A MuPAD Library for Differential Equations*

Jay Belanger[1], Marcus Hausdorf[2], and Werner M. Seiler[2]

[1] Division of Mathematics and Computer Science, Truman State University, Kirksville, Missouri 63501, USA; belanger@truman.edu
[2] Lehrstuhl für Mathematik I, Universität Mannheim, 68131 Mannheim, Germany; {hausdorf,werner.seiler}@math.uni-mannheim.de

Abstract. We present an overview of the *MuPAD* library *DETools* for the analysis of differential equations. It has been developed within an object-oriented environment for the efficient representation of differential functions. Currently, the main ingredients of the library are a fairly general package for Lie symmetry analysis and a algebraic-geometric completion package for overdetermined systems.

1 Introduction

Differential equations are of central importance in all applied sciences. Whenever a continuous process is modelled mathematically, chances are high that the model is built upon them. Thus it is not surprising that almost any computer algebra system provides functions for analysing differential equations. Within *MuPAD*, this is largely the task of the library *DETools*.

This library is still in a rather early stage of its development and only parts of the planned functionality are already realised in the upcoming new release of *MuPAD* (2.0). We describe briefly the main components of the library in its current state: a fairly general package for Lie symmetry analysis and a algebraic-geometric completion package for overdetermined systems. In addition, there exist some miscellaneous functions, especially for the symbolic and numeric analysis of mechanical systems.

Of course, *MuPAD* is not the first computer algebra system to provide such packages; there exist so many symmetry and completion packages that it is impossible to list them here. A good overview is contained in the survey article [14]. As some of these packages have been developed for many years, they are considerably larger and offer more functionality as our library in its present state. On the other hand, our library uses at several places new ideas both on the theoretical side and on the level of implementation.

DETools is developed within an object-oriented environment for differential functions realised with the `domains` package of *MuPAD* [5]. This environment contains a number of categories and domains for efficiently representing different types of differential functions (linear, polynomial, general expressions). It permits the easy interaction between different packages of the library, as the exchange of data causes no problems.

* This work has been supported by Deutsche Forschungsgemeinschaft, Landesgraduiertenförderung Baden-Württemberg and INTAS grant 99-1222.

As many users of *MuPAD* are not familiar with domains, the library has been written in such a way that no knowledge of the environment is required for using its basic functionality. Only for advanced applications it is of advantage to directly use the domains. The main reason for embedding the library in the environment is the possibility of generic implementations. For example most differential equations appearing in applications have at most polynomial non-linearities. The *DETools* library will automatically perform all computations with such equations in polynomial arithmetics which is much faster than working with general expressions.

This article is written in a rather informal style. No details on the underlying mathematics are given; they can be found in the cited references. Instead emphasis is put on presenting some concrete examples. The article is organised as follows. The next section reviews our object-oriented environment for differential equations. The following two sections discuss first the Lie symmetry package and then the completion package. The final section gives an overview of some miscellaneous functions contained in the library.

2 An Object-Oriented Environment

Object-oriented programming in computer algebra tries to mimic a categorical approach to mathematics: each object belongs to a computational domain the properties of which are determined by its category. So the general properties of our basic objects, *differential variables* and *functions*, are specified in the two categories `Cat::DifferentialVariable` and `Cat::DifferentialFunction`. Concrete representations are implemented in a series of domains within these categories. Each user may extend the environment by developing own domains. As long as they satisfy the specifications of the mentioned categories, they will interact without problems with all other domains. However, for most purposes the domains already provided should suffice.

2.1 Differential Variables

`Cat::DifferentialVariable`. We distinguish three different types of differential variables. The *dependent variables* represent the unknown functions in a differential equation; their arguments are the *independent variables*; and of course there appear *derivatives* of the dependent variables with respect to the independent ones. In order to have a consistent scheme of denoting these variables, we assume that the independent and the dependent variables are given in the form of ordered lists. Then each can be addressed by its *index* giving its position in the respective list. Derivatives are additionally characterised by a *multi* or a *repeated index*: a list showing with respect to which independent variables the differentiation takes place. For example,

$$\frac{\partial^5 u(x,y)}{\partial x^4 \partial y} \qquad \text{yields} \qquad \begin{cases} \texttt{[4,1]} & \text{as multi index,} \\ \texttt{[1,1,1,1,2]} & \text{as repeated index.} \end{cases}$$

These three ingredients constitute the specification of any differential variable in `Cat::DifferentialVariable`. The category provides a number of operations on the variables, most importantly differentiation. One must distinguish between differentiation of a dependent variable or derivative with respect to independent variables (henceforth called *total differentiation*) and *partial differentiation* of one variable with respect to another; in the latter case only 0 and 1 are possible results. Other methods include the determination of the *order* and *class* (the lowest occurring independent variable in a differentiation) of a variable, manipulations of multi indices and orderings.

`Dom::DifferentialVariable`. In order to speed up some of the computations, the representation of an element from this domain has, in addition to the three ingredients described above, two further slots in which the order and class of a variable are stored explicitly. Their values are determined every time a variable is created or changed. The form of the output is set according to how the names of the variables are given at the creation of the domain: if this is done by providing two lists with the names of the independent resp. dependent variables, repeated index notation with the indices substituted by the corresponding variable names is chosen; otherwise, if two indexed identifiers are given, multi index notation is used. An important task of `Dom::DifferentialVariable` is to convert to and from other representations for differential variables, e. g. in the `diff` or `D` notation of *MuPAD* and thus to provide a convenient interface. It reimplements many methods from `Cat::DifferentialVariable` in a more efficient way now that the concrete representation of the objects is known.

Example 1. We create two different domains of differential variables; the first one uses repeated indices, the second one multi indices. Both contain two independent variables and one dependent variable.

MuPAD
```
>> DV  := Dom::DifferentialVariable([x,t], u):
>> IDV := Dom::DifferentialVariable(x[2],u[1]):
>> DV(diff(u(x,t),x,t)):DV(u([x,t]));
>> IDV(D([1,2],u)):IDV(u([1,1]));
```
Output
```
                              u([x, t])
                              u([1, 1])
```

The last two lines show several methods to generate the derivative u_{xt} resp. $u_{x_1x_2}$ in both domains. Note that for improved readability the output is always in a condensed notation which may also be used for input. •

`Dom::RestrictedDifferentialVariable`. This domain is used for imposing restrictions on an already existing domain of differential variables. At creation, it takes as arguments this domain and a criterion which must be fulfilled by the variables belonging to the new domain. This can be for example a condition on the order or the type of the variables. The domain is mainly used in connection with linear differential functions or polynomials for specifying the type of the coefficients (see below).

2.2 Differential Functions

`Cat::DifferentialFunction`. The category of differential functions over a domain of differential variables `DV` inherits the methods from `Cat::SemiGroup`, i. e. addition of elements is defined. If multiplication is possible,[1] too, it should belong to `Cat::PartialDifferentialRing`. The category provides default implementations for a variety of operations on differential functions and infers only the methods `_plus`, `diff`, `subs` (substitution of differential variables), `solve`, `has` (checking whether a certain expression occurs in a function) and `indets` (set of all indeterminates of a function). Among the methods already implemented are the following ones. `leader` returns the leading differential variable[2]; `class` and `order` give the respective values of this leading derivative. `diffSubs` substitutes a differential variable and all its derivatives; `eval` allows to substitute general expressions for the dependent variables, which is useful for entering a solution ansatz into an equation. `totalDiff` performs total differentiations. `autosimplify` and `autoreduce` provide simplification of a list of differential function without resp. with consideration of differential consequences.

`Dom::DifferentialExpression`. This domain "lifts" the standard *MuPAD* expressions to `Cat::DifferentialFunction`. Thus besides the methods mentioned there many *MuPAD* functions like `simplify` or `normal` are lifted, too. The representation consists of two slots: the first contains the actual expression, the second a set of all occurring differential variables. Note, however, that for efficiency reasons this set is in general only a superset of the actual variables: if, as a result of an arithmetic operation, a variable disappears, it is not automatically eliminated from the set.

`Dom::LinearDifferentialFunction`. This domain is used for linear differential functions. Many operations like autoreduction and autosimplification can be performed much more efficiently (and completely algorithmically) for linear systems, so that the default implementations in `Cat::DifferentialFunction` are overwritten. The functions are linear in all differential variables not appearing in the given coefficient ring. Thus one can also represent quasilinear functions with this domain. However, it may then happen that some operations are not possible, as the result is no longer quasi-linear.

[1] This is probably the case in all domains except those representing linear functions where obviously multiplication would lead to something non-linear.

[2] The default ordering of differential variables is graded reverse lexicographic.

`Dom::DifferentialPolynomial`. *MuPAD* possesses a very fast polynomial arithmetic directly implemented in the kernel. The goal of this domain is to exploit this functionality and to enable faster manipulation of expressions polynomial in differential variables. The problem one encounters, however, is that the list of variables is in general infinite. In order to benefit from the internal polynomial arithmetic the variable lists of the polynomials to be added or multiplied must coincide. A too frequent adaption of the lists considerably slows down computations. `Dom::DifferentialPolynomial` follows the strategy to always work with lists containing *all* variables up to a fixed order. Therefore, a differential polynomial has three components: the actual polynomial, the set of occurring differential variables (as for `Dom::DifferentialExpression`) and the order up to which the polynomial contains differential variables. The variable lists are adapted only *before* two polynomials are added or multiplied (if their orders do not coincide); this is done directly on the internal list representation.

Example 2. We now create some domains of differential functions using the domain `DV` of the previous example.

MuPAD
```
>> DE := Dom::DifferentialExpression(DV):
>> RDV := Dom::RestrictedDifferentialVariable(DV,
>>                                    Types={"Indep"}):
>> RDE := Dom::DifferentialExpression(RDV):
>> DP := Dom::DifferentialPolynomial(DV, RDE):
>> LDF := Dom::LinearDifferentialFunction(Vars=[[x,y,z],u],
>>                               Rest=[Types={"Indep"}]):
>> LDF::coeffRing;
```
Output
```
Dom::DifferentialExpression(Dom::RestrictedDifferentialVariable
   (Dom::DifferentialVariable([x, y, z], u), Types = {Indep}))
```

`DE` represents arbitrary expressions in all differential variables contained in `DV`. The expressions in the domain `RDE` may dependent only on the independent variables in `DV`. The domain `DP` contains differential polynomials over `DV` with arbitrary functions of the independent variables as coefficients. The last but one line shows a shortcut in creating a domain of linear differential functions `LDF`: instead of first explicitly creating a domain of differential variables and then a coefficient domain, one may specify these information with some options and the corresponding domains are automatically generated. The output shows the generated coefficient ring.

In order to demonstrate the efficiency of the polynomial arithmetics, we compare timings[3] of additions in `Dom::DifferentialPolynomial` and

[3] All timings were obtained on a Linux-PC with Pentium III processor (733MHz) and 512MB memory.

`Dom::DifferentialExpression` by creating two differential polynomials, `dp1` and `dp2`, and converting them to equivalent differential expressions.

```
MuPAD
>> dp1 := DP((u([x,x])^2*u([x])-3*x*t^2*u([t,t])^2)^5):
>> dp2 := DP((u*u([x,t])-x^2*u([x])^3*u([t,t])+u([x])^2)^4):
>> de1 := DE(dp1): de2 := DE(dp2):
>> time(dp1+dp2 $ i=1..1000);
>> time(de1+de2 $ i=1..1000);
Output
                          170
                          420
```

The addition in `Dom::DifferentialPolynomial` is more than twice as fast as in `Dom::DifferentialExpression` (0.17 ms vs. 0.42 ms). •

2.3 Interface

Many *MuPAD* users are not familiar with domains or do not want to work with them. Indeed, for a quick calculation it is rather cumbersome, if one must first generate a number of domains and then explicitly create some elements in these etc. For this reason the *DETools* library has been written in such a way that all packages and functions within it can be used without explicit reference to domain elements.

Almost all functions of the *DETools* library operate in the following way. If they are passed as arguments elements of one of the above described domains, the computations will take place in this domain. Thus advanced users can always prescribe a special domain of computation. If differential functions or equations are passed as standard *MuPAD* expression, an automatic coercion into an appropriate domain is tried. In order to enhance efficiency, it is always attempted to coerce into a domain with as much structure as possible. Five different domains are tried in the following order:

1. linear functions with constant coefficients
2. linear functions with variable coefficients
3. polynomial functions with constant coefficients
4. polynomial functions with variable coefficients
5. differential expressions

Of course, these conversion attempts require some computation time. However, in longer calculations, as they typically occur in symmetry or completion theory, this time is well spent, if all subsequent arithmetical or differential operations are accelerated. In smaller examples the time spent on conversions does not matter at all.

3 The Lie Symmetry Package

Lie symmetry analysis [15] in one of the most important techniques to study differential equations. There exist many computer algebra packages for aiding symmetry computations; a survey can be found in [14]. Currently, the *DETools* library provides functions for setting up the determining systems for several types of symmetry generators: `detSys` for Lie point symmetries, `genDetSys` for generalised symmetries, and `ncDetSys` for a very general form of non-classical symmetries. For the analysis of the determining systems, the completion package described in the next section can be used. For lack of space we will discuss in the sequel only `detSys` and `ncDetSys`, but the properties of `genDetSys` are very similar.

3.1 Infinitesimal Generators and Prolongation

Basic objects in Lie symmetry analysis are vector fields as symmetry generators. For the analysis of differential equations these vector fields must be prolonged to fields operating on jet bundles. We have developed a special domain, `Dom::JetVectorField`, for representing such fields. It provides three main operations: one can *prolong* and *project* vector fields to higher resp. lower order jet bundles and one can *apply* a field to a differential function.

Some other useful methods are implemented, too. `ansatz` returns a generic ansatz for symmetry analysis; the names of the coefficient functions must be given as argument. `characteristic` computes the characteristic of an infinitesimal generator needed for non-classical symmetries. `evolutionary-Form` transforms a vector field into an equivalent one in evolutionary form, i.e. one where all components in the directions of the independent variables vanish. For the prolongation of evolutionary fields, which are important for generalised symmetries, the command `evolProlong` exists.

Most of these operations are straightforward and comparatively cheap. Only the prolongation requires a costly recursive determination of the coefficients. But most differential equations are sparse; so if one applies the prolonged vector field to an equation, only a small subset of all coefficients are effectively needed. Rather surprisingly, most symmetry packages described in the literature do not seem to exploit this simple fact. The `prolong` method of `Dom::JetVectorField` permits this. It accepts as a second argument either an integer, in which case the result is the full prolongation up to that order, or a list of differential variables, in this case only the coefficients in those directions are determined.

Example 3. This strategy of only determining the really needed coefficients can lead to drastic savings in computation time. As a typical example of an evolution equation consider the *Cahn-Hilliard equation*

$$u_t = uu_{xx} + u_x^2 - u_{xxxx} \ . \tag{1}$$

We create a domain of jet vector fields and a suitable ansatz $\xi\partial_x + \tau\partial_t + \eta\partial_u$. Then we compute once the full prolongation up to fourth order and once only the necessary coefficients.

MuPAD
```
>> JVF := Dom::JetVectorField(DE):
>> ans := JVF::ansatz([xi,tau,eta]):
>> time(JVF::prolong(ans,4));
>> vars := map([u([t]),u([x]),u([x,x]),u([x,x,x,x])], DV):
>> time(JVF::prolong(ans, vars));
```
Output
```
820
210
```

As one can see, the second call is about four times faster, as it does not determine any coefficient in direction of a mixed derivative $u_{t\cdots x\cdots}$. In higher-dimensional examples the difference can be even larger. •

3.2 Lie Point Symmetries

The computation of the determining system is carried out in the classical way: the prolonged ansatz is applied to the equations; the result is simplified by entering the solved original equations and then regarded as a polynomial in the derivatives; finally, the vanishing of the extracted coefficients yields the determining system. By default, a general ansatz is chosen for the infinitesimal generator. But it is also possible to work with a user defined ansatz via the option `Ansatz`. The names of the occurring unknown functions must be specified with the option `Params`.

The output domain for the determining system can be controlled by the option `Expr`. If its value is `FALSE`, the output equations are contained in a domain of linear differential functions. This is the default, if the input equations belong to one of the domains describe above (i.e. the user explicitly works with domains). If `Expr` has the value `NoDiff` (which is the default otherwise), the output consists of standard *MuPAD* expression; however, differential variables are represented in condensed notation. This is similar to `Expr=TRUE` where also pure expressions are returned but derivatives are represented in the full `diff` notation (see the example below).

With the option `AutoReduced=TRUE`, the determining system is automatically simplified. Finally, an idea first mentioned in [4] has been implemented via the option `Selected`. One may specify a sublist of the input equations; the prolonged ansatz is then applied only to these. This strategy can be fairly useful for tackling very large systems step by step.

Example 4. The determining system for the Cahn-Hilliard-equation (1) can be computed with the command

```
MuPAD
>> detools::detSys(u([t])=u*u([x,x])+u([x])^2-u([x,x,x,x]),
>>   [x,t], u, Expr=NoDiff, Autoreduced=TRUE)
                         Output
--            PHI1
|  XI1([x]) + ----, XI2([x]), PHI1([x]), XI1([t]), XI2([t])
--            2 u

    2 PHI1                                          PHI1 --
  + ------, PHI1([t]), XI1([u]), XI2([u]), PHI1([u]) - ----  |
      u                                             u    --
```

Here no user defined domains are involved in either input or output. The result consists of standard *MuPAD* expressions. But for improved readability the condensed notation `XI2([x])` is used instead of the usual notation `diff(XI2(x,t,u),x)` etc. •

3.3 Non-Classical Symmetries

Non-classical symmetries have been originally introduced by Bluman and Cole [2]. However, they and subsequently most of the authors who applied non-classical symmetries considered only what one might call the "first level". They require that the *invariant surface condition* for the used ansatz is compatible with the original differential equation, i. e. all integrability conditions are identically satisfied. However, for the existence of symmetry reductions it suffices that the invariant surface condition is consistent with the original equation, i. e. they possess common solutions. This was shown independently by Pucci and Saccomandi [18] and one of the present authors [21].

It is in principle possible to compute *any* solution of a given system of differential equations via a non-classical symmetry reduction. In practice, this seems hardly feasible, as at each level of non-classical symmetries the determining system becomes smaller, i. e. less overdetermined, but the equations in it are getting more and more complex. Furthermore, the determining systems for non-classical symmetries are non-linear. Thus it is usually impossible to compute their general solution; one must find from other considerations a useful ansatz for the symmetry generators.

Our symmetry package seems to be the first one that is able to set up the determining systems also for non-classical symmetries of "higher level". The command **`ncdetsys`** proceeds mostly in the same way as **`detSys`** does; the difference is that the invariant surface condition derived from the used ansatz (by the method `characteristic` from `Dom::JetVectorField`) is added to the input equations. This necessitates some further slight modifications, since the resulting determining equations are no longer linear. Non-classical symmetries of a "higher level" can be computed with the option **`Steps`**. The idea is to iterate the process of applying the ansatz to the combined system and restricting to the solution manifold by substituting the equations. With `Steps=1` (the default), we obtain the Bluman-Cole method; a higher number yields the above mentioned higher level symmetries.

Example 5. Bluman and Cole introduced their method for the heat equation. The relevant determining system can be computed in *MuPAD* as follows.

```
MuPAD
>> detools::ncDetSys(u([t])=u([x,x]), [x,t], u,
>>              Ansatz=xi(x,t,u)*D(x)+D(t)+eta(x,t,u)*D(u),
>>              Params=[xi,eta], Autoreduced=TRUE, Expr=NoDiff)
Output
--                                                  xi([t])
|  - eta([t]) + eta([x, x]) - 2 eta xi([x]), -------
--                                                     2

    xi([x, x])
  - ---------- + eta([x, x]) + xi xi([x]) - eta xi([u]),
       2

                                                           --
  xi([u, u]), - 2 xi([x, u]) + eta([u, u]) + 2 xi xi([u])  |
                                                           --
```

As one can see, the determining system is quasi-linear. This example also demonstrates how one can prescribe an ansatz for the symmetry generators. In this case we have set the component into t-direction to 1. In order to obtain all non-classical symmetries at the first level one would have to redo this computation without the `D(t)` in the ansatz. •

4 Completion to Involution

If one studies under- or overdetermined systems of differential equations, an important step is the so-called *completion* of the system. This comprises in particular the construction of all hidden integrability conditions of the system. A typical application is the analysis of determining systems in Lie symmetry theory (see above); but completion is also crucial for the numerical analysis [13,22,24] of such systems, for constrained mechanical systems [25] or for control theory (see examples below) to name just a few fields.

There exist many different approaches to completion but no rigorous definition of what a completed system is. Depending on the approach chosen the result of a completion may depend on additional ingredients like a ranking of derivatives or an involutive division. The completion algorithm implemented in *DETools* is rooted in the formal geometric theory of differential equations

[16,20] but also uses methods from commutative algebra, namely involutive bases [3,8,23]. For theoretical details we must refer to [12] which also contains many references to alternative approaches.

We should stress that while our completion algorithm can be used for the analysis of determining systems, it is not the most efficient choice for this particular task. In the analysis of determining systems, the main emphasis is on constructing as fast as possible as many integrability conditions as possible in the hope that they may facilitate the explicit integration of the system. In contrast, our algorithm aims at obtaining intrinsic geometric information, as it represents an efficient realisation of the Cartan-Kuranishi completion.

Former implementations of the Cartan-Kuranishi algorithm [19] working purely geometrical performed many redundant computations. Our new algorithm avoids these using some algebraic concepts. An open question is its relative efficiency compared with approaches based on differential forms [9].

4.1 The Algorithm

The general framework for our algorithm is the *Cartan-Kuranishi completion* in formal theory. A system of differential equations of order q is interpreted as a fibred submanifold $\mathcal{R}_q$ of a jet bundle $J_q\mathcal{E}$. Two natural operations are prolongation to higher order jet bundles and projection to lower order bundles which may produce integrability conditions. The algorithm completes (under some mild regularity assumptions) any system to an *involutive* one. Involution does not only imply *formal integrability*, i. e. the absence of hidden integrability conditions, but also poses some combinatorial conditions on the *symbol* $\mathcal{M}_q$ (the highest order part of the linearised system). As Fig. 1 shows the algorithm can be formulated in a completely intrinsic manner, i. e. without reference to any particular system of coordinates.

The implemented algorithm proceeds exactly as outlined above which has the advantage over, say, purely differential algebraic techniques that all intrinsic geometric information is retained. For efficiency reasons, the tests of a symbol resp. a system for involution, however, rely heavily on the algebraic theory and a special representation of the system. In general, one can describe the main idea as to partition for each equation the set of independent variables into *multiplicative* and *non-multiplicative* ones. In each step, only the prolongations by non-multiplicative variables are effectively computed. Then they are reduced by multiplicative prolongations determined only when needed. In an involutive system all non-multiplicative prolongations can be reduced to zero by the multiplicative ones.

4.2 The Completion Command

`detools::complete` expects as its first argument the list of equations to be completed. Currently, it can handle only linear equations or equations with polynomial non-linearities. The completion process can be influenced by a

Algorithm: *Cartan-Kuranishi completion*
Input: differential equation $\mathcal{R}_q$
Output: involutive differential equation $\mathcal{R}^{(s)}_{q+r}$

$r \leftarrow 0,\ s \leftarrow 0$
`repeat`
  `while` $\mathcal{M}^{(s)}_{q+r}$ is not involutive `do`
    $r \leftarrow r + 1$
  involutive $\leftarrow \left(\mathcal{R}^{(s+1)}_{q+r} = \mathcal{R}^{(s)}_{q+r}\right)$
  `if` `not`(*involutive*) `then` $s \leftarrow s + 1$
`until` *involutive*

Fig. 1. Cartan-Kuranishi completion of differential equations

number of options. `Output` controls the amount of information printed during the completion; this can range from nothing to all details of every reduction step. If the option `FractionFree=TRUE`, divisions by coefficients are avoided; otherwise the coefficient of the leading derivative is always normalised to one. In order to avoid problems with δ-singular coordinate systems, one can give some positive value i to the option `Random`. Then a linear transformation of the independent coordinates is performed with a random matrix whose entries are integers between $-i$ and i.

Example 6. We consider the linear system $u_{zz} - yu_{xx} = 0$, $u_{yy} = 0$, a classical example due to Janet.

MuPAD
```
>> sys := map([u([z,z])-y*u([x,x]), u([y,y])], LDF):
>> detools::complete(sys, Output=3);
```
Output
```
[u([z, z]) - y u([x, x]), u([y, y]), u([y, y, z]), u([x, x, y]),
   u([x, x, y, z]), u([x, x, x, x]), u([x, x, x, x, z])]
```

Besides the final result shown above, the `complete` function with the given value of the option `Output` produces a number of messages reporting on the progress of the computation. As this output is rather lengthy, we show it only in an edited form in Fig. 2.

As one can see, the output permits us to reconstruct the steps of the intrinsic Cartan-Kuranishi completion. In this particular case it produces the following sequence of differential equations

$$\mathcal{R}_2 \to \mathcal{R}_3 \to \mathcal{R}^{(1)}_3 \to \mathcal{R}^{(1)}_4 \to \mathcal{R}^{(2)}_4 \to \mathcal{R}^{(2)}_5 \,. \tag{2}$$

Current system: $\mathcal{R}_2^{(0)}$; Dimension: 8 $\mathcal{M}_2^{(0)}$ not involutive; Dimension: 4	$\mathcal{M}_4^{(1)}$ involutive; Dimension: 2 $\mathcal{R}_4^{(1)}$ not involutive
Current system: $\mathcal{R}_3^{(0)}$; Dimension: 12 $\mathcal{M}_3^{(0)}$ involutive; Dimension: 4 $\mathcal{R}_3^{(0)}$ not involutive	Current system: $\mathcal{R}_4^{(2)}$; Dimension: 12 $\mathcal{M}_4^{(2)}$ not involutive; Dimension: 1
Current system: $\mathcal{R}_3^{(1)}$; Dimension: 11 $\mathcal{M}_3^{(1)}$ not involutive; Dimension: 3	Current system: $\mathcal{R}_5^{(2)}$; Dimension: 12 $\mathcal{M}_5^{(2)}$ involutive; Dimension: 0 $\mathcal{R}_5^{(2)}$ involutive
Current system: $\mathcal{R}_4^{(1)}$; Dimension: 13	Final system: $\mathcal{R}_5^{(2)}$; Dimension: 12

Fig. 2. Output produced during the completion of the Janet example

As the final step did not produce any integrability conditions, $\mathcal{R}_4^{(2)}$ is a formally integrable but not involutive system. •

4.3 Compatibility Conditions

For systems of linear differential equations, the computation of compatibility conditions corresponds to determining a list of generators for the syzygy module of the system. The right hand side of each equation is set to a dependent variable of a new domain of differential variables (with the independent variables being the same as for the input system); all prolongations and reductions in the completion process are also simultaneously applied to these right hand sides. If the left hand side of an equation is reduced to zero, the right hand side yields a compatibility condition; thus all relations between the equations have become apparent when the completion is finished.

If this is repeated, i. e. one computes the compatibility conditions of the compatibility conditions and so on, the process terminates and yields a kind of finite free resolution, called the *Janet sequence*. One may also go in the other direction and try to find for a given system another system the compatibility of which is exactly the first system. This is referred to as computing a *potential* or a *parametrisation*. These concepts are important in control theory [26]. For checking whether a parametrisation exists, and in case that it does, for computing it, we use the method described in [17].

Example 7. Computation of compatibility conditions can be invoked with the option `Compatibility=TRUE`. For the above mentioned methods involving the Janet sequence, the commands `janet` and `parametrise` exist. In this example, we work with matrices of linear differential operations instead of systems of linear differential equations; the commands `makeMatrix` and `makeSystem` convert between these two representations.

One example of a Janet sequence are the well known operators *grad*, *rot* and *div*. Here, each operator describes the compatibility conditions of the previous one. In *MuPAD*, this looks as follows:

MuPAD

```
>> LDO := Dom::LinearDifferentialOperator([x,y,z]):
>> MLDO := Dom::Matrix(LDO):
>> grad_op := MLDO([[D(x)],[D(y)],[D(z)]]):
>> detools::janet(grad_op);
```

Output

```
-- +-    -+  +-                        -+                        --
|  | D(x) |  |  D(y), -D(x),    0      |                         |
|  |      |  |                         |  +-                -+   |
|  | D(y) |, |  D(z),    0,    -D(x)   |, | D(z), D(y), D(x) |   |
|  |      |  |                         |  +-                -+   |
|  | D(z) |  |    0,    D(z), -D(y)    |                         |
-- +-    -+  +-                        -+                        --
```

A more interesting example where it is shown that the linearised Einstein equations do not possess a potential can be found in [10]. •

5 Miscellaneous Functions

Besides the two packages discussed above, the *DETools* library contains a some further functions. However, most of them are in a less developed state. We discuss below a solver for partial differential equations and a very useful utility function. Before, we briefly mention some functions for the numerical and symbolic analysis of mechanical systems. For lack of space we cannot give a more detailed exposition, as this would require a separate paper.

The analysis of mechanical systems starts with setting up the equations of motion. The *DETools* library contains functions for generating *Euler-Lagrange equations* and *Hamiltonian equations* for finite- and infinite-dimensional systems. *Constrained systems* can already in principle be treated with the help of the completion package but improved routines for different constraint formalisms are in preparation. The equations of motion can be generated in such a way that they can directly be numerically solved with the functions of the *MuPAD* `numeric` library. There also exist several functions of the visualisation of the output either as trajectories or phase plots.

Of independent interest for the numerical analysis of differential equations are some functions for the generation and analysis of numerical methods. For example, it is possible to compute the coefficients for finite difference methods with arbitrary stencils following an algorithm of Fornberg [6,7]. For ordinary differential equation the so-called *method of modified equations* [1] has been implemented. It constructs a perturbation of the given equation which better approximates the numerical solution of the original equation obtained with some numerical scheme.

5.1 A Solver for Partial Differential Equations

In many cases, the important properties of the solutions of differential equations are found by analysing the equations rather than the solutions themselves. In some cases, however, it may be useful to have an explicit solution to examine. `detools::pdesolve` can be used to solve certain partial differential equations. The equation can be given in ordinary `diff` notation, or in condensed form, and the function to be solved for must be specified. With no optional arguments, `detools::pdesolve` will return a general solution; initial conditions can be provided to get a particular solution.

Example 8. The initial conditions are given as the values of the variables on an initial surface. Since this example has three independent variables, the initial conditions are given using two parameters. The parameters must be given as a separate list.

```
MuPAD
>> detools::pdesolve(u = u([x]) + x*u([y]) - 2*u([z]),
>>     u(x,y,z), [u=S,x=T,y=S+T,z=S-T],[S,T]);
Output
                                       2 1/2
u = exp(x - (4 x - 2 y + 2 z + x )    )
                                         2 1/2
   (2 x + z - (4 x - 2 y + 2 z + x )    )
```

The differential equation was converted into a linear differential function, and then the method `detools::charSolve` was called applying the classical method of characteristics. This is equivalent to the following.

```
MuPAD
>> de := LDF(u - u([x]) - x*u([y]) + 2*u([z])):
>> detools::charSolve(de,[u=S,x=T,y=S+T,z=S-T],[S,T]);
```

Here the user directly calls `charSolve`. This is possible for most solution methods known to `pdesolve`. •

As an alternative to finding a closed form solution, if `pdesolve` is given the option `"Series"`, it will find a power series solution. To do this, *MuPAD* must be able to solve the equation for an expression of the form $\partial^n u/\partial t^n$, where u is the dependent variable, t is one of the independent variables, and n is the order of the equation. Any initial conditions in this case can be specified by a list whose first element is an equation `t =` *value*, specifying an initial surface, and whose second element is another list, consisting of the values of $[u, \partial u/\partial t, \ldots, \partial^{n-1}u/\partial t^{n-1}]$ on the initial surface. Other optional arguments are the centre and the order of the resulting series.

Example 9. We compute a series solution around the point $(1, 2)$ of the differential equation $yu_{xx} - u_y = x$ for the initial conditions $u(1, y) = y$ and $u_x(1, y) = y + 2$.

MuPAD

```
>> detools::pdesolve(y*u([x,x]) - u([y]) = x, u(x,y),
>>                   [1,2], 3, [x=1,[y,y+2]], "Series");
```

Output

```
                                        2
2 + 4 (x - 1) + (y - 2) + 1/2 (x - 1)  + (x - 1) (y - 2) +
   O([x - 1, y - 2], 3)
```

As sufficiently many initial conditions are prescribed, no undetermined parameters appear in the solution. Without initial data the general power series solution is computed. •

5.2 The "Derivative Tree"

In computations with differential equations one relatively often encounters the following problem; thus it is rather surprising that to our knowledge nobody has published a solution so far. Assume that we are given a function $f(x, y)$ and we need f_{xx} and f_{xy}. Obviously, the most efficient way to compute these two derivatives is to first compute f_x. If one started with computing f_y, one would need one differentiation more. This example is trivial but assume that we need six different derivatives up to fourth order of a function depending on five variables, then it is less clear how one can find an optimal way to compute these derivatives with the least number of differentiations.

The *DETools* library contains a function `derList2Tree` taking as argument a list of multi indices representing the needed derivatives and returning a tree-like structure representing a way to compute these derivatives. It is based on a simple heuristic procedure,[4] since an exact algorithm seems hard to find and would perhaps incur more overhead than time saved. In all examples we have studied so far, it has produced the optimal solution.

We define for a multi index $\mu = (\mu_1, \dots, \mu_n)$ its norm $\|\mu\| = \sum_{i=1}^{n} |\mu_i|$. Using the distance function

$$d(\mu, \nu) = \begin{cases} 0 & \text{if } \|\mu - \nu\| = \|\mu\| - \|\nu\|, \\ \|\mu - \nu\| & \text{else}, \end{cases} \tag{3}$$

we proceed for an input set $\mathcal{S}$ of multi indices as follows:

1. Compute the multi index α with $\alpha_i = \min_{\mu \in \mathcal{S}} \mu_i$ and subtract it from all multi indices in $\mathcal{S}$; if the zero index occurs now in $\mathcal{S}$, it is removed

[4] We are indebted to Sven Helmer (Universität Mannheim) for this heuristics.

and the position of the corresponding multi index to be computed in the original list is inserted.
2. Determine those $\mu, \nu \in \mathcal{S}$ for which $d(\mu, \nu)$ is maximal.
3. Partition $\mathcal{S}$ into two lists: $\mathcal{S}_\mu$ contains all multi indices lying closer (with respect to the distance function d) to μ, $\mathcal{S}_\nu$ all those closer to ν.
4. Apply the above procedure recursively to $\mathcal{S}_\mu$ and $\mathcal{S}_\nu$.

Example 10. Consider the result of the call

```
MuPAD
>> detools::derList2Tree([[0,4],[2,3],[3,2],[2,1],[4,0]])
                      Output
[[[[0, 3], [[[0, 1], 1], [[2, 0], 2]]],
   [[2, 0], [[[[2, 0], 5], [[0, 1], 4, [1, 1], 3]]]]]]
```

It can be interpreted as follows. Go to `[0,3]` and branch: the first multi index of the argument list is reached at `[0,4]`; then proceed to `[2,3]` and return the second index. Start again at `[0,0]` and go to `[2,0]` this time, then branch: first to `[4,0]` (the fifth index) and second to `[2,1]` (the fourth) and `[3,2]` (the third). The total (and also minimal) number of differentiations in this example is 12. •

The function **`derList2Tree`** will be rarely called interactively, but a number of functions we have already mentioned internally use it. This includes the methods **`eval`** and **`diffSubs`** in any domain from **`Cat::Differential-Function`**. In particular, the fast prolongation of symmetry generators (see Example 3) is based on the "derivative tree".

References

1. M.O. Ahmed and R.M. Corless. The method of modified equations in Maple. Technical report, Dept. of Applied Mathematics, University of Western Ontario, London (Canada), 1997.
2. G.W. Bluman and J.D. Cole. The general similarity solution of the heat equation. *J. Math. Mech.*, 18:1025–1042, 1969.
3. J. Calmet, M. Hausdorf, and W.M. Seiler. A constructive introduction to involution. In *Proc. Int. Symp. Applications of Computer Algebra – ISACA '2000*. World Scientific, Singapore, to appear.
4. B. Champagne, W. Hereman, and P. Winternitz. The computer calculation of Lie point symmetries of large systems of differential equations. *Comp. Phys. Comm.*, 66:319–340, 1991.
5. K. Drescher. Axioms, categories and domains. Automath Technical Report No. 1, Universität Paderborn, 1996.
6. B. Fornberg. Generation of finite difference formulas on arbitrary space grids. *Math. Comp.*, 51:699–701, 1988.

7. B. Fornberg. Calculation of weights in finite difference methods. *SIAM Rev.*, 40:685–691, 1998.
8. V.P. Gerdt and Yu.A. Blinkov. Involutive bases of polynomial ideals. *Math. Comp. Simul.*, 45:519–542, 1998.
9. D.H. Hartley and R.W. Tucker. A constructive implementation of the Cartan-Kähler theory of exterior differential systems. *J. Symb. Comp.*, 12:655–667, 1991.
10. M. Hausdorf. *Geometrisch-algebraische Vervollständigung allgemeiner Systeme von Differentialgleichungen.* Master's Thesis, Universität Karlsruhe, 2000.
11. M. Hausdorf and W.M. Seiler. Differential equations in MuPAD I: An object-oriented environment. Internal Report 2000-17, Universität Karlsruhe, Fakultät für Informatik, 2000.
12. M. Hausdorf and W.M. Seiler. An efficient algebraic algorithm for the geometric completion to involution. Preprint Universität Mannheim, 2000.
13. M. Hausdorf and W.M. Seiler. Perturbation versus differentiation indices. This proceedings.
14. W. Hereman. Symbolic software for Lie symmetry analysis. In N.H. Ibragimov, editor, *CRC Handbook of Lie Group Analysis of Differential Equations, Volume 3: New Trends in Theoretical Development and Computational Methods*, chapter 13. CRC Press, Boca Raton, Florida, 1995.
15. P.J. Olver. *Applications of Lie Groups to Differential Equations.* Graduate Texts in Mathematics 107. Springer-Verlag, New York, 1986.
16. J.F. Pommaret. *Systems of Partial Differential Equations and Lie Pseudogroups.* Gordon & Breach, London, 1978.
17. J. F. Pommaret and A. Quadrat. Generalized Bezout Identity. *Appl. Alg. Eng. Comm. Comp.*, 9:91-116, 1998.
18. E. Pucci and G. Saccomandi. On the weak symmetry groups of partial differential equations. *J. Math. Anal. Appl.*, 163:588–598, 1992.
19. J. Schü, W.M. Seiler, and J. Calmet. Algorithmic methods for Lie pseudogroups. In N. Ibragimov, M. Torrisi, and A. Valenti, editors, *Proc. Modern Group Analysis: Advanced Analytical and Computational Methods in Mathematical Physics*, pages 337–344. Kluwer, Dordrecht, 1993.
20. W.M. Seiler. *Analysis and Application of the Formal Theory of Partial Differential Equations.* PhD thesis, School of Physics and Materials, Lancaster University, 1994.
21. W.M. Seiler. Involution and symmetry reductions. *Math. Comp. Model.*, 25:63–73, 1997.
22. W.M. Seiler. Indices and solvability for general systems of differential equations. In V.G. Ghanza, E.W. Mayr, and E.V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing — CASC '99*, pages 365–385. Springer-Verlag, Berlin, 1999.
23. W.M. Seiler. A combinatorial approach to involution and δ-regularity. Preprint Universität Mannheim, 2000.
24. W.M. Seiler. Completion to involution and semi-discretisations. *Appl. Num. Math.*, to appear, 2001.
25. W.M. Seiler and R.W. Tucker. Involution and constrained dynamics I: The Dirac approach. *J. Phys. A*, 28:4431–4451, 1995.
26. E. Zerz. *Topics in Multidimensional Linear Systems Theory.* Lecture Notes in Control and Information Sciences 256, Springer-Verlag, London, 2000.