Chapter 8

Computer Algebra

We will give a short overview of the application of computer algebra to differential equations and a brief introduction into the computer algebra system AXIOM. *The main part of the chapter will be the description of our* AXIOM *implementation of an environment for symbolic computations in the realm of geometric approaches to differential equations.*

8.1 Computer Algebra and Differential Equations

Most casual users of computer algebra systems think that computer algebra and differential equations means basically the design of solution algorithms. But the real situation is fairly different. Although most general purpose systems provide a kind of `solve` command for differential equations, they actually employ mainly well-known techniques and some more or less clever heuristics to choose and apply them. Especially for partial differential equations no solution algorithm treating reasonably general and complicated systems is known, hence most computer algebra systems give up very quickly.

Consequently, computer algebra applications to differential equations tend to work more indirectly. They help to study and understand properties of the solution space. Sometimes they may lead to special solutions. The reason for the use of computer algebra systems lies in the simple fact that most computations with differential equations tend to become soon very tedious. Hence such systems have proved to be an almost indispensable tool. We do not intend to give here a full survey but will briefly indicate the main directions emphasizing differential geometric methods. An excellent survey with a large bibliography oriented more towards algebraic approaches was given by Singer [143].

Almost any constructive method for differential equations has meanwhile been implemented in some computer algebra system. One can, however, distinguish certain approaches which have found most attention. One can classify them coarsely as follows:

- Completion (construction of integrability conditions)
- Symmetry analysis
- Singularity analysis
- Differential ideal theory
- Differential Galois Theory

Completion. Any approach that claims to treat general systems of partial differential equations must deal with integrability conditions. The first systematic solution was probably given by the Janet-Riquier Theory (cf. Appendix B) with the introduction of passive systems. Implementations of this ansatz have been undertaken by many authors, see e.g. [124, 135, 155]. The so-called Differential Gröbner Bases [101, 102] can also be seen as an extension of this approach. In geometric theories the notion of a passive system is replaced by involution. Hartley and Tucker [67, 68] implemented the Cartan-Kähler approach using exterior systems (cf. Appendix A). An earlier version of our AXIOM implementation of the formal approach was published in Ref. [132],

Symmetry analysis. This approach (cf. Chapter 5) has made the strongest impact on computer algebra applications to differential equations. There exist so many implementations that it is very difficult to keep an overview. We must refer here to a recent survey by Hereman [70] (an updated and extended version will appear in [71]). Most packages just set up the determining system (5.5) for symmetry generating vector fields. A few of the packages also try some heuristics to solve it [69, 134]; usually they can cope, however, only with rather simple examples.

Singularity analysis. We must distinguish two approaches: In the Painléve analysis [78] one requires that every solution has at most poles as movable singularities. Movable means here depending on initial data. This method was originally developed for ordinary differential equations and later generalized to partial differential equations [163]. It represents one of the standard tests for (completely) integrable systems. Truncated Painléve expansions are also useful for the construction of Bäcklund transformations, Lax pairs and much more [162]. An implementation for ordinary differential equations was described by Rand and Winternitz [122]. Another direction is given by the construction of formal solutions in the neighborhood of singular points of the equations [46]. It is mainly applied to ordinary differential equations.

Differential ideal theory. Here one tries to find a differential extension of algebraic ideal theory. Many of the ideas can already be found in the book of Ritt [128]. The main tools are either Gröbner bases or characteristic sets. Since the ring of differential polynomials is, however, no longer Noetherian, this generalization runs into problems, for algorithms along the lines of the Buchberger algorithm do not terminate in general [24]. As already mentioned above, Mansfield [101, 102] introduced (and implemented) so-called Differential Gröbner Bases, but due to the use of pseudo-reduction they have weaker properties than their algebraic counterpart.

Differential Galois Theory. Already Lie was looking for a differential analog of the (algebraic) Galois Theory. What is nowadays called Differential Galois Theory resembles, however, only faintly his ideas. It is mainly based on the work of Kolchin [86] and culminates in the Singer algorithm for computing Liouvillian solutions of linear ordinary differential equations with rational coefficients [142]. An implementation of this algorithm has not been achieved so far. Only the simpler Kovačic algorithm for linear second order ordinary differential equations has been implemented [129]. Pommaret [118] has developed a Differential Galois Theory following Lie's ideas and using the formal theory.

A comparison of the impact made by symmetry analysis and by Differential Galois Theory, respectively, demonstrates clearly the importance of the availability of computer algebra tools. The latter one is a hardly known theory studied only by a few experts. The former one was in the same state for many decades following Lie's original work. One reason was definitely the extremely tedious determination of the symmetry algebra for any only slightly complicated equation. As soon as computer algebra systems came up and were powerful enough, the first packages to set up at least the determining equations were developed. Since then Lie methods belong to the standard tools for treating partial differential equations.

8.2 AXIOM

AXIOM [81] was mainly developed at the IBM research center in Yorktown Heights under the name of Scratchpad II. Since 1992 it has been marketed by NAG under the new trade mark AXIOM. We are currently using Version 1.2. It is rather different compared with other general purpose computer algebra systems. Systems like REDUCE, Maple, Mathematica etc. differ not much in their principal structure: they have one basic data type for symbolic expressions and the core of the system consists of the simplifier. The real differences between the systems lay in the simplification strategy, in the power of the advanced packages (e.g. factorization, integration, equation solving) in the system library, in the user interface etc.

AXIOM has a completely different structure. It is inspired by category theory and the theory of abstract data types. Meanwhile there are attempts to incorporate this approach into other systems [60, 62], too. We will describe it briefly in this section in order to give somebody unacquainted with AXIOM a chance to understand some of the following sections. More detailed descriptions of the AXIOM type system and the underlying philosophy can be found in [36, 37, 161].

There is a three level hierarchy: Any *object* is member of one (and only one) *domain* which itself is member of a *category*. The easiest way to understand this hierarchy is to consider a concrete example: The object 5 is a member of the domain `Integer` which is a member of the category `Ring`. In the language of computer science, domains represent *abstract data types*, whereas a category is considered as a *second order type*.

Besides categories and domains there are like in most computer algebra systems *packages*. They do not implement an abstract data type but consist simply of a collection of procedures. Their structure is very similar to a domain, i.e. they first specify the exported operations, then they provide an implementation for them. Unlike a domain, however, a package does not need a representation, as it has no members.

A category specifies which operations are possible in its domains, i.e. it declares their signatures. What it does not contain are any information on the representation of objects. The category `Ring`, for instance, requires that every domain belonging to it must have an operation `+` which takes two ring elements as arguments and returns a ring element. It does not, however, give a concrete implementation of this operation, because it will probably be different in the different domains belonging to `Ring`. The addition of two integers will be provided by the domain `Integer`.

As we will see later a category can also contain implementations of operations. The big difference between a domain and a category is that the former one specifies a *representation* for its objects in terms of other data types. Within the body of the domain constructor this representation type is considered as equivalent to the new data type generated by the domain. This means that if the representation consists e.g. of a record, it is possible to access the individual slots of this record in procedures of the domain. The representation is, however, invisible outside of the domain, i.e. operations from other domains or packages which use objects of this new type cannot see the slots of the record.

A category may be a sub-category of another one. For instance every field is a ring but with additional properties. Consequently, in AXIOM the category `Field` is a refinement of `Ring` by further operations like division, i.e. it *inherits* all specifications of `Ring` and adds new ones. This avoids the tedious duplication of signatures.

Although the system's name is now AXIOM, its weak point is actually the treatment of axioms. A field is not only distinguished from a ring by the fact that it allows division but also by the fact that its multiplication is always commutative. The only way AXIOM can handle such properties is through *attributes*. `Field` has e.g. the attribute `commutative("*")`. But this amounts in practice to not much more than a comment, because the system has no way to check whether or not the implementation of `*` in a domain is commutative. Thus it is more a remainder to the programmer. It is, however, possible to ask whether a given domain has a certain attribute. This is useful, when domains or categories are passed as arguments (see below).

AXIOM's approach to symbolic computation has several advantages. It helps the programmer to ensure the correctness of his computations. Since it is a strongly typed system, every object must have a type, i.e. it belongs to a domain which determines what kind of operations can be performed with it. As long as the semantic of the used domains is sound, only well-defined operations can be called. For instance it is not possible to divide ring elements for the simple reason that no division is defined in the category `Ring`! Of course one can always circumvent this protection mechanism, but it still helps correct programming.

While this point might appear a bit vague and fancy, a real advantage of the object-oriented approach lies in the possibility of generic programming. AXIOM offers two mechanisms for this purpose. The first one is to define procedures "categorically", i.e. one implements a procedure already at the level of a category. Then every domain in the category (and every sub-category) inherits this default implementation; it can, however, still provide its own implementation overriding the default one.

A prototypical example for this technique is the Euclidean algorithm to compute greatest common divisors. AXIOM provides a sub-category of `IntegralDomain` called `EuclideanDomain`. It contains a default implementation for the operation `gcd` using the Euclidean algorithm. Every domain which belongs to `EuclideanDomain` automatically inherits it. Of course this algorithm uses operations which are differently implemented in each of these domains like multiplication. This does not matter, as the category specifies the signature of multiplication and this information suffices for the compiler.

The possibility to pass domains as parameters to another domain (or package or category) yields another way of generic programming. Again AXIOM requires that every parameter is given a type. In the case of a domain this type is the category to which it belongs. This category also specifies all operations possible with objects of the domain. Thus it is only possible to use these operations. This mechanism is for instance used in our implementation of the completion algorithm 2.3 (cf. Section 8.5). The corresponding package takes as argument a domain which represents the class of differential equations considered (e.g. linear equations or polynomial equations). It only knows that this domain lies in the category `JetBundleFunctionCategory`.

Our completion algorithm requires several non-trivial operations like computing the dimension of a submanifold or simplification of a system. The implementation of these varies strongly depending on the kind of equations considered. But this does not matter for the completion procedure which can be written completely independent. Thus one can take full advantage of specific properties of special classes of differential equations and still use the same completion procedure.

A further advantage of AXIOM is the possibility to overload procedure names, i.e. several procedures can have the same name as long as they are distinguished by different signatures. Probably the heaviest overloaded name is `coerce`. This name is always used, if objects are transformed from one representation into another equivalent one. Every domain must contain a coerce procedure to the domain OutputForm, because this determines the way its objects are displayed in the output.

The implementation of categories, domains or packages requires the use of the compiler. It is still in an experimental stage. The first official release (of a very different new one) will be with the next version of AXIOM. Interactive work with AXIOM is performed using the interpreter. It has sometimes a slightly different syntax. The main difference is, however, that in the interpreter it is no longer necessary to enter an explicit type for every object.

Since this would be very tedious, the interpreter provides a type inference mechanism which tries to derive the types of the objects in the input line. Only if there are ambiguities or the structure of the input is too complicated, the user must declare the types of at least some objects explicitly. The only information this mechanism needs are the signatures (here often called mode maps) of all used operations. (A description of this mechanism can be found in [149]).

Because of the huge size of the library it is not possible to load its complete content at the start of an AXIOM session. If the run time system needs an operation which is not "exposed", as this is called in AXIOM, it is automatically loaded. This autoload scheme does not work with user provided domains and categories. All those that will be used during the calculations must be loaded explicitly using the)load command of the interpreter. To facilitate the entering of the often very long type names the AXIOM compiler requires that every category, domain or package is assigned an abbreviation. We will also make use of this and denote everything by its abbreviation as soon as it is introduced.

The remainder of this section contains a short overview of a few domains etc. contained in the AXIOM library which have been used in our implementation. Due to the size of the library one of the biggest difficulties for an AXIOM programmer is to find the ones he needs... Basic data types comprise Integer with the subtype NonNegativeInteger, Symbol and Boolean. Rational numbers can be obtained as Fraction Integer (the domain constructor Fraction takes a ring as argument and returns its quotient field). More complicated data types can be constructed using e.g. List, Vector, Matrix or Record.

AXIOM provides a complicated hierarchy of categories for the basic structures used in algebra [38]. We will need only a few of these and most of them have already been mentioned. We will deal almost exclusively with rings, i.e. domains in Ring. Sometimes we will have additional structures or properties like a GcdDomain or a PartialDifferentialRing. The latter one adds partial differentiations with respect to elements of a given set. In our applications this set will always be the domain Symbol.

The data type for general expressions is Expression Integer. It corresponds roughly to the one basic data type used by most computer algebra systems. It can represent arbitrary expressions like $\sin(x+y)+f(x*y)$ with an unspecified function f. All the usual operations like plus, minus, differentiate, etc. can be performed within this domain. Like most other computer algebra systems AXIOM represents an element of it internally as a polynomial. The unknowns are so-called kernels. A kernel can be a simple variable like x but also a more complicated expression like $f(x*y)$. Basically everything that cannot be further simplified is a kernel (of course AXIOM disposes of a domain called Kernel for such objects). Kernels like $f(x*y)$ are generated using BasicOperator.

Polynomials are probably the most important objects in computer algebra. Hence AXIOM provides many different categories and domains for them. They vary basically in the chosen representation, as for different tasks different ones may be most efficient. SparseMultivariatePolynomial uses for instance a recursive representation and DistributedMultivariatePolynomial obviously a distributed one.

8.3 Implementation I — General Remarks

Most computer algebra applications to differential equations published so far consist of special purpose packages devoted to one specific algorithm. Communication between different packages, e.g. further processing of the output of one algorithm by another one, is then often rather awkward, as every package uses its own data structures and thus complicated conversion procedures must be implemented.

This problem could be avoided by implementing a general purpose environment for geometric computations. Such an environment should comprise basic data structures and procedures for jet bundles and differential equations as they are typically needed. Within this environment it would then be possible to implement packages for different purposes like completion to an involutive system, construction of the symmetry algebra etc.

We have started with the development of such a general environment for computations within the jet bundle formalism. We believe that AXIOM is especially well adapted for such a project because of its unique structure. A first version was implemented by J. Schü in his master's thesis [131]. Together with some minor improvements it was presented in [132]. The current version is mainly distinguished from the earlier one by much more efficient data structures and the introduction of many simplification routines.

In its current state the environment comprises over 6000 lines of AXIOM code organized in 3 categories, 11 domains and 3 packages. There is a natural hierarchy consisting of four layers:

1. Jet bundles, i.e. data structures for local coordinate systems
2. Sections, i.e. functions defined on these charts
3. Higher structures like vector fields, differential forms or a special data structure for differential equations
4. Application packages

The first layer is basically for a better looking user interface. One could argue whether it is really necessary. The only "computations" that happen in it are the construction of the output form for jet variables.

Before we will give a detailed description of the implementation, we present an overview of all the categories and domains provided in the individual layers (see also Fig. 8.1). A list of their abbreviations can be found in Fig. 8.2.

Layer 1. The category `JetBundleCategory` specifies jet bundles with a fixed number of independent and dependent variables. The order of the jet bundle is not fixed but passed as argument to the procedures needing it. An instance of this category is given by the domain `JetBundle`. It implements jet bundles with arbitrary names for the independent and dependent variables. The domain `IndexedJetBundle` takes as parameters the numbers of these variables and symbols to construct their names. The domain `JetBundleSymAna` is only needed by the symmetry package (see below).

Layer 2. The basic category is `JetBundleFunctionCategory`. There is a subcategory `BaseFunctionCategory` for functions defined only on the base space X. Currently four instances are implemented. `JetBundleExpression` is basically an extension of `Expression Integer`. Thus it represents in some sense the most general domain in this category. A similar domain is given by `JetBundleXExpression` which contains only functions of the independent variables. Its main use is to serve as coefficient ring for linear functions realized in `JetBundleLinearFunction`. `JetLazyFunction`

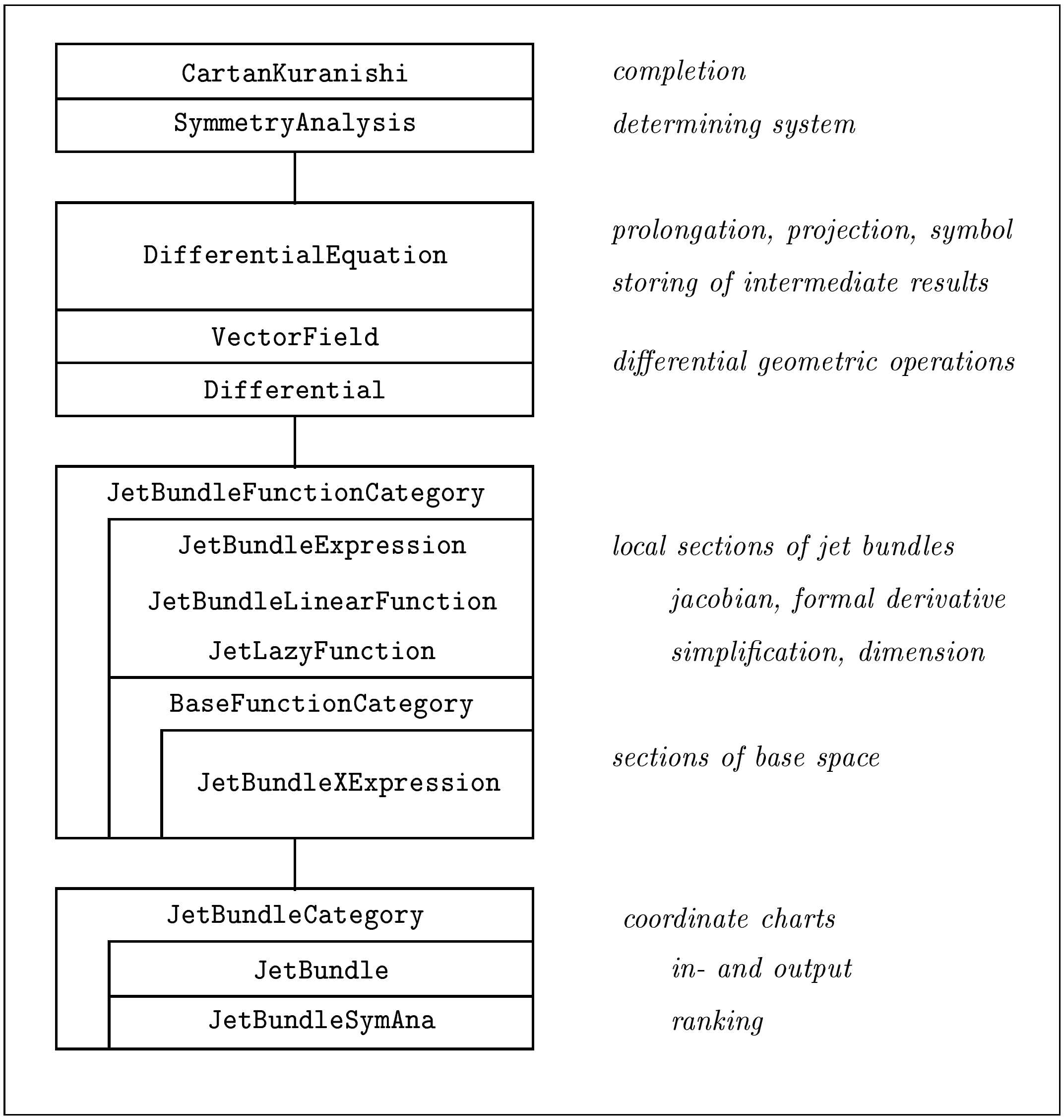

Figure 8.1: Overview of the implemented categories, domains, and packages.

implements a very special instance. It takes another domain of this category as argument and sets a lazy evaluation scheme for partial differentiations on top of it. Its main purpose is to avoid unnecessary calculations when many prolongations are needed.

Layer 3. There are two domains `VectorField` and `Differential` for vector fields and one-forms, respectively, implementing the basic differential geometric operations with them. The domain `DifferentialEquation` provides basic operations like prolongation, projection or construction and analysis of symbols and tableaux. Its main purpose is to enhance the efficiency by storing many intermediate results like Jacobians or dimensions.

Layer 4. The two application packages implemented so far are `CartanKuranishi` and `SymmetryAnalysis`. The former one contains a completion procedure as described in Section 2.4. Additionally there are some procedures for calculations with Cartan characters, the Hilbert polynomial and for gauge corrections. The symmetry package is still in a very earlier stage. Currently it contains only a primitive procedure to produce determining systems.

Besides these there are two utility packages. `SparseEchelonMatrix` provides a special representation of sparse matrices designed for the fast and efficient computation of row echelon forms. `JetCoordinateTransformation` prolongs coordinate transformations in the base space $\mathcal{E}$ into the jet bundle.

`JetBundleCategory`	`JBC`
`JetBundle`	`JB`
`IndexedJetBundle`	`IJB`
`JetBundleSymAna`	`JBSA`
`JetBundleFunctionCategory`	`JBFC`
`BaseFunctionCategory`	`BFC`
`JetBundleExpression`	`JBE`
`JetBundleXExpression`	`JBX`
`JetBundleLinearFunction`	`JBLF`
`JetLazyFunction`	`JLF`
`DifferentialEquation`	`DE`
`VectorField`	`VF`
`Differential`	`DIFF`
`CartanKuranishi`	`CK`
`SymmetryAnalysis`	`SYMANA`
`SparseEchelonMatrix`	`SEM`
`JetCoordinateTransformation`	`JCT`

Figure 8.2: Abbreviations of the new categories, domains, and packages.

Examples of the application of this environment in concrete computations will be given in Sections 8.7 and 8.8. They contain complete examples with in- and output of AXIOM sessions. Tables with most of the exported procedures can be found in Appendix D. They also contain short descriptions of them and can thus be used as a kind of reference manual.

8.4 Implementation II — Jet Bundles

This section describes in some detail the implementation of the first two layers. `JetBundleCategory` was mainly introduced to facilitate the passing of jet bundles as arguments to domains and categories of higher layers. Otherwise it would be necessary to repeat each time all the parameters of a jet bundle. Furthermore, it is convenient for some special applications to use another representation than the one provided by the domain `JetBundle`. For instance the notation used in this thesis with indexed variables is of advantage for many large systems; hence there is a domain `IndexedJetBundle`. Most operations are already implemented in `JBC`; `JB` and `IJB` provide essentially only the procedures for the generation and for the output of jet variables.

`IJB` uses as internal representation a record containing the upper and lower index of a jet variable and its type. It knows four different types: Besides the obvious ones `Indep` for the independent variables, `Dep` for the dependent ones and `Deriv` for the derivatives there is a special one called `Const`. It is assigned to the special "jet variable" `1` which is needed for differentiation and for the representation of linear functions (see below).

The upper index labels the different independent and dependent variables; the lower index is needed for derivatives. `IJB` recognizes two different notations for the lower index. Internally always standard multi-index notation is used. For in- and output the user can choose between this and repeated index notation which is the default, as it is usually more convenient for derivatives of low order. The notation can be selected with the procedure `setNotation`.

Jet variables are generated with the procedures `X`, `U` and `P`. For the special cases of only one independent or only one dependent variable, respectively, it is possible to omit the label of the variables. `P` interprets its second argument as a lower index in the currently chosen notation. `Pm` and `Pr` always use multi-index and repeated index notation, respectively.

The implemented ranking is the inverse lexicographic ranking. In addition to `<` and `>` `JBC` provides a procedure `weight` which assigns each jet variable a unique integer such that a higher ranked variable always gets a higher number. Currently there are no provisions for other rankings, although this could be useful for computations as performed in Janet-Riquier Theory. Especially rankings of the type studied by Janet can easily be parameterized by matrices [80]. This mechanism is used e.g. by the `standardform` package of Reid [124].

The `coerce` from `JBC` to `Symbol` consists of the straightforward construction of a symbol with the correct scripts. However, this operation "looses" all information about the jet variable. It is no longer possible to distinguish this symbol from others as a jet variable. Coercion to `Expression Integer (EI)` offers here much more possibilities. The corresponding procedure constructs a `BasicOperator (BOP)` which in turn can easily be coerced to `EI`.

An object of type `BOP` has associated with it a so-called property list which allows to store flags and indicators for properties of the object. We have introduced a new flag `"%jet"` to distinguish jet variables from other kernels. Furthermore we store the weight of each jet variable using the indicator `"%weight"`. This indicator is used by the type `Kernel Expression Integer` to determine an ordering. Thus we can impose our ranking even after coercion to `EI`. This is crucial for the implementation of `JBE` (see below).

`JB` uses exactly the same representation as `IJB`. However, it employs always repeated index notation. `JB` takes two lists of symbols as arguments representing the names of the independent and dependent variables. This is in many applications more natural than indexed variables. For easier input `JB` provides two additional routines with symbols as arguments.

`JBSA` is more or less identical with `JB` but computes its parameters in a special way. It takes as argument another jet bundle. Its independent and dependent variables will be the independent variables of the new jet bundle. The other arguments represent the names used for the dependent variables. The compiler needs such a domain as return type for some procedures in the symmetry package (cf. Section 8.5).

`JetBundleFunctionCategory` provides a default implementation for most operations defined by it. These can be coarsely divided into differential operations like partial or formal derivatives or computation of Jacobians, simplification routines, and some miscellaneous operations like determination of the leading derivative of a function or its effectively occurring jet variables. Furthermore there are the algebraic operations inherited from the category `PartialDifferentialRing Symbol` and some operations are lifted from `JBC` to allow for easier input.

Important among the miscellaneous operations are `jetVariables`, `solveFor`, `subst`, `order`, `leadingDer`, and `dSubst`, as these are much used by the default implementations of the other operations. For the first three of these no default implementation is provided. `jetVariables` returns a list of all effectively in a function occurring jet variables. `solveFor` tries to solve a function for a given jet variable. If that is not possible, it returns `"failed"`. `subst` substitutes a given expression for a jet variable in another given expression.

`order` yields the order as differential equation of a function; it is computed as the order of the leading derivative obtained from `leadingDer`. The default implementation of the latter one in turn calls `jetVariables` and determines the maximum of the resulting list. Whereas `subst` operates purely algebraic, `dSubst` also substitutes for derivatives of the given jet variable. This is done by repeated calls of `subst`.

For partial derivatives one must distinguish between differentiations with respect to objects of the type `Symbol` and differentiations with respect to jet variables, i.e. objects of a domain in `JBC`. The former one is inherited from `PartialDifferentialRing Symbol`. It is of no importance for the formal analysis, as such symbols are considered as parameters. Only differentiations with respect to jet variables are used in formal derivatives and thus in prolongations and similar operations.

Jacobi matrices play an important role in the higher structures discussed in the next section. To compute the formal derivative of a function according to (2.3) requires the knowledge of all its partial derivatives. There are two operations `jacobiMatrix` implemented. In one a second argument containing a list of the effectively occurring jet variables is required. This avoids a call of the procedure `jetVariables`, which can be quite expensive in some instances of `JBFC`.

Similarly, there are several different operations called `formalDiff`. Some are for a single function, some for a system; some take the Jacobi matrix as an additional argument, some compute it. One of them returns a list of the jet variables occurring in the prolonged function to facilitate the determination of its Jacobi matrix. Such "tricks" are fairly important to obtain reasonable run times.

There are two kinds of simplification routines: `simplify`, `simpMod` and `simpOne` use only algebraic operations; `reduceMod` and `autoReduce` apply also differentiations. They serve different purposes. The former ones are used to simplify local representations of differential equations. Hence they can produce an equation of lower order only, if integrability conditions occur. Indeed this is the main task of `simplify`. The latter ones are used to obtain simple generating sets for differential equations. They eliminate for example equations which are just prolongations of others. `JBFC` contains default implementations for all four mentioned simplification routines.

`simplify` has two main tasks. Besides the already mentioned exhibition of integrability conditions it should eliminate functionally dependent equations. This is important for the computation of dimensions (see below). `simpMod` is similar to `reduceMod` (see below) and simplifies a system with respect to another one. `simpOne` tries to simplify individual equations. This entails removal of unnecessary coefficients and signs or reduction of exponents in the case of non-linear equations.

The default implementation of `simplify` uses basically the leading derivatives of the equations. Obviously two equations with different leading derivatives are functionally independent. If several equations with the same leading derivatives occur, `simplify` checks first whether one of them can be solved for it using `solveFor`. If yes it performs the corresponding substitutions in the other equations. If not it tries to analyze further jet variables occurring in the equations. But obviously this will not always suffice to decide functional independence. In this case an error message is issued.

`reduceMod` reduces a system with respect to another one, i.e. it uses the equations of the second system and their prolongations to simplify the first one. `autoReduce` reduces a system with respect to itself. The default implementation depends basically on the effectivity of `solveFor`, as it tries to solve each equation for its leading derivative and to substitute it in the other equations.

As we will see in the next section the power and the efficiency of the simplification routines and here especially of `simplify` are crucial for the performance of our environment in almost any calculation. Thus special care should be applied to them in the implementation of any domain belonging to `JBFC`. It is e.g. very useful, if `simplify` returns the system in such a form that its symbol is already as close as possible to its solved form. This does not only make the handling of the symbol easier, there is a high probability that the simplification of the prolonged system will also be much easier.

There are two operations for determining dimensions: `dimension` takes as arguments a system (i.e. a list of functions) and the order q of the jet bundle of which the differential equation is a submanifold. It computes the dimension of this submanifold. `orderDim` takes the same arguments, but it assumes that all equations are of the same order as the jet bundle and calculates the dimension of the submanifold in $J_q\mathcal{E}/J_{q-1}\mathcal{E}$. In other words, if an empty list is passed as "system", then `dimension` returns $\dim J_q\mathcal{E}$ whereas `orderDim` returns $\dim S_qT^*X \otimes V\mathcal{E}$.

The default implementations of both procedures assume that the system is simplified and contains only functionally independent equations. It then subtracts the number of equations of the appropriate dimension. The reason for the introduction of `orderDim` will become clear in the next section, when we discuss the `dimension` procedure of `DifferentialEquation`.

The category `BaseFunctionCategory` specifies no additional operations. It contains only different default implementations for the procedures generating jet variables and for the coercion of jet variables to `BFC`. They result in an error message, if one attempts to generate a dependent variable or a derivative.

Simplification is especially easy in the case of linear systems. The domain `JetBundleLinearFunction` represents linear functions as a list of coefficients and a list of jet variables. This explains why `JBC` introduces the special "jet variable" `1`: it is here necessary to allow for a constant term. `simplify` calculates a row echelon form for the system. Since the lists in the chosen representation are always sorted, this puts the symbol at once in solved form. The coefficient domain must belong to `BFC`. Currently the only possibility is `JBX`. One could of course easily implement a special domain for functions with constant coefficients, but this is hardly worth while.

Simplification becomes a much more difficult problem for general expressions as implemented in `JetBundleExpression`. The current implementation assumes a polynomial dependency of the jet variables. It was originally planned to provide a special domain for polynomial equations, but there have been very bizarre problems with AXIOM. For this reason `JBE` takes currently the place of such a domain.

Polynomial equations are probably the most general equations for which there exists a general algorithm to compute dimensions of submanifolds (or better varieties). In the jet bundle the differential equations become algebraic equations. They generate an ideal. Using a Gröbner basis [9] it is easily possible to determine the dimension of the variety defined by it. `dimension` applies the algorithm described by Kredel and Weispfenning [90].

This approach has, however, two serious disadvantages. Since we use the implementation of Gröbner bases provided in the AXIOM library, we cannot track the dependency of the resulting equations of the original equations. We will see in the next section that this is quite important for the domain `DifferentialEquation`. Furthermore the dimension of jet bundles and thus the number of unknowns occurring in the polynomials grows exponentially with the order. The same holds for the number of equations, if several prolongations are needed. But the complexity of Gröbner bases calculations explodes with the number of unknowns and the number of polynomials.

The implementation of `simplify` in JBE tries therefore to avoid the use of the AXIOM `groebner` procedure as much as possible. The main strategy rests on the observation that differential equations are usually sparse, i.e. not every jet variable occurs in every equation. A large part of the simplification can often be done by simply sorting the equations according to their leading derivatives. If there are several equations with the same leading derivative, `simplify` checks whether one of them can be solved for it. In this case this equation can be substituted in the other ones. Only if this is not possible, a Gröbner basis is computed — but only for the equations with the same leading derivative.

One could argue that this approach does not necessarily compute a Gröbner basis for the complete system and that thus the use of the algorithm of Kredel and Weispfenning is not correct. But this is not true. We obtain this way indeed a Gröbner basis but not a reduced one! Obviously we construct a basis with the same leading monomials as a reduced basis. By a well-known characterization theorem for Gröbner basis this implies that we get indeed such basis. In any case the algorithm of Kredel and Weispfenning needs only these leading monomials.

Solving an equation for a given jet variable provides of course also problems, if general expressions are allowed. Currently `solveFor` can handle only the case that the function depends linearly on this variable. But it may be non-linear in other variables. One could of course try to treat some more general cases, but we do not think that this is worth while the effort. For the purpose of simplification this strategy usually suffices, as prolongations lead to quasi-linear equations.

The domain `JetLazyFun` is rather special and still in an experimental state. But it shows the great possibilities offered by a system like AXIOM. The motivation for its development have been the following considerations. If we compare the calculations performed in our completion algorithm 2.3 with other approaches to the construction of integrability conditions, it seems that we perform many unnecessary differentiations.

Most other approaches (take e.g. Reid's standard form [124]) prolong at first sight only those equations where one can suspect from the leading derivatives that cross-derivatives may generate integrability conditions. In our approach we must always prolong *all* equations in order to obtain all independent equations in the jet bundle of the next order. In principle, Proposition 2.9 tells us that for the generation of integrability conditions only the prolongations with respect to non-multiplicative variables are of interest. The other prolongations yield automatically independent equations. Zharkov and Blinkov [165, 166] proposed a new approach to Gröbner bases (of algebraic equations!) based on the completion of an equivalent linear system of partial differential equations to an involutive one. Their algorithm tries to exploit this fact by considering only non-multiplicative prolongations.

A closer analysis reveals, however, that the difference between the two approaches is not as big as it seems. The generation of the potential integrability conditions alone does not suffice. One must also check whether they represent truly new independent equations. This requires simplification and hence reduction by other equations. But now one needs the prolongations with respect to the multiplicative variables, too. Algorithms like the one proposed by Zharkov and Blinkov or like the standard form algorithm of Reid compute these prolongations each time they are needed new. Thus there also occur redundancies.

A priori it is not clear which approach will lead to less unnecessary differentiations.

One could argue that this is only a problem of implementation. One could store each computed prolongation for later reuse. But now the question arises how this can be organized efficiently. JLF could be viewed as a kind of synthesis of these two apparently so different approaches. It puts on top of another domain of JBFC a lazy evaluation scheme for partial differentiations. This means that the procedure `differentiate` does not really compute a partial derivative but only stores a pointer to the function to be differentiated and the variable with respect to which it is differentiated. Only if later the full expression is needed, the differentiation is actually performed.

Such lazy evaluation schemes have been successfully applied to the implementation of infinite objects like series [19]. There the idea is to compute only as many terms as momentarily necessary, but to be able to calculate more terms if they are needed. In our case the idea is that the most expensive part of a prolongation, namely the many partial derivatives, is not really performed. As in the approach of Zharkov and Blinkov only the prolongations which are absolutely necessary to decide the existence of integrability conditions are computed.

Of course there is a price to be paid. The representation of JLF is a complicated three level hierarchy connected by pointers. To avoid as many evaluations as possible information about the leading derivatives and the occurring jet variables are stored. This introduces a considerable overhead. Thus the use of JLF is only effective, if one can really omit many differentiations. We will discuss this again in Section 8.7 based on examples.

The guiding principle in the implementation of JLF is to use the leading derivative of a function as much as possible. Testing for zero is e.g. always a very important operation: `Ring` specifies a special operation `zero?` for this purpose. Many rings provide a special implementation of it which is more effective than the use of `=`. This is also the case in JLF where `zero?` uses the fact that the leading derivative of zero is the special jet variable `1`. Thus only if `leadingDer` yields this value an evaluation is necessary to see whether the function really vanishes.

This makes of course the effective implementation of `leadingDer` especially important. This is reflected in the choice of the representation. The lowest level represents a "lazy derivative". It consists of a record containing a function, a variable and a flag. If the value of the flag is `false`, then the differentiation has not yet been performed. The function is then the one to be differentiated. If the flag's value is `true`, the function represents already the result.

The next level represents a "lazy term", i.e. a coefficient and a pointer to a lazy derivative. In addition there are two flags and a jet variable. One flag indicates again whether or not the term has already been evaluated. In this case only the coefficient is relevant, as it contains the result. The jet variable provides an upper bound for the leading derivative of the term. The second flag indicates whether or not this bound is sharp, i.e. whether it is only a bound or really the leading derivative.

The top level record contains besides a list of pointers to lazy terms two flags, two lists of jet variables and a single jet variable. This single variable represents again an upper bound for the leading derivative of the total expression. One of the flags shows whether it is a sharp bound. The first of the two lists of jet variables represents all variables of which it is known that they occur effectively in the expression. The second one contains those which may occur. The list of lazy terms is considered as a sum. It is ordered by the leading derivatives of the terms. The second flag again indicates whether there are still unevaluated terms present. The representation of JLF is a pointer to such a record.

Pointers (in AXIOM realized with the domain `Reference`) are chosen in order to "communicate" the result of an evaluation. This is necessary, because arithmetic operations lead to a kind of proliferation of lazy derivatives or terms. If now such a lazy object is evaluated, the result should be available in all expressions containing it. This can be ensured through the use of pointers.

With this representation it is easy to implement `leadingDer`. It looks first whether the stored leading derivative represents only an upper bound. If not it is returned and no evaluation was performed. Otherwise it starts using the procedure `eval1` to evaluate as many of the lazy terms as necessary to obtain a sharp bound.

We have already mentioned that `zero?` (and similarly `one?`) is based on `leadingDer`. But many procedures implemented categorically in `JBFC` use `zero?`, e.g. `jacobiMatrix` to avoid vanishing entries. This would lead to many unwanted evaluations. We have introduced the attribute `lazyRep` to distinguish domains like `JLF` with a lazy evaluation mechanism. In the case of such a domain `JBFC` tries to avoid procedures like `zero?` which might cause evaluation as much as possible. The same holds for the matrix domain `SEM` discussed in Section 8.6.

As explained above evaluation of a lazy derivative or a lazy term may effect several expressions. Thus an expression might have a more complicated representation than necessary, as some of its lazy terms are already evaluated. The procedure `collect` collects all such terms and leads hence to a simplification of the representation. It causes, however, no evaluation and changes of course not the value of the expression.

The implementation of `simplify` is very close to the default one in `JBFC`. As far as possible only the leading derivatives are used. But if several equations with the same leading derivatives occur, they must be evaluated either to try to solve them or to use the `simplify` procedure of the base domain. Many operations of `JBFC` cannot be implemented without evaluation. Hence they should be avoided as much as possible. They include `subst`, `solveFor` and all reduction procedure.

8.5 Implementation III — Higher Structures and Applications

This section describes some details of the implementation of the layers 3 and 4. The core of the third layer is of course the domain `DifferentialEquation`. As already explained in Section 8.3 its main purpose is to enhance the efficiency by storing some intermediate results. The two other domains `VectorField` and `Differential` could also be considered as utilities.

In principle we could use simply a list of functions of some domain in `JBFC` to represent a differential equation locally. The domain `DE` has, however, a more complicated representation. The most important point is that all equations are sorted by order. This allows to keep previous results, especially previous simplifications, almost unchanged by prolongations, as only integrability conditions can affect the lower orders. Such considerations play an important role for an efficient implementation of the completion algorithm outlined in Section 2.4.

The chosen data structure holds much more information than just the equations. The top level is a record of two lists: one contains the effectively occurring orders, the other one for each order a record. This record consists of a list of equations, namely all equations of the given order in the system, the Jacobi matrix of these, and some flags indicating whether the equations of this order are already simplified (`Simp?`), whether they have already been prolonged (`Prolonged?`), or whether their contribution to the dimension of the differential equation (see below) has already been computed (`Dim?`).

Additionally the record contains an integer list (`Deriv`) to avoid the multiple generation of the same equation during prolongations. If an equation was obtained by prolonging another one, this list contains the number of the independent variable with respect to which that equation was prolonged. In the next prolongation the equation will be prolonged only with respect to independent variables with an index greater than or equal to this number.

Because of this list we try to keep track of the equations during simplification. If an equation is a combination of several other equations, then its value in `Deriv` is determined by the minimum of the values of the other equations. This strategy may not be optimal, but it is the best one which can be realized with reasonable effort.

The central operations in `DE` are `prolong`, `project`, `dimension` and `simplify` plus the ones for symbol and tableau. `simplify` is of course based on the corresponding procedure in `JBFC` and assumes that it finds all integrability conditions. It proceeds order by order starting with the equations of highest order. If at some order an integrability condition is detected, it is at once moved to the subsystem of the corresponding order. If algebraic equations occur, `simplify` checks whether they pose conditions on the independent equations. In that case an error message `"inconsistent system"` is issued. As soon as a subsystem has been simplified, the flag `Simp?` is set. If `simplify` is called again, it recognizes this flag and avoids a second, unnecessary simplification.

In the implementation of `prolong` one must take care of both mechanisms to generate integrability conditions: the classical one via elimination of the derivatives of highest order and the prolongation of equations of lower order (cf. Example 2.4). The first one is basically the task of `simplify` applied to the equations obtained by formally differentiating the highest order equations. To avoid unnecessary differentiations of lower order equations (and the subsequent simplification!) the list `Prolonged?` stores for each equation a flag whether or not its prolongation has already been computed.

There exists also a second version of `prolong` requiring two arguments, the differential equation and an order. Only for the equations up to this order it is checked whether their prolongations have already been computed. This version is useful in the completion procedure described in the next section. It allows for the fast prolongation of a system obtained by a projection. In this case only the formal derivatives of lower order equations, namely of the integrability conditions determined during the projection, are needed.

`project` can be implemented very easily, if one assumes that it is applied only to prolonged equations which have already been simplified. This assumption is reasonable within the current context of completion to involution. For other tasks it might be useful to have a more general procedure available. It is, however, not obvious, how it could be realized for arbitrary classes of equations, as it requires then highly non-trivial, non-linear computations. With this assumption `project` amounts to nothing more than eliminating all records belonging to equations of higher order than the target order.

The determination of the dimension of a differential equation is a crucial operation in the completion algorithm 2.3. `DE` takes the following approach: Using the procedure `orderDim` of `JBFC` the contribution of each order is computed separately. It is then stored in the record containing the equations of this order and the flag `Dim?` is set. As long as no integrability conditions of this specific order are appearing this result can be reused, if e.g. the dimension of the prolonged equation must be computed. In the optimal case this requires only the determination of the contribution of the new equations of highest order.

`DE` specifies four operations for symbols: `extractSymbol` extracts it from the Jacobi matrix of the highest order part of the system. `analyseSymbol` returns a record with the values of the $\beta_q^{(k)}$, the total number of multiplicative variables and the rank of the symbol.

`prolongSymbol` computes directly the prolonged symbol instead of extracting it from the prolonged differential equation. `prolongMV` takes as argument a record as computed by `analyseSymbol` and returns the corresponding record for the prolonged symbol. The values are, however, only correct, if the symbol is involutive, as this procedure is based on Lemma 3.6.

Additionally there are two procedures `tableau` to compute the matrices of tableaux using (2.42) and (2.58), respectively. One takes only one one-form as argument. The other one expects a list of one-forms and can thus be used to compute k-tableaux. To enter one-forms the domain `Differential` must be used. It represents together with the domain `VectorField` the remainder of the third layer. The implementation of both is somewhat rudimentary, as we hope that some day AXIOM will contain a reasonable environment for differential geometric calculations and then it should be used instead of some special domains like the two mentioned.

Both use an identical representation consisting of two lists. One list contains the coefficients, the other one the basis elements. The domains for both are passed as arguments. The one for the basis elements must be a jet bundle from `JBC`, the other one a domain belonging to `JBFC`. The lists are always sorted using the lesser function of `JBC`.

Besides some elementary procedures to enter one-forms and vector fields, respectively, or to access the coefficients and basis elements we have implemented the usual arithmetic operations and a few geometric operations like commutators or the contraction of a one-form with a vector field. For vector fields we have additionally implemented a procedure `prolong` which prolongs a field given on the base bundle $\mathcal{E}$ to one acting on a jet bundle of given order using to (5.4). Such a procedure is necessary for symmetry analysis.

The fourth layer contains currently the packages `CartanKuranishi` and `Symmetry-Analysis`. `CK`'s central procedure is called `complete` and completes a given differential equation to an involutive one. Whereas `complete` only displays some results (depending on the previous setting of the output flags using `setOutMode`) and returns nothing (or `void` in the AXIOM language), its second version `complete2` returns the involutive equation, the Cartan characters and a few other information. The user can set two further flags controlling reduction and simplification with `setRedMode` and `setSimpMode`. They are discussed in more detail in Section 8.7.

`complete` implements basically the detailed algorithm depicted in Fig. 2.3 on page 23. To achieve a reasonable performance it takes full advantage of the possibilities offered by the implementation of `DE`. Basically all the computations happen in two local variables: `PrevDe` contains the current differential equation which is checked for involution; `CurDe` represents its prolongation which is of course needed for the check.

If the inner loop of our algorithm requires a prolongation, `PrevDe` gets the previous value of `CurDe` and `CurDe` is newly calculated using `prolong`. Then the necessary information about the symbols are extracted. If the symbol of `PrevDe` is involutive, involution of the differential equation is decided by comparing dimensions according to Theorem 2.7.

If integrability conditions occur, `PrevDe` is newly computed from `CurDe` using `project`. As explained above `prolong` puts such conditions directly in the correct place in the representation of `DE`. Hence the simple implementation of `project` suffices here. The new value of `CurDe` could now be computed from `PrevDe`, but this would be inefficient, as `CurDe` contains already all needed equations with the exception of the prolongations of the integrability conditions. Here `complete` makes use of the second form of `prolong` which computes only these equations.

Besides `complete` `CK` contains several procedures for calculations with the Cartan characters and the Hilbert polynomial. Especially it allows the user to compute the $\alpha_q^{(k)}$

from the $\beta_q^{(k)}$ and to calculate $H_q(r)$ from the $\alpha_q^{(k)}$ and vice versa. There is a procedure to evaluate the recursion relation (3.16) for a given differentiation order. Furthermore one can compute gauge corrections.

The implementation of these operations is straightforward from the formulae of Chapter 3. It only demands a procedure `stirling` to compute the modified Stirling numbers introduced in Appendix C.1. They are determined using the recursion relation (C.5). To avoid repeated calculations of the same values, a hash table is set up storing all computed numbers.

`SYMANA` is still fairly primitive and in an experimental stage. It provides essentially one procedure `detSys` to set up the determining system (5.5) for symmetry generators (cf. Chapter 5). There exist different mode maps for this procedure. One can e.g. provide a special ansatz for the symmetry generator or a list of derivatives for which the equations of the system can be solved. The default is the most general ansatz and each equation will be solved for its leading derivative.

If the general ansatz is chosen, `detSys` returns the determining system in another data type, namely as functions over a new jet bundle with the coefficients as dependent variables. In this form the output can directly be converted into a `DifferentialEquation` and further analyzed. An example for this is given in Section 8.8. If one is searching for classical symmetries, it is of advantage to retract the output to a linear system using `linearize`.

In its current form `SYMANA` will have problems to compute the determining equations for a large system. Like all currently available symmetry packages it substitutes the equations of the original system solved for given derivatives in the conditions for the symmetry generators. Then it extracts in one loop the coefficients of all monomials. This approach is very inefficient and yields often many redundant equations. A later version will use a more sophisticated approach setting up the equations step by step and simplifying them at once.

Another project for the next version is to get rid of the restriction to equations which can be solved for some derivatives. If a Gröbner basis of the original system is given, then one can compute a normal form of all conditions and obtain this way independent of any solving the determining system. Note, that we use here standard algebraic Gröbner bases and not differential ones. This suffices, as we work in a jet bundle of fixed order.

`SYMANA` contains also a procedure `ncDetSys` to obtain the determining system for conditional symmetries as they are used in the non-classical method. It simply adds the invariant surface condition corresponding to the given ansatz for the symmetry generator to the original system and then calls `detSys`.

8.6 Implementation IV — Utilities

Since it is one of the basic ideas underlying the formal approach to differential equations to use linear algebra instead of differential operations, it is obvious that an efficient treatment of linear systems of equations is important for the performance especially of the `complete` procedure in `CK`. The determination of the $\beta_q^{(k)}$ requires to transform the symbol in row echelon form; some domains in `JBFC` may use the Jacobi matrix in `simplify` or `dimension`, too.

Earlier versions of our environment used the standard `Matrix` domain of AXIOM. However, the matrices typically occurring in our applications have some special properties which can be better exploited by implementing a new domain. The most important one is that they are sparse, often even very sparse. This reflects simply the already mentioned fact that e.g. in a second-order equation usually not all jet variables up to order two are effectively occurring. Often even of the second order derivatives just one or two are present. (2.11) implies furthermore that all prolonged symbols have exactly the same percentage of non-vanishing entries, i.e. only integrability conditions can change the sparsity.

In a recent diploma thesis Berchtold [10] compared different algorithms and data structures to compute row echelon forms and determinants of sparse matrices. He showed that for integer matrices it is of advantage to use a special "sparse" data structure like lists or polynomials for the rows of the matrix. His conclusion that this would not be the case for matrices with polynomial entries is, however, not correct and essentially due to his incorrect implementation.

Most of the matrices studied by Berchtold were still fairly dense compared with the matrices typically appearing as symbols. For such matrices Gaussian elimination amounts essentially to sorting the rows according to the position of the pivot, as the probability that two rows have their pivots in the same column is rather small.

Another result of Berchtold's thesis was that factor-free elimination methods like e.g. the Bareiss algorithm [57] are fairly inefficient. They tend to enlarge the entries considerably. In the case of a square matrix the last entry on the diagonal will contain in the end the complete determinant of the matrix! In our applications this can thoroughly effect the overall performance. Assume the `simplify` procedure of a domain in `JBFC` is based on such an elimination method. If later the resulting equations must be prolonged, these complicated expressions must be partially differentiated several times.

On the other hand divisions like they are used in the standard Gauss algorithm lead to a similar effect, as differentiation of fractions is quite expensive. It is therefore better to stay somewhat in the middle between these two extrema. The so-called primitive factor-free elimination (apparently first introduced by Monagan in Maple) divides each row by its greatest common divisor. This approach avoids collecting factors as in the Bareiss method but still does not generate fractions.

The greatest common divisor of a row can usually be computed fairly efficient, as experience shows that very often already the first two non-vanishing entries determine the final result. Actually there are two possible strategies: The classical one is compute the greatest common divisor iteratively stopping as soon as 1 is reached. Zippel [167] recommends a probabilistic strategy building to random linear combinations of the elements and computing their greatest common divisor. With probability 1 this yields the correct result. But of course one must nevertheless check that one has not obtain a too large result. Which strategy is ultimatively the best is difficult to judge. `SEM` offers both; the user can choose it with the function `setGcdMode`.

These considerations[1] lead to the implementation of the domain `SparseEchelon-Matrix (SEM)`. Its internal data structure is especially designed for the fast computation of row echelon forms. On the other hand hardly any arithmetic operations like addition or multiplication are implemented, as these tend to destroy the sparsity and are furthermore rather inefficient in this special representation.

`SEM` takes two arguments: a domain belonging to `Ring` for the entries and a domain belonging to `OrderedSet` to label the columns. A row is now represented by a record containing two lists: one for the non-vanishing entries, one for their labels. The latter one is always sorted. A matrix consists essentially of a vector of such records.

The central operations of `SEM` are `rowEchelon` and `primitiveRowEchelon`. Both return

[1] I am indebted here to M.B. Monagan who turned my attention to these problems and explained me his and Berchtold's results.

not only the matrix in row echelon form but also the corresponding transformation matrix, the used pivots and the rank of the matrix. `primitiveRowEchelon` is only available, if the domain of the entries belongs to the category `GcdDomain`. Furthermore, the transformation matrix contains then fractions from the divisions by the greatest common divisors.

The implementation of both operations is fairly different from the usual elimination methods. Instead of searching for a pivot in each column, we start by sorting all rows according to their pivots. After each elimination step changed rows are at once moved to their new places, so that the rows remain sorted all the time. The sorting is done using bubble sort. Since we can expect that our matrices are often already close to a row echelon form, this is probably the best choice. Furthermore it allows to keep easily track of the rows which is necessary for the construction of the transformation matrix.

The package `JetCoordinateTransformation` provides two procedures `transform` to prolong coordinate transformations of the base bundle $\mathcal{E}$ into higher order jet bundles. Its parameters are two jet bundles and two vectors (one for the independent and one for the dependent variables) containing expressions for the old coordinates in terms of the new ones. One procedure computes the transformation law for an old jet coordinate; the other one transforms an expression in the old coordinates using the first one and `subst`.

In the current implementation all expressions must be of the type `JBE`. The main reason for this restriction is that this represents by far the most flexible domain, as its representation is based on the domain `Expression Integer`. In it arbitrary transformation laws can be specified. Furthermore for most domains in `JBFC` it is possible to write a `retract` or `coerce` procedure to convert its objects into objects of `JBE` and vice versa.

The implementation of `transform` is straightforward. It makes use of the chain rule. [15] gives a simple recursive formula for the transformation law of a derivative of order q given the one for a derivative of one order less. It requires the inversion of the Jacobi matrix of the transformation of the independent variables. This inverse matrix is computed using an LU decomposition[2]. To enhance the efficiency transformed jet variables are stored in a hash table similar to `stirling` in `CK`.

8.7 Examples I — Completion

This and the next section demonstrate the application of the above described environment to concrete examples. Some of them have already been discussed in the previous chapters. The mentioned running times refer to the AXIOM implementation on an IBM RS 6000 Model 530 with 64MB memory.

To start we treat again the classical Example 2.19 of Janet. The upper part of Fig. 8.4 on page 137 shows the necessary input. As first step the different needed domains and packages are constructed and assigned to variables to facilitate the input. We use a jet bundle (`jb`) with 3 independent variables named `x` and one dependent variable named `u`. Derivatives are denoted by `p`. For the equations we use linear functions (`jbl`) whose coefficients can be arbitrary expressions in the independent variables (`jbx`). `de` is the data type used to represent the differential equation; `ck` denotes the package with the completion procedure.

[2]We have implemented for this purpose a package `LUDecomposition` containing three procedures: `LUDecomp` computes a LU decomposition using the algorithm of Crout [121]; `LUSolve` uses such a decomposition to solve a linear system with given right hand side; `LUInverse` determines the inverse for a matrix with given LU decomposition.

The equation is set up in two steps: First the two equations (`eq1,eq2`) of system (2.24)

$$\mathcal{R}_2 \,:\, \begin{cases} u_{zz} + y\, u_{xx} \;=\; 0\,, \\ u_{yy} \qquad\qquad = \; 0\,. \end{cases}$$

are defined. Jet variables are generated with the procedures `X, U, P`. Then a differential equation (`janet`) is generated from this system. The `setOutMode` command controls the amount of output generated and whether or not it should be in TeX. The `setRedMode` command tells `complete` to reduce the found integrability conditions as much as possible. Finally the procedure `complete` from `ck` is called. In the last two lines we have helped the interpreter to locate the used procedures by explicitly stating that they are from the package `ck`.

The output of `complete` is also contained in Fig. 8.4. With the chosen setting of the output flags we get a trace of the dimensions of all intermediate systems and symbols and of all found integrability conditions. The output is directly in TeX. One can easily see that the computation are exactly the same as in our treatment of the Janet example in Section 2.4. It is, however, a non-trivial fact that the program produces the integrability conditions in the simple form shown in Fig. 8.4. Actually many more integrability conditions are produced, for there occur quite a lot of non-multiplicative variables in the higher prolongations. They are, however, not independent. `complete` must invoke several simplification procedures to avoid redundant equations and unnecessary coefficients. In this example it succeeds in reaching the simplest form, because we deal with a linear system. For non-linear equations this is not necessarily the case.

The running time for this example is about 18 sec. It rises to 28 sec, if one takes `JBE` as data type for the equations instead of `JBLF`. There are two main reasons for this striking difference: differentiations are much faster in `JBLF` and simplification of linear systems can be performed more efficient. Although in the end the `simplify` procedure of `JBE` produces the same result as the one in `JBLF`, the latter one finds the necessary steps easier and thus faster.

Due to the many prolongations necessary in this example it also demonstrates very nicely the advantages of our special domain for sparse matrices. Compared with an earlier version of our environment which used the standard `Matrix` domain of AXIOM the mentioned running time improved by more than a factor two. It is easy to see that system (2.24) leads indeed to very sparse matrices for the symbols. Furthermore the procedure `simplify` in `JBLF` computes already a row echelon form of the whole system and thus of course also of the symbol. Hence the call of `rowEchelon` in `analyseSymbol` is in principle redundant. But because of the special way the elimination algorithm is implemented this takes hardly any computing time.

One could think that this example should be an ideal test case for the lazy differentiation scheme implemented in `JLF`. However, the result is disappointing. Instead of accelerating the computation the running time rises to almost 1 min. A closer look solves the mystery. We encounter three times a non-involutive symbol. Each time two prolongations are necessary. Tracing the intermediate systems shows that after the second prolongation all lazy differentiations used in the previous one have been fully evaluated.

The reason is that there occur many non-multiplicative variables in the equations of higher order and deciding whether or not the corresponding prolongations generated integrability conditions requires the evaluation of most of the equations. Thus in this example it is indeed necessary to compute almost all prolongations. Another aspect of this at first sight perhaps surprising performance is that the basic assumption for the development of `JLF`, namely that partial differentiations are expensive, is not correct for linear equations. With the chosen representation of JBLF the differentiation with respect to a dependent variable or a derivative requires only the extraction of an element of a list. Since the lists are not too long, this is very fast.

As an example of a non-linear system we treat the Euler Equation of hydrodynamics in a slightly unusual way to obtain an over-determined system.

$$\begin{aligned} &\partial_t u^i + \sum_{j=1}^{D} u^j \partial_{x^j} u^i = 0\,, \qquad i = 1, \dots, D\,, \\ &\sum_{j=1}^{D} \partial_{x^j} u^j = 0\,, \end{aligned} \tag{8.1}$$

where D denotes the space dimension. In our computer calculation we take the usual value of $D = 3$ and write x^4 instead of t.

In- and output are shown in Fig. 8.5 on page 138. The input is very similar to the previous example. The main difference is that we use another domain for the equations, namely JBE. They can now be arbitrary expressions in the independent and dependent variables. The integrability condition can be obtained by taking the divergence of the first D equations and subtracting the t-derivative of the last equation

$$\sum_{i,j=1}^{D} \partial_{x^i} u^j \partial_{x^j} u^i = 0\,. \tag{8.2}$$

Due to the simplification procedures this equation and some of the original equations appear different in the output. As explained in Section 8.4 the `simplify` procedure in JBE tries to solve equations for their leading derivatives and to substitute them into the other ones. If this strategy is successful, it leads to kind of normal form of the system which is better adapted for the analysis of the symbol.

The time used for simplification also dominates the running time. Although this example is much smaller than the Janet example in terms of the number of prolongations needed, it takes much longer, namely about 8 min. This time is really surprising, as the simplification does not require the computation of Gröbner bases. Nevertheless, it is frequently necessary to solve equations for their leading derivatives and to substitute the result in other equations. This process is quite time-consuming in JBE.

We have used our basic domains for jet bundles in these two examples: in the Janet example JB, for the Euler Equation IJB. One could see that the input of the equation was slightly more complicated for JB, because the interpreter needs some help in order to identify the types correctly. One can of course also use the syntax applied for the Euler Equation in JB. But this requires that one knows the used ranking of the variables. The purpose of JB is, however, to stay as close as possible to the natural notation.

The two examples differ also in the setting of the reduction and simplification control flags with `setRedMode` and `setSimpMode`. The choice of a good setting requires in some sense to know already a priori what will happen during the completion. In principle one would prefer to avoid the setting of both flags, as they only need computation time.

`setRedMode(1)` affects basically only the output. Without it many spurious integrability conditions and/or equations in the generating system displayed at the end may appear. They are either algebraically dependent of each other or derivatives of some other equations. Reduction tries to detect this. In examples like the Janet example where many prolongations are needed and integrability conditions appear at different orders, this is usually useful. In the case of the Euler Equation one would, however, gain nothing.

Deciding whether `setSimpMode(1)` helps or not is more difficult. As explained in Section 8.5 the individual equations of a system are stored order by order. With the default setting simplification only occurs within the different orders. Equations of lower order are not used. This may lead to more complicated expressions, because some simplifications are overlooked. For instance it might happen in the Janet example that some equations of higher order contain u_{yy}, although it is zero. But it turns out that in this example it is more expensive to search for such simplifications than to continue with the unsimplified expressions.

The situation is often different for non-linear systems. Here it is much more important to avoid complicated expressions, as otherwise the standard simplification within one order needs much more time. In the case of the Euler Equation one gains almost a factor of two through simplifying modulo the equations of lower order.

To stress this point we will briefly discuss a non-linear modification of the Janet example. Consider the system

$$\mathcal{R}_2 \; : \; \begin{cases} u_{zz} + v\,u_{xx} &= 0\,, \\ u_{yy} &= 0\,, \\ v_z &= 0\,, \\ v_y - v_x &= 0\,. \end{cases} \tag{8.3}$$

The completion runs completely analogously to the classical Janet example, i.e. the projections occur at the same places and $\mathcal{R}_5^{(2)}$ is involutive. The only difference is that in the second projection two integrability conditions[3] occur, thus there are three:

$$\begin{gathered} 2v_x u_{yxx} + v_{xx}u_{xx} = 0\,, \\ 4v_x^4 u_{xxxx} + v_{xx}(4v_x^2 - vv_{xx})(2v_x u_{xxx} + v_{xx}u_{xx}) = 0\,, \\ 2v_x v_{xxx} - 3v_{xx}^2 = 0\,. \end{gathered} \tag{8.4}$$

It is obvious that here the simplification routines have hard work to do. Indeed the running time for this example is slightly more than one hour. It was not possible to run it without simplification modulo lower order equations. We stopped the computation after about 10 hours!

We showed in Chapter 6 that the Dirac algorithm does nothing else than to complete the equations of motion to involution. Thus we can also use our environment for the analysis of constrained dynamics. Tombal and Moussiaux [154] presented a MACSYMA program for the Dirac algorithm. It can, however, handle only Lagrangians which are quadratic in the velocities. Furthermore they reported that their program was too slow to handle complicated systems like the Maxwell Equations. Our environment is not only capable of treating much more general systems, it proves the involution of the Maxwell Equation in a four-dimensional space-time in about 15sec.

As a simple example we will consider again the motion of a particle in three dimensions whose motion is confined to the 2-sphere of radius 1 (cf. Example 6.5). Setting all masses to 1 we have to analyze the following equations of motion

$$\mathcal{R}_2 \; : \; \begin{cases} \ddot{x} - 2\lambda x = 0\,, \\ \ddot{y} - 2\lambda y = 0\,, \\ \ddot{z} - 2\lambda z = 0\,, \\ x^2 + y^2 + z^2 - 1 = 0\,. \end{cases} \tag{8.5}$$

[3] We count here only the non-trivial conditions. Of course there appear in addition the prolongations of the two first-order equations.

The output of a corresponding AXIOM session is shown in Fig. 8.6 on page 139. λ is denoted by L. Since we are dealing with ordinary differential equations, there is no need to check for involution of the symbol. The running time is slightly over 30sec with both simplification and reduction flag set.

It might surprise that such a small example consisting only of ordinary differential equations should take twice as long as the Maxwell Equations. Especially if one takes into account that no symbols must be analyzed. But the reason is simply that the latter ones are linear. If we omit setting the simplification flag, i.e. `setSimpMode(0)`, the running time nearly doubles.

A closer analysis of the obtained integrability conditions and the displayed final system reveals some deficiencies in the treatment of non-linear equations. We will discuss them later in more detail. Here we just want to point out that in several equations the constraint was not used for simplification; in the third projection we even obtain two identical equations. Both effects are mainly due to current restrictions in the `solveFor` procedure of `JBE`.

8.8 Examples II — Other Applications

One of the classical examples in symmetry analysis is the Heat Equation. We will use `SYMANA` to calculate its determining system for classical Lie symmetries. We omit the further discussion of the symmetries, as it can be found in many places. The output of a corresponding AXIOM session is contained in Fig. 8.7 on page 140.

The first command just shows the ansatz made for the symmetry generators. If `detSys` receives only one argument, it takes automatically this most general ansatz, where every coefficients depends on all variables, i.e. in this example on x, t, u. The type of the output of `detSys` depends on the input. If one uses the general ansatz as in Fig. 8.7, one obtains a list of functions of type `JBE` over a new jet bundle in which the coefficients of the ansatz are the dependent variables. Thus the result can be used directly to generate a differential equation and then be further analyzed.

Since we are only looking for classical symmetries, we know that the determining system must be linear. Hence before generating a differential equation the individual equations are retracted to a new type, namely `JBLF`. Then the completion procedure is started. It turns out that we must proceed to third order and that four projections are necessary. The resulting equation can easily be solved explicitly (see e.g. [15, 111]).

Here we are only interested in calculating the size of the symmetry algebra. From the displayed Cartan characters we see that the general solution of the determining system contains two functions of one argument. But this is due to the linearity of the equation: solutions can be superposed. Indeed, we see that one of the equations in the determining system is just the Heat Equation itself.

In general it is difficult to deduce from the knowledge of the Cartan characters and of the dimension of the differential equation the number of arbitrary constants in the general solution, as one must take the lower order part of the arbitrary functions into account. These are usually not independent and thus difficult to count. The only way to do it is to construct a formally well-posed initial value problem. Here we can proceed simpler. The Heat Equation yields a seven-dimensional submanifold in a third-order jet bundle with two independent and one dependent variables. Hence its general solution contains seven arbitrary coefficients of order less than or equal to 3. Since the determining system is 13-dimensional, there remain 6 arbitrary constants for the finite-dimensional part of the symmetry algebra. Thus we find besides the superposition 6 linearly independent symmetry generators.

Setting up the determining system needs about 12 sec. The completion is, however, fairly slow and takes a bit longer than 90 sec. This is mainly due to the prolongation to third order. One should mention that the Heat Equation is in some respect a "lucky" example. Fairly often determining systems turn out to be in not δ-regular coordinate system. Then one cannot simply call `complete` but must start working with the k-tableaux.

Finally, we consider an example of a coordinate transformation. It is well-known that the Burgers Equation is related to the Heat Equation via the Cole-Hopf transformation. We will perform this transformation using the AXIOM package `JCT` starting with the Burgers Equation in potential form. The result can be seen in Fig. 8.3.

```
burgers:jbe1 := P [2] - P [1,1] - P([1])**2

                           2
   (8)  - u     + u  - u
           x,x     t    x
                                Time: 0.33 (IN) + 0.23 (EV) + 0.80 (OT) = 1.36 sec
transform(burgers)$jct

        - v     + v
           y,y     s
   (9)  -----------
             v
                                Time: 0.22 (IN) + 2.02 (EV) + 1.54 (OT) = 3.78 sec
transform(burgers)$jct

         - v     + v
            y,y     s
   (10)  -----------
              v
                                Time: 0.01 (IN) + 0.35 (EV) + 0.05 (OT) = 0.41 sec
```

Figure 8.3: Cole-Hopf transformation of the Burgers Equation.

The timing of the first transformation also contains the time needed for the precomputation of an inverse Jacobi matrix. Since its result is stored, subsequent transformations will be faster. In this example this effect is neclectable, as the independent variables remain unchanged and the mentioned Jacobi matrix measures their dependence of the old independent variables. `JCT` further keeps as hash table of already treated jet variables. Hence the second call with the same argument is much faster.

8.9 Outlook and Discussion

It should be clear from the discussion so far, that this environment is by far not yet finished. There are permanent changes and improvements. Most changes are needed in the domains for non-linear equations. The simplification and reduction procedures implemented in `JBE` are still too inefficient for more complicated equations. This is partly due to its current role as a substitute for a special domain for polynomial equations. Although the procedures are mainly designed for the application to polynomial equations, there had to be made concessions for more general expressions.

The most important point is reduction. In a polynomial domain one could make full use of the auto reduction algorithms proposed already by Ritt [128]. Currently `JBE` inherits the default implementation of `autoReduce` in `JBFC` which tries to solve equations for their leading derivatives and to substitute the results in the other equations. This can be very time consuming for complicated equations. Similar remarks can be made for simplification.

This effect is clearly visible in the example of the Euler Equation. Although the completion algorithm terminates after just one iteration, the computing time is much higher than in the Janet examples with its many prolongations and projections. This is especially remarkable, as a closer trace reveals that no Gröbner basis had to be calculated. If one looks at the second equation of the final result in Fig. 8.5 on page 138, one remarks that it has become considerably more complicated. This stems from “simplification”!

Even worse things are happening in the completion of the constrained system (8.5) where not only the constraint was several times not used for apparently easy simplifications and in one projection two identical integrability conditions were shown. The first effect will disappear as soon as a truly polynomial domain will be implemented. The current simplification and reduction routines rely heavily on the ability of `solveFor`. It in turn cannot treat products, if several factors contain jet variables. Since it is possible that one factor might be zero, `solveFor` returns in such cases `"failed"` to omit wrong results in the simplification.

Another point which will have to be addressed in further versions is regularity. As soon as we treat non-linear equations ranks and characters may differ on different points of a differential equation (cf. Example 2.2). If this happens, case distinctions are necessary. It is probably not efficient to treat automatically all occurring cases, as branching can occur in each step of the completion. But the output of a procedure like `complete` should at least show where problems can arise. Currently it follows just one branch without specifying the conditions.

To detect non-regularity one must inspect pivots. If a prolonged equation is not a manifold, then its Jacobi matrix has different ranks on different parts of it. The determination of a row echelon form, say for a symbol, makes usually assumptions that certain expressions, namely the used pivots, are non-vanishing. The procedures for row echelon forms in `SEM` always return the pivots; however, the higher level domains and packages like `DE` or `CK` make currently no use of this information.

Sit [144] developed an algorithm which determines all necessary case distinctions in determining the rank of a symbolic matrix. It is based on the primary decomposition of algebraic varieties. He argued that an analysis based on pivots may lead to many unnecessary branches. While this is surely correct from a theoretical point of view, his method is probably too expensive to be applied automatically in an already expensive completion algorithm. Imagine all matrices occurring in the completion of the modified Janet example in Section 8.7 would be analyzed this way!

Besides simplification and reduction δ-regularity provides the most trouble. Currently `complete` in `CartanKuranishi` can handle only equations given in a δ-regular coordinate system. Otherwise it will prolong and prolong and prolong without ever finding an involutive symbol. In principle one could implement the method present in Section 2.7 using k-tableaux. We have refrained from this approach, because it becomes quickly computationally very demanding.

Computing row echelon forms of symbolic matrices is in general very expensive. In computer algebra a matrix is considered as large, if it has more than, say, 20 rows and columns (of course this depends on the complexity of the entries). For the symbol we could partially remedy this by using a special domain for sparse matrices. This allows to handle efficiently considerably larger matrices. In the Janet example one encounters e.g. matrices with more than 60 rows. The tableaux are, however, always dense. One reaches quickly machine limits, if several prolongations are needed or the system is large.

Often one can detect already by inspection that a coordinate system is not δ-regular. One possible strategy is to use the tableaux to find a non-systatic basis of T^*X and an associated coordinate system. Then one rewrites the equation in this system and starts the completion algorithm. Of course it can always happen that the coordinate system will become again singular after some projections, but often this approach succeeds.

Our symmetry package is the first one developed in AXIOM. The REDUCE package SPDE [134] was translated [136] to its predecessor system Scratchpad II. It became, however, soon obsolete as new versions of Scratchpad II and later AXIOM appeared. `SYMANA` is still in a very early and experimental stage. The evaluation of the condition for symmetry generators is primitive compared with more sophisticated approaches, as they are e.g. described in [29].

Nevertheless, it demonstrates clearly the advantages of implementing a general environment: the basic procedure for setting up the determining equations consists of about 15 lines! Everything else is essentially provided by other domains. The many simplification and reduction routines originally implemented to support the completion procedure in `CK` are very useful to simplify determining systems. By using different domains in `JBFC` different simplification strategies can be applied.

We believe that, although there exist meanwhile so many symmetry packages, it is still possible to make significant progress in this field, even if one restricts ones attention to classical Lie symmetries (the extensions to more general symmetries are fairly straightforward anyway). We have already mentioned the possibility to handle equations which cannot be solved for some derivatives with Gröbner bases. One will have to wait whether it will be possible to provide an efficient implementation of this approach.

Another possibility for improvements is to make use of theoretical results on the form of admissible symmetry generators. Bluman [13] showed that at least for semi-linear equations one can simplify the ansatz for the generators considerably, because the coefficients' dependency of the dependent variable is very often trivial. This can save considerable computation time. First the prolongation of the generator is simpler and will lead to fewer terms, hence the resulting determining system is simpler. A lot of the work done by the packages to solve determining systems is just to rediscover through lengthy computations these simplifications!

```
jb:=JB(['x,'y,'z],['u])
jbx:=JBX jb
jbl:=JBLF(jb,jbx)
de:=DE(jb,jbl)
ck:=CK(jb,jbl)

eq1 := D('u,['z,'z])$jb::jbl + 'y::jb::jbl * D('u,['x,'x])$jb::jbl
eq2 := D('u,['y,'y])$jb::jbl
janet:de := generate [eq1,eq2]

setOutMode(14)$ck
setRedMode(1)$ck
complete janet
```

Symbol M_2 not involutive! Dimension: 4

Symbol M_3 involutive! Dimension: 4

Equation R_3 not involutive! Dimension: 12

======= 1. Projection =======

Integrability condition(s)

$$u_{y,\,x,\,x} = 0$$

==============================

Symbol $M_3^{(1)}$ not involutive! Dimension: 3

Symbol $M_4^{(1)}$ involutive! Dimension: 2

Equation $R_4^{(1)}$ not involutive! Dimension: 13

======= 2. Projection =======

Integrability condition(s)

$$u_{x,\,x,\,x,\,x} = 0$$

==============================

Symbol $M_4^{(2)}$ not involutive! Dimension: 1

Symbol $M_5^{(2)}$ involutive! Dimension: 0

*************** Final Result ***************

Equation $R_5^{(2)}$ involutive!

System without prolonged equations. Dimension: 12

$$\begin{gathered} u_{x,\,x,\,x,\,x} = 0, \\ u_{y,\,x,\,x} = 0, \\ u_{z,\,z} + y\ u_{x,\,x} = 0, \\ u_{y,\,y} = 0 \end{gathered}$$

System of finite type.

Figure 8.4: In- and output for the Janet example (2.19).

```
jb:=IJB('x,'u,'p,4,3)
jbe:=JBE jb
de:=DE(jb,jbe)
ck:=CK(jb,jbe)

eq1:jbe := P(1,[4]) + U(1)*P(1,[1]) + U(2)*P(1,[2]) + U(3)*P(1,[3])
eq2:jbe := P(2,[4]) + U(1)*P(2,[1]) + U(2)*P(2,[2]) + U(3)*P(2,[3])
eq3:jbe := P(3,[4]) + U(1)*P(3,[1]) + U(2)*P(3,[2]) + U(3)*P(3,[3])
eq4:jbe := P(1,[1]) + P(2,[2]) + P(3,[3])
euler:de := generate [eq1,eq2,eq3,eq4]

setOutMode(14)$ck
setSimpMode(1)$ck
complete euler
```

Symbol M_1 involutive! Dimension: 8

Equation R_1 not involutive! Dimension: 11

======= 1. Projection =======

Integrability condition(s)

$$2\ p_2^3\ p_3^2 + 2\ p_1^3\ p_3^1 + 2\ {p_2^2}^2 + 2\ p_1^1\ p_2^2 + 2\ p_1^2\ p_2^1 + 2\ {p_1^1}^2 = 0$$

=============================

Symbol $M_1^{(1)}$ involutive! Dimension: 7

*************** Final Result ***************

Equation $R_1^{(1)}$ involutive!

System without prolonged equations. Dimension: 14

$$\begin{gathered}
p_4^3 + u^2\ p_2^3 - u^3\ p_2^2 + u^1\ p_1^3 - u^3\ p_1^1 = 0, \\
p_4^2 - u^3\ p_1^3\ p_3^1 + \left(u^2\ p_2^2 + u^1\ p_1^2\right)\ p_2^3 - u^3\ {p_2^2}^2 - u^3\ p_1^1\ p_2^2 - u^3\ p_1^2\ p_2^1 - u^3\ {p_1^1}^2 = 0, \\
p_4^1 + u^3\ p_3^1 + u^2\ p_2^1 + u^1\ p_1^1 = 0, \\
p_3^3 + p_2^2 + p_1^1 = 0, \\
p_2^3\ p_3^2 + p_1^3\ p_3^1 + {p_2^2}^2 + p_1^1\ p_2^2 + p_1^2\ p_2^1 + {p_1^1}^2 = 0
\end{gathered}$$

Cartan characters: 3, 3, 1, 0

Figure 8.5: In- and output for the Euler Equation (8.1).

Equation R_2 not involutive! Dimension: 8

======= 1. Projection =======

Integrability condition(s)

$$2\ z\ z_t + 2\ y\ y_t + 2\ x\ x_t = 0$$

===============================

Equation $R_2^{(1)}$ not involutive! Dimension: 7

======= 2. Projection =======

Integrability condition(s)

$$\left(2\ z^2 + 2\ y^2\right)\ {y_t}^2 + 4\ x\ y\ x_t\ y_t + \left(2\ z^2 + 2\ x^2\right)\ {x_t}^2 + 4\ L\ z^4 + \left(4\ L\ y^2 + 4\ L\ x^2\right)\ z^2 = 0$$

===============================

Equation $R_2^{(2)}$ not involutive! Dimension: 6

======= 3. Projection =======

Integrability condition(s)

$$\begin{gathered} 4\ L_t\ z^2 + \left(4\ y^2 + 4\ x^2\right)\ L_t = 0, \\ 4\ L_t\ z^2 + \left(4\ y^2 + 4\ x^2\right)\ L_t = 0 \end{gathered}$$

===============================

Equation $R_2^{(3)}$ not involutive! Dimension: 5

======= 4. Projection =======

Integrability condition(s)

$$\left(4\ z^2 + 4\ y^2 + 4\ x^2\right)\ L_{t,\,t} = 0$$

===============================

************** Final Result **************

Equation $R_2^{(4)}$ involutive!

System without prolonged equations. Dimension: 4

$$\begin{gathered} y_{t,\,t} - 2\ L\ y = 0, \\ x_{t,\,t} - 2\ L\ x = 0, \\ \left(4\ z^2 + 4\ y^2 + 4\ x^2\right)\ L_{t,\,t} = 0, \\ z\ z_t + y\ y_t + x\ x_t = 0, \\ \left(2\ z^2 + 2\ y^2\right)\ {y_t}^2 + 4\ x\ y\ x_t\ y_t + \left(2\ z^2 + 2\ x^2\right)\ {x_t}^2 + 4\ L\ z^4 + \left(4\ L\ y^2 + 4\ L\ x^2\right)\ z^2 = 0, \\ \left(-z^2 - y^2\right)\ {y_t}^2 - 2\ x\ y\ x_t\ y_t + \left(-z^2 - x^2\right)\ {x_t}^2 - 2\ L\ z^4 + \left(-2\ L\ y^2 - 2\ L\ x^2\right)\ z^2 = 0, \\ 4\ L_t\ z^2 + \left(4\ y^2 + 4\ x^2\right)\ L_t = 0, \\ 4\ L_t\ z^2 + \left(4\ y^2 + 4\ x^2\right)\ L_t = 0, \\ z^2 + y^2 + x^2 - 1 = 0 \end{gathered}$$

System of finite type.

Figure 8.6: Output for the constrained system (8.5).

```
v:vf := ansatz()$sym

   tau D  + xi D  + eta D
        t       x        u

ds := detSys([eq])$sym

   [- 2tau , - 2tau , - xi   , - tau   , eta    - 2xi   , - 2tau    - 2xi ,
          u        x      u,u       u,u     u,u      u,x        u,x      u
    2eta    - xi    + xi , - tau    - 2xi  + tau , eta    - eta ]
        u,x     x,x     t       x,x      x      t     x,x      t

lds:List jbl2 := [retract(eq)  for eq in ds]
r2:de := generate lds
setOutMode(4)$ck
setRedMode(1)$ck
complete r2

   *************** Final Result ***************

            (4)
   Equation R     involutive!
             3
   System without prolonged equations. Dimension: 13

       tau     = 0
          t,t,t
        eta   = 0
           u,u
            1
    eta    + - xi = 0
       u,x   2   t
     eta    - eta = 0
        x,x      t
           1
   eta    + - tau   = 0
      u,t   4    t,t
         xi   = 0
           t,t
          xi = 0
            u
         tau = 0
            u
           1
     xi  - - tau = 0
       x   2    t
         tau = 0
            x

   Cartan characters: 2,0,0
```

Figure 8.7: Determining system for the Heat Equation.