DiffMan
an object oriented MATLAB toolbox for solving differential equations on manifolds

Kenth Engø[*]
Department of Informatics, University of Bergen
N-5020 Bergen, Norway

Arne Marthinsen[†]
Department of Mathematical Sciences, NTNU
N-7034 Trondheim, Norway

Hans Z. Munthe-Kaas[‡]
Department of Informatics, University of Bergen
N-5020 Bergen, Norway

ISSN 0333-3590 **REPORT NO 164** **February 1999**

Department of Informatics, University of Bergen, Norway

Abstract

We describe an object oriented MATLAB toolbox for solving differential equations on manifolds. The software reflects recent development within the area of geometric integration. Through the use of elements from differential geometry, in particular Lie groups and homogeneous spaces, coordinate free formulations of numerical integrators are developed. The strict mathematical definitions and results are well suited for implementation in an object oriented language, and, due to its simplicity, the authors have chosen MATLAB as the working environment. The basic ideas of *DiffMan* are presented, along with particular examples that illustrate the working of and the theory behind the software package.

AMS Subject Classification: 65L06, 34A50

Key Words: geometric integration, numerical integration of ordinary differential equations on manifolds, numerical analysis, Lie groups, Lie algebras, homogeneous spaces, object oriented programming, MATLAB, free Lie algebras

[*]Email: `Kenth.Engo@ii.uib.no`, WWW: `http://www.ii.uib.no/~kenth/`
[†]Email: `Arne.Marthinsen@math.ntnu.no`, WWW: `http://www.math.ntnu.no/~arnema/`
[‡]Email: `Hans.Munthe-Kaas@ii.uib.no`, WWW: `http://www.ii.uib.no/~hans/`

Contents

1 Introduction

DiffMan is an object oriented MATLAB [20] toolbox designed to solve differential equations evolving on manifolds. The current version of the toolbox addresses primarily the solution of ordinary differential equations. The solution techniques implemented fall into the category of geometric integrators – a very active area of research during the last few years. The essence of geometric integration is to construct numerical methods that respect underlying constraints, for instance the configuration space of a mechanical problem, and to render correctly geometric structures and invariants important to the underlying continuous problem.

The main motivations behind the *DiffMan* project are:

- To create a uniform environment where new geometric integration methods can be developed and compared.
- To provide researchers outside the field of geometric integration with a tool where they can become familiar with these methods and apply them to new problems.
- To serve as a tool for investigating the role of abstractions and coordinate free formulations in numerical software. *DiffMan* is based on high level abstractions aimed at modeling continuous mathematical structures, in a manner that is, to a large extent, independent of particular representations. This is a novel approach to numerical computing which cannot

be investigated on a purely theoretical level. The development of practical software is an essential part of this work.

DiffMan is developed by support through the *SYNODE* project and the project *Coordinate Free Methods in Numerics*, both projects partly sponsored by the Norwegian research council NFR. Information about the SYNODE project, papers and other links are found at

`http://www.math.ntnu.no/num/synode/`

DiffMan can be down-loaded from the *DiffMan* home-page at URL:

`http://www.math.ntnu.no/num/diffman/`

2 Ordinary differential equations on manifolds

DiffMan is based on recent developments within the area of generalized ordinary differential equation solvers (Lie group integrators). The framework of these solvers is based on concepts from differential geometry. There exist a large number of excellent texts that cover introductory differential geometry; among them are [1, 2, 18, 32, 33]. We will in this section briefly review some of this material, first in an intuitive informal manner, then in a more precise language, and finally we will illustrate the theory by some examples.

2.1 Intuitive concepts

The basic objects involved in the current Lie group integrators in *DiffMan* are:

- The domain where the equation evolves is a *manifold*, $\mathcal{M}$, which should be thought of as a linear or non-linear space looking *locally* like $\mathbb{R}^d$. Globally it might be very different (e.g. a sphere looks locally but not globally like $\mathbb{R}^2$).

- We assume the existence of a set of operators G generating smooth motions on $\mathcal{M}$. The operators in G can be composed and inverted. G is called a Lie group, and the set of motions is called a group action. The group action is used to advance the numerical solution on $\mathcal{M}$. The action should be 'easy' to compute, should be able to generate movements in the direction the differential equation evolves, and should somehow capture some important underlying feature of the differential equations. If the action e.g. preserves the angular momentum of a mechanical system, then also the numerical methods based on this action will preserve this property. One might compare an action in the integration of differential equations with a preconditioner in an iterative solution of a linear equation system. Both the action and the preconditioner should be easy to compute and they should somehow capture some essential features of the system one wants to solve.

- The Lie group G is associated with a linear space $\mathfrak{g}$, the Lie algebra of the group. The algebra serves two different roles:

  - It represents locally all the tangents to G at any point. By differentiating the action, we obtain also a representation of tangents to $\mathcal{M}$ at any point. This gives us a *canonical representation* of any differential equation on $\mathcal{M}$.

  - It can also be used as a flat space to represent a local region on $\mathcal{M}$ where the differential equation evolves. Several of the new algorithms are based on (locally) 'pulling' the equation back from $\mathcal{M}$ to $\mathfrak{g}$, solving it on $\mathfrak{g}$, and pushing this local solution onto $\mathcal{M}$ to advance the solution there.

- In the pull back – push forward process described above, one can make many choices that affect the numerical solution process, but not the canonical representation of the differential equation on $\mathcal{M}$. Some choices are:
  - Numerical method on $\mathfrak{g}$. Runge–Kutta [23] methods and multi-step methods [9] might be used.
  - A smooth local mapping from $\mathfrak{g}$ to G is composed with the action of G on $\mathcal{M}$ to pull the equation from $\mathcal{M}$ to $\mathfrak{g}$. A simple choice is the exponential mapping [23]. Other choices are products of exponentials [29] and for quadratic groups Cayley transforms [15, 5, 19]. To obtain time-symmetric integrators one may use versions of these coordinates centered at some (time symmetric) mid-point [8].

To sum up, this theory allow us to make a lot of choices with respect to representation of the differential equations and numerical solution techniques. A major goal of *DiffMan* is to provide a toolbox where various domains can be easily created and where one can juggle around with all possible combinations of numerical solution techniques.

2.2 Definitions

We will define the concepts above more precisely. A manifold is a topological space $\mathcal{M}$ equipped with continuous local coordinate charts $\phi_i : U_i \subset \mathcal{M} \to \mathbb{R}^d$ such that all the overlap charts $\phi_{ij} : \mathbb{R}^d \to \mathbb{R}^d$ are diffeomorphisms. The overlap charts (transition functions) ϕ_{ij} are defined as $\phi_j \circ \phi_i^{-1}|_{\phi_i(U_i \cap U_j)}$, where $\phi_i^{-1}|_{\phi_i(U_i \cap U_j)}$ means the restriction of ϕ_i^{-1} to the set $\phi_i(U_i \cap U_j)$. If p is a point in $\mathcal{M}$, we denote by $\mathrm{T}\mathcal{M}|_p$ the tangent space of $\mathcal{M}$ at p. The tangent bundle of $\mathcal{M}$ is $\mathrm{T}\mathcal{M} = \bigcup_{p \in \mathcal{M}} \mathrm{T}\mathcal{M}|_p$. A vector field, X, on a manifold is a section of its tangent bundle, i.e. to each point $p \in \mathcal{M}$ it associates a vector $X(p) \in \mathrm{T}\mathcal{M}|_p$. In *DiffMan* we use the term *domain* interchangeably with the term differentiable manifold.

A Lie group is a smooth manifold equipped with a group structure such that group multiplication and group element inversion are smooth maps. Every group has an identity element, which we will denote by e. An example of a well known Lie group is Euclidean space, $\mathbb{R}^n$, equipped with the group operation $+$ (vector addition). The inverse element of $u \in \mathbb{R}^n$ is $-u$ and the identity element is $e = 0$. In this case the group operation is both associative and commutative, but in general it is not commutative.

A Lie algebra is a vector space, $\mathfrak{g}$, equipped with a bilinear, skew-symmetric form, $[\cdot,\cdot] : \mathfrak{g} \times \mathfrak{g} \to \mathfrak{g}$, satisfying the Jacobi identity,

$$[u,[v,w]] + [w,[u,v]] + [v,[w,u]] = 0, \qquad u, v, w \in \mathfrak{g}.$$

We call $[\cdot,\cdot]$ the Lie bracket on $\mathfrak{g}$. When the vector space $\mathfrak{g}$ is $\mathbb{R}^{n\times n}$, a Lie algebra is formed by taking the usual matrix commutator as the Lie bracket. The Lie algebra of a Lie group, G, can be defined as the tangent space at the identity, $\mathfrak{g} = \mathrm{T}G|_e$, equipped with a Lie bracket

$$[u,v] = \left.\frac{\partial^2}{\partial t \partial s}\right|_{t=s=0} g(t)h(s)g^{-1}(t),$$

where $g(t), h(s) \in G$ are two curves such that $g(0) = h(0) = e$, $g'(0) = u$, and $h'(0) = v$. The general linear group, denoted by $\mathrm{GL}(n)$, is the Lie group that consists of non-singular $n \times n$ matrices. The Lie algebra of this group, $\mathfrak{gl}(n)$, is the vector space $\mathbb{R}^{n\times n}$ equipped with the matrix commutator as Lie bracket.

A (left) action of a Lie group G on a manifold $\mathcal{M}$ is a smooth mapping $\Phi : G \times \mathcal{M} \to \mathcal{M}$ such that $\Phi(e,m) = m$ for all $m \in \mathcal{M}$ and $\Phi(g, \Phi(h,m)) = \Phi(gh, m)$ for all $g, h \in G$ and $m \in \mathcal{M}$. Any

$\xi \in \mathfrak{g}$ specify a tangent $\xi_m \in \mathrm{T}\mathcal{M}|_m$ at any point $m \in \mathcal{M}$ via

$$\xi_m = \left.\frac{\mathrm{d}}{\mathrm{d}t}\right|_{t=0} \Phi(g(t), m),$$

where $g(t) \in G$ is a curve such that $g(0) = e$, $g'(0) = \xi$. One may write $g(t) = \exp(t\xi)$, where $\exp : \mathfrak{g} \to G$ is the exponential map [23], or $g(t) = \phi(t\xi)$ for any smooth function $\phi : \mathfrak{g} \to G$ such that $\phi(0) = e$, $\phi'(0) = I$. This gives an identification $\xi \mapsto \xi_m \in \mathrm{T}\mathcal{M}|_m$, which depends on the choice of action Φ, but not on the particular choice of ϕ. In *DiffMan* it is assumed that the differential equation to be solved is presented in the following *canonical form*

$$y' = F(t, y) = \xi(t, y)_y, \qquad y(0) \in \mathcal{M}, \tag{1}$$

for some function $\xi : \mathbb{R} \times \mathcal{M} \to \mathfrak{g}$ (see [28, 23, 5]). If $\xi(t, y) = \xi(t)$, the equation is of *Lie type* or *linear type*. Otherwise it is of *general type*. Equations of Lie type can be handled more efficiently than general type equations.

If the domain $\mathcal{M}$ and the algebra $\mathfrak{g}$ are provided in *DiffMan*, or if they can be constructed using the functors in Section 5.1, the user only needs to supply the function ξ. Otherwise *DiffMan* can be extended by adding new domains.

A large class of numerical algorithms are based on the assumption that the maps $\Phi(\phi(\xi), m)$ can be computed efficiently. The RKMK methods in [23] are based on $\phi(\xi) = \exp(\xi)$. More general ϕ's are discussed in [5].

The following commutative diagram, which is thoroughly discussed in [5], illustrates the relations used as a framework these algorithms:

$$\begin{tikzcd}
\mathrm{T}\mathfrak{g} \arrow[r, "\mathrm{T}\Phi_{y_0} \circ \mathrm{T}\phi"] & \mathrm{T}\mathcal{M} \\
\mathfrak{g} \arrow[u, "\mathrm{d}\phi_u{}^{-1} \circ \xi \circ \Phi_{y_0} \circ \phi"] \arrow[r, "\Phi_{y_0} \circ \phi"'] & \mathcal{M} \arrow[u, "F(y) = \xi_y"']
\end{tikzcd}$$

Here $y_0 \in \mathcal{M}$ is the initial point, $u \in \mathfrak{g}$, $\Phi_{y_0}(u) = \Phi(u, y_0)$ and $\mathrm{d}\phi_u{}^{-1} : \mathfrak{g} \to \mathfrak{g}$ is the right trivialized tangent of ϕ. If $\phi = \exp$, then $\mathrm{d}\phi_u^{-1} = \mathrm{d}\exp_u{}^{-1}$ can be expressed in terms of Lie brackets. For more general ϕ one needs an efficient algorithm for computing $\mathrm{d}\phi_u{}^{-1}$, see [5, 29] for different choices. This yields the following algorithm:

Algorithm 2.1 *Given a differential equation in the form (1), this algorithm produces a q'th order approximation to the solution, using step size h:*

- *Find an approximation $u_1 \approx u(t_0 + h)$ by integrating the following differential equation on $\mathfrak{g}$*

  $$u' = \mathrm{d}\phi_u^{-1}\big(\xi\big(t_0 + t, \Phi(\phi(u), y_0)\big)\big), \qquad u(t_0) = 0,$$

  from t_0 to $t_0 + h$ using one step with a qth order Runge-Kutta method.
- *Advance the solution on $\mathcal{M}$ to $y_1 = \Phi(\phi(u_1), y_0))\big) \approx y(t_0 + h)$.*
- *Repeat with new initial values $y_0 := y_1$, $t_0 := t_0 + h$.*

2.3 Examples

We will here just provide the most basic examples of differential equations written in the form (1). First, if $\mathcal{M} = \mathbb{R}^n$ and $\Phi(g, m) = g + m$, then $\xi_m = \xi$ and (1) acquires the well known form

$$y'(t) = \xi(t, y), \qquad \xi \in \mathbb{R}^n.$$

Another example that is frequently used to present Lie group methods is the matrix case, $G \subset \mathrm{GL}(n)$, a matrix group, $\mathcal{M} = G$ and $\Phi(g, m) = gm$ then $\xi_m = \xi m$ (matrix products), hence (1) becomes

$$y'(t) = \xi(t, m)y(t).$$

In Appendix A.3 we give an example where $G = \mathrm{SO}(n)$, the set of orthogonal matrices, $\mathcal{M} = S^n$, the n-sphere (represented as n vectors with norm 1), and $\Phi(g, m) = gm$ (matrix–vector product). This yields the form

$$y'(t) = \xi(t, m)y(t),$$

where ξ is a skew–symmetric matrix.

In [23] it is shown how isospectral flows, exponential integrators for stiff systems, Riccati equations and the setting of rigid frames can be phrased in this form. Isospectral problems is discussed in detail in [35]. Lie–Poisson equations are discussed in [6] and the use of Toeplitz actions on heat equations in [26]. In forthcoming papers, we will see how many other systems can be viewed in this context.

3 Object orientation

As of today there is really no tradition in the numerical analysis community for using object oriented languages in the development of numerical software. While object oriented languages have become ubiquitous in the computer science community, the classical computational mathematicians have been reluctant in adopting modern computer languages. We do not want to speculate about the reason for this, but merely observe that most industry based numerical software is coded in FORTRAN.

The terms 'class' and 'object' are very likely to be among the first words that you encounter in reading a book about object orientation. For a computer scientist these terms are as natural as the Butcher tableau is for a numerical analysist. In the next section we will give you a very rudimentary explanation of these terms along with ideas describing why modern computer languages are so well suited in capturing abstractions so omnipresent in pure mathematics. A somewhat random reference on object orientation found in the authors' library is [16].

3.1 Object orientation and mathematics

A class is simply a collection of 'elements' with equal properties. In mathematical terms one would say that a class is a set. The 'elements' of a class are usually called objects, and the common properties of the objects specify the class. Mathematically, the properties of a class can be stated as relations that the objects must satisfy in order to be a member of the class.

A very important and interesting aspect of object orientation is that it allows for information hiding. An object typically consists of a public and a private part. The public part can be accessed from the outside of the class, whereas the private part cannot. This enables us to hide implementation specific issues for the particular class in the private part, and can easily make changes to it, without altering the public interface of the class.

This then, naturally brings up the issue of specifying a class. A class specification can be divided up into a 'what' part and a 'how' part. 'what' describes the interaction of the class with the surroundings; what is the public interface of the class, what is the class supposed to do? 'How' the class is implemented is an issue related to the private section of the class. The surroundings do not need to know about implementational issues as long as the interaction of the class is as specified by the public interface.

This distinction between 'what's and 'how's of objects (elements) is ubiquitous in pure mathematics. This is also the reason why abstract mathematical concepts are so well suited for implementation in object oriented programming languages, see [12, 25]. Coordinate free constructions in mathematics, e.g. tensors, try to capture what the operation of an object is regardless of the coordinate system. The tensor class is then specified by properties independent of the coordinate systems, and the different choices of coordinates used in an actual implementation on a computer is deferred to the private part of the class. Hence, a specification of a class emphasizes and extracts the important features of a class, and this conforms very well with algebraic techniques so rampant in pure mathematics.

Thinking in these terms give rise to the rather contradictory term 'coordinate free numerics' [24, 25]. What is a coordinate free algorithm? The whole idea is to device algorithms specified by algebraic operations not depending on the particular representation of the object. All the methods in *DiffMan* are defined on groups and they are all very good examples of coordinate free algorithms. The group elements can have very different representations, but the algorithms are all expressed through algebraically defined operations such as group multiplication and Lie-group actions.

3.2 Object orientation in MATLAB and *DiffMan*

The intention of this section is to give you an overview of the workings of the object oriented environment in MATLAB and how this is used in *DiffMan*. This account is by no means complete, and we refer the interested reader to the MATLAB manual [20] and the *DiffMan* manual [7] for more information.

In MATLAB a class is defined by creating a directory `@myclass`, where `myclass` is the name of the class. The prefix `@` is needed in order to tell MATLAB that this is a class directory. All the public class functions are put in this directory, while the private class functions are put in a directory `@myclass/private`. Where the file is located within the class directory tree distinguishes the m-file from being a public or private function.

Every class must have its own unique constructor. The constructor is implemented in an m-file called `myclass.m`, the same name as the class itself. In MATLAB a class object is represented as a MATLAB struct, where a struct is the same as a struct in C or a record in PASCAL. This struct can have an arbitrary number of fields. To turn a MATLAB struct into an object of `@myclass` the function `class` must be called within the constructor m-file:

```
obj.field1 = n1;
obj.field2 = n2;
obj = class(obj,'myclass');
```

The user can not access the structure fields of the object directly in MATLAB. An attempt doing this will result in an error. Hence, the fields of the object struct can be viewed as part of the data representation of the object, and is private to the class. For the user to interact with the information contained in the fields of the object struct, the class must have implemented public m-files particularly doing this.

Public functions making up the interface of a class are naturally divided up into three categories: **constructors**, **observers**, and **generators**. In MATLAB there is only one constructor, but in other object oriented programming languages, like C++, it is possible to have more than one constructor. The observers of a class are the public functions that extract information from the class objects without altering the object itself. The generators of a class are those public functions that change properties of the class objects, or create new objects of the same or other classes. In *DiffMan* you will typically find this partition of the public functions when reading a class specification.

In *DiffMan* the object orientation is applied in several different ways. The domain points (elements of a manifold) are treated as members of a class. Depending on the specific properties of the domain, there are several *types* of domains implemented in *DiffMan*. Each of the different types of domains are collections of algebraically similar domain classes. The integration methods used to solve the ODEs are called time steppers, and the different time stepper methods are treated as different classes. Flows and vector fields are also implemented as two classes.

In *DiffMan* 1.5 there are three categories of domains implemented: Homogeneous spaces, Lie algebras, and Lie groups. Each domain category is further divided up into domain classes of that particular type. Hence, each of the classes within a particular domain type have similar characteristics, but there are differences that partition them into individual classes. These similar characteristics of the classes of a specific domain type are what defines the domain category. Trying to define and specify the domain category is done through the introduction of a virtual superclass in each domain category. The virtual superclass defines and takes care of operations that are common to all the classes in the domain category. This is obtained through the concept of **inheritance** in object orientation. The virtual superclass is the parent class for all the other classes in the domain category, and all the child classes inherit the parents' functions. This means that one can apply the public functions of the virtual superclass to an object of a child class. If the child class needs specifically implemented versions of any of the public functions of the parent class, this is achieved through **overloading**. Supply a public function to the child class with a matching name, and MATLAB will use this version of the public function instead of the one supplied by the parent class.

4 Objects in *DiffMan*

In *DiffMan* you will encounter 4 different types of objects: **domain**, **field**, **time stepper**, and **flow** objects. Each object is defined in such a way as to capture the mathematical essentials of a manifold, the field defining the differential equation, the numerical time stepper algorithm, and the flow operator, respectively. We will discuss each one of these objects in the following.

4.1 The domain object

Every domain object (e.g. objects of the type Lie algebra, Lie group, or homogeneous space) is built up as a MATLAB struct with two fields: `shape` and `data`. Generically, every domain object is represented as:

```
domainobject =
        shape:
         data:
```

A domain object specifies a specific point in a specific manifold. It is often useful to create a single class for representing a family of manifolds, e.g. all Lie algebras $\mathfrak{gl}(n)$ are represented by the same class `@lagl`. The shape specifies the particular manifold in the family (in this case n), while the data part represents a particular point in this manifold (in this case n by n matrices. The shape is in computer science called a **dynamic subtyping** of the class. If an object has an empty data field, it is taken to just represent the space (the subtype).

A second example is the dynamic subtyping of the homogeneous space `@hmlie`. This is the homogeneous space obtained by any Lie group acting on itself by left multiplication. Considering all the different Lie groups and Lie algebras, the shape is chosen to be an object of the particular group or algebra. Since all Lie groups and Lie algebras themselves are dynamically subtyped, the shape of `@hmlie` must be a Lie group or Lie algebra object with a preset shape. This is because we need to know a 'size' measure on the domain objects that are acting on themselves.

The user cannot directly access the contents of the `shape` and `data` fields of a domain object, since the fields belong to the private part of the domain class. In order to do this the user must use the public functions `getshape` and `getdata` to return the contents of the fields, and `setshape` and `setdata` to update the values of the private fields.

4.2 The field object

A field is defined over a manifold. Some examples of fields are vector fields, tensor fields, and divergence free vector fields. A vector field is a mathematical construction that to every element of the manifold assigns a vector. Likewise; for a tensor field a tensor is assigned to each element of the manifold. From this it is natural to conclude, since the output from different fields is not similar, that a generic field object only contains information about the manifold over which the field is defined:

```
fieldobject =
      domain:
```

To define an ordinary differential equation we only need the notion of a vector field. Tensor fields are mainly used in partial differential equations. *DiffMan* 1.5 is only devoted to the solution of ODEs evolving on manifolds. Hence, the only field class implemented is `@vectorfield`. The generic representation of a vector field object is:

```
vectorfieldobject =
      domain:
     eqntype:
        fm2g:
```

Compared to the above field object two more struct fields have been added in the vector field object. In *DiffMan* 1.5 every vector field over a manifold is represented by a function $\xi : \mathbb{R} \times \mathcal{M} \to \mathfrak{g}$ as in (1). This function is called `fm2g`, (function from $\mathcal{M}$ to $\mathfrak{g}$). If the function `fm2g` is only depending on time, the ODE is said to be linear or of Lie type, and general if the function `fm2g` depends on both time and configuration. The `eqntype` provides this information.

4.3 The time stepper object

This is where all the numerics is hidden. A time stepper is the numerical algorithm used in advancing the solution of the differential equation one step along the integral curve of the flow. There are different approaches in constructing these time steppers and a list of the available time steppers in *DiffMan* 1.5 is found in Appendix C.

4.4 The flow object

Mathematically, the flow is an operator defined by the vector field. Given the ordinary differential equation

$$y' = F(y), \qquad y(0) = p \in \mathcal{M},$$

the flow operator of this differential equation is the operator $\Psi_{F,t} : \mathcal{M} \to \mathcal{M}$ satisfying

$$\frac{d}{dt}\Psi_{F,t}(p) = F(\Psi_{F,t}(p)).$$

The classical solution of a differential equation is an integral curve of the vector field generating the flow operator. This integral curve is found by evaluating the flow operator in the initial point on the manifold.

The generic representation of a flow object is:

```
flowobject =
  vectorfield:
  timestepper:
     defaults:
```

The flow object must of course know the vector field defining it. Next, it needs to know a time stepper object. The choice of time stepper specifies the numerical algorithm to be used in the solution of the differential equation. This is really all the information that the flow object needs to know. However, for convenience, constants used in variable time stepping and nonlinear equation solving is collected in the flow object in the third field `defaults`. Unless the user changes any of these constants with the `setdefaults` function, the default values set by the flow constructor will be used.

5 The structure of *DiffMan*

The philosophy used in designing *DiffMan* is to respect the underlying continuous mathematics to an as large extent as possible. Finding the continuous flow of a field is equivalent of finding the numerical solution of a differential equation. The continuous ingredients of a differential equation are: **domain**, **field**, and **flow**. The relationships among them are that the field is defined over the domain, and the flow is defined by the field.

This general structure in the continuous case is reflected in *DiffMan* in the way the directories are organized. In the *DiffMan* root directory you will find three directories each corresponding to domain, field, and flow, see Figure 1. There is also a fourth directory; auxiliary, which collects things related to the non-mathematical workings of *DiffMan*. The structure is very modular and therefore it is very easy to add new classes.

The `domain` directory contains the domain categories, very important building blocks of *DiffMan*. Creating a domain category is done by creating a subdirectory in the `domain` directory. In *DiffMan* 1.5 there are 3 domain categories implemented: homogeneous spaces, Lie algebras, and Lie groups. The classes of a domain category are put in this subdirectory along with a virtual superclass specifying the type of domain.

The `field` directory contains field classes defined over domain classes. Think of this directory as the field category. In *DiffMan* 1.5 there is only one field class implemented: `@vectorfield`. This is the only field class needed in the solution of ordinary differential equations. In order to solve some partial differential equations it is interesting to be able to define tensor fields over manifolds. When implemented the tensor field class will be placed in the `field` directory in *DiffMan*.

The `flow` directory collects classes pertinent to the continuous flow. The numerical methods are treated as time steppers, and they are all placed in the subdirectory `timesteppers` found in the `flow` directory. The flow class is a virtual superclass with no subclasses since all the numerics is placed within the time steppers. The reason for this distinction between flow and time stepper is an attempt to isolate features common to all the numerical methods (the time steppers), e.g. variable time stepping, and place these features in the flow class.

The **auxiliary** directory includes 4 subdirectories that contain the *DiffMan* documentation, command line examples, demos, and utility functions. As the name of this directory reflects, the content is not vital to the workings of *DiffMan*.

```
DiffMan
     flow
          @flow
          timestepper
               @timestepper
               @tscg
               ⋮
               @tsrkmk
     field
          @vectorfield
     domain
          hmanifold
               @hmanifold
               @hmisospec
               ⋮
               @hmtop
          liealgebra
               @liealgebra
               @ladirprod
               ⋮
               @lasp
          liegroup
               @liegroup
               @lgdirprod
               ⋮
               @lgsp
     auxiliary
          demos
          documentation
          examples
          utilities
```

Figure 1: Schematic picture of the structure in *DiffMan*.

5.1 Functor classes

In *DiffMan* the different domain types are viewed as categories. A function on a category is called a functor. Examples of internal functors are the direct product, semi-direct product, and tangent map. The direct product functor will take n domain classes as input, and create a new domain object; the direct product of the domains. The semi-direct functor works in an analogous way. The tangent functor takes a domain manifold and turns it into the tangent bundle of that manifold. The tangent bundle is also a manifold; hence, it is a domain.

In *DiffMan* we call classes that automatically generate new domains from other ones **functor**

classes. The choice of name should be clear from the above discussion. In *DiffMan* 1.5 you will find one of the above functorial constructors implemented; the direct product of domains. The functor classes in question are `@ladirprod` and `@lgdirprod`. A future release of *DiffMan* will include the semi-direct product functor and the tangent functor.

5.2 Free Lie algebras

Of all the Lie algebra classes in *DiffMan*, the class `@lafree` (free Lie algebra) plays a special role. It can be viewed as a 'symbolic computation engine', capable of simplifying expressions involving Lie brackets, by systematic use of skew-symmetry and the Jacobi identity. When a formal expression is simplified, it may subsequently be evaluated in a concrete Lie algebra. Thus, this class is very useful for developing and simplifying algorithms in Lie algebras. Furthermore, in Appendix B we show that `@lafree` is very useful for another type of optimization, related to the fact that calls to overloaded operators are handled very inefficiently in MATLAB 5. More details on numerical computations in free Lie algebras, complexity results, and applications to Lie group integrators are found in [27].

Given an arbitrary index set I, either finite or countably infinite. The following definition is equivalent to the one in [32].

Definition 5.1 *A Lie algebra $\mathfrak{g}$ is* free *over the set I if:*

i) For every $i \in I$ there corresponds an element $X_i \in \mathfrak{g}$.

ii) For any Lie algebra $\mathfrak{h}$ and any function $i \mapsto Y_i \in \mathfrak{h}$, there exists a unique Lie algebra homomorphism $\pi : \mathfrak{g} \to \mathfrak{h}$ satisfying $\pi(X_i) = Y_i$ for all $i \in I$.

In category theory, $\mathfrak{g}$ is said to be a *universal object*, i.e. it contains a structure that is common to all Lie algebras, but nothing more. Furthermore, computations in $\mathfrak{g}$ can be applied in any concrete Lie algebra $\mathfrak{h}$ via the homomorphism π.

Computationally it is useful to represent a free Lie algebra in terms of a basis. The most used bases are the (classical) Hall basis and the Lyndon basis. They can both be regarded as generalized Hall type bases [31]. The *DiffMan* `@lafree` class is implemented using a Hall basis.

Example 5.2 *If $I = \{1, 2, 3\}$, the (classical) Hall basis consisting of elements with length ≤ 4 is given as:*

$$X_1 \quad X_2 \quad X_3$$

$$[X_1, X_2] \quad [X_1, X_3] \quad [X_2, X_3]$$

$$\begin{array}{llll} [X_1,[X_1,X_2]] & [X_1,[X_1,X_3]] & [X_2,[X_1,X_2]] & [X_2,[X_1,X_3]] \\ [X_2,[X_2,X_3]] & [X_3,[X_1,X_2]] & [X_3,[X_1,X_3]] & [X_3,[X_2,X_3]] \end{array}$$

$$\begin{array}{lll} [X_1,[X_1,[X_1,X_2]]] & [X_1,[X_1,[X_1,X_3]]] & [X_2,[X_1,[X_1,X_2]]] \\ [X_2,[X_1,[X_1,X_3]]] & [X_2,[X_2,[X_1,X_2]]] & [X_2,[X_2,[X_1,X_3]]] \\ [X_2,[X_2,[X_2,X_3]]] & [X_3,[X_1,[X_1,X_2]]] & [X_3,[X_1,[X_1,X_3]]] \\ [X_3,[X_2,[X_1,X_2]]] & [X_3,[X_2,[X_1,X_3]]] & [X_3,[X_2,[X_2,X_3]]] \\ [X_3,[X_3,[X_1,X_2]]] & [X_3,[X_3,[X_1,X_3]]] & [X_3,[X_3,[X_2,X_3]]] \\ [[X_1,X_2],[X_1,X_3]] & [[X_1,X_2],[X_2,X_3]] & [[X_1,X_3],[X_2,X_3]] \end{array}$$

Free Lie algebras are infinite dimensional algebras, which for many numerical applications need to be truncated according to some measure of the size of the terms. If ϵ is some small parameter and each generator X_i has an order $X_i = \mathcal{O}(\epsilon^{w_i})$, then $[X_i, X_j] = \mathcal{O}(\epsilon^{w_i+w_j})$. Now we might want to neglect all terms of order q or higher. Intuitively we form all brackets in a Hall basis and remove all brackets of sufficiently high order. Mathematically this is a quotient construction on a *graded free Lie algebra*. Let $X_1, \ldots, X_s$ be generators for a free Lie algebra $\mathfrak{g}$ and let $w_1, \ldots, w_s$ be positive integers. We define a grading of the Hall basis of $\mathfrak{g}$ by

$$\begin{aligned} \mathrm{grade}(X_i) &= w_i \\ \mathrm{grade}([h_i, h_j]) &= \mathrm{grade}(h_i) + \mathrm{grade}(h_j) \text{ for all } h_i, h_j \in H, \end{aligned}$$

where H denotes the Hall basis. Now let $\mathfrak{g}_q = \mathrm{span}\{\, h \in H \mid \mathrm{grade}(h) \geq q \,\}$. Clearly $\mathfrak{g}_q$ is an ideal of $\mathfrak{g}$, and $\mathfrak{g}/\mathfrak{g}_q$ is the finite dimensional Lie algebra obtained by dropping all high order terms. This construction can be defined independently on a particular choice of basis for $\mathfrak{g}$. In *DiffMan* we construct the object $\mathfrak{g}/\mathfrak{g}_q$ by issuing the command

$$\mathtt{objname} = \mathtt{lafree}(\{[\mathtt{s}, \mathtt{q}], [\mathtt{w}_1, \ldots, \mathtt{w}_s]\}).$$

In [27] we establish Witt-type formulas for computing the dimension of $\mathfrak{g}/\mathfrak{g}_q$. A tutorial example on using `@lafree` is given in Appendix B, where we apply this class in the construction of RKGL type Lie group integrators for solving equations of Lie type.

A How to solve ODEs in *DiffMan*

This appendix will teach you the basics of solving differential equations in *DiffMan*. The first section shows you how to initialize *DiffMan* and get the toolbox up and running. The next section describes a 5-step procedure to be followed when solving differential equation in *DiffMan*. Finally, the last section takes you through a very detailed example showing you the 5-step procedure in practice.

A.1 How to get started

The very first thing to do is to initialize the *DiffMan* toolbox. Make sure that you are located in the *DiffMan* directory, or that this directory is included in the MATLAB path. You can easily include the following command in your `startup.m` file, or issue it at the MATLAB prompt:

```
>> addpath('/local/path/on/your/machine/DiffMan');
```

Initializing *DiffMan* is done simply by typing the command:

```
>> dminit
```

The result of this command is that all necessary paths are set and *DiffMan* is ready for use.

The *DiffMan* facility `dmtutorial` will launch a window where you can choose to run different kinds of tutorials. One of these tutorials will guide you through 'How to solve ODEs in *DiffMan*'. This is the 5-step procedure presented in the next section. The other tutorials will guide you through other important aspects of the *DiffMan* toolbox essential to the user.

`dmhelp` is a substantially improved version of the `helpwin` facility in MATLAB. `dmhelp` will launch a *DiffMan* help window where you can get help on every function and class in *DiffMan*, and also every other function in MATLAB. Hence, when working with *DiffMan* you are urged to use `dmhelp` instead of the MATLAB functions `helpwin` and `help`.

The MATLAB `demo` utility will include *DiffMan* among its toolboxes. Running the *DiffMan* demos is another convenient way of launching the *DiffMan* tutorials, and running all the *DiffMan* command line examples.

A.2 How to solve differential equations in *DiffMan* – A 5-step procedure

Once *DiffMan* is initialized you can start solving differential equations.

In *DiffMan* we are exclusively working with objects, and these objects are members of different classes. To create an object of a particular class, invoke the constructor of that class. The constructor always has the same name as the class.

The 5-step procedure for solving differential equations in *DiffMan* is the following:

1) **Construct an initial domain object `y` in a homogeneous space**. In order to solve an initial value problem, *DiffMan* needs to know an initial condition. The initial domain object serves this purpose.

2) **Construct a vector field object `vf` over the domain object `y`**. *DiffMan* finds numerically the integral curve of this vector field through the initial domain object. A vector field object consists of three parts: `domain`, `eqntype`, and `fm2g`. Set these properties of the vector field object by the functions `setdomain`, `seteqntype`, and `setfm2g`. You retrieve them with the corresponding `get` functions.

3) **Construct a time stepper object `ts`**. The time stepper class determines the numerical method used to advance the numerical solution along the integral line. A time stepper object consists of two parts: `coordinate` and `method`. Set these properties of the time stepper object by the functions `setcoordinate` and `setmethod`. You retrieve them with the corresponding `get` functions.

4) **Construct a flow object `f`**. The flow object is defined by the vector field object. Since we are doing numerical computations the flow object also needs to know how to step forward, hence the flow object `f` also needs to know the time stepper object. To set the two properties of the flow object use the functions `setvectorfield` and `settimestepper`. You retrieve them with the corresponding `get` functions.

5) **Solve the ODE**. Solving the ODE defined by the flow object in *DiffMan* is simply done by evaluating the flow object at the initial domain object, start time, end time, and step size:

```
>> output = (y,tstart,tfinal,h);
```

Variable step size is indicated by using negative values for h. The initial step then be of length $|h|$. The struct `output` consists of three fields: `output.y` is a vector of domain objects, `output.t` is a vector of time points, and `output.rej` is a vector indicating rejection of a time step in variable time stepping.

Detailed mathematical information and definitions of flows and vector fields are found in Section 2.

The 5-step procedure will be detailed out on an example in the next section.

A.3 A detailed example

Consider solving the following differential equation on the sphere S^2:

$$\frac{\mathrm{d}y}{\mathrm{d}t} = \begin{bmatrix} 0 & t & -0.4\cos(t) \\ -t & 0 & 0.1t \\ 0.4\cos(t) & -0.1t & 0 \end{bmatrix} y(t), \qquad y(0) = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \tag{2}$$

where $y \in \mathbb{R}^3$ is a vector of unit length, and the matrix on the right hand side is a map from $\mathbb{R}$ into $\mathfrak{so}(3)$.

The homogeneous manifold in question is `@hmnsphere` which consists of the sphere manifold S^2, the Lie algebra of O(3) which is $\mathfrak{so}(3)$, and the action $\lambda : (v, m) \to \exp(v) \cdot m$ of $\mathfrak{so}(3)$ on S^2. The elements of the manifold S^2 are vectors of unit length.

Step #1: Construct an initial domain object `y` in `@hmnsphere`

The initial domain object is created by calling the constructor of `@hmnsphere`. This constructor can take an integer or an `laso` object as an argument and thereby specifying the shape of the manifold object.

```
>> y = hmnsphere(3)
y =
Class: hmnsphere
Shape-object information:
   Class: laso
   Shape: 3
```

The shape of an object in `@hmnsphere` consists of an object in the Lie algebra `laso`. The integer supplied to the constructor sets the shape of this Lie algebra object that comprises the shape of the `@hmnsphere` object. If an argument to the constructor is not supplied, the shape can be set later by use of the `setshape` function.

As mentioned in the beginning of this Section, the data representation of an object in `@hmnsphere` is a vector of unit length. If the initial condition for the ODE on the sphere is the North pole, the data of the initial object must be set equal to the North pole vector.

```
>> setdata(y,[0 0 1]');
>> y
y =
Class: hmnsphere
Shape-object information:
   Class: laso
   Shape: 3
Data:
     0
     0
     1
```

The first step is now completed.

Step #2: Construct a vector field object `vf` over the domain object `y`

A vector field is defined over a domain. The constructor of `@vectorfield` is called with the domain object as input:

```
>> vf = vectorfield(y)
```

```
vf =
Class:  vectorfield
Domain: hmnsphere
Shape-object information:
   Class: laso
   Shape: 3
Eqn type:   General
```

Already, `vf` contains a lot of information. Since the domain object was supplied as an argument for the vector field constructor, the domain information is already set. The shape of the `@hmnsphere` object is an `laso` object, and the information about this Lie algebra object is displayed as `Shape-object information`. Further, the equation type of the generator map for the vector field is set to be 'General'. This is the default value. However, equation (2) is of Lie type, so the type should be changed to 'Linear' in order to speed up the calculations.

```
>> seteqntype(vf,'Linear');
```

The generator map of equation (2) is the matrix on the right hand side of the equation. The m-file `vfex5.m` contains the necessary MATLAB code to implement the generator map.

```
>> setfm2g(vf,'vfex5');
```

How does this m-file `vfex5.m` look like? To view the file you can type `type vfex5.m` at the MATLAB prompt, or use `dmhelp` and push the button `View src`. Any way the output is:

```
function [la] = vfex5(t,y)
% VFEX5 - Generator map from RxM to liealgebra. Linear type.

la = liealgebra(y);
dat = [0 t -0.4*cos(t); -t 0 .1*t; .4*cos(t) -.1*t 0];
setdata(la,dat);
return;
```

All the generator map function files that you write on your own must have this generic structure: The file must support two arguments; the first is a scalar – time, and the second is a domain object from the homogeneous space. Output must be a Lie algebra object. To find the correct Lie algebra of the domain object call `liealgebra(y)`, which will return an object in the correct Lie algebra with preset shape information. Edit the data `dat`, and call `setdata(la,dat)` in order to set the data representation of the Lie algebra object.

Now the vector field object `vf` displays as:

```
>> vf
vf =
Class:  vectorfield
Domain: hmnsphere
Shape-object information:
   Class: laso
   Shape: 3
Map fm2g:   vfex5
Eqn type:   Linear
```

Step #3: Construct a time stepper object ts

The time stepper class decides which numerical method to use for advancing along the integral curve of the vector field. Calling any of the time stepper constructors will return a time stepper object with *default* coordinate and method. If the user prefers other coordinates or another method, these can be changed through the functions `setcoordinate` and `setmethod`. To get an overview of the different time steppers type `dmhelp timestepper` in MATLAB. In our example we want to use an RKMK method:

```
>> ts = tsrkmk
ts =
Class:  tsrkmk
Coord.: exp
Method: RK4
```

In case of `@tsrkmk` the default coordinate is `exp` and the default method is `RK4`. For a discussion of the possible choices of coordinates, see [7]. For each time stepper class there is a whole lot of methods to choose from. None of these methods can be used for all the different time stepper classes and *DiffMan* will issue an error message if a wrong selection is made.

In our example we are not satisfied with only the standard 4th-order `RK4` method, we want the more accurate answer supplied by the 6th-order Butcher method:

```
>> setmethod(ts,'butcher6')
>> ts
ts =
Class:  tsrkmk
Coord.: exp
Method: butcher6
```

To get information about the different methods while running *DiffMan*, type `dmhelp setmethod`.

Step #4: Construct a flow object f

The flow object is constructed from the vector field object `vf` and the time stepper object `ts` already created. Just calling the `@flow` constructor will create an object with a *default* time stepper. The default time stepper preset in the flow object `f` is only a matter of convenience, and must not be confused with the time stepper object created in Step #3.

```
>> f = flow
    f =
    Class: flow
    Timestepper class: tsrkmk
        Coordinates:   exp
        Method:        RK4
```

In our example we have created another time stepper object `ts` that we want to use instead of the default time stepper object supplied by the `@flow` constructor. To change the time stepper of the flow object `f` to `ts`, call the function `settimestepper`:

```
>> settimestepper(f,ts)
```

A flow is defined as the flow of some vector field. Hence, our flow object `f` must have information about this vector field.

```
>> setvectorfield(f,vf)
>> f
f =
Class: flow
Vector field information:
    Domain:          hmnsphere
    Equation type:   Linear
    Map defining DE: vfex5
Timestepper class: tsrkmk
    Coordinates:   exp
    Method:        butcher6
```

Now the flow object has the necessary information and we can go on to the next, and final, step in the 5-step solution procedure.

Step #5: Solve the ODE

Solving equation (2) with the RKMK method is done by evaluating the flow object with four arguments: initial domain object, start time, end time, and step size.

```
>> curve = f(y,0,5,0.05)
curve =
      y: [1x61 hmnsphere]
      t: [1x61 double   ]
    rej: [1x61 double   ]
```

The output `curve` is a MATLAB struct with the three fields: `y`, `t`, and `rej`. `curve.y` is a vector of objects from the homogeneous space upon which the problem is modeled. `curve.t` is a vector of scalars, the time points. `curve.rej` is a vector of integers indicating if a step was rejected or not. In our example `curve.rej` is the zero vector, since we did not use variable time stepping.

Calling `getdata(curve.y)` will access the actual data representations of all the `@hmnsphere` objects. In this case this output will be a vector three times the length of the scalar vector `curve.t`. To get the 3-vectors corresponding to each time point, the output from `getdata(curve.y)` must be reshaped into a $3\times$`length(t)` matrix where each column corresponds to a time point. To plot the data we can do the following:

```
>> t = curve.t;
>> a = getdata(curve.y);
>> a = reshape(a,3,length(t));
>> comet3(a(1,:),a(2,:),a(3,:));
```

In Figure 2 the solution of the problem is plotted.

This detailed example is found as Example 5 in the *DiffMan* toolbox and runs by typing:

```
>> dmex5
```

What's next?: Solve the same problem with a different time stepper

To use another time stepper to solve equation (2) we must repeat steps #3 through #5. We must create a new time stepper object, put this into the existing flow object, and evaluate the flow again. To use the Crouch-Grossman method we do the following:

```
>> ts2 = tscg
```

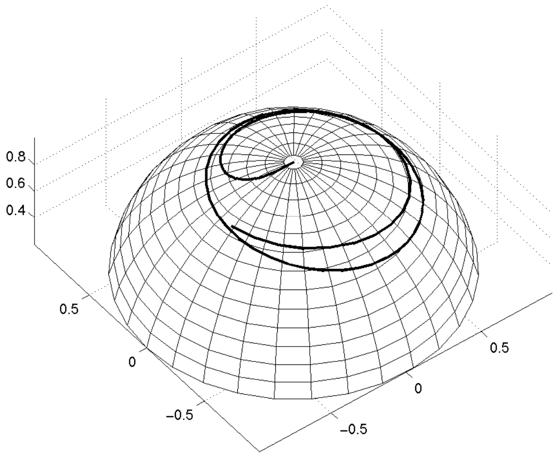

Figure 2: Plot of the solution of (2).

```
ts2 =
Class:  tscg
Coord.: exp
Method: CG3a
>> settimestepper(f,ts2)
>> f
f =
Class: flow
Vector field information:
    Domain:           hmnsphere
    Equation type:    Linear
    Map defining DE: vfex5
Timestepper class: tscg
    Coordinates:    exp
    Method:         CG3a
>> curve = f(y,0,5,0.05);
```

To plot the solution we just repeat the above plotting commands. The solution is the same, except from the fact that we have used a third-order method. This solution is not as accurate as the solution obtained by the 6th-order Butcher method used in the RKMK time stepper.

B An example using the free Lie algebra

The commands given below reflect the basic operations of Definition 5.1.

`>> fla = lafree({[p,q],[w1,w2, ... ,wp]});`
Generate a free Lie algebra from `p` symbols with grades `w1,w2, ... ,wp`. All terms of total grade greater than `q` are set to 0.

`>> Xi = basis(fla,i);`
Return the `i`'th Hall basis element in `fla`. If $1 \leq$ `i` $\leq$ `p`, return the `i`'th generator X_i.

`>> X+Y; r*X; [X,Y];`
Basic computations in the free Lie algebra.

`>> Z = eval(E,Y1,Y2, ...,Yp);`

If `E` is an element of a free Lie algebra, and `Y1,Y2, ...,Yp` are the elements of *any DiffMan* Lie algebra, this will evaluate the expression `E`, using the data set `Y1,Y2, ...,Yp` in place of the generating set. This corresponds to the homomorphism $\pi : \mathfrak{g} \to \mathfrak{h}$ in Definition 5.1. One may also write `Z = eval(E,Y);` where `Y(i) = Yi;`.

As an example of the use of this class we will consider the construction of Runge–Kutta–Gauss–Legendre (RKGL) type integrators for equations of Lie type (or linear type). To make things simple, consider a matrix equation

$$y'(t) = f(t)y(t), \qquad y(t_0) = y_0, \tag{3}$$

where $y(t)$ is a curve on a matrix Lie group and $f(t)$ is a curve in the Lie algebra. It is well known [23] that the solution for sufficiently small t can be written as

$$y(t) = \exp(\sigma(t))y_0,$$

where $\sigma(t)$ satisfy

$$\sigma'(t) = \mathrm{dexp}^{-1}_{\sigma(t)}(f(t)), \qquad \sigma(t_0) = 0. \tag{4}$$

RKGL methods are s-stage implicit methods of order $2s$. For equations of Lie type, where $f = f(t)$, the problem of solving implicit equations can be solved once and for all by a computation in a free Lie algebra. We seek an expression for $\sigma(t_0 + h)$ in terms of the s quantities $k_i = hf(t_0 + c_i)$. To develop a general integration scheme, we may regard k_i as being generators of a free Lie algebra, and find an expression for $\sigma(t_0 + h)$ by solving the implicit RK equations in the free Lie algebra. If this is done in a straightforward manner, the number of commutators involved in the expression for $\sigma(t_0 + h)$ turns out to be very large. In [27], we show that this can be overcome by changing the basis for the free Lie algebra. We introduce a new set of generators $X_1, \ldots, X_s$ as

$$k_i = \sum_j V_{i,j} X_j,$$

where $V_{i,j} = (c_i - 1/2)^{j-1}$ is a Vandermonde matrix. After this change of variables, it can be shown that $X_i = \mathcal{O}(h^i)$, thus $X_1, \ldots, X_s$ generate a graded free Lie algebra with the grading $w_i = i$.

In Figure 3 we show the computation of σ. A 6'th order method is obtained by

$$[\mathtt{sig}, \mathtt{c}, \mathtt{W}, \mathtt{s}] = \mathtt{rkgllmethod}(6),$$

which yields (numerically) the result:

$$\begin{aligned}
\mathtt{sig} &= X_1 + \tfrac{1}{12}X_3 - \tfrac{1}{12}[X_1, X_2] + \tfrac{1}{240}[X_2, X_3] + \tfrac{1}{360}[X_1, [X_1, X_3]] - \ldots \\
&\quad -\tfrac{1}{240}[X_2, [X_1, X_2]] + \tfrac{1}{720}[X_1, [X_1, [X_1, X_2]]] \\
\mathtt{c} &= \begin{pmatrix} \frac{5-\sqrt{15}}{10} & \frac{1}{2} & \frac{5+\sqrt{15}}{10} \end{pmatrix}^T \\
\mathtt{W} &= \begin{pmatrix} 0 & 1 & 0 \\ -\frac{\sqrt{5}}{15} & 0 & \frac{\sqrt{5}}{15} \\ \frac{10}{3} & -\frac{20}{3} & \frac{10}{3} \end{pmatrix} \\
\mathtt{s} &= 3
\end{aligned}$$

Figure 4 shows how these data are used in a simple timestepper. A major part of the computation is the evaluation of the expression `sig` on the data `X`, `s = eval(sig, X);`. Once the expression for `sig` is known, this line could have been written out explicitly, e.g. as

$$\mathtt{s} = \mathtt{X(1)} + \mathtt{X(3)} * \mathtt{(1/12)} - [\mathtt{X(1)}, \mathtt{X(2)}] * \mathtt{(1/12)} + \cdots.$$

```
% RKGLLMETHOD - Construct q'th order RKGL-Lie integrator
function [sig,c,W,s] = rkgllmethod(q)
   % coeffs.  for classical s-stage q'th order RKGL
   [A,b,c,s] = rkglcoeff(q);
   % change basis with Vandermonde matrix
   V = vander(c-1/2); V = V(:,s:-1:1);
   W = inv(V);
   laf = lafree({[s,q],[1:s]});
   for i = 1:s,
      k(i) = zero(laf);
      for j = 1:s, k(i) = k(i)+V(i,j)*basis(laf,j); end;
   end;
   % solve implicit RK step in laf by fixpoint iteration
   u = zeros(laf,s);
   for ord = 1:q,
      for i = 1:s, ktild(i) = dexpinv(u(i),k(i),q); end;
      for i = 1:s, u(i) = A(i,:)*ktild; end;
   end;
   sig = b*ktild;
return;
```

Figure 3: Construction of q'th order RKGL-Lie integrator.

However, the `eval` function allows optimizations that are lost in an explicit expansion. When `eval` is called, the expression is analyzed and the computation is optimized. So, if a given commutator appears several places in the same expression, like $\frac{1}{12}[X_1, X_2] + \cdots - \frac{1}{240}[X_2, [X_1, X_2]]$, the `eval` function will only compute $[X_1, X_2]$ once. An even more significant optimization is related to the overhead of calling overloaded operators. If a bracket $[X_1, X_2]$ is called in MATLAB, then the system first has to figure out in which Lie algebra X_i belongs, and then call the corresponding commutator. In the current version of MATLAB, there is a large overhead associated with this handling of overloaded operators. The `eval` function, on the other hand, is implemented such that the job of figuring out which algebra is involved is only done once. We have seen that for small matrices this optimization trick yields codes running up to ten times faster!

C The numerical time steppers in *DiffMan*

The development of *DiffMan* is an ongoing project, and the numerical time steppers implemented are subject to change in the future. Version 1.5 of *DiffMan* includes the following time steppers:

tscg: the Crouch-Grossman methods [4, 30].

tsfer: the Fer expansion methods [10, 13, 34].

tsmagnus: the Magnus series methods [17, 14, 34].

tsqq: quadrature methods for quadratic Lie groups [19].

tsrk: classical Runge-Kutta methods. These methods are well-known to the ordinary differential equation community and the classical references include [3, 11].

```
Given:  f, y0, t0, h, q.

   [sig,c,W,s] = rkgllmethod(q)
   t = t0; y = y0;
   for istp = 1:nstp,
      for i = 1:s, k(i) = h*f(t+c(i)*h); end;
      for i = 1:s, X(i) = W(i,:)*k; end;
      s = eval(sig,X);
      y = exp(s)*y;
      t = t+h;
   end;
```

Figure 4: Simple RKGL-Lie timestepper.

tsrkmk: Runge-Kutta methods of Munthe-Kaas type. A large number of papers discuss these methods, but the trilogy where the methods were derived is [21, 22, 23].

In future releases, general linear methods as well as linear multistep methods will be included in *DiffMan*, both as classical methods [11] and in a generalized setting [9].

References

[1] R. Abraham and J. E. Marsden. *Foundations of Mechanics.* Addison-Wesley, second edition, 1978.

[2] R. Abraham, J. E. Marsden, and T. S. Ratiu. *Manifolds, Tensor Analysis, and Applications.* AMS 75. Springer-Verlag, second edition, 1988.

[3] J. C. Butcher. *The Numerical Analysis of Ordinary Differential Equations.* Wiley, 1987.

[4] P. E. Crouch and R. Grossman. Numerical integration of ordinary differential equations on manifolds. *J. Nonlinear Sci.*, 3:1–33, 1993.

[5] K. Engø. On the construction of geometric integrators in the RKMK class. Technical Report No. 158, Department of Informatics, University of Bergen, Norway, 1998.

[6] K. Engø and S. Faltinsen. Integrating Lie–Poisson systems with the RKMK method. In preparation. 1999.

[7] K. Engø, A. Marthinsen, and H. Munthe-Kaas. *DiffMan — an object oriented MATLAB toolbox for solving differential equations on manifolds: User's Guide. Version 1.5.* Available on WWW at `http://www.math.ntnu.no/num/diffman/`.

[8] K. Engø, H. Z. Munthe-Kaas, and A. Zanna. On the time symmetry of Lie group integrators. In preparation. 1999.

[9] S. Faltinsen, A. Marthinsen, and H. Munthe-Kaas. Multistep methods integrating ordinary differential equations on manifolds. Manuscript, 1999.

[10] F. Fer. Résolution del l'equation matricielle $\dot{U} = pU$ par produit infini d'exponentielles matricielles. *Bull. Classe des Sci. Acad. Royal Belg.*, 44:818–829, 1958.

[11] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I, Nonstiff Problems*. Springer-Verlag, second revised edition, 1993.

[12] M. Haveraaen, V. Madsen, and H. Munthe-Kaas. Algebraic programming technology for partial differential equations. In *Proceedings of Norsk Informatikk Konferanse (NIK)*, Trondheim, Norway, 1992. Tapir.

[13] A. Iserles. Solving linear ordinary differential equations by exponentials of iterated commutators. *Numer. Math.*, 45:183–199, 1984.

[14] A. Iserles and S. P. Nørsett. On the solution of linear differential equations in Lie groups. Technical Report 1997/NA3, Department of Applied Mathematics and Theoretical Physics, University of Cambridge, England, 1997. To appear in Philosophical Transactions of the Royal Society.

[15] D. Lewis and J. C. Simo. Conserving algorithms for the dynamics of Hamiltonian systems of Lie groups. *J. Nonlinear Sci.*, 4:253–299, 1994.

[16] S. B. Lippman. *C++ Primer*. Addison Wesley, second edition, 1991.

[17] W. Magnus. On the exponential solution of differential equations for a linear operator. *Comm. Pure and Appl. Math.*, VII:649–673, 1954.

[18] J. E. Marsden and T. S. Ratiu. *Introduction to Mechanics and Symmetry*. Springer-Verlag, 1994.

[19] A. Marthinsen and B. Owren. Quadrature methods based on the Cayley transform. Technical Report Numerics No. 1/1999, The Norwegian University of Science and Technology, Trondheim, Norway, 1999.

[20] The MathWorks, Inc., 24 Prime Park Way, Natick, MA 01760-1500. *Using MATLAB*, 5.2 edition, April 1998.

[21] H. Munthe-Kaas. Lie–Butcher theory for Runge–Kutta methods. *BIT*, 35(4):572–587, 1995.

[22] H. Munthe-Kaas. Runge–Kutta methods on Lie groups. *BIT*, 38(1):92–111, 1998.

[23] H. Munthe-Kaas. High order Runge–Kutta methods on manifolds. *Applied Numerical Mathematics*, 29:115–127, 1999.

[24] H. Munthe-Kaas and M. Haveraaen. Coordinate free numerics; Part 1: How to avoid index-wrestling in tensor computations. Technical Report No. 101, Department of Informatics, University of Bergen, Norway, 1995.

[25] H. Munthe-Kaas and M. Haveraaen. Coordinate free numerics — Closing the gap between 'Pure' and 'Applied' mathematics? In *Proceedings of ICIAM-95, Zeitschrift für Angewandte Mathematik und Mechanik (ZAMM)*, Berlin, 1996. Akademie Verlag.

[26] H. Munthe-Kaas and E. Lodden. Explicit Lie group integrators for heat equations. In preparation. 1999.

[27] H. Munthe-Kaas and B. Owren. Computations in a free Lie algebra. Technical Report No. 148, Department of Informatics, University of Bergen, Norway, 1998. To appear in Philosophical Transactions of the Royal Society.

[28] H. Munthe-Kaas and A. Zanna. Numerical integration of differential equations on homogeneous manifolds. In F. Cucker and M. Shub, editors, *Foundations of Computational Mathematics*, pages 305–315. Springer Verlag, 1997.

[29] B. Owren and A. Marthinsen. Lie group integrators based on canonical coordinates of the second kind. In preparation. 1999.

[30] B. Owren and A. Marthinsen. Runge–Kutta methods adapted to manifolds and based on rigid frames. *BIT*, 39(1):116–142, 1999.

[31] C. Reutenauer. *Free Lie Algebras.* Number 7 in London Mathematical Society Monographs, New Series. London Mathematical Society, 1993.

[32] V. S. Varadarajan. *Lie Groups, Lie Algebras, and Their Representations.* GTM 102. Springer-Verlag, 1984.

[33] F. W. Warner. *Foundations of Differentiable Manifolds and Lie Groups.* GTM 94. Springer-Verlag, 1983.

[34] A. Zanna. Collocation and relaxed collocation for the Fer and the Magnus expansions. Technical Report 1997/NA17, Department of Applied Mathematics and Theoretical Physics, University of Cambridge, England, 1997.

[35] A. Zanna. *On the Numerical Solution of Isospectral Flows.* PhD thesis, Cambridge University, 1998.