On the automatic translation of classical field theory to computer simulations

Thomas Fischbacher
Hans Fangohr
School of Engineering Sciences, Southampton

Abstract We show that a broad class of physical systems based on classical field theory allow automated translation to parallelized numerical simulation code. Furthermore, we study the utility of this abstract approach for real world problems by employing a prototype compiler.

1 Introduction

XXX TODO: briefly mention the role of ocaml and python and C in the beginning - maybe using a diagram such as the one on the nmag poster - as we will keep on referring back to this!
When it comes to the simulation of physical systems governed by classical field theory, there are two broad categories. On the one hand, we have systems based on a well-established and rigidly defined physical model, such as computational fluid dynamics. For these, the most important qualities of simulation code are efficiency and scalability.

On the other hand, there are systems whose behaviour is determined by a number of often very different physical effects. Furthermore, we may have competing models for some of those effects which give effective descriptions in different physical situations. So, depending on the particular system under investigation, modeling choices can have a great influence on both the selection and form of the fields as well as the governing equations to be included in the simulation. We will call these systems 'flexibility-demanding'.

It must be emphasized that simulations of the physics of nanoscale systems frequently are flexibility-demanding. The underlying deeper reason for this is that we can understand field theory based models of mesoscopic systems not only as the result of some averaging process over atomic scales. Rather, we can also approach these systems from the other side, macroscopic instead of atomic length scales, and regard them as the result of including the leading correction terms that are bound to become irrelevant as we gradually go to ever larger length scales. This renormalization group inspired perspective on these models makes their broad scope plausible.

In this work, we will restrict ourselves to a broad generalization of reaction-diffusion type systems in arbitrary spatial dimension where differential operators involve at most two spatial derivatives and equations of motion are of elliptic type.

XXX mention that hand-crafting simulation code is tedious work. In particular, writing code to compute the Jacobian is error-prone!

1.1 Motivation (and example)

The original incentive for this work came from the field of computational micromagnetism. The physics of magnetic systems is extremely rich as magnetism both is governed and also interacts with a broad range of very different physical effects. This, of course, precisely is the reason for the considerable technical importance of magnetism, in particular in the field of data storage and processing. As the prototype compiler described in section XXXref was written with micromagnetism as a first application in mind (in form of the nmag XXXref library), one should expect that most physical systems which roughly follow the abstract structure of the basic micromagnetic model should be open to this approach.

In order to provide a first and already highly nontrivial example of a model to which these automatic translation techniques can be applied, we therefore briefly describe the structure of micromagnetic field theory.

In this model, the fundamental degrees of freedom are provided by a vector field $\vec{M}(\vec{x})$ describing local magnetization. Physically, the magnitude of $\vec{M}$ is determined by local material properties, so there actually are only two dynamical degrees of freedom at every point.[1]

The dynamics of the field $\vec{M}(\vec{x})$ is governed by the so-called Landau Lifshitz and Gilbert equation which describes the reaction of magnetisation to a effective magnetic field strength to which multiple individual effects contribute. This equation is local and non-linear, but polynomial in the contributing fields. Physically, it models magnetic precession perpendicular to the effective magnetic field strength as well as dissipative damping. It is of the form

$$d/dt\, \vec{M}(\vec{x}) = c_1 \vec{M}(\vec{x}) \times \vec{H}_{\text{eff}}(\vec{x}) + c_2 \vec{M}(\vec{x}) \times \left(\vec{M}(\vec{x}) \times \vec{H}_{\text{eff}}(\vec{x}) \right), \qquad (1)$$

or, in more general and more systematic tensor-index notation (where we suppress the spatial dependency on $\vec{x}$)

$$d/dt\, M_i = c_1 \epsilon_{ijk} M_j H_{\text{eff}\,k} + c_2 \epsilon_{ijk} M_j \epsilon_{kmn} M_m H_{\text{eff}\,n}. \qquad (2)$$

As tensor-index notation is more systematic and more powerful, in particular when fundamental degrees of freedom are higher-rank tensor fields, `nsim` employs these conventions.

The effective magnetic field strength $\vec{H}_{\text{eff}}$ is the sum of individual contributions from a number of different effects. These include an external field $\vec{H}_{\text{ext}}$ applied by the experimentor and the magnetic field produced by other regions of the sample, which is of the structure

$$H_{\text{demag}\,j}(\vec{x}) = \int d^3y\, G_{jk}(\vec{x}, \vec{y})\, M_k(\vec{y}) \qquad (3)$$

where the kernel $G_{jk}(\vec{x}, \vec{y})$ gives the magnetic field at position $\vec{x}$ produced by a localized Dirac dipole at $\vec{y}$.

[1] We may just as well describe this as a $SO(3)/SO(2)$ coset model.

In addition to these truly magnetic fields, there are other contributions which effectively behave in the same way. The most prominent among these phenomenologically models the effect of quantum-mechanical exchange, which leads to a desire of magnetisation to evolve towards a state of homogeneous alignment and is given through application of a differential operator (the vector Laplacian) to the magnetization field:

$$H_{\text{exch}\,j}(\vec{x}) = cd/dx_k\, d/dx_k\, M_j(\vec{x}). \tag{4}$$

Another effective contribution is given by crystal properties of the material which may energetically favour some magnetization directions over others. This *anisotropy* term usually is described through a truncated polynomial expansion, which is a local function of the magnetization $\vec{M}(\vec{x})$, or, in tensor-index notation:

$$H_{\text{anis}\,j}(\vec{x}) = \delta E[\vec{M}(\vec{x})]/\delta M_j(\vec{x}) \tag{5}$$

with the energy functional

$$E[\vec{M}] = E^{(0)} + E_j^{(1)} M_j + E_j^{(2)} k M_j M_k + E_k^{(3)} M_k + \ldots \tag{6}$$

(Physically speaking, $E^{(0)}$ is irrelevant and $E_j^{(1)}$ vanishes for symmetry reasons but would behave like an extra contribution to the externally applied field. Nevertheless, this is the most general form.)

In addition to these contributions (which all are of computationally very different type) to H_{eff},

$$\vec{H}_{\text{eff}}(\vec{x}) = \vec{H}_{\text{ext}}(\vec{x}) + \vec{H}_{\text{demag}}(\vec{x}) + \vec{H}_{\text{exch}}(\vec{x}) + \vec{H}_{\text{anis}}(\vec{x}), \tag{7}$$

various extensions may include other contributions as well, but in practically any case, these do not involve any new features beyond local polynomials in other fields, spatial differential operators of order at most two, and integrals over propagators from long-range interactions.

Hence, by implementing both support for these key techniques in conjunction with convenient notational conventions, a simulation framework can gain considerable expressive power and broad applicability.

2 Formal specifications of physical systems

2.1 The language

XXX Where to explain the subfield concept?

Fields with tensor indices, Equations of motion / local equations, operator specification, linalg scripts, EBNF Grammars in appendix

2.2 Field dependencies

automatic derivation - ‘inference engine’

2.3 Non-magnetic examples

Turing: Reaction-Diffusion models Ginzburg-Landau theory (+ note: coupling GL + LLG!)

2.4 Physics beyond this language

XXX Can do practically everything in field theory

Coupling field theory to non-field-theory (-¿ e.g. geant, field-particle interactions);

Limitation of the number of derivatives usually not a problem - for physical reasons.

3 The `nsim` compiler prototype

In thr following general discussion of the major components of the `nsim` prototype compiler, we will frequently refer back to the `nmag` library whenever we need an example to explain some particular technique.

3.1 Bookkeeping over field degrees of freedom

A central data structure in the `nsim` compiler is the *field layout description*, called (for historical reasons and due to `nsim`'s FEM background) '*mesh with elements (mwe)*'. The role of this field layout description is to provide information on how to physically interpret the data contained in a numerical vector that describes the state of a discretised field.

In the current implementation of `nsim`, the `mwe` data structure is constructed from a geometrical (and topological) specification of the mesh and additional information on the association of (abstractly described) elements to mesh cells as a function of the cell's associated region number. It would be rather straightforward to extend `nsim` in such a way that it also can handle other types of field discretisations that are based on more specialized meshes (such as regular grids), but this has not been implemented so far.

At present, the meshes are taken to be composed of simplicial cells of arbitrary(!) dimension (i.e. tetrahedra in three dimensions, pentatopes in four). An abstract element, which is associated to the cells of the mesh by region, provides information about the degrees of freedom belonging to this cell (and usually shared with neighbouring cells). An element may introduce multiple physical degrees of freedom, and even multiple *types* of degrees of freedom at the same time (e.g. three components of the electrical field strength, three components of the magnetic field strength, and a mass and charge density to describe the state of a plasma.) The actual location of a degree of freedom may be given as a weighted average of the simplex vertex coordinates (for higher order elements), and the corresponding shape functions are given as polynomials in barycentric coordinates.

The individual physical degrees of freedom are taken to be real-valued only. While it occasionally would be useful to support complex-valued quantities as well, this runs into problems with `nsim`, as the underlying sparse matrix linear algebra library (PETSc) cannot handle both real and complex linear algebra within the same program. This, however, is a purely technological rather than a conceptual limitation. While a generalized multiphysics framework certainly has to support scalar and vector degrees of freedom, it should also provide higher order tensors of arbitrary rank to model quantities such as stress, or flow of angular momentum (as e.g. associated to spin-polarized currents). Furthermore, fields may have internal degrees of freedom, i.e. some of the indices of a multi-indexed quantity may not correspond to spatial directions. While this situation is very common in non-abelian gauge theories (such as quantum chromodynamics where QCD 'photons' come in eight different varieties unfortunately carrying the highly misleading name 'colors'), there are few (if any) well known examples in condensed matter physics. This generalization nevertheless is so straightforward that there would be no point in not supporting it as well, however.

The field layout data structure has the capability to provide information about the type and indices of a degree of freedom as well as its physical location in space (both in real coordinates as well as in the form of a weighted average of the positions of mesh vertices), its association to mesh cells and the corresponding shape functions, the total volume associated to the 'tent' spawned by the shape functions, other degrees of freedom residing at the same site and similar bookkeeping information.

The terminology used in the `nsim` compiler is to call a numerical data vector with an associated field layout a *field*. This may contain more than one kind of (tensorial) degree of freedom, e.g. electrical and magnetic field strengths. These individual kinds are called *subfields*, and a *subfield component* is a particular subfield plus a specification of all tensor indices.

The most important operations that use field layout information are:

- Extracting a particular subfield from the numerical data describing a multi-subfield field. This situation can arise when part of the full configuration vector of all degrees of freedom (which e.g. is handled by the time integrator) has to be split and passed on to individually developed parts of the simulation that deal with specific physical sectors (such as e.g. heat diffusion).

- Re-Uniting subfields into a unified field vector (the opposite operation of the previous one).

- Extracting some degrees of freedom of a given subfield into a shorter vector containing only a selection of sites (e.g. some particular surface)

- Inserting data from a restricted vector into a longer vector (the opposite operation of the previous one).

- Setting up bookkeeping information for calling machine code which was generated to handle non-linear local 'reaction' equations later on. (I.e. assembling tables of pointers into multiple fields, one for every site, so that compiled code can access these tables to find and identify the numerical values of individual degrees of freedom living at the given site.)
- Probing the value of a field at an arbitrary position, using the analytic shape functions and interpolation.

Rather frequently, one wants to use field vectors which use different names for individual subfields, but nevertheless are structurally compatible (i.e. component-wise addition of data vectors is a sensible operation). The most evident example of such a situation is the handling of a 'reaction equation' of the type

$$M(j) = M(j) + dt * dM_dt(j)$$

Usability experiments have shown that the most appropriate approach to handle this situation is to implement *aliasing*: we provide a function that is given a field layout as well as subfield name substitutions and produces another 'sibling field layout' which is compatible with the original one (in the above sense) but uses a different set of names for subfields. This can be implemented very efficiently if care is taken that the degree of freedom (dof) data structure does not contain a reference to the subfield name but we instead use a separate function which maps a dof plus a field layout data structure to a subfield name. Then, virtually all of the entries of the field layout data structure (with the exception of the vector containing the abstract elements associated to mesh cells) can be shared between the original and the aliased field layout, making aliasing very cheap in terms of memory requirements. Furthermore, there are some slightly expensive entries in the field layout data structure which are computed on demand when first needed, such as information about all neighbouring sites that carry a given subfield. (This is useful e.g. in micromagnetism to find out whether the magnetisation vector changes so much between adjacent sites that simulation results cannot be trusted.) These can be shared in such a way (by introducing an extra indirection level) that computing this information for one field layout from a set related through aliasing will also make this information available for all others at the same time.

The present `nmag` prototype discerns between primary and sibling field layouts and does not allow secondary aliasing of aliased fields in order to limit complexity. This restriction could actually be removed.

XXX The following paragraph is not at all clear:

An important further issue with respect to fields is that the finite element discretisation of a differential operator, which is obtained by expressing fields as a linear combination of basis functions and letting the operator act on these basis functions, will always map a numerical vector of basis function coefficients to a numerical vector of inner products of the resulting field with the same set of basis functions (the inner product of two functions being given as the integral of their product over space) rather than coefficients of a resulting field. So, conceptually,

the result of the application of an operator matrix to a vector of field coefficients is a vector in the corresponding dual space of linear forms on discretised fields. These are at present referred to as 'cofields' in the `nsim` core, and the OCaml type system is used to enforce the distinction between fields and cofields. There are situations where cofields rather than fields are useful in themselves, e.g. when determining total outflow of a vector field across a boundary (the cofields already providing the volume integral). While there are operation sequences e.g. in micromagnetism which lead from a field to a cofield and then back to a field via solving a matrix equation (such as the discretised Laplace equation), it is also possible to end up with a cofield where a field is required (and vice versa). While the mesh discretisation of the identity operator provides a natural(XXX?) mapping from discretised fields to cofields, trying to invert this in order to map cofields to fields may not be the most appropriate strategy in some situations. One problem is that the inverse mapping would correspond to a nonlocal – hence effectively nonsparse – linear mapping. In particular, the cofield discretisation corresponding to a nondiscretised delta distribution field located at a mesh vertex would be localized, but produce a discretised field when using this inverse mapping which would be non-localized and exhibit ripples that decay the further one moves away from the original source. So, in many situations, it is more appropriate to employ a different mapping from cofields to fields which just divides the entries of the discretised cofields by the spatial integral over the basis function associated to the corresponding degree of freedom (the 'box method' XXX ref - and also read this paper!). During the development of the `nsim` prototype, the strong type-system enforced distinction between field and cofield data vectors gradually turned out to be more a hindrance than an advantage, suggesting that soft flagging of numerical data as representing a field or a cofield may be more appropriate.

XXX re-write above paragraph!

3.2 Primary and Dependent Fields; the inference mechanism

In a typical dynamical physical model, the fields entering the defining equations can be classified as either containing proper 'primary' degrees of freedom which (at least in principle) can be manipulated by the experimentor, or being auxiliary quantities which are computed from the dynamical degrees of freedom via application (or inverse application, such as in solving a Laplace equation) of differential operators or nonlinear local (i.e. not involving spatial derivatives) relations. More complicated auxiliary quantities that involve both differential and non-linear operations usually can be regarded as being defined in terms of more fundamental auxiliary quantities which only involve one of these two types of operations. In effect, this means that there is a dependency tree for auxiliary fields which is rooted in the primary fields. For instance, in the micromagnetic model, the time derivative of the magnetisation dM/dt is a field that depends on the primary magnetisation field as well as the auxiliary effective total magnetic field strength H_{total} which in turn depends on the (user-definable

hence primary) externally applied field strength H_{ext} as well as other auxiliary fields that in turn depend on the primary fields (here, magnetisation) in a non-linear local way (such as H_{anis}), or in a way that involves spatial derivatives but no long-range interaction (such as H_{exch}) or in a complicated way involving long-range interactions and a number of rather complicated intermediate computational steps.

XXX Maybe graphically present the full dependency tree for micromagnetism here XXX.

Auxiliary fields have to be updated in two different situations: either as part of the effort to compute the new velocity vector $\dot{y}(t)$ from the configuration vector $y(t)$ when performing time integration (or – for these purposes – equivalently, energy minimisation), or to reestablish consistency whenever the user changed the physical configuration and reads out some auxiliary quantity from the system. In the latter case, one evidently would want the system to be able to automatically infer which auxiliary fields still are up to date and which have to be recomputed (doing so if necessary) in such a way that multiple subsequent readouts of different auxiliary quantities do not induce any unnecessary computations. This basically is the same kind of control logic that also underlies the 'make' program which deals with dependencies of source and derived files belonging to a software project.

While `nsim` provides such an inference engine which will keep track of auxiliary fields that have become invalid, the present prototype requires this dependency tree to be explicitly specified by the user. As this information can be derived from the user-provided definitions of the auxiliary quantities in a fully automatized way, it would be both feasible and highly desirable to extend the compiler in such a way that explicit dependency specifications no longer are necessary (but still possible to override the default behaviour in unusual situations!)

One should furthermore note that, due to `nsim`'s capability to run a given simulation script both in single-process as well as parallel mode, an extra level of buffering is required that provides direct access to field vectors on the master node which may be distributed over the cluster. Keeping these buffers up to date also is handled by the inference engine.

3.3 Differential Operator Specification

As explained, discretised differential operator matrices map numerical vectors of field coefficients to numerical vectors containing cofield coefficients. So, given field layouts for the left hand side as well as the right hand side, we can 'compile' a purely symbolic specification how derivatives act on subfield components to a sparsely occupied (i.e. having zeroes for combinations of degrees of freedom belonging to sites not belonging to a common mesh cell) numerical matrix. The formalism used in the `nsim` prototype is strongly inspired by quantum mechanical bra-ket notation. If we consider a field ϕ to be represented as a linear combination of the basis functions $|u_i\rangle$ with coefficients c_i, i.e. $\phi = c_i|u_i\rangle$, then a linear operator M acting as $\psi = M\phi$ can be represented in matrix form

by inserting an identity $\mathbb{1} = \langle u_j | g^{(-1)}_{jk} | u_k \rangle$ blah XXX repair description!

So, we represent the most elementary linear operators like, for example, $\vec{E} = -\nabla\phi$, which can be re-written in more systematic tensor component notation as $E_j = -d/dx_j\phi$ in the form of the string:

```
- <E(0) || d/dx0 phi> - <E(1) || d/dx1 phi> - <E(2) || d/dx2 phi>
```

Here, it is helpful to introduce support for a summation convention. (The index range actually has to be specified explicitly):

```
- <E(j) || d/dxj phi>; j:3
```

XXX How to deal with second derivatives; partial integration, LHS derivatives!

XXX compiler internally uses a two-stage process employing 'thunks'; also referred to later!

Unfortunately, this notation, beautifully simple as it is, is not yet general enough to deal with a number of situations that frequently occur. In particular, a number of situations may arise that must also be supported by the grammar of formal operator specifications.

3.3.1 Surface Derivatives

We may want to take surface derivatives instead of volume derivatives, i.e. compute the magnitude of the jump of a degree of freedom at a surface across which it ceases to exist. (XXX "D/Dxj" notation)

3.3.2 Region and boundary restrictions

We may want to restrict some contributions to a differential operator to degrees of freedom associated with some particular mesh regions, or region boundaries. (XXX [vol=] and [boundary=] notation)

3.3.3 Matrix shortening

When employing region-based restrictions, we may want to construct a "shortened" version of the matrix which maps (potentially) shortened data vectors to (potentially) shortened data vectors. (XXX (L——R)= Notation)

3.3.4 Residual gauge fixing

We may want to specify that the final matrix, which would be singular, is to be modified in such a way that it still encodes the same physical content but becomes regular. In particular, when using a discretised Laplace operator to solve a potential equation, we will have to address the 'gauge symmetry' $\Delta\psi = \Delta(\psi+c)$ which reduces the rank of the matrix by one for every component of connection in the system. (The sparse linear algebra library may provide

specialized functions to deal with this problem, but it is useful not to depend on that feature.) (XXX gauge_fix= notation)

XXX explain how this is a residual of the original gauge symmetry. Also explain the component of connection issue.

3.3.5 Dirichlet Boundary Conditions

When working with Dirichlet Boundary Conditions, we may want to exclude certain rows and columns from the matrix but introduce ones on the diagonal in order to maintain regularity. (XXX LEFT=RIGHT notation.)

3.3.6 Periodic Boundary Conditions

XXX periodicity of DOFs, modification of the compiler – details on PBC and time integration discussed elsewhere!

3.3.7 'Middle Fields' XXX better name needed

XXX conductivity example

3.4 Differential Operator Compilation

Once the textual (string) representation of a differential operator has been parsed into internal form, a discretised approximation of this operator in the form of a numerical sparse matrix can be constructed by providing information on the layout of the left hand side and right hand side field. This mapping should be abstract enough to be agnostic of the particular underlying implementation of linear algebra. In particular, it can be highly useful to have the option to internally employ different representations of the numerical data. Evidently, the most important representation will be cluster-distributed sparse numerical matrices. In addition to this, it may be useful in some cases to also employ some dictionary representation of the nonzero entries of an operator to help with the construction of the semi-symbolic Jacobian, cf. section XXX.

Experiments have shown that a very useful design approach is to provide a function that maps a parsed operator plus a left and right field layout to a function which takes as an argument an incrementor function and then calls this incrementor function with the appropriate arguments (row, column, contribution) whenever an entry is to be added to the sparse operator. This mixed functional-imperative style of design may seem unusual at first and may require some getting used to, but is both rather common and often very powerful in imperative functional languages such as Lisp or OCaml. Here, its advantages are three-fold:

- Complete independence of the particular implementation of sparse linear algebra that is employed: the design is modular enough to make switching over to a different linear algebra library very easy.

- This approach allows straightforward integration of most of the other matrix manipulations required to deal with periodicity, shortened vectors, gauge fixing, etc. For example, when filling a matrix that operates on fields containing 'mirage' (XXX was mirage explained earlier?) degrees of freedom due to periodic boundary conditions, some of the matrix rows must be identical to one another and carry contributions from the multiple occurrences of the same degree of freedom on the mesh. This is easily achieved by wrapping the incrementor function to obtain a modified incrementor which uses the same calling conventions but performs multiple matrix increment operations when it deals with entries corresponding to degrees of freedom that have periodicity-related copies.

- Based on such a functional approach, it is comparatively simple to split operator compilation into an analysis and an execution phase. The purpose of the analysis phase is to scan the field layouts and produce a set of functions, one for each relevant combination of left hand side and right hand side element type. When these functions are created, all the integrals of the analytically given shape functions are worked out in symbolic form. The subsequent execution phase walks through all the simplices of the mesh and applies the appropriate function that was created in the analysis phase on the simplex, using geometry information to compute the corresponding contribution to the matrix. By having to do all the costly symbolic integration only once, a considerable speedup can be achieved.

It should be pointed out that there also is a standard object oriented design approach towards this sort of problem, which would consist of XXX. Effortwise, this could be regarded as an unnecessarily heavyweight design that can be traced back to the lack of proper closures in conventional object-oriented thinking and hence the need to emulate their capabilities in a more clumsy way wherever needed.

Structurally, the function implementing this 'differential operator vivification' (i.e. turning an operator into a 'living' function which, when called on a left and right field layout and given an incrmentor, will populate a matrix accordingly) which is provided by the `ddiffop_vivified` function belongs to the most complex core functions in the `nsim` prototype.

XXX support for middle field (rudimentary).

3.5 Solving operator equations

3.6 Nonsparse linear operators

While every differential operator will produce a sparse matrix when discretised in the straightforward way on a cell structure in space as a consequence of its localisation properties, there also are situations when long-range effects which couple degrees of freedom at arbitrary positions have to be modeled. Whenever this happens, the system often becomes very challenging to model, doubly

so in the particular example of micromagnetism as in addition to long range interactions, stiffness of the system becomes an issue as well.

From the physical perspective, long-range interactions are closely related to very light bosonic exchange particles, with the most prominent and therefore dominant case being that of a massless vector boson such as the photon. Hence, those problems that require the inclusion of long range interactions very frequently can be tackled by employing potential theory in some clever way, with the standard problem being the solution of Poisson's equation with Dirichlet boundary conditions at infinity. This can be regarded as a third class of boundary conditions of great practical importance in addition to the more straightforwardly defined Dirichlet and von Neumann boundary conditions. On unstructured meshes, a widely used technique to solve this class of problems was given by Fredkin and Koehler in XXX. Essentially, their hybrid Finite Element/Boundary Element method boils down to the introduction of a square $N \times N$ matrix, N being the number of boundary degrees of freedom, that encodes all the information about boundary-boundary interactions. For potential problems using first order elements, an analytical formula given by Lindholm that describes the potential of a triangular dipole layer with linearly varying source density can be used to compute the matrix elements. While this technique works sufficiently well to be employed e.g. in micromagnetism, there are a number of problems:

- Even for moderately sized problems, the boundary element matrix may require a considerable amount of memory, as well as setup time. (10 000 surface nodes would require $\sim$ 800 MB of RAM using standard 64-bit floatingpoint element representations). This is especially a problem with thin film geometries.
- Lindholm's formula suffers from cancellation problems in some parameter regimes.

In a FE/BE implementation, the latter problem may be alleviated to some degree by using extended floatingpoint numbers for the computation which then are rounded down, but this often is impractical. A numerically more useful equivalent of the original formula hence would be desirable. Concerning the former issue, there are approximation techniques to reduce memory footprint such as using hierarchical matrices which are based on re-writing propagators $G(x-y) \sim \sum_k c_k G(x) \cdot G(y)$ XXX reference. For presently realistic problem sizes, these can buy about an order of magnitude in terms of memory requirements. By and large, the approaches described here are (up to highly situation-specific tricks) about as good as we can get for this particular type of problem using presently existing technology. The `nsim` prototype supports all the necessary elements to implement boundary element matrix techniques (as is demonstrated in the `nmag` library), but at present does not yet provide any support for quasi-long range interactions, i.e. forces mediated by particles whose mass corresponds to a length scale somewhere between the cell and the mesh size.

3.7 Periodic Boundary Conditions

3.7.1 Periodic identification of DOFs

3.7.2 Long-Range interactions and large scale geometry

3.8 Local non-linear equations

The other elementary building block (apart from linear, differential field operators) of complex field operations is arbitrary (i.e. usually non-linear) *site-local* manipulations of fields. In combination, and through the introduction of intermediate auxiliary fields (such as e.g. H_{eff} in micromagnetism), these basic operations cover the vast majority of field operations encountered in applications.

The notion of *site-local* operations encompasses all manipulations on a set of fields which are to be performed at every mesh site in the same way, and operate on the subfield components of these fields which are located at the specific site only. This basically corresponds to 'dropping position dependency as implicit' in physical notation when giving an equation such as the Landau-Lifshitz and Gilbert equation in the form:

$$d/dt\, M_i = c_1 \epsilon_{ijk} M_j H_{\text{eff}\,k} + c_2 \epsilon_{ijk} M_j \epsilon_{kmn} M_m H_{\text{eff}\,n}. \tag{8}$$

We may want to use site-local manipulations that involve highly unusual operations which cannot be expressed easily in arithmetic notation. (For example, we may want to use differrent manipluations depending on whether some temperature is above or below a critical value.) As a consequence, maximum flexibility demands the capability to allow arbitrary user-specified code to specify local physics. This code nevertheless has to be executed fast, therefore the `nsim` prototype compiler resorts to employing runtime compilation techniques, as explained in section XXX[fastfields].

Before sending user-supplied code to the C compiler, `nsim` wraps it up in a block if `#define`/`#undef` macros which allow the user code to refer to local subfield components directly by their name and indices, regardless of the field data vector they are located in[2]. Also, `nsim` provides macros with which presence of a particular subfield at a given site may be tested, as this may depend on the site in heterostructures.

From the perspective of the user-supplied local code, the `nsim` framework provides a context in which to execute this code over all sites of a mesh on a number of fields (and co-fields) via wrapping definitions, as well as behind-the-scenes bookkeeping over site-dependent offsets into the field data vectors, plus automatic compilation-on-demand.

While it is crucial to provide maximal flexibility to the user through the option to specify arbitrary code for site-wise execution, it is just as important to also provide an alternative interface which, rather than asking the user to write C code, allows specification of equations in algebraic notation. This is achieved

[2]The observed utility of this approach was the underlying reason for implementing 'field aliasing', cf. section XXX

by employing a LALR(1) parser generated by OCaml's equivalent of Yacc from a formal grammar specification and translating algebraic notation to C code internally, which then is provided to the general mechanism. Also, algebraic notation allows auto-generation of derivatives of equations of motion (through some internal symbolic algebra) as needed for the Jacobian which should be known (at least approximatively) to successfully employ some time integrators, such as CVODE. (Cf. sections XXX[jacobi] and XXX[time-integrator].)

3.9 The Equations of Motion and Auto-Generation of the Jacobian (needed by the time integrator)

XXX better title? The Jacobian $d\dot{y}(t)/dy_j dy_k$?

XXX Semi-symbolic Jacobian

A not too uncommon situation is that the physical system under investigation may be elliptic in nature, but numerically challenging due to its stiffness. (XXX am I talking nonsense here?) In micromagnetism, this happens due to a gap of more than two orders of magnitude between the characteristic time scales of the reaction of the system to a single-site perturbation that come from different physical effects (in the simplest case, the weak long-range demagnetisation field and the effective exchange coupling that tries to make cloesly spaced magnetic degrees of freedom align and move in unison). There are a number of code bases that implement time integration algorithms for stiff systems which often can be used in a 'black box' fashion. Normally, the user of such a time integration library has to provide both a function that computes the velocities, given some particular state of the system, as well as a function that computes the Jacobian. Roughly speaking, the Jacobian helps to find a transformation from the fundamental degrees of freedom to a more useful basis which allows to separate fast processes ('ringing' of the system) which often are not excited in physical systems from the more interesting collective slow dynamics.

The Jacobian usually is a function of the configuration and time, and as it is expected to be costly to compute, the time integration library will try to avoid recomputation of the Jacobi matrix as much as possible. (Evidently, some heuristics has to be used here, as it cannot be known beforehand how many processor cycles the computation of the Jacobi matrix will take in a particular situation.)

In principle, the Jacobi matrix can be computed automatically when given the equations of motion. In reality, most physical simulations use hand-written low-level code (usually C or FORTRAN) to implement the computation of the numerical matrix entries. For systems with more complicated equations of motion (such as micromagnetism), the implementation of the code that computes the Jacobian can become a rather tedious and frustrating process that is prone to errors that are easy to make but fiendishly difficult to track down.

For this particular reason, `nsim` provides capabilities to automatically generate the Jacobian from the symbolically specified equations of motion. While it would in principle be possible to just use these as they are and take second derivatives, one discovers in practice that it makes sense to introduce some

additional flexibility in the specification of the Jacobian. One important issue is that one may use equations of motion for the Jacobian (which only needs to be approximatively right) which are slightly different from the system's actual equations of motion. For example, if extra nonphysical terms have been introduced for the sole purpose of numerical stabilization, there may be good reasons to simplify the Jacobian by not including these. (In `nmag`, this is precisely how an additional term to ensure the constant length of magnetisation vectors is treated.) A second type of important simplification that must be supported is to allow the use of approximations to derivatives of dependent fields. In the particular case of micromagnetism, it is useful to approximate $\delta H_{\text{eff}}/\delta M$ by $\delta H_{\text{exch}}/\delta M + \delta H_{\text{anis}}/\delta M$, omitting the contribution from the demagnetising field (which would at the same time be very costly, nonsparse, and not a source of fast dynamics).

The present internal design of the component providing auto-generation of the Jacobian in the `nsim` prototype presumably has not settled down to its final form yet, as it still appears to be somewhat unwieldy in some situations. At present, the interface works in such a way that the user specifies both an equation of motion, such as e.g. a string representation of the Landau Lifshitz and Gilbert equation (XXX ref or give formula again here)

$$dM_i/dt = c_1 \epsilon_{ijk} M_j H_{\text{eff},k} + c_2 \epsilon_{ijk} M_j \epsilon_{kmn} M_m H_{\text{eff}\,n}$$

```
dMdt(i) = c1 * eps(i,j,k) M(j) H_eff(k) + c2 * eps(i,j,k) M(j) eps(k,m,n) M(m) H_eff(n)
```

as well as additional information on contributions to the derivatives of the fields that appear in the equation of motion with respect to the configuration degrees of freedom. Via the python interface, this appears as a list, one entry for each field `[`$\langle$`spec`$dM/dM\rangle$`, `$\langle$`spec`$dH_{\text{eff}}/dM\rangle$`]`, where the leading entry which would give the derivative of the configuration by the configuration is ignored as this is handled internally, and the other entries (only one in this example) each are a list of contributions which will be summed and each either provide a local non-linear (at present, only polynomial) equation or a name of a sparse operator. The former case is used for providing the contribution to $\delta H_{\text{eff}}/\delta M$ that comes from the anisotropy field strength $\delta H_{\text{anis}}/\delta M$, while the latter is used to for providing the $\delta H_{\text{exch}}/\delta M$ contribution.

The system then internally 'precompiles' this information to a 'semi-symbolic Jacobian' data structure which, when provided with information about the state of the primary and dependent fields that enter the equation of motion, will allow the population of the corresponding numerical Jacobi matrix. This 'semi-symbolic Jacobian' provides a representation of a polynomial (with numerical coefficients) in the components of the primary and dependent fields for every entry of the (sparse) Jacobi matrix which can be nonzero. Naturally, this data structure can be distributed across a computing cluster rather easily, however, this feature is not yet supported by the `nsim` prototype.

Recent experimenting has shown that the semi-symbolic Jacobian data structure can become rather large if the system's equations of motion are complicated. In micromagnetism, this may easily happen when high-order anisotropy terms

appear in their most general form. One promising option to deal with this general problem seems to be to slightly re-structure the Jacobian precompiler in such a way that it operates like the differential operator vivificator described in (XXX ref). Then, it could be used to either build a semi-symbolic Jacobian data structure as before, or to directly populate the numerical matrix when given numerical information about the relevant fields, hence trading memory for execution time. It may even be an interesting option to mix these approaches and use just-in-time generation of the Jacobian for some contributions, and precompliation to a semi-symbolic Jacobian for others.

3.10 The 'linear algebra machine' abstraction

XXX actually, this is a misnomer! Should be called 'physics simulation engine'!

3.11 Time Integration

XXX how to handle the length constraints?

3.11.1 (Sundials CVODE) Need better title!

3.11.2 Nonzero Temperature effects through Langevin Dynamics

Rationale: sundials does not like us tying down the step size! Problem: does not interact well with PBC.

3.12 User Interaction

The idea of using computer simulations to study some particular aspect of the behaviour of a physical system intrinsically calls for a certain level of flexibility. One rather obvious reason is that one would like to be able to automatize the processing of simulation results to tables, diagrams, and other images. In addition to this, there are situations where one would like to use a bit of automatized choreography for performing a parametrized sequence of simulation runs. A prime example for this would be to run a series of individual micromagnetic simulations where the externally applied magnetic field strength varies in a controlled way in order to produce some particular switching sequence. Here, one would ideally want to be able to wrap some control and post-processing code around the simulation core in such a way that the mapping of a given switching sequence to a number of diagrams which describe the physics of the process is fully automatized.

3.12.1 Standalone programs vs. libraries

While such situations that basically ask for a Turing-complete level of flexibility are rather widespread, many specialized software packages have been written where the the design mistake of underestimating the need for flexibility at the beginning and trying to mend this progressively in incremental steps has been

repeated over and over again. Typically, such systems gradually evolve from being able to read very primitively structured configuration files towards using a LALR(1) parser in order to deal with more complex configuration files towards supporting some rudimentary form of variables, arithmetics, and control structure. Subsequently, a need arises for container data types such as arrays (which also introduces a number of tricky memory management issues), then file I/O, and finally the capability to interface C libraries and a debugger.[3]

A reasonable approach to avoid this problem is to provide maximum flexibility from the beginning through designing the system in such a way that it provides all of its special capabilities in the form of a software library and interface this to some existing popular programming language with a large user base.

3.12.2 The Python scripting language as an especially attractive option

For `nsim`, the design decision to incorporate Guido van Rossum's 'Python' scripting language (and hence give `nsim` the form of a Python library) was made based on a number of circumstances:

- *Gradablity*: A file that consists of a sequence of lines of the form `parameter = value`, possibly interspersed with comments, already is a valid Python script. Therefore, it is always possible to present a Python interface to the basic user as a parameter configuration file parser with additional capabilities: it is always possible to set up the system in such a way that its most basic use will not require any programming skills (should this be seen as a desirable objective).
- *Availability*: Python is widely available, under a free license, on a large number of platforms.
- *Familiarity*: Python (unlike Lisp/Scheme) has a simple syntax that is easily picked up by anyone who has had previous exposure to virtually any other imperative programming language (such as C, Perl, Matlab, or even the shell).
- *Popularity*: Python both has a sizable user base and is rather accessible for both the inexperienced casual user and the expert.
- *Power*: Python supports enough abstract concepts (e.g. container data structures such as dictionaries) to make the implementation of more sophisticated algorithms for control and data processing a realistic option.

Python support also brings some additional secondary benefits:

[3]This phenomenon, which has been encountered with word processors, CAD software, ray tracers, web browsers, databases, is commonly referred to as *Greenspun's Tenth Rule of Programming*: 'Any sufficiently complicated C or Fortran program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.'

- *Interactivity*: It is possible to use the Python interface in an interactive execute-commands-as-they-are-given fashion to explore and hence get a more intimate understanding of some particular system.
- *Libraries*: There is a large number of third-party Python libraries which easily can be used to provide additional capabilities such as database support, image generation, automatic website generation, and much more.
- *Debugging*: XXX Hans. There is one and it is useful. How should we praise that? Also: 'escape back into ipython'

However, should Python go out of fashion in the not so near future (as so many other scripting languages have done), `nsim` is designed in such a way that it would be possible to include support for another high level language with reasonable implementation effort.

3.12.3 Fast Code and Scripting Languages

One general problem inherent in this scripting language approach is that situations may arise where small pieces of user-specified code, such as e.g. a function describing a scalar potential, or a non-linear local field equation, has to be evaluated so often that only a compiled machine-code version would be sufficiently efficient not to dramatically slow down the simulation. On the other hand, it is highly desirable to retain full programming flexibility rather than artificially restricting the types of functions that can be specified.

There are a number of high level languages which both can be used interactively and produce fast machine code at the same time.[4] However, producing machine code from scripting language code is not a realistic option unless one accepts to tie the system so strongly to one particular user interface language that switching over to a different one becomes highly unrealistic. Therefore, `nsim` pursues a different approach: virtually all situations where user-specified pieces of code in a field theory simulation system have to be fast can be regarded as special cases of routines mapping $\mathbb{R}^n \to \mathbb{R}^m$, potentially with additional internal state. So, the `nsim` simulation compiler provides capabilities to regard a user-specifiable string as a piece of C code and turn this into a callable function. Internally, this works by dynamically generating a C source file which is sent to the C compiler to produce a shared object which then is linked lazily – i.e. when such a function is used for the first time – (using the `dlopen()` and `dlsym()` functions from `libdl`) into the running simulation.

This mechanism – implemented in the `fastfields` module of `nsim` – satisfies some evidently very desirable properties:

- The C compiler is not called unnecessarily: When 100 functions that have to be compiled are defined one after another before one of them is called, only a single call to the C compiler is being made. (This is achieved by lazily deferring compilation to the latest possible point in time.)

[4] Two widely known examples would be the CMU Common Lisp compiler and the SML/NJ system

- Whenever recompilation is initiated, functions that have become inaccessible through forgetting all references to them (i.e. they have become garbage) automatically disappear from the dynamically generated library. (So, re-defining such a function 1000 times during the course of a program will not increase memory usage.)
- At any given time, only a single dynamically compiled shared object module will be linked in, which means that defining and calling new functions (which induces recompilation) will have to unlink and reset the symbol-to-address associations of other dynamically compiled functions (which will once again have to be looked up using `dlsym()` after relinking).

3.13 Parallelization

The capability to use parallel computation both to speed up linear algebra as well as non-linear (site-local) operations as well as to distribute the memory requirements of data structures should be considered more a necessity than a desirable feature. The underlying reason is that parallel computation is widely used in simulations these days, hence a nonparallel simulation framework would suffer from severely restricted applicability in view of the present range of problems studied by computer simulations.

3.13.1 The MPI execution model

Concerning parallelized number-crunching (as opposed to e.g. massive parallel lookup and search operations over clusters of potentially unreliable machines, i.e. search engine technology), the situation that emerged over the last years is strongly dominated by the availability of a flexible unified interface for efficient data exchange between processes partaking in a parallel computation, namely the MPI libraries. Very broadly speaking, MPI-based programs consist of a mixture of 'ordinary' commands as well as 'collective' commands. The role of the latter can be regarded as providing a refinement of the execution semantics of the underlying programming language (most often, C) by defining a kind of 'must happen in unison across all processes participating in the computation' generalized execution flow concept. Traditionally, this is usually achieved by writing MPI-based programs in such a way that all machines run the same program code which executes the same sequence of statements on all cluster nodes, very slightly seasoned only with node-number dependent range selections that ensure every process works on its share of the (often, linear algebra based) distributed workload. In particular, the entire parallel communication choreography is pre-determined right from the start of the program.

We will call this execution model the 'choir model': the songtext distributed at the beginning to all the singers precisely defines when to sing which tune, where to loop (and how often), etc.

The obvious advantage of this model is that it is very easy to comprehend and implement. In principle, more general distributed algorithms could be (and

often are!) far more complicated. For evident reasons, the choir model is the dominant model for parallelized number crunching.

Unfortunately, this kind of execution semantics is fundamentally incompatible with the idea of providing maximum user flexibility. In particular, as soon as there is any way how program flow could depend on nonpredictable external input (such as, for example, provided interactively by a human user), the choir model no longer can be applied. Instead, we must resort to a more sophisticated model where not only the data to be worked on, but also information about the type and order of operations on that data has to be distributed at run time across all the computation nodes.

Furthermore, subtle issues arise in conjunction with dynamical distributed resource management. While memory management for conventional parallel number crunching programs usually is rather trivial, allocating all relevant data structures in the beginning, populating matrices, doing the computation, freeing all resources in the end, the situation is very different for highly dynamical applications. When does a vector which is distributed across all nodes of a cluster become inaccessible, and if this happens, how do we ensure this resource is freed in unison across all the cluster nodes?

The design decision underlying the `nsim` compiler prototype is that the user should not at all have to worry about any issues related to parallel execution, with the one exception of providing information about which part of discretized space is to be associated to which computational node. While it is by now evident that this goal is indeed achievable without noticeable performance loss for the type of physical problems `nsim` was designed for, the pototype is incomplete insofar as that some time-consuming matrix setup operations are not yet parallelized to the fullest possible extent.

3.13.2 Shielding the user from MPI

Ideally, the user should be able to set up simulations that allow parallel execution without ever having to worry about any details associated with multi-machine execution flows. One particular issue is that of debugging: if the complexity associated with parallel execution and collective MPI commands showed up at the user level, this would seriously interfere with the user's capability to inspect and analyze some particular situation interactively in a debugging session. Rather, the user should never have to concern himself with the question which statements he can execute interactively and which he cannot (as they would require coordinated execution of collective commands): things should just work as one would expect with the model of ordinary single-process execution flow in mind.

This is indeed achievable, but requires extensions to distribute not only numerical data across the cluster, but information about excution sequences as well. The basic idea is to let the master node execute the simulation script and make all the other nodes go into 'slave mode' where they receive and subsequently execute commands from the master. The commands understood by the slaves include:

- Shutdown of the system and subsequent termination of the program.
- Parallel linear algebra resource allocation (for vectors, matrices, linear solvers, etc.).
- Command sequence resource allocation: a sequence of operations which manipulate distributed resources is registered under some particular name for later execution (so that later on, a command sequence can be executed across the cluster without having to employ parallel communication to send out every single command as it is executed).
- Execution of a previously registered command sequence.
- Parallel (linear algebra and command sequence) resource de-allocation.

It is important to note that, while the slave processes contain all the machine code of the core `nsim` executable, they do not run the user's program but are driven exclusively by commands sent via MPI from the master node.

One could call this the 'organ model' (in contrast to the 'choir model') of parallel execution: the computation utilizes hardware that is spread out over a large area, and the musician has full freedom to interweave both programming (using organ stops) and playing at will from a central console.[5]

What is rather helpful here is that the core of `nsim` is based on the OCaml system, which provides a proper substitution-aware function concept that is much more powerful than the more simplistic concept of a 'routine' (which is employed by most compiler languages, such as C, and sometimes confusingly called a 'function' there as well). In particular, OCaml supports (as an experimental but highly useful feature) the serialization of functions to strings in such a way that only ('closure') parameters, but no machine code, have to be sent over the network in order to make a function which was defined on the master available on the slave nodes as well. This is used e.g. to gain maximum flexibility in distributing functions that manipulate intensive parameters across the cluster. However, one consequence is that `nsim` presumably will not work on heterogeneous multi-architecture clusters, as one should not expect OCaml data structure serialization to be able to cross architecture boundaries.

A `nsim` script that provides information how to distribute a mesh across a cluster can be either started in single-process mode

```
$ nsim simulation.py
```

or under MPI control (here, on four nodes) to execute in parallel:

```
$ mpirun -np 4 nsim simulation.py
```

Unfortunately, MPI (at least in the mpich implementation which is used by `nsim`) is somewhat ideosyncratic in the way it provides parallelism to C programs: what it effectively does is to wrap another pre-startup layer around the

[5] We would have used Franz Lisp's "Allegro Composer™" instead of Objective Caml as an implementation base for `nsim` had we found a way to teach it to our grad students.

program's `main()` function which parses and interprets MPI-related command arguments. This is then used in conjunction with a utility for remote login (such as `ssh`) for automated coordinated startup of individual processes. This has the unfortunate effect that it is not possible to start a python program in parallel using the standard python interpreter: Python does its own argument handling and gets confused by the extra program arguments introduced by MPI.[6]

The immediate consequence is that `nsim` cannot be brought into the form of an extension of a standard Python interpreter. Rather, `nsim` has to use `libpython` to implement Python support and provide an own toplevel. Unfortunately, `libpython` does not provide the `Py_Initialize()` function which is used in the conventional Python interpreter to parse Python-specific startup arguments. This means that `nsim` has to provide of its own the corresponding python argument handling capabilities to the degree needed.

It is important to note that the startup code of the `nsim` executable has to discern between a number of different situations: using `nsim` to run a user simulation program whose filename is given as a command argument induces a different startup mode than starting `nsim` interactively. Furthermore, both interactive and script mode may be used when running `nsim` in a distributed fashion under MPI control. (Interactive MPI-based parallel execution is a highly uncommon feature.) Note that regardless of whether script or interactive mode is invoked, slave processes will have to go through special startup code to enter slave mode.

3.13.3 Centrally coordinated parallel linear algebra

The 'organ' execution model described in the previous section requires the distribution and management of both linear algebra resources (vectors, matrices, linear solvers) and control resources (execution sequences of linear algebra operations). For the `nsim` prototype, the PETSc library (XXX ref) has been chosen to provide parallel linear algebra capabilities. In MPI parlance, PETSc creates an own communicator data structure to separate its own parallel communication from other MPI-based communication the program may also have to perform. The `nsim` core uses an additional communicator for its own coordination and management-related messages.

The most important issue with dynamic parallel execution is dynamic resource management: both Python and the OCaml system provide automatic resource de-allocation capabilities (of different sophistication and hence quality) that have to be extended to parallel resources. Fortunately, it is not necessary to separately address interference issues of Python's dynamical memory management with parallel resources, as all parallel resources can be managed internally by the OCaml runtime system and proper behaviour with respect to Python code is then induced automatically through the Python-OCaml interface.

[6] One should point out that the same holds for virtually any other programming language as well and basically is a `mpich` design flaw which could be avoided e.g. by using an environment variable rather than extra call parameters.

The basic idea underlying dynamical resource management is that as the master node has the sole responsibility for execution flow, garbage collection on the master node will be the sole authority on dynamic de-allocation of distributed resources. This is effected through addressing all distributed resources in an uniform way through *dynamic resource handles* which basically represent names of dynamical resources to which a finalizing function is associated – on the master machine only – that will be executed upon garbage collection. Every process participating in a parallel computation (including the master) maintains a dictionary that maps dynamic resource handles to the actual resources (i.e. this machine's share of a distributed vector, matrix, or a distributed script, etc.) When one dynamical resource (i.e. a command sequence) refers to another one (i.e. a vector), it does so by using this dynamical resource handle, so garbage collection of the command sequence resource can induce the de-allocation of other resources referred by it.

On the slave nodes, dynamic resource handles are nothing else but keys into a dictionary containing the corresponding resources. They are being managed (i.e. allocated and de-allocated) exclusively according to resource management commands received from the master. Here it is important to know that the OCaml data serializer works in such a way that turning an entity such as a distributed resource handle into a string and sending it over the network will strip it from its finalizers. Hence, parallel command sequences can be serialized and distributed from the master to the slave processes conveniently without having to worry about spurious finalizers being introduced on the slaves.

On the master, de-allocation of a distributed resource has to induce sending out the corresponding de-allocation commands to the slave nodes, as this may involve the execution of collective MPI commands. As garbage collection can and will happen at arbitrary points throughout the execution of a program, for example in the middle of a registered sequence of linear algebra commands which involves MPI-collective operations before and after the garbage collection (XXX language - repetition of the term GC), the master *must not* induce distributed de-allocation from within a distributed resource handle finalizer (as this may upset the synchronity of the sequence of MPI-collective commands across machines), but instead defer the actual sending of de-allocation requests to the next point in time when slaves are again ready to receive commands from the master. The obvious drawback of this approach is that there are situations where distributed resources refer to other distributed resources in such a way that de-allocation cannot happen simultaneously but will artificially be spread out over multiple cycles of garbage collections and entries to the parallel command distribution code. For real world applications, however, this does not pose a problem.

Evidently, the resource tables that map distributed resource handles to the actual resources must not use the actual distributed resource handles for which finalizers were registered as keys (otherwise their presence in these tables would prevent them from ever being collected!) While one may employ weak reference tables to circumvent this problem, a simpler approach is to use copies of the distributed resource handles as table keys which do not have finalizers registered.

3.14 Implementation of the `nsim` core capabilities

XXX maybe move this section?

Underneath the user interface, `nsim` has to provide not only numerical capabilities, but also some rather sophisticated bookkeeping as well as symbolic transformations. Evidently, it is highly desirable to have these computationally demanding parts implemented in a fast compiler language. The major criteria that influence the choice of a viable compiler to base such a system that is developed in an academic environment on are:

- The compiler should be sufficiently well supported to get feedback when low level technical issues such as interaction with MPI arise.
- The compiler should be able to produce efficient code for the most widespread platforms.
- The language should provide basic capabilities that are highly useful when implementing symbolic algorithms, including lambda abstraction, closures, tail recursion, and a proper garbage collector.
- The language should be sufficiently easy to pick up for new PhD students (which may be physicists or engineers rather than computer scientists) on the project that they can start to be productive very early.
- The language should allow interfacing specialized C libraries (e.g. for parallel sparse matrix linear algebra) without too much effort.

The `nsim` authors found that the Objective Caml (OCaml) system (XXX ref) satisfies these criteria very well. In addition, when the implementation of the `nsim` compiler started, there already was an OCaml-Python interface available (XXX ref) that provided a very tight coupling between the chosen user interface and the core implementation language, even to the extent of making it possible to wrap up OCaml functions to make them callable from Python and vice versa. The implementation of `nsim` only uses a subset of OCaml language features which is what is known as 'Core ML' plus a few extensions. The underlying reason is that this subset of the full language on the one hand already is sufficiently powerful to tackle practically all problems that arouse in the implementation of `nsim`, and on the other hand is sufficiently simple to be mastered by students within about three months.

During the implementation of the `nsim` prototype, the original 'Pycaml' module (which also is readily avaliable as 'pycaml' package in the popular Debian GNU/Linux distribution) was extended considerably by introducing new functions to simplify writing Python extensions in OCaml. Also, a number of memory management related problems had to be fixed. One of the strongest benefits of this module is that it magically handles all memory management related issues that have to be addressed in order to make the OCaml garbage collector and Python reference counting mechanism happy at the same time. This is a short example that demonstrates how to extend Python with a fast

(i.e. compiled to machine code) implementation of Euler's Gamma function $\Gamma(x)$ based on extended Pycaml:

```
=== OCaml ===

let rec euler_gamma =
  (* Accurate to eps_rel = 2e-10; See http://www.rskey.org/gamma.htm *)
  let pi = 4.0*.(atan 1.0) in
  let constants =
    [|75122.6331530; 80916.6278952; 36308.2951477;
      8687.24529705; 1168.92649479; 83.8676043424;
      2.50662827511|]
  in
  let nr_constants = Array.length constants in
    fun x ->
      if x<0.0
      then -.pi/.(x*.(euler_gamma (-.x))*.(sin (pi*.x)))
      else
let rec compute_sum_prod sum prod pow_x n =
  if n=nr_constants then sum/.prod
  else
    compute_sum_prod
      (sum+.constants.(n)*.pow_x)
      (prod*.(x+.(float_of_int n)))
      (pow_x*.x) (n+1)
in
let x5 = x+.5.5 in
  exp(-.x5)*.(x5**(x+.0.5))
  *.(compute_sum_prod 0.0 1.0 1.0 0)
in
let _py_gamma =
  python_interfaced_function [|FloatType|]
    (fun py_args ->
       pyfloat_fromdouble (euler_gamma (pyfloat_asdouble py_args.(0))))
in
  register_for_python [|("gamma",_py_gamma)|]
;;

=== Python ===

>>> ocaml.gamma(0.5)
1.7724538509046035
```

XXX operator compiler may already have been explained elswere as well! One of the key components of `nsim` in which the functional/symbolic capabilities of OCaml play a prominent role is the two-stage compiler of symbolic operator strings to distributed sparse matrices. The first stage uses a LALR(1) parser to analyze the operator string and turn the symbolic operator description and finite element specifications (which involve functions given as analytical expressions)

into intermediate 'thunks', which, when applied to a simplex specification (which provides spatial coordinates), will add the corresponding entries to the sparse operator matrix. Similar techniques are also used to automatically construct the Jacobian (needed for the CVODE time integrator) from the equations of motion by taking second derivatives.

4 Fundamental limitations of the approach

Number of degrees of freedom (large?)

5 Restrictions of the `nsim` prototype

XXX Bla bla... Why and what for...

For the purpose of providing an overview, we summarize the open issues that have not been implemented in the first public release of the `nsim` prototype. Most of these have been described in more detail in the preceding text.

- So far, `nsim` only supports discretisation of classical field theory based on finite elements. As most of the concepts as well as the design are readily generalized to finite difference type discretisations of space, and as most of the vital infrastructure already is present, it would be highly desirable to extend `nsim` with finite difference support. This is conceptually rather straightforward, but nevertheless a considerable amount of work.

- The `fastfields` dynamic linking mechanism at present easily gets confused if it is ever fed broken C code. Ideally, calling dynamically compiled function for which a broken definition was given should throw an exception which contains both the compiler error message and some context from the code provided by the user. This is achievable, but has not been implemented yet.

- Error reporting can (and eventually must) be improved considerably. At present, there are many situations where a very minor mistake in a simulation specification produces a highly cryptic error message (such as "index out of bounds") that is hard to track down.

- While parallelization works and nicely demonstrates that it is indeed achievable to employ parallel linear algebra while at the same time completely shielding the user from the associated complexity, present support is somewhat incomplete. In particular, matrix setup is not yet fully parallelized.

- Support for dynamic rebuilding of sparse and nonsparse operator matrices would be highly desirable. This would allow much more straightforward support for problems such as the simulation of conductivities with nonlinear dependency on electrical current, or to perform parameter studies

far more easily in which the relative placement of components is changed dynamically.

- At present, field dependencies must be specified by the user. While it would be rather straightforward to auto-derive the field dependencies from the specifications of the linear algebra execution sequences as well as the user-specified local equations, this has not been implemented yet.
- XXX Need integrated toplevel physics language w. LALR(1) parser!
- Startup argument handling of `nsim` still is rather ad-hoc. Basically, there are six different kinds of program arguments that have to be dealt with:
  - MPI startup arguments
  - PETSc related arguments
  - Arguments controlling the Python interpreter
  - `nsim` arguments (controlling e.g. the nsim loglevel)
  - `nsim` library arguments (e.g. when using `nmag`)
  - Arguments to the user's application.

  The `nsim` team has an idea for an approach that should be able to handle this complexity in a convenient way, but this has not been implemented yet.
- An important limitation of our extended Pycaml Python-OCaml interface module, which was released separately as a spin-off of the `nsim` project is that it is not thread-safe with respect to Python threads. While this has no relevance to the `nsim` project at present, the wider community may nevertheless benefit from making this module also thread-safe.
- XXX Middle fields not available in the first release of the `nsim` prototype, although rudimentary support is present.
- XXX No support yet for (quasi-)long range interactions that are mediated by other exchange particles than massless vector bosons.
- XXX Introduce bypasses!
- The code auto-generating the Jacobian from the equations of motion should be re-structured in a more abstract way so that it can either pre-compile a semi-symbolic Jacobian or be used as a vivificator, employing numerical data on field strengths to directly populate a sparse matrix (as described in section XXX).