Software Engineering for Computational Science

Lessons learned from the Nmag project

Hans Fangohr, Maximilian Albert, Matteo Franchin

2016-05-16

University of Southampton, United Kingdom

SE4Science, ICSE2016, Austin, Texas (US)

Outline

Context: computational micromagnetics

Nmag project

Architecture and languages

Ocaml performance

User interface / configuration files

Automatic code generation

Parallel execution model

Version control, tests, continuous integration

Dissemination

Complexity

Conclusions and general recommendations

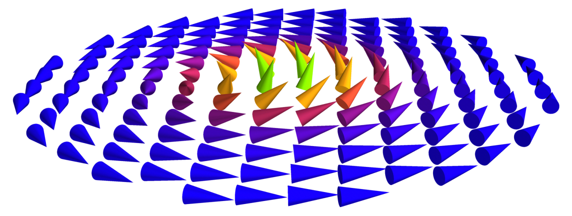
Context: computational micromagnetics

The science

- research area *computational micromagnetics*
- physics: multiple time scales (10^{-12}s to 10^{-8}s)
- physics: multiple length scales (10nm to 10,000nm)
- model: magnetization is continuous 3d-magnetisation vector field
- mathematics: time dependent integro partial differential equation (PDE)
- numerical solution:
    - semi-discretize time dependent PDE
    - finite elements/finite difference for spatial operators
    - stiff coupled system of $\sim 10^6$ ordinary differential equations
- applications: magnetic data storage, sensors, electromagnetic wave generation, spintronics

The community

- community of several thousand researchers, ∼500 papers making use of computational micromagnetics every year
- from academia and industry
- mostly physicists, material scientists, engineers
- use simulation to interpret and design
  - experiments
  - devices



Equilibrium configuration of magnetisation vectorfield in disk geometry.

Simulation tool status 2003: OOMMF

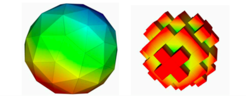
- Only one simulation code (1st release Oct 1998):
- Object Oriented MicroMagnetic Framework (OOMMF) `http://math.nist.gov/oommf/`

The Object Oriented MicroMagnetic Framework (OOMMF) project at ITL/NIST

Background on the ITL/NIST micromagnetics public code project

- finite difference space discretization
- C++ core routines, Tk/Tcl interface

Nmag project



- need a finite *element* based alternative to the finite *difference* based OOMMF code
- supporting multi-physics (magnetism + X) would be desirable
- Software name turns out to be NMAG. Possible meanings:
    - NanoMAGnetic ...
    - *n*-mag, where *n* symbolises *multiple* types of physics
- Homepage: `http://nmag.soton.ac.uk`

Nmag Time line and Team

Time line

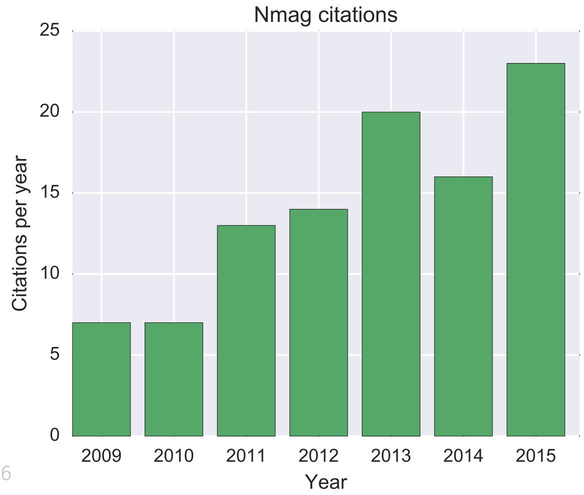
- 2003 initial plan
- 2005 funding secured (post-doc for 2 years)
- 2007 first release as open source
- 2012 actively maintenance stops

Team

- post-doc (theoretical physics)
- 2-3 PhD students (one of them [Matteo Franchin] carried on as a post-doc maintaining Nmag 'in his spare time' until 2012)
- investigator

Uptake (as of Friday 13 May 2016)

- users in academia and industry, ∼150 known by name
- 113 citations on Web of Science, 194 on Google scholar,

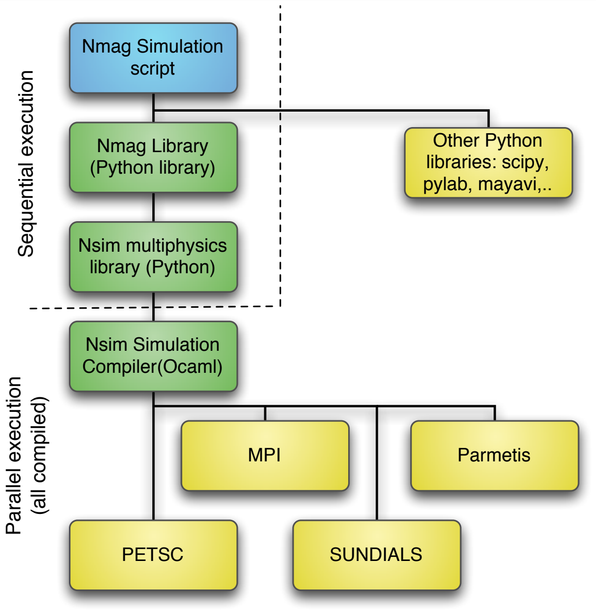


Impact

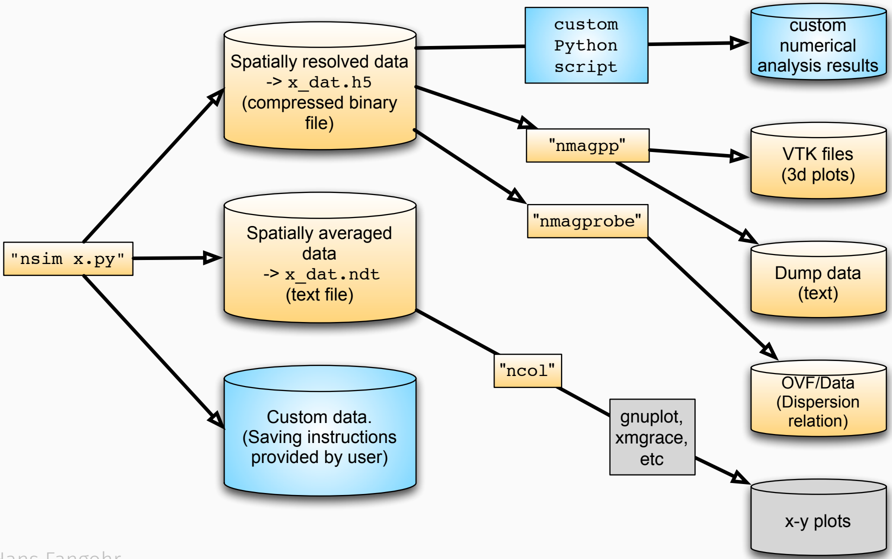
1. Ongoing (increasing??) use in research and development
2. Design influenced other micromagnetic packages
3. Flexible Open Source FE micromagnetic tool provides useful data point for ”micromagnetic standard problems”:
    - Micromagnetic standard problems are essentially systems tests with well defined input and simulation parameters
    - Used to evaluate new tools
    - Examples
        - Journal of Applied Physics 105, 113914 (2009)
        - IEEE Transactions on Magnetics 49, 524-529 (2013)
        - `http://arxiv.org/abs/1603.05419` (2016)

Architecture and languages

Outcome - overview



Outcome - data output



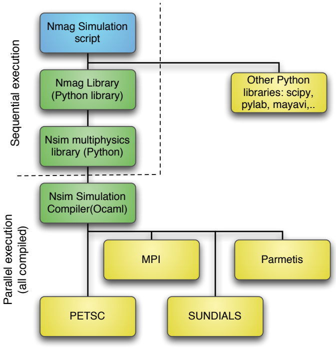
Outcome - lines of code

Output from CLOC (Count Lines Of Code)

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

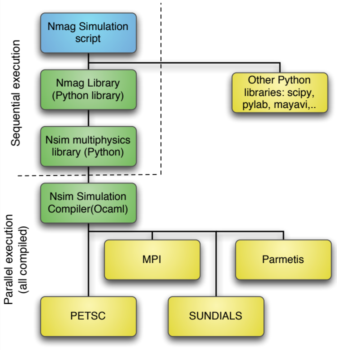
Architecture and language discussion

- Python (further discussion later)
  - user interface to please scientists
  - interactive (interpreted)
- Objective Caml
  - for complicated multi-physics finite element code
  - compiled with static types (→ fast)
  - no pointers
  - well defined C interface
  - power of functional language (suits symbolic operations)
  - team member was familiar with it
- Computational library re-use



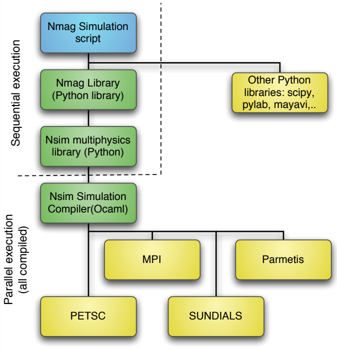
Architecture and language discussion

- Python (further discussion later) ✓
  - user interface to please scientists
  - interactive (interpreted)
- Objective Caml
  - for complicated multi-physics finite element code
  - compiled with static types (→ fast)
  - no pointers
  - well defined C interface
  - power of functional language (suits symbolic operations)
  - team member was familiar with it
- Computational library re-use



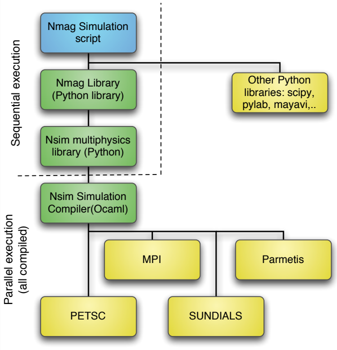
Architecture and language discussion

- Python (further discussion later) ✓
  - user interface to please scientists
  - interactive (interpreted)
- Objective Caml
  - for complicated multi-physics finite element code
  - compiled with static types ($\rightarrow$ fast)
  - no pointers
  - well defined C interface
  - power of functional language (suits symbolic operations)
  - team member was familiar with it
- Computational library re-use ✓



Architecture and language discussion

- Python (further discussion later) ✓
  - user interface to please scientists
  - interactive (interpreted)
- Objective Caml ?
  - for complicated multi-physics finite element code
  - compiled with static types ($\rightarrow$ fast)
  - no pointers
  - well defined C interface
  - power of functional language (suits symbolic operations)
  - team member was familiar with it
- Computational library re-use ✓



Was Objective Caml (OCaml) the right choice?

- Objective Caml has no acceptance in user community (user community are physicists, engineers, material scientists):
  - can easily learn Python
  - can learn C if truly required
  - tend not to touch Objective Caml:
    - have never heard of it
    - feels unusual if grown up with imperative or OO language
- Thus, no buy-in from community into this part of the code
- Not good for an open source project

Conclusion

Objective Caml was a unsuitable choice for *social* reasons.

Objective Caml (OCaml) example

From "Learn OCaml":

Polymorphism: sorting lists

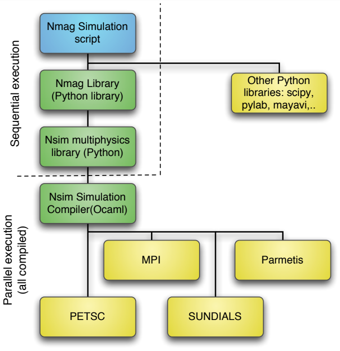
Insertion sort is defined using two recursive functions.

```
let rec sort = function
    | [] -> []
    | x :: l -> insert x (sort l)
  and insert elem = function
    | [] -> [elem]
    | x :: l -> if elem < x then elem :: x :: l
                else x :: insert elem l;;
```

Ocaml performance

Language performance results and discussion

- Baseline given by C or Fortran
- Naive Python is about 100 times slower
    - that's why we use Python only for the interface and coordination of computing flows
    - the computing intense operations are all in compiled code
        - C, C++, OCaml
- OCaml performance
    - OCaml code is strongly typed
    - compiler knows types at compile time, and can produce fast code
    - *A priori* not clear why this should be slower than C-speed



OCaml performance results

Result:

OCaml code slower than C-code (factor 4 in recent tests[1])

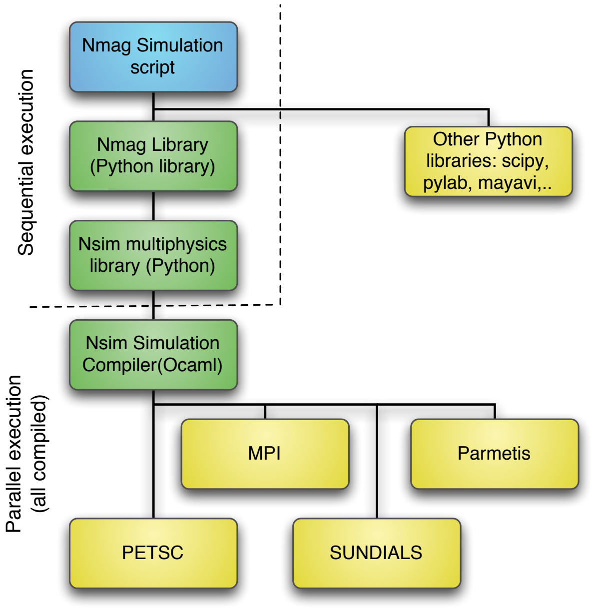
Two-reasons:

- multi-dimensional arrays are not well supported
  - can have arrays of arrays to represent matrix, but allocated memory is not contiguous
  - subarrays are not guaranteed to have the same length, making optimisation for the compiler difficult
  - the `Bigarray` module addresses these shortcomings, but `Bigarray` access is not inlined
- bounds-checking elimination, loop unrolling, and vectorisation not supported by OCaml compiler

[1] `https://github.com/fangohr/paper-supplement-ocaml-performance`

User interface / configuration files

Reminder



Sequential execution
Nmag Simulation script
Nmag Library (Python library)
Nsim multiphysics library (Python)
Other Python libraries: scipy, pylab, mayavi,..
Parallel execution (all compiled)
Nsim Simulation Compiler(Ocaml)
MPI
Parmetis
PETSC
SUNDIALS

Simulation configuration

Common requirement:

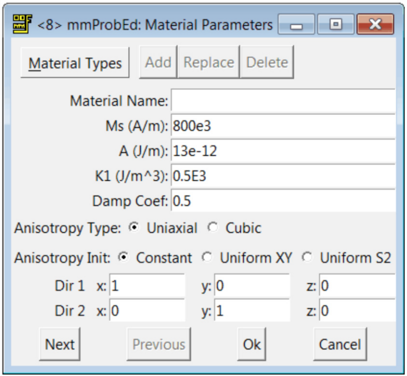
- re-use large code base with some run-specific variations:

  (particular types of physics, force-field, equation-of-motion, dimensionality, discretisation, number of particles, applied field, temperature, model assumption, …)

Common approaches to managing these 'configurations':

1. recompile code for every run
2. Graphical User Interface to set parameters manually for every run (OOMMF)
3. Main simulation tool reads configuration file (OOMMF)
4. Simulation configuration is executable script, that uses simulation package as a library (Nmag)

Starting Point: OOMMF interface (GUI)



Starting Point: OOMMF interface, Tcl configuration file

```
# MIF 2.1
Specify Oxs_BoxAtlas:atlas {
  xrange {0 30e-9}
  yrange {0 30e-9}
  zrange {0 100e-9} }
Specify Oxs_RectangularMesh:mesh {
  cellsize {2.5e-9 2.5e-9 2.5e-9}
  atlas :atlas}
Specify Oxs_UniformExchange {A  13e-12}
Specify Oxs_Demag {}
Specify Oxs_UZeeman "Hrange { { 0.5e6 0 0 0.5e6 0 0 0 } }"
Specify Oxs_EulerEvolve {
  alpha 0.5
  start_dm 0.0001
  gamma_G 0.2211e6
  absolute_step_error 0.02
  relative_step_error 0.02}
Specify Oxs_TimeDriver {
 basename test
 evolver Oxs_EulerEvolve
 stopping_dm_dt 0.01
 mesh :mesh
 stage_count 1
 stage_iteration_limit 550000
 total_iteration_limit 1000
 Ms { Oxs_UniformScalarField { value 0.86e6 } }
 m0 { Oxs_UniformVectorField {
  norm 1
  vector {1 0 1}
 } } }
Destination archive mmArchive
Schedule DataTable archive Step 1
Schedule Oxs_TimeDriver::Magnetization archive Stage 500
```

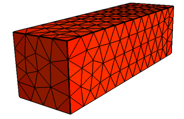
Outcome: make simulation tools a python package

```
import nmag
from nmag import SI

mat_Py = nmag.MagMaterial(
    name="Py",
    Ms=SI(0.86e6, "A/m"),
    exchange_coupling=SI(13.0e-12, "J/m"),
    llg_damping=0.5)

sim = nmag.Simulation()
sim.load_mesh("bar.nmesh.h5", [("Py", mat_Py)],
              unit_length=SI(1e-9,"m"))

sim.set_m([1, 0, 1])
sim.save_data(fields='all')
target_time = sim.advance_time(SI(100e-12, "s"))
sim.save_data(fields='all')
```



bar.nmesh.h5

Why have nmag as a library?

1. most flexible model:
    - user writes a generic Python *program* using commands from the `nmag` library:
    - can include for-loops, functions, reading/writing data, …
    - designers don't have to anticipate use cases
2. supports reproducibility: simulation setup is flexible but fully contained in one file
3. saves work in comparison to config file approach:
    - no need to invent a 'configuration file language' and parser

Why use Python as the user interface language

- very high level language (fewer lines $\rightarrow$ fewer bugs)
- large eco-system of scientific tools
- supports procedural, OO and functional programming
- *socially acceptable* and considered *easy to learn* by user community [2]

Could we do better?

No.

[2] H. Fangohr, "A Comparison of C, MATLAB, and Python as Teaching Languages in Engineering", Lecture Notes in Computer Science Volume 3039, pp 1210-1217 (2004)

Why use Python as the user interface language

- very high level language (fewer lines $\rightarrow$ fewer bugs)
- large eco-system of scientific tools
- supports procedural, OO and functional programming
- *socially acceptable* and considered *easy to learn* by user community [2]

Could we do better?

No. (Well, yes, could do a little better: Ocaml in Python, or Python in Ocaml $\rightarrow$ ask)

[2] H. Fangohr, "A Comparison of C, MATLAB, and Python as Teaching Languages in Engineering", Lecture Notes in Computer Science Volume 3039, pp 1210-1217 (2004)

Automatic code generation

To achieve flexibility regards the used equations and high performance:

- User provides equations symbolically
- package generates
- and compiles C code
- at run time

Automatic code generation Example

Equation of motion for magnetisation

- magnetization vector field $\mathbf{m}(\mathbf{r})$ defines a 3d vector at every point $\mathbf{r}$ in 3d space
- solving PDEs in every time step results in field $\mathbf{H}(\mathbf{m})$
- compute time derivative $\frac{\mathrm{d}\mathbf{m}}{\mathrm{d}t}$ that depends on $\mathbf{m}$ and $\mathbf{H}$:

$$\frac{\mathrm{d}\mathbf{m}}{\mathrm{d}t} = c_1 \mathbf{m} \times \mathbf{H} + c_2 \mathbf{m} \times (\mathbf{m} \times \mathbf{H}) \tag{1}$$

- We can rewrite (1) using index notation as:

$$\frac{\mathrm{d}m_i}{\mathrm{d}t} = \sum_{j,k} \left[c_1 \epsilon_{ijk} m_j H_k + \sum_{p,q} c_2 \epsilon_{ijk} m_j (\epsilon_{kpq} m_p H_q) \right] \tag{2}$$

Automatic code generation Example (continued)

- Repeat of last equation (2):

$$\frac{\mathrm{d}m_i}{\mathrm{d}t} = \sum_{j,k} \left[c_1 \epsilon_{ijk} m_j H_k + \sum_{p,q} c_2 \epsilon_{ijk} m_j (\epsilon_{kpq} m_p H_q) \right]$$

- Express this in small *domain specific language* (DSL) that `nsim` provides:

```
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)"""
```

(actually a string in a Python program)

Automatic code generation - discussion

Benefits:

- high flexibility – addresses research environment requirements
- high execution performance

Disadvantages:

- greater complexity of code & up-front investment
- dynamic linking not always available (for example CrayOS on HECToR supercomputer in the UK a few years back)
- installation harder (need C-compiler *at run time*)

Was it worth the effort?

Yes(-ish). It was interesting.

Automatic code generation - discussion

Benefits:

- high flexibility – addresses research environment requirements
- high execution performance

Disadvantages:

- greater complexity of code & up-front investment
- dynamic linking not always available (for example CrayOS on HECToR supercomputer in the UK a few years back)
- installation harder (need C-compiler *at run time*)

Was it worth the effort?

Yes(-ish). It was interesting.

Could we have done better?

A little: embed DSL in Python

Automatic Code Generation: Related finite element packages

FEniCS [3] and *Firedrake* [4]

- provide similar functionality as was required from `nsim`
- actively developed

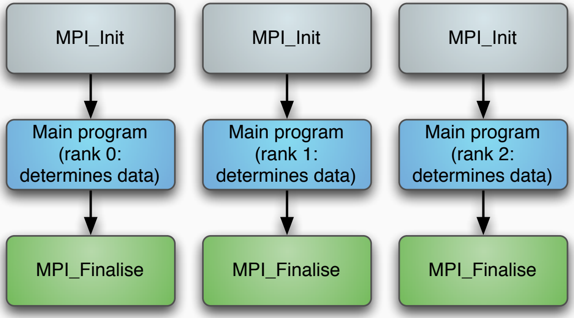
Similar designs to `nsim`:

- compiled computational core (C++)
- Python high level interface
- compilation of specialised code at run-time

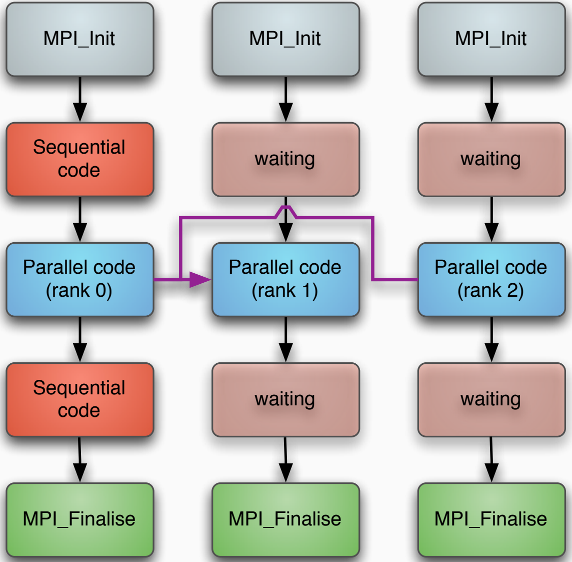
[3] `http://fenicsproject.org`
[4] `http://firedrakeproject.org`

Parallel execution model

Conventional MPI execution model



Nmag MPI execution model (”Master-Slave”)



Discussion Master Slave model

Benefits:

- End user runs truly sequential Python program
    - user does not need to know about parallelism

Disadvantages:

- for master-slave communication, we need to invent additional language, providing additional complexity
- doesn't scale for very large number of parallel processes
    - master process and user's serial Python code can become bottleneck

Lessons learned:

- avoid master-slave model for large process numbers & expert users

For Nmag, parallel execution model preference not clear: user community generally runs simulations on desktop machines, or not highly parallel.

Version control, tests, continuous integration

Version control

- CVS in 2005
- Soon(-ish) switch to SVN (SVN first release was in 2004)
- 2010 switch to Mercurial

Importance of version control?
Very high

Tests

There are some unit test and (mostly) system tests:

Makefile name	Comment	how many
check	basic system tests	56
slowcheck	slow runs	6
mpi	MPI tests	8
hlib	matrix compression tests	2
Total		72

What could we have done better?

- more unit tests
- systematic test coverage
- ideally test-driven-development

Continuous integration and release

- no continuous integration (i.e. no Jenkins / Travis CI /...)
- build process of release versions was only fully automized by the end of the active project

Could we have done better?

- Yes - automate everything, release often.
- Without automation of release, every release seems to be a major effort.

Dissemination

Documentation and Tutorial

- significant effort went into the documentation

  `http://nmag.soton.ac.uk/nmag/`
- developed by experienced teacher
- created tutorial (”Guided Tour”), that introduces
  - Nmag simulation software

    `http://nmag.soton.ac.uk/nmag/current/manual/html/guided_tour.html`
  - Mini tutorial micromagnetic modelling

    `http://nmag.soton.ac.uk/nmag/current/manual/html/tutorial/doc.html`

New users of the software are often new to the field (PhD students).

User support

- mailing list
  - hosted by University of Southampton, use Google groups for searchable archives

    `https://groups.google.com/forum/#!forum/nmag-users`
- support email (`nmag@soton.ac.uk`)
- Wiki pages (hosted by Redmine instance on Southampton server)

  `https://nmag.soton.ac.uk/community/wiki/nmag`

Software installation

- Complicated software stack:
    - Objective Caml, integrated Python interpreter
    - compilation of C code *at runtime*
    - use of libraries that can be challenging to install on their own (PETSc, MPI, Metis, CVODE/Sundials)
- Solutions
    - Debian packages
    - live-CD (Knoppix)
    - virtual machine images (vmware at the time)
    - install from source
        - $\sim$ 95MB tarball including all dependencies
        - needs 1GB space to compile, 500MB after compile
        - works on Linux (in the past also on OS X)
        - Not pretty but *very robust*

Complexity

Supporting calculations in arbitrary numbers of dimensions

- `Nsim` finite element library works in *arbitrary* number d of dimensions,
  - including $d = 1$, $d = 2$, $d = 3$ which have immediate use cases
  - but also $d = 11$, $d = 12$, $d = 42$ or any other $d \in \mathbb{N}$
- pretty complex generic code
- was never needed beyond three spatial dimensions 3d

Could we have done better?

Yes - support only $d = 1, 2$ and $d = 3$.

Complexity discussion

- attempt to support computation in arbitrary number of dimensions
- attempt to invent Python-independent domain-specific framework to support arbitrary high level language interfaces in the future
- other novel features used by very relatively few groups (ask)

Lesson

Whenever the word *arbitrary* comes up, it is worth asking:

- do we really need this, and
- do we need it now?

Conclusions and general recommendations

Recommendations primarily affecting end-users

Recommendations primarily affecting end-users

1. Embedding simulation into existing programming language provides unrivaled flexibility
2. Python is a popular language that is perceived to be easy to learn by (non-computer) scientists
3. Documentation and tutorials are important

Recommendations primarily affecting developers

Recommendations primarily affecting developers

1. Version control tool use is essential
2. System tests are essential, unit tests are very useful
3. Continuous integration is very useful
4. Limit the supported or anticipated functionality to minimize complexity and enhance maintainability

---

5. Choice of unconventional programming language can limit the number of scientists joining the project as developers
6. Code generation based on user provided equations is up-front investment but widens applicability of tool
7. OCaml not quite as fast as C/C++/Fortran

Acknowledgements

- UK's Engineering and Physical Sciences Research Council (EPSRC) from grants EP/E040063/1, EP/E039944/1 and Doctoral Training Centre EP/G03690X/1)
- European Community's FP7 Grant Agreement no. 233552 (DYNAMAG)
- European Community's Horizon 2020 Research Infrastructures project #676541 (OpenDreamKit)
- the University of Southampton

Paper (in proceedings)

Hans Fangohr, Max Albert, Matteo Franchin, *Nmag micromagnetic simulation tool – software engineering lessons learned*

1. pdf at `http://arxiv.org/abs/1601.07392`
2. code (ref [15] in paper) at `https://github.com/fangohr/paper-supplement-ocaml-performance`

Appendix

Appendix: Python calls OCaml or Ocaml calls Python?

Nmag project combines Objective Caml (OCaml) with Python code. Two options:

1. Start Python interpreter and call OCaml code from Python
2. Start OCaml programme and which starts embedded Python interpreter session

Gone with 2, but was mistake: Python interpreter that is embedded in OCaml executable is provided with Nmag source code. Thus:

- Python libraries installed on 'system' (or other) Python, not available in OCaml-Python
- users call script `x.py` with name of OCaml executable `nsim`, i.e. "`nsim x.py`", whereas "`python x.py`" would be more intuitive when `x.py` is a Python file

Appendix: Why express the equation in a string?

- rephrase: why not embed equation presentation in Python as we use Python anyway (i.e. like `sympy`)
- Ambition was to support *arbitrary* high level language for user interface
  - be prepared for the day when Python becomes unfashionable

Could we have done better?

Yes:

- Embed DSL (initially) in Python; worry about generality later
- would allow interactive exploration of symbolic equations, existing methods, documentation strings etc

Appendix: OCaml example

```
# let rec sort = function
    | [] -> []
    | x :: l -> insert x (sort l)
  and insert elem = function
    | [] -> [elem]
    | x :: l -> if elem < x then elem :: x :: l
                else x :: insert elem l;;

(* Interpreter responds with:
val sort : 'a list -> 'a list = <fun>
val insert : 'a -> 'a list -> 'a list = <fun>
*)

# sort [2; 1; 0];;
- : int list = [0; 1; 2]
# sort ["yes"; "ok"; "sure"; "ya"; "yep"];;
- : string list = ["ok"; "sure"; "ya"; "yep"; "yes"
```