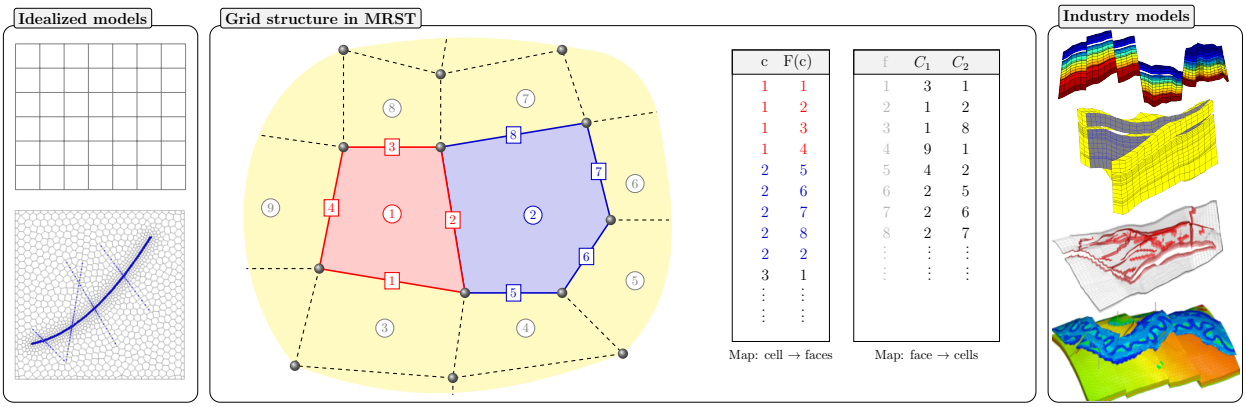


Figure 2: Illustration of the grid structure in MRST and typical grids used in subsurface flow simulation. The two tables show the mappings used to define discrete differentiation operators; for the face-to-cell mapping, only the last two columns are actually stored.

to cells. Let $\mathbf{v}[f]$ denote a discrete flux over face f with orientation from cell $C_1(f)$ to cell $C_2(f)$. Then,

$$\texttt{div}(\mathbf{v})[c] = \sum_{f \in F(c)} \operatorname{sgn}(f)\mathbf{v}[f], \qquad \operatorname{sgn}(f) = \begin{cases} 1, & \text{if } c = C_1(f), \\ -1, & \text{if } c = C_2(f). \end{cases} \tag{3}$$

Likewise, the `grad` operator maps from cell pairs $C_1(f), C_2(f)$ to faces f

$$\texttt{grad}(\mathbf{p})[f] = \mathbf{p}[C_2(f)] - \mathbf{p}[C_1(f)], \tag{4}$$

where $\mathbf{p}[c]$ is the pressure associated with cell c. Since `div` and `grad` are linear operators, they can be represented by a sparse matrix $\mathbf{D}$ so that $\texttt{grad}(\mathbf{x}) = \mathbf{D}\mathbf{x}$. If we assume zero flux across the boundary, the discrete gradient operator is the adjoint of the divergence operator, as in the continuous case, i.e., $\texttt{div}(\mathbf{x}) = -\mathbf{D}^T\mathbf{x}$. To discretize Poisson's equation (1), we also need to represent the operator $\nabla \cdot \mathbf{K}\nabla$ by defining a *transmissibility* $\mathbf{T}[f]$, so that $\mathbf{v}[f] = -\mathbf{T}[f]\texttt{grad}(\mathbf{p})[f]$. To derive concrete expression for $\mathbf{T}$, we change notation slightly and let $\mathbf{v}_{i,j}$ denote the flux from cell i to cell j. Using Darcy's law and a standard finite-difference approximation, we have that

$$\mathbf{v}_{i,j} = -\int_{\Gamma_{ij}} \mathbf{K}\nabla p \cdot \vec{n}_{ij}\, ds \approx A_{ij}\mathbf{K}_i \frac{(p_i - \pi_{i,j})\vec{c}_{i,j}}{|\vec{c}_{i,j}|^2} \cdot \vec{n}_{i,j} = T_{i,j}(p_i - \pi_{i,j}),$$

where the interface Γ_{ij} between cells i and j has area A_{ij} and directional normal $\vec{n}_{i,j}$. Moreover, $\mathbf{K}_i$ is the constant value of $\mathbf{K}$ inside cell i, π_{ij} is the pressure at the centroid of Γ_{ij}, and $\vec{c}_{i,j}$ is the vector from the cell centroid to the face centroid. A similar expression holds for cell j. If we require continuity of fluxes, $\mathbf{v}_{i,j} = -\mathbf{v}_{j,i}$, it follows that $T_{ij} = \left[T_{i,j}^{-1} + T_{j,i}^{-1}\right]^{-1}$. Constructing the discrete operators and computing the transmissibility can be done in approximately twenty lines in MATLAB using the unstructured grid format, as we will see later. In practice, you will probably want to add a few safeguards as we have done in MRST, which make the code somewhat longer. Once the operators and $\mathbf{T}$ are computed, we do not need to know any detail of the grid to discretize our flow equations, which can be done quite compactly, as shown in Figure 3.

Automatic differentiation. The basic premise of automatic differentiation (AD), also called algorithmic differentiation, is that standard function evaluations in a computer code consists of a sequence of elementary unary and binary operations, for which known derivative rules exist. The key idea is

Continuous

Incompressible flow:

$$\nabla \cdot (\mathbf{K}\nabla p) + q = 0$$

Compressible flow:

$$\frac{\partial(\phi\rho)}{\partial t} + \nabla \cdot (\rho\mathbf{K}\nabla p) + q = 0$$

Discrete in MATLAB

Incompressible flow:

```
eq = div(T .* grad(p)) + q;
```

Compressible flow:

```
eq = (pv(p).*rho(p)-pv(p0).*rho(p0))/dt ...
     + div(avg(rho(p)).*T.*grad(p))+q;
```

Figure 3: Correspondence between how flow equations are specified mathematically and implemented in MRST using the discrete operators. Here, `pv` and `rho` are functions evaluating porosity ϕ and density ρ as function of pressure, and `avg` is a mapping from cells to faces, $\texttt{avg}(\rho)[f] = \frac{1}{2}(\rho[C_1(f)] + \rho[C_2(f)])$.

now to keep track of variable values and their derivatives with respect to a set of independent variables. Consider a scalar independent variable x and a dependent variable v computed as a function of x, i.e., $v = f(x)$. Automatic differentiation introduces a new extended pair $\langle x, 1\rangle$, i.e., the value x and its derivative 1. Using this extended pair, the computer can use elementary derivative rules for unary and binary operations together with the chain rule to mechanically accumulate derivatives of v evaluated at the *specific value* x represented as $\langle f(x), f'(x)\rangle$. If, for instance, $v = \sin(x)$, then the corresponding AD-pair reads $\langle \sin(x), -\cos(x)\rangle$. Similarly, we have for binary operators

$$\langle u, u_x\rangle + \langle v, v_x\rangle = \langle u + v, u_x + v_x\rangle\,, \qquad \langle u, u_x\rangle * \langle v, v_x\rangle = \langle uv, uv_x + u_x v\rangle\,.$$

In MRST, these rules are implemented using *operator overloading* as suggested in [27], so that all function evaluations can be written exactly the same way irregardless of whether AD is used or not.

Putting it all together. Now, let us see if we can put the pieces together and implement a flow solver that is applicable to both structured an unstructured grids. For the moment, I will skip details of how the grid `G` and the permeability `K` are generated. Simple grids can be generated by a few function calls to grid-factory routines in MRST. We start by extracting grid information

```
C  = G.faces.neighbors;
C  = C(all(C ~= 0, 2), :);
cn = gridCellNo(G);
F  = G.cells.faces(:,1);
[nf,nc] = deal(size(C,1), G.cells.num);
```

The first two lines extract the last two columns of the face-to-cell map from Figure 2 and remove all external faces (indicated by one of the cell numbers being zero). The next two lines extract the two columns of the cell-to-face mapping, whereas the last line gets the number `nf` of internal face and the number of cells `nc`. Using this information, it is straightforward to construct the discrete operators

```
D = sparse([(1:nf)'; (1:nf)'], C, ones(nf,1)*[-1 1], nf, nc);
grad = @(x) D*x;
div  = @(x) -D'*x;
```

To compute the transmissibility, we start by extracting the face normal and the matrix containing vectors from cell to face centroids

```
sgn = 2*(cn == G.faces.neighbors(F, 1)) - 1;
c   = G.faces.centroids(F,:) - G.cells.centroids(cn,:);
n   = bsxfun(@times, sgn, G.faces.normals(F,:));
```

Here, the first line determines the correct sign of the face normal. Now, we have all information we need to compute the transmissibility,

```
[i,j] = deal([1 1 2 2],[1 2 1 2]);
hT = sum(c(:,i) .* bsxfun(@times, K(cn,:), n(:,j)), 2);
hT = hT./ sum(c.*c,2);
T  = 1 ./ accumarray(F, 1 ./ hT, [G.faces.num, 1]);
T  = T(all(C~=0,2),:);
```

The first line sets up of the row and column numbers of the permeability tensor, which is stored as a vector of the form $[K_{xx}, K_{xy}, K_{yx}, K_{yy}]$ for each cell in a 2D grid. (For 3D grids, $\mathbf{K}$ has nine entries.) The next two lines compute the one-sided transmissibilities, the next line their harmonic average, and the last line extracts those corresponding to internal faces. In the actual prototyping framework, you would not have to implement all the generic code lines discussed above, but rather call a function that does this for you with a lot of safeguards

```
S = setupOperatorsTPFA(G,rock);
```

Here, `rock` is a structure containing petrophysical properties, including $\mathbf{K}$.

Now, we are finally in a position to specify and solve our equations. To this end, we declare pressure as our primary variable, which hence will be considered the independent variable when linearizing the discrete equations

```
q     =  ... % this is case specific
p     = initVariablesADI(zeros(nc,1));
eq    = div(T.*grad(p))+q;
eq(1) = eq(1) + p(1);
p     = -eq.jac{1}\eq.val;
```

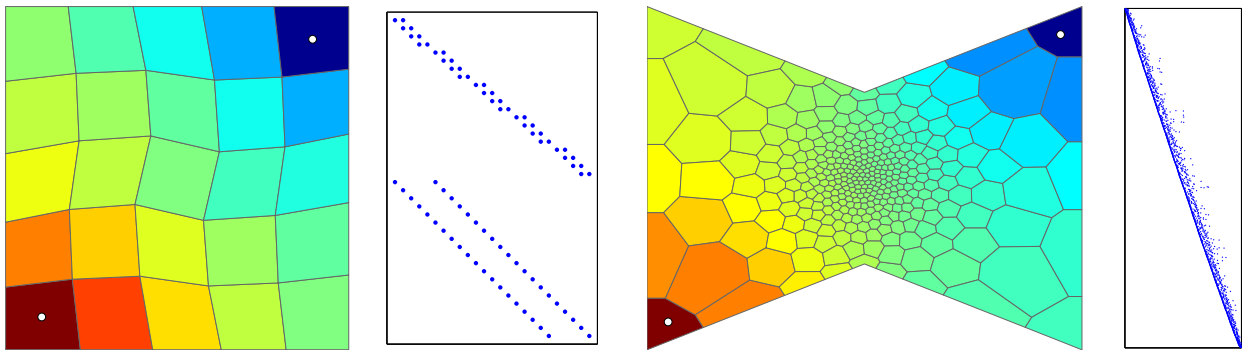
The second line defines p to be an AD-variable initialized with all zeros, the third line defines our discrete equation on residual form as shown in Figure 3. With zero Neumann conditions only, the solution is not unique and the fourth line modifies the first element of the system matrix to (somewhat arbitrarily) fix the pressure in the first cell to zero. Going back to (1), we have a residual equation on the form $\mathbf{R}(\mathbf{p}) = \mathbf{A}\mathbf{p} + \mathbf{q} = \mathbf{0}$. The last line computes the solution as $\mathbf{p} = -(\frac{\partial \mathbf{R}}{\partial \mathbf{p}})^{-1}\mathbf{q} = -\mathbf{A}^{-1}\mathbf{q}$. Figure 4 shows the setup and solution of two specific problems. The only difference between these two cases is the specification of the grid `G` and the source term q. Notice also that the same code can be used to compute pressure on complex stratigraphic and unstructured grids in 3D after trivial modifications of the `i` and `j` arrays to span 3×3 tensors in the transmissibility calculation.

To extend the code to the compressible, single-phase equation shown in Figure 3, we need to define functions that compute ρ and ϕ as functions of p and add an outer loop for the time steps and an inner Newton iteration to solve what is now a nonlinear residual equation; details are given in [20]. These single-phase problems are almost trivial, but should give you an idea of how to construct more advanced solvers.

Acknowledgments

First of all, I would like to thank Helge for the fruitful collaboration and cooperation we have had over the past 25 years. This has not only benefited the many students we have supervised together, but also my more senior colleagues, whom I continuously expose to the requirements for high scientific quality I have learned from you. By example, you have taught me that being a supervisor is similar to being a father, you do not stop caring for your children and helping just because they have left the nest. I have tried to pay this on as best as I could.

This essay was written while participating in the long program on Computational Issues in Oil Field Applications at the Institute for Pure and Applied Mathematics (IPAM) as UCLA. I thank IPAM for the invitation, the generous funding, and the great hospitality offered to me.



```
% Make grid
G = twister(cartGrid([8 8]));
G = computeGeometry(G);

% Set source terms (flow SW -> NE)
q = zeros(G.cells.num,1);
q([1 end]) = [1 -1];

% Unit insotropic permeability
K = ones(G.cells.num,4); K(:,[2 3]) = 0;
```

```
% Make grid using external grid generator
pv = [-1 -1; 0 -.5; 1 -1; 1 1; 0 .5; -1 1; -1 -1];
fh = @(p,x) 0.025 + 0.375*sum(p.^2,2);
[p,t] = distmesh2d(@dpoly, fh, 0.025, [-1 -1; 1 1], pv, pv);
G = computeGeometry(pebi(triangleGrid(p, t)));

% Set source terms (flow SW -> NE)
q = zeros(G.cells.num,1);
v = sum(G.cells.centroids,2);
[~,i1]=min(v); [~,i2]=max(v);
q([i1 i2]) = [1 -1];
```

Figure 4: Poisson problems describing single-phase flow on a rectangular grid and on an unstructured Voronoi grid; the latter is constructed from a triangulation generated by an open-source mesh generator [29]. The color plots show pressure with red denoting high pressures near the fluid source and blue low pressures near the sink. The `spy` plots show the sparsity structure of the $\mathbf{D}$ matrix used to define the discrete `div` and `grad` operators. For the rectangular grid, the upper block corresponds to $\frac{\partial}{\partial x}$ and the lower block $\frac{\partial}{\partial y}$. Permeability is specified in exactly the same way for the unstructured and structured grids.

References

[1] J. E. Aarnes, S. Krogstad, and K.-A. Lie. A hierarchical multiscale method for two-phase flow based upon mixed finite elements and nonuniform coarse grids. *Multiscale Model. Simul.*, 5(2): 337–363, 2006. doi:10.1137/050634566.

[2] J. E. Aarnes, T. Gimse, and K.-A. Lie. An introduction to the numerics of flow in porous media using Matlab. In G. Hasle, K.-A. Lie, and E. Quak, editors, *Geometrical Modeling, Numerical Simulation and Optimisation: Industrial Mathematics at SINTEF*, pages 265–306. Springer Verlag, Berlin Heidelberg New York, 2007. doi:10.1007/978-3-540-68783-2_9.

[3] J. E. Aarnes, S. Krogstad, and K.-A. Lie. Multiscale mixed/mimetic methods on corner-point grids. *Comput. Geosci.*, 12(3):297–315, 2008. doi:10.1007/s10596-007-9072-8.

[4] F. O. Alpak, M. Pal, and K.-A. Lie. A multiscale method for modeling flow in stratigraphically complex reservoirs. *SPE J.*, 17(4):1056–1070, 2012. doi:10.2118/140403-PA.

[5] K. Bao, K.-A. Lie, O. Møyner, and M. Liu. Fully implicit simulation of polymer flooding with mrst. *Comput. Geosci.*, 2017. doi:10.1007/s10596-017-9624-5.

[6] J. B. Buckheit and D. L. Donoho. WaveLab and reproducible research. In A. Antoniadis and G. Oppenheim, editors, *Wavelets and Statistics*, volume 103 of *Lecture Notes in Statistics*, pages 55–81. Springer New York, New York, NY, 1995. doi:10.1007/978-1-4612-2544-7_5.

[7] Z. Chen and T. Y. Hou. A mixed multiscale finite element method for elliptic problems with oscillating coefficients. *Math. Comp.*, 72:541–576, 2003. doi:10.1090/S0025-5718-02-01441-2.

[8] M. A. Christie and M. J. Blunt. Tenth SPE comparative solution project: A comparison of upscaling techniques. *SPE Reservoir Eval. Eng.*, 4:308–317, 2001. doi:10.2118/72469-PA. Url: `http://www.spe.org/csp/`.

[9] J. Claerbout. Reproducible computational research: A history of hurdles, mostly overcome. URL `http://sepwww.stanford.edu/sep/jon/reproducible.html`.

[10] J. F. Claerbout and M. Karrenbach. Electronic documents give reproducible research a new meaning. In *SEG Technical Program Expanded Abstracts 1992*, pages 601–604. Society of Exploration Geophysicists, 1992.

[11] H. Holden. Seven guidelines on scientific computing. Report (unpublished), Norwegian Institute of Technology, 1994.

[12] T. Y. Hou and X.-H. Wu. A multiscale finite element method for elliptic problems in composite materials and porous media. *J. Comput. Phys.*, 134:169–189, 1997. doi:10.1006/jcph.1997.5682.

[13] P. Jenny, S. H. Lee, and H. A. Tchelepi. Multi-scale finite-volume method for elliptic problems in subsurface flow simulation. *J. Comput. Phys.*, 187:47–67, 2003. doi:10.1016/S0021-9991(03)00075-5.

[14] V. Kippe, J. E. Aarnes, and K.-A. Lie. A comparison of multiscale methods for elliptic problems in porous media flow. *Comput. Geosci.*, 12(3):377–398, 2008. ISSN 1420-0597. doi:10.1007/s10596-007-9074-6.

[15] S. Krogstad, K.-A. Lie, O. Møyner, H. M. Nilsen, X. Raynaud, and B. Skaflestad. MRST-AD – an open-source framework for rapid prototyping and evaluation of reservoir simulation problems. In *SPE Reservoir Simulation Symposium, 23–25 February, Houston, Texas*, 2015. doi:10.2118/173317-MS.

[16] H. P. Langtangen. *Python scripting for computational science*, volume 3 of *Texts in Computational Science and Engineering*. Springer, 2004. doi:10.1007/978-3-662-05450-5.

[17] R. J. LeVeque. Wave propagation software, computational science, and reproducible research. In *Proceedings of the International Congress of Mathematicians, Madrid, Spain*, pages 1227–1253. European Mathematical Society, 2006.

[18] R. J. LeVeque. Top ten reasons to not share your code (and why you should anyway). *SIAM News*, 1, 2013.

[19] R. J. LeVeque, I. M. Mitchell, and V. Stodden. Reproducible research for scientific computing: Tools and strategies for changing the culture. *Comput. Sci. Engng.*, 14(4):13–17, July 2012. doi:10.1109/MCSE.2012.38.

[20] K.-A. Lie. *An Introduction to Reservoir Simulation Using MATLAB: User guide for the Matlab Reservoir Simulation Toolbox (MRST)*. SINTEF ICT, `http://www.sintef.no/Projectweb/MRST/publications`, 3rd edition, December 2016.

[21] K.-A. Lie, S. Krogstad, I. S. Ligaarden, J. R. Natvig, H. M. Nilsen, and B. Skaflestad. Open source MATLAB implementation of consistent discretisations on complex grids. *Comput. Geosci.*, 16:297–322, 2012. ISSN 1420-0597. doi:10.1007/s10596-011-9244-4.

[22] K.-A. Lie, O. Møyner, J. R. Natvig, A. Kozlova, K. Bratvedt, S. Watanabe, and Z. Li. Successful application of multiscale methods in a real reservoir simulator environment. *Comput. Geosci.*, 2017. doi:10.1007/s10596-017-9627-2.

[23] O. Møyner. *Next generation multiscale methods for reservoir simulation*. PhD thesis, Norwegian University of Science and Technology, 2016. URL `http://hdl.handle.net/11250/2431831`.

[24] O. Møyner and K.-A. Lie. A multiscale restriction-smoothed basis method for high contrast porous media represented on unstructured grids. *J. Comput. Phys.*, 304:46–71, 2016. doi:10.1016/j.jcp.2015.10.010.

[25] O. Møyner and K.-A. Lie. A multiscale restriction-smoothed basis method for compressible black-oil models. *SPE J.*, 21(06), 2016. doi:10.2118/173265-PA.

[26] MRST. The MATLAB Reservoir Simulation Toolbox, version 2016b, 12 2016. `http://www.sintef.no/MRST/`.

[27] R. Neidinger. Introduction to automatic differentiation and MATLAB object-oriented programming. *SIAM Review*, 52(3):545–563, 2010. doi:10.1137/080743627.

[28] R. D. Peng. Reproducible research and Biostatistics. *Biostatistics*, 10(3):405–408, 2009. doi:10.1093/biostatistics/kxp014.

[29] P.-O. Persson and G. Strang. A simple mesh generator in matlab. *SIAM Review*, 46(2):329–345, 2004. doi:10.1137/S0036144503429121.

[30] G. Wilson, D. A. Aruliah, C. T. Brown, N. P. Chue Hong, M. Davis, R. T. 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 Biology*, 12(1):1–7, 01 2014. doi:10.1371/journal.pbio.1001745.