

# A full-fledged micromagnetic code in less than 70 lines of NumPy

Claas Abert<sup>\*1</sup>, Florian Bruckner<sup>1</sup>, Christoph Vogler<sup>2</sup>, Roman Windl<sup>1</sup>, Raphael  
Thanhoffer<sup>1</sup>, and Dieter Suess<sup>1</sup>

<sup>1</sup>*Christian Doppler Laboratory of Advanced Magnetic Sensing and Materials, Institute of Solid State  
Physics, Vienna University of Technology, Austria*

<sup>2</sup>*Institute of Solid State Physics, Vienna University of Technology, Austria*

December 1, 2014

## Abstract

We present a complete micromagnetic finite-difference code in less than 70 lines of Python. The code makes largely use of the NumPy library and computes the exchange field by finite differences and the demagnetization field with a fast convolution algorithm. Since the magnetization in finite-difference micromagnetics is represented by a multi-dimensional array and the NumPy library features a rich interface for this data structure, the presented code is a good starting point for the development of novel algorithms.

*Keywords: micromagnetics, finite-difference method, Python*

---

<sup>\*</sup>claas.abert@tuwien.ac.at

# 1 Introduction

Micromagnetic simulations have become an integral tool for the investigation of ferromagnetic nano structures. A lot has been published on algorithms and programming paradigms used for the numerical solution of the micromagnetic equations and a variety of open and closed source micromagnetic codes is available. These codes can be roughly divided into those acting on regular cuboid grids [1, 2, 3] and those acting on irregular grids [4, 5, 6, 7, 8, 9]. Irregular-grid codes usually employ finite-elements or fast-multipole methods for spatial discretization [10]. A popular class of regular grid methods applies the finite-difference method for the computation of the exchange field and a fast convolution for the computation of the demagnetization field [11]. In this work we present a complete micromagnetic code of the latter kind that is written in only 70 lines of Python and makes largely use of the NumPy library [12].

The presented code is not able to compete with mature finite-difference codes in terms of performance and flexibility. However, it delivers all essential building blocks of a micromagnetic code and is therefore perfectly suited for prototyping new micromagnetic algorithms. In particular, the NumPy library is a good choice for the presented code since the magnetization in finite-difference micromagnetics is represented by an  $n$ -dimensional array and the NumPy library has a very powerful interface for  $n$ -dimensional arrays that supports a large variety of operations.

The paper is structured as follows. Section 2 gives a brief overview over the micromagnetic model. In Sec. 3-5 the implementation of the integral micromagnetic subproblems is described. The presented code is validated by numerical experiments in Sec. 6.

## 2 Micromagnetic Model

The central equation of dynamic micromagnetics is the Landau-Lifshitz-Gilbert equation that describes the motion of a continuous magnetization configuration  $\mathbf{m}$  in an effective field  $\mathbf{H}_{\text{eff}}$

$$\frac{\partial \mathbf{m}}{\partial t} = -\frac{\gamma}{1+\alpha^2} \mathbf{m} \times \mathbf{H}_{\text{eff}} - \frac{\alpha\gamma}{1+\alpha^2} \mathbf{m} \times (\mathbf{m} \times \mathbf{H}_{\text{eff}}) \quad (1)$$

where  $\gamma$  is the reduced gyromagnetic ratio and  $0 \leq \alpha \leq 1$  is a dimensionless damping constant. The effective field  $\mathbf{H}_{\text{eff}}$  is given by the negative variational derivative of the free energy

$$\mathbf{H}_{\text{eff}} = -\frac{1}{\mu_0 M_s} \frac{\delta U}{\delta \mathbf{m}} \quad (2)$$

where  $\mu_0$  is the magnetic constant and  $M_s$  is the saturation magnetization. Contributions to the effective field usually include the demagnetization field, the exchange field, the external Zeeman field and terms describing anisotropic effects. In this work we focus on the numerical computation of the demagnetization field, see Sec. 3, and the exchange field, see Sec. 4 as well as the integration of the Landau-Lifshitz-Gilbert equation, see Sec. 5.

For the numerical solution of (1) and (2) a regular cuboid grid is used for the spatial discretization. Every simulation cell is of size  $\Delta r_1 \times \Delta r_2 \times \Delta r_3$  and can be addressed by a multiindex  $\mathbf{i} = (i_1, i_2, i_3)$ . All spatially varying quantities such as the magnetization  $\mathbf{m}$  are thus represented by an  $n$ -dimensional array

$$\mathbf{m}(\mathbf{r}) \approx \mathbf{m}_{\mathbf{i}}. \quad (3)$$

The Python library NumPy provides the class `ndarray` for this purpose that supports a large number of operations. Despite Python being a scripting language, all collective operations of `ndarray` have a good performance due to the native implementation of the NumPy library.

### 3 Demagnetization Field

The demagnetization field accounts for the dipole-dipole interaction of the elementary magnets. For a continuous magnetization configuration the demagnetization field is given by

$$\mathbf{H}_{\text{demag}}(\mathbf{r}) = M_s \int_{\Omega} \tilde{\mathbf{N}}(\mathbf{r} - \mathbf{r}') \mathbf{m}(\mathbf{r}') d\mathbf{r}' \quad (4)$$

$$\tilde{\mathbf{N}}(\mathbf{r} - \mathbf{r}') = -\frac{1}{4\pi} \nabla \nabla' \frac{1}{|\mathbf{r} - \mathbf{r}'|} \quad (5)$$

where  $\tilde{\mathbf{N}}$  is called demagnetization tensor. This expression has the form of a convolution of the magnetization  $\mathbf{m}$  with the matrix valued kernel  $\tilde{\mathbf{N}}$ . By choice of a regular grid this convolution structure can also be exploited on the discrete level

$$\mathbf{H}_i = M_s \sum_j \tilde{\mathbf{N}}_{i-j} \mathbf{m}_j \quad (6)$$

$$\tilde{\mathbf{N}}_{i-j} = \frac{1}{\Delta r_1 \Delta r_2 \Delta r_3} \iint_{\Omega_{\text{cell}}} \tilde{\mathbf{N}} \left( \sum_k (i_k - j_k) \Delta r_k \mathbf{e}_k + \mathbf{r} - \mathbf{r}' \right) d\mathbf{r} d\mathbf{r}' \quad (7)$$

where  $\Omega_{\text{cell}}$  describes a cuboid reference cell and  $\mathbf{e}_k$  is a unit vector in direction of the  $k$ th coordinate axis. Here the magnetization is assumed to be constant within each simulation cell and the field generated by each source cell is averaged over each target cell. This results in a sixfold integral for the computation of the discrete demagnetization tensor  $\tilde{\mathbf{N}}_{i-j}$ . An analytical solution of (7) was derived by Newell et al. in [13]. The diagonal element  $N^{1,1}$  computes as

$$N_{i-j}^{1,1} = -\frac{1}{4\pi \Delta r_1 \Delta r_2 \Delta r_3} \sum_{\mathbf{k}, \mathbf{l} \in \{0,1\}} (-1)^{\sum_x k_x + l_x} f[(i_1 - j_1 + k_1 - l_1) \Delta r_1, (i_2 - j_2 + k_2 - l_2) \Delta r_2, (i_3 - j_3 + k_3 - l_3) \Delta r_3] \quad (8)$$

where the auxiliary function  $f$  is defined by

$$\begin{aligned} f(r_1, r_2, r_3) = & \frac{|r_2|}{2} (r_3^2 - r_1^2) \sinh^{-1} \left( \frac{|r_2|}{\sqrt{r_1^2 + r_3^2}} \right) \\ & + \frac{|r_3|}{2} (r_2^2 - r_1^2) \sinh^{-1} \left( \frac{|r_3|}{\sqrt{r_1^2 + r_2^2}} \right) \\ & - |r_1 r_2 r_3| \tan^{-1} \left( \frac{|r_2 r_3|}{r_1 \sqrt{r_1^2 + r_2^2 + r_3^2}} \right) \\ & + \frac{1}{6} (2r_1^2 - r_2^2 - r_3^2) \sqrt{r_1^2 + r_2^2 + r_3^2}. \end{aligned} \quad (9)$$

The elements  $N^{2,2}$  and  $N^{3,3}$  are obtained by circular permutation of the coordinates

$$N_{i-j}^{2,2} = N_{(i_2, i_3, i_1) - (j_2, j_3, j_1)}^{1,1} \quad (10)$$

$$N_{i-j}^{3,3} = N_{(i_3, i_1, i_2) - (j_3, j_1, j_2)}^{1,1} \quad (11)$$

According to Newell the off-diagonal element  $N^{1,2}$  is given by

$$\begin{aligned} N_{i-j}^{1,2} = & -\frac{1}{4\pi \Delta r_1 \Delta r_2 \Delta r_3} \sum_{\mathbf{k}, \mathbf{l} \in \{0,1\}} (-1)^{\sum_x k_x + l_x} \\ & g[(i_1 - j_2 + k_1 - l_1) \Delta r_1, (i_2 - j_2 + k_2 - l_2) \Delta r_2, (i_3 - j_3 + k_3 - l_3) \Delta r_3] \end{aligned} \quad (12)$$

where the function  $g$  is defined by

$$\begin{aligned}
g(r_1, r_2, r_3) = & (r_1 r_2 r_3) \sinh^{-1} \left( \frac{r_3}{\sqrt{r_1^2 + r_2^2}} \right) \\
& + \frac{r_2}{6} (3r_3^2 - r_2^2) \sinh^{-1} \left( \frac{r_1}{\sqrt{r_2^2 + r_3^2}} \right) \\
& + \frac{r_1}{6} (3r_3^2 - r_1^2) \sinh^{-1} \left( \frac{r_2}{\sqrt{r_1^2 + r_3^2}} \right) \\
& - \frac{r_3^3}{6} \tan^{-1} \left( \frac{r_1 r_2}{r_3 \sqrt{r_1^2 + r_2^2 + r_3^2}} \right) - \frac{r_3 r_2^2}{2} \tan^{-1} \left( \frac{r_1 r_3}{r_2 \sqrt{r_1^2 + r_2^2 + r_3^2}} \right) \\
& - \frac{r_3 r_1^2}{2} \tan^{-1} \left( \frac{r_2 r_3}{r_1 \sqrt{r_1^2 + r_2^2 + r_3^2}} \right) - \frac{r_1 r_2 \sqrt{r_1^2 + r_2^2 + r_3^2}}{3}.
\end{aligned} \tag{13}$$

Again other off-diagonal elements are obtained by permutation of coordinates

$$N_{\mathbf{i}-\mathbf{j}}^{1,3} = N_{(i_1, i_3, i_2) - (j_1, j_3, j_2)}^{1,2} \tag{14}$$

$$N_{\mathbf{i}-\mathbf{j}}^{2,3} = N_{(i_2, i_3, i_1) - (j_2, j_3, j_1)}^{1,2}. \tag{15}$$

The remaining components of the tensor are obtained by exploiting the symmetry of  $\tilde{\mathbf{N}}$ , i.e.  $N^{i,j} = N^{j,i}$ .

Listing 1: Definition of auxiliary functions  $f$  and  $g$ . The very small number  $\text{eps}$  is added to denominators in order to avoid division-by-zero errors.

```

eps = 1e-18

def f(p):
    x, y, z = abs(p[0]), abs(p[1]), abs(p[2])
    return + y/2.0*(z**2-x**2)*asinh(y/(sqrt(x**2+z**2)+eps)) \
           + z/2.0*(y**2-x**2)*asinh(z/(sqrt(x**2+y**2)+eps)) \
           - x*y*z*atan(y*z/(x * sqrt(x**2+y**2+z**2)+eps)) \
           + 1.0/6.0*(2*x**2-y**2-z**2)*sqrt(x**2+y**2+z**2)

def g(p):
    x, y, z = p[0], p[1], abs(p[2])
    return + x*y*z*asinh(z/(sqrt(x**2+y**2)+eps)) \
           + y/6.0*(3.0*z**2-y**2)*asinh(x/(sqrt(y**2+z**2)+eps)) \
           + x/6.0*(3.0*z**2-x**2)*asinh(y/(sqrt(x**2+z**2)+eps)) \
           - z**3/6.0 *atan(x*y/(z*sqrt(x**2+y**2+z**2)+eps)) \
           - z*y**2/2.0*atan(x*z/(y*sqrt(x**2+y**2+z**2)+eps)) \
           - z*x**2/2.0*atan(y*z/(x*sqrt(x**2+y**2+z**2)+eps)) \
           - x*y*sqrt(x**2+y**2+z**2)/3.0

```

The calculation of the discrete demagnetization tensor in Python is straightforward. Listing 1 shows the function definitions for the auxiliary functions  $f$  and  $g$ . Note that fractions occurring in  $f$  and  $g$  might feature zero denominators. However, limit considerations show that all fractions tend to zero in this case. In order to avoid division-by-zero errors in the implementation, a very small floating point number  $\text{eps}$  is added to all denominators.

Listing 2: Assembly of the demagnetization tensor  $\tilde{\mathbf{N}}$ . Only the six distinct components of the symmetric tensor are computed.

```

def set_n_demag(c, permute, func):

```

$\tilde{N}_{0,-1}$	$\tilde{N}_{1,-1}$	$\tilde{N}_{2,-1}$	$\tilde{N}_{-2,-1}$	$\tilde{N}_{-1,-1}$	*	0	0	0	0	0
$\tilde{N}_{0,-2}$	$\tilde{N}_{1,-2}$	$\tilde{N}_{2,-2}$	$\tilde{N}_{-2,-2}$	$\tilde{N}_{-1,-2}$		0	0	0	0	0
$\tilde{N}_{0,2}$	$\tilde{N}_{1,2}$	$\tilde{N}_{2,2}$	$\tilde{N}_{-2,2}$	$\tilde{N}_{-1,2}$		$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	0	0
$\tilde{N}_{0,1}$	$\tilde{N}_{1,1}$	$\tilde{N}_{2,1}$	$\tilde{N}_{-2,1}$	$\tilde{N}_{-1,1}$		$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	0	0
$\tilde{N}_{0,0}$	$\tilde{N}_{1,0}$	$\tilde{N}_{2,0}$	$\tilde{N}_{-2,0}$	$\tilde{N}_{-1,0}$		$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	0	0

Figure 1: Discrete convolution of the magnetization  $\mathbf{m}$  with the demagnetization tensor  $\tilde{\mathbf{N}}$ . The numbering for the tensor entries start with zero and are wrapped around for negative distances. The magnetization is zero-padded to account for the cyclic nature of the fast convolution algorithm.

```

it = np.nditer(n_demag[:,:,:c], flags=['multi_index'], op_flags=['writeonly'])
while not it.finished:
    value = 0.0
    for i in np.rollaxis(np.indices((2,)*6), 0, 7).reshape(64, 6):
        idx=map(lambda k: (it.multi_index[k]+n[k]-1)%(2*n[k]-1)-n[k]+1, range(3))
        value+=(-1)**sum(i)*func(map(lambda j: (idx[j]+i[j]-i[j+3])*dx[j], permute))
    it[0]=-value/(4*pi*np.prod(dx))
    it.iternext()

n_demag = np.zeros([2*i-1 for i in n] + [6])
for i, t in enumerate(((f,0,1,2), (g,0,1,2), (g,0,2,1),
                        (f,1,2,0), (g,1,2,0), (f,2,0,1))):
    set_n_demag(i, t[1:], t[0])

```

The implementation of the tensor assembly is shown in listing 2. Basically the outer while loop iterates over the possible distances  $i - j$  while the inner for loop corresponds to the sum in (8) and (12) respectively. Within a regular grid of size  $n_1 \times n_2 \times n_3$ , possible index distances are  $-n_x \leq i_x - j_x \leq n_x$ . Due to its symmetry only six components of the demagnetization tensor are computed. This results in a total storage size of  $2n_1 - 1 \times 2n_2 - 1 \times 2n_3 - 1 \times 6$  for the tensor. Note that the spatial ordering of the tensor array is not done by increasing distance which would require the first element to hold the tensor for the largest negative distance. Instead the numbering starts at zero distance and then cycles in a modulus fashion, see Fig. 1. This numbering is well suited for the application of the fast convolution as will be seen later.

The actual computation of the demagnetization field is performed in Fourier space where the discrete convolution in (6) reduces to a cell-wise multiplication

$$\mathcal{F}(\tilde{\mathbf{N}} * \mathbf{m}) = \mathcal{F}(\tilde{\mathbf{N}})\mathcal{F}(\mathbf{m}). \quad (16)$$

This fast-convolution algorithm reduces the computational complexity of the demagnetization-field computation from  $\mathcal{O}(N)$  for a naive implementation of the convolution to  $\mathcal{O}(N \log N)$  when using the fast Fourier transform algorithm.

Note that the fast convolution algorithm expects the convolution kernel to be of the same size as the function subject to the convolution in order to perform cell-wise multiplication in

Fourier space. The discrete magnetization however is of size  $\prod_i n_i$  whereas the discrete tensor is of size  $\prod_i 2n_i - 1$ . Hence the discrete magnetization is expanded in order to match the size of the kernel, see Fig. 1. Note furthermore that the fast convolution applies the convolution kernel in a cyclic manner

$$(f * g)_i = \sum_{j=0}^{n-1} f_{(i-j+n)\%n} \cdot g_j \quad (17)$$

, see [14]. Thus the only reasonable choice for the expansion of the magnetization is the adding of zero entries, which is referred to as zero-padding. This way, the computation of the field in a certain cell takes into account only contributions from the magnetization at physically possible cell distances.

Listing 3: Implementation of the fast convolution for the computation of the demagnetization field.

```
m_pad = np.zeros([2*i-1 for i in n] + [3])
m_pad[:n[0],:n[1],:n[2],:] = m

f_n_demag = np.fft.fftn(n_demag, axes = filter(lambda i: n[i] > 1, range(3)))
f_m_pad    = np.fft.fftn(m_pad, axes = filter(lambda i: n[i] > 1, range(3)))

f_h_demag_pad = np.zeros(f_m_pad.shape, dtype=f_m_pad.dtype)
f_h_demag_pad[:, :, :, 0] = (f_n_demag[:, :, :, (0, 1, 2)]*f_m_pad).sum(axis = 3)
f_h_demag_pad[:, :, :, 1] = (f_n_demag[:, :, :, (1, 3, 4)]*f_m_pad).sum(axis = 3)
f_h_demag_pad[:, :, :, 2] = (f_n_demag[:, :, :, (2, 4, 5)]*f_m_pad).sum(axis = 3)

h_demag = ms*np.fft.ifftn(f_h_demag_pad,
    axes = filter(lambda i: n[i] > 1, range(3)))[:n[0],:n[1],:n[2],:].real
```

The actual implementation of the fast convolution is shown in listing 3. The `fftn` and `ifftn` routines of NumPy perform an efficient three-dimensional Fourier transform. The tensor-vector multiplication in Fourier space is implemented row-wise considering the symmetry of the tensor.

The presented code is a very basic implementation of a fast convolution algorithm. Many improvements have been proposed in order to speed up this algorithm or to increase its accuracy. A simple way to increase computation speed is the proper choice of the grid size. Depending on the particular implementation, the fast Fourier transform performs better on certain list sizes. In general most FFT implementations perform best on sizes that can be decomposed into small prime factors. By additional zero padding the grid may be extended to a suitable size without changing the result of the computation. A simple extension from  $2n - 1$  to  $2n$  in every spatial dimension often leads to a noticeable speedup.

Another performance gain may be achieved by omitting Fourier transforms of zero entries. These occur due to the zero-padding of the magnetization. Likewise some inverse transforms may be omitted since only parts of the padded result are of physical meaning. This procedure reduces the number of one-dimensional Fourier transforms to 7/12, see [15]. By accounting for symmetries in the demagnetization tensor the algorithm can be further improved, see [11].

Regarding accuracy the presented algorithm for the computation of the demagnetization tensor suffers from numerical problems especially for large distances and stretched simulation cells. These problems are usually overcome by switching between analytical formulas and numerical integration schemes depending on the cell distance, see [16].

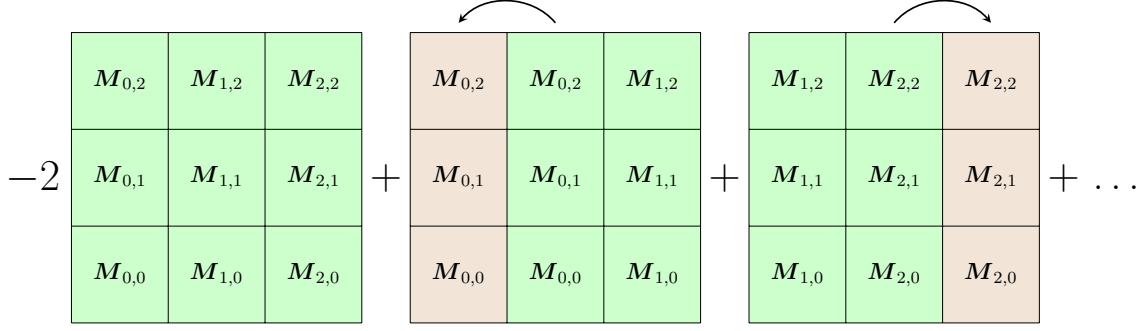


Figure 2: Computation of the Laplacian with finite differences. The boundary values are duplicated in order to account for the zero Neumann boundary conditions.

## 4 Exchange Field

The exchange field models the quantum mechanical exchange interaction. It is derived from a continuous formulation of the Heisenberg Hamiltonian and reads

$$\mathbf{H}_{\text{ex}} = \frac{2A}{\mu_0 M_s} \Delta \mathbf{m}. \quad (18)$$

The second spatial derivative in the exchange field results in an additional boundary condition to the Landau-Lifshitz-Gilbert equation (1) in order to exhibit a unique solution. It was shown in [17] that the Neumann condition

$$\frac{\partial \mathbf{m}}{\partial \mathbf{n}} = 0 \quad (19)$$

has to hold if the boundary of the considered region is an outer boundary of a magnet. The lowest order centered finite-difference approximation of the second derivative reads

$$f''(x) \approx \frac{f(x + \Delta x) - 2f(x) + f(x - \Delta x)}{\Delta x^2}. \quad (20)$$

With this, the three-dimensional Laplacian in (18) can be approximated by

$$\Delta \mathbf{m}(\mathbf{r}) \approx \sum_i \frac{\mathbf{m}(\mathbf{r} + \Delta r_i \mathbf{e}_i) - 2\mathbf{m}(\mathbf{r}) + \mathbf{m}(\mathbf{r} - \Delta r_i \mathbf{e}_i)}{\Delta r_i^2} \quad (21)$$

where  $\Delta r_i$  is naturally chosen to match the cell size. Not every simulation cell has neighboring cells in all six considered directions. In case of a missing neighbor cell the magnetization of this cell is assumed to be the same as that of the center cell. This procedure implicitly accounts for the Neumann boundary condition (19).

Listing 4: Computation of the exchange field. The `repeat` method of NumPy is used to shift the magnetization while duplicating the boundary values.

```
h_ex = -2*m*sum([1/x**2 for x in dx])
for i in range(6):
    h_ex += np.repeat(m, 1 if n[i%3]==1 else \
                      [i/3*2]+[1]*(n[i%3]-2)+[2-i/3*2],axis=i%3)/dx[i%3]**2
h_ex *= 2*A/(mu0*ms)
```

Figure 2 visualizes the implementation of (21). The finite differences for all cells are computed simultaneously by adding shifted versions of the arrays holding the magnetization values. Listing 4 shows the computation of the exchange field.

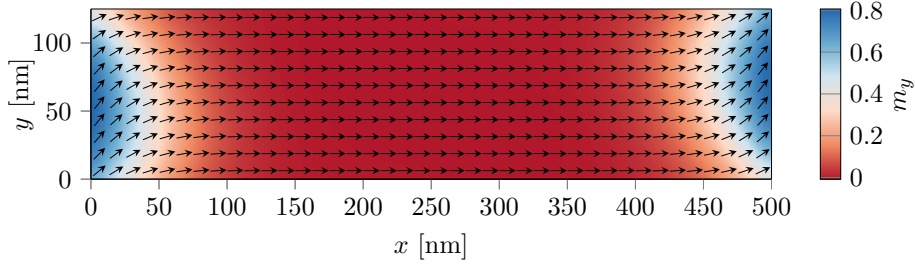


Figure 3: Magnetic s-state in a permalloy thin film as required as start configuration for the  $\mu$ Mag standard problem #4.

Possible improvements of this algorithm include the application of higher order finite-difference schemes. However, the presented scheme turns out to deliver sufficient accuracy for most problems and is implemented by many finite-difference codes.

## 5 Landau-Lifshitz-Gilbert Equation

The numerical integration of the Landau-Lifshitz-Gilbert equation in the context of finite-difference micromagnetics is usually performed with explicit schemes. Implicit schemes outperform explicit schemes in terms of stability which is especially useful in the case of stiff problems. Due to the regular and well shaped grid, the stiffness of finite-difference problems is in general lower than that of finite-element methods on possibly ill shaped irregular meshes [7]. Also the stability of the implicit methods comes at the price of nonlinear problems due to the nonlinearity of the Landau-Lifshitz-Gilbert equation.

Listing 5: Projected explicit Euler scheme for the integration of the Landau-Lifshitz-Gilbert equation.

```
dmdt = - gamma/(1+alpha**2) * np.cross(m, h) \
        - alpha*gamma/(1+alpha**2) * np.cross(m, np.cross(m, h))
m     += dt * dmdt
m     /= np.repeat(np.sqrt((m*m).sum(axis=3)), 3).reshape(m.shape)
```

The Landau-Lifshitz-Gilbert equation preserves the modulus of the magnetization at any point in space, i.e.  $\partial|\mathbf{m}|/\partial t = 0$ . A simple integration scheme that accounts for this feature is a projected explicit Euler method

$$\mathbf{m}(t+h) = \frac{\mathbf{m}(t) + h\partial_t\mathbf{m}(t)}{|\mathbf{m}(t) + h\partial_t\mathbf{m}(t)|} \quad (22)$$

that is applied cellwise. The implementation of this method is shown in listing 5.

The performance of the time integration can be significantly improved by application of higher-order integration schemes. A common choice in finite-difference micromagnetics is a Runge-Kutta scheme of 4th order with an adaptive step size based on a 5th order error estimation.

## 6 Numerical Experiments

In order to validate the implementation, the standard problem #4 proposed by the  $\mu$ Mag group is computed. A magnetic cuboid with size  $500 \times 125 \times 3$  nm and the material parameters of permalloy ( $A = 1.3 \times 10^{-11}$  J/m,  $M_s = 8.0 \times 10^5$  A/m,  $\alpha = 0.02$ ) is considered. The system



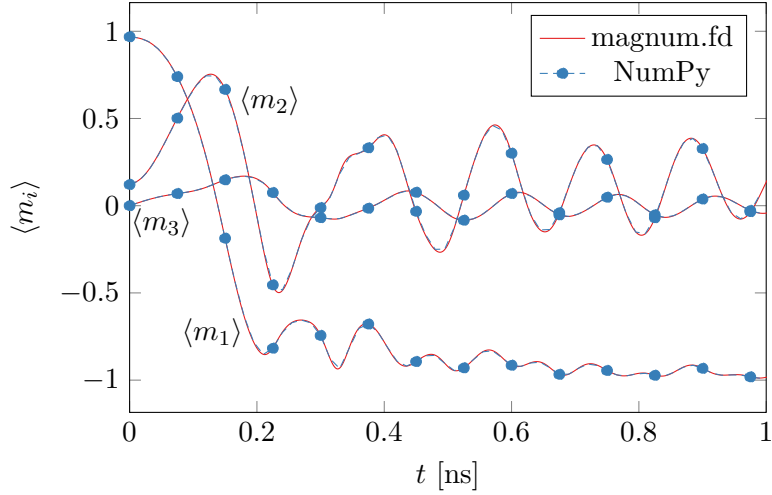


Figure 4: The time evolution of the averaged magnetization components  $\langle m_i \rangle$  for the standard problem #4.

is relaxed into a so-called s-state, see Fig. 3. In a second step an external Zeeman field with a magnitude of 25 mT is applied with an angle of  $170^\circ$  to the sample. The external field is directed opposite to the predominant magnetization direction and thus a switching process is initiated. The simulation-cell size for this problem is chosen as  $5 \times 5 \times 3$  nm and a time step of  $h = 5 \times 10^{-15}$  s is used. Figure 4 shows the time evolution of the averaged magnetization components during switching. The results are compared to a solution computed with the finite-difference code magnum.fd [18] and show a good agreement.

## 7 Conclusion

We present a simple but complete micromagnetic finite-difference code written in less than 70 lines of Python. While not competitive to mature micromagnetic simulation programs, the presented code provides all essential building blocks for complex micromagnetic computations. Due to the powerful and easy-to-use NumPy library it may serve as a prototyping environment for novel algorithms. Furthermore the presented work is meant to give a brief introduction to finite-difference micromagnetics and provides references for detailed information on the topic. A complete listing for the solution of the  $\mu$ Mag standard problem #4 is shown in the appendix and can be downloaded from [19].

## Acknowledgements

The financial support by the Austrian Federal Ministry of Science, Research and Economy and the National Foundation for Research, Technology and Development as well as the FWF project SFB-ViCoM, F4112-N13 is gratefully acknowledged.

## A Code for Standard Problem #4

```
import numpy as np
from math import asinh, atan, sqrt, pi
```

```

# setup mesh and material constants
n      = (100, 25, 1)
dx      = (5e-9, 5e-9, 3e-9)
mu0     = 4e-7 * pi
gamma   = 2.211e5
ms      = 8e5
A       = 1.3e-11
alpha   = 0.02

# a very small number
eps = 1e-18

# newell f
def f(p):
    x, y, z = abs(p[0]), abs(p[1]), abs(p[2])
    return + y/2.0*(z**2-x**2)*asinh(y/(sqrt(x**2+z**2)+eps)) \
           + z/2.0*(y**2-x**2)*asinh(z/(sqrt(x**2+y**2)+eps)) \
           - x*y*z*atan(y*z/(x * sqrt(x**2+y**2+z**2)+eps)) \
           + 1.0/6.0*(2*x**2-y**2-z**2)*sqrt(x**2+y**2+z**2)

# newell g
def g(p):
    x, y, z = p[0], p[1], abs(p[2])
    return + x*y*z*asinh(z/(sqrt(x**2+y**2)+eps)) \
           + y/6.0*(3.0*z**2-y**2)*asinh(x/(sqrt(y**2+z**2)+eps)) \
           + x/6.0*(3.0*z**2-x**2)*asinh(y/(sqrt(x**2+z**2)+eps)) \
           - z**3/6.0 *atan(x*y/(z*sqrt(x**2+y**2+z**2)+eps)) \
           - z*y**2/2.0*atan(x*z/(y*sqrt(x**2+y**2+z**2)+eps)) \
           - z*x**2/2.0*atan(y*z/(x*sqrt(x**2+y**2+z**2)+eps)) \
           - x*y*sqrt(x**2+y**2+z**2)/3.0

# demag tensor setup
def set_n_demag(c, permute, func):
    it = np.nditer(n_demag[:,:,:c], flags=['multi_index'], op_flags=['writeonly'])
    while not it.finished:
        value = 0.0
        for i in np.rollaxis(np.indices((2,)*6), 0, 7).reshape(64, 6):
            idx=map(lambda k: (it.multi_index[k]+n[k]-1)%(2*n[k]-1)-n[k]+1, range(3))
            value+=(-1)**sum(i)*func(map(lambda j: (idx[j]+i[j]-i[j+3])*dx[j], permute))
            it[0]=-value/(4*pi*np.prod(dx))
        it.iternext()

# compute effective field (demag + exchange)
def h_eff(m):
    # demag field
    m_pad[:n[0],:n[1],:n[2],:] = m
    f_m_pad = np.fft.fftn(m_pad, axes = filter(lambda i: n[i] > 1, range(3)))
    f_h_demag_pad = np.zeros(f_m_pad.shape, dtype=f_m_pad.dtype)
    f_h_demag_pad[:,:,:0] = (f_n_demag[:,:,:,(0, 1, 2)]*f_m_pad).sum(axis = 3)
    f_h_demag_pad[:,:,:1] = (f_n_demag[:,:,:,(1, 3, 4)]*f_m_pad).sum(axis = 3)
    f_h_demag_pad[:,:,:2] = (f_n_demag[:,:,:,(2, 4, 5)]*f_m_pad).sum(axis = 3)

    h_demag = np.fft.ifftn(f_h_demag_pad,
        axes = filter(lambda i: n[i] > 1, range(3))[:n[0],:n[1],:n[2],:].real

    # exchange field
    h_ex = - 2 * m * sum([1/x**2 for x in dx])
    for i in range(6):
        h_ex += np.repeat(m, 1 if n[i%3] == 1 else \
            [i/3*2]+[1]*(n[i%3]-2)+[2-i/3*2], axis=i%3)/dx[i%3]**2

```

```

    return ms*h_demag + 2*A/(mu0*ms)*h_ex

# compute llg step with optional zeeman field
def llg(m, dt, h_zeeman = 0.0):
    h = h_eff(m) + h_zeeman
    dmdt = - gamma/(1+alpha**2) * np.cross(m, h) \
            - alpha*gamma/(1+alpha**2) * np.cross(m, np.cross(m, h))
    m += dt * dmdt
    return m/np.repeat(np.sqrt((m*m).sum(axis=3)), 3).reshape(m.shape)

# setup demag tensor
n_demag = np.zeros([2*i-1 for i in n] + [6])
for i, t in enumerate(((f,0,1,2), (g,0,1,2), (g,0,2,1),
                        (f,1,2,0), (g,1,2,0), (f,2,0,1))):
    set_n_demag(i, t[1:], t[0])

m_pad = np.zeros([2*i-1 for i in n] + [3])
f_n_demag = np.fft.fftn(n_demag, axes = filter(lambda i: n[i] > 1, range(3)))

# initialize magnetization that relaxes into s-state
m = np.zeros(n + (3,))
m[1:-1, :, :, 0] = 1.0
m[(-1,0), :, :, 1] = 1.0

# relax
alpha = 0.50
for i in range(10000): llg(m, 5e-14)

# switch
alpha = 0.02
dt = 5e-15
h_zeeman = np.tile([-24.6e-3/mu0, +4.3e-3/mu0, 0.0], np.prod(n)).reshape(m.shape)

with open('sp4.dat', 'w') as file:
    for i in range(int(1e-9/dt)):
        file.write("%f_%f_%f_%f\n" % \
                    ((i*1e9*dt,) + tuple(map(lambda i: np.mean(m[:, :, :, i]), range(3)))))
        llg(m, dt, h_zeeman)

```

## References

- [1] M. J. Donahue and D. G. Porter, *OOMMF User's guide*. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, 1999.
- [2] C. Abert, G. Selke, B. Kruger, and A. Drews, "A fast finite-difference method for micromagnetics using the magnetic scalar potential," *Magnetics, IEEE Transactions on*, vol. 48, no. 3, pp. 1105–1109, 2012.
- [3] A. Vansteenkiste, J. Leliaert, M. Dvornik, M. Helsen, F. Garcia-Sanchez, and B. Van Waeyenberge, "The design and verification of mumax3," *AIP Advances*, vol. 4, no. 10, p. 107133, 2014.
- [4] 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*, vol. 345, pp. 29–35, 2013.

- [5] 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*, vol. 43, no. 6, pp. 2896–2898, 2007.
- [6] 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*, vol. 28, no. 2, pp. 366–383, 2003.
- [7] D. Suess, V. Tsiantos, T. Schrefl, J. Fidler, W. Scholz, H. Forster, R. Dittrich, and J. Miles, “Time resolved micromagnetics using a preconditioned time integration method,” *Journal of Magnetism and Magnetic Materials*, vol. 248, no. 2, pp. 298–311, 2002.
- [8] A. Kakay, E. Westphal, and R. Hertel, “Speedup of fem micromagnetic simulations with graphical processing units,” *Magnetics, IEEE Transactions on*, vol. 46, no. 6, pp. 2303–2306, 2010.
- [9] R. Chang, S. Li, M. Lubarda, B. Livshitz, and V. Lomakin, “Fastmag: Fast micromagnetic simulator for complex magnetic structures,” *Journal of Applied Physics*, vol. 109, no. 7, p. 07D358, 2011.
- [10] T. Schrefl, G. Hrkac, S. Bance, D. Suess, O. Ertl, and J. Fidler, “Numerical methods in micromagnetics (finite element method),” *Handbook of magnetism and advanced magnetic materials*, 2007.
- [11] J. E. Miltat and M. J. Donahue, “Numerical micromagnetics: Finite difference methods,” *Handbook of magnetism and advanced magnetic materials*, 2007.
- [12] S. Van Der Walt, S. C. Colbert, and G. Varoquaux, “The numpy array: a structure for efficient numerical computation,” *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 2011.
- [13] A. J. Newell, W. Williams, and D. J. Dunlop, “A generalization of the demagnetizing tensor for nonuniform magnetization,” *Journal of Geophysical Research: Solid Earth (1978–2012)*, vol. 98, no. B6, pp. 9551–9555, 1993.
- [14] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.
- [15] Y. Kanai, K. Koyama, M. Ueki, T. Tsukamoto, K. Yoshida, S. J. Greaves, and H. Muraoka, “Micromagnetic analysis of shielded write heads using symmetric multiprocessing systems,” *Magnetics, IEEE Transactions on*, vol. 46, no. 8, pp. 3337–3340, 2010.
- [16] M. J. Donahue, “Accurate computation of the demagnetization tensor.” [http://math.nist.gov/~MDonahue/talks/hmm2007-MBO-03-accurate\\_demag.pdf](http://math.nist.gov/~MDonahue/talks/hmm2007-MBO-03-accurate_demag.pdf).
- [17] G. Rado and J. Weertman, “Spin-wave resonance in a ferromagnetic metal,” *Journal of Physics and Chemistry of Solids*, vol. 11, no. 3, pp. 315–333, 1959.
- [18] “magnum.fd.” <http://micromagnetics.org/magnum.fd>.
- [19] “70 lines of NumPy.” <https://github.com/micromagnetics/70LinesOfNumpy>.