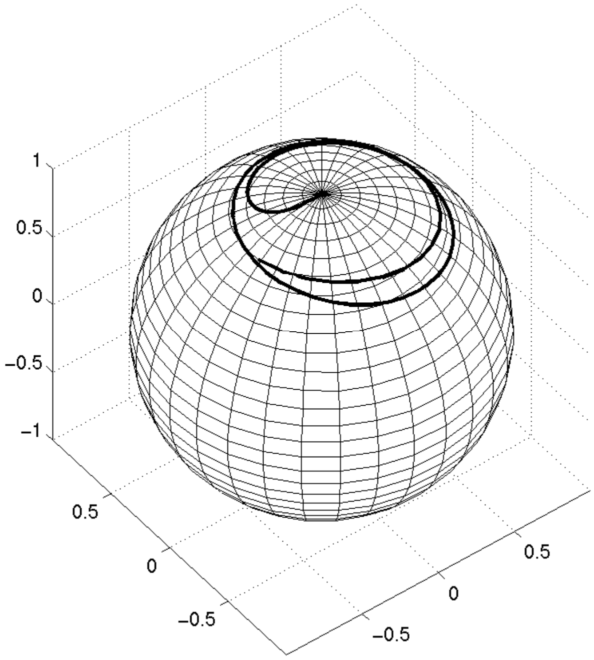
K. Engø
A. Marthinsen
H. Z. Munthe-Kaas

DiffMan User's Guide Version 1.6



March 10, 1999

ISSN 0333-3590 REPORT NO 166 March 1999

Department of Informatics, University of Bergen, Norway

Preface

DiffMan is a MATLAB toolbox for solving Ordinary Differential Equations on manifolds, based on the concept of 'Coordinate Free Numerics'. This is, loosely speaking, the idea that (whenever possible) it is important to formulate numerical algorithms generically, independent of special representations and coordinate systems. *DiffMan* inherits its basic design philosophy from the C++ package SOPHUS. The SOPHUS project was initiated by Magne Haveraaen and Hans Munthe-Kaas, University of Bergen, and aims at solving tensor field equations on sequential and parallel computers.

Some of the mathematical background of *DiffMan* is described in Appendix A. This chapter, however, is included for the sake of completeness. The user can read the rest of the manual and use *DiffMan* without having studied this chapter.

The writing of this toolbox was initiated as a project within the SYNODE project, and a number of papers describing the numerical methods in *DiffMan* is available on the SYNODE home page at URL `http://www.math.ntnu.no/num/synode/`.

We will thank Antonella Zanna for reading an earlier version of this User's Guide and always giving helpful suggestions, and Martine T. Monsen for extensive proof reading.

Kenth Engø Arne Marthinsen Hans Z. Munthe-Kaas

Contents

Chapter 1

Introduction

DiffMan is an object oriented MATLAB toolbox designed to solve differential equations evolving on manifolds. *DiffMan* 1.6 addresses primarily the issue of solving 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 continous problem. Hence, the *DiffMan* toolbox collects some of the most recent and sophisticated methods for solving ODEs in the sense of geometric integration. *DiffMan* is an ongoing project and an overall goal for future work is to also include geometric integration techniques for partial differential equations evolving on infinite dimensional manifolds.

To understand the workings of *DiffMan*, a geometric understanding of ordinary differential equations evolving on a manifold is of great importance. Consider the ordinary differential equation

$$y' = F(t, y), \qquad y(0) = y_0 \tag{1.1}$$

evolving in Euclidean space. That is; $y \in \mathbb{R}^n$ and $F(t, y)$ is a vector field assigning to each point in $\mathbb{R}^n$ and time $t \in \mathbb{R}$, a vector F. The geometric interpretation of finding a solution to equation (1.1) is to find a curve $y(t)$ starting at the point y_0 in Euclidean space, having as its tangent at every succeeding point and time the vector specified by the vector field F. Hence, we want to follow a curve in space that starts at the inital point y_0, evolving in the direction prescribed by the vector field.

This same description can be used for an ordinary differential equation evolving on a manifold. The only difference is that the manifold is in general a non-linear space, and not a linear vector space. The simplest example of a manifold is the sphere. A vector field over the manifold will at each point attach a tangent vector, just like in the Euclidean case. A solution of this ODE is a curve through the initial point y_0, that evolves on the manifold in the direction given by the tangent vector.

There is a well established theory for solving (1.1) on a Euclidean space, a fact to be extensively exploited in one of the methods discussed in Appendix A. A well-known family of methods are the Runge–Kutta methods [12]. There are also splitting methods and composition methods, see for instance [23, 24], but in *DiffMan* we will among the classical methods mainly be concerned with the Runge-Kutta family.

Availability of *DiffMan*

DiffMan 1.6 is available according to the Licence Agreement. Problem reports should be submitted by email to:

`diffman@math.ntnu.no`

Up to date information and documentation on *DiffMan* is found on WWW at URL:

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

The Authors

The authors of *DiffMan* are

Kenth Engø
Department of Informatics
University of Bergen
N-5020 Bergen, Norway
Email: `Kenth.Engo@ii.uib.no`
WWW: `http://www.ii.uib.no/~kenth/`

Arne Marthinsen
Department of Mathematical Sciences
Norwegian University of Science and Technology
N-7491 Trondheim, Norway
Email: `Arne.Marthinsen@math.ntnu.no`
WWW: `http://www.math.ntnu.no/~arnema/`

Hans Z. Munthe-Kaas
Department of Informatics
University of Bergen
N-5020 Bergen, Norway
Email: `Hans.Munthe-Kaas@ii.uib.no`
WWW: `http://www.ii.uib.no/~hans/`

Acknowledgement

This work was in part sponsored by the Norwegian Research Council under contract no. 111038/410 through the SYNODE project. The SYNODE project is found on WWW at URL:

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

Licence Agreement

Chapter 2

The *DiffMan* environment

In this chapter we will go through basic concepts in object orientation and explain how these things are done in MATLAB. We will outline the general structure of *DiffMan* and discuss the four different types of objects used in *DiffMan*. *The last section, probably the most important section in the whole user's guide, discusses how to go about getting the necessary help in DiffMan.*

2.1 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 is 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 object must satisfy in order to be a member of the class. Another 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 to 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 [13, 28]. Coordinate free constructions in mathematics, e.g. tensors, tries 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 gives rise to the rather contradictory term 'coordinate free numerics' [28, 29]. What is a coordinate free algorithm? The whole idea is to device algorithms specified by algebraic operations not dependent 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.

2.2 Object orientation in MATLAB and *DiffMan*

In MATLAB a class is defined by creating a directory `@myclass`, where `myclass` is the name of the class. The prefix `@` simply tells 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. Attempting to do 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 specifically 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 which 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; i.e. it is a category in the strict mathematical sense. 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* 1.6 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 has similar characteristics, but there are differences that partitions 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 of all the other classes in the domain category, and all the child classes inherit the parents' functions. That means that one can apply the public functions of the virtual superclass on an object of a child class. If the child class needs specific 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.

2.3 Structure of *DiffMan*

The general structure of *DiffMan* is reflected in the way the directories are organized. The structure is very modular and it is therefore very easy to add new classes of your own. In the *DiffMan* root directory you will find 3 m-files and 4 directories. The directories are:

```
auxiliary/
domain/
field/
flow/
```

The `auxiliary` directory includes 4 subdirectories which contain the *DiffMan* documentation, command line examples, demos, and utility functions. As the name of this directory reflects its contents are not vital to the workings of *DiffMan*.

The `domain` directory, however, contains the domain categories, which are very important building blocks of *DiffMan*. Think of a domain as a differentiable manifold. Creating a domain category is done by creating a subdirectory in the `domain` directory. In *DiffMan* 1.6 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 domain category.

The `field` directory contains field classes defined over domain classes. Think of this directory as the field category. Since *DiffMan* 1.6 only solves ordinary differential equations, the only field class implemented so far is `@vectorfield`. The field category is subject to change in future releases of *DiffMan*, but `@vectorfield` will stay the same. In order to solve some PDEs it is interesting to be able to define tensor fields over manifolds, hence a tensor field class is very likely to be added in a future release of *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 are 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.

2.3.1 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* 1.6 you will find one of the above functorial constructors implemented; the direct product of domains. The functor classes in question are `@ladirprod` and `@lgdirprod`. The next release of *DiffMan* will include the semi-direct product functor and the tangent functor.

2.4 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.

2.4.1 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 only 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 pre-set shape. This is because we need to know a 'size' measure on the domain objects that are acting on themselves.

The user cannot access the contents of the `shape` and `data` fields of a domain object directly, 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.

2.4.2 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 which assigns a vector to every element of the manifold. 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* 1.6 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* 1.6 every vector field over a manifold is represented by a function $\xi : \mathbb{R} \times \mathcal{M} \to \mathfrak{g}$. This function is called `fm2g`, (function from $\mathcal{M}$ to $\mathfrak{g}$). If the function `fm2g` only depends on time, the ODE is said to be linear or of Lie type, and it is said to be general if the function `fm2g` depends on both time and configuration. The `eqntype` provides this information.

2.4.3 The time stepper object

This is where all the numerics are 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* 1.6 is found in Appendix B.

Common for all the time steppers is that they work locally on the domain manifold, and because of this they need to know local coordinates on the Lie group manifold. Another common feature is that the time steppers naturally divide up into classes of methods. The steppers belonging to one particular class are typically distinguished by the order of approximation the method yields. Hence, it is important to also be able to choose a method of the preferred order.

The generic representation of a time stepper object takes the form:

```
timestepperobject =
  coordinate:
      method:
```

2.4.4 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

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

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 are 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.

2.5 How to get help in *DiffMan*

In MATLAB there are three ways to get online help. The first two provide simple help information on MATLAB functions, while the third is a huge collection of documentation stored in hypertext format which you access with a Web browser.

```
HELP     - Function help, display help text at command line
HELPWIN  - Function help, separate window for navigation
HELPDESK - Comprehensive hypertext documentation and troubleshooting
```

In *DiffMan* you will also have the possibility of using a window based help system called `dmhelp`.

```
DMHELP   - An extended and improved version of HELPWIN
```

This help system is based on the `helpwin` system in MATLAB, but several new features have been added: Access information about all the different classes and class functions, view the source code of m-files, view this User's guide online in pdf, go to the *DiffMan* home page on WWW, etc. Please note that `dmhelp` can only be invoked once you have started the *DiffMan* toolbox, see Section 3.1.

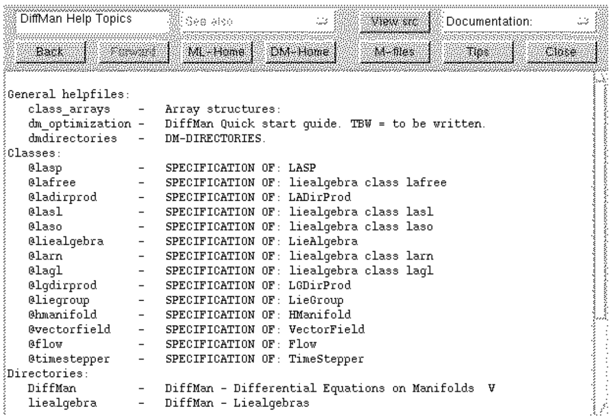


Figure 2.1: The *DiffMan* Help Window.

Once you have started up *DiffMan* you issue the command

```
>> dmhelp
```

in order to just launch the *DiffMan* Help Window, or type the command

```
>> dmhelp topic
```

to launch the *DiffMan* Help Window with the help information about `topic`.

Once the *DiffMan* Help Window system has been launched, see Figure 2.1, you can type any help topic in the upper left box of the Help Window and press 'Enter'. If the help topic has help information this will be displayed. If a line in the main text of the Help Window starts with a help topic followed by ' - ', you can also click on it directly in order to access the help information.

Consider the following examples. If you see lines in the main text Help Window looking like these, you just click on them directly in order to go to that help topic:

```
plot            - Help on any matlab functions.
liealgebra      - Help on the _directory_ 'liealgebra'.
@liealgebra     - Info on the _class_ 'liealgebra'.
@lagl           - Info on the Lie algebra gl(n).
@lafree/order   - Function 'order.m' in class 'lafree'.
```

The other option is to type the name of the help topic in the box in the upper left corner, and access the same information this way.

The first time you launch the *DiffMan* Help Window it might take some time before it appears. If you in a MATLAB session want to close the Help Window after you have finished using it, we urge you to use the 'Close' button in the window. The reason for this is that in case you want to launch the Help Window again in the same session, MATLAB will use the same window. Hitting the 'Close' button causes MATLAB to turn the visibility of the window off, and MATLAB does not have to recreate the window again when you re-enter the command `dmhelp`.

If none of the above commands seems to give you the desired explanations, you can also try the MATLAB commands: `demo`, `lookfor`, `which`, and `general`.

Chapter 3

How to use *DiffMan*

This chapter 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*, followed by a section taking you through a very detailed example demonstrating the 5-step procedure in practice. We end this chapter by discussing ODEs represented as infinitesimal generators.

3.1 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 the following is displayed in MATLAB:

```

  DiffMan Version 1.6 is initialized - 1999.03.10

  Please report any problems and/or bugs to:
       diffman@math.ntnu.no

  For more information and how to get started, try:
      >> dmtutorial
      >> dmhelp
      >> demo

>>
```

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. For further details, see Section 7.1.

`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* we urge you to use `dmhelp` instead of the MATLAB functions `helpwin` and `help`. For more information about the workings of `dmhelp`, see Section 2.5.

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.

3.2 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`. See Section 5.1 for more information.

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`. See Chapter 6 for more information.

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`.

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 = f(y,tstart,tfinal,h);
```

Variable step size is indicated by using negative values for h. The initial step is then of length $|h|$. `output` is a MATLAB structure that 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 Appendix A.

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

3.3 A detailed example

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

$$\frac{\mathrm{d}y}{\mathrm{d}t} = \begin{bmatrix} 0 & t & -0.4\cos(t) \\ -t & 0 & 0.1t \\ 0.4\cos(t) & -0.1t & 0 \end{bmatrix} y(t), \qquad y(0) = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \tag{3.1}$$

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 $\Phi : (v, m) \to \exp(v) \cdot m$ 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 a `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 which comprises the shape of the `@hmnsphere` object. If an argument to the constructor is not supplied, the shape can be set later by using 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 a `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 (3.1) is of linear type, so the type should be changed to 'Linear' in order to speed up the calculations.

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

The generator map of equation (3.1) is the matrix on the right hand side of the equation. The m-file `vfex5.m` contains the MATLAB necessary code to implement the generator map.

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

What 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`. Either 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 possible choices of coordinates, see Section 6.1. For each time stepper class there are many schemes to choose from. None of these schemes 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. Merely 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 (3.1) 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 whether a step was rejected or not. In our example `curve.rej` is the zero vector, since we did not use variable time stepping. See Section 6.3 on how to do 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 3.1 the solution of the problem is plotted on the Northern hemisphere of the unit ball.

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

```
>> dmex5
```

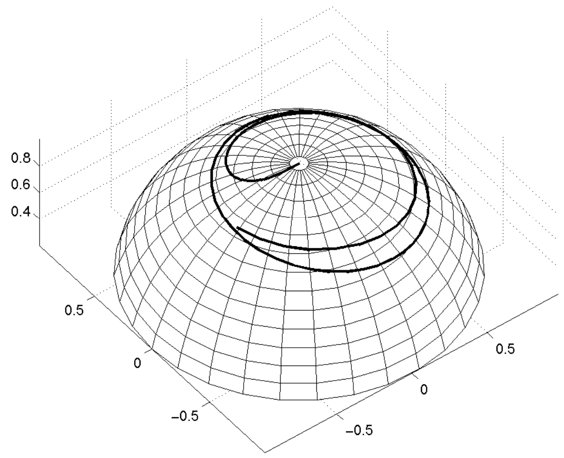


Figure 3.1: Plot of the solution of (3.1)

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

To use another time stepper to solve equation (3.1) 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
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.

3.4 ODEs as infinitesimal generators

All the ODEs that we can solve in *DiffMan* are assumed to be written in the form of an infinitesimal generator. If you are given just any ODE, it might turn out to be an impossible feat to determine the Lie-group action of which the given ODE can be written as the infinitesimal generator. The choices can be many, and there is no known general procedure for doing this. Consult Appendix A for necessary definitions and notation.

But on the other hand, there are many well-known examples of ODEs evolving on homogeneous spaces that are known to be able to be stated in the form of an infinitesimal generator. In the rest of this section we will give you examples of such ODEs, in order to aid you in your hunt for the group-action and the best setting for your particular ODE.

Recall that the ingredients of a homogeneous space is a Lie group G, a domain manifold $\mathcal{M}$, and a Lie-group action $\Phi : G \times \mathcal{M} \to \mathcal{M}$. The infinitesimal generator of the Lie-group action Φ with respect to the element $\xi \in \mathfrak{g}$ is defined as

$$\xi_{\mathcal{M}}(x) = \left.\frac{\mathrm{d}}{\mathrm{d}t}\right|_{t=0} \Phi_{\exp(t\xi)}(x), \quad \forall x \in \mathcal{M}. \tag{3.2}$$

Example 3.4.1 (Classical ODEs in $\mathbb{R}^n$)
Any classical ODE can be cast in the representation of an infinitesimal generator by making the choices $\mathcal{M} = \mathbb{R}^n, G = \mathbb{R}^n$, and $\Phi(g, x) = g + x$. Calculating the infinitesimal generator of this action (remember that in this case the exponential map is equal to the identity map) with respect to the element $\xi \in \mathfrak{g} = \mathbb{R}^n$ gives us

$$\xi_{\mathbb{R}^n}(x) = \xi. \tag{3.3}$$

Hence, any ODE

$$y' = f(t, y) \tag{3.4}$$

on $\mathbb{R}^n$ has the form of an infinitesimal generator by just choosing the generator map $\xi : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ equal to the function f in (3.4).

Example 3.4.2 (Differential equations on matrix Lie groups)
In this example we choose the manifold equal to the Lie group; that is $\mathcal{M} = G$, and let the Lie group act on itself by left multiplication, $\Phi(g, h) = gh$. In this case the exponential takes the form of the matrix exponential and the infinitesimal generator is

$$\xi_G(g) = \xi \cdot g. \tag{3.5}$$

Choosing ξ as the generator map $f : \mathbb{R} \times G \to \mathfrak{g}$ all ODEs on matrix Lie groups take on the general form

$$y' = f(t, y)y.$$

Example 3.4.3 (Isospectral flows)
For isospectral flows the domain manifold $\mathcal{M}$ is chosen as a subset of the set of $n \times n$ matrices. The Lie group is the special orthogonal group $SO(n)$ that acts upon $\mathcal{M}$ by the action $\Phi(g, m) = gmg^{-1}$. The exponential map is in this case also equal to the matrix exponential, and the infinitesimal generator of this action with respect to the element $\xi \in \mathfrak{so}(n)$ is

$$\xi_{\mathcal{M}}(m) = [\xi, m]. \tag{3.6}$$

$[\cdot, \cdot]$ is the matrix commutator. Now, choosing ξ as the generator map $f : \mathbb{R} \times \mathcal{M} \to \mathfrak{so}(n)$ all ODEs for isospectral flows take on the general form

$$y' = f(t, y)y - yf(t, y).$$

Example 3.4.4 (Lie-Poisson systems)
This example might seem very hard and abstract on a first reading, but it is a very important and interesting problem and deserves some attention. For notation and a detailed introduction see [18, 6].

A Lie-Poisson system is nothing else than a Hamiltonian problem. In this example the domain manifold is the dual space of a Lie algebra $\mathfrak{g}$*, that is* $\mathcal{M} = \mathfrak{g}^*$*. The Hamiltonian function* $H : \mathfrak{g}^* \to \mathbb{R}$ *is a conserved quantity for the flow. The Lie group* G *acts on the dual Lie algebra* $\mathfrak{g}^*$ *by the coadjoint action:* $\Phi(g,\mu) = Ad^*_{g^{-1}}(\mu)$*. The infinitesimal generator of this action with respect to* $\xi \in \mathfrak{g}$ *is*

$$\xi_{\mathfrak{g}^*}(\mu) = \mathrm{ad}^*_\xi(\mu). \tag{3.7}$$

For this particular example it turns out that the generator map is expressible by the Hamiltonian. Next, we will introduce the notion of the functional derivative, which classically is nothing else than the gradient of a function. The functional derivative $\frac{\delta F}{\delta \mu}$ *of a function* $F : \mathfrak{g}^* \to \mathbb{R}$ *is the element in the Lie algebra* $\mathfrak{g}$ *that satisfies the relation*

$$\lim_{\epsilon \to 0} \frac{1}{\epsilon} \left(F(\mu + \epsilon\xi) - F(\mu)\right) = \langle \xi, \frac{\delta F}{\delta \mu} \rangle. \tag{3.8}$$

The functional derivative can be viewed as a function from $\mathfrak{g}^*$ *to* $\mathfrak{g}$*, and hence it serves the role as a generator map for time-independent Hamiltonians. The Hamiltonian ODE on* $\mathfrak{g}^*$ *with Hamiltonian* H *takes the form*

$$\mu' = \mathrm{ad}^*_{\frac{\delta H}{\delta \mu}}(\mu). \tag{3.9}$$

What does this equation look like in concrete examples? The rigid body and the Euler equations pop up by considering the Hamiltonian $H(\mathbf{\Pi}) = \frac{1}{2}\mathbf{\Pi} \cdot I^{-1}\mathbf{\Pi}$ *modeled on the dual space of the Lie algebra* $\mathfrak{so}(3)$*. In this case (3.9) reduces to the Euler equations*

$$\frac{d\mathbf{\Pi}}{dt} = \mathbf{\Pi} \times I^{-1}\mathbf{\Pi}.$$

Chapter 4

Domain

In *DiffMan* 1.6 the following collections of domains are implemented:

```
hmanifold          % Homogeneous spaces
liealgebra         % Lie algebras
liegroup           % Lie groups
```

As a user you will work exclusively with homogeneous spaces. Thus, the collections of Lie algebras and Lie groups are mere geometric building blocks for the homogeneous spaces.

Each of the collections of domains contains a virtual superclass of that collection. The purpose of this superclass is that it serves as the parent of all the other classes, and all functions common to all the classes in the collection are implemented once and for all in this superclass. Thus, all the actual matrix class implementations contain far fewer functions than the number of functions implemented in the virtual superclass. For an overview of the actual functions implemented in the superclasses see Appendix C.

In MATLAB a class has to be prefixed by `@` in order to be recognized as a class. The constructor of the particular `@myclass` has to be an m-file with the name `myclass.m`. In the virtual superclass of each collection of domains we have overloaded ordinary arithmetic operations in order to make them work for the objects. Since the virtual superclass is the parent of all the implemented classes, they inherit these operations. The advantage of this approach is that one saves a lot of coding if a new class is to be added, and if changes has to be made these are only made one place. Do not forget that these inherited functions can again be overloaded in each of the particular classes.

In the next three subsections, all the implemented classes in each of the domain collections are listed.

4.1 Homogeneous spaces

The homogeneous space classes implemented in *DiffMan* are the following:

```
@hmanifold   - The virtual superclass
@hmisospec   - Isospectral flow: action of Lie algebra on M \in R^(nxn).
@hmlie       - The action of any Lie group and algebra on itself.
@hmnsphere   - The N-sphere with action of the orthogonal group.
@hmrigid     - The rigid body modeled as a homogenous space.
@hmrn        - The action of any algebra on R^n.
```

```
@hmtop      - The heavy top modeled as a homogenous space.
```

The virtual superclass homogeneous space is named `@hmanifold`. All the other homogeneous spaces are prefixed by `@hm` in order to emphasize that these are actual implementations of homogeneous spaces.

4.2 Lie algebras

The Lie algebra classes implemented in *DiffMan* are the following:

```
@ladirprod    - The direct product of Lie algebras.
@lafree       - The free Lie algebra of q elements up to order s.
@lagl         - The Lie algebra of the general linear group.
@larn         - The Lie algebra of R^n.
@lase         - The special Euclidean algebra.
@lasl         - The Lie algebra of the special linear group.
@laso         - The Lie algebra of the special orthogonal group.
@lasp         - The symplectic Lie algebra.
@liealgebra   - The virtual superclass of all Lie algebras.
```

The virtual superclass Lie algebra is named `@liealgebra`. All the other algebras are prefixed by `@la` in order to emphasize that these are actual implementations of Lie algebras.

4.2.1 The free Lie algebra

In this section we want to show you in a very rudimentary way how to use the free Lie algebra class `@lafree`. See also Section 7.4.2 for other examples.

The commands given below reflect the basic operations in the free Lie algebra class `@lafree`.

`>> 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. If no grades are supplied, grades equal to 1 are used.

`>> 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}` is a list of elements from *any DiffMan* Lie algebra, this will evaluate the expression `E`, using the data set `Y1,Y2, ... ,Yp` in place of the generating set.

Examples of use of these commands are:

```
>> fla = lafree({[3,3]})
fla =
   LieAlgebra class: lafree
   Shape:
     {[3  3],[1  1  1]}
```

```
   Basis vectors:
     1:[1]  2:[2]  3:[3]  4:[1,2]  5:[1,3]  6:[2,3]  7:[1,[1,2]]
     8:[1,[1,3]]  9:[2,[1,2]]  10:[2,[1,3]]  11:[2,[2,3]]  12:[3,[1,2]]
     13:[3,[1,3]]  14:[3,[2,3]]
```

```
>> x1 = basis(fla,1)
x1 =
   LieAlgebra class: lafree
   Data:
     [1]
>> x4 = basis(fla,4)
x4 =
   LieAlgebra class: lafree
   Data:
     [1,2]
>> x=random(lagl(2))
x =
Class: lagl
Shape: 2
Data:
   992/1139    -166/925
  1169/1402    1029/1307
>> y = random(x), z = random(x)
y =
Class: lagl
Shape: 2
Data:
 -1367/1546    1527/2438
 -1185/4027   -1342/1369
z =
Class: lagl
Shape: 2
Data:
-10067/13939  -1037/1721
  -129/217      104/501
>> eval(x7,cat(1,x,y,z))
ans =
Class: lagl
Shape: 2
Data:
   -76/989     -271/1666
  -452/571       76/989
```

```
>> x7 = basis(fla,7)
x7 =
   LieAlgebra class: lafree
   Data:
     [1,[1,2]]
>> [x1 x4] - x7
ans =
   LieAlgebra class: lafree
   Data:
     0
```

4.3 Lie groups

The Lie group classes implemented in *DiffMan* are the following:

```
@lgdirprod  - The direct product of Lie groups.
@lggl       - The general linear group.
@lgon       - The orthogonal group.
@lgrn       - The Lie group R^n
@lgse       - The special Euclidean group.
@lgsl       - The special linear group.
@lgso       - The special orthogonal group.
@lgsp       - The symplectic group.
```

```
@liegroup   - The virtual superclass of all Lie groups.
```

The virtual superclass Lie group is named `@liegroup`. All the other groups are prefixed by `@lg` in order to emphasize that these are actual implementations of Lie groups.

4.4 My domain is not implemented!?

Eventually, *DiffMan* will contain all possible domain classes that you can think of. Since this is just *DiffMan* 1.6 you might encounter the fact that the homogeneous space or Lie group that you want to use as a domain in your application, is not yet implemented.

If that is to happen you have at least two options: You can carefully read this guide – which is by no means complete – and peek into the source code of the other classes and try to implement the missing class yourself. If you do this we would appreciate obtaining a copy of your class so that it can be included in the general distribution. We are more than willing to answer any of your questions if you choose to do it this way.

A second option is to send us a request, and let us know about the classes that are missing, so that we can possibly include them in a future version of *DiffMan*.

Chapter 5

Field

In *DiffMan* 1.6 the only field implemented is vector fields over domains.

5.1 Vector field

A vector field F over a manifold $\mathcal{M}$ is a section of the tangent bundle of the manifold, $F : \mathcal{M} \to \mathrm{T}\mathcal{M}$. This means that at each point $m \in \mathcal{M}$, the vector field computes a tangent vector $F(m) \in \mathrm{T}\mathcal{M}_m$, or, equivalently, that $\pi \circ F = \mathrm{id}_{\mathcal{M}}$, where π is the projection from $\mathrm{T}\mathcal{M}$ to $\mathcal{M}$.

A vector field on a manifold defines the flow, and integral curves of vector fields are solutions of differential equations.

DiffMan 1.6 is based on domains that are homogeneous manifolds. As is described in Appendix A, the function evaluating the vector field should take $t \in \mathbb{R}$ and $y \in \mathcal{M}$ as arguments and return an element in the Lie algebra of the Lie group which, together with the Lie-group action and the manifold, define the homogeneous space.

The integral curves of a vector field, or, in other words, the solution of the differential equation defined by the vector field, are computed numerically in *DiffMan*. A vector field over a domain is constructed by the `vectorfield` operation. In the following example, we define an object `y` to belong to the homogeneous space defined by `hmlie` over the Lie group SO(3):

```
>> y=hmlie(lgso(3));
>> vf=vectorfield(y)
vf =
Class:  vectorfield
Domain: hmlie
Shape-object information:
   Class: lgso
   Shape: 3
Eqn type:   General
```

The vector field object is now defined, but we have to assign to it a function which actually computes the tangent vectors. Example 1 in *DiffMan* gives an example of such a function (function `vfex1`):

```
function [la] = vfex1(t,y)
% VFEX1 - Generator map from RxM to the Lie algebra. Linear type.
```

```
la = liealgebra(y);
dat = [0 t 1; -t 0 -t^2; -1 t^2 0];
setdata(la,dat);
return;
```

The function should in general receive `t` and `y` as input arguments and return a Lie algebra element. The Lie algebra is the one corresponding to the Lie group which defines, together with the Lie-group action and the manifold, the homogeneous space.

Using `setfmg`, the function is assigned to a `vectorfield` object:

```
>> setfm2g(vf,'vfex1')
```

and information about the current map is retrieved using `getfm2g`:

```
>> getfm2g(vf)
ans =
vfex1
```

When we initialized `vf` using `vectorfield`, the output

```
Eqn type:   General
```

appeared on the screen. `General` means that the vector field is of *general* type, i.e. it depends both on `t` and `y`. When the vector field only depends on `t`, we say it is a *linear* vector field. The *type* of the vector field can be changed using `seteqntype`:

```
>> seteqntype(vf,'L')
```

Type information about the current vector field is retrieved using `geteqntype`:

```
>> geteqntype(vf)
ans =
Linear
```

If we want to change the domain of the vector field, we may use the function `setdomain`. Information about current domain is retrieved using `getdomain`:

```
>> setdomain(vf,hmlie(lgso(4)))
>> getdomain(vf)
ans =
Class: hmlie
Shape-object information:
   Class: lgso
   Shape: 4
```

Chapter 6

Flow

The underlying assumption of the time-integrators in *DiffMan* is the existence of a Lie group G with Lie algebra $\mathfrak{g}$ that is endowed with Lie bracket $[\cdot,\cdot]$, a (left) Lie-group action $\Phi : G \times \mathcal{M} \to \mathcal{M}$ and a function $\xi : \mathbb{R} \times \mathcal{M} \to \mathfrak{g}$ such that the ordinary differential equation describing the problem can be written in the form (see Appendix A)

$$y' = \xi_{\mathcal{M}}(t, y), \qquad y(0) = p \in \mathcal{M}. \tag{6.1}$$

The flow operator of this differential equation is the operator $\Phi_{\exp(t\xi)} : \mathcal{M} \to \mathcal{M}$, where ξ_M is the infinitesimal generator of the action corresponding to $\xi \in \mathfrak{g}$,

$$\xi_{\mathcal{M}}(q) = \left.\frac{\mathrm{d}}{\mathrm{d}t}\right|_{t=0} \Phi_{\exp(t\xi)}(q), \qquad \forall\, q \in \mathcal{M}.$$

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.

In *DiffMan* the flow is approximated numerically using a particular time stepper method. The virtual superclass defining the flow is

```
@flow              % The virtual flow superclass
```

A flow object is defined by a vector field object and a time stepper object. The default time stepper object if from the class **tsrkmk** with coordinates **exp** and the classical fourth order Runge-Kutta method **RK4** as scheme. The flow constructor is used as follows:

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

When evaluated, the **flow** object approximates the integral curve of the flow through an initial point from time t_{start} to t_{end} using either constant or variable step size. The outer part of the time stepping process is included in **flow**, while the computation of one single step is done using the time stepper object. The advantage of this is that we write the outer part only once. The variable step size control resides here, and so do the other control structures defining the integration process. When changing time stepper object, the environment and integration parameters remain the same, and the effect of changing the time stepper object can be measured.

The `flow` object must be associated with a vector field (see e.g. Section 5.1). This is done using the `setvectorfield` routine. Let `y` be the object `hmlie(lgso(3))`. Then

```
>> vf=vectorfield(y)
vf =
Class:  vectorfield
Domain: hmlie
Shape-object information:
   Class: lgso
   Shape: 3
Eqn type:   General
```

and this vectorfield is given to the `flow` object through

```
>> setvectorfield(fl,vf)
```

The vectorfield coupled to a `flow` object is retrieved using `getvectorfield`.

In the `flow` object a number of parameters are defined. Most of these are used in the time stepper routines. The `getdefaults` routine returns the default setting:

```
>> getdefaults(fl)
ans =
        small: 0.5000
        large: 2
...
         disp: 1
```

The values are changed using `setdefaults`:

```
>> setdefaults(fl,'small',0.1)
>> getdefaults(fl)
ans =
        small: 0.1000
        large: 2
...
         disp: 1
```

6.1 Time stepping

In *DiffMan* 1.6 the following numerical time stepper algorithms are implemented:

```
@timestepper        % The virtual superclass timestepper
@tscg               % The Crouch-Grossman methods
@tsfer              % The Fer expansion methods
@tsmagnus           % The Magnus series methods
@tsqq               % Quadrature methods for quadratic Lie groups
@tsrk               % Classical Runge-Kutta methods
@tsrkmk             % Runge-Kutta methods of Munthe-Kaas type
```

A detailed mathematical exposition of each of the methods is found in Appendix A. A future release of *DiffMan* will also contain various multistep methods.

Initialization of a time stepper of type `tscg` is done as follows:

```
>> ts=tscg
ts =
Class:  tscg
Coord.: exp
Method: CG3a
```

The default coordinates are `exp` and the default scheme is `CG3a`. A list of the schemes available in *DiffMan* 1.6 is included in Appendix B. You can change time stepper scheme using `setmethod`:

```
>> setmethod(ts,'CG43')
```

and you can retrieve the parameter values of the scheme using `getmethod`:

```
>> getmethod(ts)
ans =
    RKname: 'CG43'
       RKa: [5x5 double]
       RKb: [0.6756 0 -0.1756 -0.1756 0.6756]
       RKc: [0 1.5000 1.3512 -0.3512 1]
      RKns: 5
     RKord: 4
    RKtype: 'explicit'
    RKbhat: [0.6756 0 -0.1756 -0.1756 0.6756]
```

Many of the methods are based on a certain choice of coordinate system. *DiffMan* 1.6 provides the following coordinates:

```
exp      % canonical coordinates of the first kind;
         %   defined by a single exponential map
expexp   % canonical coordinates of the second kind;
         %   defined by a product of exponential maps
cay      % coordinates based on the Cayley map;
         %   defined by a single Cayley map
caycay   % coordinates based on the Cayley map;
         %   defined by a product of Cayley maps
pade22   % coordinates based on the (2,2) diagonal Pade map;
         %   defined by a single (2,2) diagonal Pade map
```

See e.g. [5] for a discussion of different choices of coordinates. The default coordinate choice is the exponential mapping for most of the methods. You can change coordinates by using the `setcoordinate` routine:

```
>> setcoordinate(ts,'cay')
```

The `getcoordinate` routine returns the current choice of coordinate system.

You may change time stepper in *DiffMan* by using the `settimestepper` routine:

```
>> ts=tsmagnus
ts =
Class:  tsmagnus
Coord.: exp
Method: M4a
>> settimestepper(fl,ts)
```

The `gettimestepper` routine retrieves information from the `flow` object:

```
>> gettimestepper(fl)
ans =
Class:  tsmagnus
Coord.: exp
Method: M4a
```

6.1.1 My time stepper is not implemented!?

Eventually, *DiffMan* will contain most of the time stepper classes described in the literature. In *DiffMan* 1.6 you might encounter the fact that the time stepper that you want to choose is not yet implemented. If that is to happen you have at least two options: You can carefully read this guide – which is by no means complete – and peek into the source code of the other classes and try to implement the missing class yourself. If you do this, the *DiffMan* development team would appreciate receiving a copy of your class so that it can be included in the general distribution. We are more than willing to answer any of your questions if you choose to do it this way.

A second option is to send us a request, and let us know about the classes that are missing. We will then possibly include them in a future version of *DiffMan*.

6.2 Implicit methods — Solution of nonlinear equations

In *DiffMan* 1.6 the nonlinear systems of equations arising from the implicit time steppers are solved using fixed point iteration only. The iteration on the stage values continue until the error is below $\max(10^{-12}, h^p/100)$, where p is the order of the method and h is the step size.

In future versions of *DiffMan*, we will provide general nonlinear equation solvers, e.g. Newton iteration as developed by Owren and Welfert [35]. These schemes reduce to the classical Newton-iteration scheme in Euclidean space with the standard basis.

6.3 Variable step sizing

Traditional step size control strategies rely on the local error of the method. At every step this quantity is estimated, and a new step size is computed. It is natural to employ similar techniques also in the case of integration on Lie groups or more general manifolds. Description of the strategy may be found in most standard texts on integration methods for ordinary differential equations, but for completeness we include a brief overview of the procedure.

In order to attain the local error estimate $r_{k+1} = \varepsilon$ at time step $k+1$, the next step size h_{k+1} is chosen as a function of the previous step size, h_k, as follows (see e.g. [36] or [10, 11]). Let p be the order of the method. When using embedded pairs, as is usual in classical Runge-Kutta methods, we let p be the order of the lower-order approximation scheme). Furthermore, $r_k = e_k$, the error estimate, and $\alpha_{\text{pessimist}}$ is a pessimist factor which is heuristically determined (typical values are 0.8 or 0.9). We first compute

$$\widehat{h}_{k+1} = \alpha_{\text{pessimist}} \left(\frac{\varepsilon}{r_k}\right)^{1/(p+1)} h_k,$$

A typical strategy to prevent rapid oscillations of the step size is to restrict the extent of step size variation in any single step. This is obtained by letting

$$h_{k+1} = \min\{h_{\max}, \max\{\alpha_{\text{small}} h_k, \min\{\alpha_{\text{large}} h_k, \widehat{h}_{k+1}\}\}\},$$

where $h_{\max}$ is the largest allowed step size, while α_{small} and α_{large} are two constants. If the local error exceeds the tolerance by a factor more than α_{accept}, then we reject the step and retry with a smaller step size computed as above but with h_k equal to the step size we just tried. This algorithm proceeds until the local error estimate satisfies

$$e_{k+1} \leq \alpha_{\text{accept}} \varepsilon.$$

DiffMan defines default values for the constants used in variable step size integration. The names are

$\alpha_{\text{pessimist}}$	$\leftrightarrow$	`pessimist`
α_{small}	$\leftrightarrow$	`small`
α_{large}	$\leftrightarrow$	`large`
α_{accept}	$\leftrightarrow$	`accept`
ε	$\leftrightarrow$	`tol`
$h_{\max}$	$\leftrightarrow$	`hmax`

All the values can be changed by the user of *DiffMan* (see description of the flow object). The routine `getdefaults` returns the default setting of the constants:

```
>> fl=flow;
>> getdefaults(fl)
ans =
        small: 0.5000
        large: 2
    pessimist: 0.9000
...
         disp: 1
```

The values are changed using `setdefaults`:

```
>> setdefaults(fl,'large',4)
>> getdefaults(fl)
ans =
        small: 0.5000
        large: 4
    pessimist: 0.9000
...
         disp: 1
```

The variable step size algorithm will be extended in future releases of *DiffMan*.

Chapter 7

Auxiliary

7.1 Demos

Issuing the MATLAB command `demo` on the command line in *DiffMan* will launch the MATLAB Demos. Since this is done while *DiffMan* is running you will be able to find *DiffMan* and its demonstrations as one of the MATLAB toolboxes. The situation should look very similar to Figure 7.1.

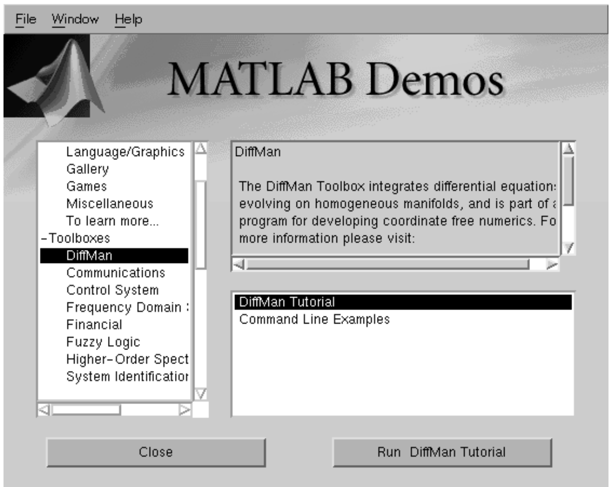


Figure 7.1: The MATLAB Demos.

The Demos feature in MATLAB is very simple to use. You just high-light what you want to learn more about, and then push the button down to the right ('Run ...'-button) to go on to more information about the specific topic. For *DiffMan* 1.6 there are two topics for further help: 'The *DiffMan* tutorials' and 'The Command Line Examples'.

Under '*DiffMan* tutorials' you will find short and well-documented slide shows which guide you through different aspects of how to use *DiffMan*. Figure 7.2 shows you the 'Tutorial' window as launched from the 'Demo' window. The two existing tutorials are on how to use *DiffMan* to solve ODEs on manifolds, and about the domains in *DiffMan*. More tutorials will be tried added in the future. If you have ideas for such tutorials please inform the *DiffMan* development team: `diffman@math.ntnu.no`.

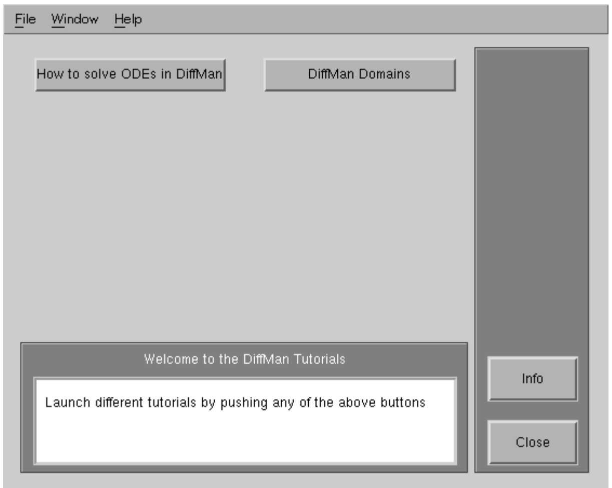


Figure 7.2: The *DiffMan* tutorials.

'Command Line Examples' collects 10 of the examples found in the *DiffMan* distribution. Accessing the CL-examples this way, the user simply has to push a button in order to run an example. See Section 7.3 for more information.

7.2 Documentation

All the *DiffMan* documentation is placed in the directory `DiffMan/auxiliary/documentation`. The User's Guide, which you are reading now, is found as both a PDF and PS-file. There are also other `.doc` and `.m`-files in this directory which are worth while looking at. However, in the present distribution of *DiffMan* they are not guaranteed to be finished.

When detailed technical notes for instance about new classes or specific problems solved in *DiffMan* are released by the *DiffMan* development team, these will also be placed in this directory as PDF and/or PS-files.

7.3 Examples

In *DiffMan* you can find several examples of solved ODEs. In *DiffMan* 1.6 these are:

```
dmex1    -          This is an ODE evolving in SO(3)
dmex1a   -          This is an ODE evolving in SO(3)
           PURPOSE: Integrate 'dmex1' with variable and constant step size.
dmex1b   -          This is an ODE evolving in SO(3)
           PURPOSE: Integrate 'dmex1' with variable and constant step size
                    for time steppers: 'tscg' and 'tsrk'.
dmex2    -          Solution of the Lorenz equations in R^3
dmex3    -          This is a linear ODE evolving in SO(3)
dmex4    -          Integration of the rigid body
dmex5    -          This is an ODE evolving on the sphere S^2
dmex6    -          Integration of an isospectral flow
dmex7    -          Integration of the Airy equation
dmex8    -          Integration of the van der Pol equation
```

```
dmex9    -          Integration of the heavy top
dmex10   -          This is an ODE evolving in SO(3)
dmex10a  -          This is an ODE evolving in SO(3)
           PURPOSE: Use "ftorder" to verify the order of the solution
                    of some fer schemes
           WARNING: This routine is very time consuming to run.
exlafree -          Using the Lie algebra @lafree.
           PURPOSE: Use the free Lie algebra to establish a
                    connection between the Magnus method and the RKMK
                    method.
extools1 -          Example of ODE evolving in SO(3).
           PURPOSE: Demonstrate the use of the flow tools routine
                    "ftorder", that computes order of approximation
                    of a given integrator and scheme.
extools2 -          Example of ODE evolving in SO(3).
           PURPOSE: Demonstrate the use of the flow tools routine
                    "fteff", that computes the efficiency of a given
                    integrator and scheme as global error versus
                    flops.
```

All these examples can be run directly in *DiffMan* by just typing the name of the example. If you are a beginner user of *DiffMan*, it might also be worth-while to view the source code of the examples with `dmhelp`. This gives you an idea of how things are done, and probably the best way to learn is to take an already existing example, copy it, and modify it, and see what happens.

The examples `dmex1` through `dmex10` are the ones found among the 'Command Line Examples' in the *DiffMan* Demo, see Figure 7.3. In order to run the examples this way, you push the example button. Make sure that the MATLAB window is visible, because output from the examples will be displayed here and in plots.

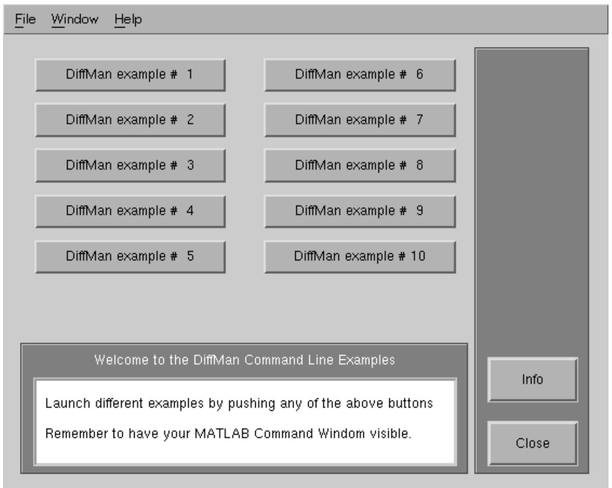


Figure 7.3: The *DiffMan* command line examples.

7.4 Utilities

The root utilities directory contains some functions which are 'nice' to have. It is not vital to know how to use all these functions, but it is very convenient to have an idea of where to look for these types of functions, in case you are in need of one. The functions located in the root utilities directory are:

```
addsubdir     - Add subdirectories to the path.
array         - Create arrays of objects.
dmargcheck    - Turn on/off input checking in DiffMan.
dmhelp        - The help facility in DiffMan.
dmprogrep     - Turn on/off progress report.
iseven        - Function to test if a number is even.
iscellempty   - Check if a cell tree consists of empty matrices.
iseven        - Check if integer is even.
repprogress   - The progress report m-file.
isinteger     - Function to test if a number is integer.
skew          - Creates a skew matrix from a 3-vector.
```

There are also two subdirectories containing class specific utilities functions:

```
flowtools     - Functions for efficiency and order checks.
lafreeutil    - Utility functions for the free Lie algebra.
```

For more information about the functions contained in these subdirectories see the next two subsections.

7.4.1 `flowtools`

DiffMan 1.6 provides the user with two special `flow` utility routines: `ftorder` and `fteff`. `ftorder` estimates both the local and global order of approximation of a timestepper. The user can provide a "correct" (or "exact") integrator. If this is not supplied, the `tsrkmk` timestepper with the `RK4` scheme is used. The first time this routine is called, an "exact" reference solution is computed. This is a rather time consuming operation, and can be avoided in the next call with the same setting. The user should notify `ftorder` that the same "exact" solution should be used, by changing the value of the flow parameter `newexact`:

```
>> setdefaults(fl,'newexact',0)
```

The example routine `extools1` uses this tool:

```
...
flA = flow; % The integrator to be estimated
ts = tsrkmk;
setcoordinate(ts,'pade22');
setmethod(ts,'RKF34');
settimestepper(flA,ts);
...
flB = flow; % "Reference" integrator - computes the "exact" solution
ts = tsrkmk;
setcoordinate(ts,'exp');
setmethod(ts,'RK4');
```

```
settimestepper(flB,ts);
...
out=ftorder(flA,flB,y);
...
% input exact solution to avoid computing it once more - example
setdefaults(flA,'newexact',0);
out2=ftorder(flA,flB,y,[],[],[],out.exact);
...
```

The `fteff` utility routine estimates the efficiency of a timestepper. The efficiency measure is global error as a function of number of flops used by the integrator. The MATLAB routine `flops` is used to count the flops usage. The sample program `extools2` demonstrates the use of the routine:

```
...
flA = flow; % The integrator to be estimated
ts = tsrkmk;
setcoordinate(ts,'pade22');
setmethod(ts,'RKF34');
settimestepper(flA,ts);
...
flB = flow; % "Reference" integrator - computes the "exact" solution
ts = tsrkmk;
setcoordinate(ts,'exp');
setmethod(ts,'RK4');
settimestepper(flB,ts);
...
out=fteff(flA,flB,y);
...
% input exact solution to avoid computing it once more - example
out2=fteff(flA,out.sol,y);
...
```

7.4.2 `lafreeutil`

This subdirectory collects some handy functions related to the free Lie algebra `@lafree`:

```
bch         - Computes the BCH formula of order q, and n flows.
bch2        - Computes the BCH formula of order q, just 2 flows.
cdopri5     - Classical dopri5 of order 5.
divisors    - All the divisors of an integer.
killing     - Killing form on a Lie algebra.
lafcount    - Count commutators in a graded free Lie algebra.
lafdim      - Dimension of a graded free Lie algebra.
magnusdim   - Magnus dimension.
mobius      - Number theoretic mu function.
rkgl        - Runge-Kutta-Gauss-Legendre coefficients of n stages, order 2n.
rkmk4       - Classical RKMK of order 4.
rkmk4mod    - Modified classical RKMK of order 4.
slegendre   - Shifted Legendre polynomial of order n.
```

For instance, `bch` and `bch2` can be used like this:

```
>> format rat
>> bch(3,3)
Fixpoint iteration, 3 steps:
```

```
   Done step: 1
   Done step: 2
   Done step: 3
Composing 3 flows:
ans =
computing commtab!
   Done step: 3
ans =
   LieAlgebra class: lafree
   Data:
     [1] + [2] + [3] + 1/2*[1,2] + 1/2*[1,3] + 1/2*[2,3] + 1/12*[1,[1,2]]
      + 1/12*[1,[1,3]] - 1/12*[2,[1,2]] + 1/6*[2,[1,3]] + 1/12*[2,[2,3]]
      - 1/3*[3,[1,2]] - 1/12*[3,[1,3]] - 1/12*[3,[2,3]]
>> bch2(4)
ans =
   LieAlgebra class: lafree
   Data:
     [1] + [2] + 1/2*[1,2] + 1/12*[1,[1,2]] - 1/12*[2,[1,2]] - 1/24*[2,[1,[1,2]]]
```

7.5 Efficiency and speed-up of *DiffMan*

First, let us state the following fact: the object orientation in MATLAB is not fast. We have tried to remedy this situation in *DiffMan* by a couple of programming tricks. The first 'trick' is to use MEX-files, and the second 'trick' is to aid the most time-consuming computation by precomputations in the free Lie algebra. In the present version of *DiffMan* only the first of these two tricks is available for exploitation.

7.5.1 Speed-up through MEX-files

Following the standard *DiffMan* 1.6 distribution there are three pre-programmed files in C that the user can compile in order to speed up the execution of *DiffMan*. A prerequisite for this to work is that there is a C compiler installed on the platform which is running MATLAB. The files are located in the directory `DiffMan/domain/liealgebra/@liealgebra`, and they are:

```
>> pwd
ans =
/../DiffMan/domain/liealgebra/@liealgebra
>> ls *.c
ans =
dexpinv.c
horzcat.c
mtimes.c
```

To compile the C files and make MEX-files you can issue the following commands in MATLAB:

```
>> mex -O dexpinv.c
>> mex -O horzcat.c
>> mex -O mtimes.c
>> ls *.mexsol
ans =
dexpinv.mexsol
horzcat.mexsol
mtimes.mexsol
```

Note that the file extension of the MEX-files will vary depending on the platform on which you are running MATLAB.

What happens now is that the next time you are running your *DiffMan* application, MATLAB will choose the MEX-files instead of the m-files for any of the three functions `dexpinv.m`, `horzcat.m`, and `mtimes.m`. This is done automatically, and the only way that the user can control this is by removing the MEX-files. The MEX-files are well tested on all the matrix groups, and they are expected to fail seriously if used on any of the functorial classes. A word of advise is that if you run into any kind of trouble using the MEX-files, remove them, issue the command `clear functions` in MATLAB, and run your problem just using the m-files (which are known to work properly).

Please be warned that the flops count which MATLAB produces when running any of the above MEX-files is not correct. The reason for this is that the flops counter is not being updated in any of the MEX-files. Hence, the results yielded by the function `fteff` are erronous when used together with these MEX-files.

7.5.2 Speed-up by use of the free Lie algebra

Unfortunately, this did not make it in time for the release of *DiffMan* 1.6. At the time of release of *DiffMan* 1.6 this was something still on the test bed, but it will definitely be included in the next release of *DiffMan*.

To give you an idea of this beautiful trick we will briefly comment on it. The trick will probably be implemented in *DiffMan* as a new version of the time stepper class `@tsrkmk`. Hence, the user will only notice the speed increase, and does not have to do anything special than choosing the 'right' RKMK time stepper class in order to take advantage of it. The trick itself constitutes of aiding the most time-consuming computation by a precomputation in the free Lie algebra `@lafree`. The object from the `@lafree` class represents the whole update of step size h that the numerical method is supposed to perform on the initial data. The free Lie algebra object acts as an operator, and once calculated the object is saved for the rest of the iteration, and whenever the time stepper is called, the object in the free Lie algebra is evaluated on the data. Hence, you end up only with an evaluation in each step, instead of a time-consuming calculation.

More on this in the next release of *DiffMan*.

Bibliography

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

[2] R. L. Bryant. An introduction to Lie groups and symplectic geometry. In D. S. Freed and K. K. Uhlenbeck, editors, *Geometry and Quantum Field Theory*, volume 1, Second Edition of *IAS/Park City Mathematics Series*. American Mathematical Society, 1995.

[3] E. Celledoni and A. Iserles. Approximating the exponential from a Lie algebra to a Lie group. Technical Report 1998/NA3, Department of Applied Mathematics and Theoretical Physics, University of Cambridge, England, 1998.

[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, 1998.

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

[7] K. Engø and A. Marthinsen. Modeling and solution of some mechanical problems on Lie groups. *Multibody System Dynamics*, 2:71–88, 1998.

[8] K. Engø, A. Marthinsen, and H. Z. Munthe-Kaas. Diffman — an object oriented MATLAB toolbox for solving differential equations on manifolds. Technical Report No. 164, Department of Informatics, University of Bergen, 1999.

[9] 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.

[10] K. Gustafsson. *Control of Error and Convergence in ODE Solvers.* PhD thesis, Department of Automatic Control, Lund Institute of Technology, Sweden, 1992.

[11] K. Gustafsson. Control-Theoretic Techniques for Stepsize Selection in Implicit Runge-Kutta Methods. *ACM TOMS*, 20(4):496–517, 1994.

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

[13] 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.

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

[15] A. Iserles, A. Marthinsen, and S. P. Nørsett. On the implementation of the method of Magnus series for linear differential equations. *BIT*, 39(2):281–304, 1999.

[16] 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.

[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. Interpolation in Lie groups and homogeneous spaces. Technical Report 139, Department of Mathematics, Arizona State University, Tempe, Arizona, 1998.

[20] A. Marthinsen, H. Munthe-Kaas, and B. Owren. Simulation of ordinary differential equations on manifolds — some numerical experiments and verifications. *Modeling, Identification and Control*, 18:75–88, 1997.

[21] A. Marthinsen and B. Owren. A note on the construction of Crouch–Grossman methods. Technical Report Numerics No. 2/1998, The Norwegian University of Science and Technology, Trondheim, Norway, 1998.

[22] 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.

[23] R. I. McLachlan. Explicit symplectic splitting methods applied to PDE's. In E. L. Allwoger, K. Georg, and R. Miranda, editors, *Lectures in Applied Mathematics: Exploiting Symmetry in Applied and Numerical Analysis*, pages 325–337. American Mathematical Society, 1993.

[24] R. I. McLachlan. On the numerical integration of ordinary differential equations by symmetric composition methods. *SIAM J. Sci. Comput.*, 16:151–168, 1995.

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

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

[27] H. Munthe-Kaas. High order Runge–Kutta methods on manifolds. *Appl. Numer. Math.*, 29:115–127, 1999.

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

[29] 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.

[30] 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.

[31] 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.

[32] P. J. Olver. *Applications of Lie Groups to Differential Equations.* GTM 107. Springer-Verlag, Second edition, 1993.

[33] B. Owren and A. Marthinsen. Integration methods based on canonical coordinates of the second kind. Manuscript 1999.

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

[35] B. Owren and B. Welfert. The Newton iteration on Lie groups. Technical Report Numerics No. 3/1996, Norwegian University of Science and Technology, Trondheim, Norway, 1996.

[36] H. J. Stetter. *Analysis of Discretization Methods for Ordinary Differential Equations.* Springer-Verlag, Berlin, 1973.

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

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

[39] 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. To appear in *SIAM J. Num. Anal.*

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

[41] A. Zanna and H. Munthe-Kaas. Iterated commutators, Lie's reduction method and ordinary differential equations on matrix Lie groups. In F. Cucker and M. Shub, editors, *Foundation of Computational Mathematics*, pages 434–441. Springer-Verlag, 1997.

Appendix A

The mathematical building blocks of *DiffMan*

In this chapter we briefly describe the mathematical background of *DiffMan*. To get a more detailed exposition of the topics the reader is recommended to study some of the more established references like [1, 2, 18, 32, 37, 38]

A.1 Homogeneous spaces

To really appreciate *DiffMan*, the most important concept to understand is the notion of a homogeneous space. All the numerical algorithms for solving ordinary differential equations have been reformulated to solve problems evolving on homogeneous spaces. We believe this to be the most general setting for solving ordinary differential equations on manifolds. It should be noticed that all the methods, when applied in the traditional setting of $\mathbb{R}^N$, reduce to the classical Runge–Kutta schemes.

Before defining homogeneous spaces, we need some mathematical building blocks.

Definition A.1.1 (Manifold)
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 $\phi_i(U_i \cap U_j)$.

While this is a quite abstract definition, in *DiffMan* we will mainly concentrate on a particular class of differentiable manifolds called *Lie groups*.

Definition A.1.2 (Lie group)
A Lie group $(G, \cdot)$ is a differential manifold G equipped with a binary operation $\cdot$ on G satisfying the axioms of a group, such that the map $\psi : G \times G \to G$ given by $\psi(q,p) = q \cdot p^{-1}$ is smooth.

In this report we will slightly abuse notation and just refer to the Lie group as G. The identity element in the Lie group will be denoted by e.

Definition A.1.3 (Tangent space)
Given a point $p \in \mathcal{M}$, we denote by $\mathrm{T}\mathcal{M}|_p$ the tangent space of $\mathcal{M}$ at p. This space is the set of

all linear derivations v_p such that

$$v_p(\lambda f_1 + \mu f_2) = \lambda v_p(f_1) + \mu v_p(f_2) \qquad \text{and} \qquad v_p(f_1 f_2) = f_1(p)v_p(f_2) + f_2(p)v_p(f_1)$$

for all functions $f_1, f_2 : \mathcal{M} \to \mathbb{R}$ defined in a neighborhood of p, and all $\lambda, \mu \in \mathbb{R}$.

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 $\mathcal{M}$, is a section of $\mathrm{T}\mathcal{M}$, i.e. to each point $p \in \mathcal{M}$ it associates a vector $X(p) \in \mathrm{T}\mathcal{M}|_p$.

Since a Lie group, G, is a manifold, there exists a tangent space, $\mathrm{T}G|_g$, at each point $g \in G$.

Definition A.1.4 (Lie algebra)
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,

$$[u,[v,w]] + [v,[w,u]] + [w,[u,v]] = 0,$$

when $u, v, w \in \mathfrak{g}$. We call $[\cdot,\cdot]$ the Lie bracket on $\mathfrak{g}$.

The Lie algebra of a Lie group is defined as the tangent space at the identity, $\mathfrak{g} = \mathrm{T}G|_e$. In this case its Lie bracket is given by

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

where $g(t), h(s) \in G$ are two curves such that $g(0) = h(0) = e$, $g'(0) = u$, and $h'(0) = v$. In the case of matrix Lie groups, the Lie bracket is the matrix commutator, $[u,v] = u \cdot v - v \cdot u$, where $\cdot$ denotes matrix multiplication.

Definition A.1.5 (Lie-group action)
A (left) Lie-group action of a Lie group G on a manifold $\mathcal{M}$ is a smooth mapping $\Phi : G \times \mathcal{M} \to \mathcal{M}$, which satisfies

$$\Phi(e,p) = p, \qquad \text{where } e \in G \text{ is the identity element,}$$

and

$$\Phi(g_1 \cdot g_2, p) = \Phi(g_1, \Phi(g_2, p)), \qquad \text{for all } g_1, g_2 \in G \text{ and } p \in \mathcal{M}.$$

We say that a Lie-group action is *effective* if $\Phi(g,p) = p$ for all $p \in \mathcal{M}$ implies that $g = e$. Furthermore, an action is *transitive* if, for arbitrary $p, q \in \mathcal{M}$, there exists a $g \in G$ such that $\Phi(g,p) \equiv q$, i.e. the space $\mathcal{M}$ consists of just a single orbit.

Based on these definitions we can now define the concept of homogeneous spaces.

Definition A.1.6 (Homogeneous space)
A homogeneous space is a manifold with a transitive Lie-group action.

Any $\xi \in \mathfrak{g}$ specifies 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 $g(0) = e$ and $g'(0) = \xi$. One may use $g(t) = \exp(t\xi)$, where $\exp : \mathfrak{g} \to G$ is the exponential map [27], or $g(t) = \phi(t\xi)$ for any smooth function $\phi : \mathfrak{g} \to G$ such that $\phi(0) = e$ and $\phi'(0) = I$. 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.

A.2 Computing flows of vector fields

We have chosen to view all the numerical methods implemented in *DiffMan* as working on homogeneous spaces. Even the classical Runge–Kutta methods are expressed in this setting.

The numerical methods in *DiffMan* assume that the differential equation to be solved is presented in the following *canonical form*

$$y' = F(t, y) = \xi_{\mathcal{M}}(t, y), \qquad y(0) = y_0 \in \mathcal{M}, \tag{A.1}$$

for some function $\xi : \mathbb{R} \times \mathcal{M} \to \mathfrak{g}$ (see [31, 27, 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.

A.2.1 Classical Runge–Kutta methods

Classical Runge–Kutta methods have been thoroughly described in a large amount of papers the last decades. These methods are implemented in *DiffMan*, but based on a slightly different understanding since we work on homogeneous spaces.

In the classical setting, the differential equation (A.1) takes on the following form. Integrate

$$u' = \xi\big(t, \Phi(u, y_0)\big), \qquad u(0) = 0,$$

with $y(t) = \Phi(u(t), y_0)$. We have here used that ϕ is the identity mapping in Euclidean space.

Algorithm A.2.1 (Classical Runge–Kutta methods)
Let $A = (a_{ij})$, $b = (b_j)$ and $c = (c_j)$ be the Butcher-coefficients of an s-stage, qth order Runge–Kutta method. The following algorithm integrates (A.1) from $t = t_n$ to $t = t_n + h$:

Assume that $y_n \approx y(t_n)$ is given
for $i = 1, 2, \ldots, s$
  $u_i = h \sum_{j=1}^{s} a_{ij} k_j$
  $k_i = \xi\left(t_n + c_i h, \Phi(u_i, y_n)\right)$
end
$v = h \sum_{j=1}^{s} b_j k_j$
$y_{n+1} = \Phi(v, y_n)$

When $\Phi(u, y) = u + y$, this reduces to the well known classical setting:

$$\begin{aligned} k_i &= \xi\left(t_n + c_i h, y_n + h\textstyle\sum_{j=1}^{s} a_{ij} k_j\right), \qquad i = 1, 2, \ldots, s, \\ y_{n+1} &= y_n + h \sum_{j=1}^{s} b_j k_j. \end{aligned}$$

If Φ is a general group action, this method attains at most order 2 on an arbitrary manifold.

A.2.2 Munthe-Kaas methods

The Munthe-Kaas methods were first described in [25]. In the succeeding paper [26], the methods were refined until their final formulation appeared in [27].

A large class of numerical algorithms are based on the assumption that the maps $\Phi(\phi(\xi), m)$ can be computed efficiently. The Munthe-Kaas methods in [27] are based on the choice $\phi(\xi) = \exp(\xi)$. More general ϕs are discussed in [5].

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

$$\begin{array}{ccc} \mathrm{T}\mathfrak{g} & \xrightarrow{\mathrm{T}\Phi_{y_0}\circ \mathrm{T}\phi} & \mathrm{T}\mathcal{M} \\ {\scriptstyle \mathrm{d}\phi_u{}^{-1}\circ\xi\circ\Phi_{y_0}\circ\phi}\uparrow & & \uparrow{\scriptstyle F=\xi_{\mathcal{M}}} \\ \mathfrak{g} & \xrightarrow[\Phi_{y_0}\circ\phi]{} & \mathcal{M} \end{array}$$

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 $\mathrm{d}\phi_u^{-1} = \mathrm{dexp}_u{}^{-1}$ 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, 33] for different choices. This yields the following algorithm:

Algorithm A.2.2 (General Munthe-Kaas methods)
Given a differential equation in the form (A.1), this algorithm produces a qth 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 $\mathfrak{g}$*

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

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

When $\phi = \exp$, this algorithm reduces to the Munthe-Kaas methods presented in [27].

Algorithm A.2.3 (Munthe-Kaas methods based on the exponential mapping)
Let $A = (a_{ij})$, $b = (b_j)$ and $c = (c_j)$ be the coefficients of an s-stage, qth order classical Runge–Kutta method. The following algorithm integrates (A.1) from $t = t_n$ to $t = t_n + h$:

> Assume that $y_n \approx y(t_n)$ is available
> for $i = 1, 2, \dots, s$
> $u_i = h\sum_{j=1}^{s} a_{ij}\tilde{k}_j$
> $k_i = \xi\left(t_n + c_i h, \Phi\big(\exp(u_i), y_n\big)\right)$
> $\tilde{k}_i = \mathrm{dexpinv}(u_i, k_i, q)$
> end
> $v = h\sum_{j=1}^{s} b_j\tilde{k}_j$
> $y_{n+1} = \Phi\big(\exp(v), y_n\big)$

Here, a qth order truncation of the dexpinv function is defined by

$$\mathrm{dexpinv}(u, v, q) = v - \frac{1}{2}[u, v] + \sum_{k=2}^{q-1}\frac{B_k}{k!}\overbrace{[u, [u, [\dots, [}^{k}u, v]]],$$

where $[\cdot,\cdot]$ is the matrix commutator defined by $[A, B] = AB - BA$ when A and B are matrices, and B_k is the kth Bernoulli number.

This method has at least order q on any manifold. In Eucliden space this algorithm reduces to the classical Runge–Kutta algorithm.

A.2.3 Magnus series methods

The solution of Lie type ordinary differential matrix equations,

$$y' = \xi_{\mathcal{M}}(t), \qquad y(0) = p \in \mathcal{M}, \tag{A.2}$$

where $\xi : \mathbb{R}^+ \to \mathfrak{g}$ for every $t \geq 0$, and $\mathcal{M} = G$ is a matrix Lie group, was recently studied by Iserles and Nørsett [16]. These methods can be viewed as a particular case of the Munthe-Kaas methods. The following commutative diagram shows the framework:

$$\begin{array}{ccc} \mathrm{T}\mathfrak{g} & \xrightarrow{\mathrm{T}\Phi_{y_0}\circ \mathrm{T}\exp} & \mathrm{T}\mathcal{M} \\ {\scriptstyle \mathrm{dexp}_u{}^{-1}\circ\xi\circ\Phi_{y_0}\circ\exp}\uparrow & & \uparrow{\scriptstyle F=\xi_{\mathcal{M}}} \\ \mathfrak{g} & \xrightarrow[\Phi_{y_0}\circ\exp]{} & \mathcal{M} \end{array}$$

Again, $y_0 \in \mathcal{M}$ is the initial point, $u \in \mathfrak{g}$, $\Phi_{y_0}(\exp(u)) = \Phi(\exp(u), y_0)$ and $\mathrm{dexp}_u^{-1} : \mathfrak{g} \to \mathfrak{g}$ is the right trivialized tangent of exp. By applying an implicit Runge–Kutta method to the ordinary differential equation on $\mathfrak{g}$ and, since $\xi = \xi(t)$, performing Picard iteration, an infinite sum of integrals appears. Magnus [17] observed that $u(t)$ can be expressed as an infinite sum of elements in $\mathfrak{g}$:

$$\begin{aligned} u(t) \quad = \quad & \int_0^t \xi(\kappa)\mathrm{d}\kappa + \tfrac{1}{2}\int_0^t \left[\xi(\kappa), \int_0^\kappa \xi(\alpha)\mathrm{d}\alpha\right]\mathrm{d}\kappa \\ & +\tfrac{1}{4}\int_0^t \left[\xi(\kappa), \int_0^\kappa \left[\xi(\alpha), \int_0^\alpha \xi(\eta)\mathrm{d}\eta\right]\mathrm{d}\alpha\right]\mathrm{d}\kappa \\ & +\tfrac{1}{12}\int_0^t \left[\left[\xi(\kappa), \int_0^\kappa \xi(\alpha)\mathrm{d}\alpha\right], \int_0^\kappa \xi(\eta)\mathrm{d}\eta\right]\mathrm{d}\kappa + \cdots . \end{aligned} \tag{A.3}$$

Iserles and Nørsett analysed this sum and presented the devices necessary to make it a useful numerical method – or rather a whole family of methods – named *Magnus series methods*. Implementation issues and error control of these numerical methods were discussed in [15].

When u is approximated by $\widetilde{u}$, say, the solution is advanced according to

$$y_{n+1} = \Phi(\exp(\widetilde{u}), y_n).$$

Algorithm A.2.4 (A fourth-order Magnus series method)
Let c_1 and c_2 be the Gauss–Legendre points

$$c_1 = \tfrac{1}{2} - \tfrac{\sqrt{3}}{6} \qquad \textit{and} \qquad c_2 = \tfrac{1}{2} + \tfrac{\sqrt{3}}{6},$$

and evaluate ξ at these abscissae values:

$$\xi_1 = \xi(t_n + c_1 h) \qquad \textit{and} \qquad \xi_2 = \xi(t_n + c_2 h).$$

A fourth-order approximation to u is then given as

$$u^{[4]} = \tfrac{1}{2}h(\xi_1 + \xi_2) - \tfrac{\sqrt{3}}{12}h^2[\xi_1, \xi_2].$$

The solution is advanced according to

$$y_{n+1} = \Phi(\exp(u^{[4]}), y_n).$$

Algorithm A.2.5 (A sixth-order Magnus series method)
Let c_1, c_2 and c_3 be the Gauss–Legendre points

$$c_1 = \tfrac{1}{2} - \tfrac{\sqrt{15}}{10}, \qquad c_2 = \tfrac{1}{2}, \qquad \textit{and} \qquad c_3 = \tfrac{1}{2} + \tfrac{\sqrt{15}}{10},$$

and evaluate ξ at these abscissae values:

$$\xi_1 = \xi(t_n + c_1 h), \qquad \xi_2 = \xi(t_n + c_2 h) \qquad \textit{and} \qquad \xi_3 = \xi(t_n + c_3 h).$$

A sixth-order approximation to u is then given by the following equations:

$$\begin{aligned}
K_1 &= \tfrac{h}{18}(5\xi_1 + 8\xi_2 + 5\xi_3)\\
K_2 &= -\tfrac{\sqrt{15}}{108} h^2 (2\xi_{12} + \xi_{13} + 2\xi_{23})\\
K_3 &= \tfrac{h^3}{432}([\xi_1 - 5\xi_3, \xi_{12}] + [5\xi_1 - \xi_3, \xi_{23}])\\
K_4 &= \tfrac{\sqrt{15}}{2160} h^4 [\xi_1, [\xi_3, \xi_{13}]]\\
u^{[6]} &= K_1 + K_2 + K_3 + K_4,
\end{aligned}$$

where $\xi_{ij} = [\xi_i, \xi_j]$. The solution is advanced according to

$$y_{n+1} = \phi(\exp(u^{[6]}), y_n).$$

The generalized Magnus series methods that solve $y' = \xi_G(t, y)$, $y(0) = y_0 \in G$, are implemented according to the algorithms presented in [39, 40]. These methods are based on the collocation idea.

Assume that approximations $\xi_i \approx \xi\big(t_n + c_i h, y(t_n + c_i h)\big)$, $i = 1, \ldots, s$, are known. Based in these values we approximate the function ξ, in the vector space $\mathfrak{g}$, around $t = t_n$ with its Lagrangian interpolating polynomial at the abscissae values $c_1, \ldots, c_s$:

$$\xi(t, y(t)) \approx \sum_{i=1}^{s} L_i\Big(\frac{t - t_n}{h}\Big)\xi_i, \tag{A.4}$$

where

$$L_i(t) = \prod_{\substack{k=1\\k\neq i}}^{s} \frac{t - c_k}{c_i - c_k}.$$

By inserting the interpolating polynomial into the Magnus series, a method solving the general problem is obtained. As in the classical case, the collocation methods are necessarily implicit. However, it is possible to derive relaxed collocation schemes that are explicit.

The two collocation based Magnus methods in *DiffMan* 1.6 are **MRC3** and **MC4**:

MRC3: *(Magnus series method of order 3 based on relaxed collocation)*

$$\begin{aligned}
Y_1 &= y_n, & \xi_1 &= \xi(t_n, Y_1),\\
Y_2 &= \Phi(\exp\big(\tfrac{h}{2}\xi_1\big), y_n), & \xi_2 &= \xi\big(t_n + \tfrac{h}{2}, Y_2\big),\\
Y_3 &= \Phi(\exp\big(h(-\xi_1 + 2\xi_2)\big), y_n), & \xi_3 &= \xi\big(t_n + h, Y_3\big),
\end{aligned}$$

with

$$y_{n+1} = \Phi\Big(\exp\big(h(\tfrac{1}{6}\xi_1 + \tfrac{2}{3}\xi_2 + \tfrac{1}{6}\xi_3) - \tfrac{h^2}{2}[\xi_1 - \xi_3, \tfrac{2}{15}\xi_2 + \tfrac{1}{30}\xi_3]\big), y_n\Big).$$

MC4: *(Magnus series method of order 4 based on collocation)*
Solve the following system

$$Y_1 = \Phi\Big(\exp\big(h(\tfrac{1}{4}\xi_1 + (\tfrac{1}{4} - \tfrac{\sqrt{3}}{6})\xi_2) + (\tfrac{5}{72} - \tfrac{\sqrt{3}}{24})h^2\xi_{12}\big), y_n\Big),$$
$$Y_2 = \Phi\Big(\exp\big(h((\tfrac{1}{4} + \tfrac{\sqrt{3}}{6})\xi_1 + \tfrac{1}{4}\xi_2) + (\tfrac{5}{72} + \tfrac{\sqrt{3}}{24})h^2\xi_{12}\big), y_n\Big),$$

where $\xi_i = \xi(t_n + c_i h, Y_i)$, $i = 1, 2$, and $\xi_{12} = [\xi_1, \xi_2]$. Advance the solution by computing

$$y_{n+1} = \Phi\Big(\exp\big(\tfrac{h}{2}(\xi_1 + \xi_2) - \tfrac{\sqrt{3}}{12}h^2[\xi_1, \xi_2]\big), y_n\Big).$$

A.2.4 Quadrature methods on quadratic Lie groups

The quadrature methods based on the Cayley transform are similar to the Magnus series methods. They are discussed in [22]. We again consider the solution of Lie type ordinary differential matrix equations,

$$y' = \xi_{\mathcal{M}}(t), \qquad y(0) = p \in \mathcal{M},$$

where $\xi : \mathbb{R}^+ \to \mathfrak{g}$ for every $t \geq 0$. For any matrix $v \in \mathbb{R}^{n\times n}$ a subalgebra of $\mathfrak{gl}(n)$ is

$$\mathfrak{g}_v = \left\{u \in \mathfrak{gl}(n) : vu + u^T v = 0\right\}.$$

Such Lie algebras are called quadratic Lie algebras. Roughly speaking, the Lie groups, G, obtained by exponentiation of these Lie algebras are called quadratic Lie groups. We immediately see that by choosing v to be

$$\mathrm{I}_n \qquad \text{or} \qquad \begin{bmatrix} 0 & \mathrm{I}_{n/2} \\ -\mathrm{I}_{n/2} & 0 \end{bmatrix} \text{ (when } n \text{ is even)},$$

we recover the Lie algebras $\mathfrak{so}(n)$ and $\mathfrak{sp}(n)$, respectively. It is well known that the computation of matrix exponentials in general is a costly operation. When constructing numerical integrators for differential equations evolving on quadratic Lie groups, we exploit the fact that there exist other mappings between the Lie algebra and the Lie group than the exponential mapping. Celledoni and Iserles [3] showed that if ϕ is any analytic function that satisfies $\phi(z)\cdot\phi(-z) = 1$, then $\phi(\mathfrak{g}_v) \subset G$.

There exist a large number of such functions, but we shall focus on the Cayley transform

$$\mathrm{cay}(z) = (1 - \tfrac{z}{2})^{-1}(1 + \tfrac{z}{2}).$$

The following commutative diagram shows the framework for the methods:

$$\begin{tikzcd} \mathrm{T}\mathfrak{g} \arrow[r, "\mathrm{T}\Phi_{y_0}\circ \mathrm{T}\mathrm{cay}"] & \mathrm{T}\mathcal{M} \\ \mathfrak{g} \arrow[u, "\mathrm{dcay}_u^{-1}\circ\xi\circ\Phi_{y_0}\circ\mathrm{cay}"] \arrow[r, "\Phi_{y_0}\circ\mathrm{cay}"'] & \mathcal{M} \arrow[u, "F=\xi_{\mathcal{M}}"'] \end{tikzcd}$$

Again, $y_0 \in \mathcal{M}$ is the initial point, $u \in \mathfrak{g}$, $\Phi_{y_0}(\mathrm{cay}(u)) = \Phi(\mathrm{cay}(u), y_0)$ and $\mathrm{dcay}_u^{-1} : \mathfrak{g} \to \mathfrak{g}$ is the right trivialized tangent of cay. By applying an implicit Runge–Kutta method to the ordinary differential equation on $\mathfrak{g}$ and, since $\xi = \xi(t)$, performing Picard iteration, an infinite sum appears. The numerical methods below are based on truncations of this sum.

The two timesteppers of the above kind implemented in *DiffMan* 1.6 are **qq4a** and **qq6a**:

qq4a: *(quadrature method of order four on quadratic Lie groups)*
Consider the Gauss-Legendre weights of order 4: $c_1 = \frac{1}{2} - \alpha$ and $c_2 = \frac{1}{2} + \alpha$ with $\alpha = \frac{\sqrt{3}}{6}$. Let ξ_i^n be the function ξ evaluated at the abscissae values, $\xi_i^n = \xi(t_n + c_i h)$, $i = 1, 2$. The resulting fourth order quadrature method based on the Cayley transform is then given by

$$\begin{aligned} u_4^n &= \tfrac{1}{2}h(\xi_1^n + \xi_2^n) - \tfrac{1}{4\sqrt{3}}h^2[\xi_1^n, \xi_2^n] - \tfrac{1}{96}h^3(\xi_1^n + \xi_2^n)^3 \\ y_{n+1} &= \Phi(\mathrm{cay}(u_4^n), y_n). \end{aligned}$$

qq6a: *(quadrature method of order six on quadratic Lie groups)*
Consider the Gauss-Legendre weights of order 6: $c_1 = \frac{1}{2} - \alpha$, $c_2 = \frac{1}{2}$ and $c_3 = \frac{1}{2} + \alpha$ with $\alpha = \frac{\sqrt{15}}{10}$, and let again $\xi_i^n = \xi(t_n + c_i h)$, $i = 1, \ldots, 3$, be the function ξ evaluated at the abscissae values. Let ω_1^n, ω_2^n and ω_3^n be defined as follows:

$$\xi_1^n = \omega_1^n - \alpha h \omega_2^n + (\alpha h)^2 \omega_3^n, \qquad \xi_2^n = \omega_1^n \qquad \text{and} \qquad \xi_3^n = \omega_1^n + \alpha h \omega_2^n + (\alpha h)^2 \omega_3^n.$$

The resulting sixth order quadrature method based on the Cayley transform is then given by

$$\begin{aligned} u_6^n &= h\omega_1 + \tfrac{1}{12}h^3(\omega_3 - [\omega_1, \omega_2] - \omega_1^3) \\ &\quad + h^5\Big(\tfrac{1}{240}([\omega_2, \omega_3] - [\omega_2, [\omega_1, \omega_2]] - \omega_1^2\omega_3 - \omega_3\omega_1^2 + [\omega_1\omega_2\omega_1, \omega_1]) \\ &\quad - \tfrac{1}{80}\omega_1\omega_3\omega_1 + \tfrac{1}{12}([\omega_1^3, \omega_2] + \omega_1^5)\Big) \\ y_{n+1} &= \Phi(\mathrm{cay}(u_6^n), y_n). \end{aligned}$$

A.2.5 Fer expansion methods

The Fer expansion was introduced by Fer [9] as a method of solving matrix differential equations in Euclidean space of the form

$$y' = f(t)y, \qquad y(0) = y_0. \tag{A.5}$$

Iserles [14] proposed numerical methods based on a similar construction. They both showed that the solution of (A.5) can be expressed as an infinite product of exponentials:

$$y(t) = \lim_{n\to\infty} \exp\big(B_0(t)\big)\exp\big(B_1(t)\big)\cdots\exp\big(B_n(t)\big)y_0, \tag{A.6}$$

in a neighborhood of 0. The B_i are matrix valued functions, and they can be computed iteratively. By letting $A_0(t) = f(t)$, we obtain $B_0(t) = \int_0^t A_0(\tau)\,\mathrm{d}\tau$. Furthermore, we construct $C_{0,k}$ as $C_{0,0}(t) = A_0(t)$ and $C_{0,i+1}(t) = [C_{0,i}(t), B_0(t)]$, $i = 0, 1, \ldots$. Here, $[\cdot,\cdot]$ denotes the matrix commutator. $A_{k+1}(t)$ and $B_{k+1}(t)$ are now computed as follows:

$$A_{k+1}(t) = \sum_{i=1}^{\infty} \frac{i}{(i+1)!} C_{k,i}(t),$$

where $C_{k,0}(t) = A_k(t)$ and $C_{k,i+1}(t) = [C_{k,i}(t), B_k(t)]$, $i = 0, 1, \ldots$, with $B_{k+1}(t) = \int_0^t A_{k+1}(\tau)\,\mathrm{d}\tau$.

In *DiffMan*, we view the differential equation in a generalized setting:

$$y' = \xi_{\mathcal{M}}(t), \qquad y(0) = y_0 \in \mathcal{M}. \tag{A.7}$$

The numerical procedure generates at each step an element $g_n \in G$ which we use to advance the numerical solution from y_n to y_{n+1}:

$$y_{n+1} = \Phi(g_n, y_n)$$

This procedure is only valid when ξ is a function of t only. It is, however, possible to extend this algorithm to also cope with y-dependent functions. This has been analysed in [39, 40].

The Fer expansion methods integrating problem (A.7) are implemented according to the algorithms presented in [14]. The implementation of the algorithm is not straight-forward. We have, therefore, only implemented in *DiffMan* methods up to order 6 with $n \leq 2$ in (A.6). The algorithm is as follows (see also [14]):

Let $A_0(t) = \xi(t)$. Assume we are given the coefficients $c_i^{(j)}$ and $w_i^{(j)}$, $i = 1, \ldots, s$ and $j = 1, \ldots, m$. Set $c_0^{(2)} = 0$ and $B_0(t_0, 0) = 0$. Evaluate, for every $1 \leq i \leq s$,

$$\begin{aligned} B_0\big(t_0, c_i^{(2)}h\big) &= B_0\big(t_0, c_{i-1}^{(2)}h\big) \\ &\quad + \big(c_i^{(2)} - c_{i-1}^{(2)}\big)h \sum_{j=1}^{s} w_j^{(1)} A_0\Big(t_0 + \big(c_{i-1}^{(2)} + (c_i^{(2)} - c_{i-1}^{(2)})c_j^{(1)}\big)h\Big) \end{aligned}$$

and

$$\begin{aligned} C_{0,0}\big(t_0 + c_i^{(2)}h\big) &= A_0\big(t_0 + c_i^{(2)}\big) \\ C_{0,k}\big(t_0 + c_i^{(2)}h\big) &= \big[C_{0,k-1}\big(t_0 + c_i^{(2)}h\big), B_0\big(t_0, c_i^{(2)}h\big)\big], \qquad 1 \leq k \leq p-1. \end{aligned}$$

Then

$$A_1\big(t_0 + c_i^{(2)}h\big) = \sum_{k=1}^{p-1} \frac{k}{(k+1)!} C_{0,k}\big(t_0 + c_i^{(2)}h\big),$$

and

$$B_1(t_0, h) = h \sum_{i=1}^{s} w_i^{(2)} A_1\big(t_0 + c_i^{(2)}h\big).$$

The generalized Fer expansion methods that solve $y' = \xi_{\mathcal{M}}(t, y)$, $y(0) = y_0 \in \mathcal{M}$, are implemented according to the algorithms presented in [39, 40]. These methods are based on the collocation idea. Again, the general implementation procedure is not straight-forward, so we have only implemented a few particular methods.

Assume that approximations $\xi_i \approx \xi\big(t_n + c_i h, y(t_n + c_i h)\big)$, $i = 1, \ldots, s$, are known. Based in these values we approximate the function ξ around $t = t_n$, with its Lagrangian interpolating polynomial at the abscissae values $c_1, \ldots, c_s$ as in (A.4). By inserting the interpolating polynomial into the Fer expansion, a method solving the general problem is obtained. As in the classical case, the collocation methods are necessarily implicit. However, it is possible to derive relaxed collocation schemes that are explicit.

The two collocation based Fer methods in *DiffMan* 1.6 are **FRC3** and **FC4**:

FRC3: *(Fer expansion method of order 3 based on relaxed collocation)*

$$\begin{aligned} Y_1 &= y_n, & \xi_1 &= \xi(t_n, Y_1), \\ Y_2 &= \Phi\big(\exp\big(\tfrac{h}{2}\xi_1\big), y_n\big), & \xi_2 &= \xi\big(t_n + \tfrac{h}{2}, Y_2\big), \\ Y_3 &= \Phi\big(\exp\big(h(-\xi_1 + 2\xi_2)\big), y_n\big), & \xi_3 &= \xi\big(t_n + h, Y_3\big), \end{aligned}$$

with

$$y_{n+1} = \Phi\big(\exp\big(h\big(\tfrac{1}{6}\xi_1 + \tfrac{2}{3}\xi_2 + \tfrac{1}{6}\xi_3\big)\big)\exp\big(-\tfrac{h^2}{2}\big[\xi_1 - \xi_3, \tfrac{2}{15}\xi_2 + \tfrac{1}{30}\xi_3\big]\big), y_n\big).$$

FC4: *(Fer expansion method of order 4 based on collocation)*
Solve the following system

$$
\begin{aligned}
&\xi_{12} = [\xi_1, \xi_2], \quad \xi_{13} = [\xi_1, \xi_3], \quad \xi_{23} = [\xi_2, \xi_3] \\
&Y_1 = y_n \\
&K_1 = \tfrac{5}{24}\xi_1 + \tfrac{1}{3}\xi_2 - \tfrac{1}{24}\xi_3, \quad K_2 = -\tfrac{11}{240}\xi_{12} + \tfrac{1}{240}\xi_{13} - \tfrac{1}{240}\xi_{23} \\
&Y_2 = \Phi(\exp(hK_1)\exp\big(\tfrac{h^2}{2}K_2\big), y_n) \\
&K_1 = \tfrac{1}{6}\xi_1 + \tfrac{2}{3}\xi_2 + \tfrac{1}{6}\xi_3, \quad K_2 = -\tfrac{2}{15}\xi_{12} - \tfrac{1}{30}\xi_{13} - \tfrac{2}{15}\xi_{23} \\
&Y_3 = \Phi(\exp(hK_1)\exp\big(\tfrac{h^2}{2}K_2\big), y_n),
\end{aligned}
$$

where $\xi_i = \xi(t_n + c_i h, Y_i)$, $i = 1, 2, 3$. Advance the solution by computing

$$
\begin{aligned}
&U_1 = h\big(\tfrac{1}{6}\xi_1 + \tfrac{2}{3}\xi_2 + \tfrac{1}{6}\xi_3\big) \\
&K_{12} = [\xi_1, \xi_2], \quad K_{13} = [\xi_1, \xi_3], \quad K_{23} = [\xi_2, \xi_3] \\
&U_2 = \tfrac{h^2}{2}\big(-\tfrac{2}{15}K_{12} - \tfrac{1}{30}K_{13} - \tfrac{2}{15}K_{23}\big) \\
&\widetilde{Y}_1 = -\tfrac{8}{315}\xi_1 - \tfrac{17}{315}\xi_2 - \tfrac{17}{5670}\xi_3, \qquad K_1 = [K_{12}, \widetilde{Y}_1], \\
&\widetilde{Y}_2 = -\tfrac{1}{168}\xi_1 - \tfrac{1}{45}\xi_2 + \tfrac{13}{5670}\xi_3, \qquad K_2 = [K_{13}, \widetilde{Y}_2], \\
&\widetilde{Y}_3 = -\tfrac{1}{42}\xi_1 - \tfrac{5}{63}\xi_2 - \tfrac{1}{126}\xi_3, \qquad K_3 = [K_{23}, \widetilde{Y}_3] \\
&U_3 = \tfrac{h^3}{3}(K_1 + K_2 + K_3),
\end{aligned}
$$

and finally,

$$y_{n+1} = \Phi(\exp(U_1)\exp(U_2 + U_3), y_n).$$

A.2.6 Crouch–Grossman methods

The Crouch–Grossman methods were first described in [4], and a general theory describing the order conditions was presented in [34].

Assume that there exists a frame on the manifold $\mathcal{M}$, i.e. a set of vector fields $E_1, \ldots, E_d$ on $\mathcal{M}$, which at each point $p \in \mathcal{M}$ span the tangent space $\mathrm{T}\mathcal{M}|_p$. A differential equation on $\mathcal{M}$ can be written in terms of this frame as

$$y' = F_y(y) = \sum_{i=1}^{d} f_i(y)E_i, \qquad \text{where } f_i : \mathcal{M} \to \mathbb{R} \text{ are smooth functions.} \tag{A.8}$$

Let F_p denote the vector field with coefficients frozen at p relative to the frame:

$$F_p(y) = \sum_{i=1}^{d} f_i(p)E_i.$$

Let $\mathfrak{g}$ be the Lie algebra generated by the frame $E_1, \ldots, E_d$ and let $G \subset \mathrm{Diff}(\mathcal{M})$ be the collection of flows on $\mathcal{M}$ generated by exponentiating $\mathfrak{g}$. Furthermore, let $\lambda : \mathfrak{g} \times \mathcal{M} \to \mathcal{M}$ be the flow operator, i.e. $y(t) = \lambda(tF, q)$ is the solution of $y' = F(y)$ with $y(0) = q$. Since

$$\left.\frac{\mathrm{d}}{\mathrm{d}t}\right|_{t=0} \lambda(tF_y, q) = F_y(y(t))\big|_{t=0} = F_y(q),$$

it follows that (A.8) is an equation of the form (A.1).

Algorithm A.2.6 (Crouch–Grossman methods)
Let $A = (a_{ij})$, $b = (b_j)$ and $c = (c_j)$ be the coefficients of an s-stage, qth order Crouch–Grossman method (see e.g. [34]). The following algorithm integrates (A.8) from $t = t_n$ to $t = t_n + h$:

Assume that $y_n \approx y(t_n)$ is available

for $i = 1, 2, \dots, s$
  $Y_i = y_n$
  for $j = 1, 2, \dots, s$
    $Y_i = \lambda(ha_{ij} F_{Y_j}, Y_i)$
  end
end

$y_{n+1} = y_n$
for $i = 1, 2, \dots, s$
  $y_{n+1} = \lambda(hb_i F_{Y_i}, y_{n+1})$
end

The flow operator λ is given as $\lambda(v, p) = \Phi\big(\exp(v), p\big)$. In Eucliden space this algorithm reduces to the classical Runge–Kutta algorithm.

Appendix B

Available time stepper schemes

The schemes available in *DiffMan* 1.6 are

```
Number   String      - Description of method
  1      'E1'        - explicit Euler, order 1
  2      'ME2'       - modified Euler, order 2
  3      'heun2'     - Heun's method, order 2
  4      'ode23'     - the method implemented in the MATLAB
                       ode23 routine
  5      'M4a'       - a Magnus method, order 4
  6      'M6a'       - a Magnus method, order 6
  7      'moan25'    - Moan's method, order 2(5)
  8      'moan35'    - Moan's method, order 3(5)
  9      'CG23'      - Crouch-Grossman method, order 2(3)
 10      'CG3a'      - Crouch-Grossman method, order 3
 11      'CG43'      - Crouch-Grossman method, order 4(3)
 12      'CG34'      - Crouch-Grossman method, order 3(4)
 13      'RK4'       - "The" Runge-Kutta method, order 4
 14      'rk45'      - Runge-Kutta method, order 4(5)
 15      'RKF34'     - Fehlberg's method of order 3(4)
 16      'RKF43'     - Fehlberg's method of order 4(3)
 17      'RKF45a'    - Fehlberg's method of order 4(5) (a)
 18      'RKF54a'    - Fehlberg's method of order 5(4) (a)
 19      'RKF45b'    - Fehlberg's method of order 4(5) (b)
 20      'RKF54b'    - Fehlberg's method of order 5(4) (b)
 21      'dopri45'   - Dormand and Prince's method of order 4(5)
 22      'dopri54'   - Dormand and Prince's method of order 5(4)
 23      'dopri5'    - Dormand and Prince's method of order 5
 24      'butcher6'  - Butcher's method, order 6
 25      'RKF78'     - Fehlberg's method of order 7(8)
 26      'RKF87'     - Fehlberg's method of order 8(7)
 27      'dopri78'   - Dormand and Prince's method of order 7(8)
 28      'dopri87'   - Dormand and Prince's method of order 8(7)
 29      'fer2a'     - Fer method of order 2
 30      'fer4G2'    - Fer method of order 4 based on Gaussian
                       quadrature
 31      'fer5GR'    - Fer method of order 5 based on
                       Gauss-Radau quadrature
```

```
32      'fer6G3'    - Fer method of order 6 based on
                      Gaussian quadrature
33      'fer6GLR'   - Fer method of order 6 based on
                      Gauss-Legendre and Gauss-Jacobi-
                      Radau quadrature
34      'FRC3'      - Fer method of order 3 based on
                      relaxed collocation
35      'MRC3'      - Magnus method of order 3 based on
                      relaxed collocation
36      'qq4a'      - Quadrature method for quadratic Lie
                      groups. Order 4.
37      'qq6a'      - Quadrature method for quadratic Lie
                      groups. Order 6.
38      'IE1'       - Implicit Euler, order 1
39      'GL2' 'IM2' - Gauss-Legendre/implicit midpoint, order 2
40      'GL4'       - Gauss-Legendre, order 4
41      'GL6'       - Gauss-Legendre, order 6
42      'FC4'       - Fer method of order 4 based on collocation
43      'MC4'       - Magnus method of order 4 based on collocation
44      'TRAP2'     - Trapozoidal rule of 2nd order.
45      'GL4S'      - Time symmetric GL method of order 4
46      'GL6S'      - Time symmetric GL method of order 6
```

You can refer to either the number of the method or to its abbreviated name. The following fields are returned:

```
.RKname - name of the method
.RKa    - matrix A from the method's Butcher tableau
.RKb    - vector b from the method's Butcher tableau
.RKc    - vector c from the method's Butcher tableau or the
          abscissae values in case of Magnus methods
.RKns   - number of stages in the method
.RKord  - order of the method
.RKtype - type of the method ('explicit', 'implicit', 'SDIRK')
.RKbhat - vector bhat from the method's Butcher tableau
          (used in error estimation)
```

You may add new schemes in the file:

`DiffMan/flow/timestepper/@timestepper/setmethod.m`

Appendix C

Summary of virtual superclass functions

C.1 Functions in @liealgebra

```
basis(a,i)
    Returns the i'th basis vector in the Lie algebra.
dcay(a,b,ord)
    The differential of the Cayley transform.
dcaycay(x,z,ord)
    The differential of the Cayley coordinates of the second kind.
dcaycayinv(x,z,ord)
    Inverse of the differential of Cayley coord. of the second kind.
dcayinv(a,b,ord)
    The inverse of the differential of the Cayley transform.
dexp(a,b,ord)
    The ord'th order approximation of the differential of exp.
dexpexp(x,z,ord)
    The differential of canonical coordinates of the second kind.
dexpexpinv(x,z,ord)
    Inverse of differential of canonical coord. of the second kind.
dexpinv(a,b,ord)
    The ord'th order approximation of the inverse differential of exp.
dexpinvtest(a,b,ord)
    The ord'th order approximation of the inverse differential of exp.
dimension(a)
    Returns the dimension of the Lie algebra vectorspace.
display(obj)
    Display a LIEALGEBRA object, or objects from a daugther class.
dist(a,b)
    Distance metric function on the Lie algebra.
dpade22(x,z,ord)
    Differential of the (2,2) Pade' approximation of exponential map.
dpade22inv(x,z,ord)
    Inverse differential of the (2,2) Pade' approximation.
eig(a)
    Overloaded version of the MATLAB built-in eigenvalue function.
getdata(g)
    Returns the data that represents the element g in the Lie algebra.
```

```
getmatrix(g)
    Returns the matrix representation of g in the Lie algebra.
getshape(g)
    Returns shape information if the Lie algebra is dynamically subtyped.
getvector(g)
    Returns a column vector representing g in the Lie algebra.
hasmatrix(g)
    Checks if the Lie algebra has a matrix representation.
hasshape(g)
    Checks if the Lie algebra has dynamic shape information.
horzcat(a,b)
    Commutator in Lie algebra.
isabelian(a)
    Checks whether or not a Lie algebra is Abelian.
isdata(a,m)
    Checks if m could be data for an element in Lie algebra.
ismatrix(a,m)
    Checks if m is a possible matrix representation in the Lie algebra.
liealgebra(varargin)
    Constructor for virtual superclass of all liealgebras.
liegroup(a)
    Picks out the liegroup corresponding to the liealgebra.
minus(u,v)
    Vector subtraction in Lie algebra.
mtimes(u,v)
    Scalar multiplication in Lie algebra.
norm(a,alt)
    Overloaded version of the MATLAB built-in norm function.
plus(u,v)
    Vector addition in Lie algebra.
project(a,m)
    Returns a matrix v which is acceptable by the Lie algebra.
random(lalg)
    Creates a random object in the Lie algebra.
sameshape(a,b)
    Chechs if a and b belong to the same Lie algebra.
setdata(a,m)
    Sets m to be the data of a.
setmatrix(a,m)
    Sets m to be the matrix data of a.
setshape(a,sh)
    Sets shape information in g.
setvector(a,v)
    Sets v to be the vector data of a.
uminus(u)
    Unary minus in Lie algebra.
zero(lalg)
    Create the zero object in a Lie algebra.
zeros(obj,sz)
    Creating an array of objects initalized to zero(obj).
```

C.2 Functions in @liegroup

```
cay(lgr,a)
    Computes the Cayley transform from Lie algebra to Lie group.
caycay(lgr,a)
    Computes Cayley coordinates of the second kind.
display(obj)
    Display a LIEGROUP object, or objects from a daugther class.
dist(a,b)
    Distance metric function on the Lie group.
eig(a)
    Overloaded version of the MATLAB built-in eigenvalue function.
exp(lgr,a)
    Computes the exponential from the Lie algebra to the Lie group.
expexp(lgr,a)
    Computes canonical coordinates of the second kind.
getdata(g)
    Returns data that represents the element g in the Lie group.
getmatrix(g)
    Returns matrix data that represents the element in the Lie group.
getshape(g)
    Returns shape information if the Lie group is dynamically subtyped.
hasmatrix(g)
    Returns if the Lie group has a matrix representation.
hasshape(g)
    Checks if the Lie group has dynamic shape information.
identity(a)
    Returns the identity object in the Lie group of a.
inv(a)
    The invers of an element in the Lie group.
invcay(a)
    Computes the inverse Cayley coordinates of the first kind.
invcaycay(a)
    Computes the inverse Cayley coordinates of the second kind.
invexpexp(a)
    Computes inverse Canonical coordinates of the second kind.
invpade22(a)
    The inverse (2,2) diagonal Pade' coordinates of first kind.
isabelian(a)
    Returns if the Lie group is Abelian or not.
isdata(a,m)
    Checks if m is data representation for the Lie group.
ismatrix(a,m)
    Checks if m is a matrix representation for the Lie group.
liealgebra(a)
    Picks out the liealgebra corresponding to the lie group.
liegroup(varargin)
    Constructor for LIEGROUP objects.
log(a)
    Computes the logarithm from the Lie group to the Lie algebra.
mtimes(a,b)
    The binary operation of two elements in the Lie group.
norm(a,alt)
    Overloaded version of the MATLAB built-in norm function.
pade22(lgr,a)
    Computes the (2,2) Pade' approximation of the exponential.
```

```
project(a,m)
    Returns a matrix v acceptable by the Lie group.
random(a)
    Creates a random object in the Lie group.
sameshape(a,b)
    Chechs if a and b belong to the same Lie group.
setdata(a,m)
    Sets m to be the data repr. of a.
setmatrix(a,m)
    Sets m to be the matrix repr. of a.
setshape(a,sh)
    Sets shape information in g.
```

C.3 Functions in @hmanifold

```
display(obj)
    Display a HMANIFOLD object, or objects from daugther classes.
dist(a,b)
    Distance metric function on the homogeneous manifold.
getdata(g)
    Returns data representation of element in homogeneous space.
getshape(g)
    Returns shape information of the homogeneous space.
hasshape(g)
    Checks if the homogeneous space has dynamic shape information.
hmanifold(varargin)
    Constructor for HMANIFOLD-objects.
invlambda(p,q)
    Returns an object v in the Lie algebra such that lambda(v,p) = q.
isdata(a,m)
    Checks if m is a data representation for the homogeneous space.
lambda(a,m,coord)
    The action of the Lie algebra on the manifold.
liealgebra(a)
    Picks out the liealgebra of the homogeneous space.
origin(a)
    Returns the origin in the homogeneous space.
project(a,m)
    Projects to a matrix acceptable in the homogeneous space.
random(a)
    Creates a random object in the homogeneous space.
sameshape(a,b)
    Checks if input has the same shape information.
setdata(a,m)
    Sets the data representation of a homogenous space object.
setshape(a,sh)
    Sets the shape information in a homogeneous space object.
stabilizer(a)
    Returns a matrix spanning the stabilizer subalgebra.
zeros(obj,sz)
    Creates an array of objects initalized to the origin object.
```

C.4 Functions in `@vectorfield`

```
display(obj)
    Display a VECTORFIELD object.
getdomain(f)
    Returns object in domain over which the vector field is defined.
geteqntype(f)
    Returns the type of the equation.
getfm2g(f)
    Returns the map describing the vector field.
setdomain(vf,dom)
    Sets dom to be the domain of the vector field vf.
seteqntype(vf,type)
    Sets the equation-type-field of vf equal to 'type'.
setfm2g(vf,map)
    Sets the map describing the differential equation.
vectorfield(varargin)
    Constructor for the vector field class.
```

C.5 Functions in `@flow`

```
display(obj)
    Display a FLOW object.
flow(varargin)
    Constructor for the flow class.
getdefaults(f)
    Returns the default values of the flow object.
gettimestepper(f)
    Returns the timestepper used by the flow.
getvectorfield(f)
    Returns the vector field defining the flow.
newstepsize(fl,varargin)
    Computes a new step size to be used by time stepper objects.
setdefaults(f,varargin)
    Sets the defaults of the flow f to ts.
settimestepper(f,ts)
    Sets the timestepper of the flow f to ts.
setvectorfield(f,vf)
    Sets the vector field of the flow f to vf.
subsref(f,s)
    Overloads the parenthesis for flow objects.
```

C.6 Functions in `@timestepper`

```
display(obj)
    Display a TIMESTEPPER object.
getcoordinate(ts)
    Returns the coordinates used by the timestepper object.
getmethod(ts)
    Returns the name of the integration scheme used.
setcoordinate(ts,coord)
    Sets the coordinates to be used by the timestepper object.
setmethod(ts,method)
    Assignes the numerical scheme to be used by the timestepper object.
subsref(ts,s)
    This function overloads the parenthesis of a timestepper object.
timestepper(varargin)
    Constructor TIMESTEPPER virtual superclass.
```

Index