

# Nmag micromagnetic simulation tool – software engineering lessons learned

Hans Fangohr  
 University of Southampton  
 SO17 1BJ Southampton  
 United Kingdom  
 fangohr@soton.ac.uk

Maximilian Albert  
 University of Southampton  
 SO17 1BJ Southampton  
 United Kingdom  
 maximilian.albert@soton.ac.uk

Matteo Franchin  
 Cambridge  
 United Kingdom  
 franchinm@gmail.com

## ABSTRACT

We review design and development decisions and their impact for the open source code Nmag from a software engineering in computational science point of view. We summarise lessons learned and recommendations for future computational science projects. Key lessons include that encapsulating the simulation functionality in a library of a general purpose language, here Python, provides great flexibility in using the software. The choice of Python for the top-level user interface was very well received by users from the science and engineering community. The from-source installation in which required external libraries and dependencies are compiled from a tarball was remarkably robust. In places, the code is a lot more ambitious than necessary, which introduces unnecessary complexity and reduces maintainability. Tests distributed with the package are useful, although more unit tests and continuous integration would have been desirable. The detailed documentation, together with a tutorial for the usage of the system, was perceived as one of its main strengths by the community.

## CCS Concepts

- Software and its engineering → Software creation and management; •Computing methodologies → Modeling and simulation; Continuous models; •Applied computing → Computer-aided design; Physics;

## Keywords

Nmag, Computational Science Software Engineering, Python, Finite Elements

## 1. SUMMARY

The Nmag software was developed from 2005 to early 2012, and first released in 2007 as open source code. Nmag has a Python user interface, and uses Objective Caml code and established High Performance Computing libraries un-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

*SE4Science'16, May 16 2016, Austin, TX, USA*

© 2016 ACM. ISBN 978-1-4503-4167-7/16/05...\$15.00

DOI: <http://dx.doi.org/10.1145/2897676.2897677>

der the hood, to solve time-dependent partial differential equations using finite element discretizations. It has been in use for nearly a decade, and with this review, we try to share experience to help improve future generations of software engineering projects in computational science.

We start by summarizing key lessons and recommendations on this page below, with pointers to more detailed discussion in the main part of the paper (Sect 2 to Sect 4).

There are two points of view to consider in the development of a tool for computational scientists. On the one hand, the *end-users* want the software to be easy to use in order to improve their science. On the other hand, the *developers* of the software care about additional aspects, such as extensibility and maintainability. We divide our recommendations according to these two different points of view:

---

<b>Recommendations primarily affecting end-users</b>	Sect.
--	-------

---

Embedding simulation into existing programming language provides unrivaled flexibility	4.1
Python-based top-levels of the tool allow users to modify it to suit their needs	4.2
Python is a popular language that is perceived to be easy to learn by (non-computer) scientists	4.2
Documentation and tutorials are important	4.11

---



---

<b>Recommendations primarily affecting developers</b>	Sect.
---	-------

---

Version control tool use is essential	4.9
System tests are essential, unit tests are very useful	4.10
Continuous integration is very useful	4.12
Limit the supported or anticipated functionality to minimize complexity and enhance maintainability	4.8
Code generation based on user provided equations is up-front investment but widens applicability of tool	4.6
Choice of unconventional programming language can limit the number of scientists joining the project as developers	4.2
OCaml not quite as fast as C/C++/Fortran	4.4

---

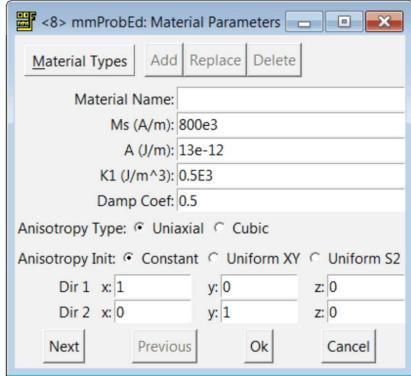


Figure 1: The OOMMF graphical user interface for problem definitions

## 2. INTRODUCTION

Nmag is a micromagnetic simulation package. Micromagnetics is a continuum theory of magnetization at the nano- and micrometer scale. The magnetization is a continuous 3d vector field defined throughout a ferromagnetic body. The dynamics of this vector field are determined by an equation of motion, which depends on solving a non-linear partial integro-differential equation. Using spatial discretization, this partial differential equation (PDE) can be solved numerically. Nmag uses a combination of finite element and boundary element methods [16, Section 2.2] to solve the integro-PDE in every time step. Within this finite element model, the magnetization is represented as 3d-vector degrees of freedom at every finite element node. For the time integration, the degrees of freedom at all nodes are treated as a system of coupled ordinary differential equations, in line with the related open source software Magpar [19] and other tools that are not freely available.

Nmag is summarized in a short paper [12]. More technical detail, in particular on the underlying multi-physics finite element library `nsim`, is given in a manuscript available on the arXiv [11]. Some aspects of the parallel execution model have been published [13]. The project home page [1] contains the code as a tar-file. For reference, the repository from which the tarball is built, is available as a Mercurial repository on Bitbucket [2]. A number of PhD students have contributed to the software, and their theses are an additional useful source of information, covering an overview of the capabilities of the code [14, chapter 5], long-range demagnetization field calculation, boundary elements and matrix compression [16], macro-geometry periodic boundary conditions and mesh generation [7].

Computational micromagnetics became feasible and accessible with the public domain release of the OOMMF Software by the National Institute of Standards and Technology (NIST) in the late 90s [8]. The OOMMF software uses a finite difference discretization (with different strengths and weaknesses than the finite element discretization but outside the scope of this paper), which is based on C++ code with a Tcl and Tk interface. A simulation run can be configured through a graphical user interface (see figure 1 for an example), or through setting parameters in a configuration file which uses Tcl syntax.

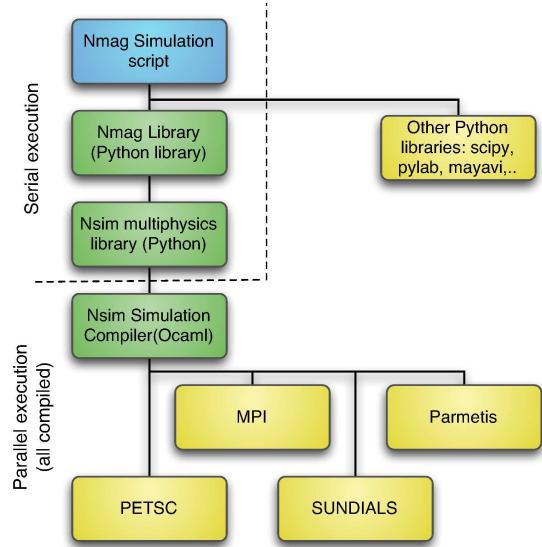


Figure 2: Nmag architecture overview

The aim of the Nmag package was, among other things, to provide a finite element based discretization approach to complement the OOMMF tool. Simultaneously, and without being aware of each others efforts, the finite element Magpar code was developed, and later released as open source [19].

In this report, we focus on software engineering aspects of the Nmag tool, including the user interface, the choice of languages and tools, and the open source model. We summarize the Nmag project in section 3, before addressing the lessons learned in section 4.

## 3. NMAG PROJECT SUMMARY

**Architecture:** Figure 2 summarizes the Nmag architecture: the blue box at the top represents the (Python) code that the end-user assembles to define the simulation computation; this includes material parameters, the file name of the finite element mesh to define the geometry, the physical process that should be simulated etc. The instructions are written as a Python script that makes use of the `nmag` Python library (top green box), which in turn composes its functionality from the (Python) `nsim` multi-physics library. The Python `Nsim` multi-physics library is an interface to the functionality provided through the implementation (bottom green box) in OCaml [4]. We make use of existing high performance computing libraries such as PETSC for sparse and dense matrix calculation, PARMETIS for partitioning the mesh across multiple MPI processes, and the CVODE time integration tools that come with the SUNDIALS tool suite. The code is parallelized with MPI.

The code implements a dependency engine for physical fields [11, Section 4.4]. This allows lazy evaluation to only compute entities when they are truly required, and to minimize the computation of fields that depend on each others.

Periodic boundary conditions are difficult in micromagnetic simulations due to the long range nature of the magnetostatic interactions. A new computational model has been developed and implemented (the 'macro geometry' [10]),

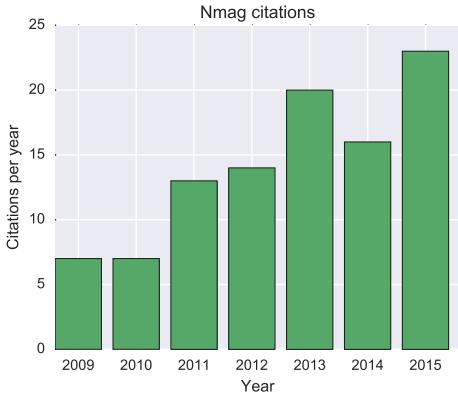


Figure 3: Nmag citations from Web of Science up to January 2016.

which has subsequently been used by the Nmag successor code Finmag [5], and other micromagnetic tools such as the GPU-based package Mumax3 [20].

**Time line:** The planning for Nmag started around 2003, funding was secured in 2005, and the work started soon afterwards. The first version was publicly released in 2007. The tool was actively maintained and further developed until January 2012, when key developers moved on and no further funding was available. Since then, the software has been hosted ‘as is’.

**Uptake:** As of February 2016, the official Nmag publication [12] has been cited 103 times in publications (Web of Science, Thomson Reuters) and 182 times on Google Scholar. More than 150 users are known by name (from the mailing list, or off-line queries), and the web site is frequented from academic and industry domain names, with 45000 visits since 2006. Figure 3 shows the breakdown of citations per year. Development and maintenance stopped early 2012, which (so far) seems to not have affected the usefulness of the tool for research.

**Nmag team:** The principal funding for the project was for a research fellow for 2.5 years. The team managed to find 3 PhD student projects that supported the lead developer as part of their work [7, 14, 16], and in the later phases there were smaller contributions from others. Later (2009–2012), a research fellow from another project contributed significantly to the project. The background of the team was in physics and mathematics. None of the participants had any formal training in software engineering, which is a common situation in the development of research software.

**Community support:** Community support involves the following tools and strategies: the Nmag webpage hosts the code, installation instructions, the manual [1], a link to the mailing list archives [3] and the Redmine Wiki. The University of Southampton is hosting the mailing list, and Google Groups is used to archive all communication in that mailing list. A Redmine server is used simply to host a Wiki that users can edit (<https://nmag.soton.ac.uk/community/wiki/nmag>).

**Software engineering process:** Parts of the software were written in an effectively plan-driven approach, broken into separate requirement analysis, design, implementation and testing phases. In particular the Nsim multi-physics core was realized as one large piece of work by the lead developer without significant subsequent change.

---

```

import nmag
from nmag import SI

# Create simulation object
sim = nmag.Simulation()

# Define magnetic material
Py = nmag.MagMaterial(
    name = 'Py', Ms = SI(1e6, 'A/m'),
    exchange_coupling = SI(13.0e-12, 'J/m'))

# Load mesh
sim.load_mesh('sphere1.nmesh.h5',
    [('sphere', Py)],
    unit_length = SI(1e-9, 'm'))

# Set initial magnetisation
sim.set_m([1, 0, 0])

# Set external field
sim.set_H_ext([0, 0, 0], SI('A/m'))

# Save and display data in a variety of ways
sim.save_data(fields='all')

# Sample demagnetisation field through sphere
for i in range(-10, 11):
    x = i * 1e-9           # x-position in metres
    H_demag = sim.probe_subfield_siv(
        'H_demag', [x, 0, 0])
    print("x =", x, ": H_demag =", H_demag)

```

---

Figure 4: Nmag end-user script example.

Other parts, in particular the Python-level micromagnetic interface `nmag` were developed in a more agile style, with multiple iterations of development, use of automated tests (section 4.10), where both refactoring and additional feature implementation was carried out in subsequent iterations.

## 4. LESSONS LEARNED

### 4.1 User interface through Python library

A key design decision was to embed the functionality of the simulation into a general purpose language, in this case Python (see also [11, section 5.11.1]).

Figure 4 shows an Nmag simulation script, which is a Python script that imports and uses the `nmag` library.

In comparison to definition of simulation configurations through configuration text files or graphical user interfaces (see for example Figure 1), this approach has a number of advantages: (i) no parser needs to be written – Python is the parser, (ii) the user has complete freedom in using Python constructs to combine the simulation commands provided by the `nmag` library as needed for the particular application, (iii) data pre- and post-processing, and calculations that take place during the simulation can make use of the Python ecosystem of available scientific libraries, (iv) the configuration of the simulation can make use of Python functions to provide, for example, initial magnetization vector field configurations, that show a complicated spatial dependence. (See also section 5.11.2 in [11]).

### 4.2 Choice of programming languages

Figure 5 shows some statistics regarding the programming languages / tools and respective lines of code in the Nmag project.

Language	files	comment lines	code lines
OCaml	174	15111	53445
Python	588	17718	49286
C	49	2548	12375
Bourne Shell	47	1232	9184
make	138	391	2831
C/C++ Header	14	410	820
SUM:	1010	37410	127941

**Figure 5: Output of `cloc (v1.65)` run on Nmag source code, tests, and documentation files.**

Nmag uses Python as the language for the user to define how the micromagnetic simulation should be carried out. It should be noted that the micromagnetic capabilities as well as the testing infrastructure and system tests are implemented at the Python level, while the OCaml code provides a generic multi-physics simulation environment and contains significant parts of the multi-physics finite element code. The OCaml multi-physics engine is capable of solving problems in areas outside micromagnetism.

Python was a good choice for the high level user-interface, and internals of the package: it is a user-friendly language [9] that has gained substantially in popularity in computational science and elsewhere since 2005; resulting in additional benefits from the critical mass of users. Anecdotal evidence and the increase of Python-based simulation tools and libraries support our point that Python is a high productivity language in computational science. In micromagnetism, packages developed after Nmag have followed the model of providing a Python library to offer their functionality [6, 5].

For Objective Caml (OCaml) as the work horse of the multi-physics engine, the situation is less clear. There are technical reasons why OCaml is a good choice, including its C interoperability interface, its (interactive) interpreter and its native code compiler. OCaml also offers automatic memory management, expressive power and functional capabilities (see section 5.13 in [11]).

In hindsight, we have identified a disadvantage of a social engineering nature: OCaml is not a wide-spread language (certainly not outside computer science and mathematics), and there are virtually no users of the Nmag software that have OCaml experience. Furthermore, OCaml is very rarely taught and due to its (powerful but somewhat non-mainstream) functional style, it presents a steep learning curve to the typical Nmag user, who tends to be a researcher in material science, engineering, physics, biology, geography and medicine (but generally not a computer scientist). For an open source project, it is important that the code is accessible by the user community to attract new contributors.

Comparison of execution performance of compiled OCaml code with C/C++ code have shown that the OCaml code can be noticeably slower than the C code. We address this point in more detail in section 4.4.

### 4.3 Python interpreter activated from OCaml

For technical reasons and the capabilities of the existing PyCaml interface [21] that connects OCaml and Python, the Nmag setup is such that the Python interpreter is called from within an OCaml executable.

In more detail: the user starts an executable called `nsim` compiled from OCaml code. This executable initializes the simulation engine, and then calls an embedded Python interpreter, which processes the user’s simulation commands (which are typically given through a file `mysim.py` to the `nsim` executable, *i.e.* the command line call would be `nsim mysim.py`). If no file to process is given, a Python interpreter is displayed in which Python commands can be entered interactively, and the `nmag` library is accessible.

An alternative setup would be that the user starts the ‘usual’ Python interpreter, and that a `nmag` package can be imported, which carries out the housekeeping work, initialization of MPI and and execution of required OCaml code when imported.

With hindsight, we suggest that the alternative arrangement would have been much preferable for a number of reasons: (i) `nsim` activates a Python interpreter (in which the `nmag` library is accessible), so it will seem more logical to the user if that command is `python` and not `nsim`. (ii) The Python interpreter coming through `nsim` is in general different from the system Python interpreter (and with the current install, see section 4.12, this Python interpreter is built from source and installed locally); and third party Python modules (such as `numpy`, `scipy`, `matplotlib`) may end up having to be installed separately for both interpreters.

### 4.4 OCaml performance

The OCaml-based multi-physics engine subdivides the computation into two phases: initialisation and time integration. In the first phase, a set of sparse and dense matrices are pre-computed and stored in memory. These matrices capture the various physical interactions that govern the behavior of the simulated system and are mainly computed by OCaml code. In the second phase, the time integration is carried out, using the pre-computed matrices. These computations are mainly done using the PETSc library. Micromagnetic simulations of real materials and devices require fine meshes corresponding to pre-computed matrices of size of tens of Gigabytes and matrix assemble times of the order of hours. It is therefore important to make the OCaml initialisation phase fast and efficient. Investigations of the OCaml engine code performance revealed some limitations of the language and its compiler when used in our context, which eventually led us to rewrite parts of the matrix initialization code in C.

First we comment on array efficiency. OCaml native arrays are unidimensional. A rank-2 tensor (*i.e.* a matrix) can still be represented as an array of arrays of floats which, however, is not memory efficient: the floats are not stored in contiguous areas of memory, requiring extra memory for storing pointers and thus additional indirections during array accesses. Moreover, the sub-arrays are not guaranteed to have the same lengths. This makes multi-dimensional arrays difficult to analyze and optimize for a compiler. While the `Bigarray` module in OCaml provides multi-dimensional arrays which are stored contiguously in memory and thus allow to overcome some of these problems, accesses to OCaml big-arrays are not inlined, leading to poor performance [15].

Second, we have found that OCaml has limited support for some compiler optimisations that are particularly useful in numerical code, such as bounds-checking elimination, loop unrolling, and vectorisation. These optimization techniques are now common in mainstream languages and are performed by freely available compilers such as `gcc` and

`clang`, which of course receive a vastly greater investment and contribution from the private software industry.

We provide electronic supplementary material [15] with sample code that underpins the results reported in this section, and additional interpretation. The examples show that rewriting an OCaml loop in C/C++ can give speedups of a factor 4. This is significant because performance critical numerical code often consist of simple for-loops.

In summary, the lack of native multi-dimensional array support is a problem for scientific code, and a better compiler would bring OCaml closer to C in terms of performance.

## 4.5 Symbolic derivation of PDE calculations at run-time

In finite elements, a mathematical problem in form of a PDE is solved in a given number of spatial dimensions  $n$ ; these are typically  $n = 3$ , or  $n = 1$ ,  $n = 2$ , corresponding to 3d space as we know it, and reduced models where a 1d or 2d space is sufficient. On these  $n$ -dimensional domains, we operate with scalars, vectors or tensors, which can have their dimensionality  $k$ . Finally, there is a variety of basis functions and (taking only continuous Galerkin elements as an example), these have their own polynomial order  $p$ .

Traditionally, the right equation for a particular mathematical operator is derived with pen and paper for particular values of the dimensions of space  $n$ , the dimensionality of the degree of freedom  $k$  and basis function order  $p$ . Once the equation has been derived, it is hard-coded as an implementation for this specific case. This code is then used to populate the finite element matrices. In the micromagnetic context, the Magpar package [19] is such an example.

Nmag's approach is different: here, the relevant analytic operations, which include differentiation and integration, are carried out symbolically (within the Nsim OCaml code base) to generate specialized instructions to compute the matrix entries for the particular operator, dimensionality, basis function order etc. that the user requires. The `nsim` code supports arbitrary basis function orders  $p$ , dimensionality of the degree of freedom  $k$  and and arbitrary dimensionality  $n$  of the domain.

This approach provides much greater flexibility to change the equations in the model (an important consideration in the context of exploratory research), or the numerical model parameters such as the order of the basis functions. It also avoids repetitive manual analytical work, and – assuming the symbolic computations are implemented correctly – reduces the chance of errors. We discuss the associated additional complexity in section 4.8.1.

## 4.6 Auto-generation of code for local field mappings

The primary entities of interest in finite element simulations are *fields*, such as the magnetization vector field, the temperature scalar field, a displacement vector field, etc. Nmag uses auto-generation of code at run time to allow the user to compute tailored expressions that map from one field to another field (details in [11, section 5.11.4]). There are two possible ways of achieving this:

(i) The user provides C code that contains the mathematical mapping operation that is required. At run time (as the user-provided string is not known before then), this user-provided C-code string is embedded into a template that provides access to the relevant field arrays and access meth-

ods, and the combined C-code is written to disk, compiled, and dynamically linked. The auto-generated code will automatically translate the user provided indices to the right memory locations, which is non-trivial for multi-physics simulations where multiple fields are defined at every node.

(ii) The user can also provide algebraic expressions which represent the required operation, which are automatically translated into C code. We provide an example to demonstrate this. In the micromagnetic problem, there is a magnetization vector field  $\mathbf{m}(\mathbf{r})$  that defines a 3d vector at every point  $\mathbf{r}$  in 3d space. In this example, we look at the mapping of the magnetization  $\mathbf{m}$  onto its time derivative  $\frac{d\mathbf{m}}{dt}$  as is necessary to compute the equation of motion (1) for this magnetization vector function  $\mathbf{m}(\mathbf{r})$ :

$$\frac{d\mathbf{m}}{dt} = c_1 \mathbf{m} \times \mathbf{H} + c_2 \mathbf{m} \times (\mathbf{m} \times \mathbf{H}) \quad (1)$$

We note that  $\mathbf{H}$  is obtained as a function of  $\mathbf{m}$  by solving certain partial differential equations [16, Section 2.2].

We can rewrite (1) using index notation as:

$$\frac{dm_i}{dt} = \sum_{j,k} \left[ c_1 \epsilon_{ijk} m_j H_k + \sum_{p,q} c_2 \epsilon_{ijk} m_j (\epsilon_{kpq} m_p H_q) \right] \quad (2)$$

For this formulation, `nsim` provides a small domain specific language. We show how equation (2) is represented as a string in this domain specific language:

```
dmdt = """%range i:3, j:3, k:3, p:3, q:3
dmdt(i) <- c1 * eps(i, j, k) * m(j) * H(k)
           + c2 * eps(i, j, k) * m(j)
           * eps(k, p, q) * m(p) * H(q)""""
```

We have found the interface (ii) useful to quickly and flexibly extend the equation of motion. The ability to specify C code directly through method (i) allows to sidestep the `nsim` framework where functionality is required that was not anticipated initially. By using C code (rather than Python code, say) good performance is achieved, in particular when loops over all degrees of freedom are involved. A very similar mechanism to method (i) is provided in the multi-physics finite element library FEniCS [17] through the `instant` module, which can be used to initialize fields with arbitrary C expressions. An approach similar to (ii) is used in FEniCS for the non-local PDE operators.

FEniCS [17] started being developed at the same time as Nsim and is now widely used, including in the micromagnetic package Magnum.fe [6] and the Nmag successor software Finmag [5]. FEniCS has core routines written in C++ and provides a Python interface.

## 4.7 Parallel execution model

The parallel execution model of Nmag is that only one Python process is running, driving slaves through MPI from within the OCaml code [13]. This allows end-users to write truly sequential Python code and to completely ignore the parallel execution of the micromagnetic equations. This is different from the FEniCS parallel model [17], where also the Python code executes in parallel. While the FEniCS model requires more thought at the Python level, it allows to add computationally demanding operations to be executed in parallel through Python. For expert users and scalability, the FEniCS model is preferable.

## 4.8 Complexity originating from generality

### 4.8.1 Arbitrary number of dimensions

As introduced in section 4.5, in finite elements, some mathematical problem is solved in a given number of spatial dimensions  $n$ ; where the `nsim` code supports calculations in arbitrary number of spatial dimensions.

While there are problems defined on space that is higher dimensional than 3d space in science and engineering, none of that functionality has been used in the lifetime of the Nmag software. The complexity of the code could have been reduced (and maintainability and accessibility increased) if we had limited its functionality to spatial dimensions  $n$  of 1, 2 and 3.

We note for context that the FEniCS [17] multi physics library follows a similar path of using symbolic calculation at run time to derive finite element matrix entries, and that the FEniCS functionality is limited to 3 or fewer dimensions.

### 4.8.2 Arbitrary high level language support

The operator notation used in (the Python) `nsim` library is representing differential operators through a string (see section 4.3.1 and example A.2 in [11]). An alternative would be to create classes in Python that represent mathematical objects and differential operators, and use operator overloading to integrate the mathematical description naturally within the Python language. The FEniCS project [17] has followed the latter path with their Python interface [18].

The motivation behind the design decision to prefer the string representation over the (object-oriented) operator representation for the differential equation was to avoid coupling the notation too tightly to the Python scripting language: by sticking to strings, one can substitute Python by another language more easily as and when required. Note that when Nmag was created Python was by far not as mainstream in scientific computing as it is today; for example, the Python numerical library `numpy` was only created in 2005.

From a user's point of view, we believe that it is preferable to use Python objects over strings to define the differential equations: this is more natural and allows to exploit auto-completion and to explore capabilities and documentation of objects when working interactively with the Python prompt.

It may have been beneficial to approach this design question in a more agile way by fully buying into the Python language and the overloaded operator notation initially and revisiting the decision as the project evolved. As a decade has passed, we now know that there is no need to introduce another high level language for Nmag: Python is doing fine, and the development of the Nmag project has stopped anyway. Another possibility would be to introduce a bridge element that translates the object oriented equations into strings when required, which would allow combining the long term strategy of expressing differential equations through strings with the ability to offer operator overloading in Python to the user.

### 4.8.3 Multi-physics capabilities

Nmag is built on the multi-physics library Nsim, and thus the micromagnetic model that is available through Nmag is only one of many possible types of PDE-based simulations that could be built on top of `nsim`. The multi-physics capabilities were indeed unique in the micromagnetic simulation landscape, but bring significant additional complexity.

While we attempted to immediately implement a multi-physics framework, and then build a micromagnetic simula-

tion on top of this, it may have been more efficient to build a micromagnetic prototype first, and use the experience gathered with this in developing a more generic framework.

We note that as the multi-physics capabilities were not part of the funded research program, they have not been developed and documented to the level of the core micromagnetic functionality, and are thus not used widely.

### 4.8.4 Associating numbers with units

All Nmag input physical entities have to be expressed as a product of a number and product of powers of dimensions (such as m, kg, s, A, K, mol, cd). For example, the Python expression `x = SI(100e-9, "m")` describes a length of 100 nano meters. For multi-physics simulations, this provides great advantages as the output fields emerging from user-defined operations automatically carry the right units. It also allows to scale numbers internally (to reduce chances of overflow etc), and to use arbitrary unit systems (such as SI, CGS, or custom). In the context of micromagnetics – Nmag's application domain – the capability met a mixed reception: some users find it useful, others dislike the overhead of having to type the dimensions, when they normally use SI units (as is enforced in the OOMMF [8] package).

## 4.9 Version control and source

Development of Nmag started in 2005 and has used three different version control systems through its history: starting with CVS, before changing to Subversion, before switching to Mercurial in 2010. Initially, the source code was included in the tarballs containing the 'source' releases of the project, and later the Mercurial repositories have been made available on Bitbucket [2]. There are separate repositories for the source, the tests, the documentation, the webpage, the distribution, and one additional meta repository that provides a script to clone them all together into the right relative subdirectory structure. Use of version control tools is essential. The distributed tools, such as Mercurial and Git, are more flexible than CVS and Subversion.

## 4.10 Testing

There about 75 tests coming with the code, combining unit tests (of only some parts of the code), with a fair number of system tests, and a few regression tests. While undoubtedly more unit tests would have been desirable, the availability of the existing tests is extremely useful as a first indicator of a working installation, etc. Test-driven development was not generally used for the project.

## 4.11 Documentation

In addition to documentation in the source code files, there is the official Nmag documentation that is available [1] as html or pdf (183 pages). Key components include a tutorial-like, step-by-step introduction and walk-through of Nmag, starting from simple and common use cases to more complicated and specialized application examples (112 pages), explanation of general concepts (7 pages), a command reference, which is built from the Python documentation strings defined in the source code (20 pages), an overview of executable scripts including their options and usage examples, and Nmag data file types (12 pages), a list of 20 frequently asked questions and answers (9 pages), a mini-tutorial into micromagnetic modeling (not specific for Nmag, but experience shows that new Nmag users are also

often new to the field). The documentation is often cited by users as being very good, and useful.

## 4.12 Installation

Nmag depends on a large number of support libraries. Many of these are of scientific origin and change often, including changes in the interface. As a result, the Nmag code needs updating to compile correctly after any such update.

For some time, there were installation options through a Debian package, a KNOPPIX Live Linux CD (this was before virtual machines were widely used), and from source.

Later, we focused our energy on providing an installation setup that is as independent as possible from version changes of libraries that are installed through the Linux distribution to maximize the chances for long term availability in a situation without maintenance resources.

The resulting setup compiles and installs all the required dependencies<sup>1</sup> from source, then builds Nmag based on these support libraries. Compilation does only work on Linux (although on a large variety of distributions and releases).

This setup has been remarkable robust, and only failed once in 4 years of no maintenance updates. The one failure was due to more recent gcc versions reporting commenting through a double slash in C header files of the hdf5 library as illegal, which previously only triggered a warning.

The price to pay for this robustness is that the compressed tarball containing all the library source code is about 91 MB in size. Once Nmag has compiled all the support libraries, these take together 1.4 GB of disk space storage. After removing temporary build files 0.5 GB remain. As the support libraries are snapshots from their source distribution in early 2012, no improvements or bug fixes developed in the support libraries will affect the Nmag compilation and executables.

It would have been useful to use continuous integration (to run tests, build documentation and releases) to support more frequent releases.

*Acknowledgments:* We acknowledge financial support from the UK's Engineering and Physical Sciences Research Council (EPSRC) from grants EP/E040063/1, EP/E039944/1 and Doctoral Training Centre EP/G03690X/1), from the European Community's FP7 Grant Agreement no. 233552 (DYNAMAG), from Horizon 2020 Research Infrastructures project #676541 (OpenDreamKit), and from the University of Southampton, all of which contributed in part to the development of the Nmag tool suite and this review.

## 5. REFERENCES

- [1] Nmag code. <http://nmag.soton.ac.uk>.
- [2] Nmag code on Bitbucket.  
<https://bitbucket.org/fangohr/nmag-meta>.
- [3] nmag-users mailing list archive.  
<https://groups.google.com/d/forum/nmag-users>.
- [4] Ocaml. <http://ocaml.org>.
- [5] Finmag — FEniCS-based micromagnetic simulation tool. 2012–2016. University of Southampton, UK.
- [6] C. Abert, L. Exl, F. Bruckner, A. Drews, and D. Suess. magnum.fe: A micromagnetic finite-element simulation code based on FEniCS. *Journal of Magnetism and Magnetic Materials*, 345:29–35, 2013.
- [7] G. Bordignon. *Simulations of ferromagnetic nano-structures*. PhD thesis, University of Southampton, 2008.  
<http://eprints.soton.ac.uk/65634/>.
- [8] M. Donahue and D. Porter. OOMMF User's Guide, Version 1.0, Interagency Report NISTIR 6376. Technical report, National Institute of Standards and Technology, Gaithersburg, MD (US), September 1999.
- [9] H. Fangohr. A comparison of C, MATLAB, and Python as teaching languages in engineering. In *Computational Science-ICCS 2004*, pages 1210–1217. Springer, 2004.
- [10] H. Fangohr, G. Bordignon, M. Franchin, A. Knittel, P. A. de Groot, and T. Fischbacher. A new approach to (quasi) periodic boundary conditions in micromagnetics: The macrogeometry. *Journal of Applied Physics*, 105(7):07D529, 2009.
- [11] T. Fischbacher and H. Fangohr. Continuum multi-physics modeling with scripting languages: the Nsim simulation compiler prototype for classical field theory. *arXiv*, abs/0907.1587, 2009.  
<http://arxiv.org/abs/0907.1587>.
- [12] T. Fischbacher, M. Franchin, G. Bordignon, and H. Fangohr. A systematic approach to multiphysics extensions of finite-element-based micromagnetic simulations: Nmag. *Magnetics, IEEE Transactions on*, 43(6):2896–2898, 2007.
- [13] T. Fischbacher, M. Franchin, G. Bordignon, A. Knittel, and H. Fangohr. Parallel execution and scriptability in micromagnetic simulations. *Journal of Applied Physics*, 105(7):07D527, 2009.
- [14] M. Franchin. *Multiphysics simulations of magnetic nanostructures*. PhD thesis, University of Southampton, 2009.  
<http://eprints.soton.ac.uk/161207/>.
- [15] M. Franchin, M. Albert, and H. Fangohr.  
<https://github.com/fangohr/paper-supplement-ocaml-performance>, DOI: 10.5281/zenodo.45235.
- [16] A. Knittel. *Micromagnetic simulations of three dimensional core-shell nanostructures*. PhD thesis, University of Southampton, 2011.  
<http://eprints.soton.ac.uk/333186>.
- [17] A. Logg, K.-A. Mardal, G. N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012.
- [18] A. Logg and G. N. Wells. DOLFIN: Automated Finite Element Computing. *ACM Transactions on Mathematical Software*, 37(2), 2010.
- [19] W. Scholz, J. Fidler, T. Schrefl, D. Suess, R. Dittrich, H. Forster, and V. Tsiantos. Scalable Parallel Micromagnetic Solvers for Magnetic Nanostructures. *Computational Materials Science*, 28:366–383, 2003.
- [20] A. Vansteenkiste, J. Leliaert, M. Dvornik, M. Helsen, F. Garcia-Sanchez, and B. Van Waeyenberge. The design and verification of mumax3. *AIP Advances*, 4(10), 2014.
- [21] A. Yerkes, T. Fischbacher, H. Fangohr, and B. Schwartz.  
<https://github.com/chemoelectric/pycaml>.

<sup>1</sup>atlas-3.6.0, cryptokit-1.2, gsl-1.14, ipython-0.1, lapack, mpich2-1.2.1p1, numarray-1.5.2, numpy-1.5.0, ocaml-3.12.0, ocamlgsl-0.6.0, ocaml-findlib-1.2.1, ParMetis-3.1.1, Python-2.7.2.tar.bz2, petsc-lite-3.1-p5, PyVTK, qhull-2003.1, tables-2.1.2, scipy-0.7.2 py-0.9.1 ply-3.3, sundials-2.3.0, HLib-1.3p19