Algorithms and Implementations for Differential Elimination

by

Allan Wittkopf

M.Sc., Simon Fraser University, 1994

B.Sc., Okanagan University–College, 1991

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
in the Department
of
Mathematics

© Allan Wittkopf 2004
SIMON FRASER UNIVERSITY
Fall 2004

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

APPROVAL

Name: Allan Wittkopf

Degree: Doctor of Philosophy

Title of thesis: Algorithms and Implementations for Differential Elimination

Examining Committee: Dr. Imin Chen
Chair

Dr. Michael Monagan
Senior Supervisor

Dr. Gregory Reid
Senior Supervisor

Dr. Robert Russell

Dr. Nils Bruin
Internal Examiner

Dr. Xiao-Shan Gao
External Examiner

Date Approved:

Abstract

The primary focus of this work is the design and implementation of efficient *differential elimination* algorithms. Such algorithms use a finite number of differentiations and eliminations to simplify over-determined systems of ordinary and partial differential equations (ODE and PDE) to a more tractable form. They can be used in exact solution methods for ODE and PDE systems, as a preprocessor for numerical solution of these systems, and for reduction of nonlinear PDE to linear PDE.

Differential elimination algorithms have been implemented to date in a number of symbolic languages, and although these algorithms are finite in theory, in practice, their complexity puts many interesting problems out of reach.

We enlarge the class of problems which can be *completed*, by which we mean simplified to a form satisfying certain theoretical properties (canonical form in the linear case, and a form that yields an existence and uniqueness theorem in the nonlinear case). Additionally we provide means of obtaining partial information (in some cases the most relevant information) for many problems that cannot be completed.

Differential elimination relies heavily on other algorithms, such as computation of multivariate polynomial greatest common divisors (GCDs). As a result, significant contributions have also been made in this area. These include enhanced versions of known algorithms for computing multivariate polynomial GCDs for both dense (many terms for the given degree) and sparse (few terms for the given degree) polynomials.

The differential elimination algorithms have been implemented in the symbolic mathematics language *Maple*, and one in the compiled language C. We demonstrate their effectiveness on problems from symmetry analysis. The GCD algorithms have been implemented in *Maple*, and we provide a detailed asymptotic comparison of these algorithms with *Maple*'s primary GCD algorithm, and a well known algorithm for dense polynomial problems.

Acknowledgements

I am deeply indebted to Dr. Gregory Reid for his guidance and encouragement, and for our collaboration in the fourteen years we have researched and designed algorithms together.

I am also deeply indebted to Dr. Michael Monagan for his guidance, encouragement, and well communicated insight into efficient algorithm design and implementation.

I am also grateful to Dr. Edgardo Cheb-Terrab for his excellent advice on the direction of the *rifsimp* implementation (even though we butted heads frequently), and his assistance in making it a solid and established part of *Maple*, Dr. Laurent Bernardin for his support and motivation for completion of this project, and for the work of Dr. Colin Rust, which formed a solid theoretical basis on which I was able to build.

I would like to thank my many teachers over the years, in particular Mr. Vogt and Mr. Sowinski for kindling my interest in the sciences, Dr. Blair Spearman whose interest in the area of Mathematics cannot help but be infectious, and Dr. Tao Tang and Dr. Bob Russell for their encouragement of and interest in my academic career.

Thanks also go to Mrs. Sylvia Holmes, Mrs. Diane Pogue, and Mrs. Maggie Fankboner for their assistance on the administrative end of things.

On a personal note, I'd like to thank my parents, Doug and Rita Wittkopf, who always made me feel that I could accomplish anything I put my mind to, and my uncle Ron Wittkopf who shared my interest in computing.

And last, but most definitely not least, I am grateful for the patience, support and love of my wife Robyn throughout the years this work has spanned.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The primary research area of this dissertation is symbolic computation, specifically symbolic simplification of over-determined systems of ordinary differential equations (ODE) and partial differential equations (PDE), otherwise known as *differential elimination*. The idea here is to transform these systems into forms from which their solutions may be more easily determined or certain properties of these systems may be more easily obtained. There are many applications for which differential elimination plays a key role. Two prominent examples include computation of symbolic solutions for ODE, and transformation of nonlinear PDE into linear PDE.

Our introduction is informal, presenting examples to introduce concepts described rigorously later in the Thesis. Note, for brevity we will simply refer to PDE, noting the information here applies to ODE equally. In addition, unless stated otherwise, when we refer to a *system* or a *polynomial system* we mean a system of PDE.

Differential elimination algorithms apply a finite number of differentiations and eliminations to the equations in a system of PDE to transform them to a more desirable form. Before discussing these algorithms in detail, it is necessary to lay some foundations for the area.

We begin with a discussion of functions and derivatives, describing the notation used throughout the dissertation, and the class of problems we consider (namely polynomially nonlinear PDE systems). We discuss *rankings* of functions and derivatives, which just like the ordering of unknowns (columns) in Gaussian elimination, is fundamental to the process of differential elimination. We describe *differential reduction*, which is the process of reducing (simplifying) one PDE with respect to another. We discuss another process that is fundamental to differential elimination, which is the computation of *integrability conditions* (consistency conditions). Such conditions can be viewed as a differential analog of S-polynomials from Gröbner basis computations. Nonlinear problems naturally lead to *case splitting* (for example, $u_x(u_x - 1) = 0$ implies that either $u_x = 0$ or $u_x = 1$), so this is described as well. These concepts are described in greater detail by Reid *et al.* [50], [54] and Rust *et al.* [59], [58].

An important application area for differential elimination algorithms is determination of symmetries of ODE and PDE systems (see Chapter 5). These symmetries, or transformations leaving invariant a system of ODE or PDE are not known a priori, and have to be determined. The symmetries for a PDE system with independent variables $x = (x_1, x_2, ..., x_n)$, and dependent variables $u = (u^1, u^2, ..., u^m)$ are given by the transformations $X(x, u)$, $U(x, u)$ in

$$(x, u) \mapsto (\hat{x}, \hat{u}) = (X(x, u), U(x, u))$$

such that the equations of the system remain the same.

Symmetry analysis has proved to be important in the exact analysis of differential equations (e.g. determination of exact solutions, mappings from intractable to tractable equations, equivalence properties, integrability properties, see Olver [47] and Bluman and Kumei [3]).

Given a differential equation, determination of the symmetries requires the analysis of an associated system of PDE (the symmetry determining system) for the linearized transformation $(\hat{x}, \hat{u}) = (X(x, u), U(x, u)) = (x + \xi(x, u)\epsilon + \mathcal{O}(\epsilon^2), u + \eta(x, u)\epsilon + \mathcal{O}(\epsilon^2))$. Most importantly for us, this associated system is usually over-determined, and in the applications in this thesis can even contain hundreds of PDE. This is an ideal application for differential elimination algorithms.

Consider the following example.

Example 1 *Consider the* ODE $u_{xx} = u^2$. *The over-determined system of* PDE *for the infinitesimal symmetries* $(x, u) \mapsto (\hat{x}, \hat{u}) = (x + \xi(x, u)\epsilon + \mathcal{O}(\epsilon^2), u + \eta(x, u)\epsilon + \mathcal{O}(\epsilon^2))$ *can* *be algorithmically obtained from the* ODE *and is given by:*

$$\begin{aligned} \xi_{uu} = 0, &\qquad -3u^2\xi_u + 2\eta_{ux} - \xi_{xx} = 0, \\ \eta_{uu} - 2\xi_{ux} = 0, &\quad -2\eta u - 2u^2\xi_x + u^2\eta_u + \eta_{xx} = 0. \end{aligned} \tag{1.1}$$

The simplified form produced by differential elimination algorithms (such as the RifSimp and DiffElim algorithms, Mansfield's algorithm [37, 38], and Boulier's algorithm [6]) subject to the ranking $\eta \prec \xi \prec \eta_u \prec \xi_u \prec \eta_x \prec \xi_x \prec \ldots$ *is:*

$$\eta_x = 0, \quad \xi_x = -\frac{\eta}{2u}, \quad \eta_u = \frac{\eta}{u}, \quad \xi_u = 0\,.$$

Note that each equation in the output form is isolated for the highest ranked derivative. In fact, for the linear case, the output form is canonical (just like reduced Gaussian form). This result was obtained from the original system (1.1) using only differentiation and elimination. The number of arbitrary constants in the solution (the initial data*), which is also the dimension of the symmetry group of the* ODE*, can be algorithmically determined from the simplified form.*

Not only can the dimension of the solution be determined (which is two), but also the structure constants of the Lie algebra of the symmetry group, and the formal power series solution for the symmetries to any finite order (Reid et al. [52, 50]). While this example can easily be completed by hand, the algorithms for determining the dimension and structure of unknown symmetry groups of differential equations are guaranteed to determine this information in a finite number of steps, regardless of the complexity of the determining system.

A brief example-based introduction to differential elimination is given in this chapter, and many other examples introducing these concepts can be found in Appendix A, including nonlinear examples in §A.8,§A.9. Those looking for more formal or more detailed treatments should consult Boulier *et al.* [6], or Rust *et al.* [58, 60].

1.1 Functions and Derivatives

In this dissertation we restrict to systems which are polynomially nonlinear in the independent variables, the dependent variables, and all derivatives in the system. We'd like to note that use of *differential extensions* can extend the applicability of the results to systems that are not polynomially nonlinear in the independent variables (e.g. for a system containing $\mathbf{e}^x$, we could replace this with $f(x)$ and append the differential equation $f_x = f$ to the system). In some cases we utilize results that apply to much more general classes of problems, such as results from Rust [58] (see §2.2) that apply to problems that are analytic in the derivatives and independent variables, applying them to the polynomial class we consider. The equations of these systems are called *differential polynomials*, and belong to a *differential polynomial ring*, which can simply be viewed as a polynomial ring over all variables and derivatives of the system combined with a finite number of commuting differentiation operations (i.e. with respect to each independent variable).

The most basic step in the theory of differential elimination algorithms is to replace differential equations by algebraic ones, by regarding derivatives as formal variables. For example, the differential equations on the left below, are replaced by the formal expressions on the right

$$\begin{aligned} \frac{\partial \Psi(x,y)}{\partial x} - y\Psi(x,y) = 0 \quad &\leftrightarrow \quad u_x - yu = 0, \\ \frac{\partial \Psi(x,y)}{\partial y} - \Psi(x,y) = 0 \quad &\leftrightarrow \quad u_y - u = 0, \end{aligned}$$

where u_x is then treated as an indeterminate, rather than the derivative of a function with respect to x. The differentiations of the formal expressions are carried out using the formal commuting total derivatives

$$D_x := \partial_x + u_x\partial_u + u_{xx}\partial_{u_x} + ..., \quad D_y := \partial_y + u_y\partial_u + u_{xy}\partial_{u_x} + ... \,. \tag{1.2}$$

From the two equations (1.2) and this definition we have

$$D_y(u_x - yu) - D_x(u_y - u) = -yu_y - u + u_x \longmapsto -y\ (u) - u + (yu) = -u,$$

where we have used the symbol $\longmapsto$ to represent the substitution from $u_x - yu = 0, u_y - u = 0$ for u_x, u_y (or the *reduction* with respect to $u_x - yu = 0, u_y - u = 0$).

Consider a polynomially nonlinear qth order differential system with n independent variables $x = (x_1, x_2, ..., x_n)$ and m dependent variables $(\Psi^1(x), \Psi^2(x), ..., \Psi^m(x))$. Extending the correspondence above to the general case, the formal indeterminate u^l corresponds to $\Psi^l(x)$, $u^l_{x_j}$ corresponds to $\frac{\partial \Psi^l(x)}{\partial x_j}$, etc. We will use the notation $u = D^0u = (u^1, u^2, ..., u^m)$,

$Du = (u^1_{x_1}, ..., u^1_{x_n}, ..., u^m_{x_1}, ..., u^m_{x_n})$, $D^2u = (u^1_{x_1x_1}, ...)$, etc. Consequently, under this correspondence, polynomially nonlinear PDE with rational coefficients in $\Psi^l(x), ..., \Psi^m(x)$ correspond to differential polynomials in the ring $\mathbf{Q}[x, u, Du, D^2u, ..., D^qu]$.

1.2 Rankings of Derivatives

Differential elimination, like Gaussian elimination, requires a ranking (or ordering of the derivatives) in the system, although for Gaussian elimination the ranking of the unknowns is usually implied by the column order.

This concept is best illustrated by a simple example for the solution of a constant coefficient linear algebraic system with n unknowns $x = (x_1, x_2, ..., x_n)$. In order to successfully simplify this system, the first step (after selecting an equation) is to choose which of the unknowns to isolate. If the unknowns were *ranked* in a certain order, say $x_1 \succ x_2 \succ ... \succ x_n$, then it would be a simple matter to select from the unknowns present in the equation, the one of greatest rank, which for the example ranking corresponds to the unknown with the smallest subscript.

One of the requirements of a ranking for a linear system is that it is a *total ordering*. So for any two distinct unknowns in the system, it is always possible to rank one of them higher than the other.

In extending this concept to differential elimination, additional considerations arise. Instead of having to rank a finite set of unknowns, we must now rank a set of unknown functions, and all possible derivatives of those functions that appear in the course of a computation. Since the derivatives that appear are not known a priori, it is generally accepted that the ranking must be a total ordering for all possible derivatives of the unknown functions (though alternative concepts are discussed in Rust [58]).

Application of a ranking that is a total ordering to an individual differential equation allows one to select the highest ranked unknown function or derivative in that equation, which we call the *leading derivative* of that equation.

The first additional requirement is *positivity*. Essentially this means that any non-vanishing derivative of one of the unknown functions or derivatives must always be ranked higher than the function or derivative itself (e.g. $u \prec u_x \prec u_{xx} \prec ...$). Failure to satisfy positivity would allow an equation such as $u_x - u = 0$ to have $u_x \prec u$, and would allow the solved form of this equation to be $u = u_x$. Successive substitution of this relation into u would give $u \longmapsto u_x \longmapsto u_{xx} \longmapsto ...$, a non-terminating process.

The second additional requirement is *invariance under differentiation*. This means that once the leading derivative $\tilde{u}$ of an equation e has been identified, then after differentiation of the equation $\partial_{x_i} e$, the leading derivative will be given by $\partial_{x_i}\tilde{u}$.

A more formal treatment of this subject, and the classification of all rankings of derivatives is given in Rust *et al.* [58, 59] (also see Carrà Ferro and Sit [12]). We summarize the axioms a ranking must satisfy in the following definition.

Definition 1 (Ranking)
We define the relation $\preceq$ to be a ranking on a set $\mathcal{S} \subseteq \{u, Du, D^2u, ...\}$ if

- *$\preceq$ is a reflexive, transitive relation on $\mathcal{S}$*
- *$\preceq$ is a total order of $\mathcal{S}$, that is for $a, b \in \mathcal{S}$ exactly one of the following three conditions hold: $a \prec b$, or $b \prec a$, or $a = b$*
- *$\preceq$ is positive, that is for any $a, \partial_{x_i} a \in \mathcal{S}$ and all $1 \le i \le n$ we have $a \prec \partial_{x_i} a$.*
- *$\preceq$ is invariant under differentiation, that is for any $a, b, \partial_{x_i} a, \partial_{x_i} b \in \mathcal{S}$ and all $1 \le i \le n$, $a \prec b \Rightarrow \partial_{x_i} a \prec \partial_{x_i} b$.*

Where $a \prec b$ is defined in the expected way, namely $a \prec b \Rightarrow a \preceq b, a \neq b$.

Two admissible rankings for Example 1, with independent variables x, u and dependent variables $\xi(x,u)$, $\eta(x,u)$, are given by

$$\xi \prec \eta \prec \xi_u \prec \eta_u \prec \xi_x \prec \eta_x \prec \xi_{uu} \prec \eta_{uu} \prec \xi_{ux} \prec \eta_{ux} \prec \xi_{xx} \prec \eta_{xx} \prec ...,$$
$$\eta \prec' \eta_u \prec' \eta_{uu} \prec' ... \prec' \eta_x \prec' \eta_{xu} \prec' \eta_{xuu} \prec' ... \prec' \xi \prec' \xi_u \prec' \xi_{uu} \prec' ...,$$

but an infinite number of rankings are possible for this system.

In the class of admissible rankings, we find it convenient to differentiate between *sequential* and *non-sequential* rankings. Put simply, a sequential ranking is any ranking for which there are only a finite number of derivatives that are of lower rank than *any* fixed derivative.

In our example rankings above, $\prec$ is a sequential ranking, while $\prec'$ is not. To see why $\prec'$ is non-sequential, it suffices to simply choose ξ as our fixed derivative, then observe that η and its infinitely many derivatives are all of lower rank. An important application of non-sequential rankings (Mansfield [37]) is the extraction of a differential equation depending on fewer variables from a system.

1.3 Reduction of PDE with respect to other PDE

It is always possible to isolate a linear PDE with respect to its leading derivative, bringing the PDE into *solved form*. The solved form can then be used to *eliminate* that derivative, and all derivatives of it, from the remainder of the system. This is most closely related to the standard algorithm for polynomial division over the ring $\mathbf{Q}[x_1, x_2, ..., x_k]$ (see §2.3 of Cox, Little and O'Shea [18]). This process is called *reduction*, and we will denote it by the symbol $\longmapsto$. In the event that we want to refer to a reduction with respect to a specific equation, then for the equation e we will denote this by $\overset{e}{\longmapsto}$. In addition, we say an equation (derivative) *can be reduced* with respect to a leading linear equation e with leading derivative $\tilde{u}$ if it contains (equals) $\tilde{u}$ or any derivative of $\tilde{u}$.

Example 2 *Given the ranking $\preceq$ and the solved form equation $e_1 := u_t = v$ where $v \prec u_t$, the expression u_{xt} can be reduced with respect to e_1 as $u_{xt} = (u_t)_x \longmapsto (v)_x = v_x$ so we write*

$$u_{xt} \overset{e_1}{\longmapsto} v_x,$$

and we say u_{xt} reduces to v_x with respect to e_1.

The extension to reduction of an expression with respect to an ordered system of solved form equations is the straightforward one (and uses a similar notation), but for later illustration we will restrict the process to be done in an ordered way (to guarantee uniqueness of the resulting reduced expression). Pseudocode for the ordered reduction process is given by the following:

OrderedReduce(*Expr*, *Sys*, $\preceq$)

Input: An expression to reduce *Expr*, an ordered system of solved form equations $Sys = [s_1, s_2, ..., s_N]$, and the ranking $\preceq$.

Output: The expression reduced with respect to the ordered system.

$NoRed := \emptyset$ $\quad$ (*derivatives that cannot be reduced by Sys*)

while $\mathbf{derivs}(Expr) \setminus NoRed \neq \emptyset$ do

  $der := \mathbf{selectmax}(\mathbf{derivs}(Expr) \setminus NoRed, \preceq)$

  for k from 1 to N do

    if $\mathbf{isderiv}(der, \mathbf{lhs}(s_k))$ then

      $dexpr := \mathbf{diffto}(s_k, der)$

      $Expr := \mathbf{substitute}(dexpr, Expr)$

      break

    else if $der = \mathbf{lhs}(s_k)$ then

      $Expr := \mathbf{substitute}(s_k, Expr)$

      break

    end if

  end loop

  if $k >$ N then

    $NoRed := NoRed \cup \{\text{der}\}$

  end if

end while

return $Expr$

end

In this algorithm **derivs** returns a set of all unknowns and derivatives appearing in the input, **isderiv** indicates if a derivative is the derivative of another with respect to some set of differentiations, **selectmax** selects the maximal derivative or unknown with respect to the input ranking $\preceq$, **diffto** differentiates the input equation until the left hand side of the equation is equal to the input derivative, and finally **substitute** replaces all occurrences of the derivative or unknown on the left hand side of the input equation into the input expression.

The fact that the order in which the equations occur in the reducing system is important is illustrated by the following simple example:

Example 3 *Given the ranking $\preceq$ and the solved form system consisting of the two equations $e_1 := u_{xt} = u_x$, and $e_2 := u_{xy} = u_y$, then ordered reduction of the expression u_{txy} by the* *system* $S_1 = [e_1, e_2]$ *gives*

$$u_{txy} \xmapsto{S_1} u_y \ (\textit{specifically } u_{txy} \xmapsto{e_1} u_{xy} \xmapsto{e_2} u_y).$$

Ordered reduction of the same expression by the system $S_2 = [e_2, e_1]$ *gives*

$$u_{txy} \xmapsto{S_2} u_{ty} \ (\textit{specifically } u_{txy} \xmapsto{e_2} u_{ty}).$$

1.4 Integrability Conditions

For differential systems, there is another consideration in addition to the systematic elimination of derivatives from the system, and it involves the discovery of additional (implicit) conditions on the differential system, making them explicit. These are called *integrability conditions* and they result from commutativity of partial differentiation, and the fact that it is possible to isolate two derivatives of the same dependent variable in a system while neither one is a derivative of the other.

Example 4 *Consider the system of equations from Example 3, namely* $e_1 := u_{xt} = u_x$, *and* $e_2 := u_{xy} = u_y$, *under the ranking* $u \prec u_x \prec u_y \prec u_t \prec u_{xx} \prec u_{xy} \prec \dots$. *It is straightforward to see that this system is in solved form with respect to the described ranking.*

By equality of mixed partial derivatives, the relation

$$0 = u_{xyt} - u_{xty} = (u_{xy})_t - (u_{xt})_y$$

must also be satisfied. Consequently, reduction with respect to the system implies

$$(u_{xy})_t - (u_{xt})_y \longmapsto (u_y)_t - (u_x)_y = u_{yt} - u_{xy} \longmapsto u_{yt} - u_y.$$

In the differential elimination process, if an integrability condition does not reduce to zero with respect to the current system, it is appended to the system as a new equation. Thus in example 4, the new equation, $e_3 := u_{yt} = u_y$ is appended to the system. Each new integrability condition that cannot be reduced to zero with respect to the current system uncovers a new constraint on the PDE system, reducing the apparent arbitrariness of its solution (e.g. in the above example the particular values of u_{yt} and u_y at a point (x_0, y_0, t_0) in a formal power series solution can no longer be chosen independently, see §1.6).

From an extension of this argument, it can be proven that (for linear systems) this is a finite process, as continuing to restrict the solution space must terminate at some point. The proof for this is closely related to Dickson's lemma (see Rust [58], Riquier [55], Boulier *et al.* [6], and Mansfield [37]).

For r equations, solved for derivatives of the same dependent variable, the number of integrability conditions to be checked by equality of mixed partials is $\mathcal{O}(r^2)$. Elaborate and efficient algorithms for avoiding redundant integrability conditions are discussed in Reid [50], Rust [58], Boulier [5], Riquier [55], and Gerdt [21], and an enhancement and practical implementation of such criteria is presented in this dissertation.

Alternative methods exist for accounting for the differential consequences of a system. One important example is use of an *involutive division* approach, which provides additional flexibility in how the (equivalent of) integrability conditions are to be computed. These approaches originated in the multipliers and non-multipliers of Riquier [55] and Janet [31]. More recently they have been used by Pommaret [49], considerably abstracted and generalized by Gerdt [20], and used by Chen and Gao [14].

1.5 The Linear Case

The information in the prior sections is sufficient construct a differential elimination algorithm for linear problems that provides output *equivalent* to the results returned from the majority of the currently available differential elimination implementations. We call the output form *completed form*, which can be thought of as a differential Gröbner basis. Note that the completed form goes by many names, including differential Gröbner basis form (Mansfield [37]), a regular differential chain (Boulier *et al.* [6]), autoreduced completed form (i.e. the Gröbner basis is autoreduced), *rif'*-form (Rust, Reid, and Wittkopf [58, 54]), and standard form (Reid [50]). Example 1 provides an illustration of the expected output form.

The first algorithm, *LinearReduce*, performs reduction of the system to remove dependencies between the leading derivatives of each equation. When a system is such that none of the leading derivatives of the equations are derivatives of (or equal to) each other, we say the system is in *leading reduced form*. Note that the *Reduce* routine does not necessarily need to be an ordered reduction (hence the use of *Reduce* rather than *OrderedReduce*).

Algorithm 1 (LinearReduce)

Input: *A linear system of* PDE $\mathcal{F}$, *and a ranking* $\preceq$.

Output: *The input system of* PDE *in leading reduced form.*

$\mathcal{S} := \emptyset$
repeat
    Select ϕ *from* $\mathcal{F}$
    $\mathcal{F} := \mathcal{F} \setminus \{\phi\}$
    $\mathcal{R} := \{f \in \mathcal{S} \mid \textit{leading derivative of } f \textit{ can be reduced by } \phi\}$
    $\mathcal{F} := \textit{Reduce}(\mathcal{F} \cup \mathcal{R}, \phi)$
    $\mathcal{S} := (\mathcal{S} \setminus \mathcal{R}) \cup \{\phi\}$
until $\mathcal{F} = \emptyset$ *or* $\mathcal{S}$ *is inconsistent*
output $\mathcal{S}$

Although the PDE output from *LinearReduce* are linearly independent, additional linear relations may result from the integrability conditions §1.4, and this is handled by the *LinearReduceComplete* algorithm below. Note that the *IntCond* routine simply computes all integrability conditions for the input system $\mathcal{S}$ as they are defined in §1.4.

Algorithm 2 (LinearReduceComplete)

Input: *A linear system of* PDE $\mathcal{F}$, *and a ranking* $\preceq$.

Output: *The input system of* PDE *in completed form.*

repeat
    $\mathcal{S} := \textit{LinearReduce}(\mathcal{S}, \preceq)$
    $\mathcal{I} := \textit{Reduce}(\textit{IntCond}(\mathcal{S}), \mathcal{S})$
    $\mathcal{S} := \mathcal{S} \cup \mathcal{I}$
until $\mathcal{I} = \emptyset$ *or* $\mathcal{S}$ *is inconsistent*
output $\mathcal{S}$

A comment should be made that the above is not a complete implementation, but rather can contain derivatives in $\mathcal{S}$ that are reducible with respect to leading derivatives of other PDE in $\mathcal{S}$. This is analogous to use of Gaussian elimination to obtain a row-echelon form instead of a reduced row-echelon form (or in Gröbner basis terminology the system is not autoreduced). In addition, a direct implementation of the above would be brutally inefficient, as there is no process by which to check which computations have already been performed. For example, the same integrability conditions need not be recomputed, and the reduction process can be made more efficient.

For linear differential systems, the above algorithm yields a method to tell whether a given differential equation is in the differential ideal generated by the system (see Carrà Ferro [11], Mansfield [37], and Boulier *et al.* [6]). In other words, it solves the differential ideal membership problem in the linear case.

1.6 Initial Data and Existence/Uniqueness

Once a system is in *completed form* §1.5, it is possible to algorithmically determine the data (initial values) required for the existence of a unique formal power series solution. This is called the *initial data* for the system, and consists of the values for all derivatives of all dependent variables in the system that cannot be eliminated by any equations in the system under the current ranking.

Example 5 *Consider the system* $\xi = \xi(x,u)$ *and* $\eta = \eta(x,u)$*,*

$$\xi_{xx} = \xi, \quad \xi_u = \xi, \quad \eta_x = \xi.$$

This system can be readily verified to be in completed form with respect to any ranking for which $\eta \prec \xi_x$*. The initial data can be obtained as*

$$\xi(x_0,u_0) = c_1, \quad \xi_x(x_0,u_0) = c_2, \quad \eta(x_0,u) = f_3(u),$$

where c_1*,* c_2*, and* $f_3(u)$ *are the arbitrary constants and arbitrary function that must be specified to construct a unique formal power series solution at* (x_0,u_0)*.*

Algorithms for describing initial data go back to Riquier [55] and Janet [31]. Efficient versions are given in Reid [50] (also see Schwarz [62]), and versions for nonlinear systems in the presence of constraints are described in Reid *et al.* [54] and Rust [58].

For a leading linear system a (local) existence and uniqueness theorem for the problem follows from the canonical form output. To be useful in applications, algorithms in the nonlinear case should have an associated existence and uniqueness theorem that can be algorithmically extracted from their output.

1.7 Case Splitting and Leading Linear Coefficient Pivots

Though the discussion so far has focused on linear PDE systems, the process (with a number of modifications) can also be applied to nonlinear PDE systems. Once a ranking is imposed, we can determine the leading derivative of a PDE (§1.2), but for nonlinear systems, there are three possibilities:

1. The equation is linear in the leading derivative, and the coefficient of the leading derivative is independent of the dependent variables and/or derivatives of the problem (but may depend upon the independent variables).
2. The equation is linear in the leading derivative, and the coefficient of the leading derivative depends upon the dependent variables and/or the derivatives of the problem.
3. The equation is nonlinear in the leading derivative.

As a few examples to clarify the above, consider the following equations in $u(x,y)$ under the ranking $u \prec u_y \prec u_x \prec u_{yy} \prec \ldots$

$$\begin{aligned} u_{xx} + u_y &= 0, \\ xu_{xx} + u_x u_y + u^4 &= 0, \\ uu_{xx} + u_y &= 0, \\ u_x u_y + 1 &= 0, \\ u_x^2 + u_y^2 - 1 &= 0. \end{aligned}$$

The first two equations fall into case 1, the third and fourth equations fall into case 2, and the fifth and final equation falls into case 3.

For the first two cases, the PDE is said to be *leading linear*, while for the last it is said to be *leading nonlinear*. The first case is covered in the preceding discussion, and the last case is to be discussed in §1.8, so for now we concentrate on the second.

Since the coefficient of the leading derivative (called the *separant* of the equation, and denoted $\mathrm{Sep}_{\preceq}(eqn)$) contains dependent variables, it could be identically zero for all solutions of the differential system. Solving the equation for that leading derivative would implicitly introduce a division by zero. Alternatively the leading linear coefficient could be non-zero for all solutions of the system, or instead be identically zero for some solutions, and non-zero for others.

Since we do not know in advance which case the leading linear coefficient belongs to, we consider two disjoint cases corresponding to setting the coefficient to zero (appending another equation to the system), and restricting the coefficient to be non-zero (appending a new entity called an *inequation* to the system). This introduces a case splitting in the reduction process. Each of the cases is considered in turn, and the full set of solutions of the system is the union of the two cases.

Treatment of the separant in this way introduces a *pivot* into the computation.

Example 6 *Consider the system $v_{xx} = 1, (v_x - 1)v_{yy} = v_y$ under a ranking with $v_x \prec v_{yy}$. The second equation is linear in its leading derivative v_{yy}, with a coefficient depending upon v_x. We pivot on its leading linear coefficient $p = v_x - 1$, splitting into two cases corresponding to $p = 0$ and $p \neq 0$.*

The $p = 0$ case results in a new equation $v_x = 1$, which reduces the first equation in the system to $0 = 1$. This is an example of an inconsistent case, as indicated by the inconsistent equation that appears when we follow that case.

The $p \neq 0$ case results in the leading linear system $v_{xx} = 1, v_{yy} = v_y/(v_x - 1), v_x \neq 1$ with the inclusion of the inequation $v_x \neq 1$.

Any non-zero case introduces an inequation into the computation. The relationship enforced by this inequation must also be satisfied for all solutions of the system. Given that the final system is in *rif'*-form, we can validate the case by performing a reduction of the inequations (the pivots) with respect to the resulting system for that case, resulting in a (possibly) simpler set of conditions, or perhaps an unsatisfied condition that marks this as an inconsistent case.

The repeated application of this process results in a binary tree of cases for the given system.

1.8 Leading Nonlinear Equations

Though most differential elimination packages produce similar output for linear PDE systems, many different approaches are used for polynomially nonlinear PDE systems, and most available packages produce a different output (*completed*) form with different properties. As a result, the names used by each package for their completed form are different (see §1.5). We are primarily concerned with properties of the *solutions* of PDE systems, specifically existence and uniqueness results. This is related to packages that concern themselves with the *radical* of a differential ideal (see Boulier *et al.* [6] for an approach which gives output yielding a membership test for the radical of a differential ideal).

Definition 2 *Radical of an ideal: Given an ideal I, the polynomial p is said to be in the radical of the ideal if for some $k \in \mathbf{N}$ we have $p^k \in I$.*

Though this definition is stated for an algebraic ideal, it also applies to differential ideals in the obvious way. For example, the differential expression $u_{xx} + u$ is in the radical of a differential ideal if $(u_{xx} + u)^k$ is in the differential ideal for some $k \in \mathbf{N}$.

One approach (the approach used in obtaining *rif'*-form) simplifies this process by handling the differential consequences of a nonlinear equation separately from the equation itself. This is done in such a way so that the nonlinear PDE can be considered as purely algebraic constraints, and as such, treated with any of the many methods available for simplifying and enforcing these algebraic constraints.

This separation is accomplished through *spawning*, which is simply the process of differentiating a leading nonlinear PDE with respect to each independent variable in the system. The newly differentiated equations will be linear in their leading derivatives (unless the leading derivative reduces), though the coefficient of the leading linear equation always contains dependent variables, and as a result a pivot *may* be introduced.

Example 7 *Consider the nonlinear PDE $\Psi = u_{xx}^2 - xu_{xx} - {u_y}^2 = 0$. Spawning this equation (differentiation with respect to both x and y) gives the following two equations:*

$$\begin{aligned} D_x\Psi &= (2u_{xx} - x)u_{xxx} - u_{xx} - 2u_{xy}u_y = 0, \\ D_y\Psi &= (2u_{xx} - x)u_{xxy} - 2u_{yy}u_y = 0, \end{aligned}$$

both having the leading linear coefficient $2u_{xx} - x$*. In solving these new equations for* u_{xxx} *and* u_{xxy} *respectively, we would introduce the coefficient as a pivot.*

With these new equations, a calculation can account for the differential consequences of the nonlinear part of the system through case splitting.

The (now) algebraic system of leading nonlinear equations can then be treated using Gröbner basis or decomposition methods, and elimination of nonlinear terms in the rest of the system (the leading linear part) can be performed using the result.

Other algorithms (for example Boulier *et al.* [6]) utilize a different approach that involves further splitting on the leading nonlinear equations. The coefficient of the highest degree of the leading derivative in an equation *eqn* is called the *initial* of the equation, and denoted $\mathrm{Init}_{\preceq}(eqn)$. Inclusion of the initial into the list of pivots allows treatment of the equation as an equation in the leading derivative alone, with all other derivatives treated as parameters of the equation.

1.9 Algebraic Algorithms

This section includes an overview of the application of algorithms based on algebraic theory (such as Gröbner Basis methods, Characteristic Set methods, and Triangular Decomposition methods) to differential elimination.

Examples of these algorithms include Mansfield's *DiffGrob2* algorithm [38], Boulier's *Rosenfeld-Gröbner* algorithm [6] available in *Maple V.5* and higher, and our *DiffElim* algorithm discussed later in this dissertation. In each case, a great deal of additional work has been done on the algorithms discussed here, and variations of the core algorithms are frequently in use, so this should only be viewed as a brief and simplified description of the concepts. For example, extensions of the *Rosenfeld-Gröbner* algorithm include Hubert's work on a factorization free approach [27], and the work of Boulier, Lemaire and Maza on a *change of ranking* for a completed system [7] (related to the Gröbner Walk algorithm of Faugère, Gianni, Lazard and Mora for the polynomial case). For a more detailed description, please consult the relevant references.

These algorithms are primarily based on *pseudo*-reduction, which is essentially reduction with respect to the leading derivative of the equation only, treating all other derivatives in the equation parametrically. Rather than presenting an algorithm for pseudo-reduction, we will simply state that it is similar to fraction free polynomial division providing references [6, 37] and give an example.

Example 8 *The pseudo-reduction of* u_{xx}^2+1 *with respect to* $\mathcal{S}=uu_{xx}-u_x$ *is accomplished in two steps:*

$$u_{xx}^2+1 \approx u(u_{xx}^2+1) = (uu_{xx})u_{xx}+u \xmapsto{\mathcal{S}} u_x u_{xx}+u,$$
$$u_x u_{xx}+u \approx u(u_x u_{xx}+u) = (uu_{xx})u_x+u^2 \xmapsto{\mathcal{S}} {u_x}^2+u^2.$$

The above example was specifically chosen, as the process of pseudo-reduction has introduced a new solution (namely $u=0$) to the pair of equations (where we also note that $u=0$ cannot be a solution of the original equation $u_{xx}^2+1=0$). To compensate for this difficulty, any multipliers used in the pseudo-reduction must be tracked as pivots (c.f. §1.7).

One core step of these algorithms is to bring the input system into *autoreduced form.* This is a form where all equations of the system are pseudo-reduced with respect to one another. An algorithm for accomplishing this task is given below (where $\mathrm{Sep}_{\preceq}()$ and $\mathrm{Init}_{\preceq}()$ are described in §1.7 and §1.8 respectively).

Algorithm 3 (AutoReduce)

Input: *A polynomially nonlinear system of* PDE $\mathcal{F}$*, a ranking* $\preceq$*, and a set of polynomial inequations* Λ*.*

Output: *The input system of* PDE *in autoreduced form, and the updated inequations.*

$\mathcal{S} := \emptyset$
repeat
    Select ϕ *from* $\mathcal{F}$
    $\mathcal{F} := \mathcal{F} \setminus \{\phi\}$
    $\Lambda := \Lambda \cup \{Sep_{\preceq}(\phi), Init_{\preceq}(\phi)\}$
    $\mathcal{R} := \{f \in \mathcal{S} \mid$ *leading term of* f *can be pseudo-reduced by* $\phi\}$
    $\mathcal{F} := PseudoReduce(\mathcal{F} \cup \mathcal{R}, \phi)$
    $\mathcal{S} := PseudoReduce(\mathcal{S} \setminus \mathcal{R}, \phi) \cup \{\phi\}$
until $\mathcal{F} = \emptyset$ *or* $\mathcal{S}$ *is inconsistent*
if $\mathcal{S}$ *is consistent*

11 $\quad\quad \mathcal{F} := \mathcal{S},\ \mathcal{S} := \emptyset$

12 $\quad\quad$ *repeat*

13 $\quad\quad\quad\quad$ *Select* ϕ *from* $\mathcal{F}$ *with smallest leading derivative*

14 $\quad\quad\quad\quad \mathcal{F} := \mathcal{F} \setminus \{\phi\}$

15 $\quad\quad\quad\quad \mathcal{S} := \mathcal{S} \cup \mathit{PseudoReduce}(\{\phi\}, \mathcal{S})$

16 $\quad\quad$ *until* $\mathcal{F} = \emptyset$

17 *end if*

18 *output* $\mathcal{S},\ \Lambda$

In comparison to Algorithm 1 in §1.5, the additional steps 10-17 are the equivalent of a back-substitution (in linear terms, these steps convert $\mathcal{S}$ from row-echelon form to reduced row-echelon form).

The output system is algebraically independent, but integrability conditions (c.f. §1.4) can yield additional algebraic relations.

Example 9 *Consider the following system of 2* PDE *in* $u(x, y)$ *under a ranking that ranks first by total differential order, then by* $u_x \succ u_y$*:*

$$u_{xxy} = u_y^2,\ u_{xyy} = u_x.$$

It is easy to see that this system is in autoreduced form, as the leading derivatives (u_{xxy} *and* u_{xyy}*) are not derivatives of one another. Taking the integrability condition between these two* PDE *gives the relation*

$$2u_{yy}u_y - u_{xx} = 0.$$

Continuing by applying autoreduction and taking all integrability conditions (ignoring the one inconsistent case that arises) gives the reduced form of this system as

$$u_x = 0,\ u_y = 0.$$

The integrability conditions are treated by the *AutoReduction Completion* algorithm below.

Algorithm 4 (AutoReduction Completion)

Input: *A polynomially nonlinear system of* PDE $\mathcal{S}$, *a ranking* $\preceq$, *and a set of polynomial inequations* Λ.

Output: *The input system of* PDE *in autoreduced differentially completed form, and the updated inequations.*

repeat
    $(\mathcal{S}, \Lambda) := AutoReduce(\mathcal{S}, \Lambda, \preceq)$
    $\mathcal{I} := PseudoReduce(IntCond(\mathcal{S}), \mathcal{S})$
    $\mathcal{S} := \mathcal{S} \cup \mathcal{I}$
until $\mathcal{I} = \emptyset$ *or* $\mathcal{S}$ *is inconsistent*
output $\mathcal{S}$, Λ

A weakness of the presented approach is that the result obtained at the end of the computation could be inconsistent, even when solutions exist for the system.

What the algorithm does provide is the so-called generic case for a system of PDE, which can be understood as the system defining the general solution of a PDE system disregarding any singular solutions.

The implementations of these algorithms provide ways to deal with these limitations, either manually or automatically.

1.10 Geometric Algorithms

This section provides a brief discussion of the geometric approach to differential elimination.

Examples of application of the geometric approach include the work of Bocharov and Bronstein [4], implementation of the *Cartan-Kuranishi* approach by Seiler [64], and our *RifSimp* algorithm [54], though many aspects of *RifSimp*'s recent development have become more algebraic.

To begin, we need the concept of a *Jet Variety*, which associates a certain set of points (the *Jet Variety*) to a given set of PDE in a certain space (the *Jet Space*).

Definition 3 (Jet Variety) *Let* $R = (R^1, R^2, ...R^N)$ *be a system of* PDE *in independent variables* $x = (x_1, x_2, ..., x_n)$ *and dependent variables* $u = (u^1, u^2, ..., u^m)$*. Let* $z^k = (u, Du, ..., D^k u)$ *represent all jet variables up to order* k*. The* Jet variety *of the system of* PDE*,* $R(x, u, Du, ..., D^k u) = R(x, z^k) = 0$ *is defined as*

$$\mathcal{R} = \{(x, z^k) \in J^k : R(x, z^k) = 0\},$$

where J^k is the kth order Jet space *(i.e. the space formed by treatment of all independent variables, dependent variables and derivatives up to order k as co-ordinates of a formal Euclidean space).*

Also key to geometric approaches are the concepts of *prolongation* and *projection.*

Definition 4 (Prolongation and Projection) *Using the same notation as Definition 3 under the conditions of Definition 6, a single* prolongation *of the jet variety is defined as*

$$\mathcal{DR} = \{q \in J^{k+1} : R(q) = 0, DR(q) = 0\}.$$

A single projection *of the jet variety is defined as*

$$\pi\mathcal{R} = \{q \in J^{k-1} : R(q, D^k u) = 0\}.$$

The definitions for the prolongation *and* projection *of the system R follow directly (the latter requiring an implementation of some form of algorithmic elimination).*

We define the *solution variety* $\mathcal{S}$ as all points on the extended graphs of solutions of the PDE system R. Clearly $\mathcal{S} \subseteq \mathcal{R}$, and when $\mathcal{S} = \mathcal{R}$, the system is said to be *locally solvable.*

A core method for the geometric approach to differential elimination is the *Cartan-Kuranishi* approach.

Approach 1 (Cartan-Kuranishi)
Input: *system of* PDE R
Output: R*, where* R *is locally solvable or inconsistent*

repeat
  while $Sym(R)$ *is not involutive do*
    $R := DR$
  end while
  while $\mathcal{R} \neq (\pi \circ \mathcal{D})\mathcal{R}$ *do*
    $R := (\pi \circ D)R$
  end while
until $Sym(R)$ *is involutive*
return R

In the above approach, there are concepts that have not been discussed, and these are the *symbol* of a PDE system, denoted $Sym(R)$, and the property of *involutivity* of the symbol. A full description of these concepts is quite involved (see Seiler [64], and Reid, Lin, and Wittkopf [51]), so we briefly discuss some of the finer points, and some considerations for the Cartan-Kuranishi approach.

Definition 5 (Symbol) *The symbol of an order r system of* PDE $(R = 0)$ *is the Jacobian matrix of R with respect to the highest order derivatives, or*

$$Sym(R) := \frac{\partial R}{\partial (D^r u)}.$$

So the symbol is the matrix of the coefficients of the highest order derivatives of the linearized PDE system.

What may not be obvious from the definition of the Cartan-Kuranishi approach is that differentiation (i.e. the $\mathcal{DR}$) is not algebraic differentiation, but rather geometric differentiation. The key distinction is that geometric differentiation has no difficulties when multiplicities are present in the algebraic formulation of the PDE system, while algebraic differentiation does.

Example 10 *Given a* PDE *in $u(x, y)$, $R^1 = (u_x^2 + u_{yy})^2$, the algebraic derivative of R^1 with respect to x is $2(u_x^2 + u_{yy})(2u_{xx}u_x + u_{xyy})$ and we note that the leading coefficient of the new* PDE *vanishes on the solutions of R^1 (as R^1 implies $u_x^2 + u_{yy} = 0$). Conversely, the geometric derivative of R^1 with respect to x is $2u_{xx}u_x + u_{xyy}$, and the leading coefficient (either $2u_x$ or 1) does not strictly vanish on solutions of R^1.*

It turns out that one can detect this problem from certain properties of the symbol of a PDE system, specifically the so-called *constant rank conditions.*

Definition 6 (constant rank conditions) *Let $R(x, z^q)$ be a qth order* PDE *system with independent and dependent variables as in Definition 3. Then $R = 0$ is said to be of constant rank at a point $(x_0, z_0^q) \in \mathcal{R} \subseteq J^q$ if there exist nonzero constants α,β such that*

$$rank(\frac{\partial R}{\partial (D^q u)}) = \alpha, \quad rank(\frac{\partial R}{\partial z^k}) = rank(\frac{\partial R}{\partial (x, z^k)}) = \beta$$

for all (x, z^q) in some neighborhood of (x_0, z_0^q).

We note that the equality of the ranks in the second condition is needed to guarantee that there are no functional relationships between the independent variables of the system.

If these constant rank conditions are satisfied, then utilization of algebraic differentiation in place of geometric differentiation is guaranteed to work.

Discussion of the *involutivity* of the symbol (see Pommaret [48] and Seiler [64]) is somewhat complex, and involves a number of concepts that are beyond the scope of this dissertation, but a (gross) over-simplification of the concept of involutivity is that it is a geometric form of the concept of integrability conditions (c.f. §1.4).

In conclusion, we note that the Cartan-Kuranishi approach, though elegant, has a number of considerations for its application as an algorithm and implementation. The constant rank condition may result in case splitting (requiring restrictions to ensure constant rank). Additionally, as mentioned in Schü, Seiler, and Calmet [61], there is no general algorithm available for determination of the rank of the differential system, as it must be computed *on the solutions* of the system. So for implementation, one must restrict to a class of problems for which this is possible, for example leading linear systems, or polynomially nonlinear systems via Gröbner basis techniques. In the latter case, the high dimension of higher order Jet spaces makes efficient use of this approach challenging.

1.11 Discussion

Many aspects of the methods of differential elimination have a long history. In 1894 Tresse [68] showed informally that by ordering derivatives, and taking integrability conditions, systems of PDE could be brought to a certain form, from which the number of parameters in their solutions could be determined.

Tresse gave what can be regarded as a sketch of one of the first proofs of what is now called Dickson's Lemma, and the barest of outlines of Gröbner Basis Theory can be discerned in his work. Tresse [68] also gave a system of invariants which determined necessary and sufficient conditions for two second-order ODE of form $u_{xx} = f(x, u, u_x)$ to be equivalent via an analytic change of variables. A by-product of Tresse's classification is that the maximal dimension of the symmetry group of such ODE is 8, and that this value is obtained if and only if the ODE is equivalent by change of variables to a linear ODE (also see [63] for an alternative approach).

In many respects the work of Tresse was not rigorous, and Riquier and Janet put this work on a surer footing [55, 31], although apart from linear systems, with rational function coefficients, these works cannot be regarded as algorithmic by the strictest modern standards.

The descendant of Tresse's method is Cartan's powerful geometric method for determining when two differential systems are equivalent [13, 47]. The classification of second order ODE using this method was carried out by Kamran and Hsu [26]. Included in this area is the work Goldschmidt [22], Pommaret [48], and Olver [47].

On the algebraic side, Ritt [56] initiated the field of differential algebra (differential rings, fields, and ideals), followed by Rosenfeld [57] and later Kolchin [36] who developed the theory extensively. This in combination with the major advance of Buchberger's algorithm [9, 2] for systematic construction of a Gröbner basis, and Wu's work on characteristic sets [76, 77], form the foundations of the algebraic approaches to PDE systems.

Symbolic languages, such as *Maple*, *Reduce*, and *Mathematica* (to name but a few) play a key role, as they provide a stable and well-tested framework in which exact symbolic computations can be performed. Algorithms have been implemented in *Maple* by Mansfield [37, 38], Boulier, Lazard, Ollivier and Petitot [6], Reid, Wittkopf and Boulton [54], in *Reduce* by Wolf [75], Hartley and Tucker [23], Gerdt [21], and Schwarz [62], and in *Axiom* by Seiler [64, 65]. A fairly comprehensive (though dated) review of available packages can be found in Hereman [24].

Many of these algorithms and implementations result from different approaches (geometric, algebraic), are most applicable to different classes of problems (linear, quasi-linear, nonlinear), and also provide an existence-uniqueness theorem for their output. The primary motivation for the development of different approaches is the degree of difficulty in bringing these systems into completed form. Despite the advances resulting from these algorithms, serious obstacles exist to their more widespread application. Though these algorithms theoretically terminate with the desired output, very often application of these algorithms to many interesting physical problems either take an unreasonable time to complete, or require more memory than is available.

1.12 Outline

Although differential elimination algorithms are in theory finite, in practice their poor complexity and tendency to explode in memory put many interesting physical problems far out of reach.

The main accomplishments described in this dissertation are to present algorithms and implementations that, though they do not remove these difficulties, greatly enlarge the class of problems that can be successfully simplified, and for those problems for which complete simplification is still out of reach, provide approaches that obtain partial information from calculations.

A brief description of the various components of this dissertation follows.

The RifSimp Algorithm

The theoretical aspects of the *RifSimp* algorithm are described extensively in Rust [58], and reviewed briefly in this dissertation. We have efficiently implemented the polynomial case algorithm, which has been part of the *Maple* language since *Maple 6*, and is capable of simplifying systems of polynomially nonlinear ODE and PDE for a wide variety of *rankings* (orderings of derivatives, see §1.2).

The implementation is a highly flexible one, and its documentation (included in Appendix A), in addition to describing the myriad of options available, also seeks to educate the user to the concepts of differential elimination, and in how best to use the *RifSimp* package.

Despite considerable efforts, however, *RifSimp* still suffers from time and memory issues, and aspects of the algorithm are still under development, as evidenced by the following.

Avoiding Redundant Integrability Conditions

The theoretical development of the polynomial *RifSimp* algorithm (the *poly_rif'* algorithm) found in Rust [58] is applicable to all polynomially nonlinear problems under *sequential* rankings. As an example to which Rust's methods do not apply, consider the application of differential elimination to an ODE system of 2 equations in $x(t)$, $y(t)$. The most common method for obtaining the symbolic solution requires the elimination of one of $x(t)$ or $y(t)$ thus obtaining an ODE in the other dependent variable only (reducing the problem to finding the solution of a single ODE). This generally requires a non-sequential ranking (e.g. the ranking $x \prec x_t \prec x_{tt} \prec \ldots \prec y \prec y_t \prec y_{tt} \prec \ldots$ could be used to eliminate $y(t)$ and obtain an ODE in $x(t)$ only).

The primary obstacle to the extension of *poly_rif'* to a broader class of rankings, including non-sequential rankings, centers around the process of identifying redundant *differential consequences* (§1.4) of the differential system before computing them. We use the term *redundancy criteria* to describe the process of removing these potentially computationally expensive conditions.

We provide an extension of the work of Rust [58], and a modification of *RifSimp* that reduces this set of conditions to be a smaller set of conditions, and finite for any admissible ranking. We also provide an efficient implementation of this redundancy criteria for all cases, improving on the work of Reid in [50]. In addition, methods are provided to yield a finite terminating generalization of *RifSimp* to the non-sequential ranking case (Rust's theory only guarantees termination in the sequential ranking case).

The MaxDim Algorithm

Gröbner bases and differential elimination methods both show a great deal of promise when dealing with systems of algebraic and differential equations respectively. Unfortunately, the complexity of these calculations often makes their all-or-nothing approach infeasible. For systems of even moderate difficulty, the process of differential elimination sometimes causes both the number and the size of the equations of a system to grow exponentially in the course of a calculation, exhausting the resources of the machine, and causing the entire calculation to fail.

For some problems, only the cases containing the solutions with the greatest generality are needed. More precisely, the cases of interest are those having the greatest number of degrees of freedom with respect to the *initial data* (§1.6) needed to uniquely specify a formal power series solution of that system. One example of this, from the area of symmetry analysis, is searching for linearizations of PDE.

For this class of problems, our *MaxDim* algorithm (Reid and Wittkopf [53]) can be used (in combination with an automatic case splitting algorithm) to search only for those most general cases. We note that in the examples we present, *RifSimp* is used, though any other automatic case splitting algorithm can be modified for use with *MaxDim*.

We show that the *MaxDim* algorithm can be used to determine the size and structure of maximal dimension symmetry groups for problems where performing the full differential elimination process will surely fail. The ease of this method, compared to the full determination of the size and structure of all symmetry groups, is illustrated in the applications chapter (Chapter 5).

The CDiffElim Environment

As mentioned in the discussion of *MaxDim*, for many systems of differential equations the current approaches are infeasible. One of the causes of these difficulties can be identified as inefficiencies in the programming environment itself, where these are not necessarily general inefficiencies, but rather inefficiencies from the point of view of utilizing the environment for differential elimination.

The approach taken here is to design a new environment, *CDiffElim*, which supports more efficient versions of the core operations needed by differential elimination algorithms.

We have developed a C implementation which allows storage of the system, and related information needed for the algorithms, in a more compact way than is currently available in general-purpose computer algebra systems. The core operations required for differential elimination are coded in a way tailored to differential elimination problems. For example, ordered arrays are used to store equations processed by the algorithms. These ordered arrays are designed around the rankings of derivatives used in differential elimination algorithms, and allow efficient storage and fast determination of quantities needed by the algorithms.

To test the performance and establish the feasibility of the *CDiffElim* environment, an experimental algorithm, the *DiffElim* algorithm, has been written in that environment. We show significant improvements in time and memory consumption through implementation of a differential elimination algorithm in *CDiffElim*, rather than in a general purpose environment.

A long description of this work has also been published in Wittkopf and Reid [74].

Applications

We then provide a brief description of Lie symmetry analysis, and Lie symmetry classification problems, two areas in which the use of automated differential elimination is essential.

Using *RifSimp*, *DiffElim* and *MaxDim* we study the symmetries of nonlinear reaction diffusion systems, nonlinear Schrödinger systems, nonlinear telegraph systems, and the d'Alembert-Hamilton system.

Polynomial Greatest Common Divisors

One common problem observed in differential elimination algorithms (or in complex symbolic computations in general) is the problem of *intermediate expression swell.* This occurs when the size of the problem increases dramatically in the course of a computation before completion, and can even occur when both the input and output are quite small.

This becomes a significant barrier in differential elimination when the input systems are nonlinear, or contain independent variables or unknown constants in their coefficients. For many of these problems, simplification related to the removal of common factors in these coefficients consumes a large portion of the computation time.

To that end, the design and implementation of Brown's dense modular GCD algorithm [8], and Zippel's sparse modular GCD algorithm [78] have been studied, and significant enhancements to both algorithms have been made. Included in this work is a detailed asymptotic analysis of Brown's algorithm, the *EEZ-GCD* algorithm of Wang [71], and our modified Brown and Zippel algorithms, *DIVBRO* and *LINZIP* respectively.

The modified algorithms have been implemented both in *Maple* and in *CDiffElim.*

Applications & Benchmarks

To round out the treatment of differential elimination we provide a set of benchmarks.

All benchmark problems considered in this dissertation involve the study of the symmetries of the equation class

$$\mu u_t + \nabla^2 u + F(x, u, u^*) = 0, \tag{1.3}$$

where $x = (x_1, x_2, ..., x_n) \in \mathbb{R}^n$, ∇^2 is the n-space Laplacian, μ is a constant and u is an m-component complex field with conjugate u^*.

Special cases of (1.3) include the Laplace and Poisson equations, vector nonlinear Schrödinger (VNLS) systems, and coupled nonlinear Reaction-Diffusion systems to name but a few.

Chapter 2

The RifSimp Algorithm and Redundancy Criteria

In this chapter we present the theoretical development of the version of the *RifSimp* algorithm for differential polynomial systems that is currently implemented in *Maple 7* and later.

We start with a bit of history. The *RifSimp* algorithm began its long and prosperous life as the *standard form* algorithm of Reid [50] in 1991. Firmly rooted in a modified version of the Riquier theory [55] it was restricted to handling linear systems of PDE. At that time, the author of this dissertation was involved with the efficient implementation of the algorithm, which was first made available as the *standard form* package, written in *Maple* 4.3, and also released in the early 90's. Interest in systems of *nonlinear* PDE, however, was on the rise, so in collaboration, Reid, Wittkopf and Boulton developed the *rif* algorithm [54] and implementation, first made available in 1996 for *Maple V.2*. The work that followed, including the work of Rust [58], and a joint work of Rust, Reid and Wittkopf [60] in the late 90's, put the work on an even more solid theoretical footing, thus paving the way for acceptance as part of core *Maple 6* in 2000, which puts us now firmly in the present tense.

We begin with a review and discussion of the material presented by Rust [58], with particular focus on the results related to removal of redundant integrability conditions (redundancy criteria). We then present an extension of the results for the linear case to application in the nonlinear case, with the same bound on the number of conditions as was given for the linear case in [58], a significant decrease in the number of conditions that need be considered.

We also provide an efficient algorithmic realization of the new redundancy criteria, which can also be efficiently used in the linear case, and show that with these modifications, the existence-uniqueness results of [58] still apply. Analogously to S-polynomial criteria for Gröbner basis methods, integrability condition redundancy criteria enable the avoidance of many unnecessary computations. They are *vital* for a successful implementation.

The chapter closes with two adaptations of Rust's *poly_rif'* algorithm, one to systems under a sequential ranking, and the other to systems with a general ranking (i.e. non-sequential), which greatly increases the utility of *RifSimp* for many applications. These modified algorithms, together with the integrability condition redundancy criteria, provide the building blocks for efficient implementation of the *RifSimp* algorithm for both the sequential and non-sequential case.

The style of presentation differs from that in [58], as we are interested in the application of *RifSimp*, so the theorems and proofs are developed in a way to ease implementation.

2.1 Notation and Preliminaries

We adopt the following notations of Rust [58]. We let n be the positive integer representing the number of dependent variables $u = (u^1, ..., u^n)$, and m the non-negative integer representing the number of independent variables $x = (x_1, ..., x_m)$ in a system of PDE. We use Δ for the set of all dependent variables and their derivatives, and use $\tilde{\Delta}$ for $\{x\} \cup \Delta$ which includes the independent variables.

We define $\mathcal{S}$ as a finite set of polynomial functions on $\tilde{\Delta}$ over $\mathbf{Q}$ (or $\mathbf{C}$), and $\preceq$ is a fixed ranking. We use $\mathrm{HD}_{\preceq}(f)$ for $f \in \mathcal{S}$ to denote the leading (highest) derivative of f with respect to $\preceq$. We also use the notation $\Delta(\mathcal{S})$ to represent all $\delta \in \Delta$ that are present in the set of functions $\mathcal{S}$, and define $\tilde{\Delta}(\mathcal{S})$ in a similar fashion.

The set of functions $\mathcal{M}$ consists of all leading linear functions in $\mathcal{S}$ (i.e. those which can be written as $\preceq$-monic rational polynomial functions on $\mathcal{M}$). The set of functions $\mathcal{N}$ consists of the remaining (leading nonlinear) functions of $\mathcal{S}$. Thus $\mathcal{S}$ is partitioned into leading linear and leading nonlinear parts as $\mathcal{M} \cup \mathcal{N}$ respectively. The notation $\mathrm{L}_{\preceq}(\mathcal{S}) := \mathcal{M}$

and $\mathrm{N}_{\preceq}(\mathcal{S}) := \mathcal{N}$ is also used. Λ is a finite set of polynomial functions on $\tilde{\Delta}$, which represents the inequations that must be satisfied (equations of the form $f \neq 0$).

Differentiation is denoted by a vector of m non-negative integer values that correspond to the independent variables as

$$D_\alpha = D_{(\alpha_1,\dots,\alpha_m)} = D_{x_1}^{\alpha_1} \dots D_{x_m}^{\alpha_m}.$$

So for a system with three independent variables, differentiation by x_1 would be denoted by $D_{(1,0,0)}$, and differentiation by x_1, x_1, x_2 would be denoted by $D_{(2,1,0)}$. In addition, δ_α^i is used as a shorthand notation for $D_\alpha u^i$.

The set of *principal derivatives* of $\mathcal{M}$ consists of the set of derivatives that may be obtained as derivatives of the leading derivative of an element of $\mathcal{M}$, or more formally

$$\mathrm{Prin}_{\preceq}\mathcal{M} := \{\delta \in \Delta \mid \exists f \in \mathcal{M} \text{ and } \alpha \in \mathbf{N}^m \text{ with } \delta = D_\alpha \mathrm{HD}_{\preceq}(f)\}.$$

The *parametric derivatives* of $\mathcal{M}$, denoted by $\mathrm{Par}_{\preceq}\mathcal{M}$, are simply all non-principal derivatives

$$\mathrm{Par}_{\preceq}\mathcal{M} := \Delta \setminus \mathrm{Prin}_{\preceq}\mathcal{M}.$$

For a field $\mathbf{F}$, we view the *initial data* for $\mathcal{M}$ as a map $a : \{x\} \cup \mathrm{Par}_{\preceq}\mathcal{M} \to \mathbf{F}$. For $x^0 \in \mathbf{F}^m$, we say that a is a specification of initial data at x^0 if $a(x) = x^0$.

We define a one-step reduction of a function g on $\tilde{\Delta}$ with respect to $D_\alpha f$ for $f \in \mathcal{M}$ as the process of substituting $\mathrm{HD}_{\preceq}(D_\alpha f) - f$ for $\mathrm{HD}_{\preceq}(D_\alpha f)$ wherever it appears in g, and we write

$$\mathrm{red}_{\preceq}(g, (\alpha, f))$$

to represent this. For example, the reduction of u_{xx} with respect to $u_x - v_z$ for a system of PDE in $u(x, y, z), v(x, y, z)$ and a ranking consistent with $v_z \prec u_x$ would be denoted by

$$\mathrm{red}_{\preceq}(u_{xx}, ((1, 0, 0), u_x - v_z)) = v_{xz}.$$

We use the notation $g \longmapsto h$, to represent the existence of some sequence of one step reductions using the functions in $\mathcal{M}$ that takes g to h. When we wish to represent a more specific set of reductions, we use the notation $g \overset{\mu}{\longmapsto} h$, where $\mu = (\alpha_1, f_1), \dots, (\alpha_r, f_r)$ is a sequence of r one-step reductions. Note that this is a slight departure from our use of $\longmapsto$ in Chapter 1, as we also used $\longmapsto$ in place of $\mathrm{red}_{\preceq}$. We define a *complete reduction* as the process of performing all possible reductions on an expression g with respect to $\mathcal{M}$, and denote this by $\mathrm{cred}_{\preceq}(f, \mathcal{M})$. We note that this is not necessarily unique.

We will also require additional notation not from [58] as follows. We define

$$D_{\alpha\beta} := D_{(\alpha_1+\beta_1,\ldots,\alpha_m+\beta_m)},\ D_{\alpha/\beta} := D_{(\alpha_1-\beta_1,\ldots,\alpha_m-\beta_m)}$$

the second of which is only well-defined if $\alpha_i - \beta_i$ is nonnegative for all i. We also define

$$\mathrm{LCM}(\alpha, \beta) := (\max(\alpha_1, \beta_1), \ldots, \max(\alpha_m, \beta_m)),$$

$$\mathrm{GCD}(\alpha, \beta) := (\min(\alpha_1, \beta_1), \ldots, \min(\alpha_m, \beta_m))$$

where this notation is also extended to differentiation as $\mathrm{LCM}(D_\alpha, D_\beta)$, $\mathrm{GCD}(D_\alpha, D_\beta)$ and to derivatives as $\mathrm{LCM}(\delta_\alpha, \delta_\beta)$, $\mathrm{GCD}(\delta_\alpha, \delta_\beta)$ in the expected way.

The material in the following sections, though it is generally stated in terms of polynomially nonlinear systems, is equally applicable to linear systems (these are simply a sub-case), so all results apply to linear systems as well.

2.2 Key Results from Rust [58]

In this section, we review key results of Rust [58] which we will either use or extend in the development that follows.

Note that the presentation here is focused on implementation of the theory, and as a result does not include results that are less relevant to implementation or definitions that are not of the constructive variety.

The first set of results is taken from the development for leading linear systems in [58], and as such, apply to systems with an arbitrary ranking $\preceq$.

The following definition provides a formal description of what we loosely refer to as simply an *integrability condition* in §1.4.

Definition 7 (Minimal Integrability Condition (5.3.1 of [58])) *Take $f, f' \in \mathcal{M}$. Let their highest derivatives be δ^i_α and $\delta^{i'}_{\alpha'}$, respectively. Let β be the* LCM *of α and α'. Then if $i = i'$, we define the minimal integrability condition of f and f' to be*

$$\mathrm{IC}_{\preceq}(f, f') := D_{\beta/\alpha} f - D_{\beta/\alpha'} f'.$$

If $i \neq i'$, then $\textsc{ic}_{\preceq}(f, f')$ *is said to be undefined.*

The combination of integrability conditions and the following corollary provide enough information to detect if a system is a *Riquier Basis*. It turns out that for linear problems this is *almost* the same as *rif'*-form (which is the output of the *RifSimp* algorithm), the only difference being that *rif'*-form requires the removal of equations that reduce to zero with respect to the rest of the system (a *Reduced Riquier Basis*).

It is important to note that in the linear case, for finite $\mathcal{M}$, the maximal number of integrability conditions is $\frac{\#\mathcal{M}(\#\mathcal{M}+1)}{2} = \mathcal{O}(\#\mathcal{M}^2)$.

Corollary 1 (Riquier Basis Integrability Conditions (5.3.1 of [58])) *Suppose that* $\textsc{ic}_{\preceq}(f, f') \longmapsto 0$ *for all* $f, f' \in \mathcal{M}$ *with* $\textsc{ic}_{\preceq}(f, f')$ *well-defined. Then* $\mathcal{M}$ *is a Riquier basis.*

The lemma below is used as an important part of the proof of the following theorem, which concerns dependencies between integrability conditions.

Lemma 1 (Derivative One-Step Reduction Lemma (5.3.2 of [58])) *Take* $\alpha \in \mathbf{N}^m$, $f \in \mathcal{M}$, $i \in \mathbf{N}_n$ *and* g *an analytic function of* $\tilde{\Delta}$. *Let* δ_1 *be the highest derivative of* $D_\alpha f$. *Let* δ_0 *be given by* $\delta_1 = D_i \delta_0$, *if this is well-defined. Then:*

$$red_{\preceq}(D_i g, (\alpha, f)) = D_i red_{\preceq}(g, (\alpha, f)) + red_{\preceq}(\frac{\partial g}{\partial \delta_1}, (\alpha, f)) D_i D_\alpha f - red_{\preceq}(\frac{\partial g}{\partial \delta_0}, (\alpha, f)) D_\alpha f.$$

If δ_0 *is not well-defined, then the last term is omitted in the above formula.*

Theorem 1 (Riquier Basis Integrability Condition Expansion Theorem (5.3.1 of [58]))

Suppose that for each pair $f, f' \in \mathcal{M}^2$ *with* $\textsc{ic}_{\preceq}(f, f')$ *well-defined there exists an expansion of* $\textsc{ic}_{\preceq}(f, f')$ *of the form*

$$\textsc{ic}_{\preceq}(f, f') = \sum_{(b,b') \in B_{f,f'}} D_{\alpha_{f,f',b,b'}} \textsc{ic}_{\preceq}(b, b')$$

where $B_{f,f'}$ *is a subset of* $\mathcal{M}^2$ *such that for each* $(b, b') \in B_{f,f'}$, $\textsc{ic}_{\preceq}(b, b') \longmapsto 0$ *and* $D_{\alpha_{f,f',b,b'}} \textsc{lcm}(\textsc{hd}_{\preceq}(b), \textsc{hd}_{\preceq}(b')) \preceq \textsc{lcm}(\textsc{hd}_{\preceq}(f), \textsc{hd}_{\preceq}(f'))$. *Then* $\mathcal{M}$ *is a Riquier basis.*

The combination of these provides the following corollary, which presents Rust's redundancy criteria for integrability conditions.

Corollary 2 (Rust's Riquier Basis Reduced Integrability Conditions 5.3.2 of [58]) *Let B be a subset of $\mathcal{M}^2$ such that $\textsc{ic}_{\preceq}(b, b') = 0$ for all $(b, b') \in B$ and for $f, f' \in \mathcal{M}$ with $\textsc{ic}_{\preceq}(f, f')$ well-defined, there exists $b \in \mathcal{M}$ such that*

1. $(f, b), (b, f') \in B$ *and*

2. $\textsc{hd}_{\preceq}(b)$ *divides* $\textsc{lcm}(\textsc{hd}_{\preceq}(f), \textsc{hd}_{\preceq}(f'))$.

Then $\mathcal{M}$ is a Riquier basis.

This criteria, though it can identify many redundant integrability conditions, does not translate well into an implementation that can be used with an algorithm like *RifSimp*, which requires the ability to incrementally determine new integrability conditions, as evidenced by the following example.

Example 11 *Consider the following system of equations in $u(x, y)$ under a ranking $\prec$ that ranks first by total differential order, then by differential order in x:*

$$\{e_1, e_2, e_3, e_4, e_5\} := \{u_{xxxx} = v_1, u_{xxxy} = v_2, u_{xxyy} = v_3, u_{xyyy} = v_4, u_{yyyy} = v_5\},$$

where the v_i contain only lower ranked u derivatives than the ones explicitly given. The full set of integrability conditions required by Corollary 1 is given by:

$$\begin{array}{llll}
D_y e_1 - D_x e_2, & D_y e_2 - D_x e_3, & D_y e_3 - D_x e_4, & D_y e_4 - D_x e_5, \\
D_{yy} e_1 - D_{xx} e_3, & D_{yy} e_2 - D_{xx} e_4, & D_{yy} e_3 - D_{xx} e_5, & \\
D_{yyy} e_1 - D_{xxx} e_4, & D_{yyy} e_2 - D_{xxx} e_5, & & \\
D_{yyyy} e_1 - D_{xxxx} e_5, & & &
\end{array}$$

where they are listed so that each row represents an increase in differential order. This can be depicted by the following staircase diagram*:*

where the diagram represents differentiations with respect to x (horizontal axis), and y (vertical axis), and we have plotted the leading derivative of each equation as an empty circle (labelled), and integrability conditions as filled circles at the LCM *of the leading derivatives.*

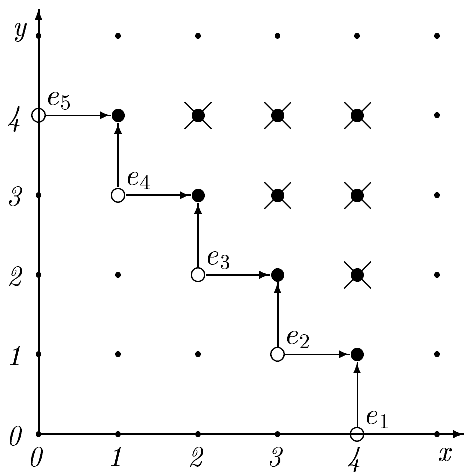

Figure 2.1: Integrability condition staircase diagram

It is straightforward to show that the first row of conditions (corresponding to the minimal staircase, *which are each drawn with a pair of arrows originating from the equations used for them) is sufficient in this case. The other conditions shown with crosses through them are redundant.*

Application of Corollary 2 to the first condition in the second row would proceed as follows. Consider the condition $\mathrm{IC}_{\preceq}(e_1, e_3) = D_{yy}e_1 - D_{xx}e_3$. *Using the corollary, this condition can be shown to be redundant with* $b = e_2$, *as* $\mathrm{HD}_{\preceq}(e_2) = u_{xxxy}$ *which divides* $\mathrm{LCM}(\mathrm{HD}_{\preceq}(e_1), \mathrm{HD}_{\preceq}(e_3)) = u_{xxxyy}$, *so* (e_1, e_2) *and* (e_2, e_3) *must remain in the reduced set. The remaining two conditions in the second row can be removed in the same manner.*

A problem then occurs when attempting to remove any of the conditions in the third and fourth rows. Consider the condition $\mathrm{IC}_{\preceq}(e_1, e_4) = D_{yyy}e_1 - D_{xxx}e_3$. *There is no* b *that satisfies the conditions of Corollary 2, so this redundant condition must be retained. Similar comments apply to the remaining two conditions, as they cannot be removed by Corollary 2 either.*

So in this case, the minimal set consists of four of the ten conditions, but we are only able to remove three, forcing the inclusion of three redundant integrability conditions.

It should be noted that if we instead began with removal of the highest order condition, and worked our way downward (in differential order) it would *be possible to remove all redundant* *conditions, but the application of these criteria is not transparent or easy to apply to general systems.*

So it was possible to utilize the criteria to reduce the conditions in this case to the minimal set, but *only if* they were considered in the right order. The implementation issue here involves the application of these criteria to a system as it is changing (through the course of the algorithm). For example, suppose we initially have e_1, e_2, e_4 and e_5 from the example, but e_3 was encountered in a later iteration. The problem is that the conditions used to reject the higher order integrability conditions involving e_3 are themselves rejected. This forces us either to inefficiently regenerate all conditions every time the system changes to eliminate redundant conditions or to simply compute, at much greater cost, the higher order redundant conditions.

We can consider the integrability condition removal criteria from Corollary 2 to be a second-order correction, or in other words it removes conditions that are redundant when considering all *pairs* of conditions we are computing. In order to remove *all* redundant conditions, we need also consider higher order corrections. For example, a condition between f and f' could be redundant as a result of conditions $(f, b), (b, c), (c, f')$, which we could call a third-order correction. Direct application of this type of approach (use of higher order corrections) is not recommended, due to efficiency considerations.

This next set of results is taken from the nonlinear development in [58], and are thus restricted to systems with a sequential ranking $\preceq$. Also note that for linear systems, a Riquier basis and a *rif'*-form are equivalent.

Definition 8 (Special Consequence and Relative Riquier Basis (6.3.1 of [58])) *Let U be a non-empty open subset of $\mathbf{F}^{\tilde{\Delta}}$ on which $\mathcal{M} \cup \mathcal{N}$ is analytic. For η an analytic function of $\mathbf{F}^{\tilde{\Delta}}$, we say that η is a special consequence of $\mathcal{N}$ if η admits an expansion of the form*

$$\eta = \sum_{i=1}^{k} h_i g_i$$

with $g_1, ..., g_k \in \mathcal{N}$ and $h_1, ..., h_k$ analytic functions on U that depend on $\{x\} \cup Par_{\preceq}\mathcal{M}$ only. We say that $\mathcal{M}$ is a Riquier basis relative to $\mathcal{N}$ on U if for all $\alpha, \alpha' \in \mathbf{N}^m$ and $f, f' \in \mathcal{M}$ with $\mathrm{HD}_{\preceq}(D_\alpha f) = \mathrm{HD}_{\preceq}(D_{\alpha'} f')$, the integrability condition $D_\alpha f - D_{\alpha'} f'$ can be reduced to a special consequence of $\mathcal{N}$.

The heart of this definition, the *Relative Riquier Basis*, can be thought of as an extension of a differential ideal to a system of PDE partitioned into its leading linear and leading nonlinear parts. All integrability conditions of the leading linear part $\mathcal{M}$ can be reduced to an expression that can be written as a sum of the elements of the leading nonlinear part $\mathcal{N}$ with coefficients in $\{x\} \cup \mathrm{Par}_{\preceq}\mathcal{M}$ (or we could say that it can be reduced to an element of the *analytic ideal* of functions on U generated by $\mathcal{N}$). The definition is stated for *analytic* functions, which is far more general than we need (polynomials). For $\mathcal{N} = \emptyset$ the definition becomes the same as that of a *Riquier Basis*.

Example 12 (Special Consequence)
For a system

$$\mathcal{M} = \{u_{xx} = u, v_x = v\},\ \mathcal{N} = \{u_x^2 - xu^2 = 0, u^3 - uv + v^2 = 0\}$$

the following expression is a special consequence

$$\frac{u_x^3}{x} - u_x v + \frac{u_x v^2}{u} = \frac{u_x}{x}\left(u_x^2 - xu^2\right) + \frac{u_x}{u}\left(u^3 - uv + v^2\right)$$

as it can be written as a sum of the elements of $\mathcal{N}$ with coefficients depending only on $\{x\} \cup Par_{\preceq}\mathcal{M} = \{x, u, u_x, v\}$.

The following theorem is quite similar, as it also has an easily interpreted analog for differential ideals.

Theorem 2 (Nonlinear Complete Reduction Theorem (6.3.1 of [58]))
Suppose that $\mathcal{M}$ is a Riquier basis relative to $\mathcal{N}$ and g is an analytic function on U, polynomial in $Prin_{\preceq}\mathcal{M}$. Then the difference between any two complete reductions of g is a special consequence of $\mathcal{N}$. In particular, any two complete reductions of g agree on the algebraic solutions to $\mathcal{N}$.

For the following theorem, let $\mathrm{redmod}_{\preceq}(g, \mathcal{M})$ denote a particular choice of complete reduction of g with respect to $\mathcal{N}$.

Theorem 3 (Relative Riquier Basis Theorem (6.3.2 of [58])) *Let $\mathcal{M}$ be a Riquier basis relative to $\mathcal{N}$ such that each $f \in \mathcal{M}$ is polynomial in the principal derivatives. Fix*

$x^0 \in \mathbf{F}^m$. Let a be a specification of initial data for $\mathcal{M}$ at $x^0 \in \mathbf{F}^m$ with $a \in U$. Then there exists a unique solution $u(x) \in \mathbf{F}[[x - x^0]]^n$ to $\mathcal{M}$ such that $D_\alpha u^i(x^0) = a(redmod_{\preceq}(\delta^i_\alpha, \mathcal{M}))$ for all $\alpha \in \mathbf{N}^m$ and $i \in \mathbf{N}_n$.

The above is a powerful extension to the existence and uniqueness theorems for *rif'*-form.

Lemma 2 (Nonlinear Sum Lemma (6.3.3 of [58])) *Suppose that*

1. *The functions h,k are analytic on U and polynomial in Prin($\mathcal{M}$).*

2. *Both $h \longmapsto \eta_h$ and $k \longmapsto \eta_k$, with η_h, η_k special consequences of $\mathcal{N}$.*

3. *For all $\alpha, \alpha' \in \mathbf{N}^m$ and $f, f' \in \mathcal{M}$ with $\text{HD}_{\preceq}(D_\alpha f) = \text{HD}_{\preceq}(D_{\alpha'} f') \preceq \delta$ there exists a special consequence η of $\mathcal{N}$ such that $D_\alpha f - D_{\alpha'} f' \longmapsto \eta$, where δ denotes the highest principal derivative that occurs in k.*

Then there exists a special consequence η' of $\mathcal{N}$ such that $h + k \longmapsto \eta'$.

We note here that no part of the proof of the above lemma requires that the ranking be sequential, so it holds equally well in the general ranking case, though its usefulness is somewhat limited as the number of conditions that must hold is potentially infinite (as for a non-sequential ranking, there can be an infinite number of derivatives less than δ).

We will take advantage of this flexibility in our development.

Lemma 3 (Nonlinear Derivative Lemma (6.3.4 of [58])) *Let g be an analytic function on U, polynomial in $Prin_{\preceq}(M)$, such that $g \longmapsto \eta_g$ for some special consequence η_g of $\mathcal{N}$. Fix $i \in \mathbf{N}_m$. Let $\delta = max_{\eta \in \mathcal{N}}(\text{HD}_{\preceq}(\eta))$ and let δ' be the highest principal derivative that occurs in g. Suppose that for all $\alpha, \alpha' \in \mathbf{N}^m$ and $f, f' \in \mathcal{M}$ with $\text{HD}_{\preceq}(D_\alpha f) = \text{HD}_{\preceq}(D_{\alpha'} f')$ such that $\text{HD}_{\preceq}(D_\alpha f) \prec D_i \delta'$ or $\text{HD}_{\preceq}(D_\alpha f) \preceq D_i \delta$, we have that $D_\alpha f - D_{\alpha'} f'$ reduces to a special consequence of $\mathcal{N}$. Suppose further that $D_i \eta$ reduces to a special consequence of $\mathcal{N}$ for all $\eta \in \mathcal{N}$. Then there exists a special consequence η' of $\mathcal{N}$ such that $D_i g \longmapsto \eta'$.*

As with the previous theorem, the above lemma requires a sequential ranking to obtain a finite number of conditions to check, but it is a vital lemma, as it provides a completeness result for reduced relative Riquier bases.

The following theorem outlines the requirements for $(\mathcal{M}, \mathcal{N})$ to be in *rif'*-form.

Theorem 4 (Nonlinear Integrability Condition Expansion Theorem (6.3.3 of [58])) *Suppose that for each pair $f, f' \in \mathcal{M}^2$ with $\mathrm{IC}_{\preceq}(f, f')$ well-defined there exists an expansion of $\mathrm{IC}_{\preceq}(f, f')$ of the form*

$$\mathrm{IC}_{\preceq}(f, f') = \sum_{(b,b') \in B_{f,f'}} D_{\alpha_{f,f',b,b'}} \mathrm{IC}_{\preceq}(b, b')$$

Here $B_{f,f'} \subseteq \mathcal{M}^2$ such that for each $(b, b') \in B_{f,f'}$, there exists a special consequence $\eta_{b,b'}$ of $\mathcal{N}$ with $IC(b, b') \longmapsto \eta_{b,b'}$, and $D_{\alpha_{f,f',b,b'}} \mathrm{LCM}(\mathrm{HD}_{\preceq}(b), \mathrm{HD}_{\preceq}(b')) \preceq \mathrm{LCM}(\mathrm{HD}_{\preceq}(f), \mathrm{HD}_{\preceq}(f'))$. We define $\eta_{max} = max_{i \in \mathbf{N}_m, \eta \in \mathcal{N}}(D_i \mathrm{HD}_{\preceq}(\eta))$. Suppose also that for all $(f, f') \in \mathcal{M}^2$ with $\mathrm{IC}_{\preceq}(f, f')$ well-defined and $\alpha, \alpha' \in \mathbf{N}^m$ such that $\mathrm{HD}_{\preceq}(D_\alpha f) = \mathrm{HD}_{\preceq}(D_{\alpha'} f') \preceq \eta_{max}$, $D_\alpha f - D_{\alpha'} f'$ reduces to a special consequence of $\mathcal{N}$. Finally, suppose that $D_i \eta$ reduces to a special consequence of $\mathcal{N}$ for all $\eta \in \mathcal{N}$. Then $(\mathcal{M}, \mathcal{N})$ is in rif'-form.

The above theorem exactly describes the integrability conditions that must hold for the system to be in *rif'*-form. Unfortunately, the conditions on the linear part of the system must hold out to the highest ranked single differentiation of the highest ranked leading derivative in $\mathcal{N}$ (labelled η_{max}). We have now lost the desirable behavior that we need only consider $\mathrm{O}(\#\mathcal{M}^2)$ integrability conditions, as evidenced by the following example.

Example 13 *Consider the following system of equations in $u(x,y)$ under a (sequential) ranking $\preceq$ that ranks first by differential order, then by differential order in x:*

$$\mathcal{M} := \{l_1, l_2\} = \{u_{xy} = v_1, u_{yy} = v_2\},\ \mathcal{N} := \{n_1\} = \{f(u_{xxxx}, ...)\},$$

where the v_i and f contain only lower ranked u derivatives than the ones explicitly given. The full set of integrability conditions required by Corollary 4 is given by

$$\begin{gathered} D_y f_1 - D_x f_2, \quad D_{xy} f_1 - D_{xx} f_2, \quad D_{yy} f_1 - D_{xy} f_2, \\ D_{xxy} f_1 - D_{xxx} f_2, \quad D_{xyy} f_1 - D_{xxy} f_2, \quad D_{yyy} f_1 - D_{xyy} f_2. \end{gathered}$$

This can be depicted by the following staircase diagram*:*

where for the most part the diagram can be interpreted as in Example 11. The labelled η_{max}

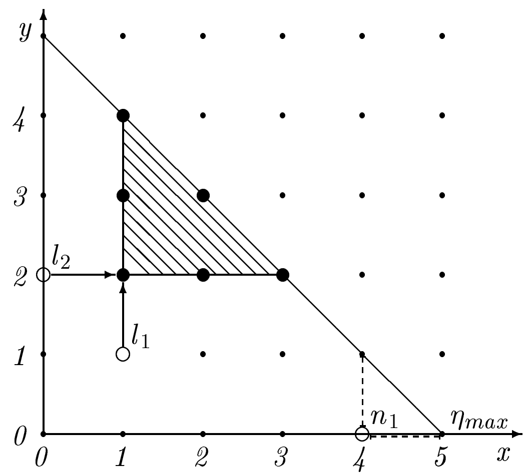

Figure 2.2: Nonlinear integrability condition staircase diagram

generates a boundary, up to and including which all integrability conditions must be checked. The shaded region shows the wedge *of integrability conditions that must be checked for the single pair of* PDE, l_1,l_2 *from* $\mathcal{M}$. *Note that in the linear case, the pair* l_1,l_2 *would only require that one integrability condition must be checked (instead of the 6 required here).*

Now consider nonlinear problems in more than 2 independent variables. The wedge described in the example extends to all m independent variables of the problem, so the condition count may be exponential in m.

The following is the existence and uniqueness result for a system in *rif'*-form.

Theorem 5 (Nonlinear Existence and Uniqueness Theorem (6.4.1 of [58])) *Let* $\mathcal{S}$ *be a finite set of analytic functions of* $\tilde{\Delta}$. *Let* $\mathcal{M}$ *denote* $L_{\preceq}(\mathcal{S})$ *and* $\mathcal{N}$ *denote* $N_{\preceq}(\mathcal{S})$. *Let* U *be an open subset of* $\{x\} \cup Par_{\preceq}\mathcal{M}$. *Suppose that* $\mathcal{S}$ *is in rif'-form on* U, *and that* α *is a specification of initial data for* $\mathcal{M}$ *that lies in* U. *Suppose further that* $\alpha(g) = 0$ *for all* $g \in \mathcal{N}$. *Let* $u(x) \in \mathbf{F}[[x - x^0]]^n$ *be the solution to* $\mathcal{M}$ *that corresponds to* α *via Theorem 3. Then* $u(x)$ *is a solution to* $\mathcal{S}$.

And now we present the core algorithms from Rust [58] required to obtain a *rif'*-form for systems of differential polynomial equations.

The first algorithm to be described is the *riq-autoreduce* algorithm (pp. 82 of Rust [58]). Note, in [58] it is referred to as the *autoreduce* algorithm, but has been renamed to prevent confusion with the autoreduce algorithm in Mansfield's *DiffGrob* package [37]. We note here that an autoreduced Riquier basis is in fact a reduced Riquier basis or a standard form [50]. Although the algorithm is stated in terms of analytic functions, it only performs reductions related to the leading linear part of the system.

Algorithm 5 (riq-autoreduce)
Input: *Two finite sets* $\mathcal{F}$,Λ *of analytic functions of* $\tilde{\Delta}$ *(the system and pivots respectively).*
Output: *Two finite sets* $\mathcal{F}'$,Λ' *such that* $\mathcal{F}'$ *is autoreduced, and equivalent to* $\mathcal{F}$ *subject to* Λ'*. In addition,* $\Lambda' \setminus \Lambda$ *is analytic.*

repeat
    repeat
        $\mathcal{M} := \{f \in \mathcal{F} \mid f$ *is* $\preceq$*-monic* $\}$
        $\mathcal{F}_{prev} := \mathcal{F}$
        for $f \in \mathcal{F}$ *do*
            $\mathcal{F} := (\mathcal{F} \setminus \{f\}) \cup \{cred_{\preceq}(f, \mathcal{M} \setminus \{f\})\}$
        end for
    until $\mathcal{F} = \mathcal{F}_{prev}$
    if $\exists f \in L_{\preceq}(\mathcal{F})$ *not monic then*
        choose such an f
        $\mathcal{F} := (\mathcal{F} \setminus \{f\}) \cup \{\frac{f}{\mathrm{HC}_{\preceq}(f)}\}$
        $\Lambda := \Lambda \cup \{\mathrm{HC}_{\preceq}(f)\}$
    end if
until $\mathcal{F} = \mathcal{F}_{prev}$
if $0 \in \mathcal{F}$ *then*
    $\mathcal{F} := \mathcal{F} \setminus \{0\}$
end if
output $(\mathcal{F}, \Lambda)$

The proof of termination is given in [58] and in essence relies on Dickson's lemma. It is not provided here.

The next algorithm is the *poly_Janet* algorithm (pp. 83 of Rust [58]). Note that this algorithm is not the same as the *Janet Basis* algorithm of Schwarz [63]. This algorithm is all that is needed if the input set of functions is (and remains) leading linear in Δ. The algorithm is slightly modified from the one in [58], as we are only interested in the case where the set of functions $\mathcal{F}$ is polynomial in $\tilde{\Delta}$.

Algorithm 6 (poly_Janet)
Input: *A finite set of functions $\mathcal{F}$ polynomial in $\tilde{\Delta}$.*
Output: *The triple $(\mathcal{F}', \Lambda, \mathit{flag})$, where flag indicates if the output is a reduced Riquier basis, inconsistent, or unsuccessful (i.e. leading nonlinear relations are present). The set $\mathcal{F}'$ is equivalent to $\mathcal{F}$ subject to Λ.*

$\Lambda = \emptyset$
repeat
    $\mathcal{F}_{prev} = \mathcal{F}$
        $(\mathcal{F}, \Lambda) :=$ *riq-autoreduce*$(\mathcal{F}, \Lambda)$
    for each integrability condition $\mathrm{IC}_{\preceq}(f, f')$ *do*
        if $\mathit{cred}_{\preceq}(\mathrm{IC}_{\preceq}(f, f'), L_{\preceq}(\mathcal{F})) \neq 0$ *then*
            $\mathcal{F} := \mathcal{F} \cup \{\mathit{cred}_{\preceq}(\mathrm{IC}_{\preceq}(f, f'), L_{\preceq}(\mathcal{F}))\}$
        end if
    end for
until $\mathcal{F} = \mathcal{F}_{prev}$
for $f \in \mathcal{F}$ *do*
    $f :=$ *numerator*(f)
end for
if $\exists f \neq 0 \in \mathcal{F} \mid f$ *independent of* Δ *or* f *has empty domain then*
    output $(\mathcal{F}, \Lambda, \mathit{inconsistent})$
else if $L_{\preceq}(\mathcal{F}) \neq \mathcal{F}$ *then*
    output $(\mathcal{F}, \Lambda, \mathit{unsuccessful})$
else
    output $(\mathcal{F}, \Lambda,$ *reduced Riquier basis*$)$
end if

The proof of termination for this algorithm also relies on Dickson's lemma (see [58] for more detail). We also note that the *poly_Janet* algorithm terminates for a non-sequential ranking, but fails whenever a leading nonlinear equation is encountered.

The next algorithm is a modification of *poly_Janet* to handle leading nonlinear equations, and that version of the algorithm is called *poly_modJanet* (pp. 101 of Rust [58]).

Algorithm 7 (poly_modJanet)
Description: *This algorithm is identical to poly_Janet, except that it does not return with flag=unsuccessful (so the failure check on lines 16 and 17 is removed), and lines 5-9 are modified to account for the additional integrability conditions required by Rust's Theorem 6.3.3 of [58] (Theorem 4) and demonstrated in Example 13.*

And the final algorithm for this section, the *poly_rif'* algorithm (pp. 101 of Rust [58]), is only applicable for sequential rankings, and provides a *rif'*-form. We note that a minor correction has been applied to line 7 of the algorithm from [58].

Algorithm 8 (poly_rif')
Input: *Two finite sets of functions* $\mathcal{S}$,Λ *polynomial in* $\tilde{\Delta}$.
Output: *A triple* $(\mathcal{F}', \Lambda', flag)$ *such that* $\mathcal{F}$ *is equivalent to* $\mathcal{S}$ *subject to* Λ' *and* $\mathcal{F}$ *is either a reduced Riquier basis, inconsistent, or a rif'-form, as indicated by flag.*

1 *repeat*
2 $\quad (\mathcal{S}, \Lambda, flag) :=$ *poly_modJanet*$(\mathcal{S}, \Lambda)$
3 $\quad$ *if flag* $\neq$ *unsuccessful then*
4 $\quad\quad$ *output* $(\mathcal{S}, \Lambda, flag)$
5 $\quad$ *end if*
6 $\quad J := \bigcup_{j=1}^{m} D_j N_{\preceq}(\mathcal{S})$
7 $\quad J := J \setminus \mathcal{S}$
8 $\quad \mathcal{S} := \mathcal{S} \cup J$
9 *until* $J = \emptyset$
10 *output* $(\mathcal{S}, \Lambda, rif'\text{-}form)$

The results and algorithms in this section will be utilized or extended in the development that follows.

2.3 All-Pairs Integrability Conditions for Polynomially Nonlinear Systems

One of the key requirements for the creation of an efficient algorithm for the polynomially nonlinear case is the ability to remove the redundant integrability conditions that appear at each stage of the algorithm, and that must be checked at the end of the algorithm to verify that we have a *rif'*-form. This is vital for practical implementation.

In the work of Rust [58] it was proven that for the linear case it is possible to reduce the number of integrability conditions to $\mathcal{O}(\#\mathcal{M}^2)$ (the *all-pairs* integrability conditions). For the nonlinear case, however, the number of integrability conditions can be significantly greater (see Example 13 in §2.2), and if a non-sequential ranking is in use, then no results have been proven.

In this section we build on the work in [58], proving that for the polynomial case and an arbitrary ranking, it is possible to reduce the number of integrability conditions to the $\mathcal{O}(\#\mathcal{M}^2)$ all-pairs integrability conditions.

This development now diverges from the development of [58]. The stronger results for the more restrictive problem are made possible by the following Lemma.

Lemma 4 *Let $\mathcal{N}$ be a lexicographic Gröbner Basis under the pure lexicographic ordering $\ll$ that agrees with $\preceq$ for all $\delta, \delta' \in \Delta(\mathcal{N})$ (we denote this the* compatible lex ordering*). Then for any η which is a special consequence of $\mathcal{N}$ (which implies η is in the ideal of $\mathcal{N}$, or $\eta \in Id(\mathcal{N})$) there exist h_i such that η can be represented as $\eta = \sum h_i g_i$ with $\mathrm{HD}_{\preceq}(g_i) \preceq \mathrm{HD}_{\preceq}(\eta)$ and $\mathrm{HD}_{\preceq}(h_i) \preceq \mathrm{HD}_{\preceq}(\eta)$ for all $i \in \phi$ where $\phi \subseteq \{1, ..., \#\mathcal{N}\}$.*

Proof For this proof we require Definition 5.37 and Theorem 5.35 (vii) of Becker & Weispfenning [2], which states that if $\mathcal{N}$ is a Gröbner Basis, then every $0 \neq \eta \in \mathrm{Id}(\mathcal{N})$ is top-reducible modulo $\mathcal{N}$. We use $\mathrm{LT}_{\ll}(\eta)$ to denote the leading term of η with respect to the lex ordering $\ll$.

We proceed by induction on the chain of top-reductions required to reduce η to zero modulo $\mathcal{N}$. We construct a vector of multipliers $h_1, ..., h_s \in \tilde{\Delta}$ that correspond to the multiples of $g_1, ..., g_s \in \mathcal{N}$ that are used in the reduction process. At any step we want to prove that:

1. The current η_i is in the ideal

2. η_i satisfies $\text{HD}_{\preceq}(\eta_i) \preceq \text{HD}_{\preceq}(\eta)$

3. $h_j = 0$ for all g_j such that $\text{HD}_{\preceq}(g_j) \succ \text{HD}_{\preceq}(\eta)$, and $\text{HD}_{\preceq}(h_j) \preceq \text{HD}_{\preceq}(\eta)$ for all g_j such that $\text{HD}_{\preceq}(g_j) \preceq \text{HD}_{\preceq}(\eta)$.

4. At any iteration of the reduction, we have $\eta = \eta_i + \sum h_j g_j$.

For our base case we have $\eta_0 = \eta \in \text{Id}(\mathcal{N})$, $\text{HD}_{\preceq}(\eta_0) = \text{HD}_{\preceq}(\eta)$, $h_1, ..., h_s = 0, ..., 0$, so none of the multipliers contain $\delta \succ \text{HD}_{\preceq}(\eta)$, and we have not used any $g_j \in \mathcal{N}$. Also $\eta = \eta_0 + \sum(0)g_j$, so the hypotheses hold.

Now we assume this is true for η_i at some step, and apply a top-reduction. In order for a top-reduction to be possible, it must be the case that $\text{LT}_{\ll}(g)$ divides $\text{LT}_{\ll}(\eta_i)$ for some $g \in \mathcal{N}$. We choose any such g, say g_k.

Our first observation is that $\text{HD}_{\preceq}(\text{LT}_{\ll}(g_k)) \preceq \text{HD}_{\preceq}(\eta)$ which holds from the division property and the inductive hypothesis $\text{HD}_{\preceq}(\text{LT}_{\ll}(\eta_i)) \preceq \text{HD}_{\preceq}(\eta)$.

Now since we have a lexicographic Gröbner basis that uses the compatible lex ordering, we have the property $\text{HD}_{\preceq}(\text{LT}_{\ll}(g)) = \text{HD}_{\preceq}(g)$ for all $g \in \mathcal{N}$, so in particular we have $\text{HD}_{\preceq}(g_k) \preceq \text{HD}_{\preceq}(\eta)$.

If we now set $c := \text{LT}_{\ll}(\eta_i)/\text{LT}_{\ll}(g_k)$ (which is a monomial in $\Delta(\mathcal{N})$ as $\text{LT}_{\ll}(g_k)$ divides $\text{LT}_{\ll}(\eta_i)$), we can also see from the inductive hypothesis that $\text{HD}_{\preceq}(c) \preceq \text{HD}_{\preceq}(\eta)$. If we now update η as $\eta_{i+1} := \eta_i - cg_k$, and set $h_k := h_k + c$, we can observe the following.

1. $\eta_{i+1} \in \text{Id}(\mathcal{N})$ from the properties of ideals and the fact that η_i,$g_k \in \text{Id}(\mathcal{N})$.

2. η_{i+1} satisfies $\text{HD}_{\preceq}(\eta_{i+1}) \preceq \text{HD}_{\preceq}(\eta)$ as it can be expressed as a combination of η_i, c, g_k, and from the inductive hypothesis we have $\text{HD}_{\preceq}(\eta_i) \preceq \text{HD}_{\preceq}(\eta)$, and we observed that $\text{HD}_{\preceq}(c) \preceq \text{HD}_{\preceq}(\eta)$, $\text{HD}_{\preceq}(g_k) \preceq \text{HD}_{\preceq}(\eta)$ earlier.

3. The only multiplier that changed is h_k, through addition of c, and $\text{HD}_{\preceq}(g_k) \preceq \text{HD}_{\preceq}(\eta)$, and $\text{HD}_{\preceq}(c) \preceq \text{HD}_{\preceq}(\eta)$, so this hypothesis continues to hold.

4. Only η and the h change during the step, so we let h represent the h values before the step, and $\tilde{h}$ the values after the step. Proceeding from the inductive hypothesis we have

$$\begin{aligned}
\eta &= \eta_i + \sum h_k g_k \\
&= \eta_i - c g_j + \sum_{k \neq j} h_k g_k + h_k g_j + c g_j \\
&= \eta_i - c g_j + \sum_{k \neq j} h_k g_k + (h_k + c) g_j \\
&= \eta_{i+1} + \sum_{k \neq j} \tilde{h}_k g_k + \tilde{h}_j g_j \\
&= \eta_{i+1} + \sum \tilde{h}_k g_k
\end{aligned}$$

as required.

So by induction the lemma is proved. □

The above Lemma allows us to state a stronger version of the nonlinear derivative Lemma 6.3.4 of [58] (Lemma 3) that avoids the restriction involving the maximal leading derivative for all $g \in \mathcal{N}$.

Theorem 6 *Let $\mathcal{N}$ be a lexicographic Gröbner basis as described in Lemma 4. Let g be a polynomial function of $\tilde{\Delta}$ such that g reduces to a special consequence of $\mathcal{N}$. Suppose that $\mathrm{IC}_{\preceq}(f, f')$ reduces to a special consequence of $\mathcal{N}$ for all $f, f' \in \mathcal{M}$ for which it is well-defined. Further suppose that $D_i \eta_j$ reduces to a special consequence of $\mathcal{N}$ for all $\eta_j \in \mathcal{N}$ and $i \in \mathbf{N}_n$. Then there exists a special consequence $\eta_{g,i}$ of $\mathcal{N}$ such that for all complete reductions μ, $D_i g \overset{\mu}{\longmapsto} \eta_{g,i}$ for all $i \in \mathbf{N}_n$.*

Proof We proceed by transfinite induction on Δ based on the rank of the leading derivative appearing in $D_i g$. Clearly this is true for $g = 0$, so we assume it is true for all functions with leading derivative lower in rank than δ, and consider $\mathrm{HD}_{\preceq}(D_i g) = \delta$. Two cases arise:

First we consider the case where g is a special consequence of $\mathcal{N}$, or $g = \sum h_j \eta_j$ where $\eta_j \in \mathcal{N}$ and the h_j are nonzero polynomial functions in $\tilde{\Delta}$. By the Leibniz rule, we have

$$D_i g = \sum \eta_j D_i h_j + \sum h_j D_i \eta_j,$$

for some η_j, $1 \leq j \leq r$. We note that the first sum above is easily shown to reduce to a special consequence of $\mathcal{N}$, as any reduction by $\mathcal{M}$ will leave the η_j unchanged. For the second sum, by assumption we know that $D_i\eta_j$ reduces to a special consequence of $\mathcal{N}$, so through straightforward application of Lemma 2 (which holds for general rankings) we know that the sum reduces to a special consequence of $\mathcal{N}$ as long as $D_\alpha f - D_{\alpha'} f'$ reduces to a special consequence for all $\text{HD}_\preceq(D_\alpha f) = \text{HD}_\preceq(D_{\alpha'} f') \preceq \max_j(\text{HD}_\preceq(\eta_j))$.

Application of Lemma 4 tells us that $\max_j(\text{HD}_\preceq(\eta_j)) \preceq \delta$ for $1 \le j \le r$, so we require the reduction of $D_\alpha f - D_{\alpha'} f'$ to a special consequence of $\mathcal{N}$ for $\text{HD}_\preceq(D_\alpha f) = \text{HD}_\preceq(D_{\alpha'} f') \preceq \delta$.

Now note that $D_\alpha f - D_{\alpha'} f' = D_\sigma \text{IC}_\preceq(f, f')$ for some σ, and $\text{HD}_\preceq(D_\alpha f - D_{\alpha'} f')$ is lower in the ranking than δ. Since by assumption we have that $\text{IC}_\preceq(f, f')$ reduces to a special consequence of $\mathcal{N}$ for all $f, f' \in \mathcal{M}$, then by the inductive hypothesis we have $D_\alpha f - D_{\alpha'} f' \longmapsto \eta_{f,f'}$ for some special consequence $\eta_{f,f'}$, and the case is complete.

We now consider the case where $g \longmapsto \eta_g$, $g \neq \eta_g$. This is proven by induction on the length of the minimal chain required to reduce g to a special consequence of $\mathcal{N}$. For the base case of the induction we use the case above, and consider g', $g' \neq g$ where $g \longmapsto g' \longmapsto \eta_g$. By the inductive hypothesis, we may assume that $D_i g' \longmapsto \eta_{g',i}$ for some special consequence $\eta_{g',i}$. Now we consider $\text{red}_\preceq(D_i g, (\alpha, f))$, which from Lemma 1 must have the form

$$\text{red}_\preceq(D_i g, (\alpha, f)) = D_i g' + h D_i D_\alpha f + k D_\alpha f,$$

where h,k are polynomial functions of $\tilde{\Delta}$ with leading terms that are of lower or equal rank as $\text{HD}_\preceq(g)$. It can be shown that either $\text{HD}_\preceq(D_i g') \prec \text{HD}_\preceq(D_i g)$ or $\text{HD}_\preceq(D_i D_\alpha f) \prec \text{HD}_\preceq(D_i g)$, so since $\text{HD}_\preceq(D_\alpha f) \prec \text{HD}_\preceq(D_i g)$ we can apply Lemma 2 twice giving $\text{red}_\preceq(D_i g, (\alpha, f)) \longmapsto \eta'$ as required.

Further, application of a similar argument as the base case allows us to weaken the conditions for $D_\alpha f - D_{\alpha'} f'$ to those stated in this theorem, specifically that $\text{IC}_\preceq(f, f')$ reduces to a special consequence of $\mathcal{N}$. Application of Theorem 2 tells us that since $D_i g$ reduces to a special consequence for some complete reduction, then it reduces to a special consequence for any complete reduction.

Now through application of the inductive hypothesis for this case we have that $D_i g \longmapsto \eta_{g,i}$ for $\text{HD}_\preceq(D_i g) = \delta$, which by transfinite induction tells us that $D_i g \longmapsto \eta_{g,i}$ in general, as required. □

The following theorem is related to Theorem 6.3.3 of [58] (Theorem 4), applied to polynomial systems with the nonlinear part in lexicographic Gröbner basis form.

Theorem 7 *Let $\mathcal{N}$ be a lexicographic Gröbner basis as described in Lemma 4, and suppose that for each pair $(f, f') \in \mathcal{M}^2$ with $\mathrm{IC}_{\preceq}(f, f')$ well-defined there exists an expansion of the form*

$$\mathrm{IC}_{\preceq}(f, f') = \sum_{(b,b') \in B_(f,f')} D_{\alpha_{f,f',b,b'}} \mathrm{IC}_{\preceq}(b, b').$$

Here $B_{f,f'}$ is a subset of $\mathcal{M}^2$ such that for each $(b, b') \in B_{f,f'}$, there exists a special consequence $\eta_{b,b'}$ of $\mathcal{N}$ with $\mathrm{IC}_{\preceq}(b, b') \longmapsto \eta_{b,b'}$ and $D_{\alpha_{f,f',b,b'}} \mathrm{LCM}(b, b') \preceq \mathrm{LCM}(f, f')$. Finally suppose that $D_i\eta$ reduces to a special consequence of $\mathcal{N}$ for all $\eta \in \mathcal{N}$. Then $(\mathcal{M}, \mathcal{N})$ is in rif'-form.

Proof The proof of this theorem is identical to that of Theorem 6.3.3 of [58] (Theorem 4), with Theorem 6 used in place of Lemma 6.3.4 (Lemma 3). We omit repetition of this proof for brevity.

We note here that the requirements of this theorem more closely resemble Theorem 5.3.1 of [58] (Theorem 1) for linear systems than the nonlinear version in Theorem 6.3.3 of [58] (Theorem 4), as the only integrability conditions required are those in B, and the ranking is not restricted to a sequential one.

This leads to the following straightforward Corollary that is related to Corollary 5.3.1 of [58] (Corollary 1) but also applies to nonlinear systems.

Corollary 3 *Let $\mathcal{N}$ be a lexicographic Gröbner basis as described in Lemma 4, and suppose that for each pair $(f, f') \in \mathcal{M}^2$ with $\mathrm{IC}_{\preceq}(f, f')$ well-defined there exists a special consequence $\eta_{f,f'}$ of $\mathcal{N}$ with $\mathrm{IC}_{\preceq}(f, f') \longmapsto \eta_{f,f'}$. Finally suppose that $D_i\eta$ reduces to a special consequence of $\mathcal{N}$ for all $\eta \in \mathcal{N}$. Then $(\mathcal{M}, \mathcal{N})$ is in rif'-form.*

Proof This corollary holds trivially setting B to all pairs $(f, f') \in \mathcal{M}^2$ with $\mathrm{IC}_{\preceq}(f, f')$ well-defined.

A constructive realization of Theorem 7 is developed in the following section.

Since the new result in the polynomially nonlinear case also provides a *rif'*-form, the nonlinear existence and uniqueness theorem (Theorem 6.4.1 of [58] or Theorem 5) applies directly.

2.4 Algorithmic Determination of Redundant Integrability Conditions

As discussed in Rust [58] and Reid *et al.* [50, 52], many of the all-pairs integrability conditions between solved-form equations are redundant.

An implementable *reduced* set of integrability conditions for the linear case was also presented in Rust [58] as Corollary 5.3.2 (Corollary 2) that uses Theorem 1 as a departure point. It is closely related to Buchberger's second criterion for redundant S-polynomials in Gröbner basis computations. Unfortunately, as described in Example 11 and the following commentary, the approach does not lend itself well to incremental algorithms.

In this section we develop an algorithm that can be used to compute a near minimal set of integrability conditions for polynomially nonlinear systems under general rankings that lends itself well to an incremental implementation.

As a basic outline of what follows, we will begin by introduction of a formal set of dependent variables $v_i^{d_i}$, one for each function f_i in $\mathcal{M}$. We will then write the all-pairs integrability conditions of Corollaries 1 and 3 in terms of these new formal variables, and construct an algorithm that provides a much smaller set of integrability conditions using Theorem 7 to prove its correctness.

Definition 9 (Syzygy Set and Associated Derivative) *For each $f_i \in \mathcal{M}$ ($\mathcal{M}$ is $\preceq$-monic) where $\text{HD}_{\preceq}(f_i) = \delta_{\alpha_i}^{d_i}$ we define a new formal dependent variable $v_i^{d_i}$ by $D_{\alpha_i} v_i^{d_i} := f_i$, where we note the subscript i allows identification of the function f_i from $\mathcal{M}$, and the α_i and d_i identify the leading derivative $\delta_{\alpha_i}^{d_i}$ of f_i.*

For all pairs $f_i, f_j \in \mathcal{M}$ with a well-defined integrability condition, we define $\beta_{ij} := \text{LCM}(\alpha_i, \alpha_j)$, and define $s_{ij} := D_{\beta_{ij}/\alpha_i} D_{\alpha_i} v_i^{d_i} - D_{\beta_{ij}/\alpha_j} D_{\alpha_j} v_j^{d_j}$ or by abuse $s_{ij} := D_{\beta_{ij}} v_i^{d_i} - D_{\beta_{ij}} v_j^{d_j}$. The syzygy set *$\mathcal{S}$ is then defined as the set of all s_{ij} obtained in this way.*

Here we note that for f_i, f_j to have a well-defined integrability condition it is necessary that the leading derivatives involve the same dependent variable δ^{d_i} (i.e. $d_i = d_j$). We define the associated derivative *of each function $s_{ij} \in \mathcal{S}$ as $\mathcal{D}_s := \delta_{\beta_{ij}}^{d_i}$.*

Since the v_i are simply formal notation, it should be understood that any reduction operations require replacement of the formal variables by their corresponding functions before application of the reduction process.

The functions in the syzygy set are linear in the formal variables $D_{\alpha_i} v_i^{d_i}$, and all have exactly two terms. Conceptually, the syzygy set can be thought of as a representation of the all-pairs integrability conditions that must be satisfied for $\mathcal{M}$ to be a Riquier basis (Corollary 1) or for $\mathcal{M} \cup \mathcal{N}$ to be a *rif'*-form (Corollary 3). This fact is established in Lemma 5 below.

Lemma 5 *If all elements $s \in \mathcal{S}$ from Definition 9 satisfy $s \longmapsto \eta$ for some special consequence $\eta \in \mathcal{N}$, then we have $\mathrm{IC}_{\preceq}(f, f') \longmapsto \eta_{f,f'}$ for some special consequence $\eta_{f,f'} \in \mathcal{N}$ for all $f, f' \in \mathcal{M}$ for which $\mathrm{IC}_{\preceq}(f, f')$ is well defined.*

Proof Consider an arbitrary element of $s_{ij} \in \mathcal{S}$. By Definition 9 we have

$$\begin{aligned} s_{ij} &= D_{\beta_{ij}} v_i - D_{\beta_{ij}} v_j \\ &= D_{\beta_{ij}/\alpha_i} f_i - D_{\beta_{ij}/\alpha_j} f_j \\ &= \mathrm{IC}_{\preceq}(f_i, f_j), \end{aligned}$$

where in the second step above we applied the unique definitions of v_i, v_j, and in the final step we applied the definition of the integrability condition. So since $s_{ij} = \mathrm{IC}_{\preceq}(f_i, f_j)$, we have that $s_{ij} \longmapsto \eta_{ij}$ implies $\mathrm{IC}_{\preceq}(f_i, f_j) \longmapsto \eta_{ij}$ for each s_{ij}.

Now since s_{ij} is arbitrary, and there is an $s \in \mathcal{S}$ for every well-defined integrability condition, we have that all integrability conditions of Corollary 3 reduce to special consequences of $\mathcal{N}$ as required. $\square$

The syzygy set provides us a model for the integrability conditions of the functions $\mathcal{M}$ that is easier to work with than $\mathcal{M}$ itself (both from a conceptual point of view, and for algorithm design). As long as we prove that operations we perform with the syzygy system obey Theorem 7 for $\mathcal{M}$, then we can be sure that the results carry over to the $\mathcal{M}$ functions.

In the following we will treat the syzygy set as a system of differential equations in the formal variables, and devise an algorithm that removes redundant all-pairs integrability conditions for $\mathcal{M}$. We first need to define a ranking on $\mathcal{S}$.

Definition 10 (Compatible Ranking) *Let $\mathcal{P}$ be any fixed permutation of $\mathbf{N}^{\#\mathcal{M}}$. We define a compatible ranking $\preceq'$ for the syzygy set $\mathcal{S}$ based on the ranking $\preceq$ on $\mathcal{M}$ and $\mathcal{P}$ as follows.*

For any two derivatives in the syzygy set, $D_\alpha v_i^k, D_\beta v_j^l$ we have

1 $\delta_\alpha^k \succeq \delta_\beta^l \Rightarrow D_\alpha v_i^k \succeq' D_\beta v_j^l$

2 $\delta_\alpha^k \preceq \delta_\beta^l \Rightarrow D_\alpha v_i^k \preceq' D_\beta v_j^l$

3 *if i occurs before j in $\mathcal{P}$ then $D_\alpha v_i^k \succeq D_\beta v_j^l$*

4 *if i occurs after j in $\mathcal{P}$ then $D_\alpha v_i^k \preceq' D_\beta v_j^l$*

It is easy to see that if the application of the ranking $\preceq'$ to $D_\alpha v_i^k, D_\beta v_j^l$ reaches step 3, then we must have $\alpha = \beta$ and $k = l$, as otherwise the ranking $\preceq$ would not be a total ordering. At which point, for $D_\alpha v_i^k \neq D_\beta v_j^l$ we must have $i \neq j$, from which it is clear that $\preceq'$ is a total ordering on the derivatives of the formal variables v_i occurring in $\mathcal{S}$.

The following will assist greatly in the proof of the redundancy criteria:

Lemma 6 *Consider three functions s_{jk}, s_{jl}, s_{kl} from the syzygy set $\mathcal{S}$ involving the three formal variables $v_j^{d_j}, v_k^{d_k}, v_l^{d_l}$ defined from $f_j, f_k, f_l \in \mathcal{M}$, and note that it must be the case that $d_j = d_k = d_l = d$. Further suppose that $D_{\beta_{jk}} v_j^d \succeq' D_{\beta_{jk}} v_k^d$ and $D_{\beta_{jl}} v_j^d \succeq' D_{\beta_{jl}} v_l^d$. Assume that there exists α such that $D_{\beta_{jk}} = D_\alpha D_{\beta_{jl}}$.*

Then there exists ρ such that $\beta_{jk} = \rho\beta_{kl}$, and $D_{\beta_{kl}} v_k^d, D_{\beta_{kl}} v_l^d$ are of strictly lower rank than $D_{\beta_{jk}} v_j^d$ under any compatible ranking $\preceq'$ from Definition 10.

Proof To begin we write out the functions from the syzygy set explicitly

$$\begin{aligned} s_{jk} &= D_{\beta_{jk}} v_j^d - D_{\beta_{jk}} v_k^d, \\ s_{jl} &= D_{\beta_{jl}} v_j^d - D_{\beta_{jl}} v_l^d, \\ s_{kl} &= D_{\beta_{kl}} v_k^d - D_{\beta_{kl}} v_l^d. \end{aligned}$$

Now consider the origin of the β in the above syzygy functions. We have

$$\beta_{jk} = \mathrm{LCM}(\alpha_j, \alpha_k), \quad \beta_{jl} = \mathrm{LCM}(\alpha_j, \alpha_l), \quad \beta_{kl} = \mathrm{LCM}(\alpha_k, \alpha_l).$$

If we define

$$\begin{aligned}\gamma_{jkl} = \textsc{gcd}(\alpha_j,\alpha_k,\alpha_l),\quad \gamma_{jk} &= \textsc{gcd}(\alpha_j,\alpha_k)/\gamma_{jkl}, & \gamma_j &= \alpha_j/(\gamma_{jk}\gamma_{jl}\gamma_{jkl}),\\ \gamma_{jl} &= \textsc{gcd}(\alpha_j,\alpha_l)/\gamma_{jkl}, & \gamma_k &= \alpha_k/(\gamma_{jk}\gamma_{kl}\gamma_{jkl}),\\ \gamma_{kl} &= \textsc{gcd}(\alpha_k,\alpha_l)/\gamma_{jkl}, & \gamma_l &= \alpha_l/(\gamma_{jl}\gamma_{kl}\gamma_{jkl}),\end{aligned}$$

then it is easy to show that

$$\begin{aligned}\alpha_j &= \gamma_j\gamma_{jk}\gamma_{jl}\gamma_{jkl}, & \beta_{jk} &= \gamma_j\gamma_k\gamma_{jk}\gamma_{jl}\gamma_{kl}\gamma_{jkl},\\ \alpha_k &= \gamma_k\gamma_{jk}\gamma_{kl}\gamma_{jkl}, & \beta_{jl} &= \gamma_j\gamma_l\gamma_{jk}\gamma_{jl}\gamma_{kl}\gamma_{jkl},\\ \alpha_l &= \gamma_l\gamma_{jk}\gamma_{jl}\gamma_{jkl}, & \beta_{kl} &= \gamma_k\gamma_l\gamma_{jk}\gamma_{jl}\gamma_{kl}\gamma_{jkl}.\end{aligned}$$

Now by hypothesis, $\beta_{jk} = \alpha\beta_{jl}$, which expressed in terms of the γ's yields $\gamma_j\gamma_k\gamma_{jk}\gamma_{jl}\gamma_{kl}\gamma_{jkl} = \alpha\gamma_j\gamma_l\gamma_{jk}\gamma_{jl}\gamma_{kl}\gamma_{jkl}$ and simplifies to $\gamma_k = \alpha\gamma_l$ after removal of common factors. By means of their construction, $\textsc{gcd}(\gamma_k,\gamma_l) = 1$, so we must have $\gamma_k = \alpha$ and $\gamma_l = 1$. Making the notational simplification $\sigma = \gamma_{jk}\gamma_{jl}\gamma_{kl}\gamma_{jkl}$ we can rewrite the functions from the syzygy set as

$$\begin{aligned}s_{jk} &= D_{\gamma_j\alpha\sigma}v_j^d - D_{\gamma_j\alpha\sigma}v_k^d,\\ s_{jl} &= D_{\gamma_j\sigma}v_j^d - D_{\gamma_j\sigma}v_l^d,\\ s_{kl} &= D_{\alpha\sigma}v_k^d - D_{\alpha\sigma}v_l^d.\end{aligned}$$

It is clear from the above that ρ exists and $\rho = \gamma_j$.

Now there are two possibilities for the leading term of the s_{kl} function.

Case 1: $\textsc{hd}_{\preceq'}(s_{kl}) = D_{\alpha\sigma}v_k^d$

By assumption (from the leading term of s_{jk}), we know $D_{\gamma_j\alpha\sigma}v_j^d \succ' D_{\gamma_j\alpha\sigma}v_k^d$ (where the inequality is strict as $j \neq k$). By positivity of the ranking, we have $D_{\gamma_j\alpha\sigma}v_k^d \succeq' D_{\alpha\sigma}v_k^d$, so by transitivity the conclusion holds.

Case 2: $\textsc{hd}_{\preceq'}(s_{kl}) = D_{\alpha\sigma}v_l^d$

By assumption (from the leading term of s_{jl}), we know that $D_{\gamma_j\sigma}v_j^d \succ' D_{\gamma_j\sigma}v_l^d$ (where the inequality is strict as $j \neq l$). This implies $D_{\gamma_j\alpha\sigma}v_j^d \succ' D_{\gamma_j\alpha\sigma}v_l^d$ by invariance of the ranking under differentiation. By positivity of the ranking, we have $D_{\gamma_j\alpha\sigma}v_l^d \succeq' D_{\alpha\sigma}v_l^d$, so by transitivity the conclusion holds.

So we have $\text{HD}_{\preceq'}(s_{jk})$ is strictly higher in rank than $\text{HD}_{\preceq'}(s_{kl})$, and the theorem is proved. □

Now the foundation has been laid to algorithmically remove redundant integrability conditions from the all-pairs integrability conditions described in Theorems 1 and 7.

Algorithm 9 (Syzygy Simplify)
Input: A syzygy system $\mathcal{S}$ with $\#\mathcal{S}$ elements
Output: A syzygy system B such that all elements $s \in \mathcal{S}$ can be written as a linear combination of derivatives of $b \in B$.

1 *Rank all elements $s \in \mathcal{S}$ in order of increasing rank based on $\text{HD}_{\preceq'}(s)$, breaking ties by the rank of the non-leading derivative in s. We label these $s_i, 1 \le i \le \#\mathcal{S}$.*

2 *Set $B = \emptyset$, $\mathcal{S}' = \emptyset$*

3 *Loop i from 1 to $\#\mathcal{S}$*

3.1 *If there exists a function $s' \in \mathcal{S}'$ such that $\text{HD}_{\preceq'}(s_i) = D_\alpha \text{HD}_{\preceq'}(s')$ for some α then reject s_i, and goto 3.4.*

3.2 *If there exist functions $s', s'' \in \mathcal{S}'$ such that $s_i = D_{\alpha'}s' - D_{\alpha''}s''$ for $\beta = \text{LCM}(\text{HD}_{\preceq'}(s'), \text{HD}_{\preceq'}(s''))$, $\alpha' = \beta/\text{HD}_{\preceq'}(s')$, $\alpha'' = \beta/\text{HD}_{\preceq'}(s'')$ then reject s_i and goto 3.4.*

3.3 $B = B \cup \{s_i\}$

3.4 $\mathcal{S}' = \mathcal{S}' \cup \{s_i\}$

4 *return B*

The rejection criteria in step 3.1 of the algorithm closely resembles application of reduction to the syzygy system, while the criteria in step 3.2 removes integrability conditions of the syzygy system itself.

Theorem 8 (Redundancy Criteria) *Given that the input $\mathcal{S}$ to Algorithm 9 is constructed as described in Definition 9, then the integrability conditions corresponding to the output B form a generating set of the set of integrability conditions $\mathcal{S}$ required for Theorems 1 and 7.*

Proof To prove that Algorithm 9 produces a generating set of integrability conditions as described by Theorems 1 and 7 we proceed by induction on the subsystem formed at each step, demonstrating that every rejected function can be represented as a linear combination of derivatives of the current B of the required form.

One key observation required here is the fact that the $\mathcal{S}$ are sorted in strictly increasing order with respect to $\preceq'$, which also means that they are sorted in increasing order (though not strict) of their associated derivative in Δ.

This initial case is clearly satisfied, as $\mathcal{S}'$ is empty, so s_1 is accepted. We now assume this is true for all $s_1, ..., s_{i-1}$, and consider s_i in the current step. There are three cases.

Case 1: s_i is rejected based on the criteria of step 3.1.

We write out s_i as $s_i = D_{\beta_{jk}} v_j^d - D_{\beta_{jk}} v_k^d$, and assume without loss of generality that $\mathrm{HD}_{\preceq'}(s_i) = D_{\beta_{jk}} v_j^d$. Then s' must have the form $s' = D_{\beta_{jl}} v_j^d - D_{\beta_{jl}} v_l^d$ where $\mathrm{HD}_{\preceq'}(s') = D_{\beta_{jl}} v_j^d$ for some l, and $D_{\beta_{jk}} v_j^d = D_\alpha D_{\beta_{jl}} v_j^d$. Now consider $t = D_{\beta_{kl}} v_k^d - D_{\beta_{kl}} v_l^d$ and apply Lemma 6 with α, $s_{jk} = s_i$, $s_{jl} = s'$, and $s_{kl} = t$ which gives us $\mathrm{HD}_{\preceq'}(s_i)$ is of greater rank than $\mathrm{HD}_{\preceq'}(t)$. So by the ranking above, and by the inductive hypothesis we know there exist expansions of the form required for Theorems 1 and 7 for s', t in terms of B.

Now to show that the expansion for s_i of the correct form, we simply observe that from the hypothesis of the theorem that $\beta_{jk} = \alpha\beta_{jl}$, and the application of Lemma 6 gives us a ρ such that $\beta_{jk} = \rho\beta_{kl}$, resulting in the expansion $s_{jk} = D_\alpha s_{jl} - D_\rho s_{kl}$. Now by the definition of compatible ranking (Definition 10) the associated derivatives for s_{jk},$D_\alpha s_{jl}$, and $D_\rho s_{kl}$ are equal, so the rank inequality for Theorem 1 holds with equality, and we have $\mathrm{LCM}(\mathrm{HD}_\preceq(f_j), \mathrm{HD}_\preceq(f_l)) = D_\alpha \mathrm{LCM}(\mathrm{HD}_\preceq(f_k), \mathrm{HD}_\preceq(f_l)) = D_\rho \mathrm{LCM}(\mathrm{HD}_\preceq(f_j), \mathrm{HD}_\preceq(f_l))$.

Finally, since $s', t \in \mathcal{S}'$ we know that they are either in B or can be obtained as linear combinations of derivatives of elements of B. In the latter case, the inductive hypothesis states that each element of $\mathcal{S}'$ can be constructed from derivatives of elements of B satisfying Theorems 1 and 7. This implies, by invariance of the ranking under differentiation and transitivity, that the rank of the associated derivatives of the elements of B are also equal or lower in rank than $\mathrm{LCM}(\mathrm{HD}_\preceq(f_j), \mathrm{HD}_\preceq(f_l))$, which completes this case.

Case 2: s_i is rejected based on the criteria of step 3.2.

We write out s_i as $s_i = D_{\beta_{kl}} v_k^d - D_{\beta_{kl}} v_l^d$. Now from step 3.2 there must exist $s', s'' \in \mathcal{S}'$

such that $s' = D_{\beta_{jk}} v_j^d - D_{\beta_{jk}} v_k^d$ and $s'' = D_{\beta_{jl}} v_j^d - D_{\beta_{jl}} v_l^d$ for some j. Using the α',α'', and β defined in that step, we obtain

$$\begin{aligned}
D_{\alpha''} s'' - D_{\alpha'} s' &= D_{\alpha''}(D_{\beta_{jl}} v_j^d - D_{\beta_{jl}} v_l^d) - D_{\alpha'}(D_{\beta_{jk}} v_j^d - D_{\beta_{jk}} v_k^d) \\
&= D_{\alpha''\beta_{jl}} v_j^d - D_{\alpha''\beta_{jl}} v_l^d - D_{\alpha'\beta_{jk}} v_j^d + D_{\alpha'\beta_{jk}} v_k^d \\
&= D_{\alpha'\beta_{jk}} v_k^d - D_{\alpha''\beta_{jl}} v_l^d \\
&= D_{\beta} v_k^d - D_{\beta} v_l^d
\end{aligned}$$

where the last line is obtained from the definition of β, as $\beta = \alpha'\beta_{jk} = \alpha''\beta_{jl}$. Now by the requirement that the resulting function is the same as s, we have $\beta = \beta_{kl}$. In terms of the associated functions in $\mathcal{M}$ we have $\text{LCM}(\text{HD}_{\preceq}(f_k), \text{HD}_{\preceq}(f_l)) = D_{\alpha'} \text{LCM}(\text{HD}_{\preceq}(f_j), \text{HD}_{\preceq}(f_k)) = D_{\alpha''} \text{LCM}(\text{HD}_{\preceq}(f_j), \text{HD}_{\preceq}(f_l))$ so the non-strict inequality surely holds, and there is a correct representation for s_i in terms of functions in $\mathcal{S}'$.

The completion of this case is accomplished through a similar argument as Case 1, from which we conclude that there is an expansion of the required form even for $s, s' \notin B$, which completes this case.

Case 3: s_i is accepted

This case follows trivially.

So from the syzygy set $\mathcal{S}$ we have constructed a subset B of conditions satisfying Theorems 1 and 7, as required. □

And finally the following obvious Lemma.

Lemma 7 *Let $\mathcal{N}$ be a lexicographic Gröbner basis as described in Lemma 4, and suppose that for every $b \in B$, for B constructed via Algorithm 9 from $\mathcal{S}$ defined for $\mathcal{M}$, the corresponding integrability condition reduces to a special consequence η_b of $\mathcal{N}$. Further, suppose that $D_i\eta$ reduces to a special consequence of $\mathcal{N}$ for all $\eta \in \mathcal{N}$. Then $(\mathcal{M},\mathcal{N})$ is in rif'-form.*

Proof The proof follows trivially from Theorem 7, the definition of $\mathcal{S}$, and application of Theorem 8. □

We now revisit Example 11 from §2.2, applying the new algorithm for removal of redundant integrability conditions.

Example 14 *Consider example 11. Using Algorithm 9 with any permutation $\mathcal{P}$ used to define $\preceq'$ gives us exactly the four non-redundant integrability conditions identified for that example, namely $D_y e_1 - D_x e_2$, $D_y e_2 - D_x e_3$, $D_y e_3 - D_x e_4$, and $D_y e_4 - D_x e_5$.*

So we have obtained a significant improvement over the criteria of Corollary 2 (as we need not concern ourselves with the order of removal), and devised an algorithm to apply it. Note that this development holds for polynomially nonlinear systems with an arbitrary ranking, i.e. we are not restricted to use of a sequential ranking in the nonlinear case.

In addition, these criteria have flexibility in the choice of the permutation used for the syzygy ranking $\preceq'$, which we can utilize to create an even more efficient implementation.

The process in Algorithm 9, and the permutation for the compatible ranking $\preceq'$ can be further simplified by a partitioning of the syzygy system based on the dependent variable from the $f_i \in \mathcal{M}$ from which the syzygys are defined. It is straightforward to see that this is valid, as there are no well-defined integrability conditions that have leading derivatives of different dependent variables. A separate permutation can then be obtained for each dependent variable, which does not affect the required conditions on the compatible ranking.

With this simplification, the nature of the freedom afforded us by the partition becomes clearer. Quite simply, the permutation imposes a ranking on the functions in $\mathcal{M}$. The nature of the algorithm will act to eliminate as many integrability conditions as possible involving the highest ranked function. So, for example, one could rank the most complex expression in $\mathcal{M}$, say $\bar{f}$, as the highest rank, thus ensuring as few as possible integrability conditions involve $\bar{f}$ and it's derivatives.

Finally we close this section with the following conjecture, which has been experimentally verified to hold for all problems in the benchmark set [72]:

Conjecture 1 (Incremental Conjecture) *The Syzygy Simplify algorithm (Algorithm 9) can be efficiently utilized to remove redundant integrability conditions of a system* as it is being reduced *under the following conditions:*

1. *The syzygy equations for integrability conditions that have already been taken are retained.*

2. *Leading linear equations are never removed from $\mathcal{M}$, except through reduction of their*

leading derivative by a leading linear equation with a leading derivative of lower order.

In the event that the conditions above are satisfied, then the additional conditions required as a result of the addition of a new leading linear equation to the system can be accounted for by appending the new syzygy conditions to the set already known, and re-running the algorithm.

2.5 Termination for the Sequential Ranking Case

In this section we provide the modifications of the *poly_rif'* algorithm (Algorithm 8) and the *poly_modJanet* algorithm (Algorithm 7) to use the new integrability condition redundancy criteria. We then demonstrate the problem with the use of these algorithms with a non-sequential ranking, namely non-termination. Finally we prove that for a sequential ranking, the modified algorithms do terminate.

First we present the modified algorithms.

Algorithm 10 (poly_rif_ICs')

Input: *Two finite sets of functions $\mathcal{S}$,Λ polynomial in $\tilde{\Delta}$.*

Output: *A triple $(\mathcal{S}', \Lambda', \mathit{flag})$ such that $\mathcal{S}'$ is equivalent to $\mathcal{S}$ subject to Λ' and $\mathcal{S}'$ is either a reduced Riquier basis, inconsistent, or a rif'-form, as indicated by flag.*

$\mathcal{S}' := \mathcal{S},\ \Lambda' := \Lambda$
repeat
    $(\mathcal{S}', \Lambda', \mathit{flag}) := \mathit{poly\_JanetICs}(\mathcal{S}', \Lambda')$
    $\mathcal{N} := N_{\preceq}(\mathcal{S}')$
    $\mathcal{N} := \mathit{Gröbner\_lex}(\mathcal{N})$
    $\mathcal{S}' := (\mathcal{S}' \setminus N_{\preceq}(\mathcal{S}')) \cup \mathcal{N}$
    if $\mathit{flag} \neq$ *unsuccessful then*
        output $(\mathcal{S}', \Lambda', \mathit{flag})$
    end if
    $J := \bigcup_{j=1}^{m} D_j \mathcal{N}$
    $J := \mathit{Gröbner\_reduce}(\mathit{cred}_{\preceq}(J, L_{\preceq}(\mathcal{S}')), \mathcal{N})$
    $\mathcal{S}' := \mathcal{S}' \cup J$

13 *until* $J = \emptyset$

14 *output* $(\mathcal{S}', \Lambda', \textit{rif'-form})$

In addition, the *poly_modJanet* algorithm (Algorithm 7) used by *poly_rif'* must be modified to remove the redundant integrability conditions, but the only change is to replace the integrability conditions considered on line 5 of the algorithm by the modified set of integrability conditions from Algorithm 9, so for brevity we simply state this and label the modified algorithm *poly_JanetICs*.

Now consider application of *poly_rif_ICs'* to a specific system under a non-sequential ranking $\preceq$. We note that a related example appears in Rust's work (pp. 91 of [58]) though our restriction that $\mathcal{N}$ is in Gröbner basis form removes the problem for that example.

Example 15 *Consider the system in* $u(x)$ *and* $v(x)$

$$\begin{aligned} u_{xx} &= u \\ 0 &= v({u_x}^2 - u^2) \end{aligned}$$

under the non-sequential ranking that ranks u *and all of its derivatives higher than* v *and all of its derivatives (i.e.* $v \prec v_x \prec v_{xx} \prec \ldots \prec u \prec u_x \prec u_{xx} \prec \ldots$ *).*

In steps 10-11 of the algorithm, we obtain a new differential relation

$$\begin{aligned} (v({u_x}^2 - u^2))_x &= v_x({u_x}^2 - u^2) + v({u_x}^2 - u^2)_x \\ &= v_x({u_x}^2 - u^2) + 2vu_x(u_{xx} - u) \\ &\longmapsto v_x({u_x}^2 - u^2) \end{aligned}$$

where the last step holds by virtue of the linear relation $u_{xx} = u$*, so the algorithm does not terminate at this time.*

Since there is only one leading linear relation, the call to poly_JanetICs in step 2 of the algorithm has no effect on the system.

Now consider the algebraic s-polynomial between two equations of the form

$$\alpha({u_x}^2 - u^2) \quad , \quad \beta({u_x}^2 - u^2)$$

with α, β *such that* $\mathrm{GCD}(\alpha, \beta) = 1$ *and the leading derivatives of* α, β *are of lower rank than* u_x*. It is straightforward to see that this s-polynomial is identically zero.*

This means that in step 5 of the algorithm, no new algebraic relations are introduced.

It is easy to verify that in steps 10-11 of the algorithm, a new relation is introduced, and it is of the form

$$v_{xx}({u_x}^2 - u^2),$$

so the algorithm does not terminate at this time.

In fact the algorithm introduces a new relation of the form $v_{x^n}({u_x}^2 - u^2)$ *in the nth iteration, so this problem never terminates.*

The key to this example is the non-sequential ranking used for the given system. The *poly_rif_ICs'* algorithm terminates in a finite number of steps as long as the ranking is sequential, as will be shown in this section. Non-sequential rankings are considered in the following section.

Theorem 9 *Algorithm 10 terminates with either a rif'-form or an inconsistent system in a finite number of steps for any system under a sequential ranking* $\preceq$.

The proof of this theorem is quite similar to the proof of termination for the *rif'* algorithm in [58] and relies heavily on Dickson's Lemma. It is provided here for completeness.

Proof Clearly if the algorithm terminates, then the output will be either inconsistent, or will be a *rif'*-form (see Lemma 7). Further, the output system $\mathcal{S}'$ will be equivalent to the input system $\mathcal{S}$ subject to the output pivots Λ'. What remains to be shown is that the algorithm always terminates.

To prove termination, we show that $\mathrm{L}_{\preceq}(\mathcal{S}')$ and $\mathrm{N}_{\preceq}(\mathcal{S}')$ eventually stabilize. We let $(\mathcal{S}'_i, \Lambda'_i, \tilde{\Delta}_i)$ represent $(\mathcal{S}', \Lambda', \tilde{\Delta})$ after the ith iteration of the loop from 2-13. Application of Dickson's Lemma to $\mathrm{L}_{\preceq}(\mathcal{S}')$ tells us that it must eventually stabilize, so we set $\mathrm{L}_{\preceq}(\mathcal{S}') = \mathcal{L}$ for all $i \geq i_0$.

Now consider the set of parametric derivatives, all of whose first derivative are principal,

$$\{\delta \in \mathrm{Par}_{\preceq}(\mathcal{L}) \mid D_j\delta \in \mathrm{Prin}_{\preceq}(\mathcal{L})\, \forall\, j \in \mathbf{N}_m\}.$$

This set must be finite by the fact that $\mathcal{L}$ is finite. Set δ_0 to be the maximum in this set with respect to $\preceq$, and note that the set $\Delta^* := \{\delta \mid \delta \preceq \delta_0\}$ is finite for any sequential ranking $\preceq$.

Now $\mathrm{N}_{\preceq}(\mathcal{S}'_i)$ is a set of polynomials in $\{x\} \cup \Delta^*$ for all $i \geq i_0$, so it only depends upon a finite number of indeterminates.

The Ascending Chain Condition (Theorem 5.2.7 of Cox, Little and O'Shea [18]) tells us that any ascending chain of ideals in a finite number of indeterminates must eventually stabilize, so in particular, the chain of ideals

$$\langle \mathrm{N}_{\preceq}(\mathcal{S}'_{i_0}) \rangle \subset \langle \mathrm{N}_{\preceq}(\mathcal{S}'_{i_0+1}) \rangle \subset \langle \mathrm{N}_{\preceq}(\mathcal{S}'_{i_0+2}) \rangle \subset ...$$

must eventually stabilize.

Now since on line 11 of the algorithm we are reducing with respect to a Gröbner basis of the ideal $\langle \mathrm{N}_{\preceq}(\mathcal{S}') \rangle$, J is either empty, or enlarges the ideal, so by the Ascending Chain Condition $\mathrm{N}_{\preceq}(\mathcal{S}')$ is eventually stable for all $i \geq i_1$.

So since both $\mathrm{L}_{\preceq}(\mathcal{S}'_i)$ and $\mathrm{N}_{\preceq}(\mathcal{S}'_i)$ are stable for all $i \geq \max(i_0, i_1)$, then the algorithm must terminate, and the theorem is proved. □

We also have an existence uniqueness theorem for the output of Algorithm 10, as Theorem 5 (Theorem 6.4.1 of [58]) applies with U defined as the open subset of $\{x\} \cup \mathrm{Par}_{\preceq}(\mathrm{L}_{\preceq}(\mathcal{S}'))$ such that for $a \in U$, $a(\lambda) \neq 0$ for all $\lambda \in \Lambda'$.

The *poly_rif_ICs* algorithm does not always provide a *rif'*-form that describes all solutions to the input system.

Example 16 *Consider the following simple system in u(x),*

$$u(u_x - 1) = 0.$$

The output of the poly_rif_ICs algorithm for this system is

$$\mathcal{S}' = \{u_x = 1\}, \ \Lambda' = \{u \neq 0\}.$$

But defining $u(x) = 0$ is also a solution of the input system for u, and is inconsistent with the output rif'-form.

This difficulty can be remedied by performing a case splitting strategy whenever nontrivial pivots are introduced. One can define a case splitting version of Algorithm 10 in a similar manner as Rust [58].

Consider steps 11-12 of the *riq-autoreduce* algorithm, which read

11 $\mathcal{F} := (\mathcal{F} \setminus \{f\}) \cup \{\frac{f}{\mathrm{HC}_{\preceq}(f)}\}$

12 $\Lambda := \Lambda \cup \{\mathrm{HC}_{\preceq}(f)\}$.

These are the only steps of 10 in which a nontrivial pivot can be introduced (i.e. an assumption on the form of the system).

We can define a case splitting version of the algorithm as follows. Just prior to step 11, consider the form of $\mathrm{HC}_{\preceq}(f)$. If it does not depend upon Δ then we can proceed as before, otherwise we must split into two cases:

1 Assume $\mathrm{HC}_{\preceq}(f) \neq 0$, then we have as before.

2 Assume $\mathrm{HC}_{\preceq}(f) = 0$. Then step 11 becomes $\mathcal{F} := (\mathcal{F} \setminus \{f\}) \cup \{\mathrm{HC}_{\preceq}(f), f - \mathrm{HC}_{\preceq}(f)\mathrm{HD}_{\preceq}(f)\}$, and step 12 is not performed.

Application of this version of the algorithm results in a binary tree of *rif'*-forms. Proof of termination of this algorithm follows from termination of the *poly_rif_ICs'* algorithm and results related to Dickson's Lemma (see [58] pp. 104 for details).

2.6 Termination for the General Ranking Case

As shown in Example 15, the *poly_rif_ICs'* algorithm does not terminate in general for non-sequential rankings. In this section we will explore the problem more fully. We will provide a modification of the algorithm that does terminate for general rankings, at the cost of introducing a greater number of pivots. Finally we conclude the section with what we expect to be a less expensive but unproven modification of the algorithm that will be more fully investigated in future work.

One of the primary causes of the failure of the arbitrary ranking version of the algorithm to achieve termination can be understood from the proof of termination for the sequential ranking case. The gist of it is this:

Once the highest ranked derivative in the leading nonlinear equations has stabilized, then only a fixed number of indeterminates can appear in those leading nonlinear equations.

This is a direct consequence of the sequential ranking (by definition). This is not the case for non-sequential rankings though, as can be seen in Example 15. In that example, u

and all derivatives were of greater rank than v and any of its derivatives. In each iteration of the algorithm, a higher derivative of v was introduced, which was always of lower rank than u, so the number of indeterminates in the leading nonlinear equations grew in each iteration, even though the set of leading derivatives of the leading nonlinear equations was stable.

The first strategy that comes to mind is to change the way in which the nonlinear equations are handled, so that the introduction of a new relation, not in the ideal, restricts the nonlinear system with respect to the nonlinear leaders, even when that new relation contains new indeterminates. This suggests the use of triangular forms, which is in fact what we will apply here. A discussion of the use of triangular forms in differential elimination can be found in Hubert [28, 29].

Definition 11 (Triangular Form and Leading Coefficient) *Let $\mathcal{T}$ be a list of non-constant polynomials $t_1, ..., t_s$ over a finite set of unknowns $\Delta_t \subset \tilde{\Delta}$ with a fixed ranking $\preceq$. This list of polynomials is called a* Triangular Form *if*

$$\mathrm{HD}_{\preceq}(t_1) \prec \mathrm{HD}_{\preceq}(t_2) \prec ... \prec \mathrm{HD}_{\preceq}(t_s).$$

We define the Leading Coefficient *of any polynomial p in Δ_t with respect to an unknown $l \in \Delta_t$ as the leading coefficient of p viewed as a polynomial in l alone, and we write $\mathrm{LC}_{\preceq}(p, l)$.*

In order to obtain a triangular form, we must introduce *pseudo-reduction* (§1.9) into the *rif* algorithm, which is simply an extension of single variable polynomial division to the use of triangular forms. The ideas are implicit in the following algorithm.

Algorithm 11 (PseudoReduce)
Input: *A polynomial p in Δ_t, and a list of polynomials $\mathcal{T}$ in Δ_t that are in triangular form with respect to the ranking $\preceq$.*
Output: *p_r, the pseudo-reduction of p with respect to $\mathcal{T}$.*

$p_r := p$
for i from 1 to $\#\mathcal{T}$ loop
    $l := \mathrm{HD}_{\preceq}(\mathcal{T}_i)$

4 $\quad\quad d := degree(\mathcal{T}_i, l)$
5 $\quad\quad$ *while* $degree(p_r, l) >= d$ *do*
6 $\quad\quad\quad p_r := \mathrm{LC}_{\preceq}(\mathcal{T}_i, l)p_r - \mathrm{LC}_{\preceq}(p_r, l)l^{(degree(p_r,l)-d)}\mathcal{T}_i$
7 $\quad\quad$ *end while*
8 $\quad$ *end for*
9 $\quad$ *output* p_r

There is one property of pseudo-reduction which is of particular interest to us in the context of use with *rif'*-form algorithms.

Lemma 8 *Let $\mathcal{T}$ be a triangular form over the unknowns Δ and let p be a new polynomial in those unknowns. Let p_r be the pseudo-reduction of p with respect to $\mathcal{T}$ obtained by Algorithm 11. Then one of the following two conditions hold:*

1. $\mathrm{HD}_{\preceq}(p_r) \neq \mathrm{HD}_{\preceq}(t_i)\ \forall\ t_i \in \mathcal{T}$, *or*

2. $degree(p_r, \mathrm{HD}_{\preceq}(p_r)) < degree(t_i, \mathrm{HD}_{\preceq}(p_r))$ *for the polynomial* $t_i \in \mathcal{T}$ *for which* $\mathrm{HD}_{\preceq}(t_i) = \mathrm{HD}_{\preceq}(p_r)$.

Proof If $\mathrm{HD}_{\preceq}(p_r) \neq \mathrm{HD}_{\preceq}(t_i)\ \forall\ t_i \in \mathcal{T}$ then the lemma is trivially true, so we assume there exists some $t_i \in \mathcal{T}$ for which $\mathrm{HD}_{\preceq}(t_i) = \mathrm{HD}_{\preceq}(p_r)$.

Now consider the loop in steps 5-7 of the algorithm. Division of the polynomial p_r by t_i will proceed until $\mathrm{degree}(p_r, \mathrm{HD}_{\preceq}(t_i)) < \mathrm{degree}(t_i, \mathrm{HD}_{\preceq}(t_i))$, decreasing the degree by at least 1 in each iteration of the loop, so clearly this is true after completion of that loop.

Further, the remaining iterations of the outer loop (steps 2-8) only involve division of p_r by elements of $t_j \in \mathcal{T}$ with $\mathrm{HD}_{\preceq}(t_j) \prec \mathrm{HD}_{\preceq}(t_i)$, so clearly $\mathrm{degree}(\mathrm{LC}_{\preceq}(t_j), \mathrm{HC}_{\preceq}(t_i)) = \mathrm{degree}(t_j, \mathrm{HC}_{\preceq}(t_i)) = 0$, so the degree of p_r in $\mathrm{HD}_{\preceq}(p_r)$ can never increase, and the lemma is proved. $\square$

Now we introduce the concept of an *extended characteristic set* (additional information on this topic can be found in Wang [70], Chou [15] or Mishra [42]).

Definition 12 (Extended Characteristic Set and Pivots) *Let $\mathcal{N}$ be a finite nonempty set of polynomials in $\mathbf{Q}[\Delta_t]$, and let $Id(\mathcal{N})$ be the ideal generated by $\mathcal{N}$. A triangular form $\mathcal{T}$ (Definition 11), is called an extended characteristic set of $\mathcal{N}$ if either of the two following properties are satisfied:*

1 *$\mathcal{T}$ consists of a single polynomial in $\mathbf{Q} \cap Id(\mathcal{N})$, or*

2 *$\mathcal{T} = \langle t_1, .., t_s \rangle$ with $\textsc{hd}_{\preceq}(t_s) \in \Delta_t$ such that*

$$\begin{aligned} t_i \in Id(\mathcal{N}), &\qquad \forall\ i = 1..s \\ PseudoReduce(n_j, \mathcal{T}) = 0, &\qquad \forall\ j = 1..\#\mathcal{N} \end{aligned}$$

In the second case, we call

$$\Lambda = \{\textsc{lc}_{\preceq}(t_i, \textsc{hd}_{\preceq}(t_i)) \mid t_i \in \mathcal{T}\}$$

the pivots of $\mathcal{T}$.

Now we present a small modification of the algorithm from §5.5 of Mishra [42] (which can be shown to be equivalent to the *CharSet* algorithm described in Wang [70]) for computation of an extended characteristic set from a set of polynomials. Note that the modification is to simply include the complete set of all polynomials computed in the algorithm in addition to the extended characteristic set.

Algorithm 12 (ExtCharSet)
Input: *A set of polynomials $\mathcal{N}$.*
Output: *$\mathcal{T}$, an extended characteristic set of $\mathcal{N}$, the extended set of polynomials $\mathcal{N}$, and the set of pivots Λ.*

$\mathcal{T} := \emptyset$, $R := \emptyset$
repeat
    $\mathcal{N} := \mathcal{N} \cup R$, $\mathcal{N}' := \mathcal{N}$, $\mathcal{T} := \emptyset$, $R := \emptyset$
    while $\mathcal{N}' \neq \emptyset$ *loop*
        Choose $f \in \mathcal{N}'$ *with the lowest leader to the lowest degree*
        $\mathcal{N}' := \mathcal{N}' \setminus \{g \in \mathcal{N}' \mid \textsc{hd}_{\preceq}(g) = \textsc{hd}_{\preceq}(f)$ *and g is not reduced with respect to f*$\}$

        $\mathcal{T} := \mathcal{T} \cup \{f\}$
    end while
    for all $f \in \mathcal{N} \setminus \mathcal{T}$ *loop*
        if $r :=$ *PseudoReduce*$(f, \mathcal{T}) \neq 0$ *then*
            $R := R \cup \{r\}$
        end if
    end for
until $R = \emptyset$
$\Lambda := \{\text{LC}_{\preceq}(t_i, \text{HD}_{\preceq}(t_i)) \mid t_i \in \mathcal{T}\}$
return $(\mathcal{T}, \mathcal{N}, \Lambda)$

Our intent is to replace our use of Gröbner basis and Gröbner reduction methods in the *poly_rif_ICs'* algorithm (Algorithm 10) by use of extended characteristic sets and pseudo-reduction. It will be shown that this modification of the algorithm can still utilize the reduced integrability conditions developed in Section 2.4, and can be proven to terminate for the arbitrary ranking case.

One small modification is required in the definition of special consequence (Definition 8), to account for the pivots introduced by the extended characteristic set algorithm. This concept introduces a new form, which we call a *rif"*-form, and we describe this below.

Definition 13 (Special Consequence II) *Let U be a non-empty open subset of $\mathbf{F}^{\tilde{\Delta}}$ on which $\mathcal{M} \cup \mathcal{N}$ is analytic. For η an analytic function of $\mathbf{F}^{\tilde{\Delta}}$, we say that η is a special consequence of $\mathcal{N}$ if η admits an expansion of the form*

$$\lambda\eta = \sum_{i=1}^{k} h_i g_i$$

with $g_1, ..., g_k \in \mathcal{N}$ and $h_1, ..., h_k$ analytic functions on U that depend on $\{x\} \cup Par_{\preceq}\mathcal{M}$ only. Here λ is a product of analytic functions that are non-zero on U.

Clearly in the above definition, the λ corresponds to some power product of the pivots introduced in the extended characteristic set algorithm. All other development in the preceding sections holds in the same manner as for the original definition of special consequence, with replacement of *special consequence* by the new definition, and replacement of *rif'*-form by *rif"*-form.

This is only true, however, until we reach the development that is concerned with the form of the polynomially nonlinear equations. For that treatment we require the following lemma (which is a modification of Lemma 4 for extended characteristic sets).

Lemma 9 *Let $\mathcal{T}$ be the extended characteristic set constructed by application of Algorithm 12 to $\mathcal{N}$ under the ranking $\preceq$. Then for any η which is a special consequence of $\mathcal{N}$ there exist h_i, λ such that η can be represented as $\lambda\eta = \sum h_i t_i$ with $\mathrm{HD}_{\preceq}(t_i) \preceq \mathrm{HD}_{\preceq}(\eta)$ and $\mathrm{HD}_{\preceq}(h_i) \preceq \mathrm{HD}_{\preceq}(\eta)$ for all $i \in \phi$ where $\phi \subseteq \{1, ..., \#\mathcal{T}\}$, and $\mathrm{HD}_{\preceq}(\lambda) \preceq \mathrm{HD}_{\preceq}(\eta)$. Here λ is a power product of functions from Λ.*

Proof The proof follows fairly straightforwardly from the properties of extended characteristic sets, and the use of Algorithm 11.

If we label $P(\mathcal{T})$ as the set of all polynomials that are pseudo-reducible by $\mathcal{T}$, it is easy to see that $\mathrm{Id}(\mathcal{N}) \subseteq \mathrm{Id}(P(\mathcal{T}))$ (this follows from the fact that $\mathcal{N} \subseteq \mathrm{Id}(P(\mathcal{T}))$). So any polynomial in $\mathrm{Id}(\mathcal{N})$ pseudo-reduces to zero with respect to $\mathcal{T}$.

Now application of Algorithm 11 to η never introduces higher ranked derivatives than those already present in η, as for any division to occur, the corresponding indeterminate must occur in η. In addition, the multiplier in the pseudo-reduction can only involve indeterminates of rank lower or equal to the one being eliminated, so higher derivatives cannot occur in λ either. As a result, Algorithm 11 constructs the very form we require, so the lemma is proved. $\square$

Again, the remaining development continues to hold, with the algorithm for the *rif*"-form given as follows.

Algorithm 13 (poly_rif_ICs")
Input: *Two finite sets of functions $\mathcal{S}$,Λ polynomial in $\tilde{\Delta}$.*
Output: *A triple $(\mathcal{S}', \Lambda', flag)$ such that $\mathcal{S}'$ is equivalent to $\mathcal{S}$ subject to Λ' and $\mathcal{S}'$ is either a reduced Riquier basis, inconsistent, or a rif"-form, as indicated by flag.*

1 $\mathcal{S}' := \mathcal{S}$, $\Lambda' := \Lambda$, $\mathcal{N} := \emptyset$
2 *repeat*
3 $\quad\mathcal{S}' := \mathcal{S}' \cup \mathcal{N}$
4 $\quad(\mathcal{S}', \Lambda', flag) :=$ *poly_JanetICs*$(\mathcal{S}', \Lambda')$

5 $\quad\quad \mathcal{N} := N_{\preceq}(\mathcal{S}')$
6 $\quad\quad \mathcal{M} := \mathcal{S}' \setminus N_{\preceq}(\mathcal{S}')$
7 $\quad\quad (\mathcal{T}, \mathcal{N}, \Lambda'') := \mathit{ExtCharSet}(\mathcal{N})$
8 $\quad\quad$ *if flag* $\neq$ *unsuccessful then*
9 $\quad\quad\quad\quad$ *output* $(\mathcal{M} \cup \mathcal{T}, \Lambda' \cup \Lambda'', \mathit{flag})$
10 $\quad\quad$ *else if* $\mathcal{T} \neq \emptyset$ *and* $\mathrm{HD}_{\preceq}(\mathcal{T}) = \emptyset$ *then*
11 $\quad\quad\quad\quad$ *output* $(\mathcal{M} \cup \mathcal{T}, \Lambda' \cup \Lambda'', \boldsymbol{inconsistent})$
12 $\quad\quad$ *end if*
13 $\quad\quad J := \bigcup_{j=1}^{m} D_j \mathcal{N}$
14 $\quad\quad J := \mathit{cred}_{\preceq}(J, \mathcal{M})$
15 $\quad\quad \mathcal{N} := \mathcal{N} \cup J$
16 $\quad\quad J := \mathit{PseudoReduce}(J, \mathcal{T})$
17 *until* $J = \emptyset$
18 *output* $(\mathcal{M} \cup \mathcal{T}, \Lambda' \cup \Lambda'', $ *rif"-form*$)$

This modification of the algorithm requires a few additional comments.

First we note that the computation splits the system into leading nonlinear ($\mathcal{N}$), and leading linear ($\mathcal{M}$) parts in steps 5 and 6, providing a clearer description of how each part is used. Next we note that the characteristic set is computed from the nonlinear equations for each step, so if new equations are introduced into $\mathcal{N}$ that make a prior pivot from Λ'' unnecessary, then this is accounted for by the algorithm. Additionally, any leading linear relations that arise in the computation naturally become part of the linear system in the following step. Finally, on line 10, the extended characteristic set is checked to see if it consists of a polynomial containing no dependent variables, and if so, the computation is flagged as inconsistent.

Clearly upon completion of the algorithm, we will have a *rif"*-form, as all derivatives of the leading nonlinear equations must be special consequences (by Definition 13) of $\mathcal{T}$, and the linear part of the algorithm remains the same, so what remains to show is that the algorithm always terminates independently of the ranking. To do so, the following theorem is essential.

Theorem 10 *Let $\mathcal{T}$, $\mathcal{N}'$ be the extended characteristic set and extended system returned by Algorithm 12. Let $g \neq 0$ be a polynomial reduced with respect to $\mathcal{T}$. Then*

$$\mathit{ExtCharSet}(\mathcal{N}' \cup \{g\}) < \mathcal{T}.$$

Where in the above the symbol $<$ is used to denote that the smaller set is more restrictive than the larger one, specifically that the degrees of the leaders of each set are equal up to a leader that has different degree, and the lesser set is lower, or that the degrees of corresponding leaders are all equal, but the lesser set contains an additional relation that is not present in the larger set.

The above theorem follows directly from proposition 5.5.1 of Mishra [42], based on the observation that $\mathcal{T}$ is the minimal ascending chain of $\mathcal{N}'$.

In simple terms, this theorem tells us that addition of a new polynomial to $\mathcal{N}$ that does not pseudo-reduce with respect to $\mathcal{T}$ either lowers the degree with respect to the leading indeterminates, or adds a relation with a new leading indeterminate. As we will see, this is the key to the proof of termination of the *poly_rif_ICs"* algorithm.

Theorem 11 *Algorithm 13 terminates for arbitrary rankings.*

Proof The linear portion of the proof is identical to that of Theorem 9, and using the same argument, we can easily show that eventually the leaders of the leading nonlinear equations must stabilize. It is repeated here for completeness.

To prove termination, we show that $\mathcal{M}$ and $\mathcal{T}$ eventually stabilize. We let $(\mathcal{M}_i, \mathcal{T}_i, \tilde{\Delta}_i)$ represent $(\mathcal{M}, \mathcal{T}, \tilde{\Delta})$ after the ith iteration of the loop from 2-17. Application of Dickson's Lemma to $\mathcal{M}$ tells us that it must eventually stabilize, so we set $\mathcal{M}_s = \mathcal{M}_i$ for all $i \geq i_0$.

Now consider the set of parametric derivatives, all of whose first derivative are principal,

$$\Delta^* = \{\delta \in \mathrm{Par}_{\preceq}(\mathcal{M}_s) \mid D_j\delta \in \mathrm{Prin}_{\preceq}(\mathcal{M}_s)\, \forall j \in \mathbf{N}_m\}.$$

This set must be finite by the fact that $\mathcal{M}_s$ is finite. We also note that this set must contain the complete set of leaders of the nonlinear equations $\mathcal{N}$, as otherwise the differentiation in step 13 would introduce new leading linear equations.

Now consider what it means for the algorithm to fail to terminate. This would mean that new relations would need to be present in J at every step. This introduces two possibilities.

The first is that the new relation introduces a new leading derivative. This can only occur a finite number of times, specifically until all derivatives in Δ^* have been introduced, so this can only account for a finite number of steps. So consider the system at some step after all of the derivatives that arise from J have been considered.

The characteristic set will consist of equations with leaders from Δ_t, each having finite degree. If we now consider the introduction of a new relation in J, we see in step 16 of the algorithm that the new relation must be pseudo-reduced with respect to $\mathcal{T}$, so by Theorem 10 the addition of the new relation to $\mathcal{N}$ in the following iteration will result in a smaller characteristic set.

Under the assumption that the number of unknowns remains static, this means that the new relation will introduce a degree drop in one of the leaders when a new characteristic set is formed.

Since the degrees of the leaders are finite, this can only occur a finite number of times, so the algorithm must terminate, as required. □

Consider the effect of applying this algorithm to Example 15. The extended characteristic set $\mathcal{T}$ computed at the end of the first iteration is

$$u_x^2 = u^2,\ v \neq 0,$$

which immediately reduces the new relation $v_x(u_x^2 - u^2)$ to zero, so the algorithm terminates on the first step returning

$$u_{xx} = u,\ u_x^2 = u^2,\ v \neq 0.$$

This completes the analysis of the *poly_rif_ICs"* algorithm for the arbitrary ranking case, but a few observations need to be made.

The approach used was akin to swatting a fly with a sledgehammer. Far too much machinery had to be developed, and far too many changes needed to be made to achieve termination for the arbitrary ranking case.

It is believed by the author that a much simpler modification of Algorithm 10 can be used to assure termination for the arbitrary ranking case:

Conjecture 2 (Wittkopf Conjecture) *Consider Algorithm 10 with the following modification. Between steps 5 and 6 of the algorithm, introduce processing that performs the following two tasks:*

1 *If* $\exists\, g_1, g_2 \in \mathcal{N}$ *such that* $\mathrm{HD}_{\preceq}(g_1) = \mathrm{HD}_{\preceq}(g_2) = \mathrm{HD}_{\preceq}(gcd(g_1, g_2))$*, then introduce the smaller cofactor into the pivots* Λ*,* $gcd(g_1, g_2)$ *into* $\mathcal{N}$*, and recompute the lexicographic Gröbner basis in step 5.*

2 *If* $\exists\, g_1 \in \mathcal{N}$ *such that* $\mathrm{HD}_{\preceq}(content(g_1, \mathrm{HD}_{\preceq}(g_1)) \neq \emptyset$ *then introduce* $content(g_1, \mathrm{HD}_{\preceq}(g_1))$ *into the pivots* Λ*,* $g_1/content(g_1, \mathrm{HD}_{\preceq}(g_1))$ *into* $\mathcal{N}$*, and recompute the lexicographic Gröbner basis in step 5.*

Then the modified algorithm produces a rif'-form for arbitrary rankings in a finite number of steps.

Consider the *poly_rif_ICs'* algorithm applied with either one of the two modifications described above on the system of Example 15. It is easy to verify that in both cases the algorithm completes in a finite number of steps.

The analysis and proof of the algorithm obtained from Conjecture 2 is beyond the scope of this development, and will be considered in a later work, though this modification is the one in use by the *RifSimp* implementation, and has been successfully used on hundreds of problems. It should be noted that the current versions of the *rif* and *DiffElim* implementations utilize this conjecture, so this must be considered when viewing the timings obtained in the benchmarks chapter (7).

Chapter 3

The *rifsimp* Implementation and MaxDim Algorithm

3.1 Introduction to the *rifsimp* Implementation

The *rifsimp* command (§A.2) is a *Maple* implementation of the *RifSimp* algorithm for performing automated differential elimination on nonlinear systems of ODE or PDE. The core concepts behind the *RifSimp* algorithm are discussed in Chapters 1 and 2, but in this chapter we describe some of the more detailed aspects of its implementation.

Input: A system of ODE/PDE (Sys) and a ranking $\preceq_d$
Output: A tree structure containing all cases

rifsimp(Sys, $\preceq_d$)
  UnC := '=' from Sys (*unclassified equations*)
  Sol := $\emptyset$ (*solved form equations*)
  NLin := $\emptyset$ (*leading nonlinear equations or equations with a nontrivial pivot*)
  Piv := '$\neq$' from Sys (*all pivots currently in effect*)
  Case := [] (*current branch on binary case tree*)
  return *do_rifsimp*(UnC, Sol, Nlin, Piv, Case, $\preceq_d$)
end *rifsimp*

```
do_rifsimp(UnC, Sol, NLin, Piv, Case, ⪯_d)
    while UnC ≠ ∅ and Nlin not reduced do
        neq := select eqn from UnC that can be solved without introducing a
            new pivot (any eqn that require new pivots are placed in NLin)
        if neq is not null then
            neq   := solve neq for highest ranked derivative
            Sol   := self reduce Sol ∪ {neq}
            Piv   := reduce Piv with respect to Sol
            if pivots are violated then
                return [Case, 'system is inconsistent']
            end if
            UnC  := UnC ∪ {integrability conditions on Sol}
            UnC  := reduce UnC with respect to Sol
        else
            neq := select eqn from NLin with pivot
            if neq is not null then
                piv := pivot for neq
                return do_rifsimp(UnC ∪ {neq}, Sol, NLin, Piv ∪ {piv ≠ 0},
                    [Case,piv ≠ 0], ⪯_d) ⋃
                    do_rifsimp(UnC ∪ {neq, piv=0}, Sol, NLin, [Case, piv = 0], ⪯_d)
            else
                NLin  := nonlinear elimination on NLin
                Sol, Piv := nonlinear elimination on Sol, Piv with respect to NLin
                    (with usual checking for inconsistency)
                UnC  := {new spawn conditions on NLin}
                UnC  := reduce UnC with respect to Sol,NLin
            end if
        end if
    end while
    return ([Case, Sol, NLin, Piv])
end do_rifsimp
```

The result will be a set of disjoint cases for the system, split on the pivots, and each case may have a linear component (*Sol*), a nonlinear component (*NLin*), and inequations (*Piv*), along with a description of the assumptions made for that case (*Case*). Note that the *Case* output is provided for information only, as the reduced form of the assumptions will be present in *Sol*, *NLin* and *Piv*.

There are many aspects of *rifsimp* that are not addressed by the brief pseudocode description above. Most of these are modifications that improve the general efficiency of the algorithm, while others provide greater detail on how certain computations are performed. These aspects are described in the following sections.

3.2 Implementation Details

The pseudocode describes *rifsimp* as processing a single equation at a time. It was found in practice that this approach was somewhat inefficient (particularly for large systems with hundreds of equations), so the actual implementation selects a subset of the unclassified equations to add to the solved form list all at once. This subset is currently selected as a portion of the unclassified system that is of smaller complexity (currently based on the length of the equations) relative to the rest of the unclassified equations, limited by the number of equations already in solved form. For example, if the solved form list already contained one hundred equations, then *rifsimp* would only add a single equation at a time, as the number of new integrability conditions required for the resulting new solved equation could be large. If instead the solved form list was empty, then *rifsimp* would feel free to add more equations all at once, if the relative complexity of these equations does not differ by a factor of 2 or greater. This behavior can be modified using the *ezcriteria* option of *rifsimp* described in §A.6.

The implementation has been designed to be incremental in nature, remembering which integrability conditions (§1.4) have already been taken, and which nonlinear equations have already been spawned (§1.8). This makes a large difference in the practical implementation of the algorithm.

The default ranking ranks first by the differential order of derivatives, then by the individual differentiations, then by dependent variable name. Constants are ranked lower than dependent variables. Specification of a ranking for *rifsimp* can be a very involved process for systems with many dependent and independent variables. There are a number of shortcuts that can be used to specify the solving order of the dependent variables of the problem (see *vars* in §A.5), and order of preference for differentiation (see *indep* in §A.5), as well as the full specification available through the *ranking* argument (§A.6). A detailed description of available rankings and their specification with examples can be found in §A.7.

3.3 One-term Equations and Polynomial Decomposition

One strategy that has provided significant efficiency gains for the algorithm is a specialized treatment of one-term equations (equations of the form *indeterminate* $= 0$) combined with polynomial decomposition methods. This is best described through an example.

Example 17 *Consider the equation $f_{xx} = 0$, where $f = f(x, y)$. This equation tells us that $f_x, f_{xy}, f_{xyy}, \ldots$ are all constant with respect to x (or that f could be written as $f(x, y) = f_0(y) + x f_1(y)$). If in addition, the system contained an equation in $f(x, y)$, $g(y)$:*

$$f_x + g_y + x f_{xy} = 0$$

then the fact that f_x, f_{xy}, and g_y are independent of x allows us to decompose this equation into the two simpler equations

$$f_x + g_y = 0, \quad f_{xy} = 0.$$

Though this result can be obtained through a combination of differential elimination and integrability conditions, it can involve significant computation, and is more efficiently accomplished using decomposition techniques.

In general, any one-term equation introduced in a computation can have the effect of decomposing equations, so whenever additional one-term equations are discovered, all equations of the system are checked for decomposition.

3.4 More Leading Nonlinear Equations

Nonlinear differential equations present in the system can be treated algebraically by virtue of the spawning (§1.8) that is performed on them. This spawning accounts for all differential consequences of the nonlinear equations, allowing them to be treated simply as algebraic equations.

This allows use of many already existing algorithms for handling nonlinear algebraic equations, such as Gröbner Bases, Ritt-Wu decomposition, and resultant based elimination techniques.

The *rifsimp* implementation initially used Gröbner Bases techniques by default, as this allowed reduction of the nonlinear equations to a canonical form without performing additional case splitting. It was observed that often the direct use of Gröbner basis methods is highly inefficient for the nonlinear subsystems occurring during the differential elimination process. As a result, other nonlinear system strategies have been employed.

The current default strategy treats the polynomially nonlinear equations by isolating for the highest power of the leading derivative present in the equation. This approach often avoids system blow-up as a result of the many S-polynomials that may be required through a pure Gröbner treatment, as the isolated equations have fewer compatibility conditions with other nonlinear equations of the system. On the down side, this approach may require additional case splitting on the coefficient of the highest power of the leading derivative in each nonlinear equation. This coefficient is a new type of pivot called the initial (§1.8) of the equation. Before termination of each case, a Gröbner basis is formed over all nonlinear equations to be sure all additional relations in the now algebraic system are accounted for. This is somewhat more efficient than Gröbner bases on the subsystems occurring during the computation of a case as all nonlinear relations are available, not just a subset.

The original Gröbner basis strategy can still be used with the *rifsimp* implementation through use of the *grobonly* and *initial* options (§A.9).

Specification of a ranking for the Gröbner basis computation is quite involved, as it combines the ranking imposed on the derivatives (§1.2,§A.7), and a term ordering on monomials that are products of these derivatives. A high level description of the nonlinear rankings available for use in *rifsimp* can be found in §A.9.

Further, *rifsimp* also provides the *factoring* option (§A.8) to allow case splitting on the factored form of nonlinear equations. Given that splitting is already being performed on the initial and the leading linear coefficient of all derivatives of each nonlinear equation, the only remaining factorization will be one where all factors contain the leading derivative of the equation. Splitting on these factors has the effect of lowering the algebraic degree of the leading derivative for each of the split cases, which naturally results in a simplification for each case, at the cost of the additional case splitting. The default setting for *factoring* in *rifsimp* is one in which factoring will have no effect unless the leading derivative of the equation is not present in the factor. As discussed above, this will have no effect unless *initial* is off.

One additional consideration when working with leading nonlinear equations in combination with inequations is the possibility that all solutions of the nonlinear system are excluded by the inequations (a so-called empty case). Though the differential consequences of the leading nonlinear equations are taken care of through spawning, no such mechanism is in place for the inequations of the system. The primary difficulty with this is illustrated by the following:

Example 18 *Consider the following* ODE *and two inequations in $f(x), u(x)$:*

$$f_x = u^2(u-1),\ 3u-2 \neq 0,\ u_x \neq 0$$

The combination of these equations imply that

$$u \neq 0,\ f_{xx} \neq 0,\ f_x \neq 0,\ f \neq 0$$

because

$$u_x \neq 0 \Rightarrow u \neq 0$$

and

$$f_x = u^2(u-1) \Rightarrow f_{xx} = uu_x(3u-2)$$

Since all product terms in $uu_x(3u-2)$ are nonzero, then $f_{xx} \neq 0$. Also $f_x \neq 0$, $f \neq 0$ follow directly from $f_{xx} \neq 0$ using the same method as for $u \neq 0$ above.

The key fact is that new differential equations are obtained through differentiation of existing equations (pivots do not need to be considered), while obtaining new inequations requires integration of products of inequations considered relative to the equations of the system. In summary, inequations are *much* more difficult to find.

Modulo this difficulty, *rifsimp* has the *checkempty* option (§A.9) that determines when a case is empty, and discards it as an inconsistent case. This is done through application of a total degree Gröbner basis algorithm to the union of the nonlinear equations $n_1 = 0,\ n_2 = 0, ..., n_m = 0$, and the equation $sP - 1 = 0$, where s is a new variable, and $P = p_1 p_2 ... p_n$ is the product of the inequations for the case. This is called the quotient of the ideals $I = \langle n_1, n_2, ..., n_m \rangle$, $J = \langle P \rangle$, and is denoted $I : J$ (see Becker and Weispfenning [2] or Cox, Little and O'Shea [18]). It is well known that this case is empty if 1 is in this ideal, and this can always be discovered through the described Gröbner basis computation in finite time.

Note that this test is purely algebraic, so does not deal with the type of problem described in Example 18.

3.5 Case Splitting Strategy

The *rifsimp* implementation proceeds by performing as much elimination as possible without introducing new pivots. In other words, it performs computations using the leading linear part of the system, using only pivots that are already known (either from the input, or from a prior case splitting). Once no further computations can be performed, the nonlinear equations are examined for possible splitting. Note that nonlinear equations refers to both leading linear equations with nontrivial pivots and leading nonlinear equations.

One type of splitting not previously discussed is a factorization based splitting that is closely related to the leading linear coefficient of the equations. Specification of the *faclimit=n* option (§A.8) allows *rifsimp* to pivot on the leading derivative of an equation, instead of the coefficient of that leading derivative (thus reversing the expected order), but only if that derivative occurs as a factor in the user specified number of equations n.

Example 19 *For the system*

$$u_x(f_{xx} + {f_x}^2 f) = 0,\ u_x({f_x}^5 - f^3 f_x + f^4) = 0$$

where u is ranked strictly higher than f, the leading derivative of both equations is u_x. By *default, rifsimp would split this system into the three disjoint cases*

$$
\begin{array}{lll}
f_{xx} + {f_x}^2 f \neq 0, & & u_x = 0 \\
f_{xx} + {f_x}^2 f = 0, & {f_x}^5 - f^3 f_x + f^4 \neq 0, & u_x = 0 \\
f_{xx} + {f_x}^2 f = 0, & {f_x}^5 - f^3 f_x + f^4 = 0. &
\end{array}
$$

Specification of faclimit=2 allows rifsimp to pivot on u_x giving the cases

$$
\begin{array}{l}
u_x \neq 0, \quad f_{xx} + {f_x}^2 f = 0, \quad {f_x}^5 - f^3 f_x + f^4 = 0 \\
u_x = 0,
\end{array}
$$

as u_x occurs as a factor in two equations.

The general structure of the case splitting strategy can be described as follows:

1. If *faclimit* has been specified, check for any leading derivative factor pivots. If these occur, pick the one with the lowest ranked leading derivative.

2. Check for any leading linear coefficient pivots. If these occur, select based on the current criteria (see below).

3. Check for any initial pivots (coefficients of the highest degree of the nonlinearly occurring leading derivative of each equation). If these occur, pick the one with the lowest ranked leading derivative.

4. If *factoring* has been specified, check for any factorization of the leading nonlinear equations. If these occur, split on the smallest of these factors based on the *Maple* length.

There are a number of options for specification of the criteria described in step 2 above. These include *smalleq*, *smallpiv*, *lowrank*, *mindim*, *[smalleq,vars]*, and *[lowrank,vars]*, and they are described in detail in §A.8. The default is *smalleq*, which tells *rifsimp* to choose the pivot from the equation that is smallest in length in any variables.

3.6 Case Restriction and the MaxDim Algorithm

For some applications (most notably Lie Symmetry classification problems as described in §5.2), it is helpful, and more efficient, to restrict the computation only to cases of interest. *rifsimp* has the capability of restricting computation to only cases having at least a specified initial data dimension for the linear part of the system for specified variables (see *mindim* in §A.8). This option is capable of handling multiple criteria, specification of different types of infinite dimensional data, etc., but the most common use is as a specification for the finite dimension in the specified dependent variables (called the *mindim* variables).

Example 20 *Consider the simple* PDE *system for $\eta(x,u)$ given by*

$$\eta_{xxxu} = \eta_{xuu}, \quad \eta_{xxu} = \eta.$$

The rif'-form of this system is

$$\eta_{xx} = \eta_u, \quad \eta_{uu} = \eta.$$

From this, the initial data can be computed as

$$\eta(x_0,u_0) = c_1, \quad \eta_x(x_0,u_0) = c_2,$$
$$\eta_u(x_0,u_0) = c_3, \quad \eta_{xu}(x_0,u_0) = c_4,$$

so the initial data is 4 dimensional.

In the above example, the initial data was computed on the completed *rif'*-form of the system, but it is possible to obtain an upper bound for the dimension of the initial data for a partially completed computation by considering the initial data of the current solved-form equations. This directly results from the fact that addition of new equations to the solved-form list can only decrease the number of free parameters.

Checking the initial data upper bound within the algorithm allows *rifsimp* to determine cases that should be ignored more rapidly than computing all cases, then rejecting any cases where the initial data is too small. This can be quite significant, as often the cases having fewer free parameters are harder to compute as they are more complex than the higher dimensional cases.

Example 21 *As a simple example, consider the Lie point symmetry classification problem for the* ODE $y'' = f(y')$ *for* $y = y(x)$*. The Lie point symmetry determining system for this problem is a single equation which is linear in the infinitesimals* $\xi(x,y)$, $\eta(x,y)$ *with coefficients that are polynomials of* y', $f(y')$ *(where* y' *is now to be viewed as an independent variable).*

$$\begin{aligned}&\eta_{xx} + 2y'\eta_{xy} + {y'}^2\eta_{yy} - y'\xi_{xx} - 2{y'}^2\xi_{xy} - {y'}^3\xi_{yy}\\ &-f_{y'}\eta_x + (f - y'f_{y'})\eta_y + (y'f_{y'} - 2f)\xi_x + ({y'}^2 f_{y'} - 3y'f)\xi_y = 0\end{aligned}$$

If we request that rifsimp find all cases where the initial data is 8 dimensional or higher in ξ, η*, we obtain a single case with the restriction that* $f_{y'y'y'y'} = 0$*. If we first find all cases for this problem, then select the case with the largest initial data, we still get the 8 dimensional case, but the run also computes the smaller dimensional cases requiring more than 25 times as long to complete.*

The *MaxDim* algorithm [53] (the *maxdimsystems* implementation is described in §A.10), is an extension of this idea to an algorithm that, given the system and the dependent variables of interest, uses *rifsimp* to compute all cases of maximal finite dimension, or all infinite dimensional cases for problems in which these exist.

A verbocode description of *MaxDim* is given as follows:

1 Set the maximal dimension d to ∞.

2 Call *rifsimp* which traverses the binary tree, stopping on any branch k when the upper bound for the initial data d_k drops below d. After a finite number of steps, either some branches complete, and output *rif'*-forms will have been found, or none will have been found.

3 If branches have completed, the algorithm terminates, returning the *rif'*-forms with d free parameters in their formal series solutions, and a list of all failed branches with the values $d_k < d$, representing the upper bounds on the dimension of initial data for the branch.

4 If no branches have completed, the maximal dimension d is set to the maximal value found on the rejected branches, and the algorithm restarts at step 2.

It should be possible to use any differential elimination algorithm in place of *rifsimp* above, as long as the algorithm has been modified to compute the initial data incrementally, and reject cases where it is below the desired bound.

3.7 Additional Algorithms and Tools

The *Rif* package in *Maple* also has implementations of algorithms to compute the initial data and Taylor series of an ODE or PDE system that is in *rif'*-form, and for visualization of the structure of a multiple case *rifsimp* result.

The *initialdata* command (§1.6, §A.11) is the implementation of an algorithm that can efficiently compute initial data of PDE systems with a large number of dependent variables. The concepts used in this algorithm are clearly described in Reid [50], though the algorithm differs from the one described there.

The *rtaylor* command (§A.12) can compute Taylor series expansions of a large system of PDE simultaneously, and is described in Reid [50].

The *caseplot* command (§A.13) is a stunning visual tool that can be used to display the underlying structure of a computation with many cases in *rif'*-form. It can provide visual information on the initial data of the individual cases, and on the splittings performed in obtaining each case. Examples of the use of *caseplot* can be found in §5.3.1, §5.3.2, and §5.3.3

Chapter 4

Fast Differential Elimination in C

In this chapter we introduce the *CDiffElim* environment, written in C, and the *DiffElim* algorithm written in *CDiffElim* for differential elimination of PDE systems.

This environment has strategies for addressing difficulties encountered in differential elimination algorithms, such as exhaustion of computer memory due to intermediate expression swell, and failure to complete due to the massive number of calculations involved. These strategies include low-level memory management strategies and data representations that are tailored for efficient differential elimination algorithms. These strategies, which are coded in a low-level C implementation, seem much more difficult to implement in high-level general-purpose computer algebra systems.

The *rifsimp* implementation, described in the previous chapter, is a mature package that allows a convenient and detailed analysis of over-determined differential systems.

The primary motivation behind the design and implementation of the *CDiffElim* environment and *DiffElim* algorithm was to address several difficulties encountered when using the *rifsimp* implementation, specifically memory blow-up and speed issues. As the *rifsimp* implementation evolved, more and more complicated strategies were implemented, which resulted in significant changes to its core algorithms. This resulted in a code that performs very well for many problems, but is difficult to change and adapt for other problems.

These issues motivated the long-term project that evolved into the *CDiffElim* environment. The first step of the project was to design compact data structures needed to efficiently implement basic operations of differential elimination. Once the design and implementation of these data structures was in place, the basic algorithms were built. This was followed by the implementation of some of the more general algorithms (such as reduction, differential reduction, square-free factorization, etc.).

Once these elements were all in place, the next phase in the *CDiffElim* project was to implement the *DiffElim* algorithm using these components.

Advantages of the *CDiffElim* environment over a general-purpose computer algebra system include the following:

- As is shown in the benchmarks in Chapter 7, a careful implementation in C can offer significant reduction in the time (as much as a factor of 100) and memory usage required to simplify large and/or complex differential systems over an equivalent algorithm written in a general-purpose computer algebra system.
- The algorithms and data structures can be optimized for efficiency when working with differential elimination problems. In contrast, an efficient implementation of equations with ordered terms within *Maple* is not currently available.
- The general structure of the high-level algorithms should be kept simple and flexible, so that changes can be made easily.

Disadvantages of a low-level stand-alone implementation compared with an implementation in a general-purpose computer algebra system are by no means insignificant, and include more difficult bug fixing, and lack of access to the large number of packages and general advances available in a general-purpose computer algebra system.

4.1 The CDiffElim Implementation

The *CDiffElim* environment uses a number of well known algorithms (not restricted to symbolic algebra) to enhance the speed of symbolic computations. One of these, for example, is the *binary buddy* memory management system, which is a system that allocates large blocks of data using *malloc*, then parcels out the memory for each allocation based on a binary storage structure (see Knuth [34]). It was found that use of this memory management scheme gave a speed increase of a factor of 2 (for a Linux libc5 system) over use of the raw *malloc* and *free* functions in the C standard library, which are generally implemented using system calls.

At present, the *CDiffElim* environment is only capable of working with polynomial systems in the dependent and independent variables. In many cases, it is possible to represent functions of the independent variables such as $\sin x$, $\ln x$, and $\mathbf{e}^x$ using differential extensions, (i.e. using differential equations to define these functions as noted in §1.1), so this limitation is not as severe as it might initially seem.

In this section we present some of the core data structures being used in the *CDiffElim* implementation, and briefly discuss some associated enhancements.

4.1.1 Considerations

For most differential elimination problems, the input systems are relatively sparse in their indeterminates (with respect to their algebraic degree), so our focus will be on methods for sparse equations in many indeterminates. For this reason, equations are stored using a sparse distributed representation.

In differential elimination, the ability to store the terms of equations in the order determined by the ranking is vital to an efficient implementation. Without this ordered storage, the process of reduction requires $\mathcal{O}(k)$ term comparisons for a k term equation to determine the leading term. Application of a reduction process to the entire k term equation checking each term in the equation for a reduction, even when no reduction occurs, will then require $\mathcal{O}(k^2)$ operations.

Though ordered storage is possible in *Maple* using lists or arrays, the core algorithms (such as multiplication, division, and GCDs) are not implemented for these structures. At present, each use of these algorithms would require the conversion of the ordered representation into a *Maple* equation, and the conversion of the result back to ordered form. The conversion back would require a sort, which is of $\mathcal{O}(k\ \log(k))$ expense for a k term equation.

Development of the *CDiffElim* environment in C gives us the ability to design an ordered equation data type, and build algorithms to efficiently perform basic operations on these ordered equations.

4.1.2 Data Structures

Numerical Representation: The numerical data type (hereafter referred to as a *varint*) supports either an arbitrary precision integer or a modular integer (required for fast modular GCD and resultant computations). Arbitrary precision integers are stored in base 65536 allowing fast machine computation of integer addition, subtraction, division, multiplication, and GCD (see Knuth [35]). The Karatsuba algorithm for fast multiplication has not been implemented, but the well known binary GCD algorithm is in place.

Indeterminates: The unknowns in the equations (currently including dependent variables, independent variables, and constants), are stored in a global symbol table. This table also stores information allowing rapid computation of the derivatives of these indeterminates. All objects external to this table are stored by reference.

Terms: Monomial terms are constructed of two components, a coefficient (which is a varint), and an indeterminate list. The indeterminate list is ordered (with respect to the ranking of the derivatives), and contains the reference to each indeterminate, and its algebraic degree for that term. We note that the representation for a term is sparse with respect to all possible indeterminates, that is it only stores references to the indeterminates that are present in the term.

Equations: Equations are constructed as an ordered list of terms, in the order prescribed by the term ranking. There are two data structures used for equations, namely linked lists or *red-black trees* (see Cormen, Leiserson and Rivest [17]).

The *red-black tree* is an implementation of a partially balanced binary tree that supports $\mathcal{O}(\log(k))$ maximum, minimum, search, insert and delete functions for an k element set. We note that these are similar in concept to AVL trees (see Aho, Hopcroft, and Ullman [1]).

The most commonly used form is the linked list structure, but the binary tree representation is needed for algorithms where the ordered form of the equations cannot easily be retained (input parsing and differentiation for example).

Other Structures: In addition to these base data structures, more specialized algorithm-specific data structures are also implemented. For fast GCD computation, for example, modular univariate equations and modular matrix data structures are needed.

Depth	Number of calls	Multiplications per call (exact)	Term comparisons per call (bound)
1	1	0	lk
2	2	0	$\frac{l}{2}k$
...	...	...	...
$\log_2(l) - 1$	$\frac{l}{2}$	0	$2k$
$\log_2(l)$	l	k	0

Table 4.1: Polynomial multiplication operation counts

4.1.3 Basic Operations

Basic operations on integers are implemented based on the algorithms discussed in Knuth [35]. This includes multi-precision addition, subtraction, multiplication, division, and the binary GCD algorithm. Polynomial addition and subtraction is implemented as a merge of the two ordered input polynomials, with $\mathcal{O}(k + l)$ cost, where k,l are the number of terms in each of the polynomials.

4.1.4 Polynomial Multiplication

The ordered structure of equations allows for a recursive multivariate polynomial multiplication algorithm that maintains the ordered structure at little additional asymptotic cost. Given that we have multivariate polynomials with k and l terms, the cost of performing the multiplication is $\mathcal{O}(k\,l)$. A straightforward implementation of this multiplication, followed by a merge sort of the terms, would have an asymptotic complexity of $\mathcal{O}(kl\ \log(kl))$. Rather than using this method the smaller of the two factors is split into two equations of equal size and the routine is called recursively. Once there is only one term remaining the product with the larger equation is computed and the result returned. When the routine receives the two recursive results it then performs a single-pass merge of the computed results requiring the same number of comparisons as the total number of terms in the two equations. We order the equations so that $k \geq l$.

There are a total of kl multiplications, and an upper bound of $kl\ \log_2(l)$ term comparisons, so the worst case running time of this ordered polynomial multiplication algorithm is $\mathcal{O}(kl\ \log_2(l))$. It should also be noted that the worst case only occurs when there are very few like terms in the resulting product. In practice, for differential elimination computations, this is not often the case, and the complexity is largely dominated by the $\mathcal{O}(kl)$ multiplication cost.

4.1.5 Polynomial Division

The ordered structure for polynomials discussed above facilitates an efficient implementation of division. The resulting division algorithm is a modification of the classical polynomial division algorithm that expedites the division of polynomials having many terms by polynomials having relatively few. This modification is most relevant to division of ordered polynomial data structures.

We consider the problem: *given* $a, b \in \mathbf{Z}[x, y, ...]$ *find* q *such that* $q \in \mathbf{Z}[x, y, ...]$ *and* $a = b\,q$. We assume that a and b are stored in order with respect to a ranking $\prec$. For purposes of the following analysis we state that b has l_1 terms, q has l_2 terms, and a has $k \leq l_1 l_2$ terms (terms here refer to the number of distinct monomials). Note that we do not consider the class of problems where there is a remainder.

It is straightforward to see that classical division of a by b will require l_2 steps, each involving one term division, l_1 multiplications, and l_1 additions, so the complexity of classical division is $\mathcal{O}(l_1 l_2)$. One may also observe that when $l_1 \ll l_2$, or $l_2 \ll l_1$, the expense of performing the division is roughly $\mathcal{O}(\max(l_1, l_2))$ and that the worst case occurs in the *balanced* case when $l_1 \approx l_2$ with complexity $\mathcal{O}({l_1}^2)$.

A problem arises in the implementation of the classical polynomial division algorithm for ordered equations. Since polynomial addition is implemented as a merge of two ordered equations, division when $l_1 \ll l_2$ can have complexity $\mathcal{O}({l_2}^2)$. Note that in a private communication between Michael Monagan and Allan Steel it was observed that a version of the computer algebra language *Magma* suffered from this problem, so it is not an uncommon pitfall. As an illustration of this problem, consider the ordered division of the polynomial

$$r = xz^d + xz^{d-1} + ... + x - yz^d - yz^{d-1} - ... - y \tag{4.1}$$

by $x - y$ (we will refer to the polynomial r as the remainder). Since $(xz^d)/x = z^d$, the first term of the quotient is z^d, and in the first step of the division process $xz^d - yz^d$ is subtracted from the remainder, resulting in the cancellation of the xz^d term. In subtracting the yz^d term from the remainder, d term comparisons must be made to find the location of the yz^d term to perform the subtraction. A similar pattern repeats d times. The total number of term comparisons required for this division is $d(d+1)/2$, which is $\mathcal{O}({l_2}^2)$, the same complexity as the balanced case.

So we see that the problem occurs as the direct result of having to find the location in the remainder of a term from the product of the divisor term and the dividend. Examination of the process of division easily reveals a few facts. The quotient terms resulting from each step of the division process are generated in strictly descending order with respect to the ranking. This is only natural, as the equation is being stored in descending order with respect to the ranking, and the ranking is preserved under multiplication. Also, using a similar observation, the location of the product of a term of the divisor and the current term of the quotient will always have a location in the remainder that follows the product of that term of the divisor and the prior term in the quotient.

To avoid the efficiency difficulties mentioned earlier, we take advantage of these observations to drastically reduce the number of term comparisons in the implementation of the division process in *CDiffElim*.

In the first division step, we store a location in the remainder where we begin searching for the location of each term of the quotient-divisor term product for each term of the divisor. In all following division steps, we use these stored locations as a starting point to determine where the product of the next quotient term and the corresponding term of the divisor belongs, and adjust the stored location as necessary.

As an illustration of the concepts described above, consider the state of the remainder of (4.1) after the first division step (where the stored locations are marked with the underbrace):

$$\underbrace{xz^{d-1}} + \ldots + x - \underbrace{yz^{d-1}} - \ldots - y. \tag{4.2}$$

In the next division step, the stored locations correspond exactly to the terms required when subtracting $xz^{d-1} - yz^{d-1}$ from the remainder.

An asymptotic result for the number of monomial comparisons for this modified algorithm can be obtained as follows:

The total number of comparisons needed for the first division step is at most $k < l_1 l_2$. The worst case number of comparisons needed in all following division steps is $\mathcal{O}(l_1 l_2)$ for each term of the divisor (here $l_1 l_2$ is the maximum possible length of the dividend). The reasoning for this is that the stored location for each term of the dividend is only ever moved to lower order terms in the remainder, and the maximum number of terms in the remainder is $l_1 l_2$. So we obtain the number of comparisons as $\mathcal{O}(k + {l_1}^2 l_2)$, which when $l_1 \ll l_2$ gives $\mathcal{O}(l_2)$, the same order as the number of arithmetic operations required for the division.

In conclusion, when b has few terms relative to a, we can expect we are in the $l_1 \ll l_2$ case, and use this modified division algorithm. When the number of terms of a and b are on the same order, the unmodified algorithm should be used instead.

4.1.6 Differentiation

The ordered approach used for addition, multiplication and division does not lend itself well towards differentiation, so an alternative approach is needed.

The derivative of an equation is constructed through differentiation of each term. As each term is differentiated, it is inserted into a *red-black* tree structure (§4.1.2). This has the advantage that any common terms, resulting from differentiation of different terms, will not incur more than a single term memory allocation, if implemented correctly.

For example, consider the differentiation of a quadratic polynomial having l terms, with a result having $k \leq 2l$ terms. The cost of the differentiation into ordered form requires approximately $2\,l\,\log_2(k)$ comparisons (due to the use of the partially balanced binary tree), and k memory allocations. An algorithm that were to perform all differentiations and sort afterwards, would require the same number of term comparisons if performed with a fast sorting algorithm, but would require $2l$ memory allocations instead.

4.1.7 Multivariate polynomial GCDs

The basis for all multivariate GCD code in *CDiffElim* is a recent modification of Brown's dense modular GCD algorithm by Monagan and Wittkopf [43] (see §6.4), and a new modification of Zippel's sparse modular GCD algorithm (see §6.6). Both algorithms include an inexpensive probabilistic method for obtaining degree bounds for each variable in the GCD, and as a result allow, with high probability, rapid determination of a trivial GCD, which is the most common case in many of the operations of differential elimination.

In the process of differential elimination, GCD computations are required frequently. Examples include removal of factors depending upon only the independent variables of the problem, and detection of factor-based case splitting.

These algorithms, and a highly detailed discussion and analysis of modular methods for GCD computation, can be found in Chapter 6.

4.2 The DiffElim Algorithm

As a test of the *CDiffElim* environment, the *DiffElim* algorithm has been implemented in this environment. A basic outline of the algorithm resembles a modification of the Buchberger algorithm [10].

Consider systems that are linear in the dependent variables and their derivatives, with coefficients that are polynomial functions of the independent variables. For such systems, the *DiffElim* algorithm terminates in a finite number of steps, and the output is in *rif'*-form, yielding an existence and uniqueness theorem for the input (see Rust *et al.* [60, 58]). Although *DiffElim* is capable of taking inputs that are nonlinear in the dependent variables and their derivatives, the fact that it is a pure generalization of Gröbner Bases to differential polynomial rings means that there is no guarantee of termination.

Given the input system, one equation is selected at a time, and reduced with respect to the already simplified equations. The input ranking is simply a differential ranking (§1.2). The D-polynomials (hereafter referred to as *DPolys*), in the linear context, are the integrability conditions (§1.4) using the redundancy criteria of Rust (provided as Theorem 2 of Chapter 2). For nonlinear systems, they represent an extended form of integrability condition combined with algebraic S-polynomials. We omit the details here, as the condition set for the nonlinear case is not necessarily finite.

Rather than forming the DPolys, references are stored, and the DPolys are computed when needed (making it possible to store the infinite sets of conditions for the nonlinear case via a description). Once all DPolys are either appended to the system (possibly resulting in new DPolys) or reduced to zero, the algorithm terminates, and returns the reduced system with all integrability conditions satisfied.

Algorithm 14 (DiffElim)

Input: *An input system of* PDE U, *a ranking* $\prec_d$, *and a set of criteria* C *(§4.2.1).*

Output: *The reduced form of the input.*

$G := \{\}$, $S := \{\}$
while $U \neq \{\}$ *or* $S \neq \{\}$ *do*
  $neweq := SelectFrom(U,\ S,\ \prec_d,\ C_{select})$
  if $neweq$ *is a DPoly then*
    $S := S \setminus \{neweq\}$
    $neweq := MakeDPoly(neweq,\ G)$
  else
    $U := U \setminus \{neweq\}$
  end if
  $neweq := Reduce(neweq,\ G,\ \prec_d,\ C_{reduce})$
  if $CheckCriteria(neweq,\ C_{accept})$ *is false then*
    $U := U \cup neweq$
  else if $neweq$ *is not identically zero then*
    $G := Reduce(G,\ neweq,\ \prec_d,\ C_{reduce})$
    $G_r := HasReduced(G)$
    if elements of G *have reduced (i.e.* $G_r \neq \{\}$*) then*
      $G := G \setminus G_r$
      $U := U \cup G_r$
      $S := S \setminus GetRefs(G_r,\ S)$
    end if
    $S := S \cup BuildDPolyRefs(neweq,\ G,\ \prec_d,\ C_{compat})$
    $G := G \cup neweq$
  end if
end while
return G

We now present a brief description of some of the components of the algorithm.

SelectFrom: Determines the next best equation to use for the system. The determination of the equation or DPoly to be used is based on the C_{select} criteria. The selection criteria has the capability of choosing to compute a DPoly instead of selecting an equation from U

if none of the equations in U satisfy the criteria.

MakeDPoly: Constructs DPolys, which for linear systems are the integrability conditions (§1.4).

Reduce: Reduces the equations given in the first input by the equations in the second input. The type of reduction to perform is based on the C_{reduce} criteria. For linear systems the reduction is that defined in (§1.3).

CheckCriteria: Validates that the equation still satisfies the selection criteria after being reduced. This is necessary as the process of reduction can cause the complexity of the input equation to increase significantly. For example the equation can become excessively large, or become more dense in the indeterminates of lower rank. The decisions are made based on the C_{accept} criteria.

HasReduced: Determines which of the input equations have been changed by the last reduction. This allows the implementation to be *update* driven.

GetRefs: Obtains all DPoly references from the second input that depend on the equations in the first input. Again required for the update driven implementation.

BuildDPolyRefs: Constructs reference form for all DPolys between the input equation, and the current solved form system G.

4.2.1 Adjustable Criteria

There are various criteria that can be specified for a run of the algorithm. These criteria, in combination with the input ranking, offer some flexibility as to the manner in which the system is reduced. The currently available criteria are:

$\mathbf{C_{select}}$**:** The selection criteria is used to determine a good candidate equation or DPoly to adjoin to the simplified system. Currently this criteria can be based on 4 measures:

1. The overall size of the equation;
2. The differential order of the equation;
3. The algebraic degree of the equation;
4. The size of the coefficient of the leading derivative in the equation.

A hybrid of the above criteria may also be specified. For DPolys, the quantities are estimated from the pair of equations that are used to create the DPoly.

$\mathbf{C_{accept}}$: As mentioned in the discussion of *CheckCriteria*, the complexity of an equation can increase greatly in the process of reduction. This criteria can be set to delay the use of equations of this type until further equations are uncovered (that may reduce this equation to a simpler form). This may be used to limit the growth of equations during reduction to a certain percentage of the original size. Alternatively it can be used as a screen, with a very loose setting for C_{select}, to force the algorithm to consider the reduced form of the DPolys.

$\mathbf{C_{reduce}}$: This criteria controls whether pure reduction, pseudo-reduction (§1.9), or a combination of the two is being used. By a combination of the two, we refer to pseudo-reduction over a restricted set of variables. In this case of limited pseudo-reduction (the type of reduction used for the *rifsimp* implementation), any leading nonlinear equations are treated as purely algebraic constraints.

$\mathbf{C_{compat}}$: This criteria controls how DPolys are generated, and is closely related to C_{reduce}. If pseudo-reduction is in use, this generates the spawn (§1.7) of any leading nonlinear equations (by differentiating each equation with respect to each independent variable) and normal integrability conditions for any leading linear equations. The spawn method is needed to enforce any differential consequences of leading nonlinear equations. If pure reduction is in use, the extended form of the DPolys is used. If the reduction form is mixed, then a combination of the methods is used.

For all systems that are treated in the applications in Chapter 5 and the benchmarks in Chapter 7, the algorithm was used with the settings that give it the same behavior as the *rifsimp* implementation uses by default.

It should be noted that although the algorithm is not necessarily finite for nonlinear systems, it will terminate with a result if it is possible to reduce the input system to one that is leading linear.

The author *has* found great utility in the implementation for nonlinear problems, even in cases where the algorithm does not terminate, as there is an option to store the intermediate system in *Maple* format after some fixed number of iterations, resulting in a sequence of files for the progress of the algorithm on the system. Often it is possible to examine these intermediate results in *Maple*, and determine a useful manual splitting of the system that will allow *DiffElim* to terminate in a finite number of iterations.

Chapter 5

Applications

5.1 Lie Symmetries

The symmetries, or transformations leaving invariant a system of partial differential equations, are not known a priori, and have to be determined.

Applications of symmetries include mapping known solutions to new solutions, reducing the order of ODE and the number of variables in PDE, construction of similarity solutions or solutions invariant under a symmetry, and mapping of a given PDE to a more tractable PDE (e.g. transformation of a nonlinear PDE to a linear PDE). See Bluman [3] or Olver [46] for more detail.

One method for determination of the symmetries of a PDE system in independent variables $x = (x_1, x_2, ..., x_n)$ and dependent variables $u = (u_1(x), u_2(x)..., u_m(x))$ is to substitute transformations of the form

$$(x, u) \mapsto (\hat{x}, \hat{u}) = (X(x, u), U(x, u))$$

into the PDE system and equate to the non-transformed PDE. This results in a complex and highly nonlinear over-determined system of PDE for the unknown functions X, U.

Lie's breakthrough was to simplify the problem of finding continuous symmetries by linearizing these transformations about the identity. These linearized transformations, called *infinitesimal Lie symmetry transformations*, satisfy an associated **linear** system of over-determined partial differential equations.

Following this approach, one would seek linearized or infinitesimal symmetries of the form

$$(x, u) \mapsto (\hat{x}, \hat{u}) = (x + \xi\epsilon + \mathcal{O}(\epsilon^2), u + \eta\epsilon + \mathcal{O}(\epsilon^2)), \tag{5.1}$$

where ξ, η, called infinitesimals, are unknown functions of x, u. The symmetry vector field, $\xi \cdot \nabla_x + \eta \cdot \nabla_u$, is tangent to the flow induced by the group transformation of the symmetry.

A simple but inefficient way to obtain the symmetry determining system for ξ and η is to substitute the linearized form of the transformations above into the PDE system. The determining system is then obtained from the coefficient of the ϵ term. Lie [46] gave a much more elegant and efficient method in which the determining equations result from applying the prolongation of the linearization operator to the PDE.

There are many computer algebra programs for automatically generating such infinitesimal symmetry determining systems (see the extensive review article by Hereman [24], and also see Ibragimov [30]). We obtained the determining systems for this thesis using Hickman's package [25].

Differential elimination can be used to greatly simplify these determining systems.

5.2 Lie Symmetry Classification Problems

The analysis of Lie symmetries for systems of ODE or PDE can be extended in a straightforward manner to cover a broad class of ODE or PDE systems within a single computation. This is accomplished through the introduction of arbitrary functions into the system.

Example 22 *Consider the following 3 separate* ODE*,*

$$\begin{aligned} y''(x) &= g(x)y(x), \\ y''(x) &= xy(x)^2, \\ y''(x) &= \frac{y(x)^2 - x\mathrm{e}^{y(x)}}{x^3 y(x) + \cos(x)}. \end{aligned}$$

Lie symmetry information for all three ODE *could be obtained from the analysis of the single* ODE

$$y''(x) - f(x, y(x)) = 0,$$

where $f(x, y(x))$ is considered to be an arbitrary function.

Another example of a situation in which this type of system can arise is in a physical model, where prescribed functions (such as a wave function or forcing function) are part of the model.

Different forms of these *classification functions* can result in distinctly different symmetry properties for the ODE or PDE system. A *Lie symmetry classification problem* then arises, where it is desirable to obtain information on the symmetries for some (or perhaps all) possible forms of the classification functions.

In performing a case splitting differential elimination analysis on these systems (having chosen an appropriate ranking), the determining equations split naturally into a finite number of cases for the different forms of the classification functions, allowing one to characterize the symmetries of different subclasses of the system.

Though it may not be possible to solve for the infinitesimal Lie symmetries for each subclass, information can still be obtained, such an existence-uniqueness theorem for solutions of the subclass, and the number and structure of the symmetry group for the subclass.

Naturally, conditions are obtained to allow detection if a specific form of the classification functions belongs to a class.

Example 23 *Consider the* ODE *for $y(x)$*

$$y''(x) = ay(x)(y'(x) + by(x)) \tag{5.2}$$

for arbitrary a, b. The determining system for the infinitesimal Lie symmetries of (5.2) given by ξ, η from $(x, y) \mapsto (x + \xi\epsilon + \mathcal{O}(\epsilon^2), y + \eta\epsilon + \mathcal{O}(\epsilon^2))$ is

$$\begin{aligned}
\eta_{xx} &= 0, \\
-2ay\eta_y + \xi_{yy} - 2\eta_{xy} &= 0, \\
-a\xi - 3aby^2\eta_y - ay\eta_x + 2\xi_{xy} - \eta_{xx} &= 0, \\
-2aby\xi - 2aby^2\eta_x + aby^2\xi_y - ay\xi_x + \xi_{xx} &= 0
\end{aligned}$$

A case splitting differential elimination algorithm rapidly gives three cases depending upon a,b, all of which can be solved for their infinitesimals giving

$$(i) \quad a \neq 0, b \neq 0 \qquad \xi = c_1,\ \eta = 0,$$

$$\begin{array}{lll} (ii) & a \neq 0, b = 0 & \xi = c_1 - c_2 x,\ \eta = c_2 y, \\ (iii) & a = 0, b = 0 & \xi = c_1 + c_2 x + c_3 y + c_4 x^2 + c_5 xy \\ & & \eta = c_6 + c_7 x + c_8 y + c_4 xy + c_5 y^2, \end{array}$$

where $c_1, .., c_8$ are arbitrary constants for each case. The number of independent arbitrary constants gives the dimension of the symmetry group for that case.

One may only be interested in classes where certain conditions are satisfied. One quite useful condition is restriction to classes having *at least* a specified dimension for the initial data (see §1.6). This allows one to find the class or classes having the maximal symmetry group (the classes with the highest dimensional initial data). This approach is used heavily in the following applications.

5.3 Maximal Group of nonlinear Reaction Diffusion Equations

In [53] we considered coupled nonlinear reaction diffusion systems (see Murray [45]) of the form

$$u_t - u_{xx} = f(u), \tag{5.3}$$

where u is an m-component vector. The components of u are, for example, concentrations of interacting chemicals in combustion or competing species in biological applications [45]. The group theoretic study of such systems was initiated in Zulehner and Ames [81] (also see Vol. I, §10.13 of Ibragimov [30]).

Following Lie's linear approach (§5.1), we considered the transformation

$$(t, x, u) \mapsto (\hat{t}, \hat{x}, \hat{u}) = (t, x, u) + (\tau, \xi, \eta)\, \epsilon + \mathcal{O}(\epsilon^2),$$

where $\tau\partial_t + \xi\partial_x + \eta \cdot \nabla_u$ is the infinitesimal generator of the symmetry group.

5.3.1 Single Component Equation

We consider the nonlinear reaction diffusion system (5.3) for a single component

$$u_t - u_{xx} = f(u), \tag{5.4}$$

where the nonlinearity is imposed by including the additional inequation $f_{uu} \neq 0$ in the system. Applying Lie's linear approach to (5.4) we obtain the system of 7 equations

$$\begin{aligned}
2\eta_{xu} - \xi_{xx} + \xi_t + 3f\xi_u &= 0, \\
\eta_t - \eta_{xx} + f\eta_u - f_u\eta - 2f\xi_x &= 0, \\
\tau_t - \tau_{xx} + f\tau_u - 2\xi_x &= 0, \\
\eta_{uu} - 2\xi_{xu} = 0, \tau_{xu} + \xi_u &= 0, \\
\xi_{uu} = 0, \tau_u = 0, \tau_x &= 0,
\end{aligned} \tag{5.5}$$

for the infinitesimal symmetries from $\tau(t,x,u)\partial_t + \xi(t,x,u)\partial_x + \eta(t,x,u)\partial_u$.

Running with *rifsimp* with the initial data requirement set to ∞ (see §A.6 and §A.8),

```
rifsimp(sys, [tau,xi,eta], casesplit, mindim=infinity);
```

in which `sys` is a list containing equations (5.5) and the nonlinearity requirement $f_{uu} \neq 0$ The calculation completes with two cases, both of which violate the initial data constraint, the first case having initial data with dimension at most 4, the second at most 3.

Re-running *rifsimp* with `mindim=4` gives a result with the case tree (see caseplot in §A.13) in Figure 5.1.

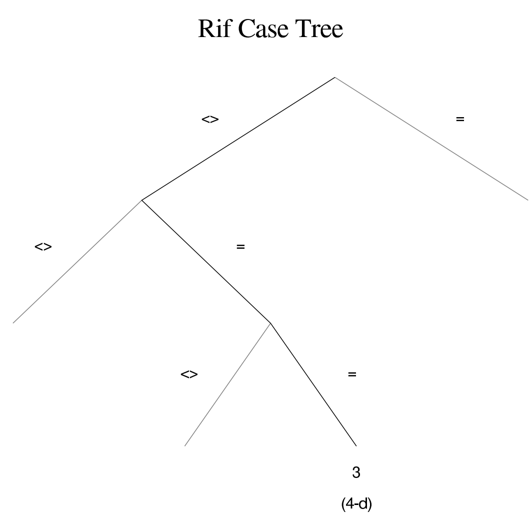

Figure 5.1: Single component Reaction-Diffusion

So we find exactly one four-dimensional case. For this case there are two conditions on

$f(u)$,

$$f_u f_{uu} f_{uuu} + f_{uu}^3 + f f_{uuu}^2 = 0, \quad f_{uuuu} = 2\frac{f_{uuu}^2}{f_{uu}},$$

which can be readily solved to give

$$f(u) = a(u+b)\ln(c(u+b))$$

where a, b, c are arbitrary constants. This is a well-known result given in §10.2, Vol. I of [30], that the one component diffusion equation with maximal (4 dimensional) group is $u_t - u_{xx} = a(u+b_1)\ln(c_1(u+b_1))$.

The exact solution for the infinitesimals can also be obtained, giving

$$\begin{aligned}
\eta(t,x,u) &= (C_1 + C_2 x)(u+b)\mathrm{e}^{-at}, \\
\xi(t,x,u) &= C_3 - 2C_2\frac{\mathrm{e}^{-at}}{a}, \\
\tau(t,x,u) &= C_4
\end{aligned}$$

where C_1, C_2, C_3, C_4 are the arbitrary constants that correspond to our 4-dimensional initial data.

5.3.2 Two Component Uncoupled System

In [53] we considered the two component reaction-diffusion system where $u = (u, v)$ which is given by

$$u_t - u_{xx} = f(u), \qquad v_t - v_{xx} = g(v)$$

where the nonlinearity is imposed by the additional inequation $f_{uu} \neq 0$.

Running *rifsimp* on this system, together with the inequation, resulted in 8 infinite dimensional cases as shown in Figure 5.2. Analysis of each of the infinite branches revealed that the equation $g_{vv} = 0$ is common to all. These are all cases in which the second equation becomes linear with respect to v, and admits an easily interpreted infinite dimensional group, which was not pursued further.

When we re-ran the calculation adding the assumption $g_{vv} \neq 0$, we found that there are no infinite cases, obtaining an upper bound of 5 for the dimension of the initial data.

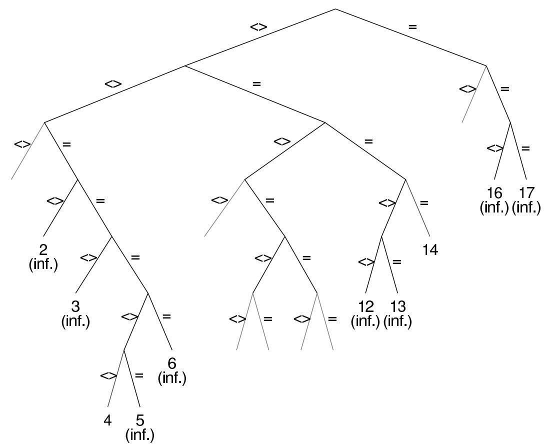

Figure 5.2: Uncoupled Reaction-Diffusion (infinite)

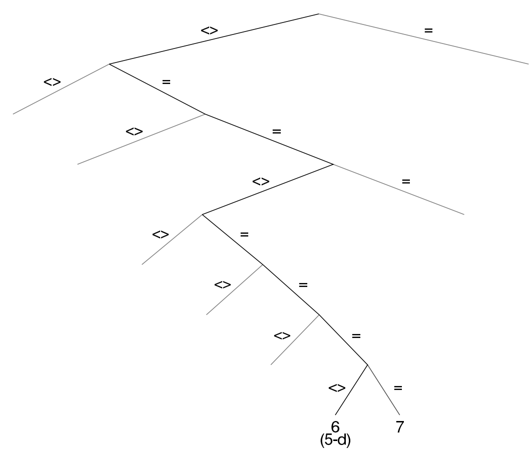

Figure 5.3: Uncoupled Reaction-Diffusion (finite)

Running *rifsimp* with `mindim=5` gave us exactly one case, as shown by the `caseplot` in Figure 5.3. We also obtained a system of 3 PDE, and 4 inequations for $f(u), g(v)$:

$$f_{uuu} = \frac{-f_{uu}g_{vv}g - f_{uu}f_u^2 + f_{uu}^2 f + f_u g_v f_{uu}}{f f_u - f g_v},$$

$$g_{vvv} = \frac{g_{vv}^2 g_v - f_u g_{vv}^2}{f f_{uu} - g_{vv} g},$$

$$-2 f f_{uu} g_{vv} g + g^2 g_{vv}^2 - f_u g_v f f_{uu} - g_{vv} f_u g_v g$$

$$+ g_v^2 f f_{uu} + f^2 f_{uu}^2 + f_u^2 g_{vv} g = 0,$$

$$f_{uu} \neq 0, g_{vv} \neq 0, f f_{uu} - g_{vv} g \neq 0.$$

$$f_u^2 - f_u g_v - f f_{uu} + g_{vv} g \neq 0,$$

which was solved using *Maple*'s *dsolve* with assistance from *rifsimp* to obtain

$$\begin{aligned} f(u) &= a(u+b_1)\ln(c_1(u+b_1)), \\ g(v) &= a(v+b_2)\ln(c_2(v+b_2)). \end{aligned}$$

This is an predictable extension of the result from §5.3.1 and §10.2, Vol. I of Ibragimov [30].

5.3.3 Two Component Partially Coupled System

In [53] we considered the two component reaction-diffusion system given by

$$u_t - u_{xx} = f(u), \qquad v_t - v_{xx} = g(u,v), \tag{5.6}$$

where the nonlinearity is imposed by the inequation $f_{uu} \neq 0$, and the coupling by the inequation $g_u \neq 0$.

Rather than using *rifsimp* and manually adjusting the minimum dimension, we used the *MaxDim* algorithm (see §3.6, §A.10) to compute the cases with largest initial data. We obtained a total of 67 cases for this problem, of which 41 had infinite initial data, and 18 were rejected, as the initial data is finite. Investigation revealed that *all* infinite cases have two branching equations in common, specifically $g_{vv} = 0$ and $g_{uv} = 0$.

This means that $g(u,v)$ is of the form $g(u,v) = g_1 v + g_2(u)$, and with this restriction, the second equation becomes linear in v. Once the nonlinear equation for u alone is solved, we have an inhomogeneous linear equation in v to solve. This is a predictable class with infinite groups, which we did not pursue further.

To find pure nonlinear cases, we then examined the three separate cases given by

Case (I): $g_{uv} \neq 0, g_{vv} = 0$;

Case (II): $g_{uv} = 0, g_{vv} \neq 0$;

Case (III): $g_{uv} \neq 0, g_{vv} \neq 0$.

Application of *MaxDim* to Case (I) showed that there are three solutions having maximal 6 dimensional initial data (see Figure 5.4). The first case (referenced by 16 in Figure 5.4)

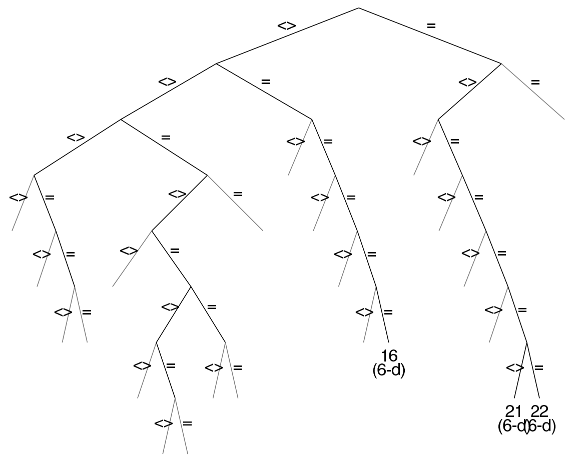

Figure 5.4: Partially Coupled Reaction-Diffusion

gave a straightforward set of differential equations for f, g, which can be readily solved to give

$$
\begin{aligned}
f(u) &= a(u+b)^2, \\
g(u,v) &= auv + d_1 u + d_2 v + c,
\end{aligned} \tag{5.7}
$$

where a, b, c, d_1, and d_2 are arbitrary parameters.

The second case (21 in Figure 5.4) gave a somewhat more complex system of differential equations, involving four PDE, and a constraint

$$
\begin{aligned}
g_{uuu} = \frac{2g_v g_{uu}^2 - 2f_u g_{uu}^2 - 2g_u g_{uv} g_{uu}}{f g_{uu} + g_{uv} g}, \\
g_{uuv} = \frac{g_{uv} g_{uu} g_v - g_{uv}^2 g_u - g_{uv} g_{uu} f_u}{f g_{uu} + g_{uv} g}, \\
g_{vv} = 0, f_{uu} = g_{uv}, \\
-2g_v g_u g_{uv} f g_{uu} + g_v^2 g_{uu}^2 f + f^2 g_{uv} g_{uu}^2 + 2f g_{uv}^2 g_{uu} g \\
-f_u g_{uu}^2 g_v f + f_u g_{uu} g_v g_{uv} g + g_{uv}^3 g^2 + g_u^2 g_{uv}^2 f \\
-g_u g_{uv}^2 f_u g + g_{uv} g_{uu} f_u f g_u - g_{uv} g_{uu} f_u^2 g = 0.
\end{aligned} \tag{5.8}
$$

Since $f(u), g(u,v)$ and all their derivatives are ranked lower than the infinitesimals, the *rifsimp* implementation gives the f, g subsystem in *rif'*-form. This also means that no new conditions arise through differentiation of the constraint (see Rust *et al.* [60]). These can also be explicitly solved to give

$$\begin{aligned} f(u) &= a_1(u+b_1)\ln(c_1(u+b_1)), \\ g(u,v) &= a_1(v+b_2)\ln(c_2(u+b_1)) + a_2(u+b_1), \end{aligned}$$

where a_1, a_2, b_1, b_2, c_1, and c_2 are arbitrary parameters.

The third case can be shown to be a sub-case of the second, so there are two distinct forms for $f(u), g(u,v)$ that give 6 dimensional initial data.

Exploration of Case (II) revealed that the initial data was at most 4 dimensional. Case (III), the most computationally challenging of the three, required the use of *DiffElim* (Chapter 4) to complete, and was shown to be at most 5 dimensional. So we have found the maximal finite cases.

5.3.4 Extension to Greater Number of Components

In [53] we explored the extension of one of the maximal cases (5.7) for the partially coupled system (5.6) to a system involving a greater number of components.

Extensive calculations in the three component case, made possible by the *DiffElim* implementation, and based on the form of the two component system, led us to conjecture the following form for $f = (f_1, ..., f_n)$:

$$\begin{aligned} f_1(u_1) &= a(u_1+b)^2, \quad a \neq 0, \\ f_i(u_1, ..., u_n) &= a u_1 u_i + \sum_{j=1}^{n} d_{i,j} u_j + c_i, \quad i = 2, ..., n \end{aligned}$$

For $n = 1, 2, ..., 7$ it was found that the dimension of the Lie group of symmetries for the n-component system is $n^2 + 2$. These calculations, which resulted in a single case, are used as a test problem in §7.1 to compare the efficiency of the *rifsimp* implementation and the *DiffElim* implementation.

5.4 Determination of Symmetries of Nonlinear Schrödinger Systems

In this section we discuss the application of differential elimination to the infinitesimal symmetry defining systems for vector nonlinear Schrödinger systems of the form

$$iu_t + \nabla^2 u + F\left(x, t, u \cdot u^*\right) u = 0, \tag{5.9}$$

for which both the spatial dimension n, and the number of components in the vector potential m can be increased arbitrarily.

It was found that these systems were sufficiently challenging to motivate improvements for both *rifsimp* and *DiffElim*.

We briefly describe some peculiarities in the formation of the infinitesimal symmetry defining systems for complex problems, then proceed to obtain the form of the scalar nonlinear Schrödinger equation having a maximal dimensional symmetry group, and finally generalize this form to vector nonlinear Schrödinger systems.

5.4.1 Generating Symmetry Defining Systems for Nonlinear Schrödinger Systems

Following Lie's linearized approach we seek infinitesimal symmetries of the form

$$(t, x, u) \mapsto (\hat{t}, \hat{x}, \hat{u}) = (t, x, u) + (\tau, \xi, \eta)\epsilon + \mathcal{O}(\epsilon^2),$$

where the infinitesimals of the symmetries, τ, ξ, η are unknown functions of t, x, u.

Lie Theory is an analytic theory, in which all quantities are regarded as complex, including dependent and independent variables. Thus the VNLS system cannot be treated directly (see Mansfield, Clarkson, and Reid [40]) since it contains the non-analytic function $u \cdot u^* = |u|^2$. To circumvent this difficulty u is expressed in terms of its real and imaginary parts, embedding the given system in an analytic system. The determining equations can then be obtained as described in §5.1.

Alternatively, one may use the formal complex conjugate of (5.9):

$$-iu_t^* + \nabla^2 u^* + F\left(x, t, u \cdot u^*\right) u^* = 0.$$

Here we have assumed that x and t are real, and F is a real analytic function, so that $F^* = F$, and u^* is treated as a new dependent variable. The determining equations for the original and conjugate PDE can also be obtained as described in §5.1.

Both methods are equivalent through a simple change of coordinates.

5.4.2 Maximal symmetry group of $iu_t + u_{xx} + F(x, t, uu^*)u = 0$

In [74] we determined the form of the scalar ($m = 1$) nonlinear Schrödinger equation in a single spatial dimension ($n = 1$) having the maximal dimension Lie symmetry group. The condition that F is a function of the product uu^* (but not u and u^* independently) was imposed by adjoining the differential equation

$$u^* F_u - u F_{u^*} = 0 \tag{5.10}$$

to the determining system which was generated for an arbitrary form of $F(t, x, u, u^*)$.

The *MaxDim* algorithm (§3.6, §A.10) was applied to the determining system to find the maximal dimension symmetry group.

As a result of the calculation, and an additional calculation with the minimum dimension specified to be 5, we obtained the following theorem.

Theorem 12 *When $m = n = 1$, (5.9) admits a maximal 6 dimensional symmetry group if and only if it has the form*

$$iu_t + u_{xx} + \left(a(t)x^2 + b(t)x + c(t) + d|u|^4\right) u = 0 \tag{5.11}$$

where $a(t), b(t), c(t)$ are arbitrary functions of t, and d is an arbitrary nonzero constant. When $m = n = 1$ the form

$$iu_t + u_{xx} + \left(b(t)x + c(t) + d|u|^{2\sigma}\right) u = 0 \tag{5.12}$$

admits a 5 dimensional symmetry group where $b(t), c(t)$ are arbitrary functions of t, d is an arbitrary nonzero constant, and $\sigma \neq 0, 2$.

We note that there are other forms of F for the case $m = n = 1$ with 5 dimensional symmetry groups that are not discussed here.

This result is a generalization of the well-known result (see Bluman and Kumei [3]) for linear Schrödinger equations of form $iu_t + u_{xx} + F(x,t)u = 0$. In particular the result is that disregarding the infinite superposition group, such an equation has a maximal symmetry group if and only if $F(x,t) = a(t)x^2 + b(t)x + c(t)$. We also direct the reader to a classification of $F = f(x,t) + g(x,t)|u|^2$ given by Gagnon and Winternitz [30].

5.4.3 Generalization to VNLS systems

In our work in [73] we were motivated to consider higher dimensional generalizations of the results in Theorem 12. We investigated the case where $F = |u|^2$ in (5.9), and we empirically obtained the formula for the dimension of the associated group $\mathcal{G}$ as

$$\dim(\mathcal{G}) = 2 + m^2 + n(n+3)/2 + \delta(n-2) \tag{5.13}$$

where $\delta(n-2)$ is the Kronecker-Delta function ($\delta(0) = 1$, $\delta(x) = 0$ for $x \neq 0$).

In [74] we considered systems of the form:

$$iu_t + \nabla^2 u + \left(a(t)|x|^2 + b(t)\cdot x + c(t) + d|u|^{2\sigma}\right)u = 0.$$

Calculations for low values of m, n, made possible through the use of *DiffElim*, led us to the class where $\sigma = \frac{2}{n}$, or

$$iu_t + \nabla^2 u + \left(a(t)|x|^2 + b(t)\cdot x + c(t) + d|u|^{\frac{4}{n}}\right)u = 0, \tag{5.14}$$

and to the table of the group dimension results below:

Spatial Dimension (n)	Number of components (m) 1	2	3	4
1	6	9	14	21
2	9	12	17	24
3	13	16	21	
4	18	21	26	
5	24	27		
6	31	34		
7	39			
8	48			

Table 5.1: Group dimension results for (5.14)

We were able to fit the results in Table 1 to a formula, which agreed with result (5.13) for the overlapping cases with $n = 2$ and $a(t) = b(t) = c(t) = 0$. This led to the following conjecture:

Conjecture 3 *The dimension of the Lie group of symmetries of (5.14) is:*

$$dim(\mathcal{G}) = m^2 + n(n+3)/2 + 3. \tag{5.15}$$

Moreover, this is the maximal group of symmetries of (5.9).

A large portion of the 1999 book *The Nonlinear Schrödinger Equation* (Sulem [66]) is devoted to the nonlinear Schrödinger equation with power law nonlinearities of the form

$$iu_t + \nabla^2 u + |u|^{2\sigma} u = 0. \tag{5.16}$$

The existence and behavior of solutions, particularly blowup or wave collapse solutions, depends strongly on the exponent σ.

In the supercritical case, $\sigma > \frac{2}{n}$ with rapidly decaying initial conditions, functional analytic methods show that blowup occurs in finite time (e.g. see Theorem 5.2, page 95, of [66] and many variations on this result). Additionally the blowup is self-similar. A similar result, requiring stronger assumptions on the initial conditions, guaranteeing blowup, also holds in the critical case $\sigma = \frac{2}{n}$. Some global existence results are known for the sub-critical case $\sigma < \frac{2}{n}$.

It is interesting that the critical case, $\sigma = \frac{2}{n}$, is picked out in our results as the maximal dimension symmetry case for $n = m = 1$. This extra symmetry was noted by Taranov [67] (also see [66], pp. 33 – 37).

The class (5.14) is much broader than (5.16). It would be interesting to see if the criticality, super-criticality and sub-criticality behavior of blowup solutions is retained by our extended class (5.14).

5.5 Nonlinear Telegraph System

Consider the nonlinear telegraph system

$$v_x = u_t, \ v_t = C(u,x)u_x + B(u,x), \tag{5.17}$$

where $C_u \neq 0$ and $B_u \neq 0$. We note that we briefly stated this result in [53], but it is provided here with supporting detail. Application of *MaxDim* with $d = \infty$, yields several infinite cases. One of these cases has the classifying conditions

$$\begin{aligned}
C_{xx} &= \frac{2C_u^2 C B_{ux} + (6C_u^2 - 4CC_{uu})B_u^2 + (4CC_{uu} - 7C_u^2)C_x B_u - C_u^3 B_x + (2C_u^2 - CC_{uu})C_x^2}{C_u^2 C} \\
C_{ux} &= \frac{2CC_{uu}B_u + 3C_u^2 C_x - 3C_u^2 B_u - CC_{uu}C_x}{C_u C}, \\
B_{uu} &= \frac{2C_u^2 C_x - 2C_u^2 B_u - CC_{uu}C_x + 2CC_{uu}B_u}{C_u C}.
\end{aligned} \tag{5.18}$$

Application of *rifsimp* with an elimination ordering on B and its derivatives provides the PDE

$$C_{uux} = \frac{3C_u C_{uuu} C_x - 2CC_{ux}C_{uuu} - 4C_x C_{uu}^2 + 4C_u C_{ux} C_{uu}}{-2CC_{uu} + 3C_u^2} \tag{5.19}$$

in $C(u, x)$ alone.

It is possible to obtain the 3-infinite infinitesimal symmetry group for this PDE,

$$(u, x, C) \mapsto (u, x, C) + (f_1(x), c_1 + \frac{c_2}{\sqrt{C}} + f_2(x) + f_3(x)u, -2f_3(x)C + c_1 C)\epsilon + \mathcal{O}(\epsilon^2),$$

where c_1 and c_2 are arbitrary constants, and $f_1(x)$, $f_2(x)$ and $f_3(x)$ are arbitrary functions of x.

With a restriction of these symmetries, the invariant surface condition can be solved by the method of characteristics giving the invariants

$$r = up(x), \;\; v = \frac{C}{p(x)^2},$$

where the arbitrary function $p(x)$ is related to $f_1(x)$ and $f_3(x)$.

This then gives a solution of (5.19) as $C(u, x) = F(p(x)u)p(x)^2$ for some form of the function F. Substitution into (5.19) yields no restrictions, so we have found a solution for the PDE for general F, p.

The form of $B(u, x)$ can then be obtained from $C(u, x)$ and (5.18), and a further simplification $F(s) = sf(s)$ gives the class of nonlinear telegraph equations

$$v_x = u_t, v_t = \frac{p}{u} f\left(pu\right) u_x + p_x f\left(pu\right),$$

where both p and f are arbitrary functions of their arguments.

The type of infinite symmetry group admitted by (5.17) for this class satisfies the conditions for Bluman and Kumei's Linearization Theorem [3]. That theorem shows that

$$X = \int \frac{dx}{p(x)}, T = t, U = p(x)u, V = v$$

transforms the system above to

$$V_X = U_T, V_T = f(U)\, U_X/U,$$

which is known to be linearizable by hodograph transformation. When $f(pu) = b/(pu)$ and $p(x) = \exp(ax)$, a, b constant, the above system is $v_x = u_t$, $v_t = bu_x/u^2 + ab/u$ and is linearizable as was shown in Varley and Seymour [69].

5.6 d'Alembert-Hamilton System

As a final example for this chapter, we consider the $n+1$ d'Alembert-Hamilton system considered by Collins [16]

$$\begin{aligned} u_{x_1x_1} + u_{x_2x_2} + ... + u_{x_nx_n} - u_{tt} &= f(u), \\ u_{x_1}^2 + u_{x_2}^2 + ... + u_{x_n}^2 - u_t^2 &= 1 \end{aligned} \tag{5.20}$$

The objective is to discover the forms of $f(u)$ for which an analytic solution exists.

The actual problem considered here is a transformation of (5.20), where $t \mapsto ix_{n+1}$, making the equations symmetric with respect to all independent variables.

To date, the only prominent differential elimination package capable of working with arbitrary functions of the dependent variables of the problem is Mansfield's *DiffGrob2* package (see [37],[38] and [39]). It is still possible to analyze this system with other packages by means of a hodograph transformation

$$(x_1, x_2, ..., x_n, x_{n+1}, u) \mapsto (X_1, X_2, ..., X_n, U, X_{n+1}),$$

thus interchanging the roles of one of the independent variables and the dependent variable.

As noted by Mansfield in [39], this system has, thus far, required the use of specialized techniques (such as writing the system in terms of invariants) for classification based on

$f(u)$ for all $n > 1$. This is a direct result of the tendency of differential elimination on systems with large symmetry groups to explode in both time and memory use.

Application of the most recent version of *rifsimp* (since *Maple 7*) was able to complete the classification of the $2+1$ system in reasonable time and memory (requiring 344 c.p.u seconds and 19 MB of memory running on a PII 333MHz machine under Linux). This computation results in the following theorem, which agrees with [39].

Theorem 13 *The d'Alembert-Hamilton system (5.20) has an analytic solution only when*

$$f(u) = \frac{\epsilon}{u+c}$$

where c is arbitrary, and ϵ can take the values 0, 1, or 2.

Chapter 6

Polynomial Greatest Common Divisors (GCD)

6.1 Introduction

In this chapter we investigate the efficient computation of greatest common divisors (GCDs) of multivariate polynomials over the integers $\mathbf{Z}$. Such implementations are vital for many practical applications in Computer Algebra, including differential elimination algorithms.

Initially we describe Brown's algorithm [8], which is most efficient for dense GCD problems. Unfortunately, Brown's algorithm is inefficient when the GCD is small, but we describe a modification of the algorithm, the *DIVBRO* algorithm of Monagan and Wittkopf [43], that is efficient when the GCD is small.

These algorithms are basically dense algorithms, and are most efficient when the number of variables is small, or the inputs and result are dense. Since this is not often the case in many applications, we describe the Zippel approach to tackling these problems [78]. The Zippel approach, however, was originally designed for leading monic GCD with respect to the main variable of the problem, which limits its applicability. We present a modification of this algorithm, *LINZIP*, which requires only modular linear algebra to extend the method to the leading non-monic case.

Finally we present a detailed asymptotic comparison of the *BROWN*, *DIVBRO*, *EEZ-GCD* (Wang [71]) and *LINZIP* algorithms, showing that the two new algorithms (*DIVBRO* and *LINZIP*) are superior for many important classes of GCD problems.

Throughout this chapter we will use the following notations:

$\mathbf{x} : x_1, x_2, ..., x_m$ represents all variables of the problem.

$||a||_\infty$ denotes the height of a polynomial (the magnitude of the largest integer coefficient).

$\mathrm{lc}_\mathbf{x} : R[\mathbf{x}] \to R$ is the leading coefficient of a polynomial using lexicographical order (with $x_1 > x_2 > ... > x_m$).

$\mathrm{cont}_\mathbf{x} : R[\mathbf{x}] \to R$ is the content of a polynomial (the GCD of the coefficients in R).

$\mathrm{pp}_\mathbf{x} : R[\mathbf{x}] \to R[\mathbf{x}]$ is the primitive part of a polynomial (i.e. divided by its content).

$\Delta_\mathbf{x} : R[\mathbf{x}] \to \mathbf{N}^m$ is the vector degree of a polynomial using lexicographical order (with $x_1 > x_2 > ... > x_m$).

$\mathrm{res}_{\mathbf{x_1}} : (R[\mathbf{x}], R[\mathbf{x}]) \to R[x_2, ..., x_n]$ is the resultant of two polynomials with respect to x_1.

6.2 Integer GCDs, Univariate GCDs, and Homomorphisms

To begin, we briefly review of some of the concepts used in modular GCD algorithms.

The first and best known algorithm in the area of GCDs is Euclid's algorithm for computing the GCD of two integers. The algorithm itself is quite simple, but it is the basis of all GCD algorithms discussed in this dissertation.

Algorithm 15 (Euclid's Algorithm)

Input: *Two positive integers a and b.*

Output: *Their integer greatest common divisor g.*

while $b \neq 0$ do

  Set $a = rem(a, b)$

  exchange a and b

end loop

return a

Where rem is the integer remainder operation that computes q,r such that $a = qb + r, 0 \leq r < b$.

The proof that the above algorithm provides the greatest common divisor of a and b is omitted for brevity, but is simply based on the easily verified fact that $\text{GCD}(a,b) = \text{GCD}(\text{rem}(a,b),b)$.

The *Extended Euclidean Algorithm* (or EEA for short) is an extension of the above algorithm that also computes s and t such that $sa + tb = \text{GCD}(a,b)$, and can be obtained with small modifications to the Euclid algorithm. The EEA and its extensions are heavily used in the area of GCD computation, but are also used to obtain solutions of *diophantine* equations.

Now for the extension to problems in the ring $\mathbf{R}[x]$, the same algorithm can be used with some small modifications, but issues arise, in that it may be necessary to extend the coefficient domain $\mathbf{R}$ into a *Euclidean Domain.* The simplest possible definition of a Euclidean domain is one in which Euclid's algorithm works, and requires that the coefficient domain be a field (since we require the ability to perform the remainder operation in the algorithm), but the interested reader can consult Geddes [19] for a more detailed discussion.

In the case of computation of univariate GCD over the integers, one needs to perform intermediate computations in $\mathbf{Q}[x]$, the ring of polynomials with rational coefficients. The unfortunate problem here is that direct application of Euclid's algorithm results in growth in the rational coefficients of the problem. One additional consideration is that any integer GCD that occurs between the contents of the two polynomials will be lost, so must be computed separately, and will represent the content of the GCD in $\mathbf{Z}$.

Example 24 (Euclid in $\mathbf{Z}[x]$) *Consider the computation of the univariate* GCD *of the two polynomials:*

$$\begin{aligned} a &= 3074x^{10} - 160x^9 - 13301x^8 - 4376x^7 - 10476x^6 + 1970x^5 \\ &\qquad + 6004x^4 + 15198x^3 + 8000x^2 + 8179x + 5976, \\ b &= 4558x^{10} - 8529x^9 + 2193x^8 - 13728x^7 + 481x^6 - 20336x^5 \\ &\qquad + 6740x^4 - 5313x^3 + 7382x^2 - 2104x + 6336 \end{aligned}$$

As a first step, we compute the contents of a,b and take their GCD*, obtaining the value* 1 *(the content of the resulting* GCD*). If we now proceed by direct application of Euclid's algorithm,* *working over* $\mathbf{Q}$*, we obtain the result*

$$\begin{aligned} g &= -\frac{3116642696510423648439585612204073264823320648412304}{37909302439451406673262854250877356794828191974827}x^5 \\ &\quad +\frac{294022895897209778154677887943780496681445344189840}{22299589670265533337213443676986680467545995279 31}x^4 \\ &\quad +\frac{2881424379792655825915843301849048867478164373060432}{37909302439451406673262854250877356794828191974827}x^3 \\ &\quad +\frac{4586757175996472539212975051922975748230547369361504}{37909302439451406673262854250877356794828191974827}x^2 \\ &\quad +\frac{588045791794419556309355775887560993362890688379 68}{2229958967026553333721344367698668046754599527931}x \\ &\quad +\frac{4233929700919820805427361586390439152212812956333696}{37909302439451406673262854250877356794828191974827}. \end{aligned}$$

As noted previously, we want our result in $\mathbf{Z}[x]$*, so to obtain this we need to clear denominators, and set the content to* 1 *(computed earlier). In doing so, we obtain the correct result, namely* $53x^5 - 85x^4 - 49x^3 - 78x^2 - 17x - 72$.

Clearly direct application of the Euclid algorithm to this univariate problem caused significant growth in the size of the coefficients throughout the computation. This coefficient growth is in fact exponential in the number of steps of the algorithm, and is typical of naive application of Euclid's algorithm for this class of problems. Many enhancements are possible, such as a monic implementation (where the polynomials are scaled to have leading coefficients of 1), a fraction free form of the algorithm, and one which removes the effect of the multiplier two steps back, but in all cases the coefficient growth is at least linear in the number of steps in the algorithm, and we are working with expressions having larger coefficients than are necessary to represent the inputs and solution of the problem.

A modular approach can be used to obtain better performance for this problem. The advantage of a modular approach is that the coefficients are mapped directly into a finite field $\mathbf{Z}_p$ for prime p where no coefficient growth can occur. We start with a quick illustration of the approach.

Example 25 (Euclid in $\mathbf{Z}_p[x]$) *Consider the problem in the prior example, computing modulo* 211*. We have:*

$$a \equiv 120x^{10} + 51x^9 + 203x^8 + 55x^7 + 74x^6 + 71x^5 + 96x^4 + 6x^3 + 193x^2 + 161x + 68$$

$$b \equiv 127x^{10} + 122x^9 + 83x^8 + 198x^7 + 59x^6 + 131x^5 + 199x^4 + 173x^3 + 208x^2 + 6x + 6$$

We compute the GCD mod 211 *obtaining*

$$g \equiv x^5 + 82x^4 + 15x^3 + 110x^2 + 143x + 134 \pmod{211},$$

which (after appropriate scaling and setting into the symmetric range) gives our GCD

$$53x^5 - 85x^4 - 49x^3 - 78x^2 - 17x - 72.$$

Many details are skipped in the above, such as *What prime should we use? What is the "appropriate scaling"? Does this always work?* We discuss these details now.

In general, it is possible to choose a sufficiently large prime so that the answer can always be directly constructed as in the example, but this is not typically done, as it is more efficient to use a sequence of smaller primes and apply the Chinese Remainder Theorem to reconstruct the result. In addition to an asymptotic improvement to the performance, the primes can be cleverly chosen to reduce the expense of computing a modular GCD (for example, selection of primes so that all coefficient computations for the Euclid algorithm can be done in hardware integer arithmetic).

As for the second question, to obtain the result in $\mathbf{Z}[x]$ from the set of results in $\mathbf{Z}_{p_i}[x]$ one needs to know how to scale the images of the solution (the relative scaling). This is called the *normalization problem*, and can be done in the univariate case by setting the leading coefficient of the GCD to the image of the integer GCD of the leading coefficients of the inputs modulo the current prime. For the example, $\gamma = \text{GCD}(\text{lc}_x(a), \text{lc}_x(b)) = \text{GCD}(3074, 4558) = 106$, and $106 \bmod 211 = 106$, but this is a factor of 2 times too large for the problem. The important thing to note is that the leading coefficient of the GCD must divide the GCD of the leading coefficients of the inputs a,b (γ above), but there may be some extra corresponding to the integer GCD of the leading coefficients of the cofactors. The good news is that once the result is obtained, the spurious integer content can be removed with a sequence of integer GCD computations.

As a side note, the handling of integer contents is not entirely straightforward. It is possible that both the inputs have a common integer GCD component. This is easily computed as the integer GCD of all coefficients of both inputs (the *content*). This can either be simply computed, or computed and removed from the inputs (either way is sufficient). Now the resulting computed GCD should be integer content free, allowing removal of any spurious integer content that is the direct result of the normalization scaling of the problem.

And now for the final question: *Does this always work?* The answer is no, and there are two cases in which it may fail. One of these cases is dubbed the *bad prime* case, and the other the *unlucky prime* case.

Definition 14 (Bad Prime) *A prime p is bad if*

$$\deg_x(\text{GCD}(a \bmod p, b \bmod p)) < \deg_x(g).$$

i.e. if the degree of the GCD *is too low.*

Example 26 (Bad Prime) *Consider the example problem, but perform the computation modulo the prime* 53*. We get:*

$$\begin{aligned} a &\equiv 52x^9 + 2x^8 + 23x^7 + 18x^6 + 9x^5 + 15x^4 + 40x^3 + 50x^2 + 17x + 40 \\ b &\equiv 4x^9 + 20x^8 + 52x^7 + 4x^6 + 16x^5 + 9x^4 + 40x^3 + 15x^2 + 16x + 29 \end{aligned}$$

Computing the GCD $\bmod 53$ *gives:*

$$g \equiv x^4 + 33x^3 + 19x^2 + 32x + 42 \pmod{53},$$

which is of too low a degree.

The main problem with the bad prime case is that there is no way to scale the image based on the leading coefficient estimate for the GCD, so the image cannot be used for reconstruction via Chinese remaindering.

It is easily shown that a *bad prime* must divide the leading coefficient of the GCD. Detection of the possibility that we have chosen a bad prime is dead simple. We need only check the image of the scaling factor γ modulo the chosen prime. If it is non-zero then the prime is not bad. Note also that even if it is zero, then this does not guarantee that the prime is bad, as it could divide the leading coefficients of both cofactors and not the leading coefficient of the GCD, but detecting this is more difficult, so simply rejecting potentially bad primes is a suitable approach.

So given that the chosen prime is not bad, we can state that the computed GCD is at worst a polynomial multiple of the true GCD. To detect that it is a true image of the required GCD the chosen prime must not be *unlucky*.

Definition 15 (Unlucky Prime) *A prime p is unlucky if*

$$\deg_x(\text{GCD}(\bar{a} \bmod p, \bar{b} \bmod p)) > 0,$$

i.e. if the degree of the GCD *of the cofactors is not zero.*

Example 27 (Unlucky Prime) *Consider the example problem, but perform the computation modulo the prime* 13. *We get:*

$$\begin{aligned} a &\equiv 6x^{10} + 9x^9 + 11x^8 + 5x^7 + 2x^6 + 7x^5 + 11x^4 + x^3 + 5x^2 + 2x + 9 \pmod{13}, \\ b &\equiv 8x^{10} + 12x^9 + 9x^8 + 9x^5 + 6x^4 + 4x^3 + 11x^2 + 2x + 5 \pmod{13}. \end{aligned}$$

So first we note that the prime is not bad, as it does not divide $\gamma = 106$. Computing the GCD mod 13 *gives*

$$g \equiv x^6 + 8x^5 + 2x^4 + 6x^3 + 9x^2 + 11x + 12, \quad \pmod{13},$$

which is a problem, as we already know that our GCD *should only be degree 5.*

So in the example, the chosen prime 13 is unlucky. Unfortunately there is no efficient way to detect an unlucky prime in advance. The general approach adopted by many algorithms is to look for an image with too high a degree, relative to the other images, and often includes a division check at the end (which can also be used to check for termination of the algorithm).

This seems suitable as long as unlucky primes are not too plentiful, and this is made clear by the following (simplified) lemma from Geddes [19]

Lemma 10 (Unlucky Primes) *Let a, b be polynomials in $\mathbf{Z}[x]$ with* GCD *g and cofactors $\bar{a}, \bar{b}$. Then p is an unlucky prime if and only if it divides the resultant $r = res_x(\bar{a}, \bar{b})$.*

Obtaining the resultant of the cofactors is much too expensive (for efficient computation of a GCD) but this result does put a finite bound on the number of unlucky primes for a specific problem, making the criteria described earlier sufficient for the task.

For the example problem, the resultant is 1019581298696173769946 which factors as $(2)(3^3)(13)(1452395012387711923)$, so there are only 4 unlucky primes for this problem.

Finally we will briefly describe some of the problems associated with computing GCDs in more than one variable.

All the algorithms that follow at some point reduce multivariate problems to univariate problems by evaluation of the variables until a univariate problem is obtained. Alas the *unlucky* and *bad* problems apply to the evaluation process in much the same way as they do for primes in the modular process. This makes some sense, in that computing mod p, and computing mod $x_i - \alpha_i$ are, loosely speaking, similar operations.

The extension for *bad evaluations* is fairly obvious; selection of a *bad* evaluation point corresponds to choosing the evaluation so that the degrees of the polynomials in the remaining variables drop. For example, in computing the GCD of $(x + y - 1)((y - 1)x + 1)$ and $(x + y - 2)((y - 1)x + 1)$, which is $(y - 1)x + 1$, making the choice $y = 1$ will destroy the GCD.

The extension for *unlucky evaluations* is similar. Selection of an *unlucky* evaluation point corresponds to choosing the evaluation so that a GCD appears between the cofactors. For example, in computing the GCD of the relatively prime polynomials $x^2 + y - 1$ and $(y - 1)x^2 + 1$, the choice $y = 2$ could prove troublesome, as then both polynomials are $x^2 + 1$, making it appear as though they have a nontrivial GCD.

As with the bad prime case, the *bad evaluation* case is easy to avoid in advance, but the *unlucky evaluation* case is not, so it is typically handled as an integral part of the algorithm.

The final problem we discuss is specific to lifting-based algorithms such as *EEZ-GCD*, and it is called the *bad zero* problem. Lifting methods use the EEA to reconstruct variable dependence. The lifting process only requires one of the GCD inputs, and attempts to reconstruct the GCD and cofactor from the chosen input. In addition, only a single prime, and a single evaluation of each variable are used throughout. For example, for a problem with $a = \bar{a}g$, we would need to solve the EEA for σ, τ in

$$\sigma g + \tau \bar{a} = c$$

for a given value of c.

Now one requirement of the approach (for the EEA to work) is that the two factors in use

(g and $\bar{a}$) must be relatively prime. If not, then c needs to be a multiple of GCD$(g, \bar{a})$, which is not always possible. Even if the initial g and $\bar{a}$ are relatively prime before evaluation, once they are evaluated down to univariate polynomials, they may lose their relative primality.

Before we disregard this as being rare, we need to consider the following fact: *lifting based algorithms are significantly more efficient when the evaluation points are chosen to be zero*. Unfortunately this is also the most common case where the relative primality is lost, as zero evaluations remove terms from the problem inputs.

For example, consider the computation of a GCD where the form of $a = \bar{a}g$ is $(x^3 + yz^3)(x^2 + yz)$. For this problem, the two input factors are relatively prime, but the primality is lost if we evaluate *any* of the variables to zero.

The case where evaluating a variable to zero causes the two factors to have a common GCD is called the *bad zero* problem.

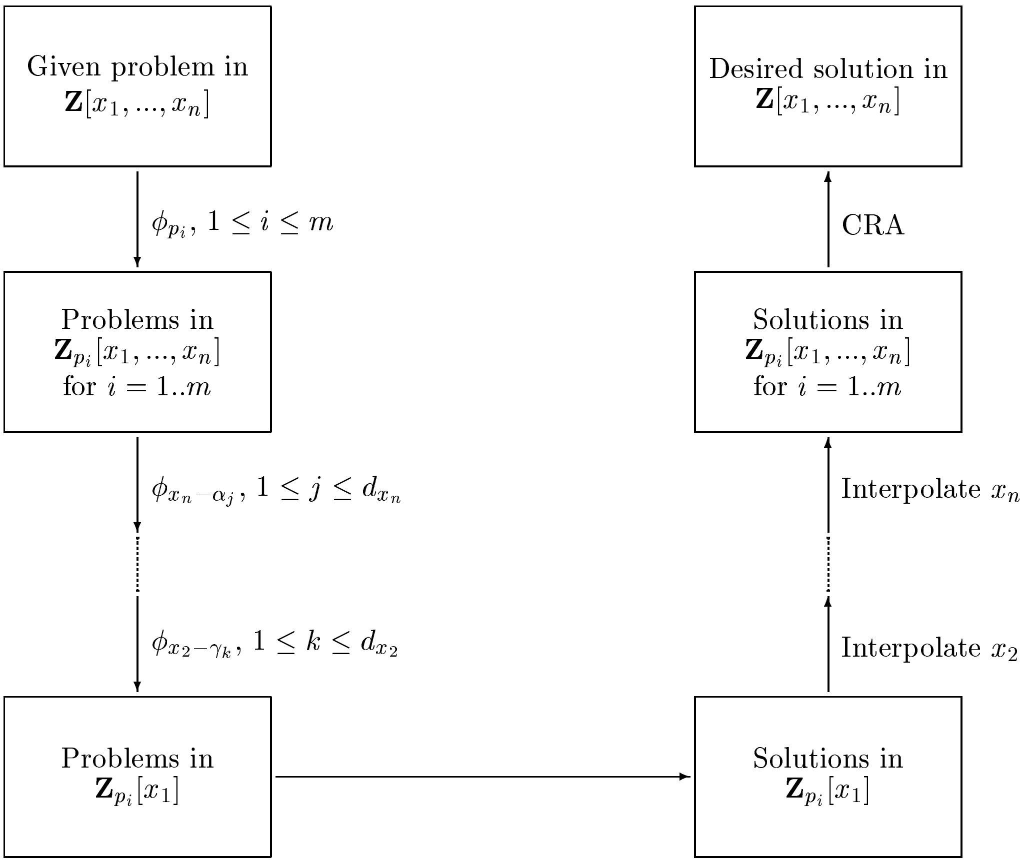

Figure 6.1: Homomorphism Diagram for Brown's Algorithm

6.3 Brown's Dense GCD Algorithm

Brown's description of the "dense modular GCD algorithm" in [8] is reproduced here for the purpose of asymptotic analysis. Brown's algorithm has two subroutines, M and P. The main subroutine M maps a polynomial GCD computation from $\mathbf{Z}[\mathbf{x}]$ (or $\mathbf{Z}[x_1, ..., x_n]$) into one or more GCD computations in $\mathbf{Z}_p[\mathbf{x}]$ for $p \in p_0, p_1, \ldots$, and applies the Chinese remainder theorem to obtain the solution in $\mathbf{Z}[\mathbf{x}]$. Brown's presentation of subroutine P is similar. It maps a polynomial GCD computation from $\mathbf{Z}_p[x_n][x_1, ..., x_{n-1}]$ into one or more GCD computations in $\mathbf{Z}_p[x_1, ..., x_{n-1}]$ by reducing the input polynomials modulo $x_n = \alpha$ for sufficiently many $\alpha \in \mathbf{Z}_p$, then recovers the dependence of x_n by application of the Chinese remainder theorem.

An important observation to make about Brown's algorithm is that both subroutines have been deliberately designed so that their termination does not require a trial division of the inputs by the GCD, nor multiplication of the GCD and its cofactors to verify that the results are correct. This feature leads to good performance for some GCD problems but poor performance for others.

As hinted in the prior section, both algorithms will suffer from the *bad* and *unlucky* prime and evaluation problems, and these are handled by the algorithms.

Algorithm 16 (Brown M, 1971)

Input: $a, b \in \mathbf{Z}[\mathbf{x}] \setminus \{0\}$, $\mathbf{x} = x_1, ..., x_n$

Output: $g = \text{GCD}(a, b)$, $\bar{a} = a/g$, $\bar{b} = b/g$

1 *Compute integer contents and their* GCD.
Set $ca = \text{cont}_{\mathbf{x}}(a) \in \mathbf{Z}$, $a = a/ca$, $cb = \text{cont}_{\mathbf{x}}(b) \in \mathbf{Z}$, $b = b/ca$, *and* $cg = \text{GCD}(ca, cb)$.

2 *Compute the correction coefficients.*
Set $la = \text{lc}_{\mathbf{x}}(a) \in \mathbf{Z}$, $lb = \text{lc}_{\mathbf{x}}(b) \in \mathbf{Z}$, *and* $\gamma = \text{GCD}(la, lb)$.

3 *Estimate bound for twice the height of* $\gamma \times g$, $\gamma \times \bar{a}$, $\gamma \times \bar{b}$ *as* $B = 2\gamma \max(||a||_\infty, ||b||_\infty)$.

4 *Set* $m = 1$ *(the product of the moduli).*
Set $\mathbf{e} = \min(\Delta_{\mathbf{x}} a, \Delta_{\mathbf{x}} b) \in \mathbf{Z}^n$.

Loop:

6 *Choose a new prime* p *that does not divide* la *or* lb.

7 *Compute* $a_p = a \bmod p, b_p = b \bmod p$.

8 *Compute* $g_p, \bar{a}_p, \bar{b}_p \in \mathbf{Z}_p[\mathbf{x}]$ *the monic* GCD *of* a_p *and* b_p *and their cofactors using algorithm P. If P fails (too few evaluation points are available) goto Loop.*

9 *Test for* a, b *relatively prime.*
If $\Delta_{\mathbf{x}} g_p = \mathbf{0}$ *then output* cg, $(ca/cg)a$, $(cb/cg)b$.

10 *If* $\Delta_{\mathbf{x}} g_p > \mathbf{e}$ *then (skip this unlucky prime) goto Loop.*

11 *Leading coefficient correction in* $\mathbf{Z}$.
Set $g_p = \gamma \times g_p \bmod p$.

12 *First image?*
If $m = 1$ *then set* $g_m = g_p$, $\bar{a}_m = \bar{a}_p$, $\bar{b}_m = \bar{b}_p$, $m = p$, $\mathbf{e} = \Delta_{\mathbf{x}} g_p$ *and goto Loop.*

13 *If* $\Delta_{\mathbf{x}} g_p < \mathbf{e}$ *then all previous primes were unlucky. Restart the algorithm keeping only the current prime.*

Set $g_m = g_p$, $\bar{a}_m = \bar{a}_p$, $\bar{b}_m = \bar{b}_p$, $m = p$, $\mathbf{e} = \Delta_{\mathbf{x}} g_p$ *and goto Loop.*

14 *Combine the new image with the old using the CRA and express the result in the symmetric range for* $\mathbf{Z}_p$.

Set $\bar{a}_m = \mathrm{CRA}([\bar{a}_p, \bar{a}_m], [p, m])$, $\bar{b}_m = \mathrm{CRA}([\bar{b}_p, \bar{b}_m], [p, m])$,

$g_m = \mathrm{CRA}([g_p, g_m], [p, m])$.

15 *Set* $m = m \times p$, *if* $m \leq B$ *then goto Loop.*

16 *Test for termination.*

Without explicitly computing $g_m \times \bar{a}_m$ *and* $g_m \times \bar{b}_m$ *determine if* $||g_m \times \bar{a}_m||_\infty < m/2$ *and* $||g_m \times \bar{b}_m||_\infty < m/2$. *If so then we have* $m \,|\, (\gamma a - g_m \times \bar{a}_m)$ *and* $m \,|\, (\gamma b - g_m \times \bar{b}_m)$ *and also* $\gamma a - g_m \times \bar{a}_m = 0$ *and* $\gamma b - g_m \times \bar{b}_m = 0$ *over* $\mathbf{Z}$, *hence we are done; so set* $g = \mathrm{pp}_{\mathbf{x}}(g_m)$, $\delta = \mathrm{lc}_{\mathbf{x}}(g) \in \mathbf{Z}$, $\bar{a} = \bar{a}_m/\delta$, $\bar{b} = \bar{b}_m/\delta$.

Output $cg \times g$, $(ca/cg)\bar{a}$, $(cb/cg)\bar{b}$.

17 *Goto Loop – either we do not have enough primes yet or all primes are unlucky.*

Algorithm 17 (Brown P, 1971)

Input: $a, b \in \mathbf{Z}[x_n][x_1, ..., x_{n-1}] \setminus \{0\}$, p *prime.*

Output: $g = \mathrm{GCD}(a, b)$, $\bar{a} = a/g$, $\bar{b} = b/g$

1 *If* $n = 1$ *then compute the* GCD *using Euclid's algorithm and the cofactors by division and return, otherwise compute contents in* $\mathbf{Z}_p[x_n]$ *and their* GCD.

Set $ca = \mathrm{cont}_{x_1,...,x_{n-1}}(a) \in \mathbf{Z}_p[x_n]$, $a = a/ca$,

$cb = \mathrm{cont}_{x_1,...,x_{n-1}}(b) \in \mathbf{Z}_p[x_n]$, $b = b/ca$, *and* $cg = \mathrm{GCD}(ca, cb)$.

2 *Compute the correction coefficients.*

Set $la = \mathrm{lc}_{x_1,...,x_{n-1}}(a) \in \mathbf{Z}_p[x_n]$, $lb = \mathrm{lc}_{x_1,...,x_{n-1}}(a) \in \mathbf{Z}_p[x_n]$, $\gamma = \mathrm{GCD}(la, lb)$.

3 *Bound the degree in* x_n *of* $\gamma \times g$, $\gamma \times \bar{a}$, $\gamma \times \bar{b}$. *Set* $B = \deg_{x_n} \gamma + max(\deg_{x_n} a, \deg_{x_n} b)$.

4 *Set* $m = 1$ *(the product of the moduli). Set* $\mathbf{e} = \min(\Delta_{x_1,...,x_{n-1}} a, \Delta_{x_1,...,x_{n-1}} b) \in \mathbf{Z}^{n-1}$.

Loop:

6 *Choose a new evaluation point* $\alpha \in \mathbf{Z}_p$ *such that* $x_n - \alpha$ *does not divide* $la \times lb$. *If no such evaluation point exists then* p *is too small and the algorithm fails.*

7 *Compute* $a_\alpha = a \bmod \langle x_n - \alpha\rangle \in \mathbf{Z}_p[x_1, ..., x_{n-1}]$, *and*
$b_\alpha = b \bmod \langle x_n - \alpha\rangle \in \mathbf{Z}_p[x_1, ..., x_{n-1}]$.

8 *Compute* $g_\alpha, \bar{a}_\alpha, \bar{b}_\alpha \in \mathbf{Z}_p[x_1, ..., x_{n-1}]$ *the monic* GCD *of* a_α *and* b_α *and their cofactors with a recursive call to this algorithm.*

9 *Test for* a, b *relatively prime.*
If $\Delta_{x_1,...,x_{n-1}} g_\alpha = \mathbf{0}$ *then output* cg, $(ca/cg)a$, $(cb/cg)b$.

10 *If* $\Delta_{x_1,...,x_{n-1}} g_\alpha > \mathbf{e}$ *then (skip this unlucky evaluation) goto Loop.*

11 *Leading coefficient correction in* $\mathbf{Z}_p[x_n]$.
Set $g_\alpha = \gamma(\alpha) \times g_\alpha \bmod p$.

12 *First image?*
If $m = 1$ *then set* $g_m = g_\alpha$, $\bar{a}_m = \bar{a}_\alpha$, $\bar{b}_m = \bar{b}_\alpha$, $m = x_n - \alpha$, $\mathbf{e} = \Delta_{x_1,...,x_{n-1}} g_\alpha$ *and goto Loop.*

13 *If* $\Delta_{x_1,...,x_{n-1}} g_\alpha < \mathbf{e}$ *then all previous evaluations were unlucky.*
Restart the algorithm keeping only the current evaluation.
Set $g_m = g_\alpha$, $\bar{a}_m = \bar{a}_\alpha$, $\bar{b}_m = \bar{b}_\alpha$, $m = x_n - \alpha$, $\mathbf{e} = \Delta_{x_1,...,x_{n-1}} g_\alpha$ *and goto Loop.*

14 *Combine the new image with the old using the CRA.*
Set $\bar{a}_m = \mathrm{CRA}([\bar{a}_\alpha, \bar{a}_m], [x_n - \alpha, m])$, $\bar{b}_m = \mathrm{CRA}([\bar{b}_\alpha, \bar{b}_m], [x_n - \alpha, m])$,
$g_m = \mathrm{CRA}([g_\alpha, g_m], [x_n - \alpha, m])$.

15 *Set* $m = m \times (x_n - \alpha)$.
If $\deg_{x_n} m \le B$ *then goto Loop.*

16 *Test for termination.*
If $\deg_{x_n} g_m + \deg_{x_n} \bar{a}_m = \deg_{x_n} \gamma + \deg_{x_n} a$ *and* $\deg_{x_n} g_m + \deg_{x_n} \bar{b}_m = \deg_{x_n} \gamma + \deg_{x_n} b$ *then set* $g = \mathrm{pp}_{x_1,...,x_{n-1}}(g_m)$, $\delta = \mathrm{lc}_{x_1,...,x_{n-1}} g$, $\bar{a} = \bar{a}_m/\delta$, $\bar{b} = \bar{b}_m/\delta$, *and output* $cg \times g$, $(ca/cg)\bar{a}$, $(cb/cg)\bar{b}$.

17 *Goto Loop – all* α'*s are unlucky.*

In order to determine the complexity of Brown's algorithm, the central quantities of interest will be the number of primes p used by algorithm M and the number of evaluation points α used by algorithm P. The reader will note that these quantities, computed in step [3] and used in step [15] of both algorithms, are determined by the size of the inputs a and b and not by the size of $g, \bar{a}$ and $\bar{b}$.

6.4 *DIVBRO*: The Modified Brown Algorithm

A modification of the Brown algorithm, *DIVBRO*, was presented by Monagan and Wittkopf in [43], and provides greater efficiency for many classes of problems. It uses the well known alternative of a division test to check for algorithm termination, and only constructs the GCD, not the cofactors.

The new aspects of the approach that provide the increased efficiency are as follows:

Early Termination: Since we are only reconstructing g, we may need fewer primes in algorithm M than required for reconstruction of g and the cofactors. In addition, the bounds on the coefficient size of g are often pessimistic, forcing too many additional images to be computed. If we apply the criteria that we perform the division check in M once g is unchanged for an iteration, with high probability we need only perform that division once, and we will compute at most one extra image.

Degree Bounds: Since only the GCD is to be constructed, having reasonable bounds on the degree of the GCD in $x_1, ..., x_n$ can be used as a stopping criteria for algorithm P, and helps with detection of unlucky primes and evaluations.

One-Pass Newton: Since we have degree bounds, the reconstruction of the current variable in algorithm P can be done more efficiently, as we can perform the Newton interpolation once all images have been computed, instead of reconstructing incrementally.

Bivariate Optimizations: In the bivariate case, additional optimizations are available, including a way to remove the division test in algorithm P and improved handling of content computations for the GCD. These optimizations make a significant difference for bivariate problems.

The details of the bivariate optimizations can be found in [43], but in this presentation we are primarily concerned with the general multivariate case, so we will not discuss the bivariate case further here. We present the modified algorithms below, where we note that the numbering of the steps from Brown algorithms has been retained when possible (so the numbering is not entirely sequential):

Algorithm 18 (DIVBRO M)

Input: $a, b \in \mathbf{Z}[\mathbf{x}] \setminus \{0\}$, $\mathbf{x} = x_1, ..., x_n$

Output: $g = \text{GCD}(a, b)$

1 *Compute integer contents and their* GCD.
Set $ca = \text{cont}_{\mathbf{x}}(a) \in \mathbf{Z},\ a = a/ca,\ cb = \text{cont}_{\mathbf{x}}(b) \in \mathbf{Z},\ b = b/ca,$ *and* $cg = \text{GCD}(ca, cb)$.

2 *Compute the correction coefficient.*
Set $la = \text{lc}_{\mathbf{x}}(a) \in \mathbf{Z},\ lb = \text{lc}_{\mathbf{x}}(b) \in \mathbf{Z},$ *and* $\gamma = \text{GCD}(la, lb)$.

4 *Set* $m = 1$ *(the product of the moduli).*
Compute degree bounds $\mathbf{e}$ *for the* GCD *in* $x_1, ..., x_n$.

Loop:

6 *Choose a new prime* p *that does not divide* γ.

7 *Compute* $a_p = a \bmod p, b_p = b \bmod p$.

8 *Compute* $g_p \in \mathbf{Z}_p[\mathbf{x}]$ *the monic* GCD *of* a_p *and* b_p *using algorithm* P.
If P *fails (too few evaluation points are available, or unlucky) goto Loop.*

9 *Test for* a, b *relatively prime: If* $\Delta_{\mathbf{x}} g_p = \mathbf{0}$ *then output* cg.

11 *Leading coefficient correction in* $\mathbf{Z}$.
Set $g_p = \gamma \times g_p \bmod p$.

12 *First image?*
If $m = 1$ *then set* $g_m = g_p$, $m = p$, $\mathbf{e} = \Delta_{\mathbf{x}} g_p$ *and goto Loop.*

13 *If* $\Delta_{\mathbf{x}} g_p < \mathbf{e}$ *then all previous primes and the degree bounds were unlucky.*
Restart the algorithm keeping only the current prime and the new bounds.
Set $g_m = g_p$, $m = p$, $\mathbf{e} = \Delta_{\mathbf{x}} g_p$ *and goto Loop.*

14 *Combine the new image with the old using the CRA and express the result in the symmetric range for* $\mathbf{Z}_p$.
Set $g_m = \text{CRA}([g_p, g_m], [p, m])$.

15 *Set* $m = m \times p$, *and if* g_m *changed in step 14 goto Loop.*

16 *Test for termination computing* $g = \text{pp}_{\mathbf{x}}(g_m)$ *and performing a division check to assure that* g *divides both* a *and* b. *If so, we are done so output* $cg \times g$.

17 *Goto Loop – either we do not have enough primes yet or all primes are unlucky.*

Algorithm 19 (DIVBRO P)

Input: $a, b \in \mathbf{Z}[x_1, ..., x_{n-1}] \setminus \{0\}$, *p prime, and the degree bounds* $\mathbf{e_0}$.

Output: $g = \text{GCD}(a, b)$ *or* **Fail**

1 *If* $n = 1$ *then compute the* GCD *using Euclid's algorithm and return, otherwise compute contents in* $\mathbf{Z}_p[x_n]$ *and their* GCD.
Set $ca = \text{cont}_{x_1,...,x_{n-1}}(a) \in \mathbf{Z}_p[x_n]$, $a = a/ca$,
$cb = \text{cont}_{x_1,...,x_{n-1}}(b) \in \mathbf{Z}_p[x_n]$, $b = b/ca$, *and* $cg = \text{GCD}(ca, cb)$.

2 *Compute the correction coefficient.*
Set $la = \text{lc}_{x_1,...,x_{n-1}}(a) \in \mathbf{Z}_p[x_n]$, $lb = \text{lc}_{x_1,...,x_{n-1}}(a) \in \mathbf{Z}_p[x_n]$, $\gamma = \text{GCD}(la, lb)$.

3 *Set the initial degree bounds for this computation,* $\mathbf{e} = \mathbf{e_0}$.

4 *Set* $g_{seq} = \mathbf{Null}$, $v_{seq} = \mathbf{Null}$.

Loop:

6 *Choose a new evaluation point* $\alpha \in \mathbf{Z}_p$ *such that* $x_n - \alpha$ *does not divide* γ. *If no such evaluation point exists then p is too small, so return* **Fail**.

7 *Compute* $a_\alpha = a \bmod \langle x_n - \alpha \rangle \in \mathbf{Z}_p[x_1, ..., x_{n-1}]$, *and*
$b_\alpha = b \bmod \langle x_n - \alpha \rangle \in \mathbf{Z}_p[x_1, ..., x_{n-1}]$.

8 *Compute* g_α *the monic* GCD *of* a_α *and* b_α *with a recursive call to this algorithm.*

10 *If the recursive call fails* ($n > 2$), *or if the* GCD *is of too high a degree* ($n = 2$) *then (skip this unlucky evaluation) goto Loop* unless *this has occurred twice with no successful images, in which case we assume that a higher prime or evaluation is unlucky and return* **Fail**.

9 *Test for* a, b *relatively prime: If* $\Delta_{x_1,...,x_{n-1}} g_\alpha = \mathbf{0}$ *then output* cg.

11 *Leading coefficient correction in* $\mathbf{Z}_p[x_n]$.
Set $g_\alpha = \gamma(\alpha) \times g_\alpha \bmod p$.

12 *First image?*
If $g_{seq} = \mathbf{Null}$ *then set* $g_{seq} = g_\alpha$, $v_{seq} = \alpha$, $\mathbf{e} = \Delta_{x_1,...,x_{n-1}} g_\alpha$ *and goto Loop.*

13 *If* $\Delta_{x_1,...,x_{n-1}} g_\alpha < \mathbf{e}$ *then all previous evaluations and the degree bound were unlucky. Restart the algorithm keeping only the current evaluation.*
Set $g_{seq} = g_\alpha$, $v_{seq} = \alpha$, $\mathbf{e} = \Delta_{x_1,...,x_{n-1}} g_\alpha$ *and goto Loop.*

14 *Add the new image and evaluation point*
Set $g_{seq} = g_{seq}, g_\alpha, \;\; v_{seq} = v_{seq}, \alpha$

15 *If we have enough images to construct to the degree bound in* x_n *plus* $\deg_{x_n}(\gamma)$ *then goto 18, otherwise goto Loop.*

18 *Compute the* GCD g *via Newton interpolation in* x_n *using the sequence of* GCDs g_{seq} *and points* v_{seq}.

19 *Test for correctness*
Remove the content $g = \mathrm{pp}_{x_1,\ldots,x_{n-1}}(g_m)$ *and check that* g *divides* a *and* b*. If so, output* $cg \times g$*, otherwise return* **Fail** *(unlucky content in* x_n*).*

The main advantage of this approach is that the core computational complexity of the algorithm is based on the coefficient size and the degree of g, unlike Brown's algorithm which is based on the coefficient size and the degree of the inputs a and b.

6.5 Zippel's Sparse GCD Algorithm

Zippel [78] proposed an algorithm (which can be used in combination with Brown's algorithm) that is more efficient for sparse problems. The main idea behind this algorithm is as follows: *Once we know the form of the* GCD *for a specific prime or evaluation, we can use this information to more efficiently compute additional images for other primes or evaluations.*

Example 28 *Consider the computation of the bivariate* GCD $x^3 + 12xy + 100$*. Brown's P algorithm would call Brown's M algorithm to compute the* GCD g *as*

$$g \equiv x^3 + 12xy^2 + 9 \pmod{13}.$$

continuing to follow the Brown approach would require additional calls to Brown's M algorithm to compute other images, say, $\bmod 17$, $\bmod 23$*, etc.*

The Zippel approach instead utilizes information about the form of the GCD *obtained in the first call assuming that the* GCD *is of the form* $x^3 + \alpha xy^2 + \beta$ *for some* α, β.

If this is the case, all information required for a new image $\bmod 17$ *can be obtained from a single univariate* GCD *call.*

The GCD *is computed* $\bmod \langle y-2,17\rangle$ *as* $x^3+14x+15$, *which is then equated to the image of the assumed form of the* GCD *mod* $\langle y-2,17\rangle$ *as* $x^3+4\alpha x+\beta$, *and we obtain* $\alpha \equiv 12 \bmod 17$, $\beta \equiv 15 \bmod 17$, *giving us the new image*

$$g \equiv x^3+12xy^2+15 \pmod{17}.$$

So as hinted at by the example, recursive calls to compute new GCD images can be replaced by the solution of a sequence of linear systems, and a bit of manipulation.

Zippel's algorithm M is described here.

Algorithm 20 (Zippel M)

Input: $a, b \in \mathbf{Z}[\mathbf{x}] \setminus \{0\}$

Output: $g = \text{GCD}(a, b)$

1 *Compute integer* GCD $cg = \text{GCD}(\text{cont}_{\mathbf{x}}(a), \text{cont}_{\mathbf{x}}(b))$

2 *Compute the correction coefficient* $\gamma = \text{GCD}(\text{lc}_{\mathbf{x}}(a), \text{lc}_{\mathbf{x}}(b))$

3 *Estimate coefficient bound* $B = 2\gamma \max(||a||_\infty, ||b||_\infty)$

4 *Compute a modular* GCD *image* g_1 *with a call to the Zippel P algorithm for a random prime* p_1.

5 *Assume that of step 4 produces a correct image of the* GCD *with no missing terms, and that the prime is not unlucky. We call the assumed form* g_f. *Count the largest number of non-zero terms in any coefficient of the main variable* x_1, *calling this* n_x.

6 *Set* $g_{seq} = g_1$ *and* $p_{seq} = p_1$.

7 *repeat*

7.1 *Choose a new prime* p_i *that it does not divide* γ.

7.2 *Set* $S = \emptyset$, $n_i = 0$

7.3 *repeat*

7.3.1 *Choose* $\alpha_2, ..., \alpha_n \in \mathbf{Z}_p$ *at random such that* $\deg_{x_1}(a \bmod I) = \deg_{x_1}(a)$, *or* $\deg_{x_1}(b \bmod I) = \deg_{x_1}(b)$, *where* $I = \langle x_2-\alpha_2, ..., x_n-\alpha_n\rangle$.

7.3.2 *Compute* $g_i = \text{GCD}(a \bmod I, b \bmod I)$.

7.3.3 *Check if* $\deg_{x_1}(g_i) < \deg_{x_1}(g_f)$. *If this occurs then our original image was unlucky, so goto 4.*

7.3.4 *Check if* $\deg_{x_1}(g_i) > \deg_{x_1}(g_f)$. *If this occurs then our current image is unlucky, so goto 7.3.1.*

7.3.5 *Add the equations obtained from equating coefficients of* g_i *and the evaluation of* $g_f \bmod I$ *to* S, *and set* $n_i = n_i + 1$.

Until $n_i \geq n_x$

7.4 *We should now have a sufficient number of equations in* S *to solve for all unknowns in* $g_f \bmod p_i$ *so attempt this now, scaling the leading coefficients of the* g_i *to* $\gamma \bmod p_i$.

7.5 *If the system is under-determined, then we need more images, so goto 7.3.1.*

7.6 *If the system is inconsistent, then our original image must be missing terms, or the original prime was unlucky, so goto 4.*

7.7 *The resulting solution is consistent and determined, so we have a new image* g_i, *so add to* g_{seq} *and add the prime to* p_{seq}

Until we have a sufficient number of primes to reconstruct our GCD *to* B.

8 *Reconstruct our candidate* GCD g *using Chinese remaindering on* g_{seq},p_{seq}.

9 *Perform a division test on* g *relative to the inputs. If it fails our original image must be missing terms, or our original prime was unlucky, so goto 4.*

10 *return* g.

Algorithm P can be easily obtained from algorithm M by replacing the coefficient bound by a degree bound, the call to Zippel P by a recursive call in one fewer variable, and the selection of new primes and evaluations mod a prime by the selection of new random values v_i for the current variable x_j and evaluations mod $\langle x_2 - \alpha_2, ..., x_{j-1} - \alpha_{j-1}, p\rangle$, where p is the prime passed down from algorithm Zippel M.

A number of comments must be made on the above algorithm, most notably when and why failures occur, and limitations of the algorithm.

- The algorithm heavily depends on the completeness of the GCD image computed in step 2, where by completeness we mean that the image contains all the terms present in the GCD. It is straightforward to see that for a t term polynomial the probability that a term may be missed is $\mathcal{O}(\frac{t}{p})$, where p is the current prime, and $p \gg t$. This is not such a restriction as long as either t is sufficiently small, or p is sufficiently large, but can be a problem for a greater number of terms or small primes.

- Bad results will always be detected by either the checks within the algorithm, or the division test at the end, so if the algorithm returns a result it will be correct.

- The termination of the algorithm depends upon the allowable choices of p, and the form of the resulting GCD. If $p \gg t$ then the algorithm will likely terminate in the first pass, while for $p \simeq t$ the algorithm is unlikely to terminate.

- The algorithm, as stated in [78], is only applicable to GCDs that are monic in the main variable, which means the well known *content* and *normalization* problems are not handled by this approach for the multivariate part (in Zippel P).

In the following section we will develop a modification of this algorithm that handles the *content* and *normalization* problems, extending the algorithm to apply to a larger class of problems.

6.6 ***LINZIP***: The Modified Zippel Algorithm

In this section we will develop a modification of the Zippel sparse algorithm that is applicable to all inputs in $\mathbf{Z}[x_1, ..., x_n]$.

The first problem we consider is the *normalization* problem. This is best illustrated through a pair of examples, one of which is a straightforward modification, the other is not.

Example 29 *Consider the computation of the bivariate* GCD $3x^3y^2 + 12y + 100$ *at the top level using the Zippel algorithm (20). In addition, we assume that we have* $\gamma = 3$, *so we know the leading coefficient of our* GCD. *The image obtained in step 4 of the algorithm, with* $p_1 = 13$ *would be*

$$g \equiv x^3y^2 + 4y + 3 \pmod{13},$$

so our expected form of the GCD *, after scaling through by* γ, *is* $g_f = 3x^3y^2 + \alpha y + \beta$.

Since the $\mathcal{O}(x^0)$ term has two terms, we need to compute two modular images for our next prime to reconstruct the y dependence. For the next prime, $p_2 = 17$, we choose $y = 1, 2$ and obtain the GCD*s $x^3 + 9$ and $x^3 + 16$ respectively.*

Before converting to a linear system we need to normalize the computed GCD*, as otherwise the linear relations will give an incorrect result. Fortunately this is easy to do here, as we know the form of the leading coefficient, including the y dependence, which is $3y^2$. As a result we must scale our first univariate* GCD *by $3(1)^2 = 3$, and the second by $3(2)^2 = 12$ before equating, giving:*

$$\begin{aligned} 3x^3 + \alpha + \beta &= 3x^3 + 10 \\ 12x^3 + 2\alpha + \beta &= 12x^3 + 5 \end{aligned}$$

Solving this system for $\alpha, \beta \bmod 17$ gives $\alpha = 12, \beta = 15$ resulting in the new image

$$g_2 \equiv 3x^3y^2 + 12y + 15 \pmod{17}.$$

Additional images for other primes can be computed in the same way.

So for a coefficient known up to a scaling there is no problem. What is interesting is what occurs when we can no longer scale a single coefficient of the GCD for reconstruction, as illustrated by the next example.

Example 30 *Consider the computation of the bivariate* GCD *$x^3(3y^2 - 90) + 12y + 100$ at the top level using the Zippel algorithm (20). The image obtained in step 2 of the algorithm, with $p_1 = 13$ would be*

$$g_1 = x^3y^2 + 9x^3 + 4y + 3 \pmod{13},$$

so our expected form of the GCD *is $g_f = x^3y^2 + \alpha x^3 + \beta y + \gamma$.*

Since each x term of the GCD *has two terms, we need to compute two modular images for our next prime to reconstruct the y dependence. For the next prime, $p_2 = 17$, we choose $y = 1, 2$ and obtain the* GCD*s $x^3 + 12$ and $x^3 + 8$ respectively.*

Now we run into a problem when attempting to normalize the computed GCD*s to construct the linear system we require. The difficulty is this: All coefficients of the* GCD *have unknown parameters present, so we have no way of scaling the* GCD *unless we know the exact form of one of the coefficients.*

A solution to the problem illustrated by the above example could be obtained in a number of ways. One approach involves computation of one of the required coefficients exactly, for example by factoring the leading coefficient, as is done by the *EEZ-GCD* algorithm. We wish to avoid this as it could be expensive, and it complicates the implementation of the algorithm.

This problem only occurs when **all** coefficients of the GCD with respect to the main variable have more than one term. In any other case, we can choose **any** single term coefficient to use for scaling (not just the leading or trailing coefficient) as the form of single term coefficients is always known up to a scaling of the entire GCD.

It is clear that the problem boils down to how to scale the univariate GCD images in such a way that the solution of the linear system produces the correct result.

The approach followed now is quite simple in concept. It is to treat the scaling factors of the computed univariate GCDs as unknowns as well. This results in a larger linear system, and sometimes requires additional univariate GCD images, but never requires additional multivariate GCD computations. We call this the *multiple scaling* case (as opposed to the *single scaling* case).

If we allow all univariate GCDs to be scaled along with all coefficients of the assumed form of the multivariate GCD, the resulting system will be under-determined by exactly 1 unknown, as the computation is only determined up to a scaling factor. Rather than fixing an unknown in the form of the GCD to 1, we instead fix the scaling factor of the first GCD to 1 (for reasons to be explained later).

The following example illustrates this approach.

Example 31 *Consider the computation of the bivariate* GCD *from Example 30:* $(3y^2-90)x^3+12y+100$*. Just as in that example, we obtain for* $p_1=13$

$$g \equiv x^3y^2+9x^3+4y+3 \pmod{13},$$

and expected form of the GCD $g_f=\alpha x^3y^2+\beta x^3+\gamma y+\sigma$*.*

Instead of computing two univariate GCD *images for the new prime* $p_2=17$*, we compute three, choosing* $y=1,2,3$ *and obtaining the* GCD*s* x^3+12*,* x^3+8*, and* x^3 *respectively. We* *form the modified system as follows:*

$$\begin{aligned}
\alpha x^3 + \beta x^3 + \gamma + \sigma &= m_1\,(x^3+12) = x^3+12, \\
4\alpha x^3 + \beta x^3 + 2\gamma + \sigma &= m_2\,(x^3+8), \\
9\alpha x^3 + \beta x^3 + 3\gamma + \sigma &= m_3\,(x^3),
\end{aligned}$$

where m_2, m_3 *are the new unknown scaling factors, and we have set the first scaling factor* m_1 *to* 1.

Solving this system yields $\alpha = 7, \beta = 11, \gamma = 11, \sigma = 1$*, with scaling factors* $m_2 = 5, m_3 = 6$*, so our new* GCD *image is given by*

$$g \equiv 7x^3y^2 + 11x^3 + 11y + 1 \pmod{17},$$

which is consistent with our GCD.

Now we explain the reason for fixing a multiplier value instead of an unknown in the GCD. In general it is possible to select a prime or evaluation so that the chosen scaling term in the GCD is zero, but there is no way to detect it. For the example, suppose we set $\alpha = 1$. In this case, the evaluation $y = 0$ is not bad (the form of the GCD is still cubic in x) but $\alpha = 0$, so the resulting system will be incorrect. Attempting to set $\beta = 1$ will have the same problem if we used the prime $p = 5$. In contrast, as long as we choose our primes and evaluations so that the leading term of the GCD does not vanish (which is detected as a bad prime or evaluation), then the scaling factors can never be zero, so setting one of them to a fixed value will never result in this problem.

One might wonder why the multiple scaling case can be even mildly efficient, as we are constructing a system that ties together all unknowns of the problem through the multipliers. This is in direct contrast to the single scaling case, for which each degree in x_1 has an independent subsystem. The trick is to realize that the resulting system is highly structured, and the structure can be exploited to put the solution expense of the multiple scaling case on the same order as the solution expense of the single scaling case.

Example 32 (Multiple Scaling) *Consider the linear algebra required in the computation of the* GCD *for a problem with the image:*

$$g_f = (a_{32}y^2+a_{31}y+a_{30})x^3+(a_{23}y^3+a_{22}y^2+a_{21}y+a_{20})x^2+(a_{13}y^3+a_{11}y+a_{10})x+(a_{01}y^2+a_{00})$$

We find that we will have a sufficient number of equations to solve for all unknowns if we compute 4 images, where we note that for this problem the number of required images is exactly the same as for the single scaling case. The resulting Matrix, involving 16 equations for the 15 unknowns (12 image unknowns and 3 unknown scaling factors) has the following structure:

$$\left|\begin{array}{cccccccccccccccc}
c & c & c &&&&&&&&&& \mathbf{c} &&& \\
c & c & c &&&&&&&&&&& c && \\
c & c & c &&&&&&&&&&&& c & \\
c & c & c &&&&&&&&&&&&& c \\
&&& c & c & c & c &&&&&& \mathbf{c} &&& \\
&&& c & c & c & c &&&&&&& c && \\
&&& c & c & c & c &&&&&&&& c & \\
&&& c & c & c & c &&&&&&&&& c \\
&&&&&&& c & c & c &&& \mathbf{c} &&& \\
&&&&&&& c & c & c &&&& c && \\
&&&&&&& c & c & c &&&&& c & \\
&&&&&&& c & c & c &&&&&& c \\
&&&&&&&&&& c & c & \mathbf{c} &&& \\
&&&&&&&&&& c & c && c && \\
&&&&&&&&&& c & c &&& c & \\
&&&&&&&&&& c & c &&&& c
\end{array}\right| \tag{6.1}$$

Where the equations are ordered by decreasing degree in x then by image number, and the unknowns are ordered in the same order as the image, with the scaling factors on the right, and the c's denote (possibly) non-zero entries, all other entries being strictly zero. Note also that the first column of multipliers (the bold c's) have fixed values.

The solution of the above system can be easily computed by solution of a number of smaller subsystems corresponding to the rectangular blocks of non-zero entries augmented with the multiplier columns. Once the subsystems are upper triangular, remaining rows, only involving the multipliers, can be used to compute the multiplier values, which can then be back-substituted into the subsystems to obtain the image coefficients.

Note that for the single scaling case, the structure is similar to the above, except that the scaling columns would be replaced by a single column of constants. For that case, all blocks are clearly independent, making the efficient solution of the problem obvious.

This approach does offer a solution to our problem, but it also introduces another difficulty, as illustrated by the following example:

Example 33 *Consider the computation of the bivariate* GCD $(y+2)x^3+12y^2+24y$. *Using a call to the Brown algorithm, we can obtain our first image for* $p_1 = 13$ *as*

$$g_1 = x^3y + 2x^3 + 12y^2 + 11y \pmod{13},$$

and expected form of the GCD $g_f = \alpha x^3 y + \beta x^3 + \gamma y^2 + \sigma y$.

Just as for the prior example, we know we need at least three univariate GCD *images for the new prime* $p_2 = 17$. *We choose* $y = 1, 2, 3$ *and obtain the* GCD*s* x^3+12, x^3+7, *and* x^3+2 *respectively. We form the modified system as follows:*

$$\begin{aligned}
\alpha x^3 + \beta x^3 + \gamma + \sigma &= x^3 + 12,\\
2\alpha x^3 + \beta x^3 + 4\gamma + 2\sigma &= m_2(x^3+7),\\
3\alpha x^3 + \beta x^3 + 9\gamma + 3\sigma &= m_3(x^3+2),
\end{aligned}$$

In attempting to solve this system, we find that it is under-determined, so we add a new evaluation point, $y = 4$, *obtaining a* GCD *of* $x^3 + 14$, *and the new equation*

$$4\alpha x^3 + \beta x^3 + 16\gamma + 4\sigma = m_4(x^3+14).$$

In attempting to solve the new system of equations, we find that it is still under-determined. In fact, we could continue to choose new evaluation points for y *until we run out of points* $\pmod{p}$, *and the linear system would remain under-determined.*

This is the case for any chosen prime and set of evaluations, so the algorithm fails to find the GCD *for this problem.*

What is not necessarily obvious from the above example is the cause of the failure, which is the presence of a content in the GCD with respect to the main variable x, namely $y+2$. The existence of a content in the evaluated variables can be immediately recognized as a source of problems for the algorithm, as it can be absorbed into the multipliers on the right hand side of the formed system for the equations with multipliers, and otherwise absorbed into each of the unknowns (for the first equation where the multiplier is set to 1).

This is exactly what occurs in the above example, as the content in y is absorbed into the unknowns, so only the relative ratio between terms can be computed, and we can never obtain a solution for the coefficients in our candidate form.

It is clear that content can cause a problem for the algorithm, so just as for many other sparse algorithms (*EEZ-GCD* for example), the content *must* be removed before the algorithm is called.

Here we note that the content removal problem has been dealt with in a very elegant fashion in Kaltofen and Trager [33], wherein a transformation of the form $x_i \mapsto x_1 + \alpha x_i$ for random α and $2 \le i \le n$ is applied to the input polynomials to force the content to become part of the GCD in the main variable. We also note that efficient use of this technique requires treatment of the input polynomials as black boxes, as otherwise the polynomials become dense, and as we are focusing on direct modular techniques, where such a technique would result in excessive expression swell, we discuss this approach no further here.

One may ask *is it sufficient to simply remove the* GCD *content before calling the algorithm to take care of this problem?* Unfortunately, the answer is no, as certain choices of primes and evaluation points can cause an *unlucky content* to appear in the GCD.

Definition 16 (Unlucky Content) *Given $a \in \mathbf{Z}[x_1, ..., x_n]$ with $cont_{x_1}(a) = 1$, a prime p is said to introduce an unlucky content if $cont_{x_1}(a \bmod p) \neq 1$. Similarly for $a \in \mathbf{Z}[x_1, ..., x_n]$ with $cont_{x_1}(a) = 1$ an evaluation $x_i = \alpha_i$ is said to introduce an unlucky content if $cont_{x_1}(a \bmod \langle x_i - \alpha_i \rangle) \neq 1$.*

Consider, for example, computation of the GCD $x(y+1)+y+14$. If we choose $p = 13$ the GCD has a content of $y+1$, while for any other prime, or over $\mathbf{Z}[x, y]$, no content is present. This is a significant consideration in the design of the algorithm, and must be carefully checked. An argument can be made as to the similarity between the probability of obtaining an unlucky content, and the probability of selecting an unlucky prime or evaluation, and the fact that these are both equally uncommon, so a similar approach will be used in the design of the algorithm for handling unlucky contents. Specifically we will design the algorithm so these problems are not detected in advance, but rather through their effect, so that detection of this problem does not become a bottleneck of the algorithm.

We now present the *LINZIP M* algorithm, which computes the GCD in $\mathbf{Z}[\mathbf{x}]$ from a number of images in $\mathbf{Z}_p[\mathbf{x}]$, and the *LINZIP P* algorithm, which computes the GCD in $\mathbf{Z}_p[x_1, ..., x_n]$ from a number of images in $\mathbf{Z}_p[x_1, ..., x_{n-1}]$. We emphasize that any content of the GCD with respect to x_1 must be removed before the initial call to the *LINZIP M* algorithm, though it is not required that the cofactor contents be removed.

Algorithm 21 (LINZIP M)

Input: *$a, b \in \mathbf{Z}[\mathbf{x}]$ such that $\textsc{gcd}(\text{cont}_{x_1}(a), \text{cont}_{x_1}(b)) = 1$ and degree bounds $d_{\mathbf{x}}$ on the* GCD *in* $\mathbf{x}$

Output: $g = \textsc{gcd}(a, b) \in \mathbf{Z}[\mathbf{x}]$

1 *Compute the scaling factor $\gamma = \textsc{gcd}(\text{lc}_{\mathbf{x}}(a), \text{lc}_{\mathbf{x}}(b)) \in \mathbf{Z}$*

2 *Choose a random prime p such that $\gamma_p = \gamma \bmod p \neq 0$, and set $a_p = a \bmod p$, $b_p = b \bmod p$, then compute from these a modular* GCD *image $g_p \in \mathbf{Z}_p[x_1, ..., x_n]$ with a call to LINZIP P. If the algorithm returns* **Fail**, *repeat, otherwise set $d_{x_1} = \deg_{x_1}(g_p)$ and continue.*

3 *Assume that g_p has no missing terms, and that the prime is not unlucky. We call the assumed form g_f. There are two cases here.*

3.1 *If there exists a coefficient of x_1 in g_f that is a monomial, then we will normalize by setting the integer coefficient of that monomial to 1. Count the largest number of terms in any coefficient of x_1 in g_f, calling this n_x.*

3.2 *If there is no such coefficient, then multiple scaling must be used. Compute the minimum number of images needed to determine g_f for a new prime with multiple scaling, calling this n_x.*

4 *Set $g_m = \frac{\gamma_p}{\text{lc}_{\mathbf{x}}(g_p)} \times g_p \bmod p$ and $m = p$.*

5 *Repeat*

5.1 *Choose a new random prime p such that $\gamma_p = \gamma \bmod p \neq 0$, and set $a_p = a \bmod p$, $b_p = b \bmod p$.*

5.2 *Set $S = \emptyset$, $n_i = 0$.*

5.3 *Repeat*

5.3.1 *Choose $\alpha_2, ..., \alpha_n \in \mathbf{Z}_p$ at random such that for $I = \langle x_2 - \alpha_2, ..., x_n - \alpha_n \rangle$ we have $\deg_{x_1}(a_p \bmod I) = \deg_{x_1}(a)$, $\deg_{x_1}(b_p \bmod I) = \deg_{x_1}(b)$, and set $a_1 = a_p \bmod I$, $b_1 = b_p \bmod I$*

5.3.2 *Compute $g_1 = \textsc{gcd}(a_1, b_1)$*

5.3.3 *If* $\deg_{x_1}(g_1) < d_{x_1}$ *then our original image and form* g_f *and degree bounds were unlucky, so set* $d_{x_1} = \deg_{x_1}(g_1)$ *and goto 2.*

5.3.4 *If* $\deg_{x_1}(g_1) > d_{x_1}$ *then our current image is* g_1 *unlucky, so goto 5.3.1, unless the number of failures exceeds* $\min(1, n_i)$*, in which case we assume* p *is unlucky and goto 5.1.*

5.3.5 *Add the equations obtained from equating coefficients of* g_1 *and the evaluation of* $g_f \bmod I$ *to* S*, and set* $n_i = n_i + 1$.

Until $n_i \geq n_x$

5.4 *We may now have a sufficient number of equations in* S *to solve for all unknowns in* $g_f \bmod p$ *so attempt this now, calling the result* g_p.

5.5 *If the system is inconsistent then our original image must be incorrect (missing terms or unlucky), so goto 2.*

5.6 *If the system is under-determined, then record the degrees of freedom, and if this has occurred twice before with the same degrees of freedom then assume that an unlucky content problem was introduced by the current prime* p *so goto 5.1. Otherwise we need more images so goto 5.3.1.*

5.7 *The resulting solution is consistent and determined, so we have a new image* g_p *Set* $g_p = \frac{\gamma_p}{\mathrm{lc}_{\mathbf{x}}(g_p)} \times g_p \bmod p$, $g_m = \mathrm{CRA}([g_p, g_m], [p, m])$, $m = m \times p$.

Until g_m *has stopped changing for one iteration.*

7 *Compute* $g_c = \mathrm{pp}_{\mathbf{x}}(g_m)$ *and check that* $g_c \mid a$ *and* $g_c \mid b$*. If not we need more primes, so goto 5.1.*

8 *Return* g_c.

Algorithm 22 (LINZIP P)

Input: $a, b \in \mathbf{Z}_p[x_1, ..., x_n]$*, a prime* p*, and degree bounds* $d_{\mathbf{x}}$ *on the* GCD *in* $\mathbf{x}$

Output: $g = \mathrm{GCD}(a, b) \in \mathbf{Z}_p[x_1, ..., x_n]$ *or* **Fail**

0 *Check the* GCD *of the inputs for content in* x_n*, if present return* **Fail**.

1 *Compute the scaling factor* $\gamma = \mathrm{GCD}(\mathrm{lc}_{x_1,\ldots,x_{n-1}}(a), \mathrm{lc}_{x_1,\ldots,x_{n-1}}(b)) \in \mathbf{Z}_p[x_n]$

2 *Choose* $v \in \mathbf{Z}_p$ *at random such that* $\deg_{x_1,\ldots,x_{n-1}}(a \bmod \langle x_n - v\rangle) = \deg_{x_1,\ldots,x_{n-1}}(a)$, $\deg_{x_1,\ldots,x_{n-1}}(b \bmod \langle x_n - v\rangle) = \deg_{x_1,\ldots,x_{n-1}}(b)$*, and set* $a_v = a \bmod \langle x_n - v\rangle$, $b_v =$

$b \bmod \langle x_n - v\rangle$, *then compute from these a modular* GCD *image* $g_v \in \mathbf{Z}_p[x_1, ..., x_{n-1}]$ *with a recursive call to LINZIP P* $(n > 2)$ *or via the Euclidean algorithm* $(n = 2)$. *If for* $n > 2$ *the algorithm returns* **Fail** *or for* $n = 2$ *we have* $\deg_{x_1}(g_v) > d_{x_1}$ *then return* **Fail**, *otherwise set* $d_{x_1} = \deg_{x_1}(g_v)$ *and continue.*

3 *Assume that* g_v *has no missing terms, and that the evaluation is not unlucky. We call the assumed form* g_f. *There are two cases here.*

3.1 *If there exists a coefficient of* x_1 *in* g_f *that is a monomial, then we will normalize by setting the integer coefficient of that monomial to 1. Count the largest number of terms in any coefficient of* x_1 *in* g_f, *calling this* n_x.

3.2 *If there is no such coefficient, then multiple scaling must be used. Compute the minimum number of images needed to determine* g_f *for a new evaluation of* x_n *with multiple scaling, calling this* n_x.

4 *Set* $g_{seq} = \frac{\gamma(v)}{\mathrm{lc}_{x_1,...,x_{n-1}}(g_v)} \times g_v \bmod p$ *and* $v_{seq} = v$.

5 *Repeat*

5.1 *Choose a new random* $v \in \mathbf{Z}_p$ *such that* $\deg_{x_1,..,x_{n-1}}(a \bmod \langle x_n - v\rangle)$ $= \deg_{x_1,..,x_{n-1}}(a)$, $\deg_{x_1,..,x_{n-1}}(b \bmod \langle x_n - v\rangle) = \deg_{x_1,..,x_{n-1}}(b)$, *and set* $a_v = a \bmod \langle x_n - v\rangle$, $b_v = b \bmod \langle x_n - v\rangle$.

5.2 *Set* $S = \emptyset$, $n_i = 0$.

5.3 *Repeat*

5.3.1 *Choose* $\alpha_2, ..., \alpha_{n-1} \in \mathbf{Z}_p$ *at random such that for* $I = \langle x_2 - \alpha_2, ..., x_{n-1} - \alpha_{n-1}\rangle$ *we have* $\deg_{x_1}(a_v \bmod I) = \deg_{x_1}(a)$, $\deg_{x_1}(b_v \bmod I) = \deg_{x_1}(b)$, *and set* $a_1 = a_v \bmod I$, $b_1 = b_v \bmod I$

5.3.2 *Compute* $g_1 = \mathrm{GCD}(a_1, b_1)$

5.3.3 *If* $\deg_{x_1}(g_1) < d_{x_1}$ *then our original image and form* g_f *and degree bounds were unlucky, so set* $d_{x_1} = \deg_{x_1}(g_1)$ *and goto 2.*

5.3.4 *If* $\deg_{x_1}(g_1) > d_{x_1}$ *then our current image is* g_1 *unlucky, so goto 5.3.1, unless the number of failures exceeds* $\min(1, n_i)$, *in which case we assume* $x_n = v$ *is unlucky and goto 5.1.*

5.3.5 *Add the equations obtained from equating coefficients of* g_1 *and the evaluation of* $g_f \bmod I$ *to* S, *and set* $n_i = n_i + 1$.

Until $n_i \geq n_x$

5.4 *We should now have a sufficient number of equations in* S *to solve for all unknowns in* $g_f \bmod p$ *so attempt this now, calling the result* g_v.

5.5 *If the system is inconsistent then our original image must be incorrect (missing terms or unlucky), so goto 2.*

5.6 *If the system is under-determined, then record the degrees of freedom, and if this has occurred twice before with the same degrees of freedom then assume the content problem was introduced by the evaluation of* x_n *so goto 5.1. Otherwise we need more images so goto 5.3.1.*

5.7 *The resulting solution is consistent and determined, so we have a new image* g_v

Set $g_{seq} = g_{seq}, \frac{\gamma(v)}{\mathrm{lc}_{x_1,\ldots,x_{n-1}}(g_v)} \times g_v,\ v_{seq} = v_{seq}, v$

Until we have $d_{x_n} + \deg_{x_n}(\gamma) + 1$ *images.*

6 *Reconstruct our candidate* GCD g_c *using Newton interpolation (dense) on* g_{seq},v_{seq}, *then removing content in* x_n.

7 *Loop some fixed number of times*

7.1 *Choose* $\alpha_2, ..., \alpha_n \in \mathbf{Z}_p$ *at random such that* $I = \langle x_2 - \alpha_2, ..., x_n - \alpha_n \rangle$ *is not a bad evaluation point, and set* $a_1 = a \bmod I$, $b_1 = b \bmod I$, *and* $g_v = g_c \bmod I$.

7.2 *Compute* $g_1 = \mathrm{GCD}(a_1, b_1)$ *and if* g_1 *does not divide* g_v *then our original image must be bad, so goto 2.*

8 *Return* g_c.

A number of comments are required for the above algorithms, and a discussion of the correctness and termination of the overall algorithm follows.

1. The degree bounds of the GCD have very specific uses. The degree bound of the GCD in the main variable x_1 is used to detect unlucky primes and evaluations, but only detects those that involve x_1, and we update the degree bound *whenever* we compute a GCD of lower degree in x_1. The degree bounds of the GCD in the non-main variables $x_2, ..., x_n$ are used to compute the number of images needed in the Newton interpolation in step 6 of *LINZIP P*, and are not updated by the algorithm.

2. The number of required images for the multiple scaling case computed in step 3.2 can be the same as the number of required images for the single scaling case computed in step 3.1 (if the GCD is of certain forms), and up to at most 50% higher. The worst case is quite infrequent, and will only occur when there are only two coefficients with respect to the main variable, each having exactly the same number of terms. For three coefficients, this is limited to 33%, for 10 it is limited to 10%, etc. The extra expense of this step can usually be reduced by an intelligent choice of the main variable x_1. The exact formula for the number of images needed depends upon the term counts for a problem. For a problem with coefficients having term counts of $n_1, ..., n_s$ for the coefficients with respect to x_1 and a maximum term count of $n_{\max}$ is given by $\max(n_{\max}, \lceil(\sum_{i=1}^{s} n_i - 1)/(s-1)\rceil)$.

3. The check in step 0 of *LINZIP P* is used to detect an unlucky content in the initial GCD introduced higher up in the recursion by either a prime or evaluation. Note that we can always detect the problem (if not the source of the problem) in this way, as any content in the GCD with respect to x_1 will eventually show up as a univariate content as we evaluate $x_n, x_{n-1}, ...$

4. The check in step 5.6 of either algorithm is intended to check for an unlucky content introduced by the evaluation (*LINZIP P*) or prime (*LINZIP M*) chosen in step 5.1 of both algorithms. Since it is possible that a new random image from step 5.3.1 does not necessarily constrain the form of the GCD (even without the content problem) we check for multiple failures before rejecting the current iteration of loop 5.

5. The *LINZIP P* algorithm has been modified to perform a probabilistic division test in step 7 instead of testing that $g_c \mid a$ and $g_c \mid b$. If a few iterations are performed, then the result is correct with high probability, and the test is substantially less expensive. There is, however, a chance that the test fails to detect an incorrect answer, so the termination division test in *LINZIP M* must be retained to make the algorithm deterministic.

6. A major enhancement could be realized through the use of a highly efficient implementation of Brown's P algorithm once we evaluate down to 2 variables. The reasoning here is that by the time we evaluate down to 2 variables, the problem will very likely be dense in those variables. In this case, the use of the Brown approach could provide an efficiency improvement, and if implemented as a general strategy (i.e. always evaluate down to a bivariate problem instead of a univariate one), would make the use of multiple scaling less likely (as now we are considering coefficients with respect to 2 variables), and would reduce the size of the linear systems to be solved.

7. There is an enhancement to the above algorithm that is not explicitly described, and it is the following: Do not always throw away images when repeating step 2 as the result of an inconsistent system in step 5.5. Instead retain the image, and consider it with respect to the new computed image. If the images differ with respect to relatively few terms, then use them both to determine the expected form for g. Note that this is typically the main failure case for the algorithm (i.e. it has a much higher probability than an unlucky prime/evaluation or an unlucky content). This has the advantage that even for problems where the number of terms is close to the value of the prime, all terms of the image will eventually be found, and in the unlikely event that one of the images chosen for use was unlucky, this will only enlarge the number of terms to be solved for, and will not result in an incorrect solution.

We now discuss the correctness and termination of the algorithm. We need to consider 4 main problems with sparse modular methods, namely bad primes or evaluations, unlucky contents, unlucky primes or evaluations, and missing terms (in an initial image).

The treatment of bad primes and bad evaluations is straightforward, and is handled for the first prime or first evaluation by the vector degree checks in step 2 of the algorithms, handled for the current prime or current evaluation by the vector degree checks in step 5.1 of the algorithms, and handled for the univariate images in step 5.3.1 of the algorithms.

The treatment of the unlucky content problem for the first prime or first evaluation is handled in step 0 of *LINZIP P* by the single variable content check. As in point 3 above we emphasize that this check will always detect the problem at some level of the recursion, specifically the level containing the last variable contained in the unlucky content (as all the other variables in the content have been evaluated, so the content becomes univariate). We note that there is no efficient way to detect which prime or evaluation introduced the unlucky content. It may have been introduced by the prime chosen in *LINZIP M* or any evaluation in prior calls (for x_j with $j > n$) to *LINZIP P* in the recursion. Since this problem is rare, we handle this by rejecting the case all the way back up to the *LINZIP M* algorithm, which results in a completely new prime and set of evaluations. This strategy is fairly efficient, as only evaluations (modular and variable) and other single variable content checks have been performed before a failure is detected at any level of the recursion.

The introduction of an unlucky content by the prime or evaluation chosen in step 5.1 of either algorithm will be handled in the combination of steps 5.4 and 5.6. Note that an unlucky content is only a problem when multiple scaling is in use (as it is the scaling of the image that is the issue). The result is a system with additional degrees of freedom, so this *always* results in an under-determined system. The check in step 5.6 handles this, as eventually we will obtain a solution for all variables but the free ones resulting from the unlucky content, so the degrees of freedom will stabilize, and we will go back to step 5.1 choosing a new prime or evaluation.

The treatment of unlucky primes is less straightforward.

First we consider an unlucky evaluation in step 2 of *LINZIP P* for x_n for which the factor added to the GCD depends upon x_1. If the degree bound d_{x_1} is tight, then this will be detected at a lower level of the recursion by step 2 of *LINZIP P* when $n = 2$. If the degree bound d_{x_1} is not tight, then the GCD computed in that step may be unlucky, but we proceed with the computation. Once we reach loop 5, we begin to choose new evaluation points for x_n, and with high probability we will choose a new point that is not unlucky in step 5.1, the problem will be detected in step 5.3.3, and we will go back to step 2, and compute a new image. In the worst case, all evaluations in step 5.1 may also be unlucky, introducing the same factor to the GCD, and we will proceed to step 6, and reconstruct an incorrect result. Note that if the factor is in fact different, then the equations accumulated in step 5.3.5 will most likely be inconsistent, and this problem will most likely be detected in steps 5.4 and 5.5. Step 7 will again perform checks much like those in step 5.3.3, and will detect this problem with high probability, but if it does not, an invalid result may be returned from *LINZIP P*.

If we continue to choose unlucky evaluations we will eventually return an incorrect image to *LINZIP M*.

This problem (as well the unlucky prime case for step 2 of *LINZIP M*) is handled by the structure of the *LINZIP M* algorithm. Since the steps are essentially the same, the same reasoning follows, and we need the computation to be unlucky through all iterations of loop 5. Now in this case, since the form of the GCD is incorrect, it is unlikely that g_m will stabilize, and we will continue to loop. Note that in the event that g_m does stabilize, the invalid image will not divide a and b, so step 7 will put us back into the loop. Now within that loop, which will not terminate until we have found the GCD, step 5.3.4 will eventually detect this problem, as we must eventually find a prime that is lucky.

So the unlucky case for the initial image is handled when the factor added to the GCD depends upon x_1.

Now consider the case where the unlucky evaluation or prime is chosen in step 2 of either algorithm, and the factor added to the GCD is independent of x_1. In this case, the factor is actually a content with respect to x_1, so this is handled by the same process as the unlucky content problem, specifically it is handled on the way down by step 0 of *LINZIP P*.

Now if an unlucky prime or evaluation occurs in step 5.1 of either algorithm, it will either raise the degree in x_1, in which case it will be detected in step 5.3.4 of either algorithm, or it will be independent of x_1, in which case it is a content. If the content is purely a contribution of the cofactors, then this case will not cause a problem for the algorithm, as it will simply reconstruct the new GCD image without that content present (as a result of the assumed form).

The only type of unlucky evaluation that can occur in step 5.3.1 of either algorithm must raise the degree of the GCD in x_1, so is handled by step 5.3.4.

The final issue to discuss is the case where the initial image is missing terms in either algorithm. When this occurs, the resulting system will likely be inconsistent, so will be detected by step 5.5 with high probability, but this may not be the case. If the problem is not detected in any iteration of loop 5, then an incorrect image will be reconstructed in step 6 of *LINZIP P*. The additional check(s) in step 7 of *LINZIP P* will, with high probability, detect this problem with the new images, but if this also fails, then we return an incorrect image from *LINZIP P*.

Again assuming a sequence of failures to detect this problem, we arrive at *LINZIP M*. Now we will compute new images in *LINZIP M* until g_c divides both a and b, so the problem must eventually be detected.

Note that the missing term case is the most likely failure case, as unlucky primes, unlucky evaluations, and unlucky contents are generally quite unlikely. The probability of choosing a prime or evaluation that causes a term to vanish is $\mathcal{O}(\frac{t}{p})$, where t is the number of terms in the polynomial, and p is the prime. So for sparse problems, and a sufficiently large prime, the chance of a bad evaluation is small, and the initial image is likely not missing terms. Conversely if the prime is small, or the problem is large and dense (having many terms), it is possible that the algorithm may never terminate. The algorithm, however, is designed for sparse problems, so this is not an issue as long as primes of a suitable size are used. Note that the enhancement described as item 7 earlier can be used for this case.

6.7 The *EEZ-GCD* Sparse GCD Algorithm

The *EEZ-GCD* algorithm is also based on modular computation, though it takes a distinctly different approach than the methods described thus far. To understand the method, it is necessary to lay some foundations for the p-adic and I-adic lifting methods.

The information in this section is provided primarily to give the flavor of the method, and the details for efficient implementation of the algorithms are left for the asymptotic analyses.

6.7.1 Lifting Methods

The basic approach used in these lifting methods is to pose the problem in the form of an equation, $f(u) = 0$, where $u \in \mathbf{Z}[x_1, ..., x_n]$, and obtain a solution u_1 in a simpler domain $\mathbf{Z}_p[x_1]$ such that u_1 satisfies $f(u_1) = 0 \bmod \langle p, x_2 - \alpha_2, ..., x_n - \alpha_n \rangle$. The solution u_1 is then *lift*ed to the solution $u \in \mathbf{Z}[x_1, ..., x_n]$.

A generic description of a lifting process (that parallels the one in use by *EEZ-GCD*) is presented in the following diagram:

There are some significant differences between this approach and the modular approaches described earlier.

1 This approach performs only one computation of the actual problem in the simpler domain, $\mathbf{Z}_p[x_1]$, then uses that solution to reconstruct the solution of the full problem.

2 Only one prime and set of evaluations is used for the entire process.

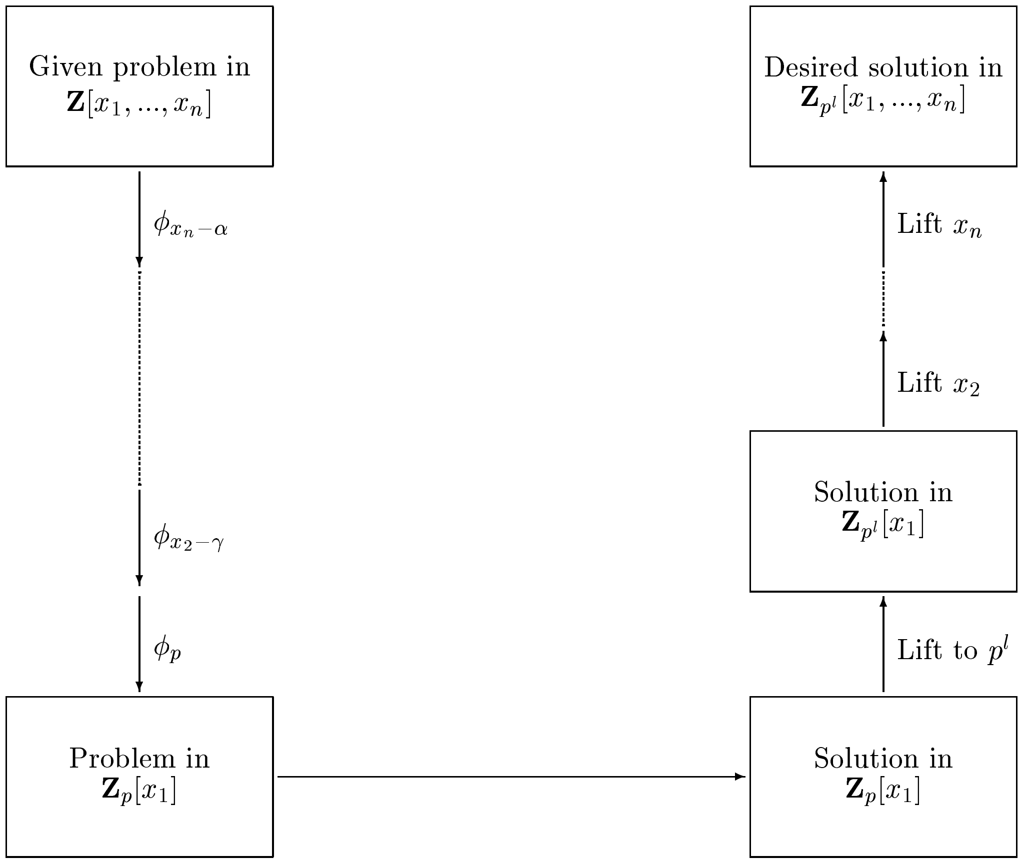

Figure 6.2: Homomorphism Diagram for Lifting Approach

3 The integer coefficient reconstruction is performed at the lowest level, *before* reconstruction of the dependence of the variables $x_2, ..., x_n$.

Note that l is chosen to be sufficiently large so that the solution in $\mathbf{Z}_{p^l}$ is the same as the solution in $\mathbf{Z}$. The lifting process for the variables can proceed one variable at a time, or all variables simultaneously.

6.7.2 Application to GCD (p-adic lifting)

The lifting approach utilized to compute GCD is known as Hensel lifting, and is slightly different than the lifting process described in the prior subsection.

The equation to be solved depends upon two expressions, u, w, and is of the form $F -$

$uw = 0$, where $F \in \mathbf{Z}[x_1, ..., x_n]$ is one of the polynomials for which we are computing the GCD, and u, w are the GCD and cofactor of that polynomial.

With this in mind, the first step, finding the solution in $\mathbf{Z}_{p^l}[x_1]$ (effectively in $\mathbf{Z}[x_1]$), is simply the process of evaluating the polynomial inputs at $\langle p, x_2 - \alpha_2, ..., x_n - \alpha_n \rangle$, computing the GCD u_1, then selecting F to be one of the polynomials, and w_1 to be the corresponding cofactor. We then lift the solution to one with coefficients in $\mathbf{Z}$ by applying *univariate Hensel lifting* which we give a basic description of below.

Note we are free to choose either of the polynomial inputs as the one used in the reconstruction, which allows us some flexibility that can be leveraged to increase the efficiency of the algorithm.

To see how the *univariate Hensel lifting* algorithm works, we re-write u, w, F for the problem in $\mathbf{Z}[x_1]$ in p-adic form (base p) as follows

$$\begin{aligned} u &= u^0 + u^1 p + u^2 p^2 + ..., \\ w &= w^0 + w^1 p + w^2 p^2 + ..., \\ F &= F^0 + F^1 p + F^2 p^2 + ..., \end{aligned}$$

where $u^i, w^i, F^i \in \mathbf{Z}_p[x_1]$.

Now consider the equation we want to solve, $F - uw = 0$. Application of the re-written forms above gives

$$\begin{aligned} F - uw &= (F^0 + F^1 p + ...) - (u^0 + u^1 p + ...)(w^0 + w^1 p + ...) \\ &= (F^0 + u^0 w^0) + (F^1 + u^0 w^1 + w^0 u^1) p + ..., \end{aligned}$$

from which it is obvious (by considering the equation $\bmod p$) that the $\mathcal{O}(p^0)$ term is satisfied immediately by setting $u^0 = u_1$ and $w^0 = w_1$.

We then want to improve the approximation by computing higher order terms, namely $u^1, w^1, u^2, w^2, ...$, requiring that

$$F - (u^0 + u^1 p + ... + u^i p^i)(w^0 + w^1 p + ... + w^i p^i) \equiv 0 \pmod{p^{i+1}}.$$

We will label these improved approximations u_i, defining these as $u_{i+1} = u_i + u^i p^i$, with a similar definition for w.

We will assume the use of *linear* lifting to obtain the higher order terms, and not *quadratic* lifting. Using *linear* lifting we obtain a single higher order term (such as u^1,u^2,...) in each step. Use of *quadratic* lifting allows doubling the number of known higher order terms in each step, but the computations in each step are more expensive, and the overall cost is shown to be asymptotically equivalent by Miola and Yun [41], assuming that classical integer multiplication and division are used.

We assume that we have computed u_i, w_i and show how to obtain u^i, w^i, thus obtaining the approximations u_{i+1}, w_{i+1}.

By definition, we have:

$$F - u_i w_i \equiv 0 \pmod{p^i}.$$

Now consider the exact solutions u,w:

$$\begin{aligned} u &= u^0 + u^1 p + \ldots + u^{i-1} p^{i-1} + u^i p^i + \mathcal{O}(p^{i+1}) \\ &= u_i + u^i p^i + \mathcal{O}(p^{i+1}), \\ w &= w_i + w^i p^i + \mathcal{O}(p^{i+1}). \end{aligned}$$

Substitution into our equation $F - uw$, computing $\bmod\, p^{i+1}$ gives

$$\begin{aligned} F - (u_i + u^i p^i + \mathcal{O}(p^{i+1}))(w_i + w^i p^i + \mathcal{O}(p^{i+1})) \bmod p^{i+1} &= 0 \\ F - u_i w_i - (u^i w_i + u_i w^i) p^i + \mathcal{O}(p^{i+1}) \bmod p^{i+1} &= 0 \\ \frac{F - u_i w_i}{p^i} - (u^i w_i + u_i w^i) \bmod p &= 0 \\ \frac{F - u_i w_i}{p^i} - (u^i w_1 + u_1 w^i) \bmod p &= 0, \end{aligned}$$

so our solution updates u^i,w^i can be obtained from the modular diophantine equation

$$u^i w_1 + u_1 w^i = \frac{F - u_i w_i}{p^i} \bmod p, \tag{6.2}$$

though the actual implementation uses a more efficient incremental approach for the computation of the right-hand-side of (6.2).

The solution of this equation can be easily accomplished via the extended Euclidean algorithm (EEA), as $\mathbf{Z}_p[x_1]$ is a Euclidean domain.

So we can obtain the initial approximation, and can update the approximation to one that is satisfied $\bmod\, p^l$. Once the solution has been improved so that $F - u_i w_i = 0$, we have obtained the solution in $\mathbf{Z}[x_1]$.

There are a few additional details for the above algorithm, and these are listed here:

1 The use of the EEA to compute the solutions for equation 6.2 requires that there is no common GCD between u_1 and w_1. If this is not the case, then either the prime was a bad choice, or there is actually a GCD between u and w, in which case the other polynomial for which we are computing the GCD could be used instead, or some linear combination of the two polynomials.

2 The presented algorithm only provides a solution for monic inputs, since the output must be monic. The modification for non-monic problems can be easily devised by an appropriate scaling of F by its leading coefficient, and reconstruction of u,w each scaled by the leading coefficient of the original F. The true GCD and cofactor can then be recovered by removal of the content of the reconstructed $u, w \in \mathbf{Z}_{p^l}[x_1]$ viewed as polynomials in $\mathbf{Z}[x_1]$.

3 There are considerations in the selection of the prime p, and cases where the choice of the prime can be bad (non-monic case) or unlucky. For example, an unlucky choice of p could cause the GCD to be too high a degree, then there would exist no u, w corresponding to the u_0, w_0 inputs. This must be detected and handled, and the algorithm must indicate failure for these cases.

4 Solutions for equation 6.2 only require a single application of the EEA, as once we have the solution for $u_1\sigma + w_1\tau = 1 \pmod p$, all solutions to 6.2 can be obtained through polynomial multiplication and division.

These considerations are discussed in detail in Geddes [19], along with a pseudocode description of the algorithm.

6.7.3 Application to GCD (I-adic lifting)

The prior subsection only described reconstruction from $\mathbf{Z}_p[x_1]$ to $\mathbf{Z}_{p^l}[x_1]$, and in this subsection we will complete the description by reconstructing the variable dependencies from $\mathbf{Z}_{p^l}[x_1]$ to $\mathbf{Z}_{p^l}[x_1, ..., x_n]$.

The *EEZ-GCD* algorithm reconstructs the dependencies one variable at a time (though all dependencies can be reconstructed simultaneously [19]), so the solution is lifted from

$\mathbf{Z}_{p^l}[x_1]$ to $\mathbf{Z}_{p^l}[x_1, x_2]$, then from $\mathbf{Z}_{p^l}[x_1, x_2]$ to $\mathbf{Z}_{p^l}[x_1, x_2, x_3]$, and so on, until we have the required solution.

The concepts behind the process are identical to those of the univariate case, but with the solution written I-adically instead of p-adically.

So we have the solution $F - uw = 0 \bmod I$, where $I = \langle x_2 - \alpha_2, ..., x_n - \alpha_n \rangle$ is the ideal used to compute the original GCD u_1.

We then write our solutions $u, w \in \mathbf{Z}_{p^l}[x_1, x_2]$ (in the form of a Taylor series) as

$$\begin{aligned} u &= u^0 + u^1(x_2 - \alpha_2) + u^2(x_2 - \alpha_2)^2 + ..., \\ w &= w^0 + w^1(x_2 - \alpha_2) + w^2(x_2 - \alpha_2)^2 + ..., \end{aligned}$$

where $u^i, w^i \in \mathbf{Z}_p[x_1]$. Once again, u^0, w^0 are known, and we need to compute the higher order approximations.

In a similar manner as for the univariate case, we easily obtain the solution as

$$u^i w_1 + u_1 w^i = \frac{F - u_i w_i}{(x_2 - \alpha_2)^i} \bmod \langle p^l, x_2 - \alpha_2 \rangle. \tag{6.3}$$

At this point the similarities with the univariate case end. Equation (6.3) is again a diophantine equation, but the underlying domain, $\mathbf{Z}_p[x_2 - \alpha_2][x_1]$ is no longer Euclidean, so we cannot directly obtain the update terms u^i, w^i via the EEA. A more involved process must be used to solve this problem, and the algorithm for computing solutions of multivariate diophantine equations is presented in [19], and will also be presented in a later section for the purpose of complexity analysis.

For this approach there are also a number of additional considerations:

1. For efficiency one would like to choose the simplest ideal, namely $\langle x_2, ..., x_n \rangle$, as then the reconstruction for a variable would only need to reconstruct the coefficients that are not identically zero. For example, when reconstructing the dependence of x_2 as described above, if $\alpha_2 = 0$, then if u, w were sparse, many of the diophantine equations to be solved would have zero right-hand-sides, which naturally have the solution $u^i = w^i = 0$, so no multivariate diophantine equation would need to be solved for these. Unfortunately, this is not always possible, as the choice of $x_2 = 0$ can cause a number of problems. For sparse polynomials, setting one of the variables to zero could cause the leading coefficient to vanish (bad evaluation), it could result in a GCD between the actual GCD and the cofactor (bad zero), or it could result in a GCD between the cofactors (unlucky evaluation). These cases are very common, and often non-zero points must be chosen. For a non-zero evaluation point, *every* coefficient must be constructed up to the degree of the variable being reconstructed.

2. The leading coefficient problem becomes more difficult here, as if the F has a leading coefficient that depends upon $x_2, ..., x_n$, then it may be necessary to factor that coefficient (a potentially expensive process), and properly split the factors between the GCD and the cofactor.

These issues are discussed the paper of Wang [71] that introduces the *EEZ-GCD* algorithm.

6.8 General Asymptotic Comparison

It is not possible to provide a rigorous and exhaustive asymptotic work estimate for the standard (and new) GCD algorithms, as they are heavily dependent upon the specific instance of the problem being computed, and the complexity of the underlying arithmetic in $\mathbf{Z}$ and $\mathbf{Z}_p[x]$. Certain algorithms perform better than others depending upon certain aspects of the problem (for example, the form of the leading coefficient of the GCD, whether the problem has content with respect to the main variable, the structure of the non-zero terms, etc.).

In this section, we provide asymptotic estimates for the most *common* form of problems, for both sparse and dense cases, and provide some analysis for how these may change under special circumstances.

We use the following simplification assumptions throughout the analysis.

1. We ignore the cost of manipulating the exponent vectors when multiplying two monomials. So multiplication of two monomials with coefficients of 1 is $\mathcal{O}(1)$.

2. We consider arithmetic operations in $\mathbf{Z}_p$, where p is a small (machine sized) prime, to have $\mathcal{O}(1)$ expense.

3. We assume the use of classical integer and polynomial arithmetic. So for two integers in $\mathbf{Z}$ with C_1, C_2 digits respectively, addition and subtraction have an expense of $\mathcal{O}(\max(C_1, C_2))$, multiplication has an expense of $\mathcal{O}(C_1 C_2)$, and division of a $C_1 + C_2$ digit number by a C_1 digit number (with a C_2 digit result) has an expense of $\mathcal{O}(C_1 C_2)$. Similarly, for two dense polynomials in $\mathbf{Z}_p[x]$ with degrees N_1, N_2, addition and subtraction have an expense of $\mathcal{O}(\max(N_1, N_2))$ and multiplication and division (as defined for the integer case) have an expense of $\mathcal{O}(N_1 N_2)$.

4. We ignore the effect of addition on the coefficient size when computing the product of two polynomials. So multiplication of two polynomials with coefficient sizes of C_1 and C_2 will result in a polynomial with coefficient size $C_1 + C_2$. Note that a standard exception for this assumption is to consider the product of two fully dense polynomials of degree N in d variables with all coefficients being 1. In this case the product has a maximal coefficient of $(N+1)^d$ (the number of terms in one polynomial). Considering that we are working with machine-sized integers (32 bits), the number of terms needed to add one word (digit) to the size of the maximal coefficient in a product operation is over 4 billion. So with this in mind we will ignore this effect.

5. We assume that the inputs are free of content with respect to the main variable with the following justification. For the dense algorithms (*Brown* and *DIVBRO*), the existence of a content should affect both equally, and unnecessarily complicates the analysis. For the sparse algorithms (*EEZGCD* and *LINZIP*), the inputs must be content free with respect to the main variable for the algorithms to function correctly, and the cost of computing and removing the content would again be equal.

With these assumptions it is straightforward to show that the cost of Chinese remaindering for a C digit coefficient is $\mathcal{O}(C^2)$, and similarly the cost of interpolation of values in $\mathbf{Z}_p$ to obtain a degree N polynomial in $\mathbf{Z}_p[x]$ is $\mathcal{O}(N^2)$. The cost of the GCD and EEA algorithms for two polynomials $\in \mathbf{Z}_p[x]$ of degree N_1, N_2 can be shown to be $\mathcal{O}(N_1 N_2)$.

We consider the GCD problem $g = \text{GCD}(a, b)$ with cofactors $\bar{a}, \bar{b}$. We use the symbols N, T, and C to represent the maximum degree (in any single variable), the number of terms, and the length of the coefficients respectively, using subscripts to denote which of g,$\bar{a}$,$\bar{b}$ they correspond to. So for example, N_g represents the maximum degree of g, and $T_{\bar{b}}$ represents the number of terms in $\bar{b}$. We write these symbols with multiple subscripts to denote the maximum over those quantities. So for example, $N_{g\bar{a}\bar{b}} = \max(N_g, N_{\bar{a}}, N_{\bar{b}})$. We let n represent the number of variables in the problem, and use $\mathbf{Z}[\mathbf{x}]$ to describe $\mathbf{Z}[x_1, ..., x_n]$, the domain of our problem.

6.8.1 Asymptotic complexity of sub-algorithms

In this section we will analyze a number of sub-algorithms that are used in the GCD algorithms we describe later.

Degree bounds

Degree bounds are needed for the new algorithms, namely *DIVBRO* and *LINZIP*, but these bounds may be used to make other algorithms more efficient. Degree bounds would only be useful if the expense of computing them is smaller than the GCD computations that use them. We discuss an efficient approach to compute these bounds now.

As discussed previously, a GCD computed with respect to a prime or evaluation that is not bad results in at worst a polynomial multiple of the true GCD g with respect to that prime or evaluation. This is the core of the approach: We perform the GCD computation mod a prime, evaluating out all but a single variable, and the resulting GCD will have degree *no lower* than the degree of g in that variable. In other words, this computation provides a bound on the degree of g in that variable. We would like to emphasize here that if the prime and random evaluations are not bad, and the prime is not trivially small, the computed bound will be exact with high probability.

To reduce expense, we perform the bound computation for all variables with respect to a single prime, so the expense of the initial mod is given by $\mathcal{O}((C_g + C_{\bar{a}})(N_g + N_{\bar{a}})^n + (C_g + C_{\bar{b}})(N_g + N_{\bar{b}})^n) = \mathcal{O}(C_{g\bar{a}} {N_{g\bar{a}}}^n + C_{g\bar{b}} {N_{g\bar{b}}}^n)$ for the dense case, and $\mathcal{O}((C_g + C_{\bar{a}}) T_g T_{\bar{a}} + (C_g + C_{\bar{b}}) T_g T_{\bar{b}}) = \mathcal{O}(C_{g\bar{a}} T_g T_{\bar{a}} + C_{g\bar{b}} T_g T_{\bar{b}})$ for the sparse case.

Now we could proceed by direct evaluation of all but a single variable in turn, computing each GCD as we go, but we can improve on this. We evaluate half the variables twice, splitting into two problems that have half the variables of the initial problem. Continuing in this manner, we need to take $\log_2(n)$ steps to obtain all univariate pairs needed to compute the bounds.

For the dense case, the first evaluation with respect to half the variables has a cost of

$\frac{n}{2}\mathcal{O}({N_{g\bar{a}}}^n + {N_{g\bar{b}}}^n)$, which we perform twice, for a total cost of $\mathcal{O}(n({N_{g\bar{a}}}^n + {N_{g\bar{b}}}^n))$. For the next set of evaluations the polynomials are now dense in only $\frac{n}{2}$ variables, which significantly lowers the costs. It is easily shown that the expense of the 4 evaluations required at this level is $\mathcal{O}(n({N_{g\bar{a}}}^{\frac{n}{2}} + {N_{g\bar{b}}}^{\frac{n}{2}}))$. If we then proceed until there is only one variable left, we obtain a sum, but from the form of that sum it is clear that all remaining terms in the sum are asymptotically smaller than the first term (corresponding to the first pair of evaluations), so the total expense of the evaluations in the dense case is $\mathcal{O}(n({N_{g\bar{a}}}^n + {N_{g\bar{b}}}^n))$. We note that in this case the total cost of computing the powers of all variables is $n\mathcal{O}(\max(N_{g\bar{a}}, N_{g\bar{b}}))$, which can be ignored, as it is asymptotically insignificant.

For the sparse case, the first evaluation with respect to half the variables has a cost of $\frac{n}{2}\mathcal{O}(T_g T_{\bar{a}\bar{b}}) + \frac{n}{2}\mathcal{O}(N_{g\bar{a}\bar{b}})$, where the second term corresponds to the expense of computing the powers of the variables to be evaluated. We perform this computation twice, for a total cost of $\mathcal{O}(nT_g T_{\bar{a}\bar{b}} + nN_{g\bar{a}\bar{b}})$. For the next set of evaluations we assume we have stored the computed powers of the variables to be evaluated. There will be at most the same number of terms present, and we require 4 evaluations with respect to $\frac{n}{4}$ variables for two polynomials in $\frac{n}{2}$ variables with a total cost of $\mathcal{O}(nT_g T_{\bar{a}\bar{b}})$, which is the same as the first term of the expense at the first evaluation level. This pattern continues, and we obtain a sum, which when evaluated gives the expense as $\mathcal{O}(n\log_2(n)T_g T_{\bar{a}\bar{b}} + nN_{g\bar{a}\bar{b}})$.

Now given that the expense in computing the n GCDs in $\mathbf{Z}_p[x]$ is $\mathcal{O}(nN_{g\bar{a}}N_{g\bar{b}})$ for both cases, we obtain the estimates for the degree bound computation in Table 6.1.

Dense	$\mathcal{O}(C_{g\bar{a}}{N_{g\bar{a}}}^n + C_{g\bar{b}}{N_{g\bar{b}}}^n + n({N_{g\bar{a}}}^n + {N_{g\bar{b}}}^n) + nN_{g\bar{a}}N_{g\bar{b}})$
Sparse	$\mathcal{O}(C_{g\bar{a}}T_g T_{\bar{a}} + C_{g\bar{b}}T_g T_{\bar{b}} + n\log_2(n)T_g T_{\bar{a}\bar{b}} + nN_{g\bar{a}}N_{g\bar{b}})$

Table 6.1: GCD Degree Bound Computation Expense

In addition to providing degree bounds, this method also allows detection of trivial GCDs with high probability, and also allows for detection of variables that are not present in the GCD, allowing a simplification of the problem (see the comments at the end of the next section).

Univariate content computation in $\mathbf{Z}_p[x]$

Most of the algorithms have an early step where it is necessary to compute a univariate content; i.e. the content of the GCD in the current variable.

The direct method of computation is quite expensive, as it involves a GCD computation for every coefficient in the current variable. For example, in the dense case with n variables, we have $\mathcal{O}(N^n)$ terms, and $\mathcal{O}(N^{n-1})$ coefficients that are degree N in the current variable. The expense of the direct method is easily seen to be $\mathcal{O}(N^{n+1})$ for the dense case, and $\mathcal{O}(TN^2)$ for the sparse case.

There is a more efficient way to compute this, with high probability, in only a single GCD computation. The approach is simply to take the one smallest coefficient in x_n, and a random linear combination of all other coefficients, and compute the GCD of these two polynomials in x_n. Using this approach one needs to take care that the random linear combination is not bad (i.e. no degree drop in x_n), and if this is so, the computed GCD will be at worst a polynomial multiple of the content (in the unlucky case). To reduce this possibility, another combination could be obtained, and the GCD could be computed again. This could be repeated some fixed number of times, reducing the probability of an unlucky content computation to nearly zero.

The effect of this approach is most dramatic for the cases where the content is small, and as this is a probabilistic result, the removal of the content (through division) makes it deterministic. We do not include the content removal (division) cost here, but do consider it in the individual algorithms.

So the cost of performing the linear combination is $\mathcal{O}((N_g + N_{\bar{a}})^n + (N_g + N_{\bar{b}})^n) = \mathcal{O}({N_{g\bar{a}}}^n + {N_{g\bar{b}}}^n)$ in the dense case, and $\mathcal{O}(T_g T_{\bar{a}} + T_g T_{\bar{b}}) = \mathcal{O}(T_g T_{\bar{a}\bar{b}})$ in the sparse case, and cost of the GCD computation is $\mathcal{O}((N_g + N_{\bar{a}})(N_g + N_{\bar{b}})) = \mathcal{O}(N_{g\bar{a}} N_{g\bar{b}})$ in both cases.

We also note that a similar approach can be used for the computation of the integer content in the Brown *M* and *LINZIP M* algorithms, so we simply provide the analogous results without detail. Note that we *do* need to apply our assumption that the coefficients of a, b are $\mathcal{O}(C_g + C_{\bar{a}}), \mathcal{O}(C_g + C_{\bar{b}})$ respectively to obtain the results below.

The total asymptotic cost for a single univariate content computation and for a single integer content computation for a, b is given in Table 6.2.

Case	Direct Method	Improved Method
Dense Univariate	$\mathcal{O}((N_{g\bar{a}}{}^{n-1} + N_{g\bar{b}}{}^{n-1})N_{g\bar{a}}N_{g\bar{b}})$	$\mathcal{O}(N_{g\bar{a}}{}^{n} + N_{g\bar{b}}{}^{n} + N_{g\bar{a}}N_{g\bar{b}})$
Sparse Univariate	$\mathcal{O}(T_g T_{\bar{a}\bar{b}} N_{g\bar{a}} N_{g\bar{b}})$	$\mathcal{O}(T_g T_{\bar{a}\bar{b}} + N_{g\bar{a}} N_{g\bar{b}})$
Dense Integer	$\mathcal{O}((N_{g\bar{a}}{}^{n} + N_{g\bar{b}}{}^{n})C_{g\bar{a}}C_{g\bar{b}})$	$\mathcal{O}(C_{g\bar{a}}N_{g\bar{a}}{}^{n} + C_{g\bar{b}}N_{g\bar{b}}{}^{n} + C_{g\bar{a}}C_{g\bar{b}})$
Sparse Integer	$\mathcal{O}(T_g T_{\bar{a}\bar{b}} C_{g\bar{a}} C_{g\bar{b}})$	$\mathcal{O}(C_{g\bar{a}} T_g T_{\bar{a}} + C_{g\bar{b}} T_g T_{\bar{b}} + C_{g\bar{a}} C_{g\bar{b}})$

Table 6.2: GCD Content Computation Expense

We will assume that this approach is used whenever an integer or univariate content needs to be computed.

Now in addition to this approach being used to compute contents, a similar approach can be used in combination with the degree bounds in the prior section to simplify the problem. Much as we did with the content computation, we can take a random linear combination of the coefficients of the inputs a, b with respect to all variables that are *not* in the GCD. With high probability we will have reduced the number of variables in the problem to only those that are in the GCD, and will still obtain the same result in a computation with fewer variables.

Multivariate content computation

For the sparse algorithms, namely the *EEZ-GCD* and *LINZIP* algorithms, the input polynomials must have no content with respect to the main variable x_1 to succeed. This condition needs to be enforced prior to calling the algorithm, and may in some cases be used to choose a better main variable (i.e. a main variable that is present in all factors of the GCD, hence no content problem).

The straightforward approach here is unthinkable, namely performing at least one, and possibly several GCD computations in one fewer variable, as the expense of this approach will likely be on the order of the expense of the GCD computation itself. Fortunately we can adapt the approach used for computing the univariate contents of the GCD to this problem.

Once a main variable is chosen, we can select the smallest coefficient in that variable, over both a and b, take a random linear combination of the remaining coefficients (making sure the combination is not bad), and compute the GCD of these expressions. With high probability this will compute the content of the problem in a single GCD in one fewer variable.

We will not analyze this in detail, as we will primarily be comparing the efficiency of the sparse algorithms with each other, and both have this additional cost associated with them, so in the analysis we assume $\mathrm{cont}_{x_1}(a, b) = 1$.

Polynomial division

Here we consider the expense of polynomial division of the inputs a, b by the GCD g in $\mathbf{Z}[\mathbf{x}]$. This is required as a correctness check at the end of all the presented algorithms except Brown's algorithm.

We consider performing the division of the expanded polynomials a, b by the expanded GCD g. We perform the division in much the same way as a Gröbner basis reduction with respect to a lexicographic ordering, starting with the leading term of a, b. We note that a great deal of care must be taken in implementation of this algorithm to prevent the term comparisons from becoming a significant factor.

For the dense case, the division of a by g will require as many division steps as terms present in $\bar{a}$, $\mathcal{O}({N_{\bar{a}}}^n)$, with an expense of $\mathcal{O}(C_g C_{\bar{a}} {N_g}^n)$ per step. This makes the overall expense of the division $\mathcal{O}(C_g C_{\bar{a}} {N_g}^n {N_{\bar{a}}}^n)$, basically the product of the sizes of the quotient and divisor. Similarly the division of b by g will have overall expense $\mathcal{O}(C_g C_{\bar{b}} {N_g}^n {N_{\bar{b}}}^n)$.

For the sparse case a similar approach can be used. Division of a by g will require $\mathcal{O}(T_{\bar{a}})$ steps with an expense of $\mathcal{O}(C_g C_{\bar{a}} T_g)$ per step, giving an overall expense of $\mathcal{O}(C_g C_{\bar{a}} T_g T_{\bar{a}})$. Similarly the division of b by g will have an overall expense of $\mathcal{O}(C_g C_{\bar{b}} T_g T_{\bar{b}})$.

So we have the complexity estimates for classical polynomial division in Table 6.3, and we note that this is the same as the complexity of classical polynomial multiplication of $\bar{a}, \bar{b}$ times g (the product between the quotients and the divisor).

Dense Division	$\mathcal{O}(C_g {N_g}^n (C_{\bar{a}} {N_{\bar{a}}}^n + C_{\bar{b}} {N_{\bar{b}}}^n))$
Sparse Division	$\mathcal{O}(C_g T_g (C_{\bar{a}} T_{\bar{a}} + C_{\bar{b}} T_{\bar{b}}))$

Table 6.3: Classical Polynomial Division Expense

6.8.2 Asymptotic complexity of *Brown's* Algorithm

We first consider the simplest algorithm, and determine the complexity of Brown's algorithm for a general number of variables. For both algorithms (M and P), all steps that clearly have a lower order effect are ignored.

First consider the content computation in step 1 of algorithm M. This involves computations with coefficients of size $C_{g\bar{a}}$,$C_{g\bar{b}}$ for $\mathcal{O}({N_{g\bar{a}}}^n)$, $\mathcal{O}({N_{g\bar{b}}}^n)$ terms in the dense case, and $\mathcal{O}(T_g T_{\bar{a}})$, $\mathcal{O}(T_g T_{\bar{b}})$ terms in the sparse case. We will utilize the improved content computation scheme, and we obtain expenses (Table 6.2) of $\mathcal{O}(C_{g\bar{a}}{N_{g\bar{a}}}^n + C_{g\bar{b}}{N_{g\bar{b}}}^n + C_{g\bar{a}}C_{g\bar{b}})$ for the dense case, and $\mathcal{O}(C_{g\bar{a}}T_g T_{\bar{a}} + C_{g\bar{b}}T_g T_{\bar{b}} + C_{g\bar{a}}C_{g\bar{b}})$ for the sparse case. Since we assume for the analysis that $\mathrm{cont}_{\mathbf{x}}(g) = 1$, there will be no content to remove.

Next consider the number of primes needed by algorithm M. This is initially chosen as $B = 2\gamma\max(||a||_\infty, ||b||_\infty)$. As discussed in Brown [8], the exceptional cases where the estimate for B is too small are generally rare, so we will assume it is sufficient. Based on our assumptions we have $||a||_\infty = \mathcal{O}(C_{g\bar{a}})$ and $||b||_\infty = \mathcal{O}(C_{g\bar{b}})$, so we can obtain the worst case bound $log(B) = \mathcal{O}(\min(C_{g\bar{a}}, C_{g\bar{b}}) + \max(C_{g\bar{a}}, C_{g\bar{b}}))$. It is straightforward to see that this is equivalent to the best case bound, so we obtain $log(B) = \mathcal{O}(C_{g\bar{a}\bar{b}})$, and the number of iterations of the loop will be proportional to the length of the maximum integer coefficient in $g, \bar{a}$, and $\bar{b}$.

Consider the expense for the steps inside the loop for all iterations. For a single pass, the modular reductions in step 7 of algorithm M have a cost per coefficient that is proportional to the length of the coefficient. This means that a single reduction of a, b in the dense case has associated cost $\mathcal{O}(C_{g\bar{a}} \times {N_{g\bar{a}}}^n + C_{g\bar{b}} \times {N_{g\bar{b}}}^n)$, and a single reduction of a, b in the sparse case has associated cost $\mathcal{O}(C_{g\bar{a}} \times T_g T_{\bar{a}} + C_{g\bar{b}} \times T_g T_{\bar{b}})$, so for all iterations these become $\mathcal{O}(C_{g\bar{a}\bar{b}}(C_{g\bar{a}}{N_{g\bar{a}}}^n + C_{g\bar{b}}{N_{g\bar{b}}}^n))$ and $\mathcal{O}(C_{g\bar{a}\bar{b}}(C_{g\bar{a}}T_g T_{\bar{a}} + C_{g\bar{b}}T_g T_{\bar{b}}))$ for the dense and sparse cases respectively. We denote the cost of the call to Brown's P algorithm for n variables in step 8 by $\mathrm{CostP}_d(n)$, $\mathrm{CostP}_s(n)$ for the dense and sparse cases respectively, so their total expense is multiplied by $\mathcal{O}(C_{g\bar{a}\bar{b}})$.

Consider the overall cost of performing the Chinese remaindering to obtain the GCD and cofactors in $\mathbf{Z}[\mathbf{x}]$ over all iterations of the loop. This is known to be proportional to the square of the number of images used for each coefficient, so this is given by $\mathcal{O}({C_{g\bar{a}\bar{b}}}^2 \times ({N_g}^n + {N_{\bar{a}}}^n + {N_{\bar{b}}}^n)) = \mathcal{O}({C_{g\bar{a}\bar{b}}}^2 {N_{g\bar{a}\bar{b}}}^n)$ for the dense case, and $\mathcal{O}({C_{g\bar{a}\bar{b}}}^2 \times (T_g + T_{\bar{a}} + T_{\bar{b}})) =$

$\mathcal{O}({C_{g\bar{a}\bar{b}}}^2 T_{g\bar{a}\bar{b}})$ for the sparse case.

As for the cost of the content computation and removal in step 16, we can bound it by the cost of the Chinese remaindering. We again utilize the improved content computation scheme, which for the computation of the three required contents can be shown to be $\mathcal{O}({C_{g\bar{a}\bar{b}}}^2 + C_{g\bar{a}\bar{b}} \times ({N_g}^n + {N_{\bar{a}}}^n + {N_{\bar{b}}}^n)) = \mathcal{O}({C_{g\bar{a}\bar{b}}}^2 + C_{g\bar{a}\bar{b}} {N_{g\bar{a}\bar{b}}}^n)$ for the dense case, and $\mathcal{O}({C_{g\bar{a}\bar{b}}}^2 + C_{g\bar{a}\bar{b}} \times (T_g + T_{\bar{a}} + T_{\bar{b}})) = \mathcal{O}({C_{g\bar{a}\bar{b}}}^2 + C_{g\bar{a}\bar{b}} T_{g\bar{a}\bar{b}})$ for the sparse case. Removal of the computed contents has expense $\mathcal{O}({C_{g\bar{a}\bar{b}}}^2 {N_{g\bar{a}\bar{b}}}^n)$ for the dense case, and $\mathcal{O}({C_{g\bar{a}\bar{b}}}^2 T_{g\bar{a}\bar{b}})$ for the sparse case, so the total asymptotic expense for this step is the same as the expense of the Chinese remaindering.

Combining and simplifying the above gives the asymptotic cost estimates

$$\begin{aligned}
\mathrm{CostM}_d(n) &= \mathcal{O}({C_{g\bar{a}\bar{b}}}^2 {N_{g\bar{a}\bar{b}}}^n) + C_{g\bar{a}\bar{b}} \times \mathrm{CostP}_d(n), \\
\mathrm{CostM}_s(n) &= \mathcal{O}(C_{g\bar{a}\bar{b}} T_g (C_{g\bar{a}} T_{\bar{a}} + C_{g\bar{b}} T_{\bar{b}}) + {C_{g\bar{a}\bar{b}}}^2 T_{g\bar{a}\bar{b}}) + C_{g\bar{a}\bar{b}} \times \mathrm{CostP}_s(n),
\end{aligned}$$

for the dense and sparse cases, respectively.

Now consider the cost of algorithm P for n variables. We need to take a good deal more care in the sparse case, as a polynomial that is sparse in n variables may actually be dense when considered as a polynomial in fewer variables. To handle this, we will bound the term counts by their dense equivalents. So, for example, a polynomial having originally T_g terms, when evaluated down to a univariate polynomial, will have at most $\min(T_g, N_g + 1)$ terms.

For the content computation in step 1 we again utilize the improved method (Table 6.2) giving expenses of $\mathcal{O}({N_{g\bar{a}}}^n + {N_{g\bar{b}}}^n + N_{g\bar{a}} N_{g\bar{b}})$ for the dense case, and $\mathcal{O}(\min(T_g T_{\bar{a}}, {N_{g\bar{a}}}^n) + \min(T_g T_{\bar{b}}, {N_{g\bar{b}}}^n) + N_{g\bar{a}} N_{g\bar{b}})$ for the sparse case. Since the GCD is assumed to be content free, we expect that the content is small (the content is 1 with high probability as it can only be introduced by the prime or evaluations), so the cost of its removal is $\mathcal{O}({N_{g\bar{a}}}^n + {N_{g\bar{b}}}^n)$ in the dense case and $\mathcal{O}(\min(T_g T_{\bar{a}}, {N_{g\bar{a}}}^n) + \min(T_g T_{\bar{b}}, {N_{g\bar{b}}}^n))$ in the sparse case.

The analysis of the number of evaluation points needed for algorithm P is similar to the analysis of the number of primes needed for algorithm M. In summary, for all cases, $\mathcal{O}(N_{g\bar{a}\bar{b}})$ evaluation points are needed to reconstruct the dependence of x_n for $g, \bar{a}$, and $\bar{b}$.

Consider the expense for the steps inside the loop for all iterations. For a single pass, the modular evaluations in step 7 of algorithm P have $\mathcal{O}(1)$ cost per term (a multiplication in $\mathbf{Z}_p$). This means that a single evaluation of a,b in the dense case has associated cost $\mathcal{O}({N_{g\bar{a}}}^n + {N_{g\bar{b}}}^n)$, and a single evaluation of a,b in the sparse case has associated cost $\mathcal{O}(\min(T_g T_{\bar{a}}, {N_{g\bar{a}}}^n) + \min(T_g T_{\bar{b}}, {N_{g\bar{b}}}^n) + N_{g\bar{a}\bar{b}})$, so for all iterations these become $\mathcal{O}(N_{g\bar{a}\bar{b}}({N_{g\bar{a}}}^n + {N_{g\bar{b}}}^n))$ and $\mathcal{O}(N_{g\bar{a}\bar{b}}(\min(T_g T_{\bar{a}}, {N_{g\bar{a}}}^n) + \min(T_g T_{\bar{b}}, {N_{g\bar{b}}}^n) + N_{g\bar{a}\bar{b}}))$ for the dense and sparse cases, respectively. In step 8 we have the recursive call to algorithm P with one fewer variable, so the total expense is simply multiplied by $\mathcal{O}(N_{g\bar{a}\bar{b}})$.

Consider the overall cost of performing the Chinese remaindering (equivalent to Newton interpolation) to reconstruct the dependence of x_n. This is known to be proportional to the square of the number of images used for each coefficient, so this is given by $\mathcal{O}({N_{g\bar{a}\bar{b}}}^2 \times ({N_g}^{n-1} + {N_{\bar{a}}}^{n-1} + {N_{\bar{b}}}^{n-1})) = \mathcal{O}({N_{g\bar{a}\bar{b}}}^{n+1})$ for the dense case, and $\mathcal{O}({N_{g\bar{a}\bar{b}}}^2 \times (\min(T_g, {N_g}^{n-1}) + \min(T_{\bar{a}}, {N_{\bar{a}}}^{n-1}) + \min(T_{\bar{b}}, {N_{\bar{b}}}^{n-1})))$ for the sparse case, and for the latter we apply the notation $\mathcal{O}({N_{g\bar{a}\bar{b}}}^2 \times \min(T, N^{n-1})_{g\bar{a}\bar{b}})$ for brevity.

Finally, as for the M algorithm, the cost of the content computations in step 16 is bounded by the cost of the Chinese remaindering, so combining and simplifying the above gives the asymptotic cost estimates

$$\begin{aligned}
\mathrm{CostP}_d(n) &= \mathcal{O}({N_{g\bar{a}\bar{b}}}^{n+1}) + N_{g\bar{a}\bar{b}}\mathrm{CostP}_d(n-1), \\
\mathrm{CostP}_s(n) &= \mathcal{O}(N_{g\bar{a}\bar{b}}(\min(T_g T_{\bar{a}}, {N_{g\bar{a}}}^n) + \min(T_g T_{\bar{b}}, {N_{g\bar{b}}}^n)) + {N_{g\bar{a}\bar{b}}}^2 \min(T, N^{n-1})_{g\bar{a}\bar{b}}) \\
&\quad + N_{g\bar{a}\bar{b}}\mathrm{CostP}_s(n-1),
\end{aligned}$$

for the dense and sparse cases, respectively.

Once we are left with only one variable (i.e. $n = 1$), we apply the Euclidean algorithm to the inputs, which has a cost that depends only upon the degree, having expense $\mathcal{O}(N_{g\bar{a}} N_{g\bar{b}})$ for both cases. The expense of the division to obtain the cofactors $\bar{a}$, $\bar{b}$ is $\mathcal{O}(N_g N_{\bar{a}} + N_g N_{\bar{b}})$, so we obtain $\mathrm{CostP}_d(1) = \mathrm{CostP}_s(1) = \mathcal{O}(N_{g\bar{a}} N_{g\bar{b}})$.

These recurrences are easily solved from the bottom up.

First consider the dense case, for which the complexity has a similar structure for any number of variables. Since the cost for a single variable is $\mathcal{O}(N_{g\bar{a}} N_{g\bar{b}})$, we compute

$$\begin{aligned}
\mathrm{CostP}_d(2) &= \mathcal{O}({N_{g\bar{a}\bar{b}}}^3) + N_{g\bar{a}\bar{b}} \times \mathcal{O}(N_{g\bar{a}} N_{g\bar{b}}) &&= \mathcal{O}(2{N_{g\bar{a}\bar{b}}}^3), \\
\mathrm{CostP}_d(3) &= \mathcal{O}({N_{g\bar{a}\bar{b}}}^4) + N_{g\bar{a}\bar{b}} \times \mathcal{O}(2{N_{g\bar{a}\bar{b}}}^3) &&= \mathcal{O}(3{N_{g\bar{a}\bar{b}}}^4), \\
\mathrm{CostP}_d(4) &= \mathcal{O}({N_{g\bar{a}\bar{b}}}^5) + N_{g\bar{a}\bar{b}} \times \mathcal{O}(3{N_{g\bar{a}\bar{b}}}^4) &&= \mathcal{O}(4{N_{g\bar{a}\bar{b}}}^5),
\end{aligned}$$

which gives $\mathrm{CostP}_d(n) = \mathcal{O}(nN_{g\bar{a}\bar{b}}{}^{n+1})$ for algorithm P, and combining with the result for algorithm M we obtain

$$\mathrm{CostM}_d(n) = \mathcal{O}(C_{g\bar{a}\bar{b}}{}^2 N_{g\bar{a}\bar{b}}{}^n + nC_{g\bar{a}\bar{b}} N_{g\bar{a}\bar{b}}{}^{n+1}). \tag{6.4}$$

For the sparse case we will proceed differently. We let n_x be the greatest number of variables in which the inputs and outputs can be considered dense polynomials in those variables. More precisely, we choose n_x, $2 \le n_x \le n$ so that $T_g T_{\bar{a}} = \mathcal{O}(N_{g\bar{a}}{}^{n_x})$ and $T_g T_{\bar{b}} = \mathcal{O}(N_{g\bar{b}}{}^{n_x})$, but $T_g T_{\bar{a}} = o(N_{g\bar{a}}{}^{n_x+1})$ and $T_g T_{\bar{b}} = o(N_{g\bar{b}}{}^{n_x+1})$. With this definition of n_x we state that the complexity of Brown's algorithm M for n variables is given by

$$\mathrm{CostP}_s(n) = \mathcal{O}(n_x N_{g\bar{a}\bar{b}}{}^{n+1}).$$

We provide an outline of the proof. For $n = 2$ we obtain

$$\begin{aligned}\mathrm{CostP}_s(2) &= \mathcal{O}(N_{g\bar{a}\bar{b}}(\min(T_g T_{\bar{a}}, N_{g\bar{a}}{}^2) + \min(T_g T_{\bar{b}}, N_{g\bar{b}}{}^2)) + N_{g\bar{a}\bar{b}}{}^2 \min(T, N)_{g\bar{a}\bar{b}}) \\ &\quad + \mathcal{O}(N_{g\bar{a}\bar{b}} N_{g\bar{a}} N_{g\bar{b}}).\end{aligned}$$

We claim that the above expression is bounded by $\mathcal{O}(N_{g\bar{a}\bar{b}}{}^3)$, and further for sufficiently dense g,$\bar{a}$,$\bar{b}$ this bound is tight. Clearly it is tight if g,$\bar{a}$,$\bar{b}$ are dense, but this is not necessary, as they need only be dense in x_1, not in all variables. This can be most easily seen from the third term in the estimate, which becomes $\mathcal{O}(N_{g\bar{a}\bar{b}}{}^3)$ when g,$\bar{a}$,$\bar{b}$ are dense in x_1.

Now if we assume that our statement is true for $i-1$, $i \le n_x$ then we have

$$\begin{aligned}\mathrm{CostP}_s(i) &= \mathcal{O}(N_{g\bar{a}\bar{b}}(N_{g\bar{a}}{}^i + N_{g\bar{b}}{}^i) + N_{g\bar{a}\bar{b}}{}^{i+1}) + N_{g\bar{a}\bar{b}} \times \mathrm{CostP}_s(i-1) \\ &= \mathcal{O}(N_{g\bar{a}\bar{b}}{}^{i+1}) + N_{g\bar{a}\bar{b}} \times \mathcal{O}((i-1)N_{g\bar{a}\bar{b}}{}^i) \\ &= \mathcal{O}(iN_{g\bar{a}\bar{b}}{}^{i+1}).\end{aligned}$$

Now if we consider what happens for $i = n_x + 1$ we obtain

$$\begin{aligned}\mathrm{CostP}_s(n_x+1) &= \mathcal{O}(N_{g\bar{a}\bar{b}}(T_g T_{\bar{a}} + T_g T_{\bar{b}}) + N_{g\bar{a}\bar{b}}{}^{n_x} T_{g\bar{a}\bar{b}}) + N_{g\bar{a}\bar{b}} \times \mathrm{CostP}_s(n_x) \\ &= o(N_{g\bar{a}\bar{b}}{}^{n_x+1}) + N_{g\bar{a}\bar{b}} \times \mathcal{O}(n_x N_{g\bar{a}\bar{b}}{}^{n_x+1}) \\ &= \mathcal{O}(n_x N_{g\bar{a}\bar{b}}{}^{n_x+1}),\end{aligned}$$

and this continues to hold for all $i > n_x + 1$.

It may not be entirely clear in the analysis, but the primary expense in the sparse case comes from the recursive calls for $i \leq n_x$, unlike the dense case where the contribution from the current variable and the recursion have similar complexity. This provides some motivation for the Zippel approach, and also indicates that if the cases for small i can be coded efficiently, an implementation of this algorithm may be suitable for moderately sparse inputs.

In any event, combining the expense of algorithm P with the result for algorithm M we obtain the complexity estimate for the sparse case as

$$\mathrm{CostM}_s(n) = \mathcal{O}(C_{g\bar{a}\bar{b}} T_g (C_{g\bar{a}} T_{\bar{a}} + C_{g\bar{b}} T_{\bar{b}}) + {C_{g\bar{a}\bar{b}}}^2 T_{g\bar{a}\bar{b}} + C_{g\bar{a}\bar{b}} n_x {N_{g\bar{a}\bar{b}}}^{n+1}). \tag{6.5}$$

6.8.3 Asymptotic complexity of *DIVBRO*

We now consider the complexity of the *DIVBRO* algorithm for sparse and dense multivariate polynomials.

The expense for the content computation in step 1 of algorithm M is the same as that of Brown M, namely $\mathcal{O}(C_{g\bar{a}} {N_{g\bar{a}}}^n + C_{g\bar{b}} {N_{g\bar{b}}}^n + C_{g\bar{a}} C_{g\bar{b}})$ for the dense case, and $\mathcal{O}(C_{g\bar{a}} T_g T_{\bar{a}} + C_{g\bar{b}} T_g T_{\bar{b}} + C_{g\bar{a}} C_{g\bar{b}})$ for the sparse case. Since we assume for the analysis that $\mathrm{cont}_{\mathbf{x}}(g) = 1$, there will be no content to remove. We also note that this expense bounds the single integer GCD in step 3. The expense of the degree bound computations from step 4 can be taken directly from Table 6.1, and we denote these as $B_d(n), B_n(n)$ for the dense and sparse cases, respectively.

Now consider the number of primes needed by algorithm *DIVBRO M*. Recall that with high probability the algorithm will use at most one more prime than needed to reconstruct $\frac{\gamma}{\mathrm{lc}_{\mathbf{x}}(g)} \times g$, where $\gamma = \mathrm{GCD}(lc_{\mathbf{x}}(a), lc_{\mathbf{x}}(b))$. If we use C_γ to represent the length of the integer $\frac{\gamma}{\mathrm{lc}_{\mathbf{x}}(g)}$, then the number of primes required is $\mathcal{O}(C_g + C_\gamma) = \mathcal{O}(C_{g\gamma})$.

Consider the expense for the steps inside the loop for all iterations. The expense of performing the modular evaluations in step 7 of *DIVBRO M* will be the same as for Brown's algorithm M, but will be repeated $\mathcal{O}(C_{g\gamma})$ times, so the total expense for all iterations will be $\mathcal{O}(C_{g\gamma} \times (C_{g\bar{a}} {N_{g\bar{a}}}^n + C_{g\bar{b}} {N_{g\bar{b}}}^n))$ for the dense case, and $\mathcal{O}(C_{g\gamma} \times (C_{g\bar{a}} T_g T_{\bar{a}} + C_{g\bar{b}} T_g T_{\bar{b}}))$ for the sparse case. We denote the expense of the call to *DIVBRO P* for n variables in step 8 by $\mathrm{CostP}_d(n), \mathrm{CostP}_s(n)$ for the dense and sparse cases, respectively, so their total expense is multiplied by $\mathcal{O}(C_{g\gamma})$.

Consider the overall expense of performing the Chinese remaindering to obtain the GCD in $\mathbf{Z}[\mathbf{x}]$ over all iterations of the loop. The analysis is the same as for Brown's M algorithm except that we are only reconstructing g, so the expense is given by $\mathcal{O}({C_{g\gamma}}^2 \times {N_g}^n)$ for the dense case, and $\mathcal{O}({C_{g\gamma}}^2 \times T_g)$ for the sparse case.

The content computation in step 16 is bounded by the expense of the Chinese remaindering (using the same reasoning as for Brown's algorithm).

If we denote the dense and sparse division costs in step 16 by $D_d(n)$ and $D_s(n)$ respectively, and combine and simplify the results above, we obtain the asymptotic cost estimates

$$\begin{aligned}
\mathrm{CostM}_d(n) &= \mathcal{O}(C_{g\bar{a}}C_{g\bar{b}} + C_{g\gamma}(C_{g\bar{a}}{N_{g\bar{a}}}^n + C_{g\bar{b}}{N_{g\bar{b}}}^n)) + C_{g\gamma} \times \mathrm{CostP}_d(n) + B_d(n) + D_d(n), \\
\mathrm{CostM}_s(n) &= \mathcal{O}(C_{g\bar{a}}C_{g\bar{b}} + C_{g\gamma}T_g(C_{g\bar{a}}T_{\bar{a}} + C_{g\bar{b}}T_{\bar{b}})) + C_{g\gamma} \times \mathrm{CostP}_s(n) + B_s(n) + D_s(n),
\end{aligned}$$

for the dense and sparse cases respectively.

Now consider the expense of algorithm *DIVBRO P* for n variables. Again we need to take care here for the sparse case, as a polynomial that is sparse in n variables may actually be dense when considered as a polynomial in fewer variables.

As for Brown's P algorithm, the expense of computing the contents in step 1 will be $\mathcal{O}({N_{g\bar{a}}}^n + {N_{g\bar{b}}}^n + N_{g\bar{a}}N_{g\bar{b}})$ for the dense case, and $\mathcal{O}(\min(T_gT_{\bar{a}}, {N_{g\bar{a}}}^n) + \min(T_gT_{\bar{b}}, {N_{g\bar{b}}}^n) + N_{g\bar{a}}N_{g\bar{b}})$ for the sparse case. Also as for Brown's P algorithm, we assume the content is $\mathcal{O}(1)$, so its removal has the same expense as its computation.

We will need sufficiently many evaluation points to reconstruct $\frac{\gamma}{\mathrm{lc}_{x_1,\ldots,x_{n-1}}(g)}g$, so if the degree of $\frac{\gamma}{\mathrm{lc}_{x_1,\ldots,x_{n-1}}(g)}$ in x_n is denoted N_γ, the number of evaluation points will be $\mathcal{O}(N_g + N_\gamma) = \mathcal{O}(N_{g\gamma})$.

Consider the expense for the steps inside the loop for all iterations. The expense of performing the modular evaluations in step 7 of *DIVBRO P* will be the same as for Brown's algorithm P, but will be repeated $\mathcal{O}(N_{g\gamma})$ times, so the total expense for all iterations will be $\mathcal{O}(N_{g\gamma}({N_{g\bar{a}}}^n + {N_{g\bar{b}}}^n))$ for the dense case and $\mathcal{O}(N_{g\gamma}(\min(T_gT_{\bar{a}}, {N_{g\bar{a}}}^n) + \min(T_gT_{\bar{b}}, {N_{g\bar{b}}}^n) + N_{g\bar{a}\bar{b}}))$ for the sparse case. In step 8 we have the recursive call to *DIVBRO P* with one fewer variable, so the total expense is simply multiplied by $\mathcal{O}(N_{g\gamma})$.

Now outside the loop, we consider the expense of performing the Newton interpolation in step 18 to reconstruct the dependence of x_n. As with Chinese remaindering, this is known to be proportional to the square of the number of images used for each coefficient, so this is given by $\mathcal{O}({N_{g\gamma}}^2 \times {N_g}^{n-1})$ for the dense case, and $\mathcal{O}({N_{g\gamma}}^2 \times \min(T_g, {N_g}^{n-1}))$ for the sparse case.

Again we ignore the content computations in step 19 by the same reasoning as *DIVBRO M*. The expense of the division tests in step 19 are given by $\mathcal{O}({N_g}^n {N_{\bar{a}\bar{b}}}^n)$ for the dense case, and $\mathcal{O}(T_g T_{\bar{a}\bar{b}})$ for the sparse case (as the coefficients are $\mathcal{O}(1)$ in *DIVBRO P*). We combine and simplify the above giving the asymptotic cost estimates

$$\begin{aligned}
\mathrm{CostP}_d(n) &= \mathcal{O}(N_{g\gamma} {N_{g\bar{a}\bar{b}}}^n) + \mathcal{O}({N_g}^n {N_{\bar{a}\bar{b}}}^n) + N_{g\gamma}\mathrm{CostP}_d(n-1),\\
\mathrm{CostP}_s(n) &= \mathcal{O}(N_{g\gamma}(\min(T_g T_{\bar{a}}, {N_{g\bar{a}}}^n) + \min(T_g T_{\bar{b}}, {N_{g\bar{b}}}^n)) + {N_{g\gamma}}^2 \min(T_g, {N_g}^{n-1}))\\
&\quad + N_{g\gamma}\mathrm{CostP}_s(n-1),
\end{aligned}$$

where the expense of the division is written separately for the dense case (the source of the $\mathcal{O}({N_g}^n {N_{\bar{a}\bar{b}}}^n)$ term), and is asymptotically smaller than the other expenses for the sparse case (if the polynomials are sufficiently sparse).

Again the single variable case, $n = 1$, has expense $\mathrm{CostP}_d(1) = \mathrm{CostP}_s(1) = \mathcal{O}(N_{g\bar{a}} N_{g\bar{b}})$.

Before solving these recurrences we will make an observation. The C_γ factor is at worst bounded by $C_{\bar{a}\bar{b}}$ for *DIVBRO P* and the N_γ factor is at worst bounded by $N_{\bar{a}\bar{b}}$ for *DIVBRO M*, and in the event that these bounds are tight, the recurrences (ignoring the division cost) are of the same form as those obtained for Brown's algorithm, so their solution is identical. In many cases, C_γ, N_γ are small, with one notable exception, namely computation of a square-free polynomial. It is still possible to reduce the size of this factor, by utilizing a trailing coefficient normalization instead of a leading coefficient normalization. With this in mind, we assume that cases where C_γ, N_γ are at their worst are generally rare, and proceed with the analysis assuming that $C_\gamma = \mathcal{O}(C_g)$ and $N_\gamma = \mathcal{O}(N_g)$. We now solve these recurrences under the new assumptions from the bottom up.

As for Brown's P algorithm, the solution of the recurrence is fairly straightforward, and we have

$$\begin{aligned}
\mathrm{CostP}_d(2) &= \mathcal{O}(N_g {N_{g\bar{a}\bar{b}}}^2 + {N_g}^2 {N_{\bar{a}\bar{b}}}^2) + N_g \times \mathcal{O}(N_{g\bar{a}} N_{g\bar{b}}) &&= \mathcal{O}(2N_g {N_{g\bar{a}\bar{b}}}^2 + {N_g}^2 {N_{\bar{a}\bar{b}}}^2),\\
\mathrm{CostP}_d(3) &= \mathcal{O}(N_g {N_{g\bar{a}\bar{b}}}^3 + {N_g}^3 {N_{\bar{a}\bar{b}}}^3) + N_g \times \mathrm{CostP}_d(2) &&= \mathcal{O}(3N_g {N_{g\bar{a}\bar{b}}}^3 + {N_g}^3 {N_{\bar{a}\bar{b}}}^3),\\
\mathrm{CostP}_d(4) &= \mathcal{O}(N_g {N_{g\bar{a}\bar{b}}}^4 + {N_g}^4 {N_{\bar{a}\bar{b}}}^4) + N_g \times \mathrm{CostP}_d(3) &&= \mathcal{O}(4N_g {N_{g\bar{a}\bar{b}}}^4 + {N_g}^4 {N_{\bar{a}\bar{b}}}^4),
\end{aligned}$$

so for the *DIVBRO P* algorithm we obtain $\mathrm{CostP}_d(n) = \mathcal{O}(nN_gN_{g\bar{a}\bar{b}}{}^n + N_g{}^nN_{\bar{a}\bar{b}}{}^n)$, where we note that the second term in the estimate is a direct result of the division test only. An observation can be made that for $N_g \ll N_{\bar{a}\bar{b}}$ the core contribution to the asymptotic expense comes from the evaluations and divisions performed at the top level of the recursion, and the expense simplifies to $\mathrm{CostP}_d(n) = \mathcal{O}(N_gN_{g\bar{a}\bar{b}}{}^n + N_g{}^nN_{\bar{a}\bar{b}}{}^n)$. This contrasts with Brown's *P* algorithm, where the current and recursive components contribute equally to the asymptotic expense in all cases.

We obtain the total expense for the *DIVBRO* algorithm in the dense case (including the division cost from Table 6.3 and the degree bound cost from Table 6.1) as

$$\begin{aligned}\mathrm{CostM}_d(n) &= \mathcal{O}(C_{g\bar{a}}C_{g\bar{b}} + C_g(C_{g\bar{a}}N_{g\bar{a}}{}^n + C_{g\bar{b}}N_{g\bar{b}}{}^n) + nC_gN_gN_{g\bar{a}\bar{b}}{}^n) \\ &\quad + \mathcal{O}(C_gN_g{}^n(C_{\bar{a}}N_{\bar{a}}{}^n + C_{\bar{b}}N_{\bar{b}}{}^n)). \qquad (6.6)\end{aligned}$$

For the sparse case we will proceed as in Brown, with the same definition for n_x, $1 < n_x \le n$ the number of variables after which the inputs begin to appear sparse. The solution in this case is given by

$$\mathrm{CostP}_s(n) = \mathcal{O}(N_g{}^{n-n_x+1}N_{g\bar{a}\bar{b}}{}^{n_x}\sum_{i=2}^{n_x}\alpha^{i-2}),$$

for $n, n_x \ge 2$ and $\alpha = \frac{N_g}{N_{g\bar{a}\bar{b}}} \le 1$.

We now proceed to outline the proof.

For $n = 2$ we can obtain the solution directly from $\mathrm{CostP}_s(1)$ and the recurrence for $\mathrm{CostP}_s(n)$ evaluated at $n = 2$ as

$$\mathrm{CostP}_s(2) = \mathcal{O}(N_g(N_{g\bar{a}}{}^2 + N_{g\bar{b}}{}^2) + N_g{}^2N_g + N_gN_{g\bar{a}}N_{g\bar{b}})$$

where we have utilized the assumption that $n_x \ge 2$ so the minimums are known. It is clear that all terms are bounded by $N_gN_{g\bar{a}\bar{b}}{}^2$, and further that the bound is tight for any of N_g,$N_{\bar{a}}$,$N_{\bar{b}}$ maximal. This agrees with the form of the solution.

Now we assume that the estimate holds for $n = j - 1 < n_x$, so for $n = j$ we obtain

$$\begin{aligned}\mathrm{CostP}_s(j) &= \mathcal{O}(N_g(N_{g\bar{a}}{}^j + N_{g\bar{b}}{}^j) + N_g{}^2N_g{}^{j-1}) + N_g \times \mathrm{CostP}_s(j-1) \\ &= \mathcal{O}(N_gN_{g\bar{a}\bar{b}}{}^j) + N_g \times \mathcal{O}(N_gN_{g\bar{a}\bar{b}}{}^{j-1}\sum_{i=2}^{j-1}\alpha^{i-2})\end{aligned}$$

$$
\begin{aligned}
&= \mathcal{O}(N_g {N_{g\bar{a}\bar{b}}}^j) + N_{g\bar{a}\bar{b}}\alpha \times \mathcal{O}(N_g {N_{g\bar{a}\bar{b}}}^{j-1} \sum_{i=2}^{i-1} \alpha^{i-2}) \\
&= \mathcal{O}(N_g {N_{g\bar{a}\bar{b}}}^j (1 + \alpha \sum_{i=2}^{j-1} \alpha^{i-2})) \\
&= \mathcal{O}(N_g {N_{g\bar{a}\bar{b}}}^j \sum_{i=2}^{j} \alpha^{i-2}).
\end{aligned}
$$

Also for $n = n_x + 1$ we obtain

$$
\begin{aligned}
\mathrm{CostP}_s(n_x+1) &= \mathcal{O}(N_g(T_g T_{\bar{a}} + T_g T_{\bar{b}}) + {N_g}^2 T_g) + N_g \mathrm{CostP}_s(n_x) \\
&= \mathcal{O}(N_g {N_{g\bar{a}\bar{b}}}^{n_x}) + N_g \times \mathcal{O}(N_g {N_{g\bar{a}\bar{b}}}^{n_x} \sum_{i=2}^{n_x} \alpha^{i-2}) \\
&= \mathcal{O}({N_g}^2 {N_{g\bar{a}\bar{b}}}^{n_x} \sum_{i=2}^{n_x} \alpha^{i-2}),
\end{aligned}
$$

and this pattern continues for any number of variables, each level of the recurrence $n > n_x$ simply multiplying the complexity by the factor N_g.

And now a comment on the sum factor. Since $\alpha \le 1$, and $n_x \le n$, then this factor f is purely in the range $1 \le f \le n$. The analysis could have proceeded more smoothly by simply setting it to n, but this misses an interesting feature of the complexity. Specifically, if $\alpha = \frac{N_g}{N_{g\bar{a}\bar{b}}} \le \frac{1}{2}$, then the factor f is bounded by 2 regardless of the number of variables in the problem. In fact for any fixed value less than 1, there is a finite limit to the size of this factor that is independent of the number of variables in the problem. Note also that when $\alpha = 1$, $N_g = N_{g\bar{a}\bar{b}}$, and the result for the complexity is identical to the complexity of Brown's algorithm.

Using the shorthand $f = \sum_{i=2}^{n_x} \alpha^{i-2}$ we obtain the total expense for the *DIVBRO* algorithm in the sparse case as

$$
\begin{aligned}
\mathrm{CostM}_s(n) &= \mathcal{O}(C_{g\bar{a}} C_{g\bar{b}} + C_g T_g (C_{g\bar{a}} T_{\bar{a}} + C_{g\bar{b}} T_{\bar{b}}) + C_g f {N_g}^{n-n_x+1} {N_{g\bar{a}\bar{b}}}^{n_x} \\
&\quad + n \log_2(n) T_g T_{\bar{a}\bar{b}} + n N_{g\bar{a}} N_{g\bar{b}}), \qquad (6.7)
\end{aligned}
$$

where we have accounted for the division cost from Table 6.3, $\mathcal{O}(C_g T_g (C_{\bar{a}} T_{\bar{a}} + C_{\bar{b}} T_{\bar{b}}))$, as it is bounded by the second term in the above cost (the expense of the modular evaluations in algorithm *M*), and included the degree bound cost from Table 6.1.

There are a few practical points to note when comparing the Brown algorithm and the *DIVBRO* algorithm. The first is that *DIVBRO* was designed for use on small GCD problems, specifically those for which the division expense is not dominant. Consider the expenses of the Brown and *DIVBRO* algorithms when the size of g (in coefficient length, term count and degree) is $\mathcal{O}(1)$ in Table 6.4.

Brown Dense	$\mathcal{O}(C_{\bar{a}\bar{b}}{}^2 N_{\bar{a}\bar{b}}{}^n + nC_{\bar{a}\bar{b}}N_{\bar{a}\bar{b}}{}^{n+1})$
Brown Sparse	$\mathcal{O}(C_{\bar{a}\bar{b}}(C_{\bar{a}}T_{\bar{a}} + C_{\bar{b}}T_{\bar{b}}) + C_{\bar{a}\bar{b}}{}^2 T_{\bar{a}\bar{b}}{}^2 + C_{\bar{a}\bar{b}}n_x N_{\bar{a}\bar{b}}{}^n)$
DIVBRO Dense	$\mathcal{O}(C_{\bar{a}}C_{\bar{b}} + C_{\bar{a}}N_{\bar{a}}{}^n + C_{\bar{b}}N_{\bar{b}}{}^n)$
DIVBRO Sparse	$\mathcal{O}(C_{\bar{a}}C_{\bar{b}} + C_{\bar{a}}T_{\bar{a}} + C_{\bar{b}}T_{\bar{b}} + N_{\bar{a}\bar{b}}{}^{n_x} + n\log_2(n)T_{\bar{a}\bar{b}} + nN_{\bar{a}}N_{\bar{b}})$

Table 6.4: Small GCD Dense Algorithm Expense Comparison

For the dense case, the *DIVBRO* algorithm is asymptotically faster by at least $\mathcal{O}(N_{\bar{a}\bar{b}})$, as the expense is one degree lower. In addition the expense resulting from the coefficient operations is significantly smaller. For the sparse case the effect is even more dramatic, as n_x is usually quite small, likely between 1 and 3 for moderately sparse problems, so there is a significant savings there, as well as fewer coefficient operations.

The balanced case, where $N_g = \mathcal{O}(N_{\bar{a}}) = \mathcal{O}(N_{\bar{b}})$ will be discussed in a later section.

6.8.4 Asymptotic complexity of *EEZ-GCD*

The analysis for the *EEZ-GCD* algorithm is significantly more complex than that of the prior analyses, and involves a number of algorithms. It should be noted that modular operations generally need to be done in the symmetric range, as we are working with representations for signed integers.

A large portion of the expense is associated with the solution of the diophantine equations that arise in the course of the computation. There are two algorithms, the p-adic algorithm, which lifts an image from $\mathbf{Z}_p[x_1]$ to $\mathbf{Z}_{p^l}[x_1]$, and the I-adic algorithm, which lifts an image from $\mathbf{Z}_{p^l}[x_1, ..., x_n]/\langle x_n - \alpha_n\rangle$ to $\mathbf{Z}_{p^l}[x_1, ..., x_n]$.

One other key point is that only one of the inputs for the GCD problem is used for the lifting part of the algorithm, which can offer notable efficiency improvements if one input is significantly larger or more dense than the other.

For this analysis, without loss of generality, we choose a to be the polynomial product to be reconstructed, and we assume that the inputs are in two variables or more. We will continue to use subscript notation to describe quantities C, N, and T, and we also define $C = C_{uw}$ $(= C_{g\bar{a}})$ as this quantity occurs with great frequency in the analysis.

p-adic Algorithm

The algorithm for lifting an image in $\mathbf{Z}_p[x_1]$ to an image in $\mathbf{Z}_{p^l}[x_1]$ requires the solution of univariate diophantine equations in $\mathbf{Z}_p[x_1]$. Fortunately we are in a Euclidean domain, so we can apply the EEA to obtain the initial solution, and lift it p-adically.

The linear lifting algorithm is as follows:

Algorithm 23 (UniHensel)

Input: *A monic polynomial* $a \in \mathbf{Z}[x_1]$, *a prime* p, *relatively prime polynomials* $u_0, w_0 \in \mathbf{Z}_p[x_1]$ *satisfying* $a - u_0 w_0 = 0 \bmod p$, *and a bound* B *on the coefficient size of* u, w *on output.*

Output: $u, w \in \mathbf{Z}_{p^l}[x_1]$ *such that* $a - uw = 0$ *in* $\mathbf{Z}[x_1]$, *or* **Fail** *if no such* u, w *exist.*

1 *Solve* $\sigma u_0 + \tau w_0 = 1$ *for* $\sigma, \tau \in \mathbf{Z}_p[x_1]$ *using the* EEA

2 *Initialize, set* $e = a - u_0 w_0, \ \ m = p, \ \ u = u_0, \ \ w = w_0$

3 *Loop until* $e = 0$ *or* $m > 2B$

3.1 *Set* $e = e/p$

3.2 *Compute image of error in* $\mathbf{Z}_p[x_1]$, *set* $c = e \bmod p$

3.3 *Obtain diophantine solutions* u^i, w^i*:*
Set $s = \sigma c \bmod p, \ \ t = \tau c \bmod p,$
Divide s *by* w_0 *to get* $q, r \in \mathbf{Z}_p$ *satisfying* $s = w_0 q + r \bmod p,$
Set $w^i = r, \ \ u^i = t + q u_0 \bmod p$

3.4 *Update the error* e *and the images* u, w*:*
Set $e = e - (wu^i + uw^i) - mu^i w^i,$
$u = u + mu^i, \ \ w = w + mw^i$

3.5 *Update the coefficient magnitude, set* $m = m \times p$

4 *If* $e = 0$ *then return the solutions* u, w, *otherwise return* **Fail**.

Some implementations of the *EEZ-GCD* algorithm utilize a modification of this algorithm that provides better performance for cases where non-zero evaluation points are used. The l in the algorithm is specified by the *EEZ-GCD* algorithm (based on the object it is intended to reconstruct) instead of the bound B, and the solution is not required to satisfy $a - uw = 0$ but rather $a - uw = 0 \pmod{p}^l$. The advantage this provides is to avoid the use of a larger l as a result of the coefficient growth (which disappears as the variables are reconstructed) caused by the use of non-zero evaluation points. The change to the algorithm is minor, specifically the bound check in step 3 is changed to a fixed loop from 2 to l (retaining the early stop criteria for $e = 0$), and step 4 is changed to simply return u, w. This is the form of the algorithm we will use in the analysis, which means that $l \in \mathcal{O}(C)$ (while for non-zero evaluation points this is not true for the original algorithm).

We assume use of a machine sized prime p, so arithmetic operations in $\mathbf{Z}_p$ are $\mathcal{O}(1)$.

First we consider the dense case. The expense of step 1 is $\mathcal{O}(N_u N_w)$, and the computation of the error in step 2 is $\mathcal{O}(CN_u N_w)$, so as we will see, the expense is dominated by the expense of the loop. The average expense of steps 3.1, 3.2 is $\mathcal{O}(CN_{uw})$. The computation of the diophantine solutions in step 3.3 involves a number of multiplications between $\mathcal{O}(N)$ sized objects (all $\bmod p$), and can be shown to be $\mathcal{O}({N_{uw}}^2)$. The computation of the updates in step 3.4 is dwarfed by the update to the error, in which the first two products each have an average expense of $\mathcal{O}(CN_u N_w)$, and the third product has an average expense bounded by this. The expense of the computation inside the loop is $\mathcal{O}({N_{uw}}^2 + CN_u N_w)$, and as a result the overall asymptotic complexity is $\mathcal{O}(C{N_{uw}}^2 + C^2 N_u N_w)$.

Now we consider the sparse case. Note that even if u_0, w_0 are sparse, the polynomials σ, τ computed in step 1 will be dense. The expense of step 1 is the same as for the dense case, as the asymptotic expense of the EEA is independent of the density of the input polynomials. The expense of step 2 is $\mathcal{O}(CT_u T_w)$. The average expense of steps 3.1, 3.2 is $\mathcal{O}(CT_u T_w)$. All operations in step 3.3 involve multiplication or division of a dense and sparse polynomial, so the expense can be obtained as $\mathcal{O}(T_u T_w N_{uw})$. As in the dense case the computation of the updates in step 3.4 is dwarfed by the update to the error, having an average expense of $\mathcal{O}(CT_u T_w)$. The expense of the computation inside the loop is $\mathcal{O}(T_u T_w N_{uw} + CT_u T_w)$, and as a result the overall asymptotic complexity is $\mathcal{O}(N_u N_w + CT_u T_w N_{uw} + C^2 T_u T_w)$.

In some implementations, a shortcut is taken, and the modulus m is chosen so that

$m > 2B$. In some cases m is chosen to be a prime, while in others it may be chosen as a power of a prime. As a result, the entire univariate Hensel computation can be avoided, all expense being absorbed by the initial GCD computation. For this approach, the initial GCD computation takes the place of the Hensel lifting, and the cost is no longer negligible, so we compute it now.

In contrast to the Hensel approach, the large modulus approach now depends on the characteristics of the second GCD input b. Once we obtain the images in $\mathbf{Z}_p[x_1]$, the computation of the GCD is easily shown to be $\mathcal{O}(N_{uw}N_{u\bar{b}}C_{uw}C_{u\bar{b}})$. For the case where the two GCD inputs have similar characteristics, this simplifies to $\mathcal{O}({N_{uw}}^2C^2)$, which we see to have somewhat less attractive asymptotic performance in comparison to the small prime approach (at least when $N_{uw} = \mathcal{O}(C)$).

A summary of the results for the univariate Hensel algorithm is provided in Table 6.5.

Dense	$\mathcal{O}(C{N_{uw}}^2 + C^2N_uN_w)$
Sparse	$\mathcal{O}(N_uN_w + CT_uT_wN_{uw} + C^2T_uT_w)$
Large Modulus	$\mathcal{O}(N_{uw}N_{u\bar{b}}C_{uw}C_{u\bar{b}})$

Table 6.5: Univariate Diophantine Solution Expense

Multivariate Diophantine Equations

As mentioned previously, the solution of multivariate diophantine equations in $\mathbf{Z}_{p^l}$ is necessary for the *EEZ-GCD* algorithm. This analysis splits into a greater number of cases than the analysis of the p-adic algorithm, as the computations for dense and sparse cases are quite different, and the use of a zero evaluation point (versus a non-zero evaluation point) makes a significant difference to the complexity.

The algorithm is recursive in nature, and is as follows:

Algorithm 24 (MultiDio)

Input: *The prime p and power l of the coefficient domain $\mathbf{Z}_{p^l}$, polynomials $u, w, c \in \mathbf{Z}_{p^l}[x_1, ..., x_n]$, evaluation points $I = \langle x_2 - \alpha_2, ..., x_n - \alpha_n\rangle$, and the maximum degree d of the result with respect to any of $x_2, ..., x_n$.*

Output: $\sigma, \tau \in \mathbf{Z}_{p^l}[x_1, ..., x_n]$ *such that* $\sigma u + \tau w = c \in \mathbf{Z}_{p^l}[x_1, ..., x_n]$.

1 *If* $I = \emptyset$ *(we are in* $\mathbf{Z}_{p^l}[x_1]$*)*

1.1 *Compute* $u_0 = u \bmod p$, $w_0 = w \bmod p$, $c_0 = c \bmod p$

1.2 *Solve* $su_0 + tw_0 = 1$ *for* $s, t \in \mathbf{Z}_p[x_1]$ *using the* EEA

1.3 *Set* $v = c_0 s \mod p$

1.4 *Divide* v *by* w_0 *to get* $q, r \in \mathbf{Z}_p[x_1]$ *satisfying* $v = qw_0 + r$

1.5 *Set* $\sigma = r$, $\tau = c_0 t + u_0 q \mod p$

1.6 *Initialize, set* $m = p$, $e = c - \sigma u - \tau w$

1.7 *Loop* k *from* 2 *to* l *while* $e \neq 0$

1.7.1 *Set* $e = e/p$, $c_k = e \bmod p$

1.7.2 *Set* $v = c_k s \mod p$

1.7.3 *Divide* v *by* w_0 *to get* $q, r \in \mathbf{Z}_p[x_1]$ *satisfying* $v = qw_0 + r$

1.7.4 *Set* $q = c_k t + u_0 q \mod p$

1.7.5 *Update, set* $e = e - ru - qw$, $\sigma = \sigma + mr$, $\tau = \tau + mq$, $m = m \times p$

1.8 *return* σ, τ

2 *Evaluate:* $u_0 = u \bmod \langle x_n - \alpha_n \rangle$, $w_0 = w \bmod \langle x_n - \alpha_n \rangle$, $c_0 = c \bmod \langle x_n - \alpha_n \rangle$

3 *Recursively solve the diophantine equation* $\sigma u_0 + \tau w_0 = c_0$ *for* $\sigma, \tau \in \mathbf{Z}_{p^l}[x_1, ..., x_{n-1}]$, *and on failure return* **Fail**.

4 *Initialize: Set* $e = c - \sigma u - \tau w$, $m = 1$

5 *Loop* k *from* 1 *to* d *while* $e \neq 0$

5.1 *Set* $m = m \times (x_n - \alpha_n)$

5.2 *Set* c_k *to the* $(x_n - \alpha_n)^k$ *coefficient of the Taylor expansion of* e *about* $x_n = \alpha_n$.

5.3 *if* $c_k = 0$ *then next loop*

5.4 *Recursively solve the diophantine equation* $su_0 + tw_0 = c_k$ *for* $s, t \in \mathbf{Z}_{p^l}[x_1, ..., x_{n-1}]$, *and on failure return* **Fail**.

5.5 *Set* $s = m \times s$, $t = m \times t$

5.6 *Update the error* e *and images* σ, τ
Set $e = e - tu - sw$, $\sigma = \sigma + s$, $\tau = \tau + t$

6 *If* $e = 0$ *then return the solutions* σ, τ, *otherwise return* **Fail**.

As before we use subscript notation for the bounds for the degree (N), coefficient length (C), and term counts (T) for the quantities involved in the algorithm. The degree and form of the results are quite important for the overall analysis, and the values of these can be approximated in the analysis of the calling algorithm.

A few observations are in order here, and these are specific to the case where this algorithm is in use by the *EEZ-GCD* algorithm.

In all cases, the outputs of the algorithm, τ, σ are bounded by the degrees and term counts of u, w respectively. This is because they are used either as the updates for u, w if the algorithm is being called from *EEZ-GCD*, or they are part of the reconstruction of the updates to τ, σ if the algorithm is being called recursively. This means that $N_\tau \leq N_u$ and $N_\sigma \leq N_w$.

Additionally, the form of c entering into the algorithm is known in terms of the solution. Specifically, $c = u\sigma + w\tau$, where σ,τ are the outputs. This means that $N_c = \max(N_u + N_\sigma, N_w + N_\tau)$, implying $N_c \leq N_u + N_w$ (through use of $N_\sigma \leq N_w$, $N_\tau \leq N_u$). This also means that $T_c \leq T_u T_w$ for the sparse case.

Finally, the evaluations of u,w required in step 2 of the algorithm, and the diophantine solution required in step 1.2 of the algorithm are already known from computations performed in *EEZ-GCD*, so we will assume that this information is stored and can be used by this algorithm.

Base Case

Here we will look at the base case where $I = \emptyset$. For both the sparse and dense cases, the expense of the EEA in 1.2 is the same, $\mathcal{O}(C^2 N_u N_w)$, and the outputs s, t are both dense, having degrees N_w,N_u respectively. We note that the outputs of the algorithm, σ, τ will, however, be sparse in the sparse case, and the updates to these values must have similar sparsity.

For the dense case, the modular reductions in step 1.1 have expense $\mathcal{O}(C(N_c+N_u+N_w))$. The product in step 1.3 has expense $\mathcal{O}(N_c N_w)$ as does the quotient in step 1.4. The two products in step 1.5 have expense $\mathcal{O}(N_c N_u)$, and the error computation in step 1.6 has expense $\mathcal{O}(N_\sigma N_u + N_\tau N_w)$. Now within the loop, the average expense for both computations in step 1.7.1 is $\mathcal{O}(CN_c)$. The expense of steps 1.7.2 through 1.7.4 and the error update in step 1.7.5 have the same expense as steps 1.3-1.6, and the other updates in step 1.7.5 have average expense $\mathcal{O}(C(N_\sigma + N_\tau))$. Combining all expenses, both outside the loop and for all iterations of the loop, and removing obvious lower order terms, gives $\mathcal{O}(N_u N_w) + \mathcal{O}(C^2(N_c + N_\sigma + N_\tau)) + \mathcal{O}(CN_c(N_w + N_u))$, which further simplifies to $\mathcal{O}(N_u N_w) + \mathcal{O}(C^2 N_c + CN_{uw} N_c)$ where we have *intentionally* written the cost for the EEA separately, as it is already computed within the *EEZ-GCD* algorithm.

For the sparse case, the evaluations in step 1.1 have expense $\mathcal{O}(C(T_c + T_u + T_w))$. The product in step 1.3 has expense $\mathcal{O}(T_c N_w)$ and the quotient in step 1.4 has expense $\mathcal{O}(T_w N_c)$. The products in step 1.5 have expense $\mathcal{O}(T_c N_u + T_u N_c)$, and the error computation in step 1.6 has expense $\mathcal{O}(T_\sigma T_u + T_\tau T_w) = \mathcal{O}(T_c)$. Now within the loop, the average expense for both computations in step 1.7.1 is $\mathcal{O}(CT_c)$. The expense of steps 1.7.2 through 1.7.4 and the error update in step 1.7.5 have the same expense as steps 1.3-1.6, and the other updates in step 1.7.5 have average expense $\mathcal{O}(C(T_\sigma + T_\tau))$. Combining all expenses, both outside the loop and for all iterations of the loop, and removing obvious lower order terms, gives $\mathcal{O}(N_u N_w) + \mathcal{O}(C^2(T_c + T_\sigma + T_\tau)) + \mathcal{O}(CT_c(N_w + N_u) + CN_c(T_w + T_u))$, which further simplifies to $\mathcal{O}(N_u N_w) + \mathcal{O}(C^2 T_c + CN_{uw} T_c + CT_{uw} N_c)$, where *again* the separation for the EEA is intentional. We can further simplify this to $\mathcal{O}(C^2 N_u N_w) + \mathcal{O}(C^2 T_c + CN_{uw} T_c)$ as $N_c = \max(N_u + N_c, N_w + N_\sigma) = \mathcal{O}(N_{uw})$ and $T_{uw} \le T_u + T_w \le T_u T_\sigma + T_u T_\tau = \mathcal{O}(T_c)$.

Dense Non-zero

Now we look at the overall algorithm for the dense case when the evaluation point α_n is non-zero. Note that we consider here the case where the inputs are *fully dense*, i.e. that monomials $x_1^{i_1} x_2^{i_2} x_2^{i_3} \ldots$ where $0 \le i_1, i_2, i_3, \ldots \le N$ are present. Alternatively we could assume the inputs are *total degree dense* (where in the above, the restriction becomes $0 \le i_1 + i_2 + i_3 + \ldots \le N$), but the asymptotic cost for either case is the same.

The evaluations mod $\langle x_n - \alpha_n \rangle$ in step 2 have associated expense of $\mathcal{O}(C^2({N_u}^n + {N_w}^n + {N_c}^n))$, which is a count of the terms being evaluated. This can be simplified to $\mathcal{O}(C^2 {N_c}^n)$, both because $N_c > N_u$,$N_c > N_w$, and because the computation for u_0, w_0 is already known.

We then have the recursive call in step 3 with one fewer variable, which we denote as

$\text{Cost}_{dn}(n-1)$. The error computation in step 4 involves two products of dense polynomials in n and $n-1$ variables, which have an expense of $\mathcal{O}(C^2({N_\sigma}^{n-1}{N_u}^n + {N_\tau}^{n-1}{N_w}^n))$.

The loop starting on step 5 runs to the maximum degree of the diophantine solution for the current variable, which is $N_{\sigma\tau}$.

Further analysis of the algorithm requires estimates of the size of (number of terms in) the objects in the computation, so we determine these now. The polynomial factor m in step 5.1 is growing linearly in x_n with k. The size of the coefficient c_k computed in step 5.2 will be $\mathcal{O}({N_c}^{n-1})$, and the size of the error will be bounded by $\mathcal{O}({N_c}^{n-1}(N_c - k))$ as in each iteration the coefficient c_k is being set to zero. The diophantine solutions will have sizes of $\mathcal{O}({N_\sigma}^{n-1})$ for σ and $\mathcal{O}({N_\tau}^{n-1})$ for τ.

The update of the factor in step 5.1 has a cost of $\mathcal{O}(C^2k)$. The extraction of the coefficient c_k in step 5.2 will require $\mathcal{O}(C^2{N_c}^{n-1}(N_c - k))$ expense for the kth loop, as each of the remaining terms of the error have a contribution as a result of the non-zero evaluation point. Since we are in the dense case, c_k is likely non-zero, so we proceed to the recursive call in step 5.4 with expense $\text{Cost}_{dn}(n-1)$.

The multiplication in step 5.5 computes the products of the diophantine solutions and m, so the expense for the kth loop is $\mathcal{O}(C^2k({N_\sigma}^{n-1} + {N_\tau}^{n-1}))$. The updates to σ, τ, and e in step 5.6 are dominated by the update to e which involves multiplication of dense polynomials in n variables with the results of step 5.5, with an expense for the kth loop of $\mathcal{O}(C^2k({N_\sigma}^{n-1}{N_u}^n + {N_\tau}^{n-1}{N_w}^n))$.

The various terms of the expense can be combined and written as

$$\text{Cost}_{dn}(n) = C^2 \sum_{k=0}^{N_{\sigma\tau}} \Big({N_c}^{n-1}(N_c - k) + k({N_\sigma}^{n-1}{N_u}^n + {N_\tau}^{n-1}{N_w}^n)\Big) + \sum_{k=0}^{N_{\sigma\tau}} \text{Cost}_{dn}(n-1)$$

where the expenses of steps outside the loop have been combined with those inside as the $k = 0$ term of the sum, and the expenses of steps 5.1 and 5.5 have been dropped as they are of lower order than that of step 5.6.

We note here that evaluation of the first sum is straightforward, as the terms linear in k can be replaced by their average values without affecting the result, so it is easily shown to sum to $\mathcal{O}(N_{\sigma\tau}{N_c}^n + {N_{\sigma\tau}}^2({N_\sigma}^{n-1}{N_u}^n + {N_\tau}^{n-1}{N_w}^n))$. Given that the expense of the base case is $\text{Cost}_{dn}(1) = \mathcal{O}(C^2N_c + CN_{uw}N_c)$ a straightforward induction proof can be used to show that the expense is dominated by the highest level of the recursion, giving

$$\mathrm{Cost}_{dn}(n) = \mathcal{O}(C^2(N_{\sigma\tau}{N_c}^n + {N_{\sigma\tau}}^2({N_\sigma}^{n-1}{N_u}^n + {N_\tau}^{n-1}{N_w}^n))).$$

A further simplification results from closer examination of the ${N_c}^n$ term in the above expression. Since we know that $N_c = \mathcal{O}(N_{uw})$ then ${N_c}^n = \mathcal{O}({N_{uw}}^n)$, so the expense associated with that term is dominated by the other, giving us the simplified cost

$$\mathrm{Cost}_{dn}(n) = \mathcal{O}(C^2{N_{\sigma\tau}}^2({N_\sigma}^{n-1}{N_u}^n + {N_\tau}^{n-1}{N_w}^n)). \tag{6.8}$$

Dense Zero

Now we look at the overall algorithm for the dense case when the evaluation point α_n is zero. The evaluations mod $\langle x_n - 0\rangle$ in step 2 simply involve a scan of the existing terms, only retaining those not containing x_n, with associated expense $\mathcal{O}({N_u}^n + {N_w}^n + {N_c}^n)$. Using the same reasoning as for the dense case, we can simplify this to $\mathcal{O}({N_c}^n)$. The recursive call in step 3 has expense $\mathrm{Cost}_{dz}(n-1)$, and the computation of the error in step 4 has the same expense as the non-zero case, namely $\mathcal{O}(C^2({N_\sigma}^{n-1}{N_u}^n + {N_\tau}^{n-1}{N_w}^n))$.

The loop in step 5 is repeated $N_{\sigma\tau}$ times. The monomial multiplication by x_n in step 5.1 is trivial, and the coefficient extraction in step 5.2 only requires a term scan of the error, having expense bounded by $\mathcal{O}({N_c}^n(N_c - k))$. Again we proceed to step 5.4 with expense $\mathrm{Cost}_{dz}(n-1)$. The monomial multiplication in step 5.5 has an expense of $\mathcal{O}({N_u}^{n-1} + {N_w}^{n-1})$. Again, the error update is the dominant term in step 5.6, but in this case the s, t used in the update only have $\mathcal{O}({N_\sigma}^{n-1})$ and $\mathcal{O}({N_\tau}^{n-1})$ terms respectively, so the cost is $\mathcal{O}(C^2({N_\sigma}^{n-1}{N_w}^n + {N_\tau}^{n-1}{N_u}^n))$.

The various terms of the expense can be combined and written as

$$\mathrm{Cost}_{dz}(n) = \sum_{k=0}^{N_{\sigma\tau}} \left({N_c}^{n-1}(N_c - k) + C^2({N_\sigma}^{n-1}{N_u}^n + {N_\tau}^{n-1}{N_w}^n)\right) + \sum_{k=0}^{N_{\sigma\tau}} \mathrm{Cost}_{dz}(n-1)$$

where the expenses of steps outside the loop have again been combined with those inside as the $k = 0$ term of the sum, and the expenses of steps 5.1 and 5.5 have again been dropped as they are of lower order than that of step 5.6.

The first sum has a value of $\mathcal{O}(N_{\sigma\tau}{N_c}^n + C^2N_{\sigma\tau}({N_\sigma}^{n-1}{N_u}^n + {N_\tau}^{n-1}{N_w}^n))$, which we simplify to $\mathcal{O}(C^2N_{\sigma\tau}({N_\sigma}^{n-1}{N_u}^n + {N_\tau}^{n-1}{N_w}^n))$ using an argument similar to the one in the dense non-zero case. Now since the expense of the base case is given by $\mathrm{Cost}_{dz}(1) = \mathcal{O}(C^2 N_c + C N_{uw} N_c)$ a straightforward induction proof can be used to show that the expense is again dominated by the highest level of the recursion, giving

$$\mathrm{Cost}_{dz}(n) = \mathcal{O}(C^2 N_{\sigma\tau}({N_\sigma}^{n-1} {N_u}^n + {N_\tau}^{n-1} {N_w}^n)). \tag{6.9}$$

Sparse Non-zero

Now we look at the overall algorithm for the sparse case when the evaluation point α_n is non-zero. This is the more interesting case, as it occurs often in practice.

We note that even if the inputs are sparse, the intermediate σ, τ computed in a single invocation of the algorithm become dense in the current variable. This is a direct result of the non-zero evaluation point and the fact that an approximation of a single monomial, say x_2^j, will be represented as a dense polynomial in x_2 until degree j is reached, at which point the dense polynomial will collapse to x_2^j. Upon completion of the computation, the σ, τ returned from the algorithm are in fact sparse (we have completed the computation out to the highest degree, so all approximations have collapsed to the correct sparse terms). For the same reason, the error term, though initially sparse, grows in size as the computation proceeds, becoming dense in the current variable fairly rapidly, then reduces in size (but slowly) as k approaches $N_{\sigma\tau}$ (as a greater number of terms are correctly computed). This indicates that we may have a significant variation of the complexity of the algorithm based on whether most of the terms are of a low degree in the current variable (least expensive case), or of a high degree in the current variable (most expensive case).

We will proceed with 2 analyses for this case, the first of which we will call the typical case. Here the terms are roughly evenly distributed with respect to the degree of the current variable. In the second case, which we will call the special case, we assume that only one of the terms is of degree $\mathcal{O}(N)$.

For either case, the expense of step 2 is $\mathcal{O}(C^2(T_u + T_w + T_c)) = \mathcal{O}(C^2 T_c)$, where we assume that the powers α_n^i have been previously computed and stored. The expense of the recursive call is given by $\mathrm{Cost}_{sn}(n-1, T_c, T_\sigma, T_\tau)$. Note that for this estimate we are including the term counts for c, σ, and τ because they are not necessarily constant. Use of T_c,T_σ, and T_τ is viable for the first call, as for sufficiently sparse polynomials use of a non-zero evaluation point will not significantly reduce the term counts.

The expense of step 4 will be $\mathcal{O}(C^2(T_\sigma T_u + T_\tau T_w))$, and the loop will repeat $N_{\sigma\tau}$ times, with step 5.1 having expense $\mathcal{O}(C^2k)$, but now the analyses start to diverge, as the expense of the remaining steps will depend on the structure of the solution σ,τ.

For the typical case, the number of terms for e and c_k can be bounded by $\mathcal{O}(T_ck)$, $\mathcal{O}(T_c)$ terms in iteration k respectively. Since we have a non-zero evaluation point, it is unlikely that $c_k = 0$, so we proceed to the recursive call with expense $\mathrm{Cost}_{sn}(n-1, T_c, T_\sigma, T_\tau)$.

The number of terms in the diophantine solutions s,t computed in step 5.4 are bounded by T_σ,T_τ respectively, so the expense of step 5.5 is $\mathcal{O}(C^2k(T_\sigma + T_\tau))$, and the expense of step 5.6 is dominated by the error update with expense $\mathcal{O}(C^2k(T_\sigma T_u + T_\tau T_w))$.

Combining, performing the obvious sum and simplifying, we obtain

$$\mathrm{Cost}_{sn}(n, T_c, T_\sigma, T_\tau) = \mathcal{O}(C^2 {N_{\sigma\tau}}^2 (T_\sigma T_u + T_\tau T_w)) + \sum_{k=0}^{N_{\sigma\tau}} \mathrm{Cost}_{sn}(n-1, T_c, T_\sigma, T_\tau)$$

where we use $T_c = \mathcal{O}(T_u T_\sigma + T_w T_\tau)$, again combine expenses outside and inside the loop, and again drop the expense of steps 5.1, 5.5, as they are lower order than that of step 5.6.

Now use of the base case $\mathrm{Cost}_{sn}(1, T_c, T_\sigma, T_\tau) = \mathcal{O}(C^2T_c + CN_{uw}T_c)$ and solution of the recurrence gives the asymptotic expense as

$$\mathrm{Cost}_{sn}(n, T_c, T_\sigma, T_\tau) = \mathcal{O}(C^2 {N_{\sigma\tau}}^n T_c + C {N_{\sigma\tau}}^{n-1} N_{uw} T_c), \tag{6.10}$$

where we note that in this case, unlike the dense cases, the primary contribution to the expense comes from the recursive call (for $n > 2$).

For the special case, the sizes for the error and c_k can be approximated by $\mathcal{O}(k)$, $\mathcal{O}(1)$ terms in iteration k respectively. Again by the same reasoning as the typical case, we proceed to the recursive call with expense $\mathrm{Cost}_{sn}(n-1, 1, 1, 1)$.

Step 5.5 has expense $\mathcal{O}(C^2k)$, and the expense of step 5.6 is dominated by the error update with expense $\mathcal{O}(C^2k(T_u + T_w))$.

Combining, performing the obvious sum and simplifying, we obtain

$$\begin{aligned}\mathrm{Cost}_{sn}(n, T_c, T_\sigma, T_\tau) &= \mathcal{O}(C^2(T_\sigma T_u + T_\tau T_w) + C^2 {N_{\sigma\tau}}^2 (T_u + T_w)) \\ &+ \mathrm{Cost}_{sn}(n-1, T_c, T_\sigma, T_\tau) + \sum_{k=1}^{N_{\sigma\tau}} \mathrm{Cost}_{sn}(n-1, 1, 1, 1)\end{aligned}$$

where we have eliminated negligible expenses, and also used $T_c = \mathcal{O}(T_u T_\sigma + T_w T_\tau)$.

Now with use of the base case $\text{Cost}_{sn}(1, T_c, T_\sigma, T_\tau) = \mathcal{O}(C^2 T_c + C N_{uw} T_c)$ it is possible to show that the expense is given by

$$\text{Cost}_{sn}(n, T_c, T_\sigma, T_\tau) = \mathcal{O}(C^2 {N_{\sigma\tau}}^n + C N_{uw} {N_{\sigma\tau}}^{n-1} + C(nC + N_{uw})T_c + C^2(n + N_{\sigma\tau}) N_{\sigma\tau} T_{uw})).$$

For this result, it is likely that the first two terms describe the dominant expense, but the remaining terms, which likely drop out for some small n, are retained to indicate that the number of terms do have an effect on the complexity.

It is also worthy of note that comparison of the expenses of the typical and special cases shows primarily a difference in how T_c enters into the complexity, and for problems where T_c is quite small, the complexity is essentially the same. This makes sense given that for a very small T_c there will be very few terms with degree > 1 (in fact very few terms at all) for the problem, so we expect the similarity.

Sparse Zero

Now we look at the overall algorithm for the sparse case when the evaluation point α_n is zero. This is the case where the algorithm has greatest efficiency. For this case, the analysis needs to be the most precise for the result to accurately reflect the complexity of the algorithm.

Since we are in the sparse zero case, the sum of the terms of the Taylor coefficients of σ,τ being computed will be equal to the total number of terms of σ,τ upon completion of the algorithm. Furthermore, due to the fact that the recursive calls in the algorithm are always utilizing u_0 and w_0, the c_k being used for recursive calls of the algorithm will be given by $\sigma^k u_0 + \tau^k w_0$, where σ^k,τ^k use the notation of I-adic lifting described in §6.7.3, or explicitly $\sigma = \sigma^0 + x_n\sigma^1 + x_n^2\sigma^2 + x_n^3\sigma^3 + ...$, and $\tau = \tau^0 + x_n\tau^1 + x_n^2\tau^2 + x_n^3\tau^3 + ...$

The important observation here is that since we are using a zero evaluation point, the σ^k,τ^k will only contain the terms in σ,τ that are degree k in x_n. To utilize this information, we will use the notation $T_{\sigma,k}$ to represent the number of terms in σ^k, and $T_{\tau,k}$ to represent the number of terms in τ^k. We apply a similar notation for u,w, but for c we use the notation $T_{c,k}$ to represent the number of terms in c_k *in the kth iteration*, which will be bounded by $T_{\sigma,k}T_{u,0} + T_{\tau,k}T_{w,0}$.

The values of u_0, v_0 computed in step 2 of the algorithm are already known, so we need only concern ourselves with the computation of c_0 which has expense $\mathcal{O}(T_c)$ for the term scan. The recursive call in step 3 will have expense $\mathrm{Cost}_{sz}(n-1, T_{c,0}, T_{\sigma,0}, T_{\tau,0})$, and the error computation in step 4 will have expense $\mathcal{O}(C^2(T_{\sigma,0}T_u + T_{\tau,0}T_w))$.

Now the loop in step 5 will be executed $N_{\sigma\tau}$ times, though not all steps of the loop are necessarily executed in each pass. The monomial multiplication by x_k in step 5.1 is trivial, and the coefficient extraction in step 5.2 only requires a term scan of the error, having expense bounded by $\mathcal{O}(T_c)$. In the event that the coefficient c_k computed in step 5.2 is non-zero, we will proceed to 5.4, with associated expense $\mathrm{Cost}_{sz}(n-1, T_{c,k}, T_{\sigma,k}, T_{\tau,k})$. The cost in step 5.5 is $\mathcal{O}(T_{\sigma,k} + T_{\tau,k})$ (which can be ignored as we will see it is less expensive than step 5.6), and finally the expense in step 5.6 is dominated by the products involved in the error computation with an expense of $\mathcal{O}(C^2(T_{\sigma,k}T_u + T_{\tau,k}T_w))$.

As our first simplification, we will claim that the conditional execution of steps 5.4-5.6 can be ignored because for the cases in which those steps are not run we need $\sigma^k = 0$ and $\tau^k = 0$, and as we will see the expense of those steps will be zero for $T_{c,k} = 0, T_{\sigma,k} = 0, T_{\tau,k} = 0$.

We can combine the expense of step 4 and all iterations of step 5.6 into the sum

$$\begin{aligned}
\sum_{k=0}^{N_{\sigma\tau}} \mathcal{O}(C^2(T_{\sigma,k}T_u + T_{\tau,k}T_w)) &= \mathcal{O}(C^2\left(\left(\sum_{k=0}^{N_{\sigma\tau}} T_{\sigma,k}\right) T_u + \left(\sum_{k=0}^{N_{\sigma\tau}} T_{\tau,k}\right) T_w\right)) \\
&= \mathcal{O}(C^2(T_\sigma T_u + T_\tau T_w)),
\end{aligned}$$

where we see that the contribution of 5.6 to the expense is indeed zero for $T_{\sigma,k} = 0, T_{\tau,k} = 0$. We can also combine the expense of steps 3 and 5.4 into a single sum, and combining this with the result above and including all remaining steps gives the complexity estimate

$$\mathrm{Cost}_{sz}(n, T_c, T_\sigma, T_\tau) = \mathcal{O}(C^2(T_\sigma T_u + T_\tau T_w) + N_{\sigma\tau}T_c) + \sum_{k=0}^{N_{\sigma\tau}} \mathrm{Cost}_{sz}(n-1, T_{c,k}, T_{\sigma,k}, T_{\tau,k}),$$

where we see that our claim that the conditional execution of steps 5.4-5.6 is taken care of will be true as long as $\mathrm{Cost}_{sz}(n-1, 0, 0, 0) = 0$.

Combining with the base case $\mathrm{Cost}_{sn}(1, T_c, T_\sigma, T_\tau) = \mathcal{O}(C^2T_c + CT_cN_{uw})$, we see that all terms in the base case and the recursion are proportional to one of T_c, T_σ, T_τ, so the conditional execution of steps 5.4-5.6 is taken care of, and we will now state and prove the solution.

Theorem 14 *Using the base case* $\mathrm{Cost}_{sn}(1,T_c,T_\sigma,T_\tau)=\mathcal{O}(C^2T_c+CT_cN_{uw})$ *and the recurrence* $\mathrm{Cost}_{sz}(n,T_c,T_\sigma,T_\tau)=\mathcal{O}(C^2(T_\sigma T_u+T_\tau T_w)+N_{\sigma\tau}T_c)+\sum_{k=0}^{N_{\sigma\tau}}\mathrm{Cost}_{sz}(n-1,T_{c,k},T_{\sigma,k},T_{\tau,k})$, *and all preceding definitions for T,N, the expense of the multivariate diophantine algorithm for the sparse zero case is given by*

$$\mathrm{Cost}_{sn}(n,T_c,T_\sigma,T_\tau)=\mathcal{O}(nC^2T_c+(n-1)N_{\sigma\tau}T_c+CN_{uw}T_c). \tag{6.11}$$

Proof

We will prove this by induction. Since the base case ($n=1$) is in agreement with the expense (6.11) we need only show that it is true for $n+1$ given that it holds for n. Consider the sum in the recurrence, we have

$$\begin{aligned}
\sum_{k=0}^{N_{\sigma\tau}}\mathrm{Cost}_{sz}(n,T_{c,k},T_{\sigma,k},T_{\tau,k}) &= \sum_{k=0}^{N_{\sigma\tau}}\mathcal{O}((nC^2+(n-1)N_{\sigma\tau}+CN_{uw})T_{c,k}) \\
&= \mathcal{O}((nC^2+(n-1)N_{\sigma\tau}+CN_{uw})\sum_{k=0}^{N_{\sigma\tau}}(T_{\sigma,k}T_{u,0}+T_{\tau,k}T_{w,0})) \\
&= \mathcal{O}((nC^2+(n-1)N_{\sigma\tau}+CN_{uw})(T_\sigma T_{u,0}+T_\tau T_{w,0})) \\
&\le \mathcal{O}((nC^2+(n-1)N_{\sigma\tau}+CN_{uw}))(T_\sigma T_u+T_\tau T_w)) \\
&= \mathcal{O}((nC^2+(n-1)N_{\sigma\tau}+CN_{uw})T_c)
\end{aligned}$$

where we have used $T_\sigma T_u+T_\tau T_w=\mathcal{O}(T_c)$ in the first step, $\sum_{k=0}^{N_{\sigma\tau}}T_{\sigma,k}=T_\sigma$ and $\sum_{k=0}^{N_{\sigma\tau}}T_{\tau,k}=T_\tau$ in the third step, $T_{u,0}\le T_u$ and $T_{w,0}\le T_w$ in the fourth step, and $T_\sigma T_u+T_\tau T_w=\mathcal{O}(T_c)$ in the last step.

Utilizing this result in the recurrence gives

$$\begin{aligned}
\mathrm{Cost}_{sn}(n+1,T_c,T_\sigma,T_\tau) &= \mathcal{O}((C^2+N_{\sigma\tau})T_c)+\mathcal{O}((nC^2+(n-1)N_{\sigma\tau}+CN_{uw})T_c) \\
&= \mathcal{O}((n+1)C^2T_c+nN_{\sigma\tau}T_c+CN_{uw}T_c),
\end{aligned}$$

as required, so by induction on n the result is proved. □

Sparse Partial Zero

It is often the case that zero evaluation points can be used for some of the variables, while non-zero evaluation points must be used for others. This case is not covered by the prior two analyses, but the form of the complexity for the sparse non-zero and sparse zero cases is quite suggestive.

We call the number of variables for which we can use a zero evaluation point n_0, so naturally the number of variables for which we require a non-zero evaluation point is given by $n - n_0 - 1$. Since we already have the recurrences and results for the sparse non-zero and zero cases they can be applied in a number of ways to obtain the result for the mixed case. In proceeding with the analysis it was discovered that the asymptotic complexity for the case where the variables are arranged so that $x_{n-n0+1}, ..., x_n$ utilize zero evaluation points is better than for the case where we arrange the variables so that $x_2, ..., x_{n_0+1}$ utilize zero evaluation points.

The result for $x_{n-n0+1} = 0, ..., x_n = 0$, stated without proof is

$$\text{Cost}_{sn_0}(n, T_c, T_\sigma, T_\tau) = \mathcal{O}(C{N_{\sigma\tau}}^{n-n_0-1}(CN_{\sigma\tau} + N_{uw})T_c + n_0(C^2 + N_{\sigma\tau})T_c), \qquad (6.12)$$

and the result for $x_2 = 0, ..., x_{n_0+1} = 0$ is

$$\text{Cost}_{sn_0}(n, T_c, T_\sigma, T_\tau) = \mathcal{O}(C{N_{\sigma\tau}}^{n-n_0-1}(CN_{\sigma\tau} + N_{uw})T_c + {N_{\sigma\tau}}^{n-n_0-1}n_0(C^2 + N_{\sigma\tau})T_c),$$

where we note that in the latter case, the exponential behavior with respect to $N_{\sigma\tau}$ is coupled with the expense from the sparse computation, while in the former it is not.

We will utilize the first (better) result, and we note that although it is in exact agreement with the result for the sparse non-zero case for $n_0 = 0$, it is not in agreement with the result for the sparse zero case for $n_0 = n - 1$, and this can be explained by the fact that we are using the sparse non-zero case as a base case, and it is only valid for $n \geq 2$.

Summary

We summarize our results for the analysis of the multivariate diophantine algorithm in Table 6.6, where all but the base cases are only valid for $n \geq 2$, and the sparse mixed case is only valid for $n_0 < n - 1$.

Dense Base	$\mathcal{O}(C^2N_c + CN_{uw}N_c)$
Dense Non-zero	$\mathcal{O}(C^2{N_{\sigma\tau}}^2({N_\sigma}^{n-1}{N_u}^n + {N_\tau}^{n-1}{N_w}^n))$
Dense Zero	$\mathcal{O}(C^2N_{\sigma\tau}({N_\sigma}^{n-1}{N_u}^n + {N_\tau}^{n-1}{N_w}^n))$
Sparse Base	$\mathcal{O}(C^2T_c + CN_{uw}T_c)$
Sparse Non-zero	$\mathcal{O}(C^2{N_{\sigma\tau}}^nT_c + C{N_{\sigma\tau}}^{n-1}N_{uw}T_c)$
Sparse Zero	$\mathcal{O}(nC^2T_c + (n-1)N_{\sigma\tau}T_c + CN_{uw}T_c)$
Sparse Mixed (n_0)	$\mathcal{O}(C{N_{\sigma\tau}}^{n-n_0-1}(CN_{\sigma\tau} + N_{uw})T_c + n_0(C^2 + N_{\sigma\tau})T_c)$

Table 6.6: Multivariate Diophantine Solution Expense

Complexity of *EEZ-GCD* - monic inputs

We now have the foundation necessary to present and analyze the *EEZ-GCD* algorithm for the case where the inputs are *leading monic* in the main variable. The simplified analysis presented here assumes that the prime and evaluations do not introduce a bad zero and are not unlucky.

Algorithm 25 (EEZMonic)

Input: *Polynomials* $a, b \in \mathbf{Z}[x_1, ..., x_n]$ *for which we want to compute the* GCD g. *A prime* p *and power* l *so that* p^l *is larger than twice the magnitude of the largest coefficient in* a. *A set of evaluations* $I = \langle x_2 - \alpha_2, ..., x_n - \alpha_n \rangle$.

Output: $g, \bar{a} \in \mathbf{Z}[x_1, ..., x_n]$ *such that* $g = \mathrm{GCD}(a, b)$ *and* $a = \bar{a}g$, *or* **Fail**.

1 *Compute images of inputs, Set* $a_1 = a \bmod I$, $a_0 = a_1 \bmod p$, $b_0 = b \bmod \langle I, p \rangle$

2 *Compute* GCD *in* $\mathbf{Z}_p[x_1]$, *Set* $u_0 = \mathrm{GCD}(a_0, b_0) \pmod p$, $w_0 = \frac{a_0}{u_0} \bmod p$
If $\mathrm{GCD}(u_0, w_0) \neq 1$ *then return* **Fail**.

3 *Find* $u, w \in \mathbf{Z}_{p^l}[x_1]$ *such that* $a_1 - uw \equiv 0 \pmod{p^l}$ *using the univariate Hensel algorithm (23)*

4 *Loop* i *from 2 to* n

4.1 *Set* $u_i = u, w_i = w$

4.2 *Compute* $a_i = a \bmod \langle x_{i+1} - \alpha_{i+1}, ..., x_n - \alpha_n, p^l \rangle$

4.3 *Initialize, Set* $e = a_i - uw \bmod p^l$, $m = 1$

4.4 *Loop* k *from* 1 *to* $\deg_{x_i}(a)$ *while* $e \neq 0$

4.4.1 *Set* $m = m \times (x_i - \alpha_i) \bmod p^l$

4.4.2 *Set* c_k *to* $(x_n - \alpha_n)^k$ *coefficient of the taylor expansion of* e *about* $x_n = \alpha_n$.

4.4.3 *If* $c_k = 0$ *then next loop*

4.4.4 *Solve* $\sigma u_i + \tau w_i = c_k$ *for* $\sigma, \tau \in \mathbf{Z}_{p^l}[x_1, ..., x_{i-1}]$ *using the MultiDio algorithm (24), and on failure return* **Fail**.

4.4.5 *Set* $\sigma = m\sigma \bmod p^l$, $\tau = m\tau \bmod p^l$

4.4.6 *Update error and solution*

$$\text{Set } e = e - \sigma u - \tau w - \sigma\tau \bmod p^l,\ \ u = u + \tau \bmod p^l,\ \ w = w + \sigma \bmod p^l$$

4.5 *If* $e \neq 0$ *then return* **Fail**.

5 *Apply division test to assure that* u *divides* a *and* b*, if not return* **Fail**.

6 *Return* u, w

Before beginning the analysis, we will briefly describe some aspects of the algorithm. The failure in step 2 has two possible causes: it may be a bad zero, or it may be that $\bar{a}, g$ have a common factor. In the latter case we need to either use b for the reconstruction, or if the same problem occurs then we must use some linear combination of a and b. The failures in steps 4.4.4 and 4.5 catch the unlucky cases where the additional factor appearing in the GCD is not actually a factor of a in $\mathbf{Z}[x_1, ..., x_n]$. The division test in step 5 catches the other unlucky case where the additional factor is an actual factor of a in $\mathbf{Z}[x_1, ..., x_n]$. In addition, we need to apply the division test to a also, as we are choosing a value of l that is sufficient to reconstruct the coefficients of a only (ignoring the effect of the non-zero expansion points). Note that bad evaluations are not a concern here, as the inputs are leading monic in the main variable.

Again, the analysis needs to split into the four combinations, dense or sparse with non-zero or zero evaluation points. As in the prior analyses we use subscripted forms for the maximum degree (N), the number of terms (T) and the coefficient length (C) for $g, \bar{a}, \bar{b}$. Clearly we want to choose a to have the smaller cofactor, and we use the notation $C = C_{g\bar{a}} = \mathcal{O}(l)$.

Dense Non-zero

We begin the analysis with the internals of the loop at step 4. Before performing the analysis, however, we look at the sizes of the expressions involved in the computation.

At the start of each loop of 4, the u,w are dense in $i-1$ variables, so they have $\mathcal{O}({N_g}^{i-1})$,$\mathcal{O}({N_{\bar{a}}}^{i-1})$ terms, respectively. The error is initially dense in i variables, so it starts with $\mathcal{O}((N_g+N_{\bar{a}})^i)$ terms, and as each iteration of the inner loop 4.4 zeros another error coefficient, this drops as $\mathcal{O}((N_g+N_{\bar{a}}-k)(N_g+N_{\bar{a}})^{i-1}) = \mathcal{O}((N_{g\bar{a}}-k){N_{g\bar{a}}}^{i-1})$. In addition, the number of terms in c_k will be given by the lower order term of the remaining error, which is $\mathcal{O}((N_g+N_{\bar{a}})^{i-1}) = \mathcal{O}({N_{g\bar{a}}}^{i-1})$. In the loop 4.4, u,w grow from being dense in $i-1$ variables to dense in i variables, so a suitable bound on their terms in the kth loop is given by $\mathcal{O}(k{N_g}^{i-1})$,$\mathcal{O}(k{N_{\bar{a}}}^{i-1})$ for u,w respectively.

We now proceed with the analysis of the loop. First we assume that we have computed and retained the value of $a_i \bmod p^l$ outside of the loop to provide better efficiency (discussed later), so there is no cost for step 4.2 inside the loop, and the computation of the error in step 4.3 has expense $\mathcal{O}(C^2{N_g}^{i-1}{N_{\bar{a}}}^{i-1})$ based on the initial sizes of u,w.

The loop 4.4 will repeat $N_{g\bar{a}}$ times, and step 4.4.1 will have expense $\mathcal{O}(C^2k)$ as m is a dense polynomial of degree k in x_i. The extraction of the coefficient in step 4.4.2 will have expense $\mathcal{O}(C^2(N_{g\bar{a}}-k){N_{g\bar{a}}}^{i-1})$ as every term remaining in the error will have a contribution due to the non-zero evaluation point. We will clearly proceed past step 4.4.3, as the input is dense, so the expense of the call to the multivariate diophantine algorithm in step 4.4.4 (Table 6.6) is $\mathcal{O}(C^2{N_{g\bar{a}}}^2({N_{\bar{a}}}^{i-1}{N_g}^i+{N_g}^{i-1}{N_{\bar{a}}}^i)) = \mathcal{O}(C^2{N_{g\bar{a}}}^3{N_{\bar{a}}}^{i-1}{N_g}^{i-1})$. The expense of step 4.4.5 will be given by $\mathcal{O}(C^2k({N_g}^{i-1}+{N_{\bar{a}}}^{i-1}))$, and will enlarge σ,τ to have $\mathcal{O}(k{N_{\bar{a}}}^{i-1})$,$\mathcal{O}(k{N_g}^{i-1})$ terms respectively. Step 4.4.6 will be dominated by the error update, where all three products have an expense of $\mathcal{O}(C^2k^2{N_{\bar{a}}}^{i-1}{N_g}^{i-1})$.

Combining these expenses, and dropping the obvious lower order terms gives

$$\begin{aligned}\mathrm{Cost}_{4i} &= \mathcal{O}(C^2{N_g}^{i-1}{N_{\bar{a}}}^{i-1}) + \sum_{k=1}^{N_{g\bar{a}}}\mathcal{O}(C^2(N_{g\bar{a}}-k){N_{g\bar{a}}}^{i-1}) \\ &\quad + \sum_{k=1}^{N_{g\bar{a}}}\mathcal{O}(C^2{N_{g\bar{a}}}^3{N_{\bar{a}}}^{i-1}{N_g}^{i-1}) + \sum_{k=1}^{N_{g\bar{a}}}\mathcal{O}(C^2k^2{N_{\bar{a}}}^{i-1}{N_g}^{i-1}).\end{aligned}$$

We can compute the sums in the above to obtain

$$\begin{aligned}\mathrm{Cost}_{4i} &= \mathcal{O}(C^2(N_gN_{\bar{a}})^{i-1} + C^2{N_{g\bar{a}}}^{i+1} + C^2{N_{g\bar{a}}}^4(N_gN_{\bar{a}})^{i-1} + C^2{N_{g\bar{a}}}^3(N_{\bar{a}}N_g)^{i-1})\\ &= \mathcal{O}(C^2{N_{g\bar{a}}}^4(N_gN_{\bar{a}})^{i-1}),\end{aligned}$$

where we observe that the third term in the error, the one corresponding to the expense of the multivariate diophantine algorithm, is asymptotically larger than the others.

Now for the analysis of the entire algorithm, consider the evaluations in step 1. As mentioned earlier, we would like to precompute the required values of a_i needed in step 4.2 of the algorithm, both for this algorithm and for the multivariate diophantine algorithm. This is a natural fit, as we already need to perform these evaluations (at least in some form) for step 1 of the algorithm, in preparation for computing the required GCD. We would proceed by initially computing $a \bmod p^l$, but this is unnecessary, as p^l has been chosen so that $a \bmod p^l = a$. Now we loop through, evaluating out one variable at a time, with expense $\mathcal{O}(C^2{N_{g\bar{a}}}^i)$ for evaluation with respect to x_i, in the end obtaining a_1, which is required for the univariate Hensel algorithm. As a final step, we evaluate $\bmod\ p$ obtaining the a_0 needed for the GCD computation. This entire process has expense bounded by $\mathcal{O}(C^2{N_{g\bar{a}}}^n)$, the expense of the first evaluation. The required computation for b, however, can be done a bit more efficiently as the entire process, except the initial mod, can be computed in $\mathbf{Z}_p[...]$, so the expense for this is simply given by $\mathcal{O}(C_{g\bar{b}}{N_{g\bar{b}}}^n)$.

This precomputation of the a_i significantly reduces the expense of step 4.2 of the algorithm, for if one were to proceed as the algorithm is written, evaluations would need to be performed multiple times, so the overall expense would be at least $\mathcal{O}(nC^2{N_{g\bar{a}}}^i)$, adding a factor of n to the expense of all executions of that step, so even if this does not affect the complexity of the result, it is a useful optimization for an implementation.

The GCD in step 2 has expense $\mathcal{O}(N_{g\bar{a}}N_{g\bar{b}})$, the univariate Hensel algorithm in step 3 has expense $\mathcal{O}(C{N_{g\bar{a}}}^2 + C^2N_gN_{\bar{a}})$, and we denote the cost of the division in step 5 by $D_d(n)$.

We now have all information necessary to estimate the cost of the algorithm for this case, which (dropping obvious lower order terms) is given by

$$\begin{aligned}\mathrm{Cost}_{dn} &= \mathcal{O}(C^2{N_{g\bar{a}}}^n + C_{g\bar{b}}{N_{g\bar{b}}}^n + N_{g\bar{a}}N_{g\bar{b}} + C{N_{g\bar{a}}}^2 + C^2N_gN_{\bar{a}})\\ &\quad + D_d(n) + \sum_{i=2}^{n}\mathcal{O}(C^2{N_{g\bar{a}}}^4(N_gN_{\bar{a}})^{i-1}).\end{aligned}$$

It is straightforward to see that for the terms in the sum, the one for $i = n$ will be asymptotically largest, having an expense of $\mathcal{O}(C^2 {N_{g\bar{a}}}^4 (N_g N_{\bar{a}})^{n-1})$, and in addition this term is larger than any of the other terms not involving b. If we also apply the result for the dense division $D_d(n)$ from Table 6.3, we obtain

$$\mathrm{Cost}_{dn} = \mathcal{O}(C_g C_{\bar{b}} (N_g N_{\bar{b}})^n + N_{g\bar{a}} N_{g\bar{b}} + C^2 {N_{g\bar{a}}}^4 (N_g N_{\bar{a}})^{n-1}). \tag{6.13}$$

Dense Zero

As with the prior analysis, we begin with the internals of the loop at step 4. The sizes of the u,w, the error, and the coefficient c_k are identical to those discussed at the start of the dense non-zero analysis, so we will not repeat the information here.

As with the non-zero case, we precompute the required a_i values, so step 4.2 is accounted for, and the computation of the initial error in 4.3 has expense $\mathcal{O}(C^2 {N_g}^{i-1} {N_{\bar{a}}}^{i-1})$.

The loop 4.4 will repeat $N_{g\bar{a}}$ times, and the monomial product in step 4.4.1 will have negligible expense. The extraction of the coefficient in step 4.4.2 will have expense $\mathcal{O}((N_g + N_{\bar{a}} - k)(N_g + N_{\bar{a}})^{i-1}) = \mathcal{O}((N_{g\bar{a}} - k){N_{g\bar{a}}}^{i-1})$ as we need only perform a scan of the remaining terms in the error e. We will again proceed past step 4.4.3, as the input is dense, so the expense of the call to the multivariate diophantine algorithm in step 4.4.4 (Table 6.6) is $\mathcal{O}(C^2 N_{g\bar{a}} ({N_{\bar{a}}}^{i-1} {N_g}^i + {N_g}^{i-1} {N_{\bar{a}}}^i)) = \mathcal{O}(C^2 {N_{g\bar{a}}}^2 (N_g N_{\bar{a}})^{i-1})$. The expense of step 4.4.5 will be given by $\mathcal{O}({N_g}^{i-1} + {N_{\bar{a}}}^{i-1})$, but unlike the non-zero case, the size of σ,τ will be unchanged. Step 4.4.6 will be dominated by the error update, where the first two products have an expense of $\mathcal{O}(C^2 k (N_{\bar{a}} N_g)^{i-1})$, and the third product has a lower order expense.

Combining these expenses, and dropping the obvious lower order terms gives

$$\begin{aligned}
\mathrm{Cost}_{4i} &= \mathcal{O}(C^2 {N_g}^{i-1} {N_{\bar{a}}}^{i-1}) + \sum_{k=1}^{N_{g\bar{a}}} \mathcal{O}((N_{g\bar{a}} - k){N_{g\bar{a}}}^{i-1}) + \sum_{k=1}^{N_{g\bar{a}}} \mathcal{O}(C^2 {N_{g\bar{a}}}^2 (N_g N_{\bar{a}})^{i-1}) \\
&\quad + \sum_{k=1}^{N_{g\bar{a}}} \mathcal{O}(C^2 k (N_{\bar{a}} N_g)^{i-1}),
\end{aligned}$$

and computing the sums we obtain

$$\begin{aligned}
\mathrm{Cost}_{4i} &= \mathcal{O}(C^2 N_{g\bar{a}} (N_g N_{\bar{a}})^{i-1} + {N_{g\bar{a}}}^{i+1} + C^2 {N_{g\bar{a}}}^3 (N_g N_{\bar{a}})^{i-1} + C^2 {N_{g\bar{a}}}^2 (N_{\bar{a}} N_g)^{i-1}) \\
&= \mathcal{O}(C^2 {N_{g\bar{a}}}^3 (N_g N_{\bar{a}})^{i-1}),
\end{aligned}$$

where again the third term in the error, the one corresponding to the expense of the multivariate diophantine algorithm, is asymptotically larger than the others.

Now for the analysis of the entire algorithm, consider the evaluations in step 1. As for the dense non-zero case we will precompute the required values of a_i needed in step 4.2 of the algorithm. In contrast to the dense non-zero case, we need not perform any coefficient arithmetic for the computation of a_i, and can actually obtain all required values in a single pass over a with expense $\mathcal{O}({N_{g\bar{a}}}^n)$. Finally evaluating $a_1 \bmod p$ does require coefficient arithmetic, but only $\mathcal{O}(C)$ and only for $\mathcal{O}(N_{g\bar{a}})$ terms, so the total expense for all evaluations of a is given by $\mathcal{O}({N_{g\bar{a}}}^n + CN_{g\bar{a}})$. Analysis of the required computation for b is identical, with total expense $\mathcal{O}({N_{g\bar{b}}}^n + C_{g\bar{b}}N_{g\bar{b}})$.

The GCD in step 2 has expense $\mathcal{O}(N_{g\bar{a}}N_{g\bar{b}})$, the univariate Hensel algorithm in step 3 has expense $\mathcal{O}(C{N_{g\bar{a}}}^2 + C^2N_gN_{\bar{a}})$, and we denote the cost of the division in step 5 by $D_d(n)$.

We now have all information necessary to estimate the cost of the algorithm for this case, which (dropping obvious lower order terms) is given by

$$\begin{aligned}\mathrm{Cost}_{dz} &= \mathcal{O}({N_{g\bar{a}}}^n + {N_{g\bar{b}}}^n + C_{g\bar{b}}N_{g\bar{b}} + N_{g\bar{a}}N_{g\bar{b}} + C{N_{g\bar{a}}}^2 + C^2N_gN_{\bar{a}}) \\ &\quad + D_d(n) + \sum_{i=2}^{n} \mathcal{O}(C^2{N_{g\bar{a}}}^3(N_gN_{\bar{a}})^{i-1}).\end{aligned}$$

It is straightforward to see that for the terms in the sum, the one for $i = n$ will again be asymptotically largest, having an expense of $\mathcal{O}(C^2{N_{g\bar{a}}}^3(N_gN_{\bar{a}})^{n-1})$, and is again larger than any of the other terms not involving b. If we also apply the result for the dense division $D_d(n)$ from Table 6.3, we obtain

$$\mathrm{Cost}_{dn} = \mathcal{O}(C_gC_{\bar{b}}(N_gN_{\bar{b}})^n + N_{g\bar{a}}N_{g\bar{b}} + C^2{N_{g\bar{a}}}^3(N_gN_{\bar{a}})^{n-1}). \tag{6.14}$$

Sparse Non-zero

As with the prior cases, we will begin the analysis with the internals of the loop at step 4. The analysis of this loop is closely related to that of the sparse non-zero case for the multivariate diophantine algorithm. Recall from that analysis that the non-zero evaluation point caused the intermediate σ,τ and error to become dense in the current variable, and we will see the same behavior here with respect to u,w and e. Also, as with the sparse non-zero multivariate diophantine analysis, we assume that use of a non-zero evaluation point does not significantly reduce the number of terms in u,w.

As with the prior cases, we assume that we have computed values of $a_i \bmod p^l$ outside of the loop, so step 4.2 has no expense. The expense of the computation of the initial error in step 4.3 is simply $\mathcal{O}(C^2 T_g T_{\bar{a}})$.

The loop in step 4.4 repeats $N_{g\bar{a}}$ times, and the expense of step 4.4.1 is $\mathcal{O}(C^2 k)$, as m is a dense polynomial of degree k in x_i. The expense of step 4.4.2, the extraction of the degree k Taylor coefficient from the error, can be estimated as $\mathcal{O}(C^2 k T_g T_{\bar{a}})$ as the error has approximately $\mathcal{O}(k T_g T_{\bar{a}})$ terms, and all terms of degree k and higher contribute to c_k as a result of the non-zero evaluation point. In addition c_k will have $\mathcal{O}(T_g T_{\bar{a}})$ terms. Since we have a non-zero evaluation point we will proceed to step 4.4.4, and the expense of the multivariate diophantine algorithm for this case (Table 6.6) simplifies to $\mathcal{O}(C^2 T_g T_{\bar{a}} {N_{g\bar{a}}}^i)$. Within steps 4.4.5 and 4.4.6, the error update in the latter is asymptotically largest, so the expense for these steps is $\mathcal{O}(C^2 k^2 T_g T_{\bar{a}})$.

Combining these expenses (dropping lower order terms) gives

$$\mathrm{Cost}_{4i} = \mathcal{O}(C^2 T_g T_{\bar{a}}) + \sum_{k=1}^{N_{g\bar{a}}} \mathcal{O}(C^2 T_g T_{\bar{a}} {N_{g\bar{a}}}^i + C^2 T_g T_{\bar{a}} k^2)).$$

Further simplification of the above, together with the fact that $i \geq 2$ gives us the simple complexity estimate $\mathrm{Cost}_{4i} = \mathcal{O}(C^2 T_g T_{\bar{a}} {N_{g\bar{a}}}^{i+1})$. We note that in the case $i = 2$ the error update in step 4.4.6 and the multivariate diophantine solution in step 4.4.4 contribute equally to the expense, while for $i > 2$ only the multivariate diophantine solution contributes to the expense.

Now for the analysis of the entire algorithm, consider the evaluations in step 1. As with the prior cases, we precompute all a_i values in this step. In this case, we evaluate one variable at a time, with expense $\mathcal{O}(C^2 T_g T_{\bar{a}})$ for each, then evaluate the final univariate result with negligible expense $\mathcal{O}(C T_g T_{\bar{a}})$. This entire process has expense bounded by $\mathcal{O}(C^2(n-1) T_g T_{\bar{a}})$. As with the dense non-zero case, the required computation for b can be done more efficiently as the entire process can be computed $\bmod\, p$, so the expense for the evaluation of b is $\mathcal{O}((C_{g\bar{b}} + n) T_g T_{\bar{b}})$.

The GCD in step 2 has expense $\mathcal{O}(N_{g\bar{a}} N_{g\bar{b}})$, the univariate Hensel algorithm in step 3 has expense $\mathcal{O}((C^2 + C N_{g\bar{a}}) T_g T_{\bar{a}} + N_g N_{\bar{a}})$, and we denote the cost of the division in step 5 by $D_s(n)$.

We now have all information necessary to estimate the cost of the algorithm for this case, which (dropping obvious lower order terms) is given by

$$\begin{aligned}\mathrm{Cost}_{sn} &= \mathcal{O}(C^2(n-1)T_gT_{\bar{a}} + (C_{g\bar{b}}+n)T_gT_{\bar{b}} + N_{g\bar{a}}N_{g\bar{b}} + (C^2 + CN_{g\bar{a}})T_gT_{\bar{a}}) \\ &+ \sum_{i=2}^{n} \mathcal{O}(C^2T_gT_{\bar{a}}{N_{g\bar{a}}}^{i+1}) + D_s(n).\end{aligned}$$

It is straightforward to see that for the terms in the sum, the one for $i = n$ will be asymptotically dominant, and will eliminate a number of lower order terms. Combining with the division cost $D_s(n) = \mathcal{O}(C_gT_g(C_{\bar{a}}T_{\bar{a}} + C_{\bar{b}}T_{\bar{b}})$ from Table 6.3 and simplifying gives

$$\mathrm{Cost}_{sn} = \mathcal{O}((C_gC_{\bar{b}} + n)T_gT_{\bar{b}} + N_{g\bar{a}}N_{g\bar{b}} + C^2({N_{g\bar{a}}}^{n+1} + n)T_gT_{\bar{a}}). \tag{6.15}$$

Sparse Zero

As with the prior cases, we will begin the analysis of the loop at step 4. For this analysis we will use similar notation as for the analysis of the sparse zero case of the multivariate diophantine algorithm. Recall from that analysis that for any given set of evaluations, the number of terms in each of the expressions is smaller, as evaluation of some of the variables to zero only seeks to reduce the size of the inputs and the result. We will simplify the notation for the initial analysis by using the standard notation (e.g. T_g,$T_{\bar{a}}$, etc.), but keep in mind that T_g really means the number of terms in u at the current evaluation level, and $T_{g,0}$ represents the number of terms in u one level of evaluation deeper.

Again we compute and retain a_i outside of the loop, so step 4.2 has no expense, and initially u,w have $i-1$ variables, so the error computation in step 4.3 is $\mathcal{O}(C^2T_{g,0}T_{\bar{a},0})$.

The loop in step 4.4 repeats $N_{g\bar{a}}$ times, and the expense of the monomial product in step 4.4.1 is negligible. The expense of step 4.4.2 is at worst $\mathcal{O}(T_gT_{\bar{a}})$ for the term scan, and c_k will have $\mathcal{O}(T_{\bar{a},0}T_{g,k} + T_{g,0}T_{\bar{a},k})$ terms, as it is the sum of the products of the order k and 0 Taylor coefficients of u,w.

If we proceed to step 4.4.4, the expense of that step is $\mathcal{O}((iC^2+CN_{g\bar{a}}+(i-1)N_{g\bar{a}})(T_{\bar{a},0}T_{g,k}+T_{g,0}T_{\bar{a},k}))$. Again, the error update in step 4.4.6 is asymptotically larger than any other computations in steps 4.4.5, 4.4.6, and we can compute the asymptotic complexity of the error update to be $\mathcal{O}(C^2(T_{\bar{a},k}\sum_{j=0}^{k-1}T_{g,i}+T_{g,k}\sum_{j=0}^{k-1}T_{\bar{a},i}+T_{\bar{a},k}T_{g,k}))$, which we simplify to $\mathcal{O}(C^2(T_{\bar{a},k}T_g+T_{g,k}T_{\bar{a}}))$.

Combining these expenses (dropping lower order terms) gives us

$$\begin{aligned}
\mathrm{Cost}_{4i} &= \mathcal{O}(C^2T_{g,0}T_{\bar{a},0})+\sum_{k=1}^{N_{g\bar{a}}}\mathcal{O}(T_gT_{\bar{a}}+(iC^2+CN_{g\bar{a}}+(i-1)N_{g\bar{a}})\\
&\quad \times(T_{\bar{a},0}T_{g,k}+T_{g,0}T_{\bar{a},k})+C^2(T_{\bar{a},k}T_g+T_{g,k}T_{\bar{a}}))\\
&= \mathcal{O}(N_{g\bar{a}}T_gT_{\bar{a}})+\mathcal{O}((iC^2+CN_{g\bar{a}}+(i-1)N_{g\bar{a}})(T_{\bar{a},0}T_g+T_{g,0}T_{\bar{a}}))+\mathcal{O}(C^2T_{\bar{a}}T_g)\\
&= \mathcal{O}((C^2+N_{g\bar{a}})T_gT_{\bar{a}})+\mathcal{O}((iC^2+CN_{g\bar{a}}+(i-1)N_{g\bar{a}})(T_{\bar{a},0}T_g+T_{g,0}T_{\bar{a}}))
\end{aligned}$$

where we have avoided doing anything special with the sum, as the contributions for the conditionally executed steps, 4.4.4-4.4.6, are zero if the steps are not to be run.

We introduce a new notation, namely that the term counts $T_{g:i}$,$T_{\bar{a}:i}$ are used to denote the number of terms in u,w respectively at the end of loop i, or alternatively these can be thought of as the number of terms in u,w evaluated with respect to $\langle x_{i+1},...,x_n\rangle$. Using this notation, the above result becomes $\mathcal{O}((C^2+N_{g\bar{a}})T_{g:i}T_{\bar{a}:i}+(iC^2+CN_{g\bar{a}}+(i-1)N_{g\bar{a}})(T_{\bar{a}:i-1}T_{g:i}+T_{g:i-1}T_{\bar{a}:i}))$.

Now for the analysis of the entire algorithm, consider the evaluations in step 1. Again we perform the required a_i computations outside the loop. Recall that p,l are chosen so that p^l is twice the size of any coefficient of a, so all a_i can be computed in a single term scan with expense $\mathcal{O}(T_gT_{\bar{a}})$. The value of a_0 is computed from a_1 with expense $\mathcal{O}(CT_{g:1}T_{\bar{a}:1})$. Application of a similar approach to b gives expense $\mathcal{O}(T_gT_{\bar{b}}+C_{g\bar{b}}T_{g:1}T_{\bar{b}:1})$.

For step 2, the GCD computation has expense $\mathcal{O}(N_{g\bar{a}}N_{g\bar{b}})$, and the expense of the following division is asymptotically smaller. The expense of the lifting in step 3 was computed earlier to be $\mathcal{O}((C^2+CN_{g\bar{a}})T_{g:1}T_{\bar{a}:1}+N_gN_{\bar{a}})$, and we denote the cost of the division in step 5 by $D_s(n)$.

We now have all components to represent the cost of the algorithm for this case, which (dropping obvious lower order terms) is given by

$$\begin{aligned}
\mathrm{Cost}_{sz} &= \mathcal{O}(T_g(T_{\bar{a}}+T_{\bar{b}})+C_{g\bar{b}}T_{g:1}T_{\bar{b}:1}+N_{g\bar{a}}N_{g\bar{b}}+(C^2+CN_{g\bar{a}})T_{g:1}T_{\bar{a}:1})\\
&+ \sum_{i=2}^{n}\mathcal{O}((C^2+N_{g\bar{a}})T_{g:i}T_{\bar{a}:i}+(iC^2+CN_{g\bar{a}}+(i-1)N_{g\bar{a}})(T_{\bar{a}:i-1}T_{g:i}+T_{g:i-1}T_{\bar{a}:i})).\\
&+ D_s(n).
\end{aligned}$$

From this we can obtain a *very* coarse estimate of the work involved in the algorithm, by assuming that $T_{j:i} = \mathcal{O}(T_j)$, giving (after combining with the division cost from Table 6.3 and suitable simplification)

$$\mathrm{Cost}_{sz} = \mathcal{O}(C_g C_{\bar{b}} T_g T_{\bar{b}} + N_{g\bar{a}} N_{g\bar{b}} + nCN_{g\bar{a}} T_g T_{\bar{a}} + n^2(C^2 + N_{g\bar{a}}) T_g T_{\bar{a}}), \tag{6.16}$$

where the first two terms account for all complexity associated with b (steps 1,2, and 5 only), and the third and fourth terms describe the complexity of the lifting algorithm. Of these terms, we claim that the third term is likely the dominant term for the lifting algorithm with the following justification. It can be observed that operations involving coefficient arithmetic have a much more visible effect than those that are pure data structure operations, so we expect the $nCN_{g\bar{a}}T_gT_{\bar{a}}$ term will dominate the $n^2N_{g\bar{a}}T_gT_{\bar{a}}$ term, unless n is very large. It is also likely that the $nCN_{g\bar{a}}T_gT_{\bar{a}}$ term will dominate the $n^2C^2T_gT_{\bar{a}}$ term unless the coefficients and number of variables are quite large. The third term from the result above is in agreement with the analysis of Moses and Yun [44].

To obtain a finer estimate, we consider the behavior of $T_{g:i}, T_{\bar{a}:i}$ as we vary i. First we have $T_{g:n} = T_g$ and $T_{\bar{a}:n} = T_{\bar{a}}$, but consider now $T_{g:n-1}, T_{\bar{a}:n-1}$. For these we have evaluated the u,w with respect to $x_n = 0$, which will clearly reduce the number of terms, but by how much? To get a basic approximation we assume that at most half of the terms in u and w are degree 0 in x_n (a pessimistic estimate for most problems). In addition, we know that we must have $T_{g:1}, T_{\bar{a}:1}$ non-zero, and in fact a minimum of 2 terms to avoid the bad zero problem. Applying this logic recursively to all evaluation levels provides us with the estimate

$$T_{j:i} = \mathcal{O}(T_j 2^{i-n}) + \mathcal{O}(1).$$

Following through the analysis with this estimate simplifies the sum to

$$\begin{aligned}
& \sum_{i=2}^{n} \mathcal{O}((C^2 + N_{g\bar{a}})T_{g:i}T_{\bar{a}:i} + (iC^2 + CN_{g\bar{a}} + (i-1)N_{g\bar{a}})(T_{g:i-1}T_{\bar{a}:i} + T_{\bar{a}:i-1}T_{g:i})) \\
= & \sum_{i=2}^{n} \mathcal{O}((C^2 + N_{g\bar{a}})(T_gT_{\bar{a}}2^{2(i-n)} + (T_g + T_{\bar{a}})2^{i-n} + 1)) \\
& + \sum_{i=2}^{n} \mathcal{O}(CN_{g\bar{a}}(T_gT_{\bar{a}}2^{2(i-n)-1} + (T_g + T_{\bar{a}})2^{i-n} + 1)) \\
& + \sum_{i=2}^{n} \mathcal{O}(i(C^2 + N_{g\bar{a}})(T_gT_{\bar{a}}2^{2(i-n)-1} + (T_g + T_{\bar{a}})2^{i-n} + 1))
\end{aligned}$$

$$
\begin{aligned}
&= \mathcal{O}((C^2+N_{g\bar{a}})T_gT_{\bar{a}} + n(C^2+N_{g\bar{a}})) + \mathcal{O}(CN_{g\bar{a}}T_gT_{\bar{a}} + nCN_{g\bar{a}}) \\
&\quad + \mathcal{O}(n(C^2+N_{g\bar{a}})T_gT_{\bar{a}} + n^2(C^2+N_{g\bar{a}})) \\
&= \mathcal{O}(CN_{g\bar{a}}T_gT_{\bar{a}} + n((C^2+N_{g\bar{a}})T_gT_{\bar{a}} + CN_{g\bar{a}}) + n^2(C^2+N_{g\bar{a}})),
\end{aligned}
$$

so the assumed form for u,w results in a decoupling of the term density and the number of variables for the core expense of the algorithm, and may provide a better estimate of its complexity.

In addition, the form of the complexity estimate for the sum provides us additional information that allows us to better choose the order of the evaluations. Specifically we would see the greatest efficiency if the evaluations were chosen in such a way that $x_n = 0$ eliminated the greatest number of terms in a (and hence u,w also), $x_{n-1} = 0$ eliminated the greatest number of remaining terms in a_{n-1}, etc.

We state here the complete result using the assumed form for u,w

$$
\begin{aligned}
\mathrm{Cost}_{sz} &= \mathcal{O}(C_gC_{\bar{b}}T_gT_{\bar{b}} + N_{g\bar{a}}N_{g\bar{b}} + CN_{g\bar{a}}T_gT_{\bar{a}} \\
&\quad + n((C^2+N_{g\bar{a}})T_gT_{\bar{a}} + CN_{g\bar{a}}) + n^2(C^2+N_{g\bar{a}})) \qquad (6.17)
\end{aligned}
$$

Sparse Partial Zero

As with the analysis for the multivariate diophantine algorithm, we also consider the case where some of the evaluation points are zero (we use n_0), and some are non-zero ($n - n_0 - 1$).

Based on the results of the analysis for the multivariate diophantine algorithm, we expect that the greatest efficiency for *EEZ-GCD* will be realized if we choose the variable order so that the zero evaluations occur at the highest level. More specifically, the variables are ordered so that we have $\langle x_2 - \alpha_2, ..., x_{n-n_0} - \alpha_{n-n_0}, x_{n-n_0+1}, ..., x_n \rangle$.

For the most part we can utilize results from the analysis of the sparse zero and sparse non-zero cases. Using these results, we can state that the expense for loop 4 for the non-zero evaluation points ($i \le n - n_0$) will be given by $\mathcal{O}(C^2T_gT_{\bar{a}}{N_{g\bar{a}}}^{i+1})$, so the expense for all iterations with $i \le n - n_0$ will be dominated by the loop for $i = n - n_0$ with expense $\mathcal{O}(C^2T_gT_{\bar{a}}{N_{g\bar{a}}}^{n-n_0+1})$. Combining the analysis of the sparse zero case with the sparse mixed diophantine result gives us the expense for loop 4 for the zero evaluation points ($i > n - n_0$) as $\mathcal{O}((C^2+N_{g\bar{a}})T_gT_{\bar{a}} + (C^2{N_{g\bar{a}}}^{n-n_0} + (i-n+n_0)(C^2+N_{g\bar{a}}))(T_{g,0}T_{\bar{a}} + T_{\bar{a},0}T_g))$.

Applying similar assumptions as the sparse zero case on the form of g and $\bar{a}$, we approximate the number of terms in iteration $i >= n - n_0$ to be $T_{j:i} = \mathcal{O}(T_j 2^{i-n}) + \mathcal{O}(1)$, and obtain the following sum for all iterations of loop 4

$$\begin{aligned}
\text{Cost}_4 &= \mathcal{O}(C^2 T_{g:n-n_0} T_{\bar{a}:n-n_0} {N_{g\bar{a}}}^{n-n_0+1}) + \sum_{i=n-n_0+1}^{n} \mathcal{O}((C^2 + N_{g\bar{a}}) T_{g:i} T_{\bar{a}:i}) \\
&\quad + \sum_{i=n-n_0+1}^{n} \mathcal{O}(C^2 {N_{g\bar{a}}}^{n-n_0} (T_{g:i-1} T_{\bar{a}:i} + T_{\bar{a}:i-1} T_{g:i})) \\
&\quad + \sum_{i=n-n_0+1}^{n} \mathcal{O}((i-n+n_0)(C^2 + N_{g\bar{a}})(T_{g:i-1} T_{\bar{a}:i} + T_{\bar{a}:i-1} T_{g:i})) \\
&= \mathcal{O}(C^2 {N_{g\bar{a}}}^{n-n_0+1} (2^{-2n_0} T_g T_{\bar{a}} + 1)) + \sum_{i=n-n_0+1}^{n} \mathcal{O}((C^2 + N_{g\bar{a}})(2^{2(i-n)} T_g T_{\bar{a}} + 1)) \\
&\quad + \sum_{i=n-n_0+1}^{n} \mathcal{O}(C^2 {N_{g\bar{a}}}^{n-n_0} (2^{2(i-n)-1} T_g T_{\bar{a}} + 1)) \\
&\quad + \sum_{i=n-n_0+1}^{n} \mathcal{O}((i-n+n_0)(C^2 + N_{g\bar{a}})(2^{2(i-n)-1} T_g T_{\bar{a}} + 1)) \\
&= \mathcal{O}(C^2 {N_{g\bar{a}}}^{n-n_0+1} T_g T_{\bar{a}}) + \mathcal{O}((C^2 + N_{g\bar{a}})(T_g T_{\bar{a}} + n - n_0)) \\
&\quad + \mathcal{O}(C^2 {N_{g\bar{a}}}^{n-n_0} (T_g T_{\bar{a}} + n - n_0)) + \mathcal{O}((C^2 + N_{g\bar{a}})((n-n_0) T_g T_{\bar{a}} + (n-n_0)^2)) \\
&= \mathcal{O}(C^2 {N_{g\bar{a}}}^{n-n_0+1} T_g T_{\bar{a}} + (n-n_0)(C^2 {N_{g\bar{a}}}^{n-n_0} + (C^2 + N_{g\bar{a}}) T_g T_{\bar{a}}) \\
&\quad + (n-n_0)^2 (C^2 + N_{g\bar{a}})).
\end{aligned}$$

The expense of the evaluations required for step 4.2 will be dominated by the non-zero evaluations. The expense of the evaluations of a can be bounded by $\mathcal{O}(C^2(n-n_0-1) T_g T_{\bar{a}})$, and the expense of the single evaluation of b can be bounded by $\mathcal{O}((C_{g\bar{b}} + n - n_0) T_g T_{\bar{b}})$. The other expenses are all as in the sparse non-zero case, so we can estimate the cost of the algorithm for this case (including all contributions) as

$$\begin{aligned}
\text{Cost}_{n_0} &= \mathcal{O}(C_g C_{\bar{b}} T_g T_{\bar{b}} + N_{g\bar{a}} N_{g\bar{b}} + C^2 {N_{g\bar{a}}}^{n-n_0+1} T_g T_{\bar{a}} \\
&\quad + (n-n_0)(C^2 {N_{g\bar{a}}}^{n-n_0} + (C^2 + N_{g\bar{a}}) T_g T_{\bar{a}}) + (n-n_0)^2 (C^2 + N_{g\bar{a}}),
\end{aligned} \tag{6.18}$$

where we note that for the case of a *single* non-zero evaluation point, the estimate becomes cubic in $N_{g\bar{a}}$.

6.8.5 Asymptotic complexity of *LINZIP*

The analysis for the *LINZIP* algorithm is more straightforward than for the *EEZ-GCD* algorithm, as the number of algorithms used, and the number of cases is smaller. Specifically, the difference between the complexity of the *LINZIP P* and *LINZIP M* algorithms is minimal, so the analysis for *LINZIP M* will heavily utilize the results from *LINZIP P*.

Note that in the analysis, we exclude the unlucky and bad cases, as these should be sufficiently rare and do not cause catastrophic failure as they do for *EEZ-GCD*. In addition, we ignore the missing term problem, as it should be rare for sufficiently sparse inputs (and can also be handled using a variant of point 7 at the end of §6.6 without significantly affecting the asymptotic cost).

LINZIP P: Base Case

Some consideration must be made when determining which n corresponds to the base case of the *LINZIP P* algorithm. Clearly when entering *LINZIP P* with only 2 variables, most of the structure is redundant, as we need only compute a single image for each reconstruction (i.e. loop 5.3 is useless and the linear solve in step 5.4 is trivial). This suggests that for practical implementation, when we have reduced to the bivariate case we may want to include an optimized Brown-type algorithm (e.g. *DIVBRO*).

This is unnecessary for the analysis, as the expense for the base case of $n = 2$ would be the same as the expense for the *DIVBRO* algorithm.

LINZIP P: Dense Case

In step 0 of algorithm 22 we perform a univariate content check for content in the current variable, which we see from Table 6.2 has an expense of $\mathcal{O}({N_{g\bar{a}}}^n + {N_{g\bar{b}}}^n + N_{g\bar{a}}N_{g\bar{b}}) = \mathcal{O}({N_{g\bar{a}\bar{b}}}^n + N_{g\bar{a}}N_{g\bar{b}})$, and in step 1 we compute the scaling factor γ via a univariate GCD of the leading coefficients with expense $\mathcal{O}(N_{g\bar{a}}N_{g\bar{b}})$.

In step 2, the inputs a, b are evaluated with respect to $x_n = v_n$, having $\mathcal{O}({N_{g\bar{a}\bar{b}}}^n)$ expense, which is followed by a recursive call to *LINZIP P* for one fewer variable, with expense $\mathrm{Cost}_{Pd}(n-1)$.

In step 3, we compute the number of univariate images needed to reconstruct a new image in one fewer than the current number of variables, which is $n_i = {N_g}^{n-2}$ for the dense case. We note here that the number of images needed with scaling is at most 50% higher than without, so this choice does not affect the asymptotic complexity.

The loop in step 5 will repeat $\mathcal{O}(N_g + N_\gamma) = \mathcal{O}(N_{g\gamma})$ times, and step 5.1 will have expense $\mathcal{O}({N_{g\bar{a}\bar{b}}}^n)$.

The loop in step 5.3 will repeat $\mathcal{O}(n_i) = \mathcal{O}({N_g}^{n-2})$ times. Step 5.3.1 for the evaluation of a,b with respect to $x_2 = \alpha_2, ..., x_{n-1} = \alpha_{n-1}$ will have expense $\mathcal{O}((n-1){N_{g\bar{a}\bar{b}}}^{n-1})$ (for the computation of $n-1$ products in $\mathbf{Z}_p$ for each term). Step 5.3.2 is a univariate modular GCD with expense $\mathcal{O}((N_g + N_{\bar{a}})(N_g + N_{\bar{b}})) = \mathcal{O}(N_{g\bar{a}}N_{g\bar{b}})$, and we will assume that step 5.3.5 will have negligible expense with compared with the following linear solve (i.e. placing the data in the matrix structure is less expensive than performing the linear solve). This completes loop 5.3 for which the expense is given by $\mathcal{O}({N_g}^{n-2}(N_{g\bar{a}}N_{g\bar{b}} + (n-1){N_{g\bar{a}\bar{b}}}^{n-1}))$.

Now for step 5.4, the analysis for the multiple and single scaling cases is slightly different, though as we will see the asymptotic result is the same. We will start with the single scaling case. We need to reconstruct all terms for each coefficient in $x_1 \in \mathbf{Z}_p[x_2, ..., x_{n-1}]$. Since the input is dense, the number of terms for the coefficient of x_1^j is $\mathcal{O}({N_g}^{n-2})$. The solution process is simply a modular linear solve with expense of $\mathcal{O}(S^3)$ for S unknowns, so in this case the total expense of obtaining the values for all coefficients is given by $\sum_{j=0}^{N_g} \mathcal{O}({N_g}^{3n-6}) = \mathcal{O}({N_g}^{3n-5})$.

For the multiple scaling case, we need to structure the computation as described in §6.6. Doing so, only one dimension of each modular matrix will be larger, but no larger than the number of images used. This simply increases the number of columns in each matrix by ${N_g}^{n-2}$, but this is on the order of the number of columns we already have, so it simply inserts a factor of 2 into the expense of the solve. The factor of 2 can be explained as follows: The total number of row operations on the matrix (which is $\mathcal{O}(S^2)$) will be the same for both the multiple and single scaling cases, as we are working with the GCD *unknown* part of the Matrix, and carrying the additional columns along. Each of these row operations is twice as expensive as in the single scaling case, so the overall adjustment to the expense of the linear solve will be a factor of 2. Finally, the solution for the multipliers will require the solution of a ${N_g}^{n-2}$ by ${N_g}^{n-2}$ matrix, so this adds one term to the sum above, but the asymptotic result remains the same.

This completes loop 5, and the total expense of that loop is given by

$$\begin{aligned} & N_{g\gamma} \times \mathcal{O}({N_{g\bar{a}\bar{b}}}^n + {N_g}^{n-2}(N_{g\bar{a}}N_{g\bar{b}} + (n-1){N_{g\bar{a}\bar{b}}}^{n-1}) + {N_g}^{3n-5}) \\ & = \mathcal{O}(N_{g\gamma}{N_{g\bar{a}\bar{b}}}^n + N_{g\gamma}{N_g}^{n-2}(N_{g\bar{a}}N_{g\bar{b}} + (n-1){N_{g\bar{a}\bar{b}}}^{n-1}) + N_{g\gamma}{N_g}^{3n-5}). \end{aligned}$$

The Newton interpolation in step 6 has a known expense of $\mathcal{O}({N_{g\gamma}}^2)$ for each coefficient, so the total expense is $\mathcal{O}({N_{g\gamma}}^2{N_g}^{n-1})$. The loop in step 7 will not affect the overall complexity, as the test in the loop is no more expensive than a single iteration of loop 5, and it is only being run a fixed number of times (generally once or twice).

Now as for the *DIVBRO* algorithm, using the same arguments (primarily that we do have some freedom in choosing the scaling coefficient) we make the reasonable assumption that $N_g = \mathcal{O}(N_\gamma)$, which somewhat simplifies the earlier expenses, and combining these (dropping lower order terms) we obtain

$$\begin{aligned} \mathrm{Cost}_{Pd}(n) &= \mathrm{Cost}_{Pd}(n-1) \\ &\quad + \mathcal{O}(N_g{N_{g\bar{a}\bar{b}}}^n + {N_g}^{n-1}(N_{g\bar{a}}N_{g\bar{b}} + (n-1){N_{g\bar{a}\bar{b}}}^{n-1}) + {N_g}^{\max(3n-4,n+1)}) \\ &= \mathrm{Cost}_{Pd}(n-1) + \mathrm{Cost}_{Pd0}(n), \end{aligned}$$

where we use $\mathrm{Cost}_{Pd0}(n)$ to represent the cost at the current level only.

Note that the choice of using a trailing (rather than a leading) coefficient for scaling requires a similar adjustment be applied to the check for bad evaluations and the chosen scaling coefficient for the multiple scaling case.

LINZIP M: Dense Case

In step 1 of algorithm 21 we are computing the scaling factor γ via a single integer GCD with expense $\mathcal{O}(C_{g\bar{a}}C_{g\bar{b}})$.

In step 2 we compute the inputs a, b modulo a prime p, having $\mathcal{O}(C_{g\bar{a}}{N_{g\bar{a}}}^n + C_{g\bar{b}}{N_{g\bar{b}}}^n)$ expense, which is followed by a call to *LINZIP P* for the total number of variables for the problem, with expense $\mathrm{Cost}_{Pd}(n)$.

In step 3, we compute the number of univariate images needed to reconstruct a new image with respect to a different prime, which is $n_i = {N_g}^{n-1}$ for the dense case. Again, use of multiple scaling will increase the number of images by at most 50% which will not significantly affect the asymptotic complexity.

The loop in step 5 will repeat $\mathcal{O}(C_g+C_\gamma) = \mathcal{O}(C_{g\gamma})$ times, and step 5.1 will have expense $\mathcal{O}(C_{g\bar{a}}{N_{g\bar{a}}}^n + C_{g\bar{b}}{N_{g\bar{b}}}^n)$.

The analysis for the loop starting in step 5.3 will be identical to that of *LINZIP P*, with the exception that we will need to execute the loop $\mathcal{O}({N_g}^{n-1})$ times, and the evaluations will have an expense of $\mathcal{O}(n{N_{g\bar{a}\bar{b}}}^n)$ (for the computation of n products in $\mathbf{Z}_p$ for each term), so the overall expense of that loop is $\mathcal{O}({N_g}^{n-1}(N_{g\bar{a}}N_{g\bar{b}} + n{N_{g\bar{a}\bar{b}}}^n))$.

For step 5.4, the same arguments as for *LINZIP P* apply for the multiple and single scaling cases, so we need only consider the single scaling case. Again the analysis is nearly identical, noting that now we need to reconstruct all variables, and that the number of terms for the coefficient of x_1^j is $\mathcal{O}({N_g}^{n-1})$. The sum for the expense of the linear solves becomes $\sum_{j=0}^{N_g} \mathcal{O}({N_g}^{3n-3}) = \mathcal{O}({N_g}^{3n-2})$.

The expense of the Chinese remaindering in step 5.7 is known to accumulate to $\mathcal{O}({C_{g\gamma}}^2)$ for each coefficient over all iterations, so the total contribution to the expense is given by $\mathcal{O}({C_{g\gamma}}^2{N_g}^n)$.

This completes loop 5, and the total expense of that loop is given by

$$\begin{aligned}
&C_{g\gamma} \times \mathcal{O}(C_{g\bar{a}}{N_{g\bar{a}}}^n + C_{g\bar{b}}{N_{g\bar{b}}}^n + {N_g}^{n-1}(N_{g\bar{a}}N_{g\bar{b}} + n{N_{g\bar{a}\bar{b}}}^n) + {N_g}^{3n-2}) + \mathcal{O}({C_g}^2{N_g}^n)\\
&= \mathcal{O}(C_{g\gamma}(C_{g\bar{a}}{N_{g\bar{a}}}^n + C_{g\bar{b}}{N_{g\bar{b}}}^n) + C_{g\gamma}{N_g}^{n-1}(N_{g\bar{a}}N_{g\bar{b}} + n{N_{g\bar{a}\bar{b}}}^n) + C_{g\gamma}{N_g}^{3n-2}).
\end{aligned}$$

Again with a similar argument as for *LINZIP P* we assume $C_\gamma = \mathcal{O}(C_g)$, and denoting the expense of the division test by $D_d(n)$, gives our total expense as

$$\begin{aligned}
\mathrm{Cost}_{Md}(n) &= \mathrm{Cost}_{Pd}(n) + \mathcal{O}(C_{g\bar{a}}C_{g\bar{b}} + C_gC_{g\bar{a}}{N_{g\bar{a}}}^n + C_gC_{g\bar{b}}{N_{g\bar{b}}}^n\\
&\quad +C_g{N_g}^{n-1}(N_{g\bar{a}}N_{g\bar{b}} + n{N_{g\bar{a}\bar{b}}}^n) + C_g{N_g}^{3n-2} + {C_g}^2{N_g}^n + D_d(n)\\
&= \mathrm{Cost}_{Pd}(n) + \mathrm{Cost}_{Md0}(n),
\end{aligned}$$

where we use $\mathrm{Cost}_{Md0}(n)$ to represent the cost at the current level only.

LINZIP: Dense Result

We can now combine the results of the *LINZIP P* and *LINZIP M* analyses to obtain an overall asymptotic estimate for the dense case.

The first consideration here is that the recursion is constant, so the solution of the recurrence is simply given by

$$\mathrm{Cost}_{Md}(n) = \mathrm{Cost}_{Md0}(n) + \sum_{i=2}^{n} \mathrm{Cost}_{Pd0}(i).$$

Examination of the form of $\mathrm{Cost}_{Pd0}(i)$ shows that when compared to $\mathrm{Cost}_{Pd0}(n)$, all the other $\mathrm{Cost}_{Pd0}(i)$ terms are asymptotically insignificant, so this simplifies to

$$\mathrm{Cost}_{Md}(n) = \mathrm{Cost}_{Md0}(n) + \mathrm{Cost}_{Pd0}(n).$$

This can be written (at length) as:

$$\begin{aligned}\mathrm{Cost}_{Md}(n) \;=\; & \mathcal{O}(C_{g\bar{a}}C_{g\bar{b}} + C_g C_{g\bar{a}} {N_{g\bar{a}}}^n + C_g C_{g\bar{b}} {N_{g\bar{b}}}^n + C_g {N_g}^{n-1}(N_{g\bar{a}}N_{g\bar{b}} + n{N_{g\bar{a}\bar{b}}}^n) \\ & + C_g {N_g}^{3n-2} + {C_g}^2 {N_g}^n + {N_g}^{n-1}(N_{g\bar{a}}N_{g\bar{b}} + (n-1){N_{g\bar{a}\bar{b}}}^{n-1}) \\ & + N_g {N_{g\bar{a}\bar{b}}}^n + {N_g}^{\max(3n-4,n+1)} + C_g {N_g}^n(C_{\bar{a}}{N_{\bar{a}}}^n + C_{\bar{b}}{N_{\bar{b}}}^n)),\end{aligned}$$

where we have added in the division cost from Table 6.3. Simplification of the above, including the (asymptotically insignificant) expense of the degree bound computation from Table 6.1 gives

$$\begin{aligned}\mathrm{Cost}_{Md}(n) \;=\; & \mathcal{O}(C_{g\bar{a}}C_{g\bar{b}} + C_g C_{g\bar{a}} {N_{g\bar{a}}}^n + C_g C_{\bar{a}} {N_g}^n {N_{\bar{a}}}^n + C_g C_{g\bar{b}} {N_{g\bar{b}}}^n + C_g C_{\bar{b}} {N_g}^n N_{\bar{b}}^n \\ & + C_g {N_g}^{n-1}(N_{g\bar{a}}N_{g\bar{b}} + n{N_{g\bar{a}\bar{b}}}^n) + C_g {N_g}^{3n-2}), \end{aligned} \tag{6.19}$$

which again emphasizes that the greatest expense for the dense case occurs at the top level, as all terms from $\mathrm{Cost}_{Pd0}(n)$ are of lower order than the contribution from $\mathrm{Cost}_{Md0}(n)$.

Note: use of this algorithm for the dense case is ill-advised for a number of reasons. The first of which is that the probability of computing a correct initial image in step 1 of the algorithm is inversely proportional to the number of terms present in the GCD, which means that for sufficiently dense problems it will fail frequently (though this is somewhat alleviated by the multiple image strategy described in point 7 following the algorithm in §6.6). The second of which is that the expense of solving the linear system is cubic in the number of unknowns, from which we expect that this direct approach will be quite inefficient for this case. There is an approach described in Zippel [79, 80], and Kaltofen and Lee [32] that can be used to reduce the expense of the linear solve from cubic to quadratic, but as we will see this will not be the preferred approach for dense problems, so the analysis of this case is provided primarily for discussion.

LINZIP P: Sparse Case

In step 0 of algorithm 22 we perform a univariate content check for content in the current variable, which we see from Table 6.2 has an expense of $\mathcal{O}(T_g T_{\bar{a}\bar{b}} + N_{g\bar{a}} N_{g\bar{b}})$, and in step 1 we need to compute the scaling factor γ via a univariate GCD of the leading coefficients with expense $\mathcal{O}(N_{g\bar{a}} N_{g\bar{b}})$.

In step 2, the input a,b are evaluated with respect to $x_n = v_n$, having $\mathcal{O}(T_g T_{\bar{a}\bar{b}})$ expense, which is followed by a recursive call to *LINZIP P* for one fewer variable, with expense $\mathrm{Cost}_{Ps}(n-1)$.

In step 3, we compute the number of univariate images needed to reconstruct a new image in one fewer than the current number of variables, which is $n_i = T_g$. Again the number of images required by multiple scaling does not significantly affect the asymptotic estimate.

The loop in step 5 will repeat $\mathcal{O}(N_g + N_\gamma) = \mathcal{O}(N_{g\gamma})$ times, and step 5.1 will have expense $\mathcal{O}(N_{g\bar{a}\bar{b}} + T_g T_{\bar{a}\bar{b}})$, where the first term is the cost of computing the values of x_n^i, and the second is the cost of performing the evaluations.

The loop in step 5.3 will repeat $\mathcal{O}(n_i) \le \mathcal{O}(T_g)$ times. The expense in step 5.3.1 for the evaluation with respect to $x_2 = v_2, ..., x_{n-1} = v_{n-1}$ will be $\mathcal{O}((n-1)(N_{g\bar{a}\bar{b}} + T_g T_{\bar{a}\bar{b}}))$. Step 5.3.2 is a univariate modular GCD with expense $\mathcal{O}(N_{g\bar{a}} N_{g\bar{b}})$, and we will assume that step 5.3.3 will have negligible expense with compared with the following linear solve (i.e. placing the data in the matrix structure is less expensive than performing the linear solve). This completes loop 5.3, and the total expense of that loop is given by $\mathcal{O}(T_g(N_{g\bar{a}} N_{g\bar{b}} + (n-1)(N_{g\bar{a}\bar{b}} + T_g T_{\bar{a}\bar{b}})))$.

Now for step 5.4, the analysis for the multiple and single scaling cases is again equivalent, due to the arguments put forth for the prior cases, so we will only consider the single scaling case. We need to reconstruct all terms for each coefficient in $x_1 \in \mathbf{Z}_p[x_2, ..., x_{n-1}]$. Since the input is sparse the number of terms for the coefficient of x_1^j is highly problem dependent. For a truly sparse polynomial of moderate degree, this could be as small as $\mathcal{O}(1)$, or for a highly structured polynomial, for example, one where half the terms of the GCD are degree 0 in x_1, this could be as large as $\mathcal{O}(T_g)$. We use $\mathcal{O}(T_g)$ for the analysis. It is straightforward to see that the overall cost of the linear solve is limited to $\mathcal{O}({T_g}^3)$, which only occurs for the worst case, as any case where the terms of the GCD are more evenly distributed in the powers of x_1 will result in less expense (the expense in the number of solves is linear, but cubic in the size of the system to be solved).

This completes loop 5, and the total expense of that loop is given by

$$\begin{aligned} & N_{g\gamma} \times \mathcal{O}(N_{g\bar{a}\bar{b}} + T_g T_{\bar{a}\bar{b}} + T_g(N_{g\bar{a}} N_{g\bar{b}} + (n-1)(N_{g\bar{a}\bar{b}} + T_g T_{\bar{a}\bar{b}})) + {T_g}^3) \\ & = \mathcal{O}(N_{g\gamma} T_g (N_{g\bar{a}} N_{g\bar{b}} + (n-1)(N_{g\bar{a}\bar{b}} + T_g T_{\bar{a}\bar{b}})) + N_{g\gamma} {T_g}^3). \end{aligned}$$

The Newton interpolation in step 6 has a known expense of $\mathcal{O}({N_{g\gamma}}^2)$ for each coefficient, so the total expense is $\mathcal{O}({N_{g\gamma}}^2 T_g)$, and with a similar argument as for the dense case, the expense in loop 7 is bounded by the expense of loop 5.

Again with a similar argument as for the dense case we assume that $N_\gamma = \mathcal{O}(N_g)$, so combining the above expenses (dropping lower order terms) we obtain

$$\begin{aligned} \mathrm{Cost}_{Ps}(n) &= \mathrm{Cost}_{Ps}(n-1) + \mathcal{O}(N_g T_g (N_{g\bar{a}} N_{g\bar{b}} + (n-1)(N_{g\bar{a}\bar{b}} + T_g T_{\bar{a}\bar{b}})) + N_g {T_g}^3) \\ &= \mathrm{Cost}_{Ps}(n-1) + \mathrm{Cost}_{Ps0}(n), \end{aligned}$$

where we use $\mathrm{Cost}_{Ps0}(n)$ to represent the cost at the current level only.

Now this analysis surely provides an asymptotic bound for the expense of the algorithm, but it is surely a worst case bound, for the following reasons:

1. We estimated the number of times that we repeat loop 4.3 to be $\mathcal{O}(T_g)$, but this is only true when one of the x_1^j coefficients of the GCD has $\mathcal{O}(T_g)$ terms. It is more likely that the terms are more evenly distributed amongst the powers of x_1, and we can estimate this as perhaps $\mathcal{O}({T_g}^{\frac{1}{2}})$.

2. In the event that the largest coefficient is not $\mathcal{O}(T_g)$, as discussed in the prior point, our estimate for the cost of the solution of the systems of equations in step 4.4 is pessimistic. We would expect instead to need to solve $\mathcal{O}({T_g}^{\frac{1}{2}})$ systems of size $\mathcal{O}({T_g}^{\frac{1}{2}})$, which drops the expense of that step to $\mathcal{O}(N_g {T_g}^2)$.

With these considerations, if the size of the largest coefficient of x_1 in the input is $\mathcal{O}({T_g}^{\frac{1}{2}})$, the complexity of the algorithm is given by

$$\mathrm{Cost}_{Ps0}(n) = \mathcal{O}(N_g {T_g}^{\frac{1}{2}}(N_{g\bar{a}} N_{g\bar{b}} + (n-1)(N_{g\bar{a}\bar{b}} + T_g T_{\bar{a}\bar{b}})) + N_g {T_g}^2).$$

For a highly sparse input, where the number of terms at each degree is $\mathcal{O}(1)$, the expense of the solution process in step 5.4 is dropped as a lower order term, and the complexity would be given by

$$\mathrm{Cost}_{Ps0}(n) = \mathcal{O}(N_g(N_{g\bar{a}} N_{g\bar{b}} + (n-1)(N_{g\bar{a}\bar{b}} + T_g T_{\bar{a}\bar{b}}))).$$

LINZIP M: Sparse Case

In step 1 of algorithm 21 we are computing the scaling factor γ via a single integer GCD with expense $\mathcal{O}(C_{g\bar{a}} C_{g\bar{b}})$.

In step 2, we compute the inputs a,b modulo a prime p, having $\mathcal{O}(C_{g\bar{a}} T_g T_{\bar{a}} + C_{g\bar{b}} T_g T_{\bar{b}})$ expense, which is followed by a call to *LINZIP P* for the total number of variables for the problem, with expense $\mathrm{Cost}_{Ps}(n)$.

In step 3, we compute the number of univariate images needed to reconstruct an image with respect to a new prime, which is $n_i = T_g$. We note that again the number of images required by multiple scaling does not affect the asymptotic estimate.

The loop in step 5 will repeat $\mathcal{O}(C_g + C_\gamma) = \mathcal{O}(C_{g\gamma})$ times, and step 5.1 will have expense $\mathcal{O}(C_{g\bar{a}} T_g T_{\bar{a}} + C_{g\bar{b}} T_g T_{\bar{b}})$.

The analysis for the loop starting in step 5.3 is nearly identical to that of the sparse *LINZIP P*, with associated expense $\mathcal{O}(T_g(N_{g\bar{b}} N_{\bar{a}\bar{b}} + n(N_{g\bar{a}\bar{b}} + T_g T_{\bar{a}\bar{b}})))$. Also, step 5.4 is identical to the sparse *LINZIP P* case with associated expense $\mathcal{O}({T_g}^3)$.

The expense of the Chinese remaindering in step 5.7 is known to accumulate to $\mathcal{O}({C_{g\gamma}}^2)$ for each coefficient over all iterations, so the total contribution to the expense is given by $\mathcal{O}({C_{g\gamma}}^2 T_g)$.

This completes loop 5, and the total expense of that loop is given by

$$C_{g\gamma} \times \mathcal{O}(C_{g\bar{a}}T_gT_{\bar{a}} + C_{g\bar{b}}T_gT_{\bar{b}} + T_g(N_{g\bar{b}}N_{\bar{a}\bar{b}} + n(N_{g\bar{a}\bar{b}} + T_gT_{\bar{a}\bar{b}})) + {T_g}^3) + \mathcal{O}({C_{g\gamma}}^2T_g)$$
$$= \mathcal{O}(C_{g\gamma}C_{g\bar{a}}T_gT_{\bar{a}} + C_{g\gamma}C_{g\bar{b}}T_gT_{\bar{b}} + C_{g\gamma}T_g(N_{g\bar{b}}N_{\bar{a}\bar{b}} + n(N_{g\bar{a}\bar{b}} + T_gT_{\bar{a}\bar{b}})) + C_{g\gamma}{T_g}^3 + {C_{g\gamma}}^2T_g).$$

Noting that the expense of the division test (Table 6.3) is at worst equivalent to the first two terms in the expense for loop 5, and using the assumption $C_\gamma = \mathcal{O}(C_g)$ as in the dense case, our total expense simplifies to

$$\begin{aligned}
\mathrm{Cost}_{Ms}(n) &= \mathrm{Cost}_{Ps}(n) + \mathcal{O}(C_{g\bar{a}}C_{g\bar{b}} + C_gT_g(C_{g\bar{a}}T_{\bar{a}} + C_{g\bar{b}}T_{\bar{b}}) \\
&\quad + C_gT_g(N_{g\bar{b}}N_{\bar{a}\bar{b}} + n(N_{g\bar{a}\bar{b}} + T_gT_{\bar{a}\bar{b}})) + C_g{T_g}^3) \\
&= \mathrm{Cost}_{Ps}(n) + \mathrm{Cost}_{Ms0}(n),
\end{aligned}$$

where we use $\mathrm{Cost}_{Ms0}(n)$ to represent the cost at the current level only.

With the same considerations as used for *LINZIP P*, for the case where the largest coefficient with respect to x_1 has $\mathcal{O}({T_g}^{\frac{1}{2}})$ terms, the complexity of the algorithm is given by

$$\mathrm{Cost}_{Ms0}(n) = \mathcal{O}(C_{g\bar{a}}C_{g\bar{b}} + C_gT_g(C_{g\bar{a}}T_{\bar{a}} + C_{g\bar{b}}T_{\bar{b}}) + C_g{T_g}^{\frac{1}{2}}(N_{g\bar{b}}N_{\bar{a}\bar{b}} + n(N_{g\bar{a}\bar{b}} + T_gT_{\bar{a}\bar{b}})) + C_g{T_g}^2).$$

For a highly sparse input, where the number of terms at each degree is $\mathcal{O}(1)$, the solution process in step 5.4 is again dropped as a lower order term, and the complexity would be given by

$$\mathrm{Cost}_{Ms0}(n) = \mathcal{O}(C_{g\bar{a}}C_{g\bar{b}} + C_gT_g(C_{g\bar{a}}T_{\bar{a}} + C_{g\bar{b}}T_{\bar{b}}) + C_g(N_{g\bar{b}}N_{\bar{a}\bar{b}} + n(N_{g\bar{a}\bar{b}} + T_gT_{\bar{a}\bar{b}}))).$$

LINZIP: Sparse Result

We can now combine the results of the *LINZIP P* and *LINZIP M* analyses to obtain an overall asymptotic estimate for the sparse case.

As in the dense case, recursion is constant, so the solution of the recurrence is simply given by

$$\mathrm{Cost}_{Ms}(n) = \mathrm{Cost}_{Ms0}(n) + \sum_{i=2}^{n} \mathrm{Cost}_{Ps0}(i).$$

We can evaluate the sum, giving

$$\begin{aligned}\mathrm{Cost}_{Ms}(n) &= \mathcal{O}(C_{g\bar{a}}C_{g\bar{b}} + C_gT_g(C_{g\bar{a}}T_{\bar{a}} + C_{g\bar{b}}T_{\bar{b}}) + C_gT_g(N_{g\bar{b}}N_{\bar{a}\bar{b}} + n(N_{g\bar{a}\bar{b}} + T_gT_{\bar{a}\bar{b}})) \\ &\quad + C_g{T_g}^3 + (n-1)N_gT_gN_{g\bar{a}}N_{g\bar{b}} + \frac{n(n-1)}{2}N_gT_g(N_{g\bar{a}\bar{b}} + T_gT_{\bar{a}\bar{b}}) \\ &\quad + (n-1)N_g{T_g}^3),\end{aligned}$$

which after minor simplification, addition of the (asymptotically insignificant) degree bound expense from Table 6.1, and grouping gives

$$\begin{aligned}\mathrm{Cost}_{Ms}(n) &= \mathcal{O}(C_{g\bar{a}}C_{g\bar{b}} + C_gT_g(C_{g\bar{a}}T_{\bar{a}} + C_{g\bar{b}}T_{\bar{b}}) \\ &\quad + T_g(N_{g\bar{a}}N_{g\bar{b}} + n(N_{g\bar{a}\bar{b}} + T_gT_{\bar{a}\bar{b}}) + {T_g}^2)(C_g + nN_g)).\end{aligned} \tag{6.20}$$

As with the analysis of the *LINZIP P* and *LINZIP M* algorithms, we can obtain the cost for the case where g has a maximum of $\mathcal{O}({T_g}^{\frac{1}{2}})$ terms for any x_1^j coefficient as

$$\begin{aligned}\mathrm{Cost}_{Ms}(n) &= \mathcal{O}(C_{g\bar{a}}C_{g\bar{b}} + C_gT_g(C_{g\bar{a}}T_{\bar{a}} + C_{g\bar{b}}T_{\bar{b}}) \\ &\quad + {T_g}^{\frac{1}{2}}(N_{g\bar{a}}N_{g\bar{b}} + n(N_{g\bar{a}\bar{b}} + T_gT_{\bar{a}\bar{b}}) + {T_g}^{\frac{3}{2}})(C_g + nN_g)).\end{aligned} \tag{6.21}$$

Similarly for the $\mathcal{O}(1)$ case we obtain

$$\begin{aligned}\mathrm{Cost}_{Ms}(n) &= \mathcal{O}(C_{g\bar{a}}C_{g\bar{b}} + C_gT_g(C_{g\bar{a}}T_{\bar{a}} + C_{g\bar{b}}T_{\bar{b}}) \\ &\quad + (N_{g\bar{a}}N_{g\bar{b}} + n(N_{g\bar{a}\bar{b}} + T_gT_{\bar{a}\bar{b}}))(C_g + nN_g)).\end{aligned} \tag{6.22}$$

6.9 Summary and Discussion

We summarize our results for the complexity of the *Brown*, *DIVBRO*, *EEZ-GCD* and *LINZIP* algorithms in Table 6.7, where the author believes these to be the most detailed asymptotic results available for the *Brown* and *EEZ-GCD* algorithms.

Brown Dense	$\mathcal{O}(C_{g\bar{a}\bar{b}}{}^2 N_{g\bar{a}\bar{b}}{}^n + nC_{g\bar{a}\bar{b}}N_{g\bar{a}\bar{b}}{}^{n+1})$
Brown Sparse	$\mathcal{O}(C_{g\bar{a}\bar{b}}T_g(C_{g\bar{a}}T_{\bar{a}} + C_{g\bar{b}}T_{\bar{b}}) + C_{g\bar{a}\bar{b}}{}^2 T_{g\bar{a}\bar{b}} + C_{g\bar{a}\bar{b}}n_x N_{g\bar{a}\bar{b}}{}^{n+1})$
DIVBRO Dense	$\mathcal{O}(C_{g\bar{a}}C_{g\bar{b}} + C_g(C_{g\bar{a}}N_{g\bar{a}}{}^n + C_{g\bar{b}}N_{g\bar{b}}{}^n) + nC_gN_gN_{g\bar{a}\bar{b}}{}^n$ $+C_gN_g{}^n(C_{\bar{a}}N_{\bar{a}}{}^n + C_{\bar{b}}N_{\bar{b}}{}^n))$
DIVBRO Sparse	$\mathcal{O}(C_{g\bar{a}}C_{g\bar{b}} + C_gT_g(C_{g\bar{a}}T_{\bar{a}} + C_{g\bar{b}}T_{\bar{b}}) + C_gfN_g{}^{n-n_x+1}N_{g\bar{a}\bar{b}}{}^{n_x}$ $+n\log_2(n)T_gT_{\bar{a}\bar{b}} + nN_{g\bar{a}}N_{g\bar{b}})$
EEZ-GCD Dense Non-zero	$\mathcal{O}(C_gC_{\bar{b}}(N_gN_{\bar{b}})^n + N_{g\bar{a}}N_{g\bar{b}} + C^2N_{g\bar{a}}{}^4(N_gN_{\bar{a}})^{n-1})$
EEZ-GCD Dense Zero	$\mathcal{O}(C_gC_{\bar{b}}(N_gN_{\bar{b}})^n + N_{g\bar{a}}N_{g\bar{b}} + C^2N_{g\bar{a}}{}^3(N_gN_{\bar{a}})^{n-1})$
EEZ-GCD Sparse Non-zero	$\mathcal{O}((C_gC_{\bar{b}} + n)T_gT_{\bar{b}} + N_{g\bar{a}}N_{g\bar{b}} + C^2(n + N_{g\bar{a}}{}^{n+1})T_gT_{\bar{a}})$
EEZ-GCD Sparse Zero $\mathcal{O}(1)$	$\mathcal{O}(C_gC_{\bar{b}}T_gT_{\bar{b}} + N_{g\bar{a}}N_{g\bar{b}}$ $+nCN_{g\bar{a}}T_gT_{\bar{a}} + n^2(C^2 + N_{g\bar{a}})T_gT_{\bar{a}})$
EEZ-GCD Sparse Zero $\mathcal{O}(\frac{1}{2})$	$\mathcal{O}(C_gC_{\bar{b}}T_gT_{\bar{b}} + N_{g\bar{a}}N_{g\bar{b}} + CN_{g\bar{a}}T_gT_{\bar{a}}$ $+n((C^2 + N_{g\bar{a}})T_gT_{\bar{a}} + CN_{g\bar{a}}) + n^2(C^2 + N_{g\bar{a}}))$
EEZ-GCD Sparse Mixed (n_0)	$\mathcal{O}(C_gC_{\bar{b}}T_gT_{\bar{b}} + N_{g\bar{a}}N_{g\bar{b}} + C^2N_{g\bar{a}}{}^{n-n_0+1}T_gT_{\bar{a}}$ $+(n-n_0)(C^2N_{g\bar{a}}{}^{n-n_0} + (C^2 + N_{g\bar{a}})T_gT_{\bar{a}})$ $+(n-n_0)^2(C^2 + N_{g\bar{a}}))$
LINZIP Dense	$\mathcal{O}(C_{g\bar{a}}C_{g\bar{b}}$ $+C_g(C_{g\bar{a}}N_{g\bar{a}}{}^n + C_{\bar{a}}N_g{}^nN_{\bar{a}}{}^n + C_{g\bar{b}}N_{g\bar{b}}{}^n + C_{\bar{b}}N_g{}^nN_{\bar{b}}{}^n)$ $+C_gN_g{}^{n-1}(N_{g\bar{a}}N_{g\bar{b}} + nN_{g\bar{a}\bar{b}}{}^n) + C_gN_g{}^{3n-2})$
LINZIP Sparse $\mathcal{O}(T_g)$	$\mathcal{O}(C_{g\bar{a}}C_{g\bar{b}} + C_gT_g(C_{g\bar{a}}T_{\bar{a}} + C_{g\bar{b}}T_{\bar{b}})$ $+T_g(N_{g\bar{a}}N_{g\bar{b}} + n(N_{g\bar{a}\bar{b}} + T_gT_{\bar{a}\bar{b}}) + T_g{}^2)(C_g + nN_g))$
LINZIP Sparse $\mathcal{O}(T_g{}^{\frac{1}{2}})$	$\mathcal{O}(C_{g\bar{a}}C_{g\bar{b}} + C_gT_g(C_{g\bar{a}}T_{\bar{a}} + C_{g\bar{b}}T_{\bar{b}})$ $+T_g{}^{\frac{1}{2}}(N_{g\bar{a}}N_{g\bar{b}} + n(N_{g\bar{a}\bar{b}} + T_gT_{\bar{a}\bar{b}}) + T_g{}^{\frac{3}{2}})(C_g + nN_g))$
LINZIP Sparse $\mathcal{O}(1)$	$\mathcal{O}(C_{g\bar{a}}C_{g\bar{b}} + C_gT_g(C_{g\bar{a}}T_{\bar{a}} + C_{g\bar{b}}T_{\bar{b}})$ $+(N_{g\bar{a}}N_{g\bar{b}} + n(N_{g\bar{a}\bar{b}} + T_gT_{\bar{a}\bar{b}}))(C_g + nN_g))$

Table 6.7: General Asymptotic Results for GCD Algorithms

As a simplification of Table 6.7, and for purposes of discussion, we consider the balanced case, where the degree, density, and coefficients of the GCD and cofactors are equal, and we assume that in the sparse case the polynomials will appear sparse when viewed in 2 variables. The simplified results are presented in Table 6.8.

Brown Dense	$\mathcal{O}(C^2N^n + nCN^{n+1})$
Brown Sparse	$\mathcal{O}(C^2T^2 + CN^{n+1})$
DIVBRO Dense	$\mathcal{O}(C^2N^n + nCN^{n+1} + C^2N^{2n})$
DIVBRO Sparse	$\mathcal{O}((C^2 + n\log_2(n))T^2 + CN^n + nN^2)$
EEZ-GCD Dense Non-zero	$\mathcal{O}(C^2N^{2n+2})$
EEZ-GCD Dense Zero	$\mathcal{O}(C^2N^{2n+1})$
EEZ-GCD Sparse Non-zero	$\mathcal{O}(C^2(N^{n+1} + n)T^2)$
EEZ-GCD Sparse Zero $\mathcal{O}(1)$	$\mathcal{O}(N^2 + nC^2NT^2 + n^2(C^2 + N)T^2)$
EEZ-GCD Sparse Zero $\mathcal{O}(\frac{1}{2})$	$\mathcal{O}(N^2 + CNT^2 + n((C^2 + N)T^2 + CN) + n^2(C^2 + N))$
EEZ-GCD Sparse Mixed (n_0)	$\mathcal{O}(C^2N^{n-n_0+1}T^2 + (n - n_0)(C^2N^{n-n_0} + (C^2 + N)T^2)$ $+(n - n_0)^2(C^2 + N))$
LINZIP Dense	$\mathcal{O}(C^2N^{2n} + nCN^{2n-1} + CN^{3n-2})$
LINZIP Sparse $\mathcal{O}(T_g)$	$\mathcal{O}(C^2T^2 + T(N^2 + n(N + T^2))(C + nN))$
LINZIP Sparse $\mathcal{O}(T_g{}^{\frac{1}{2}})$	$\mathcal{O}(C^2T^2 + T^{\frac{1}{2}}(N^2 + n(N + T^2))(C + nN))$
LINZIP Sparse $\mathcal{O}(1)$	$\mathcal{O}(C^2T^2 + (N^2 + n(N + T^2))(C + nN))$

Table 6.8: Balanced Asymptotic Results for GCD Algorithms

So clearly Brown's algorithm gives the best asymptotic cost for balanced dense problems. Specifically the expense of Brown's algorithm scales as N^n, while the *EEZ-GCD* and *DIVBRO* algorithms scale as N^{2n}, and the *LINZIP* algorithm scales as N^{3n} (though this can be reduced to $\mathcal{O}(N^{2n})$, see Zippel [79, 80], and Kaltofen and Lee [32]).

The *EEZ-GCD* algorithm clearly gives the best asymptotic cost for sparse problems, but only in the cases where we can choose zero evaluation points. Alas, problems where this is not possible arise quite frequently in practice. Some obvious examples include homogeneous (or nearly homogeneous) polynomials, polynomials where the leading term of greatest degree has all variables present (e.g. leading term is $(x_1x_2x_3)^n$), and other less obvious cases where upon evaluating with respect to $\langle x_2, ..., x_n \rangle$ the g, $\bar{a}$ have a nontrivial GCD. Since the *EEZ-GCD* algorithm can be made to work with a mix of zero and non-zero evaluation points we can look at the results for that case above, which displays an interesting feature of the algorithm. As soon as *one* non-zero evaluation point is chosen, the expense is at least cubic in the degree of the problem (see the *EEZ-GCD Sparse Mixed* result with $n_0 = n - 2$, which simplifies to $\mathcal{O}(C^2N^3T^2)$), so the expense of the algorithm jumps by $\mathcal{O}(N^2)$ for a single non-zero evaluation point.

Perhaps in part because of this fact, and in part because of the frequency of this problem, some implementations don't bother to fully implement the zero case, simply doing a single check, and on failure utilizing all non-zero evaluation points. The unfortunate result of this is that the complexity becomes exponential in the degree, so the asymptotic cost begins to resemble that of Brown's algorithm. In practice, however, use of *EEZ-GCD* with non-zero evaluation points is significantly more efficient than Brown's algorithm, and this can be explained in that the exponential nature of *EEZ-GCD* comes from a single source (step 4.4.4 of Algorithm 25) at the outermost iteration, when we are lifting the final variable of the problem. In contrast, the contribution to the exponential nature of Brown's algorithm comes from all levels of the implementation.

The *LINZIP* algorithm does not suffer from the bad zero problem, so it can be used for any sparse problem, and it does not display the same exponential behavior of *EEZ-GCD* with non-zero evaluation points. In addition, the asymptotic complexity of *LINZIP* is fairly competitive with the complexity of *EEZ-GCD* when the latter requires even a single non-zero evaluation point. Also, the core computations for *LINZIP* are all performed $\bmod p$, so can be implemented in compiled code. Another aspect of this is that all coefficient operations for *EEZ-GCD* use full sized integers, while machine integers can be used for *LINZIP P*, so the factor of C^2 in the expense of *EEZ-GCD* can become significant. The author would like to note that variants of the *EEZ-GCD* algorithm that do not exhibit this $\mathcal{O}(C^2)$ behavior are possible, but then begin to more closely resemble the other algorithms, as a sequence of primes must be used at the top level, requiring Chinese remaindering, and it is expected that the data structure overhead would become more significant.

When comparing with existing implementations, in *Maple*, for example, the $\mathcal{O}(T^3)$ system solution can be implemented in the *LinearAlgebra:-Modular* package, as with the numerous univariate GCD computations required in the course of the algorithm. Though this does not change the asymptotic behavior of the algorithm, it does make a significant difference in the constants.

We will obtain highly efficient implementations of all GCD algorithms discussed in this thesis, and present an implementation comparison in a future work. In the implementations thus far, the asymptotic results appear to give an accurate description of the algorithm performance for large problems. In particular, some surprising results, such as the decoupling of the dependency of the *EEZ-GCD* sparse zero $\mathcal{O}(\frac{1}{2})$ case on the parameters of the problem, and the cubic degree dependence of the *EEZ-GCD* sparse mixed case with one nonzero evaluation, have been empirically verified.

Chapter 7

Benchmarks

In this chapter we focus on a comparison of the *RifSimp* algorithm implemented in *Maple* (*rifsimp*), to the *DiffElim* implementation in C. The benchmark problems are systems of determining equations for symmetries of ODE and PDE. These provide a scalable set of problems, for which the number of components in a system, the number of spatial dimensions, or both, can be varied to increase or decrease the difficulty of the system.

We have made a number of determining systems and benchmarks available on the web at `http://www.cecm.sfu.ca/~wittkopf/systems.html` [72], including:

- The classical Laplace equation for 3-16 spatial dimensions;
- The classical harmonic oscillator for 3-16 spatial dimensions;
- The VNLS system with potential $F = |u|^2$ for 1-4 components in 1-10 spatial dimensions;
- The VNLS system with general potential for 1-4 components in 1-10 spatial dimensions.

All but the last of the above systems are discussed in [74]. The last collection of systems has been provided for experimentation, allowing different forms of the potential to be used.

We note that the systems themselves are quite large, in some cases containing thousands of equations, thus making web distribution more appropriate than inclusion in this thesis. For brevity, we restrict the benchmarks in this chapter to the most challenging of the available systems, though as mentioned other benchmarks are available on the web.

7.1 Partially Coupled Reaction Diffusion Systems

The Lie infinitesimal determining systems for the partially coupled reaction diffusion systems given by

$$u_{1t} - u_{1xx} = a(u_1 + b)^2, \tag{7.1}$$

$$u_{it} - u_{ixx} = au_1u_i + \sum_{j=1}^{n} d_{i,j}u_j + c_i, \quad i = 2, ..., n \tag{7.2}$$

and discussed in §5.3.4 were used as a benchmark system for $n = 1, 2, ..., 7$ to compare the running time and memory use of *rifsimp* and *DiffElim*.

The results are summarized in Table 7.1, where memory is measured in megabytes (MB), and time in c.p.u. seconds running on a PII 333MHz Linux machine.

		rifsimp		DiffElim	
n	dim	Time	Mem	Time	Mem
1	3	2.41	2.56	–	0.69
2	6	4.63	2.69	0.01	0.71
3	11	9.55	3.00	0.04	0.74
4	18	19.93	3.12	0.09	0.81
5	27	42.72	3.56	0.26	0.92
6	38	91.28	4.12	0.53	1.03
7	51	202.99	4.50	0.81	1.43

Table 7.1: Run-time Comparison

For these problems, the *DiffElim* implementation has a running time between $\frac{1}{164}$ and $\frac{1}{250}$ of that of *rifsimp*, and requires less than $\frac{1}{3}$ the memory to perform differential elimination on these systems.

7.2 Statistics on speed, memory and relative performance

In this section, the family of VNLS equations (5.14) is used to test our implementation of the *DiffElim* algorithm and compare it to the *rifsimp* implementation.

7.2.1 Statistics on the input systems

When Hickman's *Maple* package *Symmetry* [25], was applied to the VNLS system for the general form of the potential, the numbers of determining equations it generated are listed in Table 7.2.

Spatial Dimension (n)	Number of components (m) 1	2	3	4
1	46	248	678	1408
2	120	624	1704	3552
3	222	1144	3126	6528
4	352	1808	4944	10336
5	510	2616	7158	14976
6	696	3568	9768	20448
7	910	4664	12744	26752
8	1152	5904	16176	33888

Table 7.2: Number of determining equations for the VNLS system (1.3)

It's nice to note that we were able to fit the number of PDE in the determining systems to an exact polynomial in m and n, $\mathrm{CountDets}(n, m)$, given by

$$\mathrm{CountDets}(n, m) = (4n^2 + 8n)m^3 + (10n^2 + 24n + 8)m^2 - 8m. \tag{7.3}$$

The total length of the PDE present in the input determining systems (as measured by *Maple*'s length function) is also recorded in Table 7.3.

Spatial Dimension (n)	Number of components (m) 1	2	3	4
1	5983	32239	96003	218539
2	12967	69611	209583	483283
3	23171	123711	373143	863147
4	37027	196459	591723	1368499
5	54967	289775	870363	2009707
6	77423	405579	1214103	2797139
7	104827	545791	1627983	3741163
8	137611	712331	2117043	4852147

Table 7.3: Length of determining systems for the VNLS system (1.3)

These are the lengths of the determining system for the arbitrary form of the potential

F in (1.3). Substitution of the particular form of F from (5.14) increases their size by $35\% - 60\%$.

It was also possible to fit this data to a polynomial function, LengthDets(n, m), that is cubic in n, and quartic in m:

$$\begin{aligned} \text{LengthDets}(m,n) &= (56n^2+112n)m^4 \\ &+ (16n^3+356n^2+876n+616)m^3 \\ &+ (40n^3+584n^2+1542n+1204)m^2 \\ &+ (16n^3+182n^2+416n-36)m+3. \end{aligned}$$

To obtain and verify this result, we also generated determining systems for $m = 5, 6$ which are not included in Tables 2 and 3. These relationships for the input equation count and total length hold *exactly* for all determining systems that were generated, which is quite surprising considering the sensitivity of *Maple*'s length function to mild changes in the form of the input.

Similar results were found for the classical Laplace equations and the oscillator Schrödinger equation, and are presented in Wittkopf and Reid [74].

7.2.2 Statistics on the output systems

Since the input equation counts and system sizes were found to be exact polynomial functions of m, n, it is natural to inquire whether the same is true for the output. This was found to be the case for the equation count for all systems successfully simplified, and this relationship is given by

$$\text{CountOut}(m,n) = \frac{1}{3}m^3 + \frac{5}{2}m^2 + (\frac{49}{6} + 4n)m + \frac{1}{6}n^3 + n^2 + \frac{17}{6}n + 1, \tag{7.4}$$

where additional systems for $m = 5, 6$ not present in the table were also used to validate the result.

The computed output equation counts are given in Table 7.4.

The total length of PDE present in the output determining systems, as measured by *Maple*'s length function, was also recorded, and is given in Table 7.5.

Spatial Dimension (n)	Number of components (m) 1	2	3	4
1	20	42	73	115
2	31	57	92	138
3	46	76	115	
4	66	100	143	
5	92	130		
6	125	167		
7	166			
8	216			

Table 7.4: Output Determining System Equation Counts for (5.14)

Spatial Dimension (n)	Number of components (m) 1	2	3	4
1	10312	55011	189599	493215
2	24782	101519	289161	673075
3	47177	166767	420947	
4	77578	250853	585057	
5	118698	359112		
6	168984	488021		
7	232633			
8	306252			

Table 7.5: Output Determining System Lengths (5.14)

Unfortunately it was not possible to polynomially fit the lengths of the output systems. For the dependence on n, it initially appears to be exponential for small n values, but this pattern seems to fail for larger n.

It is remarkable that the output length is always larger than the input length, but what is not so obvious is the structure of the output. Though larger, it is more useful, as it provides significant additional information that the input system does not (such as the dimension and structure of the symmetry groups).

7.2.3 Comparison of running times for rifsimp and DiffElim

We have recorded the running times for *rifsimp* and *DiffElim* in Table 7.6 as a function of m, n. In all tables, time is measured in c.p.u. seconds running on a PII 333 MHz PC under Linux. Any cases that failed due to insufficient memory (i.e. required more than 128 MB), are marked by **Mem**, while cases not attempted (as the result of the failure of an easier system for the same m) are blank.

Spatial Dimension (n)	Number of components (m) 1	2	3	4
1	0.1 10.1	2.5 1446.0	7.1 15238.6	77.4 **Mem**
2	0.2 29.0	3.6 709.2	36.1 18202.4	70.0
3	0.4 59.7	25.4 3065.7	106.0 **Mem**	**Mem**
4	0.7 115.2	26.7 **Mem**	309.5	
5	1.2 242.9	103.9 **Mem**	**Mem**	
6	1.9 451.6	309.5		
7	3.0 940.2	**Mem**		
8	4.5 1572.8			

Table 7.6: Run-time comparison between DiffElim (top) and rifsimp (bottom) for the reduction of the determining system of (5.14)

The average ratio of *DiffElim* running time, to *rifsimp* running time in the above table is about 400. The performance was noticed to be exponential in m and n, and a least squares fit of the data to the form $\mathrm{Time}(m,n) := C\exp(an)\exp(bm)$ gives

$$\begin{aligned} \text{rifsimp}: &\quad \mathrm{Time}(m,n) \approx 0.3\,\mathrm{e}^{3.4n}\,\mathrm{e}^{0.7m}, \\ \text{DiffElim}: &\quad \mathrm{Time}(m,n) \approx 0.006\,\mathrm{e}^{2.4n}\,\mathrm{e}^{0.7m}, \end{aligned}$$

where we have liberally rounded the coefficients in the formulae since they are based on only a few data points. Indeed the standard deviation of the linear fit of the log of the run time to the linear relation $an + bm + \ln(C)$ here is 0.96 for *DiffElim*, and 0.40 for *rifsimp*.

7.2.4 Comparison of Memory Use for rifsimp and DiffElim

The memory usage for *rifsimp* and *DiffElim*, as a function of m, n, is shown in Table 7.7. Again, any cases that failed due to insufficient memory (> 128 MB), are marked by **Mem**, while cases not attempted are blank.

Spatial Dimension (n)	Number of components (m) 1	2	3	4
1	0.8 3.1	1.7 24.1	2.6 51.9	16.3 **Mem**
2	0.8 3.9	2.2 12.7	10.5 201.0	10.4
3	0.9 4.6	6.3 31.2	23.5 **Mem**	**Mem**
4	1.1 6.1	6.3 **Mem**	19.9	
5	1.3 9.2	29.8	**Mem**	
6	1.5 11.4	48.3		
7	1.9 16.0	**Mem**		
8	2.5 20.9			

Table 7.7: Memory usage comparison between DiffElim (top) and rifsimp (bottom) for the reduction of the determining system of (5.14)

The average ratio of *DiffElim* memory use, to *rifsimp* memory use in the above table is about 9. The memory usage also appears to be exponential in m and n, and a least squares fit of the data to the form $\mathrm{Mem}(m,n) := C\exp(an)\exp(bm)$ gives

$$\begin{aligned} \text{rifsimp}: \quad \mathrm{Mem}(m,n) &\approx 0.4\,\mathrm{e}^{1.7n}\,\mathrm{e}^{0.3m}, \\ \text{DiffElim}: \quad \mathrm{Mem}(m,n) &\approx 0.1\,\mathrm{e}^{1.2n}\,\mathrm{e}^{0.3m}, \end{aligned}$$

where we make the same cautionary remarks about the fits as in the prior section.

7.3 Discussion

For the class of benchmark problems, the implementation of the *DiffElim* algorithm in the new *CDiffElim* environment outperformed *rifsimp* on average by a factor of 400, using significantly less memory in the process.

The *CDiffElim* environment has an equation data structure well suited to differential elimination, enhanced memory management, and fast algorithms for standard operations on ordered equations. In addition it was found that the enhanced GCD implementations described in Chapter 6, most specifically the *LINZIP* implementation (§6.6), provided an impressive performance boost over the algorithm previously implemented (an early variant of *DIVBRO* §6.4).

The *rifsimp* implementation, however, is *much* more flexible (to get some idea of this, simply consult the help pages in Appendix A), and is written in a multi-purpose environment, providing even greater flexibility.

The *CDiffElim* environment is still in a developmental stage, with many algorithms still needed (such as multivariate factorization), and much work remains to be done on interfacing and improving portability (currently only implemented under Linux).

In hindsight, though the *CDiffElim* environment shows substantial improvements over use of a general-purpose computer algebra system, the maintenance of such an environment, including integration of new algorithms, better ways of performing some computations, etc. requires an equally (if not more) substantial quantity of effort and work. The ideal approach would be the addition of some of the more relevant ideas *to* an already well tested and well established computer algebra system, thus providing many of the benefits of both approaches, while avoiding the primary drawbacks.

Chapter 8

Conclusions

Within this dissertation we provided a moderately detailed introduction to the area of differential elimination, describing key components for some of the various approaches used.

Building on the work of Rust [58], we provided a means of reducing the number of compatibility conditions (integrability conditions) that need be considered for nonlinear problems to be $\mathcal{O}(r^2)$ for r PDE in all cases. We reiterate that previously (in the useful case of non-sequential rankings) a potentially infinite number of conditions needed to be considered.

In addition we developed an easily implementable and highly efficient algorithm for identifying further redundant conditions in the $\mathcal{O}(r^2)$ set. Not previously noted is the fact that this algorithm is easily adapted for use by other algorithms, such as Gröbner basis computation, and is expected to provide significant efficiency gains there as well.

These results allow for a much more efficient implementation of the *RifSimp* algorithm because we avoid redundant conditions which were always of higher differential order, thus requiring significantly more work to reduce and eliminate.

We provided a detailed description of the *RifSimp* algorithm implementation in *Maple* as the *rifsimp* command [54], and recounted the core implementation details that result in the current efficient implementation of *RifSimp*. This was followed by a description of the *MaxDim* algorithm [53], and its use with *RifSimp* (though, as mentioned, it can be used with other differential elimination algorithms). The author would like to note that the *Rif* package has been part of the commercially distributed version of *Maple* since *Maple 6*, and the *MaxDim* algorithm has been part of *Rif* since *Maple 7*. It comes with an extensive set of help pages, available in the appendix, and provides efficient differential elimination, both for users, and as a tool utilized automatically by other parts of Maple (symbolic ODE solution for example).

We described the implementation of a specialized environment for differential elimination [73, 74], focusing on the core requirements for specialization to differential elimination applications. Within that environment, the first implementation of the *LINZIP* algorithm for efficient computation of sparse multivariate GCDs appeared. It was found that the addition of this GCD algorithm provided significant enhancements for many classes of problems, in some cases reducing their run-time by as much as a factor of 5. Clearly efficient GCD computation is a key part of differential elimination, which hardly needs to be stated given that some problems spend more than 80% of their time computing GCDs.

This led us to two variants of GCD algorithms, the *DIVBRO* algorithm [43], and the *LINZIP* algorithm. A detailed asymptotic comparison of these algorithms and two other well known algorithms is also provided, clearly indicating for which (large) classes of problems these new variants are the most relevant.

Following this, we utilized our differential elimination implementations in the area of Lie symmetry analysis, verifying some known results, and providing some new results as well [53, 73, 74].

And finally we concluded with a comparison of *rifsimp* in *Maple* and the *DiffElim* algorithm implemented in our specialized differential elimination environment, demonstrating that use of the specialized environment can enhance efficiency by approximately two orders of magnitude [73, 74]. In addition, timings and systems have been made available to others in the area of differential elimination as a means of testing or improving the efficiency of their algorithms [72].

There are, however still a few open questions and some work to do:

Conjecture #1: Can we use the Syzygy Simplify algorithm (Algorithm 9) incrementally?

Conjecture #2: Can we weaken the conditions for termination of the *poly_rif_ICs'* algorithm (Algorithm 10) for non-sequential rankings?

Project #1: Efficient implementation of all studied GCD algorithms in *Maple*, and a benchmark comparison relating to the complexity results of Chapter 6.

We will endeavor to discover the answers to these questions as part of our continuing research.

Appendix A

The *rifsimp* Implementation Help Pages

A.1 Introduction to the Rif Subpackage Version 1.1

Description

The Rif subpackage of `DEtools` is a powerful collection of commands for the simplification and analysis of systems of polynomially nonlinear ODEs and PDEs.

The package includes a command to simplify systems of ODEs and PDEs by converting these systems to a canonical form (reduced involutive form), graphical display of results for ease of use, a command to determine the initial data required for the existence of formal power series solutions of a system, and a command to generate formal power series solutions of a system.

Rif is used by both `dsolve` and `pdsolve` to assist in solution of ODE/PDE systems (see `dsolve,system`, `pdsolve,system`).

In addition, the `casesplit` command of the `PDEtools` package extends the functionality of Rif by allowing differential elimination to proceed in the presence of nearly all non-polynomial objects known to Maple (over one hundred of these, including trigonometric, exponential, and generalized hypergeometric functions, fractional exponents, etc.).

Though the results obtained from the `rifsimp` command are similar to those obtained from the `diffalg` package, the commands use different approaches, one of which may work better for specific problems than the other.

The Rif package generalizes the Standard Form package for linear ODE/PDE systems to polynomially nonlinear ODE/PDE. The linear system capabilities of the Standard Form package for the simplification of ODE/PDE systems are also present as part of Rif.

The improvements over the most recently released version of Standard Form (1995) include

```
1) Full handling of polynomially nonlinear systems
2) Automatic case splitting
3) Flexible nonlinear ranking
4) Handling of inequation constraints (expr<>0)
5) Speed and memory efficiency
```

The improvements over the release 1.0 of Rif (Maple 6) include

```
1) New function maxdimsystems to find the most general solutions for case
   splitting problems
2) Improvements to case visualization and initial data computation
3) Greater flexibility in rifsimp with the addition of new options for
   control of case splitting, declaration of arbitrary functions and/or
   constants, detection of empty cases, and much more
4) More efficient handling of nonlinear systems via new nonlinear equation
   methods
5) Significant overall speed and memory enhancements
6) Automatic adjustment of results to remove inconsistent cases, and
   their effect on the returned consistent cases.
```

The functions available

```
rifsimp          Simplifies systems of polynomially nonlinear ODEs and PDEs
                 to canonical form. Splits nonlinear equations into cases,
                 using Groebner basis techniques to handle algebraic
                 consequences of the system. Accounts for all differential
                 consequences of the system.

maxdimsystems    Also simplifies systems of polynomially nonlinear ODEs
                 and PDEs, but performs case splitting automatically,
                 returning the most general cases (those with the highest
                 number of parameters in the initial data).

rifread          Loads a partially completed rifsimp calculation for
                 viewing and/or manual manipulation. rifsimp must be
                 told to save partial calculations using the storage
                 options.

checkrank        Provides information on ranking to allow determination
                 of an appropriate ranking to use with rifsimp.
```

```
caseplot          Takes the case split output of rifsimp, and provides
                  a graphical display of the solution case tree.

initialdata       Obtains the initial data required by an ODE/PDE system to
                  fully specify formal power series solutions of the system.
                  Typically the output of rifsimp is used as input for this
                  procedure, but any ODE/PDE system in the correct form can
                  be used.

rtaylor           Calculates the Taylor series of solutions of an ODE/PDE
                  system to any order. Just as for initialdata, any
                  ODE/PDE system in the correct form can be used.
```

For theory used to produce the **rifsimp** and **maxdimsystems** algorithm, and related theory, please see the following:

Becker, T., Weispfenning, V. *Groebner Bases: A Computational Approach to Commutative Algebra.* New York: Springer-Verlag, 1993.

G.W. Bluman and S. Kumei, "Symmetries and Differential Equations", *Springer-Verlag*, vol. 81.

Boulier, F., Lazard, D., Ollivier, F., and Petitot, M. "Representation for the Radical of a Finitely Generated Differential Ideal". *Proc. ISSAC 1995.* ACM Press, 158-166.

Carra-Ferro, G. "Groebner Bases and Differential Algebra". *Lecture Notes in Comp. Sci.* 356 (1987): 128-140.

Goldschmidt, H. "Integrability Criteria for Systems of Partial Differential Equations". *J. Diff. Geom.* 1 (1967): 269-307.

Mansfield, E. 1991. Differential Groebner Bases. Ph.D. diss., University of Sydney.

Ollivier, F. "Standard Bases of Differential Ideals". *Lecture Notes in Comp. Sci.* 508 (1991): 304-321.

G.J. Reid and A.D. Wittkopf, "Determination of Maximal Symmetry Groups of Classes of Differential Equations", *Proc. ISSAC 2000.* ACM Press, 272-280

Reid, G.J., Wittkopf, A.D., and Boulton, A. "Reduction of Systems of Nonlinear Partial Differential Equations to Simplified Involutive Forms". *Eur. J. Appl. Math. 7* (1996): 604-635.

Rust, C.J. 1998. "Rankings of Derivatives for Elimination Algorithms, and Formal Solvability of Analytic PDE". Ph.D. diss., University of Chicago.

Rust, C.J., Reid, G.J., and Wittkopf, A.D. "Existence and Uniqueness Theorems for Formal Power Series Solutions of Analytic Differential Systems". *Proc. ISSAC 1999.* ACM Press, 105-112.

For a review of other algorithms and software (but more closely tied to symmetry analysis), please see the following.

Hereman, W. "Review of Symbolic Software for the Computation of Lie Symmetries of Differential Equations". *Euromath Bull.* 1 (1994): 45-79.

For a detailed guide to the use the Standard Form package, the predecessor of **rifsimp**, please see:

Reid, G.J. and Wittkopf, A.D. "The Long Guide to the Standard Form Package". 1993. Programs and documentation available on the web at http://www.cecm.sfu.ca/~wittkopf.

A.2 rifsimp - simplify overdetermined polynomially nonlinear PDE or ODE systems

Calling Sequences

rifsimp(**system**, **options**)

rifsimp(**system**, **vars**, **options**)

Parameters

system - list or set of polynomially nonlinear PDEs or ODEs (may contain inequations)

vars - (optional) list of the main dependent variables

options - (optional) sequence of options to control the behavior of **rifsimp**

Description

The **rifsimp** function can be used to simplify or rework overdetermined systems of polynomially nonlinear PDEs or ODEs and inequations to a more useful form – relative standard form. The **rifsimp** function does not solve PDE systems, but provides existence and uniqueness information, and can be used as a first step to their solution. As an example, inconsistent systems can be detected by **rifsimp**.

Basically, given an input PDE system, and a list of dependent variables or constants to solve for, **rifsimp** returns the simplified PDE system along with any existence conditions required for the simplified system to hold.

Detailed examples of the use of **rifsimp** for various systems (along with some explanation of the algorithm) can be found in `rifsimp,overview`.

Other options are sometimes required along with the specification of the system and its solving variables. For common options, please see `rifsimp[options]`, and for more advanced use, please see `rifsimp[adv_options]`.

For a description of all possible output configurations, see `rifsimp,output`.

Examples

1. Overdetermined Systems

As a first example, we have the overdetermined system of two equations in one dependent variable **y(x)**.

```
> sys1:=[x*diff(y(x),x)^2*diff(y(x),x,x)^2-2*x*diff(y(x),x)*
> diff(y(x),x,x)*y(x)*diff(y(x),x,x,x)+x*y(x)^2*
> diff(y(x),x,x,x)^2-y(x)*diff(y(x),x,x)+diff(y(x),x)^2=0,
> -diff(y(x),x)*diff(y(x),x,x)+y(x)*diff(y(x),x,x,x)+
> 2*y(x)^2*diff(y(x),x,x)^2-4*y(x)*diff(y(x),x,x)*
> diff(y(x),x)^2+2*diff(y(x),x)^4=0];
```

$$\mathit{sys1} := [x\,(\tfrac{\partial}{\partial x}\,\mathrm{y}(x))^2\,(\tfrac{\partial^2}{\partial x^2}\,\mathrm{y}(x))^2 - 2\,x\,(\tfrac{\partial}{\partial x}\,\mathrm{y}(x))\,(\tfrac{\partial^2}{\partial x^2}\,\mathrm{y}(x))\,\mathrm{y}(x)\,(\tfrac{\partial^3}{\partial x^3}\,\mathrm{y}(x))$$
$$+\,x\,\mathrm{y}(x)^2\,(\tfrac{\partial^3}{\partial x^3}\,\mathrm{y}(x))^2 - \mathrm{y}(x)\,(\tfrac{\partial^2}{\partial x^2}\,\mathrm{y}(x)) + (\tfrac{\partial}{\partial x}\,\mathrm{y}(x))^2 = 0, -(\tfrac{\partial}{\partial x}\,\mathrm{y}(x))\,(\tfrac{\partial^2}{\partial x^2}\,\mathrm{y}(x))$$
$$+\,\mathrm{y}(x)\,(\tfrac{\partial^3}{\partial x^3}\,\mathrm{y}(x)) + 2\,\mathrm{y}(x)^2\,(\tfrac{\partial^2}{\partial x^2}\,\mathrm{y}(x))^2 - 4\,\mathrm{y}(x)\,(\tfrac{\partial^2}{\partial x^2}\,\mathrm{y}(x))\,(\tfrac{\partial}{\partial x}\,\mathrm{y}(x))^2 + 2\,(\tfrac{\partial}{\partial x}\,\mathrm{y}(x))^4 =$$
$$0]$$

Now call **rifsimp**.

```
> with(DEtools):
> ans1:=rifsimp(sys1);
```

$$\mathit{ans1} := \mathrm{table}([\mathit{Solved} = \left[\frac{\partial^2}{\partial x^2}\,\mathrm{y}(x) = \frac{(\frac{\partial}{\partial x}\,\mathrm{y}(x))^2}{\mathrm{y}(x)}\right],\ \mathit{Case} = [[\mathrm{y}(x) \neq 0,\ \tfrac{\partial^3}{\partial x^3}\,\mathrm{y}(x)]],$$
$$\mathit{Pivots} = [\mathrm{y}(x) \neq 0]$$
$$])$$

We see that the system has been reduced (for the case **y(x)** not identically zero) to one much simpler equation that can now be handled with Maple's **dsolve**.

```
> dsolve(convert(ans1[Solved],'set'),{y(x)});
```

$$\{\mathrm{y}(x) = e^{(\_C1\,x)}\,\_C2\}$$

In addition to assisting **dsolve** in obtaining exact solutions, the simplified form can be used in combination with the `initialdata` function to obtain the required initial data for the solved form.

```
> id1:=initialdata(ans1[Solved]);
```

$$\mathit{id1} := \mathrm{table}([\mathit{Infinite} = [],\ \mathit{Finite} = [\mathrm{y}(x_0) = \_C1,\ \tfrac{\partial}{\partial x}\,\mathrm{y}(x_0) = \_C2]])$$

In this case, it gives us the expected results for a second order ODE, but it can also calculate the required initial data for more complex PDE systems (see `initialdata` for more information). Numerical methods can now be successfully applied to the reduced system, with initial conditions of the type calculated by **initialdata**.

```
> dsolve(convert(ans1[Solved],set) union {y(0)=1,D(y)(0)=2},
> {y(x)}, numeric, output=array([0,0.25,0.5,0.75,1.0]));
```

$$\left[\begin{array}{c} \left[x,\ \mathrm{y}(x),\ \frac{\partial}{\partial x}\,\mathrm{y}(x)\right] \\ \left[\begin{array}{ccc} 0 & 1. & 2. \\ .25 & 1.64870721760397 & 3.29741443520794 \\ .5 & 2.71795936016842 & 5.43591872033684 \\ .75 & 4.48001357532832 & 8.96002715065664 \\ 1.0 & 7.38492244417078 & 14.7698448883416 \end{array}\right] \end{array}\right]$$

Finally, we can also obtain a formal power series (Taylor series) for the local solution of this problem (see `rtaylor` for more information).

```
> tay_ser:=rtaylor(ans1[Solved],order=4);
```

$$\mathit{tay\_ser} := \left[\mathrm{y}(x) = \mathrm{y}(x_0) + (\frac{\partial}{\partial x}\,\mathrm{y}(x_0))\,(x - x_0) + \frac{1}{2}\,\frac{(\frac{\partial}{\partial x}\,\mathrm{y}(x_0))^2\,(x-x_0)^2}{\mathrm{y}(x_0)} + \frac{1}{6}\,\frac{(\frac{\partial}{\partial x}\,\mathrm{y}(x_0))^3\,(x-x_0)^3}{\mathrm{y}(x_0)^2} + \frac{1}{24}\,\frac{(\frac{\partial}{\partial x}\,\mathrm{y}(x_0))^4\,(x-x_0)^4}{\mathrm{y}(x_0)^3}\right]$$

The above series is fully determined when given the initial data described earlier:

```
> tay_ser:=rtaylor(ans1[Solved],id1,order=4);
```

$$\mathit{tay\_ser} := [\mathrm{y}(x) = \mathit{\_C1} + \mathit{\_C2}\,(x - x_0) + \frac{1}{2}\,\frac{\mathit{\_C2}^2\,(x-x_0)^2}{\mathit{\_C1}} + \frac{1}{6}\,\frac{\mathit{\_C2}^3\,(x-x_0)^3}{\mathit{\_C1}^2} + \frac{1}{24}\,\frac{\mathit{\_C2}^4\,(x-x_0)^4}{\mathit{\_C1}^3}]$$

2. Inconsistent Systems

rifsimp can often determine whether a system is inconsistent.

```
> sys2:=[diff(u(x,y),x)^2+diff(u(x,y),y,y)=0,
> u(x,y)^3-diff(u(x,y),x)+x=0]:
> rifsimp(sys2);
```

$$\mathrm{table}([\mathit{status} = \text{“system is inconsistent”}])$$

3. Constrained Mechanical Systems

This example shows the use of **rifsimp** as a preprocessor for a constrained mechanical system (that is, a Differential-Algebraic Equation or DAE system). The method of Lagrange formulates the motion of a bead of nonzero mass **m** on a frictionless wire of shape **Phi(x,y)** under gravity as follows:

```
> ConstrainedSys:= [m<>0, Phi(x(t),y(t))=0,
> m*diff(x(t),t,t)=-lambda(t)*D[1](Phi)(x(t),y(t)),
> m*diff(y(t),t,t)=-lambda(t)*D[2](Phi)(x(t),y(t))-m*g];
```

$$\mathit{ConstrainedSys} := [m \neq 0,\ \Phi(\mathrm{x}(t),\, \mathrm{y}(t)) = 0,\ m\,(\tfrac{\partial^2}{\partial t^2}\,\mathrm{x}(t)) = -\lambda(t)\,\mathrm{D}_1(\Phi)(\mathrm{x}(t),\, \mathrm{y}(t)),\ m\,(\tfrac{\partial^2}{\partial t^2}\,\mathrm{y}(t)) = -\lambda(t)\,\mathrm{D}_2(\Phi)(\mathrm{x}(t),\, \mathrm{y}(t)) - m\,g]$$

Note that a mass falling under gravity without air resistance corresponds to:

```
> Phi:= (x,y) -> 0;
```

$$\Phi := 0$$

We pick the example of a pendulum, so the bead moves on a circular wire.

```
> Phi:= (x,y) -> x^2 + y^2 - 1;
```

$$\Phi := (x,\, y) \to x^2 + y^2 - 1$$

```
> ConstrainedSys;
```

$$[m \neq 0,\ \mathrm{x}(t)^2 + \mathrm{y}(t)^2 - 1 = 0,\ m\,(\tfrac{\partial^2}{\partial t^2}\,\mathrm{x}(t)) = -2\,\lambda(t)\,\mathrm{x}(t),\ m\,(\tfrac{\partial^2}{\partial t^2}\,\mathrm{y}(t)) = -2\,\lambda(t)\,\mathrm{y}(t) - m\,g]$$

Such constrained systems present great difficulties for numerical solvers. Of course, we could eliminate the constraint using polar coordinates, but in general this is impossible for mechanical systems with complicated constraints. For example, just replace the constraint with another function, so that the wire has a different shape.

```
> Phi2:= (x,y) -> x^4 + y^4 - 1;
```

$$\Phi 2 := (x,\, y) \to x^4 + y^4 - 1$$

(See Reid *et al.*, 1996, given in the reference and package information page `Rif`). The pendulum is the classic example of such systems that have been the focus of much recent research due to their importance in applications.

```
> SimpSys:=rifsimp(ConstrainedSys);
```

$SimpSys :=$ table([$Constraint =$
$[m\,(\frac{\partial}{\partial t}\,\mathrm{y}(t))^2 + 2\,\mathrm{y}(t)^2\,\lambda(t) - \mathrm{y}(t)\,m\,g + \mathrm{y}(t)^3\,m\,g - 2\,\lambda(t) = 0,\ \mathrm{x}(t)^2 + \mathrm{y}(t)^2 - 1 = 0],$
$Solved =$
$$\left[\frac{\partial^2}{\partial t^2}\,\mathrm{y}(t) = -\frac{2\,\lambda(t)\,\mathrm{y}(t) + m\,g}{m},\ \frac{\partial}{\partial t}\,\lambda(t) = -\frac{3}{2}\,(\frac{\partial}{\partial t}\,\mathrm{y}(t))\,m\,g,\ \frac{\partial}{\partial t}\,\mathrm{x}(t) = -\frac{\mathrm{y}(t)\,(\frac{\partial}{\partial t}\,\mathrm{y}(t))}{\mathrm{x}(t)}\right],$$
$Case = [[\mathrm{x}(t) \neq 0,\ \frac{\partial}{\partial t}\,\mathrm{x}(t)],\ [(\mathrm{y}(t) - 1)\,(\mathrm{y}(t) + 1) \neq 0,\ \frac{\partial}{\partial t}\,\lambda(t)]],$
$Pivots = [m \neq 0,\ \mathrm{x}(t) \neq 0,\ \mathrm{y}(t) - 1 \neq 0,\ \mathrm{y}(t) + 1 \neq 0]$
])

```
> initialdata(SimpSys);
```

table([$Constraint = [$
$\_C6\,\_C4^2 + 2\,\_C3^2\,\_C1 - \_C3\,\_C6\,\_C5 + \_C3^3\,\_C6\,\_C5 - 2\,\_C1 = 0,$
$\_C2^2 + \_C3^2 - 1 = 0],\ Infinite = [],$
$Finite = [\lambda(t_0) = \_C1,\ \mathrm{x}(t_0) = \_C2,\ \mathrm{y}(t_0) = \_C3,\ \frac{\partial}{\partial t}\,\mathrm{y}(t_0) = \_C4,\ g = \_C5,\ m = \_C6],$
$Pivots = [\_C6 \neq 0,\ \_C2 \neq 0,\ \_C3 - 1 \neq 0,\ \_C3 + 1 \neq 0]$
])

Although there is no simplification as in the previous examples, **rifsimp** has found additional constraints that the initial conditions must satisfy (**Constraint** and **Pivots** in the **initialdata** output), and obtained an equation for the time derivative of **lambda(t)**. Maple's `dsolve[numeric]` can then be used to obtain a numerical solution, in the manner of the previous example.

4. Lie Symmetry Determination

This example shows the use of **rifsimp** to assist in determination of Lie point symmetries of an ODE. Given the ODE:

```
> ODE:=2*x^2*y(x)*diff(y(x),x,x)-x^2*(diff(y(x),x)^2+1)+y(x)^2;
```

$$ODE := 2\,x^2\,\mathrm{y}(x)\,(\frac{\partial^2}{\partial x^2}\,\mathrm{y}(x)) - x^2\,((\frac{\partial}{\partial x}\,\mathrm{y}(x))^2 + 1) + \mathrm{y}(x)^2$$

We can obtain the system of PDE for it's Lie point symmetries via:

```
> sys := {gensys(ODE, [xi,eta](x,y))};
```

$$
\begin{aligned}
&sys := \{\eta(x,\,y)\,x\,y^2 + \eta(x,\,y)\,x^3 - 2\,x^3\,y\,(\tfrac{\partial}{\partial x}\,\xi(x,\,y)) + 2\,x\,y^3\,(\tfrac{\partial}{\partial x}\,\xi(x,\,y))\\
&+ x^3\,y\,(\tfrac{\partial}{\partial y}\,\eta(x,\,y)) - x\,y^3\,(\tfrac{\partial}{\partial y}\,\eta(x,\,y)) - 2\,\xi(x,\,y)\,y^3 + 2\,(\tfrac{\partial^2}{\partial x^2}\,\eta(x,\,y))\,x^3\,y^2,\\
&- 3\,x^3\,y\,(\tfrac{\partial}{\partial y}\,\xi(x,\,y)) + 3\,x\,y^3\,(\tfrac{\partial}{\partial y}\,\xi(x,\,y)) - 2\,x^3\,y\,(\tfrac{\partial}{\partial x}\,\eta(x,\,y))\\
&+ 4\,(\tfrac{\partial^2}{\partial y\,\partial x}\,\eta(x,\,y))\,x^3\,y^2 - 2\,(\tfrac{\partial^2}{\partial x^2}\,\xi(x,\,y))\,x^3\,y^2,\\
&\eta(x,\,y)\,x^3 - x^3\,y\,(\tfrac{\partial}{\partial y}\,\eta(x,\,y)) + 2\,(\tfrac{\partial^2}{\partial y^2}\,\eta(x,\,y))\,x^3\,y^2 - 4\,(\tfrac{\partial^2}{\partial y\,\partial x}\,\xi(x,\,y))\,x^3\,y^2,\\
&-x^3\,y\,(\tfrac{\partial}{\partial y}\,\xi(x,\,y)) - 2\,(\tfrac{\partial^2}{\partial y^2}\,\xi(x,\,y))\,x^3\,y^2\}
\end{aligned}
$$

We can then use **rifsimp** to greatly simplify this system:

```
> rifsys:=rifsimp(sys);
```

$$
\begin{aligned}
&rifsys := \mathrm{table}([\\
&Solved = [\tfrac{\partial^2}{\partial x^2}\,\eta(x,\,y) = \frac{-\eta(x,\,y)\,x + \xi(x,\,y)\,y}{x^3},\ \tfrac{\partial}{\partial x}\,\xi(x,\,y) = \frac{\eta(x,\,y)}{y},\\
&\tfrac{\partial}{\partial y}\,\eta(x,\,y) = \frac{\eta(x,\,y)}{y},\ \tfrac{\partial}{\partial y}\,\xi(x,\,y) = 0]\\
&])
\end{aligned}
$$

From this point, **pdsolve** can be used to obtain the infinitesimals for the Lie point symmetries via:

```
> pdsolve(rifsys['Solved']);
```

$$
\begin{aligned}
&\{\xi(x,\,y) = -x\,\_C2 + 2\,x\,\_C3 - 2\,x\,\_C3\,\ln(x) + \_C1\,x + x\,\_C2\,\ln(x) + x\,\_C3\,\ln(x)^2,\\
&\eta(x,\,y) = (\_C1 + \_C2\,\ln(x) + \_C3\,\ln(x)^2)\,y\}
\end{aligned}
$$

A.3 rifsimp[overview] - description of algorithm concepts and use

Description

The **rifsimp** algorithm is essentially an extension of the Gaussian elimination and Groebner basis algorithms to systems of nonlinear PDEs.

This help page presents successively more involved examples of the use of **rifsimp** for the simplification of systems of equations, explaining what the algorithm does, and introducing ideas that are helpful for effective use of the algorithm.

Examples

Example 1: **rifsimp** form of a linear algebraic system

```
> with(DEtools):
> rifsimp([x - 2*y = 1, x + 2*z = 4]);
```

$$\mathrm{table}([Solved = [x = -2\,z + 4,\, y = \frac{3}{2} - z]])$$

This is the same result that one would expect from the Gaussian elimination algorithm. Note that **rifsimp** has chosen to solve for **x** and **y** in terms of **z**. For this problem the order of solving, called a *ranking*, is **x** > **y** > **z** (the default ranking for **rifsimp**). Given the inequality, **rifsimp** looks at the unknowns in an equation, and chooses the greatest one to solve for. This is a purely illustrative example, because Maple has specialized functions that solve such systems.

Example 2: **rifsimp** form of a linear ODE system

```
> sys:= [3*x(t)-diff(x(t),t)=0, diff(x(t),t,t)-2*x(t)=0];
```

$$sys := [3\,\mathrm{x}(t) - (\frac{\partial}{\partial t}\,\mathrm{x}(t)) = 0,\, (\frac{\partial^2}{\partial t^2}\,\mathrm{x}(t)) - 2\,\mathrm{x}(t) = 0]$$

```
> rifsimp(sys);
```

$$\mathrm{table}([Solved = [\mathrm{x}(t) = 0]])$$

As described in the prior example, **rifsimp** requires a ranking to determine the unknown for which to solve in an equation. For this ODE system, the ranking (using prime notation for derivatives) defaults to the following:

```
x  <  x' <  x'' <  x'''  <  ...
```

So the first equation is solved for **x'** giving **x' = 3*x**. The second equation is then differentially simplified with respect to the first, and gives **3*x'-2*x = 0** after the first simplification, and finally **7*x = 0** after the second. This equation is then solved for **x**, then back substitution into the first equation gives **0 = 0**.

For more general systems, **rifsimp** accounts for all of the derivative consequences of a system in a systematic and rigorous way.

Example 3: **rifsimp** form of a nonlinear algebraic system

```
> sys := [x^10-2*x^5*y^2+y^4-y^3+1, 2*x^10-4*x^5*y^2+2*y^4+3*y^3-3,
> x^5-y^2+y^3-1];
```

$$sys := [x^{10} - 2\,x^5\,y^2 + y^4 - y^3 + 1,\, 2\,x^{10} - 4\,x^5\,y^2 + 2\,y^4 + 3\,y^3 - 3,\, x^5 - y^2 + y^3 - 1]$$

```
> rifsimp(sys);
```

$$\mathrm{table}([\mathit{Constraint} = [x^5 - y^2 = 0,\, -1 + y^3 = 0]])$$

The output was returned in the **Constraint** entries which means algebraic constraints as opposed to differential constraints. For this system, **rifsimp** gives the Groebner basis form of the algebraic system as output, as can be seen by running the **Groebner** package on the same system (with the appropriate ranking).

```
> Groebner[gbasis](sys,tdeg(x,y));
```

$$[-1 + y^3,\, x^5 - y^2]$$

In general, **rifsimp** on algebraic systems is closer to a factoring case splitting Groebner basis (see `rifsimp[nonlinear]`, and Maple's `Groebner` package). **rifsimp** does not use Maple's **Groebner** package, but instead its own, optimized for differential elimination.

Example 4: **rifsimp** form of a nonlinear ODE system

```
> sys:= [x(t)^3 + x(t) + 1 = 0,diff(x(t),t,t)-3*diff(x(t),t)=0,
> diff(x(t),t,t,t)-2*diff(x(t),t)=0];
```

$$sys := [\mathrm{x}(t)^3 + \mathrm{x}(t) + 1 = 0,\, (\tfrac{\partial^2}{\partial t^2}\,\mathrm{x}(t)) - 3\,(\tfrac{\partial}{\partial t}\,\mathrm{x}(t)) = 0,\, (\tfrac{\partial^3}{\partial t^3}\,\mathrm{x}(t)) - 2\,(\tfrac{\partial}{\partial t}\,\mathrm{x}(t)) = 0]$$

```
> rifsimp(sys);
```

$$\mathrm{table}([\mathit{Solved} = [\tfrac{\partial}{\partial t}\,\mathrm{x}(t) = 0],\, \mathit{Constraint} = [\mathrm{x}(t)^3 + \mathrm{x}(t) + 1 = 0]])$$

Now the output is present in both the **Solved** and **Constraint** lists. The equation is present in the constraint list because the leading indeterminate of the equation occurs nonlinearly (for information on the leading indeterminate, please see `rifsimp[ranking]`). Again, to obtain the answer, a ranking of the derivatives was required. The ranking used ranks "higher derivatives in **x(t)** before lower derivatives". This is a fairly standard ranking called the *total degree ranking.*

It is clear that the solution of the constraint equation gives **x(t) = constant**, so from that point of view, the equation in the derivative of **x(t)** is redundant, but it needs to be retained for the proper functioning of other algorithms, such as `initialdata`, and `rtaylor`. The additional equation in the **Solved** is a differential consequence of the **Constraint**, as can be verified through differentiation. This redundant equation is called a *spawn* equation, as it is a differential constraint spawned (through differentiation by t) from the nonlinear equation. Redundant equations can be removed with the **clean** or **fullclean** options as follows (see `rifsimp[options]`):

```
> rifsimp(sys,clean=[spawn]);
```

$$\mathrm{table}([\mathit{Constraint} = [\mathrm{x}(t)^3 + \mathrm{x}(t) + 1 = 0]])$$

```
> rifsimp(sys,fullclean);
```

$$\mathrm{table}([\mathit{Constraint} = [\mathrm{x}(t)^3 + \mathrm{x}(t) + 1 = 0]])$$

Example 5: **rifsimp** form of a linear PDE system

```
> sys:= [diff(u(x,y),x,x) - y*diff(u(x,y),x) = 0,
> diff(u(x,y),y) = 0];
```

$$sys := [(\frac{\partial^2}{\partial x^2}\,\mathrm{u}(x,\,y)) - y\,(\frac{\partial}{\partial x}\,\mathrm{u}(x,\,y)) = 0,\ \frac{\partial}{\partial y}\,\mathrm{u}(x,\,y) = 0]$$

```
> rifsimp(sys);
```

$$\mathrm{table}([\mathit{Solved} = [\frac{\partial}{\partial x}\,\mathrm{u}(x,\,y) = 0,\ \frac{\partial}{\partial y}\,\mathrm{u}(x,\,y) = 0]])$$

In this example, **rifsimp** used the default ranking (where indices are used to denote differentiation)

```
u < u[x] < u[y] < u[xx] < u[xy] < u[yy] < ...
```

This ranking satisfies the following properties:

1. It is preserved under differentiation (**u** < **u[x]** implies **u[x]** < **u[xx]**).

2. **u** is less than any derivative of **u**.

These two properties are crucial for the termination of the algorithm. Fortunately, the default rankings chosen by **rifsimp** obey these properties, so these do not have to be considered. In general, many such orderings are possible, and methods for specifying them are discussed in `rifsimp[ranking]` and `checkrank`.

As an illustration, we will manually simplify the above system to **rifsimp** form. As the first step, we solve each equation for its leading derivative (given the ranking described above). This gives **u[xx] = y u[x]** and **u[y] = 0**. Alas, we do not have our final answer, but no differential simplifications remain. How is this possible? Well, for PDE systems we have one further consideration called integrability conditions.

After looking at our solved system for a time, we may realize that differentiation of the first equation by **y** and differentiation of the second equation by **x** twice will give the derivative **u[xyy]** on the left hand side of both equations. If the solution to the system is differentiable to third order (an assumption implicitly made by **rifsimp**), we differentiate the equations to the described order and subtract, giving the new, nontrivial equation

```
0 = D[y](u[xx])-D[xx](u[y]) = D[y](y u[x])+D[xx](0) = y u[xy]+u[x]
```

Differential simplification of this equation with respect to the current system yields **u[x] = 0**, which is then used to simplify the system and give us the answer returned by **rifsimp**.

To summarize, for systems of linear PDEs, the **rifsimp** form of the system can be obtained by the following heavily simplified algorithm, for a given input ranking $<$:

```
Rifsimp-Linear(input: system, <)
   neweqns := system
   rifsys := empty
   while neweqns not empty do
      rifsys := rifsys union neweqns
      rifsys := diff-gauss-elim(rifsys,rifsys,<)
      neweqns := integrability-cond(rifsys)
      neweqns := diff-gauss-elim(neweqns,rifsys,<)
   end-while
   return(rifsys)
end
```

Care must be taken in how the diff-gauss-elim routine operates on the solved rifsys.

Example 6: **rifsimp** form of a nonlinear PDE system

This system is considered in the paper Reid *et al.*, 1996 (see `Rif`) and is used as an example in Rust, 1998. For the ranking **u** < **u[x]** < **u[y]** < **u[xx]** < **u[xy]** < **u[yy]** < **...**, the initial system consists of one leading linear PDE, **u[xx]-u[xy] = 0** (with leading derivative **u[xx]**), and one leading nonlinear, PDE **u[y]^2+u[y]-u = 0** (with leading derivative **u[y]**). No linear elimination can be done in the single leading linear PDE, so the leading nonlinear PDE is differentiated with respect to both independent variables to spawn the two leading linear PDEs: **(2 u[y]+1) u[xy]-u[x] = 0** and **(2 u[y]+1) u[yy]-u[y] = 0** (for more information on spawning, see `rifsimp[nonlinear]`). These are solved for their leading derivatives, yielding **u[xy] = u[x]/(2 u[y]+1)** and **u[yy] = u[y]/(2 u[y]+1)**, subject to the pivot or inequation condition **2 u[y]+1 <> 0**. Integrability conditions are then taken across the leading linear PDE and simplified subject to the system, yielding one more PDE: **u[x] u[y]-u[x]^2 = 0**. When this equation is spawned, the resulting equations vanish to zero upon reduction by the current system, so the algorithm then terminates. In summary, constraints are treated algebraically, and the leading linear equations are treated differentially. The system is then viewed modulo the constraint equations, using the concept of a *relative basis*, rigorously described in the thesis of Rust (see references on the `Rif` help page).

```
> sys:= [diff(u(x,y),x,x) - diff(u(x,y),x,y) = 0,
> diff(u(x,y),y)^2 + diff(u(x,y),y) - u(x,y) = 0];
```

$$sys := [(\frac{\partial^2}{\partial x^2}\,\mathrm{u}(x,\,y)) - (\frac{\partial^2}{\partial y\,\partial x}\,\mathrm{u}(x,\,y)) = 0,\ (\frac{\partial}{\partial y}\,\mathrm{u}(x,\,y))^2 + (\frac{\partial}{\partial y}\,\mathrm{u}(x,\,y)) - \mathrm{u}(x,\,y) = 0]$$

```
> rifsimp(sys);
```

$$\mathrm{table}([\mathit{Solved} = \left[\frac{\partial^2}{\partial x^2}\mathrm{u}(x,y) = \frac{\frac{\partial}{\partial x}\mathrm{u}(x,y)}{\%1},\ \frac{\partial^2}{\partial y\partial x}\mathrm{u}(x,y) = \frac{\frac{\partial}{\partial x}\mathrm{u}(x,y)}{\%1},\ \frac{\partial^2}{\partial y^2}\mathrm{u}(x,y) = \frac{\frac{\partial}{\partial y}\mathrm{u}(x,y)}{\%1}\right],$$

$$\mathit{Case} = [[\%1 \neq 0,\ \frac{\partial^2}{\partial y^2}\,\mathrm{u}(x,y)]],\ \mathit{Constraint} = [$$

$$-(\frac{\partial}{\partial x}\,\mathrm{u}(x,y))\,(\frac{\partial}{\partial y}\,\mathrm{u}(x,y)) + (\frac{\partial}{\partial x}\,\mathrm{u}(x,y))^2 = 0,\ (\frac{\partial}{\partial y}\,\mathrm{u}(x,y))^2 + (\frac{\partial}{\partial y}\,\mathrm{u}(x,y)) - \mathrm{u}(x,y) = 0$$

$$],$$

$$\mathit{Pivots} = [\%1 \neq 0]$$

$$])$$

$$\%1 := 2\,(\frac{\partial}{\partial y}\,\mathrm{u}(x,\,y)) + 1$$

Example 7: **rifsimp** form of a nonlinear PDE system with cases

This problem arises from the determination of Lie symmetry groups for different values of **a**,**b** for the ODE:

```
> ODE := diff(y(x),x,x)-2*a^2*y(x)^3+2*a*b*x*y(x)-b;
```

$$ODE := (\frac{\partial^2}{\partial x^2}\,\mathrm{y}(x)) - 2\,a^2\,\mathrm{y}(x)^3 + 2\,a\,b\,x\,\mathrm{y}(x) - b$$

The Lie symmetry determining equation can be obtained through use of the `DEtools[gensys]` function as follows:

```
> sys := {gensys(ODE, [xi,eta](x,y))};
```

$$\begin{aligned}
sys := \{&-\eta(x,\,y)\,(6\,a^2\,y^2 - 2\,a\,b\,x) - 2\,(2\,a^2\,y^3 - 2\,a\,b\,x\,y + b)\,(\frac{\partial}{\partial x}\,\xi(x,\,y)) \\
&+ (2\,a^2\,y^3 - 2\,a\,b\,x\,y + b)\,(\frac{\partial}{\partial y}\,\eta(x,\,y)) + 2\,\xi(x,\,y)\,a\,b\,y + (\frac{\partial^2}{\partial x^2}\,\eta(x,\,y)), \\
&(\frac{\partial^2}{\partial y^2}\,\eta(x,\,y)) - 2\,(\frac{\partial^2}{\partial y\,\partial x}\,\xi(x,\,y)),\ \frac{\partial^2}{\partial y^2}\,\xi(x,\,y), \\
&-3\,(2\,a^2\,y^3 - 2\,a\,b\,x\,y + b)\,(\frac{\partial}{\partial y}\,\xi(x,\,y)) + 2\,(\frac{\partial^2}{\partial y\,\partial x}\,\eta(x,\,y)) - (\frac{\partial^2}{\partial x^2}\,\xi(x,\,y))\}
\end{aligned}$$

When viewed as a system in **xi** and **eta**, it is a linear system, but if we are also considering **a**,**b** to be unknowns, the system becomes effectively nonlinear. Now what we want to accomplish is to find the equations for the symmetry generators for all values of **a**,**b**. We can do this by allowing **rifsimp** to perform case splitting on the system using the **casesplit** option (see `rifsimp[cases]` for more detail).

```
> ans:=rifsimp(sys,casesplit);
```

$$ans := \mathrm{table}([1 = \mathrm{table}([\mathit{Solved} = [\eta(x,\,y) = 0,\, \xi(x,\,y) = 0],$$
$$\mathit{Case} = [[a \neq 0,\, \tfrac{\partial}{\partial y}\,\xi(x,\,y)],\, [b \neq 0,\, \eta(x,\,y)]],$$
$$\mathit{Pivots} = [a \neq 0,\, b \neq 0]]),$$
$$2 = \mathrm{table}([\mathit{Solved} = [\tfrac{\partial}{\partial x}\,\eta(x,\,y) = 0,\, \tfrac{\partial}{\partial x}\,\xi(x,\,y) = -\frac{\eta(x,\,y)}{y},\, \tfrac{\partial}{\partial y}\,\eta(x,\,y) = \frac{\eta(x,\,y)}{y},$$
$$\tfrac{\partial}{\partial y}\,\xi(x,\,y) = 0,\, b = 0],$$
$$\mathit{Case} = [[a \neq 0,\, \tfrac{\partial}{\partial y}\,\xi(x,\,y)],\, [b = 0,\, \eta(x,\,y)]],$$
$$\mathit{Pivots} = [a \neq 0]]),$$
$$3 = \mathrm{table}([\mathit{Solved} = [\tfrac{\partial^3}{\partial y^2\,\partial x}\,\eta(x,\,y) = 0,\, \tfrac{\partial^3}{\partial y^3}\,\eta(x,\,y) = 0,$$
$$\tfrac{\partial^2}{\partial x^2}\,\eta(x,\,y) = b\,(2\,(\tfrac{\partial}{\partial x}\,\xi(x,\,y)) - (\tfrac{\partial}{\partial y}\,\eta(x,\,y))),$$
$$\tfrac{\partial^2}{\partial x^2}\,\xi(x,\,y) = -3\,(\tfrac{\partial}{\partial y}\,\xi(x,\,y))\,b + 2\,(\tfrac{\partial^2}{\partial y\,\partial x}\,\eta(x,\,y)),\, \tfrac{\partial^2}{\partial y\,\partial x}\,\xi(x,\,y) = \frac{1}{2}\,(\tfrac{\partial^2}{\partial y^2}\,\eta(x,\,y)),$$
$$\tfrac{\partial^2}{\partial y^2}\,\xi(x,\,y) = 0,\, a = 0],$$
$$\mathit{Case} = [[a = 0,\, \tfrac{\partial}{\partial y}\,\xi(x,\,y)]]]),$$
$$\mathit{casecount} = 3])$$

The structure of the output of **rifsimp** has changed form into a recursive table for multiple cases. The returned table has a **casecount** entry, and individual **rifsimp** answers for each case indexed by the numbers **1,2,3**. For the three cases, the form of the **Solved** equations for **xi**,**eta** is different. This indicates that for different values of **a**,**b** the structure of the Lie symmetry group is different, and provides the systems that must be satisfied for each group. This case splitting also covers all possible values of **a**,**b**, so it fully describes the dependence of the determining system on these values, and no other systems for other values of **a**,**b** exist.

The structure of the case tree can be observed in a plot using `caseplot`.

```
> caseplot(ans);
```

A.4 rifsimp[output] - description of algorithm output

Description The output of the **rifsimp** command is a table. A number of possible table formats (or entries) may or may not be present as they depend on specific options. The format changes from a table containing the simplified system to a nested table when **casesplit** is activated.

Single Case

```
Solved         This entry gives all equations that have been solved in
               terms of their leading indeterminate.

Pivots         This entry gives all inequations (equations of the form
               expression <> 0) that hold for this case. These may be part
               of the input system, or decisions made by the program during
               the calculation (see rifsimp[cases] for information on
               pivots introduced by the algorithm).

Case           This entry describes what assumptions were made to arrive
               at this simplified system (see rifsimp[cases].)

Constraint     This table contains all equations that are nonlinear in
               their leading indeterminate. The equations in this list
               form a Groebner basis and can be viewed as purely algebraic
               because any differential consequences that result from these
               equations are already taken care of through spawning
               (see rifsimp[nonlinear]).
               If the initial option has been specified, then these
               equations are isolated for the highest power of their
               leading derivatives, otherwise they are in the form expr=0.

DiffConstraint This table entry contains all equations that are nonlinear
               in their leading indeterminate, but either are not in
               Groebner basis form or have differential consequences that
```

	are not accounted for (see spoly and spawn in rifsimp[nonlinear]). Whenever equations appear in this entry, the system is in incomplete form and must be examined with care.
UnSolve	This table entry contains all equations that rifsimp did not attempt to solve (see unsolved in rifsimp[adv_options]). Whenever equations appear in this entry, the system is in incomplete form and must be examined with care.
UnClass	This table entry contains all equations that have not yet been examined (i.e. Unclassified). This entry is only present when looking at partial calculations using rifread, or when a computation is halted by mindim.
status	If this entry is present, then the output system is missing due to either a restriction or an error. The message in this entry indicates what the restriction or error is.
dimension	This entry is only present when the mindim option is used (see rifsimp[cases]), or for maxdimsystems. For the case where a single constraint is in effect (such as through use of the option mindim=8), the right-hand side is a single number (the dimension for the case). For multiple constraints, it is a list of dimension counts, one for each constraint in the mindim specification.

Here are examples of status messages:

"system is inconsistent"	No solution exists for this system
"object too large"	Expression swell has exceed Maple's ability to calculate

`"time expired"`	`Input time limit has been exceeded (see ctl, stl and itl in rifsimp[options])`
`"free count fell below mindim"`	`Free parameters have fallen below the minimum (see mindim in rifsimp[adv_options])`

Of the above, only the "object too large" message actually indicates an error.

To summarize, if the input system is fully linear in all indeterminates (including unknown constants), then only the **Solved** entry will be present. If the system is (and remains) linear in its leading indeterminates throughout the calculation, but has indeterminate expressions in the leading coefficients, then **Solved**, **Pivots**, and **Case** will be present. If equations that are nonlinear in their leading indeterminates result during a calculation, then **Constraint** will also be present. If the **status** entry is present, then not all information for the case will be given.

If **mindim** is used, then the **dimension** entry will be set to the dimension of the linear part of the system for the case when **status** is not set, and an upper bound for the dimension of the case if the count fell below the minimum dimension requirement.

Multiple Cases

For multiple cases (using the **casesplit** option), numbered entries appear in the output table, each of which is itself a table of the form described above.

For example, if a calculation resulted in 10 cases, the output table would have entries '1=...',..., '10=...', where each of these entries is itself a table that contains **Solved**, **Pivots**, **Case**, or other entries from the single case description.

In addition to the numbered tables is the entry **casecount**, which gives a count of the number of cases explored. Cases that rejected for reasons other than inconsistency will have the **Case** entry assigned, in addition to the **status** entry. Inconsistent cases, for multiple case solutions, are removed automatically.

So what is the difference between **Pivots** and **Case**?

The **Pivots** entry contains the inequations for the given case in simplified form with respect to the output system. The **Case** entry is a list with elements of the form [**assumption,leading derivative**] or [**assumption,leading derivative,"false split"**]. It describes the actual decision made for the case split in unsimplified form (i.e. as it was encountered in the algorithm). The **assumption** will be of the form **expr<>0** or **expr=0**, where **expr** depends on the dependent variables, derivatives and/or constants of the problem. The **leading derivative** is the indeterminate the algorithm was isolating that required the **assumption**. If the third **"false split"** entry is present, then it was later discovered that one branch of the split is entirely inconsistent, so the actual splitting was a **false split**ting, as the displayed assumption is always true with respect to the rest of the system. For example, if the algorithm were to split on an equation of the form **a*diff(f(y),y)+f(y)=0**, the **Case** entries that correspond to this case split are [**a<>0,diff(f(y),y)**], and [**a=0,diff(f(y),y)**]. If it was found later in the algorithm that **a=0** leads to a contradiction, then the **Case** entry would be given by [**a<>0,diff(f(y),y),"false split"**].

Note that when `faclimit` or `factoring` are used (of which **factoring** is turned on by default), it is possible to introduce a splitting that does not isolate a specific derivative. When this occurs, the case entry will be of the form [**assumption,"faclimit"**] or [**assumption,"factor"**].

Occasionally both the **Case** and **Pivots** entries contain the same information, but it should be understood that they represent different things.

Important

As discussed above, some options have the effect of preventing **rifsimp** from fully simplifying the system. Whenever **DiffConstraint** or **UnSolve** entries are present in the output, some parts of the algorithm have been disabled by options, and the resulting cases must be manually examined for consistency and completeness.

Examples

```
> with(DEtools):
```

As a first example, we take the overdetermined system of two equations in one dependent variable **f(x)**, and two constants **a** and **b**.

```
> sys1:=[a*diff(f(x),x,x)-f(x),b*diff(f(x),x)-f(x)];
```

$$sys1 := [a\,(\frac{\partial^2}{\partial x^2}\,\mathrm{f}(x)) - \mathrm{f}(x),\; b\,(\frac{\partial}{\partial x}\,\mathrm{f}(x)) - \mathrm{f}(x)]$$

Call **rifsimp** for a single case only (the default).

```
> ans1:=rifsimp(sys1);
```

$$ans1 := \mathrm{table}([\mathit{Case} = [[a \neq 0, \frac{\partial^2}{\partial x^2} \mathrm{f}(x)], [b \neq 0, \frac{\partial}{\partial x} \mathrm{f}(x)], [b^2 - a \neq 0, \mathrm{f}(x)]],$$
$$\mathit{Solved} = [\mathrm{f}(x) = 0],$$
$$\mathit{Pivots} = [a \neq 0, b \neq 0, b^2 - a \neq 0]$$
$$])$$

We see that under the given assumptions for the form of **a** and **b** (from **Pivots**), the only solution is given as **f(x)=0** (from **Solved**). Now, run the system in multiple case mode using **casesplit**.

```
> ans1m:=rifsimp(sys1,casesplit);
```

$$ans1m := \mathrm{table}([1{=}\mathrm{table}([\mathit{Case} = [[a \neq 0, \frac{\partial^2}{\partial x^2}\mathrm{f}(x)], [b \neq 0, \frac{\partial}{\partial x}\mathrm{f}(x)], [b^2 - a \neq 0, \mathrm{f}(x)]],$$
$$\mathit{Solved} = [\mathrm{f}(x) = 0],$$
$$\mathit{Pivots} = [a \neq 0, b \neq 0, b^2 - a \neq 0]$$
$$]), 2 = \mathrm{table}([\mathit{Case} = [[a \neq 0, \frac{\partial^2}{\partial x^2} \mathrm{f}(x)], [b \neq 0, \frac{\partial}{\partial x} \mathrm{f}(x)], [b^2 - a = 0, \mathrm{f}(x)]],$$
$$\mathit{Solved} = [\frac{\partial}{\partial x} \mathrm{f}(x) = \frac{\mathrm{f}(x)}{b}, a = b^2],$$
$$\mathit{Pivots} = [b \neq 0]$$
$$]), 3 = \mathrm{table}([\mathit{Case} = [[a \neq 0, \frac{\partial^2}{\partial x^2} \mathrm{f}(x)], [b = 0, \frac{\partial}{\partial x} \mathrm{f}(x)]], \mathit{Solved} = [\mathrm{f}(x) = 0, b = 0],$$
$$\mathit{Pivots} = [a \neq 0]$$
$$]), 4 = \mathrm{table}([\mathit{Case} = [[a = 0, \frac{\partial^2}{\partial x^2} \mathrm{f}(x)]], \mathit{Solved} = [\mathrm{f}(x) = 0, a = 0]]),$$
$$\mathit{casecount} = 4$$
$$])$$

We see that we have four cases:

```
> ans1m[casecount];
```

$$4$$

All cases except 2 have **f(x)=0**.

Looking at case 2 in detail, we see that under the constraint **a = b^2** (from **Solved**) and **b <> 0** from **Pivots**, the solution to the system will be given by the remaining ODE in **f(x)** (in **Solved**). Note here that the constraint on the constants **a** and **b**, together with the assumption **b <> 0**, imply that **a <> 0**, so this constraint is not present in the **Pivots** entry due to simplification. It is still present in the **Case** entry because **Case** describes the decisions made in the algorithm, not their simplified result. Also, case 4 has no **Pivots** entry. This is because no assumptions of the form **expression <> 0** were used for this case.

One could look at the **caseplot** with the command:

```
> caseplot(ans1m);
```

As a final demonstration involving this system, suppose that we are only interested in nontrivial cases where **f(x)** is not identically zero. We can simply include this assumption in the input system, and **rifsimp** will take it into account.

```
> ans1a:=rifsimp([op(sys1),f(x)<>0],casesplit);
```

$$\mathit{ans1a} := \mathrm{table}([\mathit{Case} = [[a \neq 0,\ \frac{\partial^2}{\partial x^2}\,\mathrm{f}(x),\ \text{"false split"}],\ [b \neq 0,\ \frac{\partial}{\partial x}\,\mathrm{f}(x),\ \text{"false split"}]],$$
$$\mathit{Solved} = [\frac{\partial}{\partial x}\,\mathrm{f}(x) = \frac{\mathrm{f}(x)}{b},\ a = b^2],$$
$$\mathit{Pivots} = [\mathrm{f}(x) \neq 0]$$
$$])$$

We see that the answer is returned in a single case with two **false split** Case entries. This means the computation discovered that the **a=0** and **b=0** cases lead to contradictions, so the entries in the Case list are labelled as **false split**s, and the alternatives for the binary case splittings (cases with **a=0** or **b=0**) are not present.

For the next example, we have a simple inconsistent system:

```
> sys2:=[diff(u(x),x,x)+diff(u(x),x)^2-1,diff(u(x),x,x)+1];
```

$$\mathit{sys2} := [(\frac{\partial^2}{\partial x^2}\,\mathrm{u}(x)) + (\frac{\partial}{\partial x}\,\mathrm{u}(x))^2 - 1,\ (\frac{\partial^2}{\partial x^2}\,\mathrm{u}(x)) + 1]$$

```
> rifsimp(sys2);
```

$$\mathrm{table}([\mathit{status} = \text{"system is inconsistent"}])$$

So there is no solution **u(x)** to the above system of equations.

The next example demonstrates the **UnSolve** list, while also warning about leaving indeterminates in unsolved form.

```
> sys3:=[diff(f(x),x)*(diff(g(x),x)-g(x))+f(x)^2,diff(g(x),x)-g(x)];
```

$$\mathit{sys3} := [(\frac{\partial}{\partial x}\,\mathrm{f}(x))\,((\frac{\partial}{\partial x}\,\mathrm{g}(x)) - \mathrm{g}(x)) + \mathrm{f}(x)^2,\ (\frac{\partial}{\partial x}\,\mathrm{g}(x)) - \mathrm{g}(x)]$$

So we run **rifsimp**, but only solve for **f(x)**, leaving **g(x)** in unsolved form. Unfortunately, the resulting system is inconsistent, but this is not recognized because equations containing only **g(x)** are left unsolved. As discussed earlier in the page, these equations come out in the **UnSolve** list.

```
> rifsimp(sys3,[f],unsolved);
```

$$\mathrm{table}([\mathit{Case} = [[\%1 \neq 0, \frac{\partial}{\partial x}\,\mathrm{f}(x)]], \mathit{Solved} = [\frac{\partial}{\partial x}\,\mathrm{f}(x) = -\frac{\mathrm{f}(x)^2}{\%1}], \mathit{Pivots} = [\%1 \neq 0],$$
$$\mathit{UnSolve} = [0 = \%1]$$
$$])$$
$$\%1 := (\frac{\partial}{\partial x}\,\mathrm{g}(x)) - \mathrm{g}(x)$$

When equations are present in the **UnSolve** list, they must be manually examined.

Here is a nonlinear example.

```
> sys4:=[diff(f(x),x,x)+f(x),diff(f(x),x)^2-f(x)^2];
```

$$\mathit{sys4} := [(\frac{\partial^2}{\partial x^2}\,\mathrm{f}(x)) + \mathrm{f}(x),\ (\frac{\partial}{\partial x}\,\mathrm{f}(x))^2 - \mathrm{f}(x)^2]$$

By default **rifsimp** spawns the nonlinear equation to obtain a leading linear equation, and performs any required simplifications. The end result gives the following output:

```
> rifsimp(sys4,casesplit);
```

$$\mathrm{table}([\mathit{Case} = [[\mathrm{f}(x) = 0, \frac{\partial}{\partial x}\,\mathrm{f}(x), \text{“false split”}]], \mathit{Solved} = [\mathrm{f}(x) = 0]])$$

We have only one consistent case. Attempting to perform this calculation with the **spawn=false** option gives the following:

```
> rifsimp(sys4,casesplit,spawn=false);
```

$$\mathrm{table}([\mathit{Solved} = [\frac{\partial^2}{\partial x^2}\,\mathrm{f}(x) = -\mathrm{f}(x)], \mathit{DiffConstraint} = [0 = (\frac{\partial}{\partial x}\,\mathrm{f}(x))^2 - \mathrm{f}(x)^2]])$$

So it is clear that by disabling spawning, the system is not in fully simplified form (as indicated by the presence of the **DiffConstraint** entry), and we do not obtain full information about the system.

A.5 rifsimp[options] - common options

Description

The **rifsimp** options are given after the input system, and optionally the **vars**. The following are the most commonly used options:

vars

This is the optional second argument to **rifsimp**. It indicates which indeterminates are to be solved for. By default **rifsimp** attempts to solve for all dependent variables with the same differential order and differentiations with equal precedence, breaking ties alphabetically. **rifsimp** solves for constants when only constants remain in an equation.

The selection of the solving indeterminate (called the leading indeterminate) of an equation is performed based on the *ranking* imposed on the system. This argument can be used in nested list form to modify the indeterminates to solve for. For example, if **f(x)**, **g(x)**, and **h(x)** were the dependent variables of a problem, and we wanted to isolate all of these with equal precedence, we could specify **vars** as [**f,g,h**]. If instead we wanted to eliminate **f(x)** from as much of the system as possible, we could specify [[**f**],[**g,h**]] instead, which tells **rifsimp** to solve for **f(x)** and its derivatives with higher precedence than **g(x)**, **h(x)** and any of their derivatives, regardless of the differential order of **f(x)**. Under this nested list ranking, and equation of the form **g"' f-g" h"=0** would be solved for **f** giving **f=g" h"/g"'**. See `rifsimp[ranking]` and `checkrank` for more detail.

indep=[ind1,ind2,...]

This option specifies the names of the unknowns to be treated as independent variables. By default, only those unknowns given in the dependency list of all unknown functions in the input system are considered to be independent variables. All other unknowns are considered constants. Treating an independent variable as though it were a constant will result in an incomplete answer, so you must use this option when required (see `examples` below). Please note that **rifsimp** always views unknowns present in a dependency list of an unknown function as independent variables, even when not specified by this option.

The order in which the independent variables are specified in this option affects the selection of the leading indeterminate (the indeterminate to be solved for) in an equation. See `rifsimp[ranking]` and `checkrank` for details.

arbitrary=[v1,v2,...]

This option specifies a list of parameters or functions that should be treated as arbitrary. Any special cases where these parameters or functions take on specific values are rejected as inconsistent cases. **Note:** This means that no relations purely in the arbitrary parameters or functions can be present in the input system (with the sole exception of dependency related equations such as **diff(f(x,y),y)=0**). If a constraint is required for one of these unknowns, then it is no longer truly arbitrary, but rather restricted, so at least one of the unknowns in the constraint must be a solving variable.

This option generalizes the concept of a `field of constants` as used in the `diffalg` package to functions of the independent variables. It is most useful when only generic results are needed, but it may be the case that the result is invalid for specific values of these parameters (for example, if **a** was a parameter, and it occurred as a denominator, then the solution is only valid for **a<>0**).

casesplit

This option indicates that the program should explore all cases. A case split is most often introduced when, among the remaining unsolved equations, there is an equation that is linear in its highest ranked indeterminate (called the *leading indeterminate*, see `rifsimp[ranking]` or `checkrank`), and the coefficient of that indeterminate (called the *pivot*) may or may not be zero (see `rifsimp[cases]`).

Isolation of the leading indeterminate in this equation introduces a case split – namely the two possibilities, **pivot** $<>$ **0** and **pivot** $=$ **0**. The more generic case, **pivot** $<>$ **0**, is explored first. Once that case (possibly containing further case splits) is completed, the case with **pivot** $=$ **0** is explored. This results in a case tree, which can be represented graphically (see `caseplot`).

In addition to producing multiple cases, this option changes the output of **rifsimp**. For more detail, see `rifsimp[output]`.

gensol

This option indicates that the program should explore all cases that have the possibility of leading to the most general solution of the problem. Occasionally it is possible for **rifsimp** to compute only the case corresponding to the general solution of the ODE/PDE system.

When this option is given, and this occurs, **rifsimp** will return the single case corresponding to the general solution. When it is not possible for **rifsimp** to detect where in the case tree the general solution is, then multiple cases are returned, one or more of which correspond to the general solution of the ODE/PDE system, and others may correspond to singular solutions. For some particularly difficult problems, it is possible that the entire case tree is returned. **Note:** this option cannot be used with the **casesplit**,**casecount**, **pivsub**, and **mindim** options.

ctl='time'

This option imposes a limit on the amount of time (in CPU seconds) that would be required to compute each case from scratch (i.e. using the extended option **casesplit=[...]** described in `rifsimp[adv_options]`). It is clear that this option may count computation time more than once, as the time prior to a case split is counted for each case after the split. This bounds the total time consumed to be no greater than the imposed limit times the number of cases obtained (and most often, significantly less as per the prior comment). For highly nonlinear systems, it may be impossible to obtain all cases in a reasonable amount of time. Use of this option allows the simpler cases to be computed, and the more expensive cases to be deferred. **Note:** there are two other time limit related options, **stl** and **itl**, and these are discussed in the `rifsimp[adv_options]` page.

clean=[cleaning criteria]

This option is a fine-tuning control over the system(s) output by rifsimp. There are three types of *cleaning* that the algorithm can perform:

Pivots: These are the inequations resulting from case splitting in the system (or present on input). As an example, consider the pivot **diff(f(x),x)<>0**. This pivot clearly implies that **f(x)<>0**, so on completion, the pivot **f(x)<>0** is considered to be redundant. There are three options for specification of pivot cleaning:

```
nopiv        perform no cleaning, returning all pivots obtained
             in the course of the computation.
piv          perform cleaning of obvious redundant pivots (i.e.
             those that can be detected by inspection of the
             pivots list on output.
fullpiv      perform thorough cleaning of pivots, including
```

```
              removal of pivots that are redundant only for
              solutions of the output case/system.
```

Note that the **fullpiv** criteria may remove pivots that would require a great deal of computation to recover. For example, consider the simple case described earlier for **diff(f(x),x)**<> 0. If later in the algorithm, **diff(f(x),x)** was reduced, and looked nothing like the original pivot, on completion **f(x)<>0**, which is a consequence of the original pivot, would be removed.

One-Term Equations: These are simply equations of the form **derivative=0**. These are retained in the course of the computation (for efficiency) even when they reduce modulo one of the other equations of the system. These equations can be quite helpful in attempting to integrate of the results of **rifsimp**. The **oneterm** criteria indicates that the redundant one-term equations should be removed, while **nooneterm** indicates that they should be retained.

Spawn: The spawning process is only used with nonlinear equations in **rifsimp**, and results in equations that are derivatives of any nonlinear equations in the output. These spawned equations are necessary for the proper running of `initialdata` and `rtaylor`. The following options specify spawned equation cleaning:

```
nospawn      do not remove any redundant spawned equations
spawn        remove any redundant spawned equations that are not
             in solved form.
fullspawn    remove all redundant spawned equations.
```

By default, the **clean** settings are [**piv,oneterm,nospawn**].

fullclean

This is a shortcut specification for the **clean** options for **rifsimp** (see above). This corresponds to the specification **clean=**[**fullpiv,oneterm,fullspawn**].

Examples

```
> with(DEtools):
```

This example highlights the difference between treating the unknown **y** as a constant and treating it as an independent variable; by default, the code assumes that **y** is a constant.

```
> sys1:=[y*f(x)+g(x)];
```

$$sys1 := [y\,\mathrm{f}(x) + \mathrm{g}(x)]$$

```
> rifsimp(sys1);
```

$$\mathrm{table}([\mathit{Solved} = [\mathrm{f}(x) = -\frac{\mathrm{g}(x)}{y}],\ \mathit{Pivots} = [y \neq 0],\ \mathit{Case} = [[y \neq 0,\ \mathrm{f}(x)]]])$$

Specification of **y** as an independent variable gives the following.

```
> rifsimp(sys1,indep=[x,y]);
```

$$\mathrm{table}([\mathit{Solved} = [\mathrm{f}(x) = 0,\ \mathrm{g}(x) = 0]])$$

This next example demonstrates the use of the **casesplit** option. We consider the Lie-symmetry determining system for the following ODE:

```
> ODE:=diff(y(x),x,x)+(2*y(x)+f(x))*diff(y(x),x)+diff(f(x),x)*y(x);
```

$$\mathit{ODE} := (\frac{\partial^2}{\partial x^2}\,\mathrm{y}(x)) + (2\,\mathrm{y}(x) + \mathrm{f}(x))\,(\frac{\partial}{\partial x}\,\mathrm{y}(x)) + (\frac{\partial}{\partial x}\,\mathrm{f}(x))\,\mathrm{y}(x)$$

The Lie symmetries are given as the solution of the following system of determining PDEs (as generated using `DEtools[odepde]`):

```
> sys:=[coeffs(expand(odepde(ODE,[xi(x,y),eta(x,y)], y(x))),_y1)];
```

$$\begin{aligned}
&sys := [2\,(\tfrac{\partial}{\partial x}\,\xi(x,\,y))\,(\tfrac{\partial}{\partial x}\,\mathrm{f}(x))\,y + \eta(x,\,y)\,(\tfrac{\partial}{\partial x}\,\mathrm{f}(x)) + 2\,(\tfrac{\partial}{\partial x}\,\eta(x,\,y))\,y + (\tfrac{\partial}{\partial x}\,\eta(x,\,y))\,\mathrm{f}(x)\\
&\quad + (\tfrac{\partial^2}{\partial x^2}\,\eta(x,\,y)) - (\tfrac{\partial}{\partial y}\,\eta(x,\,y))\,(\tfrac{\partial}{\partial x}\,\mathrm{f}(x))\,y + \xi(x,\,y)\,(\tfrac{\partial^2}{\partial x^2}\,\mathrm{f}(x))\,y,\ -(\tfrac{\partial^2}{\partial y^2}\,\xi(x,\,y)),\\
&(\tfrac{\partial}{\partial x}\,\xi(x,\,y))\,\mathrm{f}(x) - (\tfrac{\partial^2}{\partial x^2}\,\xi(x,\,y)) + \xi(x,\,y)\,(\tfrac{\partial}{\partial x}\,\mathrm{f}(x)) + 2\,(\tfrac{\partial^2}{\partial y\,\partial x}\,\eta(x,\,y)) + 2\,\eta(x,\,y)\\
&\quad + 3\,(\tfrac{\partial}{\partial y}\,\xi(x,\,y))\,(\tfrac{\partial}{\partial x}\,\mathrm{f}(x))\,y + 2\,(\tfrac{\partial}{\partial x}\,\xi(x,\,y))\,y,\\
&-2\,(\tfrac{\partial^2}{\partial y\,\partial x}\,\xi(x,\,y)) + 2\,(\tfrac{\partial}{\partial y}\,\xi(x,\,y))\,\mathrm{f}(x) + (\tfrac{\partial^2}{\partial y^2}\,\eta(x,\,y)) + 4\,(\tfrac{\partial}{\partial y}\,\xi(x,\,y))\,y]
\end{aligned}$$

So running this system with **rifsimp**:

```
> ans:=rifsimp(sys,[xi,eta]):
> ans['Solved'];
```

$$[\xi(x,\,y) = 0,\ \eta(x,\,y) = 0]$$

And we see that the given ODE has no point symmetries for general **f(x)**.

We may want to know if there are particular forms of **f(x)** for which point symmetries exist (this is called a classification problem). Running **rifsimp** with **casesplit**:

```
> ans:=rifsimp(sys,[xi,eta],casesplit):
> ans['casecount'];
```

4

so we see there are cases.

We could use the **caseplot** command to give a pictorial view of the case tree with the following command.

```
> caseplot(ans);
```

Looking at case 3 in detail:

```
> copy(ans[3]);
```

$$\mathrm{table}([\mathit{Solved} = \left[\frac{\partial}{\partial x}\xi(x, y) = \frac{-\xi(x, y)\,(\frac{\partial}{\partial x}\mathrm{f}(x)) - 2\,\eta(x, y)}{2\,y + \mathrm{f}(x)},\right.$$

$$\frac{\partial}{\partial x}\eta(x, y) = \frac{(\frac{\partial}{\partial x}\mathrm{f}(x))\,(2\,\xi(x, y)\,(\frac{\partial}{\partial x}\mathrm{f}(x)) + 4\,\eta(x, y) + \mathrm{f}(x)^2\,\xi(x, y) + 2\,\xi(x, y)\,\mathrm{f}(x)\,y)}{2\,\mathrm{f}(x) + 4\,y}$$

$$\left., \frac{\partial}{\partial y}\xi(x, y) = 0,\ \frac{\partial}{\partial y}\eta(x, y) = \frac{\xi(x, y)\,(\frac{\partial}{\partial x}\mathrm{f}(x)) + 2\,\eta(x, y)}{2\,y + \mathrm{f}(x)},\ \frac{\partial^2}{\partial x^2}\mathrm{f}(x) = -\mathrm{f}(x)\,(\frac{\partial}{\partial x}\mathrm{f}(x))\right],$$

$$\mathit{Pivots} = [\frac{\partial}{\partial x}\mathrm{f}(x) \neq 0],$$

$$\mathit{Case} = [[-3\,(\frac{\partial}{\partial x}\mathrm{f}(x))\,(\frac{\partial^2}{\partial x^2}\mathrm{f}(x)) + 2\,(\frac{\partial^3}{\partial x^3}\mathrm{f}(x))\,y + (\frac{\partial^3}{\partial x^3}\mathrm{f}(x))\,\mathrm{f}(x) + 2\,(\frac{\partial}{\partial x}\mathrm{f}(x))^2\,y$$

$$-\,2\,(\frac{\partial}{\partial x}\mathrm{f}(x))^2\,\mathrm{f}(x) + \mathrm{f}(x)^2\,(\frac{\partial^2}{\partial x^2}\mathrm{f}(x)) + 2\,\mathrm{f}(x)\,(\frac{\partial^2}{\partial x^2}\mathrm{f}(x))\,y = 0,\ \xi(x, y)],$$

$$[\frac{\partial}{\partial x}\mathrm{f}(x) \neq 0,\ \frac{\partial^2}{\partial x^2}\mathrm{f}(x)]]$$

$$])$$

so we see we have a 2 parameter Lie group for the specific form of **f(x)** given by:

```
> dsolve(diff(f(x),x,x) = -f(x)*diff(f(x),x));
```

$$\mathrm{f}(x) = \frac{\tanh(\frac{1}{2}\,\frac{(x + \_C2)\,\sqrt{2}}{\_C1})\,\sqrt{2}}{\_C1}$$

A.6 rifsimp[adv_options] - advanced options

Description

The **rifsimp** options are given after the input system, and optionally the **vars**.

The following are the advanced options:

store[=name]

Store the system after each iteration in a file called **name.m** (default is RifStorage.m). Note: when multiple cases are being examined, only the latest iteration of the current case is available. Iterations can be observed by setting **infolevel[rifsimp]** to 2 or greater. This file can be read using the `rifread` command.

storeall[=name]

Store the system iterations separately in a file called **name_cs_i.m**, where **cs** is the current case number, and **i** is the current iteration number (default is RifStorage_cs_i.m). This file can be read using the `rifread` command.

ezcriteria=[quantity, size ratio]

The **ezcriteria** option allows control over how quickly equations are chosen for solving. **quantity** is a factor that adjusts the number of equations selected, while the **size ratio** tells **rifsimp** how large a difference in size can exist between the smallest and largest equations chosen in a single iteration.

The number of equations selected in a single iteration is based upon the number that have already been solved. So if we have no equations solved for their leading derivative, then the algorithm will select at most 10 new equations, whereas if there are 200 equations already in solved form, only one new equation will be selected. The option works this way to control how much work is done in each iteration, since adding one equation when there are no equations in solved form requires no checking of compatibility (integrability) conditions, while adding one equation to the 100 already in solved form may require checking of 100 compatibility conditions.

The size ratio enforces a maximum difference in size between the smallest and largest equations chosen in an iteration. If the size ratio was set to 2 (the default), and the three equations remaining to be solved have sizes of 100, 190, 400, 500, then even if the number to be selected based on the quantity was more than 2, only the first two equations will be selected, as equations 3 and 4 are more than double the size of equation 1.

Primary uses of this option are for the ultra-conservative setting of [1,0.9], and the setting of [infinity,infinity] at the other extreme.

tosolve=n

This option indicates that the first **n** equations or expressions in the input system are to be placed into solved form immediately. If an inequation is present in the first **n** equations, it is not counted as one of the first **n** equations. This option is only available when the system has been specified in list form.

unsolved

This tells Maple not to solve for any dependent variables or constants not explicitly mentioned in **vars**. It should only be used used when **rifsimp** is being used as a preprocessor. It is inadvisable to use this option with nonlinear equations, as it can result in infinite loops, may prevent inconsistent cases from being recognized, and does not result in a disjoint set of cases. This option cannot be used with the **arbitrary** option (see `rifsimp[options]`).

stl='time'

This option imposes a limit on the amount of time (in CPU seconds) spent on the calculation since the last case split occurred. Use of this option allows the simpler calculations to be performed, and the more expensive calculations to be deferred. Note, this option is quite different from **ctl**, as it does not come into effect until the first splitting starts, and only bounds time between splittings. This has the effect of screening out cases that are not making progress quickly enough.

itl='time'

This option imposes a limit on the amount of time (in CPU seconds) spent on any single iteration. An iteration consists of selecting one or more equations to isolate, adding them to the solved list, simplifying the existing system with respect to the new simplified equations, and then considering all consequences that the new equations have with respect to the already-solved equations in the system. Iterations in a calculation can be viewed by setting the **rifsimp** infolevel to 2 or greater.

ranking=[[ind weights 1, dep weights 1], [ind weights 2, dep weights 2], ...]

This gives the linear ordering used to determine the relative weights of the dependent variable derivatives. See `rifsimp[ranking]` for more information.

casesplit=[‘<>‘,‘=‘,‘< >‘]

This option turns on case splitting and also gives the start of the tree which will be examined. See `rifsimp[cases]` for further details.

mindim=...

This option is used to limit the cases being examined based on the solution space dimension of the linear part of the system. The simple specification is given as **mindim='n'**. If **vars** is specified as a simple list, this gives the minimum of **n** free 0-dimensional parameters in the **vars** dependent variables. If **vars** contains lists, it is specified as part of a ranking (see `rifsimp[ranking]`), and only the first element of **vars** is used to compute the dimension. Once the condition fails, the current case is discarded.

NOTE: **mindim** only looks at the linear part of the current system, so if nonlinear constraints are present, the actual dimension may be lower than the dimension reported.

It is possible to give more detailed and/or multiple specifications for a calculation. Please see `rifsimp[cases]` for the more advanced usage, and additional information.

pivselect=choice

This option gives some control over how **rifsimp** performs case splitting. More information can be found in `rifsimp[cases]`.

nopiv=[var1, ...]

This option gives a list of all variables and constants which cannot be present in pivots. This is most useful for quasi-linear systems for which the linearity of the resulting system is of greatest importance. This may also have the effect of reducing the size of the case tree in these systems. The unsolved equations are then treated as fully nonlinear (see `rifsimp[nonlinear]`).

faclimit=n

Typically the pivot chosen is the coefficient of the leading indeterminate in the equation. In the event that the leading indeterminate is itself a factor of the equation, and this same leading indeterminate factor occurs in **n** or more equations, then it is chosen as the pivot rather than the coefficient. See `rifsimp[cases]` for more information.

This is most useful for Lie determining systems having many inconsistent cases, since many equations of this form typically occur during the calculation. The answer may be obtained without this option, but it has the beneficial effect of reducing the number of cases.

factoring=desc

This option is used to control introduction of pivots that are not necessarily the leading coefficient of an equation. Basically, if no leading linear equations remain in the unsolved equations, then depending on the value of this parameter, if the unsolved equations can be factored, a split will be performed on that factor. If **desc=none**, then this is disabled; if **desc=nolead**, then factors may only be chosen if they do not contain the leading derivative (this is the default); if **desc=all** then any factor can be split on. See `rifsimp[cases]` for more detail.

Note that use of **factoring=none** can result in non-termination for some nonlinear problems.

checkempty

This option tells **rifsimp** to try to eliminate any empty cases in the computed result(s). These can only occur when both pivots and nonlinear equations are present in the result, and is quite uncommon. This is not done by default (See `rifsimp[nonlinear]` for details).

grobonly

This option tells **rifsimp** to only use Groebner basis simplification for the leading nonlinear equations instead of the more efficient method described in `rifsimp[nonlinear]`.

initial

This option tells **rifsimp** to isolate the highest power of the leading derivative of each equation in the **Constraint** list, performing further case splitting on the coefficient if required. This causes the output case structure to more closely resemble that of `diffalg`, and changes the form of the equations in **Constraint**. This option is enabled by default unless **grobonly** is specified (see `rifsimp[output]`). For more information on this option, please see `rifsimp[cases]`.

grob_rank=[[1,deg,none],[1,ilex]]

This option controls the ranking for algebraic Groebner basis sub-computations. See `rifsimp[nonlinear]` for details.

spoly=[true,false]

This option indicates whether to generate S-polynomials during the Groebner basis algorithm (default is **true**). Setting **spoly=false** has some risk associated with it, and if nonlinear equations are present in the output, then it is possible that not all consequences of these equations have been obtained (hence they are placed in the DiffConstraint list rather than the Constraint list). If an orderly ranking is used (see `rifsimp[ranking]`, and **spawn** is set to **true**, then only algebraic simplifications remain, but these may still result in an inconsistent system when considered with the Pivot equations.

spawn=[true,false]

This option indicates whether to perform a differential spawn of nonlinear equations (default is **true**). Setting this to **false** makes the program ignore any differential consequences of the polynomially nonlinear equations. This is only useful if **rifsimp** is being used as a single step of a different calculation, since an incomplete answer will be obtained. See `rifsimp[nonlinear]`.

For a description and examples of the above options, please see the specific pages describing those options:

```
ranking                               See rifsimp,ranking
casesplit, mindim, pivselect, nopiv,  See rifsimp,cases
faclimit, pivsub, factoring, initial  "
grobonly, grob_rank, spoly, spawn,    See rifsimp,nonlinear}
checkempty                            "
```

A.7 rifsimp[ranking] - understanding rankings and related concepts

Description

What is a ranking?

A ranking is essentially an ordering of all indeterminates in a system. To introduce rankings, we must first introduce some related concepts.

Dependent variable

This term describes any unknown function that is present in the input system. As an example, consider a system involving **f(x,y)** and its derivatives, **g(x,y)** and its derivatives, and **exp(x)**. In a calculation, you may want to view **f(x,y)** as the solving variable, and **g(x,y)** as a parameter. Even in this case, these are both considered to be dependent variables. Because **exp(x)** is a known function, it is not considered to be a dependent variable.

Independent variable

For a problem containing **f(x,y)** and its derivatives, **g(x,y)** and its derivatives, and **exp(x)**, **x** and **y** are the independent variables.

Constants

Any unknown not having a dependency, and not occurring in a dependency, is considered to be a constant. This is true even if it appears in a known function. For example, in the equation **a*f(x,y)+sin(c)*g(x,y)**, both **a** and **c** are considered to be constants.

Note: The distinction between independent variables and constants is vital, since mistaking one for the other will not give a valid result for a system. For information on the specification of independent variables, please see `rifsimp[options]`.

Indeterminate

An indeterminate can be any constant, dependent variable, or derivative of a dependent variable. This does not include any known functions or independent variables. This is exactly the group of items that a ranking is defined for.

With these definitions, a more precise definition of ranking for a system is now possible:

Ranking

A ranking is a strict ordering of all indeterminates appearing in a system in the course of a calculation. Note that it is necessary to rank more than just the indeterminates appearing in the initial system, because higher derivatives may appear in the course of the algorithm.

Leading indeterminate

The leading indeterminate of an equation is defined as the indeterminate in that equation that is of the highest rank (maximal with respect to the ranking).

The concept of a leading indeterminate is important for understanding how the **rifsimp** algorithm works, because any equation in which the leading indeterminate appears linearly is solved with respect to that indeterminate.

Properties of a ranking

Rankings have a number of properties, some of which are required for the proper performance and termination of the algorithm, others of which may be helpful in tackling specific systems.

In any of the descriptions below, **v1** and **v2** are indeterminates that may depend on some of **x1, x2, x3, ...**

Preservation of ranking under differentiation

Given a ranking **v1** > **v2**, this ranking also holds after equal differentiation of both **v1** and **v2** with respect to any independent variable, for all **v1, v2** where **v1** and **v2** are indeterminates.

Note: You must restrict yourself to non-vanishing indeterminates (for example, for **h(x)** > **g(x,y)**, differentiation with respect to **y** makes **h(x)** vanish, so the rule does not apply).

This property is required for the proper running of **rifsimp**. Once an equation is solved for the leading indeterminate, any differentiation of that equation (assuming that the leading indeterminate does not vanish) is also in solved form with respect to its new leading indeterminate (the derivative of the prior leading indeterminate).

Positive ranking

Given a ranking >, it must be true that **diff(v1,x1)** > v1 for all indeterminates **v1** and all independent variables **x1**, as long as **diff(v1,x1)** is nonzero.

This property is required for the proper running of **rifsimp**, because it prevents any infinite chain of differential substitutions from occurring.

As an example, consider the solved form of the equation **u[t]-u[tt] = 0** under a non-positive ranking **u[t] = u[tt]**. Differential elimination of the derivative **u[xt]** with respect to this equation will give **u[xtt]**, then **u[xttt]**, then **u[xtttt]**, and so on. It will never terminate.

Total-degree ranking

Let **dord()** give the total differential order of an indeterminate with respect to all independent variables. Then for **v1** and **v2** derivatives of the same dependent variable, given a ranking $>$, then **dord(v1)** larger than **dord(v2)** implies **v1** $>$ **v2**.

For **rifsimp** to run correctly, a ranking does not have to be total degree. In some cases this does allow **rifsimp** to run faster, however. For those familiar with Groebner basis theory, a total degree ranking is similar to a total degree Groebner basis ordering, because calculations usually terminate more quickly than they would with a lexicographic ordering.

Orderly ranking

A ranking is said to be orderly if, for any indeterminate, no infinite sets of indeterminates that are lower in the ordering exist.

As an example of a ranking that is not orderly, we consider a ranking of **f(x)**, **g(x)**, and of all derivatives. If we choose to solve a system using **rifsimp** for **f(x)** in terms of **g(x)** (by specification of only **f(x)** in the solving variables), then this is not an orderly ranking, because **g(x)** and all of its derivatives are of lower rank than **f(x)** and any of its derivatives.

For **rifsimp** to run correctly, using the defaults for nonlinear equations, a ranking is not required to be orderly (see `rifsimp[nonlinear]`.)

The Default ranking

The **rifsimp** default ranking is an orderly, total-degree ranking that is both positive and preserved under differentiation. Note that if the solving variables are specified, we may no longer be using the default ranking (since specification of solving variables alters the ranking. See "Specification of a ranking" below.

On input, **rifsimp** assumes that all dependent variables present in the input system are solving variables, and that all constants are to be viewed as parameters. The set of dependent variables is then sorted alphabetically, along with the set of constants. In contrast, the set of independent variables is ordered based on order of appearance in the dependency lists of the dependent variables.

A description of the sorting method for independent variables can be found in the "Specification of a ranking" section. Note that this sort is used to break ties for derivatives of equal differential order (under some circumstances).

Under the above restriction, the default ranking is defined by the following algorithm, which returns **true** if **v1** is greater than **v2** with respect to the default ordering (and **false** if **v1** is less than **v2**).

```
rank-greater(v1,v2)
   #Criterion 1: Solving variabless
   If v1 is a solving variable, and v2 is not, then
      return(true)
   If v2 is a solving variable, and v1 is not, then
      return(false)
   #Criterion 2: Total Degreee
   If dord(v1) is larger than dord(v2), then
      return(true)
   If dord(v2) is larger than dord(v1), then
      return(false)
   #Criterion 3: Independent Variable Differential Order
   Loop i := each independent variable in sorted order
      If diff. order of v1 in i is larger than order of v2 in i, then
         return(true)
      If diff. order of v2 in i is larger than order of v1 in i, then
         return(false)
   end loop
   #Criterion 4: Dependent Variablee
   If dependent var for v1 occurs before v2, then
      return(true)
   If dependent var for v2 occurs before v1, then
```

```
    return(false)
end
```

The following examples are for a system containing **f(x,y,z)**, **g(x,y)**, and **h(x,z)**, and derivatives with the constants **a** and **b**. The system is recognized with the following (already sorted):

```
Dependent Variables     [f(x,y,z), g(x,y), h(x,z)]
Independent Variables   [x,y,z]
Constants               [a,b]
```

So, by default, **f, g,** and **h** are considered solving variables, and **a** and **b** parameters.

When the criteria are considered in order, any pair of distinct indeterminates are ranked by exactly one of them (as the ranking process then stops and returns the result). Of course to reach, say, criterion 3, criteria 1 and 2 must not differentiate the inputs. The following is a list of examples of each criterion in action:

```
Criterion 1   a < f, a < g, b < f,
              b < g, a < f[x], b < h[xxzz]
Criterion 2   f[xyy] < f[xxyy], f[x] < h[zz], g[xx] < h[xxz],
              h[z] < f[xx], f < h[x], h < f[x]
Criterion 3   f[xxy] < f[xxx], h[z] < f[y], g[y] < f[x],
              f[xz] < g[xy], f[yz] < f[xz], h[z] < h[x]
Criterion 4   g[xxy] < f[xxy], b < a, h[z] < f[z],
              g[xy] < f[xy], h[xz] < f[xz], g < f
```

Equivalence classes

Any step of the ranking process can be viewed as separating all possible indeterminates into a sorted chain of equivalence classes. When considering all criterion simultaneously, the size of each equivalence class must be exactly one. Sometimes viewing the ranking from the point of view of equivalence classes helps visualize how ranking is performed. As an example, we illustrate the equivalence classes for the prior example for each criterion considered independently of the other criteria:

Criterion 1: Rank solving variables higher than parameters

```
{a, b} < {f, g, h, f[x], f[y], f[z], g[x], g[y], h[x], h[z],
 f[xx], f[xy], `...`}}
```

Criterion 2: Rank by differential order

```
{f, g, h} < {f[x], f[y], f[z], g[x], g[y], h[x], h[z]}
 < {f[xx], f[xy], `...`} < ...}
```

Criterion 3: Rank by differential degree in each independent variable in turn

```
{f, g, h} < {f[x], g[x], h[x]} < {f[y], g[y]} < {f[z], h[z]}
 < {f[xx], g[xx], h[xx]} < {f[xy], g[xy]} < ...}
```

Criterion 4: Rank by dependent variable or constant name

```
{b} < {a} < {h, h[x], h[z], h[xx], h[xz], `...`}
 < {g, g[x], g[y], g[xx], g[xy], `...`}
 < {f, f[x], f[y], f[z], f[xx], `...`}}
```

So the process is equivalent to determining in which equivalence class each indeterminate falls, and checking if this criterion distinguishes the two input indeterminates.

Specification of a ranking

Three options can be used to control the ranking in **rifsimp**. Two of these perform simple modifications to the default ranking, while the third allows complete control of the ranking.

Specification of solve variables

As mentioned in the "Default ranking" section, specification of solving variables in a manner or order different from the default order changes the ranking. The solving variables can be specified in two ways:

1. Simple List

When the **vars** parameter is entered as a simple list, it affects the ranking in the following ways:

Criterion 1 of the default ranking is changed to add an additional class of indeterminates, which is specified by the solving variables. Specifically, any indeterminate mentioned in **vars** is ranked higher than any indeterminate not in **vars**. Any dependent variables not mentioned in **vars** are still ranked higher than any constants not mentioned in **vars**.

As an example, suppose an input system contained **f(x,y,z), g(x,y), h(x,z), a, b,** and **c**. If **vars** had been specified as [**f(x,y,z),b,h(x,y)**], then **f, h**, any **f** or **h** derivatives, and the constant **c** would be ranked higher than **g** and any **g** derivatives, which would be ranked higher than the constants **a** and **b**. Using equivalence classes, criterion 1 becomes

```
{a, b} < {g, g[x], g[y], g[xx], `...`}
 < {f, b, h, f[x], f[y], f[z], h[x], h[z], f[xx], f[xy], `...`}
```

where the new equivalence class is on the right.

Criterion 4 of the default ranking is changed to reflect the same order as the specified **vars**, so when criterion 4 is reached, **f** is ranked higher than **b**, which in turn is ranked higher than **g**, which in turn is ranked higher than **h**, and so on.

Using equivalence classes, we have the following:

```
{b} < {a} < {g, g[x], g[y], g[xx], g[xy], `...`}
< {h, h[x], h[z], h[xx], h[xz], `...`} < {c}
< {f, f[x], f[y], f[z], f[xx], `...`}
```

This is an unusual ranking (since it allows for **c** to be solved in terms of **h, g**, and derivatives of **g**), but it was chosen to highlight the flexibility of rankings.

2. Nested List

This is just a variation of the simple list specification that allows multiple equivalence classes to be specified. It is activated by the presence of a list in the **vars** input. When this specification is used, every entry in the **vars** list is interpreted as an equivalence class (even if it is not a list itself). This is best illustrated by an example. Use the same system as the prior example. If **vars** is specified as [**f(x,y,z)**,[**g(x,y),c**]], we notice that the second entry is a list, so the equivalence classes for criterion 1 become

```
{a, b} < {h, h[x], h[z], h[xx], `...`} <
 {g, c, g[x], g[y], g[xx], `...`} < {f, f[x], f[y], f[z], f[xx],
f[xy], `...`}
```

An equivalence class has been added for **f** and its derivatives, then one for **g** and **c**, and **g** derivatives, then the two that are created by default.

We can interpret this as "solve for **f** in terms of all other indeterminates; if the expression does not contain an **f**, then solve for **g** or **c** in terms of all other indeterminates, and so on".

Criterion 4 is changed to reflect the order in which the entries of **vars** appear in their equivalence classes:

```
{b} < {a} < {h, h[x], h[z], h[xx], h[xz], `...`} < {c}
< {g, g[x], g[y], g[xx], g[xy], `...`} < {f, f[x], f[y], f[z], f[xx],
`...`}
```

Option: indep=[...]

This option allows for specification of the independent variables, but modifies the ordering as well. Put simply, it specifies the order in which criterion 3 looks at the dependent variables.

First, recall how the default ordering works. To begin with, the set of independent variables is sorted alphabetically. Then, the set of independent variables is sorted based on their occurrence in the dependent variable lists, where the dependent variable lists are considered in the same order in which they are ranked. If the order of a dependency list disagrees with the order of another dependency list, only the one of higher rank one is used.

As an example, consider a system containing **f(x,y), g(y,x)**. In this case the independent variables are sorted in the order **[x,y]** if **f** is ranked higher than **g**, but in the order **[y,x]** if the reverse is true.

For a system containing **f(x,y), g(x,z)**, the independent variables are sorted **[x,y,z]**, because ties are broken alphabetically.

The specification of **indep=[...]** enforces the order specified in the list, so if the input contains **f(x,y,z)** and **h(x,y,z)** and we specify **[z,x,y]**, then the independent variables are ordered as specified.

These two ways of controlling the ordering of a system are sufficient for most purposes, but you can also fully specify the exact ordering to be used for a system.

Advanced specification of a ranking

This method requires a bit of description first:

Say that we have a system with **n** independent variables (we call these **x1,...,xn**), and **m** dependent variables (we call these **V1,...,Vm**). For each derivative, we can then construct a list of **n+m** elements, where the first **n** elements are the differentiations of the derivative with respect to the corresponding independent variable, and the remaining **m** elements are all zero, except for the one that corresponds to the dependent variable of the derivative.

Ranking examples:

Say that we have a system with independent variables **indep** = **[x,y,z]**, and dependent variables **[f,g,h,s]**. Then the vector for **g[xxz]** can be constructed as **[2,0,1, 0,1,0,0]**. This vector then contains all the information required to identify the original derivative. From the last four items in the list, we see that our dependent variable is **g** (since the 1 corresponds to the placement of **g** in the dependent variable list). We can also see from the first three elements of the list that it is differentiated twice with respect to **x**, 0 times with respect to **y**, and once with respect to **z** (where we are matching up the first three elements of the list to the corresponding independent variables).

With the same system, we may obtain:

```
f[xyz]     [1,1,1, 1,0,0,0]
h[xxxxx]   [5,0,0, 0,0,1,0]
s[xzz]     [1,0,2, 0,0,0,1]
g          [0,0,0, 0,1,0,0]
```

Now we have specified a way to turn each derivative into a list of integer values. Using this, we now can create a new list called a criterion, which must be of the same length and must be specified with integer values. The dot product of the derivative list and the criterion is called the weight of that derivative with respect to the criterion.

So for the above example, if we specified the criterion list to be **[1,0,0, 0,0,0,0]**, then **g[xxz]** would have weight 2, **f[xyz]** would have weight 1, **h[xxxxx]** would have weight 5, and so on.

Now when two derivatives are being compared with respect to one of these list criteria, the ranking would be determined by their respective weights. So, for example, **f[xyz]** < **g[xxz]** with respect to **[1,0,0, 0,0,0,0]**, because **weight(f[xyz]) = 1** is less than **weight(g[xxz]) = 2** with respect to **[1,0,0, 0,0,0,0]**.

The new ranking can then be constructed as a list of these criteria which, during a comparison, are calculated and compared in order. The construction of a ranking in this way is called a *Janet* ranking.

As an example, we can construct the default ranking as a criterion list for the example system as:

```
ranking = [ [0,0,0, 1,1,1,1], # This corresponds to criterion 11
            [1,1,1, 0,0,0,0], # This corresponds to criterion 22
            [1,0,0, 0,0,0,0], # These three lines are criterion 33
            [0,1,0, 0,0,0,0],
            [0,0,1, 0,0,0,0],
            [0,0,0, 4,3,2,1]] # This corresponds to criterion 44
```

So if we compared **f[xyz]** to **f[xyy]**, the weights for the first entry would be 1 and 1, for the second entry 3 and 3, for the third entry 1 and 1, and for the fourth entry 1 and 2, at which point it is recognized that **f[xyz]** $<$ **f[xyy]**.

Specification of the ranking to rifsimp

The ranking is specified on the command line to **rifsimp** as 'ranking = list of criteria', where the criterion list is as described above. We recommend that you specify the dependent variables and independent variables so that the order is known and the ranking behaves as expected.

Additional notes

In the event that the input ranking does not fully specify a ranking (two different indeterminates are not ranked differently by the input ranking), the default ranking is then used (see examples). If the system contains constants, and any of the entries of the input ranking do not have corresponding entries for these constants, then the entries are padded with zeros.

Examples

For examples we will take as input single equations or a system of decoupled equations and observe their solved form in the output. They will be solved for their leading indeterminate.

```
> with(DEtools):
> sys:=[diff(g(x,y,t),x,x)+diff(f(x,y,t),x,y)+
> diff(f(x,y,t),y,y)+diff(f(x,y,t),t)=0];
```

$$sys := [(\frac{\partial^2}{\partial x^2}\,\mathrm{g}(x,\,y,\,t)) + (\frac{\partial^2}{\partial y\,\partial x}\,\mathrm{f}(x,\,y,\,t)) + (\frac{\partial^2}{\partial y^2}\,\mathrm{f}(x,\,y,\,t)) + (\frac{\partial}{\partial t}\,\mathrm{f}(x,\,y,\,t)) = 0]$$

By default, the above will be solved for the **g** derivative, as **f** and **g** have equal weight (criterion 1). The equation is differential order 2, so this narrows it down to the three second order derivatives (criterion 2), but **x** derivatives are of greater weight than **y** derivatives (criterion 3), so the equation will be solved for **g[xx]**:

```
> rifsimp(sys);
```

$$\mathrm{table}([\mathit{Solved} = [\frac{\partial^2}{\partial x^2}\,\mathrm{g}(x,\,y,\,t) = -(\frac{\partial^2}{\partial y\,\partial x}\,\mathrm{f}(x,\,y,\,t)) - (\frac{\partial^2}{\partial y^2}\,\mathrm{f}(x,\,y,\,t)) - (\frac{\partial}{\partial t}\,\mathrm{f}(x,\,y,\,t))]])$$

So how can we solve for **f** instead? The obvious way is to give **f** more weight than **g** by declaring it as the solving variable by using **vars** (alter criterion 1):

```
> rifsimp(sys,[f]);
```

$$\mathrm{table}([\mathit{Solved} = [\frac{\partial^2}{\partial y\,\partial x}\,\mathrm{f}(x,\,y,\,t) = -(\frac{\partial^2}{\partial x^2}\,\mathrm{g}(x,\,y,\,t)) - (\frac{\partial^2}{\partial y^2}\,\mathrm{f}(x,\,y,\,t)) - (\frac{\partial}{\partial t}\,\mathrm{f}(x,\,y,\,t))]])$$

What if we wanted to solve for the **y** derivative of **f**? Well, we could then also weight **y** derivatives greater using **indep**:

```
> rifsimp(sys,[f],indep=[y,x]);
```

$$\mathrm{table}([\mathit{Solved} = [\frac{\partial^2}{\partial y^2}\,\mathrm{f}(x,\,y,\,t) = -(\frac{\partial^2}{\partial x^2}\,\mathrm{g}(x,\,y,\,t)) - (\frac{\partial^2}{\partial y\,\partial x}\,\mathrm{f}(x,\,y,\,t)) - (\frac{\partial}{\partial t}\,\mathrm{f}(x,\,y,\,t))]])$$

Good, but what if we want to solve for the **t** derivative of **f**. This is an unusual example because we are solving for a lower order derivative in terms of higher order derivatives. We could specify a new ranking that weights **t** derivatives higher than everything else:

```
> sample_rank:=[[0,0,0,1,0], # Solve for f over g
> [0,0,1,0,0]]:# t derivs are greatest
> ivars:=       [x,y,t]:
> dvars:=              [f,g]:
```

With the above ranking, **f** is always greater than **g**, and **t** derivatives are always greater than **x** or **y** derivatives of any order. We have to declare the order of occurrence in the command line arguments so that we can match the independent and dependent variables to the **sample_rank** table. (These are typed in above for visualization.)

```
> rifsimp(sys,dvars,indep=ivars,ranking=sample_rank);
```

$$\mathrm{table}([\mathit{Solved} = [\frac{\partial}{\partial t}\,\mathrm{f}(x,\,y,\,t) = -(\frac{\partial^2}{\partial x^2}\,\mathrm{g}(x,\,y,\,t)) - (\frac{\partial^2}{\partial y\,\partial x}\,\mathrm{f}(x,\,y,\,t)) - (\frac{\partial^2}{\partial y^2}\,\mathrm{f}(x,\,y,\,t))]])$$

Note: We did not specify a full ranking, but instead specified as much as we required, then let the default ranking take over.

Note that a ranking like the one above is natural for certain classes of equations. As an example, consider the heat equation **u[t] = u[xx]+u[yy]+u[zz]** where the form of the solved equation is only physically meaningful when solving for the time derivatives in terms of the space derivatives, even when the space derivatives are of higher differential order.

As a final example, we construct a strange ranking that weights **t** derivatives twice as heavily as **x** derivatives. This is done for the following:

```
> sys:=[diff(f(x,t),x,x,x)+diff(f(x,t),t)+diff(f(x,t),x,x),
> diff(g(x,t),t)+diff(g(x,t),x,x)];
```

$$sys := [(\frac{\partial^3}{\partial x^3}\,\mathrm{f}(x,\,t)) + (\frac{\partial}{\partial t}\,\mathrm{f}(x,\,t)) + (\frac{\partial^2}{\partial x^2}\,\mathrm{f}(x,\,t)),\,(\frac{\partial}{\partial t}\,\mathrm{g}(x,\,t)) + (\frac{\partial^2}{\partial x^2}\,\mathrm{g}(x,\,t))]$$

```
> sample_rank:=[[1,2,0,0], # t derivs have 2* weight of x
> [0,1,0,0]]:# t derivs are greatest
> ivars:=       [x,t]:
> dvars:=       [f,g]:
> rifsimp(sys,dvars,indep=ivars,ranking=sample_rank);
```

$$\mathrm{table}([\mathit{Solved} = [\frac{\partial^3}{\partial x^3}\,\mathrm{f}(x,\,t) = -(\frac{\partial}{\partial t}\,\mathrm{f}(x,\,t)) - (\frac{\partial^2}{\partial x^2}\,\mathrm{f}(x,\,t)),\,\frac{\partial}{\partial t}\,\mathrm{g}(x,\,t) = -(\frac{\partial^2}{\partial x^2}\,\mathrm{g}(x,\,t))]])$$

A.8 rifsimp[cases] - case splitting and related options

Description

What is a Case Split?

If an input system is nonlinear with respect to any of its indeterminates (including constants), case splitting may occur.

As an example, consider the ODE in **u(t)**: **u'-a u''** = **0**. **rifsimp** can only isolate the leading indeterminate **u''** when **a** <> **0**. Unfortunately, we do not know if this condition holds, so two possibilities result; either **a** <> **0** or **a** = **0**.

Consider as a second example the equation **u'^2 a-u'-u^2=0**. **rifsimp** needs to isolate the highest power of the leading derivative, **u'^2** which introduces the same two possibilities as for the first example. This second type of pivot is called the *initial* of the equation.

This prompts the following definitions:

Pivot

The coefficient of the highest power of the leading indeterminate in an unsolved equation. It is not known whether this coefficient is nonzero or whether it vanishes with respect to the resulting system.

Case Split

A case split in the **rifsimp** algorithm is the separate handling of the cases of nonzero or zero pivot during the solution of an equation in the system. By default (without **casesplit** on), only the nonzero case is explored.

In the **u[t]-a u[tt]** = **0** example, the pivot is **a**, and the generic **a** <> **0** case is handled by default.

Exploring All Cases

For some systems, a simplified result for all cases is required. To obtain this, use the **casesplit** option. Specification of **casesplit** in the arguments to **rifsimp** tells it to examine all cases in the following way:

Proceed to find the generic solution by assuming that all pivots are nonzero. After each solution is found, back up to the most recent nonzero assumption, change to a zero assumption, and proceed to find the solution under this new restriction. The process is complete when no nonzero assumptions are left to back up to.

The result is a tree of disjoint cases, with the simplified forms presented in a table. Each case is stored as a numbered entry in the output table, having all the standard entries, along with an entry called **Case** that can be used to reconstruct a tree of case splittings using the `caseplot` command. This **Case** entry is a list of assumptions made to obtain the answer for that case. Each assumption is a two or three element list, the first element is the assumption made, the second is the derivative being isolated that required the assumption, and if the third entry is present, it indicates that is was later found that the assumption is always true, so the assumption does not represent a true case split of the problem, but rather a *false split*.

There are three options below that mildly alter this behavior, and these are the **faclimit**, **factoring**, and **grobonly** options. The first two allow pivots to be introduced to the system that do not necessarily occur as the coefficient of a specific leading derivative. When one of these pivots occur, then the second element of the assumption is the name of the option (rather than the leading derivative). The third option, **grobonly**, has the effect of introducing fewer pivots (as equations that are nonlinear in their leading derivative, such as the equation in the second example at the top of this page, do not directly introduce case splittings), but has the disadvantage that a less efficient nonlinear equation handling method (pure Groebner basis) must be employed. For more detail on the output see `rifsimp[output]`, and for more detail on nonlinear equation handling see `rifsimp[nonlinear]`.

Case Restrictions

Two options can be used to limit or restrict which cases are explored:

1) **casesplit=[‘<>‘,‘=‘,‘< >‘]**

This option turns on **casesplit** and gives the start of the tree to be examined. It is used primarily for repeating a calculation, or for avoiding particularly difficult cases. **Note:** A rerun of the system with additional equations is likely to change the order of the pivots, so this option is only reliable for the same input as the initial calculation. If a calculation for a specific case is to be repeated with additional equations, it would be better to append the assumptions made in the **Case** list to the original system, and run **rifsimp** with the combined system.

2) **mindim=[...]**

This option is used to limit the cases being examined based on the solution space dimension of the linear part of the current system. Though the dimension of the solution space is not known until the computation is complete, an upper bound is available by looking at the current **Solved** equations. **Note:** In the presence of nonlinear constraints, the computed dimension represents an upper bound on the dimension of the case. This could be used, for example, to find all forms of a second order ODE that are related by a Lie-point transformation to the equation **y"=0**, by setting the minimum free parameters of the Lie-determining system to 8. The minimum dimension can be specified in a number of ways, including some shortcuts:

```
mindim=n              Minimum of n free 0-dimensional parameters in
                      the dependent variables in vars.

mindim=[v,n]          Minimum of n free 0-dimensional parameters in
                      the dependent variables v (v is either a
                      dependent variable name or set of more than one).

mindim=[v,n,d]        Minimum of n free d-dimensional parameters in
                      the dependent variables v, where v is as above.

mindim=[c1,c2,...]    Here c1,c2,... are conditions of the [v,n,d] type.
```

When this option is used, a **dimension** entry is also provided for each case in the output system. See the information in `rifsimp[output]` for more detail.

Note: When using multiple specifications, each must be a full specification defining the number, the dependent variables, and the dimension of the required data.

If any of the input conditions fail, computation is halted on the current case, it is tagged with the status *"free count fell below mindim"*, and the computation proceeds to the next case.

Pivoting and Pivot Selection

The **rifsimp** algorithm proceeds by putting as many PDEs into solved form as possible without introducing new pivots. When it has reached a point where none of the remaining unsolved equations can be solved without the introduction of a pivot, it must then decide which equation to solve. By default, **rifsimp** chooses the leading linear unsolved equation that is smallest in size, but this behavior can be modified by the **pivselect** option described below. The new pivot for the chosen equation then results in a case split if **casesplit** is set, otherwise exploration of only the nonzero (generic) case if **casesplit** is not set. If none of the equations are leading linear, then no standard pivot can be found, so **rifsimp** then attempts to factor the leading nonlinear equations searching for factors that do not depend on the leading derivative. If these are found, then the smallest factor (based on the Maple `length` function) is chosen, and a **"factor"** split is performed. This behavior can be modified through use of the **factoring** option described below. If nonlinear equations remain, then the coefficient of the highest degree of the leading derivative in the equation (the *initial* of the equation) is split on if **initial** has been specified (see below) or if **grobonly** has not been specified (see `rifsimp[nonlinear]`).

The following options relate to pivoting and to the control of pivoting in the course of the algorithm:

1) **pivselect=choice**

During the course of a computation, **rifsimp** proceeds by performing as much elimination as possible before introducing a pivot. Once no further elimination can be done, a pivot must be introduced, and there is generally more than one possible choice. There are currently six possible choices for pivot selection:

```
"smalleq"          Choose the pivot belonging to the smallest
                   equation (based on the Maple length function).
                   This is the default.

"smallpiv"         Similar to "smalleq", but the length of the
                   pivots are compared instead.

"lowrank"          Choose the pivot for the equation with
                   leading derivative of lowest rank. Ties
```

```
                    are broken by equation length.

"mindim"            Choose the pivot for the equation with
                    leading derivative that will reduce the size
                    of the initial data by the greatest amount.
                    This option can only be used in combination
                    with the mindim option described above.

["smalleq",vars]    Same as "smalleq" above, but it only looks at
                    equations with leading derivatives in the
                    vars sequence.

["lowrank",vars]    Same as "lowrank" above, but it only looks at
                    equations with leading derivatives in the
                    vars sequence.
```

The choice of a pivoting strategy can significantly affect the efficiency of a computation for better or worse. In addition to efficiency considerations, the selection of different pivoting strategies can also simplify the resulting case structure, though the choice that gives the best case structure is highly problem dependent.

The "smalleq" and "smallpiv" options are generally focussed on efficiency. "smalleq" makes the computation more efficient by selecting smaller equations for splitting, while "smallpiv" makes the computation more efficient by introducing smaller pivots.

The "lowrank" and "mindim" options are focussed on keeping the number of cases to a minimum.

2) **nopiv=[var1, ...]**

This option gives a list of all variables and constants that cannot be present in pivots. This is most useful for quasi-linear systems for which the linearity of the resulting system is of greatest importance. This may also have the effect of reducing the size of the case tree in these systems. The unsolved equations are then treated as fully nonlinear (see `rifsimp[nonlinear]`).

3) **faclimit=<n>**

Typically the pivot chosen is the coefficient of the leading indeterminate in the equation. In the event that the leading indeterminate is itself a factor of the equation, and this same leading indeterminate factor occurs in <n> or more equations, then it is chosen as the pivot rather than as the coefficient. This is most useful for Lie determining systems having many inconsistent cases, since in these systems, many equations of this form typically occur during the calculation. The answer may be obtained without this option, but it can significantly reduce the number of cases returned.

4) **factoring=desc**

As briefly described above, by default equations that factor are split if one of the factors does not depend upon the leading derivative of that equation. This option can be used to modify that behavior based on the value of **desc** used in the **factoring** option:

```
nolead        This is the default. Only split on a factor
              if it does not contain the leading derivative
              of the equation.

all           Split on any factors, even if they contain the
              leading derivative.

none          Do not perform factored splitting.
```

In any event, the **factoring** pivots are always the last to be considered (i.e. faclimit and regular pivots are used if available). **Note: factoring="all"** should be used with caution as it has the potential to significantly increase the number of cases returned. **Note: factoring="none"** can result in non-termination for some nonlinear problems.

5) **pivsub**

When introducing a new pivot, attempt to reword that pivot as a new dependent variable when it is advantageous to do so. For example, if the new pivot is **a(x)+b(x)-c(x,y)**, then **rifsimp** creates a new dependent variable, say **F1(x,y)**, and adds an equation to the system indicating that **F1(x,y) = a(x)+b(x)^2-c(x,y)**. The ranking would be arranged such that the new equation would be solved in terms of an existing indeterminate in the pivot, and the overall effect would be to make the new pivot a single-term expression.

The primary advantage of this strategy is that, after introducing a pivot, the solution procedure usually results in the occurrence of that pivot and its derivatives, sometimes to a high polynomial degree. Leaving the pivot in its regular form can result in extensive expression swell, whereas the reworded pivot remains quite small and manageable.

The new dependent variables are chosen as **F.(i),F.(i+1)**, and so on, where **i** is larger than the largest occurrence of an **F.(j)** in the present system, and is not globally assigned.

6) **initial**

This option only applies to systems which have nonlinear **Constraint** equations in their output. The *initial* is the coefficient of the highest power of the leading derivative in a polynomially nonlinear equation (the *leading nonlinear derivative*). For example, in the equation **u[x]^2*u+u[x]^2+u[x]+u-1**, the initial is the coefficient of **u[x]^2**, which is **u+1**. By default, **rifsimp** splits on the initial, unless **grobonly** is specified (in which case it does not need to), introducing additional cases in the output. With these additional cases, **rifsimp** isolates the leading nonlinear derivatives in the **Constraint** equations. When **grobonly** is not specified, only the form of the **Constraint** output is different.

Note: These options do not require that **casesplit** be on, but are typically most useful in that situation.

Other Case-related Options

Another option that is also related to case-splitting is the **gensol** option, which may return multiple cases for some problems:

gensol

This option indicates that the program should explore all cases that have the possibility of leading to the most general solution of the problem. Occasionally it is possible for **rifsimp** to compute only the case corresponding to the general solution of the ODE/PDE system. When this option is given, and this occurs, **rifsimp** will return the single case corresponding to the general solution. When it is not possible for **rifsimp** to detect where in the case tree the general solution is, then multiple cases are returned, one or more of which correspond to the general solution of the ODE/PDE system, and others correspond to singular solutions. For some particularly difficult problems, it is possible that the entire case tree is returned. **Note:** this option cannot be used with the **casesplit**,**casecount**, **pivsub**, and **mindim** options.

Examples

```
> with(DEtools):
```

Suppose we have the ODE shown below. We want to determine conditions on **g(y(x))** and **f(x)** that allow the equation to be mapped to **y" = 0** by an invertible change of variables of the form **Y=Y(x,y), X=X(x,y)**. It is known that the equation can be mapped to **y"=0** if the Lie group of the equation is 8-dimensional. This is the perfect opportunity to use the **mindim** option in combination with cases, to tell **rifsimp** that we are only interested in cases with 8 dimensions or more:

```
> ODE1:=diff(y(x),x,x)+g(y(x))*diff(y(x),x)+f(x)*y(x);
```

$$\mathit{ODE1} := (\frac{\partial^2}{\partial x^2}\,\mathrm{y}(x)) + \mathrm{g}(\mathrm{y}(x))\,(\frac{\partial}{\partial x}\,\mathrm{y}(x)) + \mathrm{f}(x)\,\mathrm{y}(x)$$

We can use the DEtools **odepde** command to generate the determining system for this ODE, and we obtain the following:

```
> sys1:=[coeffs(expand(odepde(ODE1,[xi(x,y),eta(x,y)], y(x))),_y1)];
```

$$\begin{aligned}
\mathit{sys1} := [&\mathrm{g}(y)\,(\frac{\partial}{\partial x}\,\eta(x,\,y)) + \eta(x,\,y)\,\mathrm{f}(x) - (\frac{\partial}{\partial y}\,\eta(x,\,y))\,\mathrm{f}(x)\,y + \xi(x,\,y)\,(\frac{\partial}{\partial x}\,\mathrm{f}(x))\,y \\
&+ 2\,(\frac{\partial}{\partial x}\,\xi(x,\,y))\,\mathrm{f}(x)\,y + (\frac{\partial^2}{\partial x^2}\,\eta(x,\,y)),\; -(\frac{\partial^2}{\partial y^2}\,\xi(x,\,y)), \\
&-2\,(\frac{\partial^2}{\partial y\,\partial x}\,\xi(x,\,y)) + 2\,(\frac{\partial}{\partial y}\,\xi(x,\,y))\,\mathrm{g}(y) + (\frac{\partial^2}{\partial y^2}\,\eta(x,\,y)), \eta(x,\,y)\,(\frac{\partial}{\partial y}\,\mathrm{g}(y)) \\
&+ 3\,(\frac{\partial}{\partial y}\,\xi(x,\,y))\,\mathrm{f}(x)\,y + (\frac{\partial}{\partial x}\,\xi(x,\,y))\,\mathrm{g}(y) + 2\,(\frac{\partial^2}{\partial y\,\partial x}\,\eta(x,\,y)) - (\frac{\partial^2}{\partial x^2}\,\xi(x,\,y))]
\end{aligned}$$

Applying **rifsimp**:

```
> ans1:=rifsimp(sys1,[xi,eta],casesplit,mindim=8):
> caseplot(ans1);
```

Issuing the caseplot command above would show that there is one case for which this occurs. This case is given by:

```
> copy(ans1[3]);
```

$$\begin{aligned}
&\mathrm{table}([\mathit{Solved} = [\tfrac{\partial^3}{\partial y^2\,\partial x}\,\eta(x,\,y) = -2\,\mathrm{f}(x)\,(\tfrac{\partial}{\partial y}\,\xi(x,\,y)),\ \tfrac{\partial^3}{\partial y^3}\,\eta(x,\,y) = 0,\\
&\tfrac{\partial^2}{\partial x^2}\,\xi(x,\,y) = 3\,(\tfrac{\partial}{\partial y}\,\xi(x,\,y))\,\mathrm{f}(x)\,y + (\tfrac{\partial}{\partial x}\,\xi(x,\,y))\,\mathrm{g}(y) + 2\,(\tfrac{\partial^2}{\partial y\,\partial x}\,\eta(x,\,y)),\ \tfrac{\partial^2}{\partial x^2}\,\eta(x,\,y)\\
&= -\mathrm{g}(y)\,(\tfrac{\partial}{\partial x}\,\eta(x,\,y)) - \eta(x,\,y)\,\mathrm{f}(x) + (\tfrac{\partial}{\partial y}\,\eta(x,\,y))\,\mathrm{f}(x)\,y - \xi(x,\,y)\,(\tfrac{\partial}{\partial x}\,\mathrm{f}(x))\,y\\
&-2\,(\tfrac{\partial}{\partial x}\,\xi(x,\,y))\,\mathrm{f}(x)\,y,\ \tfrac{\partial^2}{\partial y\,\partial x}\,\xi(x,\,y) = (\tfrac{\partial}{\partial y}\,\xi(x,\,y))\,\mathrm{g}(y) + \frac{1}{2}\,(\tfrac{\partial^2}{\partial y^2}\,\eta(x,\,y)),\\
&\tfrac{\partial^2}{\partial y^2}\,\xi(x,\,y) = 0,\ \tfrac{\partial}{\partial y}\,\mathrm{g}(y) = 0],\\
&\mathit{Case} = [[\tfrac{\partial^2}{\partial y^2}\,\mathrm{g}(y) = 0,\ \tfrac{\partial}{\partial x}\,\xi(x,\,y)],\ [(\tfrac{\partial}{\partial y}\,\mathrm{g}(y))\,\mathrm{g}(y) = 0,\ \tfrac{\partial}{\partial y}\,\xi(x,\,y)]],\\
&\mathit{dimension} = 8\\
&])
\end{aligned}$$

So the original ODE is equivalent to **y”=0** when **g’(y)** is zero, regardless of the form of **f(x)**.

As a demonstration of the **faclimit** option, consider the following system:

```
> sys2:=[eta(x,y)*(3*f(x)+2*g(x)-h(x)),
> eta(x,y)*(f(x)-2*g(x)-5*h(x)+2),
> eta(x,y)*(-f(x)-3*g(x)+h(x)-3),
> eta(x,y)*(3*f(x)-3*g(x)-5*h(x)-1)];
```

$$\begin{aligned}
&\mathit{sys2} := [\eta(x,\,y)\,(3\,\mathrm{f}(x) + 2\,\mathrm{g}(x) - \mathrm{h}(x)),\ \eta(x,\,y)\,(\mathrm{f}(x) - 2\,\mathrm{g}(x) - 5\,\mathrm{h}(x) + 2),\\
&\eta(x,\,y)\,(-\mathrm{f}(x) - 3\,\mathrm{g}(x) + \mathrm{h}(x) - 3),\ \eta(x,\,y)\,(3\,\mathrm{f}(x) - 3\,\mathrm{g}(x) - 5\,\mathrm{h}(x) - 1)]
\end{aligned}$$

The regular case-splitting strategy produces an undesirable result for this system, namely more cases than required to describe the system:

```
> ans2_1:=rifsimp(sys2,[eta],casesplit);
```

$ans2\_1 :=$ table([
$1 =$ table([$Solved = [\eta(x, y) = 0]$, $Pivots = [\%2 \neq 0]$, $Case = [[\%2 \neq 0, \eta(x, y)]]]), 2$
$=$ table([$Solved = [\eta(x, y) = 0, \mathrm{f}(x) = -\frac{2}{3}\,\mathrm{g}(x) + \frac{1}{3}\,\mathrm{h}(x)]$, $Pivots = [\%1 \neq 0]$,
$Case = [[\%2 = 0, \eta(x, y)], [\%1 \neq 0, \eta(x, y)]]$
]), $3 =$ table([$Solved = [\eta(x, y) = 0, \mathrm{f}(x) = \frac{3}{2}\,\mathrm{h}(x) - \frac{1}{2}, \mathrm{g}(x) = -\frac{7}{4}\,\mathrm{h}(x) + \frac{3}{4}]$,
$Pivots = [\mathrm{h}(x) - 1 \neq 0]$,
$Case = [[\%2 = 0, \eta(x, y)], [\%1 = 0, \eta(x, y)], [\mathrm{h}(x) - 1 \neq 0, \eta(x, y)]]$
]), $4 =$ table([$Solved = [\mathrm{f}(x) = 1, \mathrm{g}(x) = -1, \mathrm{h}(x) = 1]$,
$Case = [[\%2 = 0, \eta(x, y)], [\%1 = 0, \eta(x, y)], [\mathrm{h}(x) - 1 = 0, \eta(x, y)]]$
]),
$casecount = 4$
])
$\%1 := 4\,\mathrm{g}(x) + 7\,\mathrm{h}(x) - 3$
$\%2 := 3\,\mathrm{f}(x) + 2\,\mathrm{g}(x) - \mathrm{h}(x)$

So we get 4 cases. Now set **faclimit** set to 2:

```
> ans2_2:=rifsimp(sys2,[eta],casesplit,faclimit=2);
```

$ans2\_2 :=$ table([1 = table([$Solved = [\mathrm{f}(x) = 1, \mathrm{g}(x) = -1, \mathrm{h}(x) = 1]$,
$Pivots = [\eta(x,y) \neq 0]$, $Case = [[\eta(x,y) \neq 0,$ "faclimit"]]
]), 2 = table([$Solved = [\eta(x, y) = 0]$, $Case = [[\eta(x,y) = 0,$ "faclimit"]]]),
$casecount = 2$
])

So although both **ans2_1** and **ans2_2** equally valid, it is clear that **ans2_2** would be preferred.

As an example of the factoring option, consider the following purely algebraic system:

```
> sys := [x^3-7*x+6];
```

$$sys := [x^3 - 7\,x + 6]$$

With default options, we obtain

```
> rifsimp(sys,casesplit);
```

$$\mathrm{table}([Constraint = [x^3 - 7\,x + 6 = 0]])$$

With full factoring enabled, we obtain

```
> rifsimp(sys,casesplit,factoring=all);
```

$\text{table}([1 = \text{table}([\mathit{Solved} = [x = 2],\ \mathit{Case} = [[x - 1 \neq 0,\ \text{“factor”}],\ [x + 3 \neq 0,\ \text{“factor”}]]]),$
$2 = \text{table}([\mathit{Solved} = [x = -3],\ \mathit{Case} = [[x - 1 \neq 0,\ \text{“factor”}],\ [x + 3 = 0,\ \text{“factor”}]]]),$
$3 = \text{table}([\mathit{Solved} = [x = 1],\ \mathit{Case} = [[x - 1 = 0,\ \text{“factor”}]]]),$
$\mathit{casecount} = 3$
$])$

So we see that the system has been split into three disjoint cases. Also note that the **Case** entries describe the path the computation took, and there are no **Pivots** entries. This is because the pivots resulting from the case splits are identically satisfied for each case.

A.9 rifsimp[nonlinear] - information and options specific to nonlinear equations

Description

Leading nonlinear equations

This help page explains how **rifsimp** handles equations which are polynomially nonlinear in their leading indeterminate (called leading nonlinear equations), or equations that are leading linear, but have a coefficient that depends on pivot restricted variables (see **nopiv** in `rifsimp[cases]`).

As an example, consider the equation **u[tt] u-u[t]^2 = 0**. This equation is linear in its leading indeterminate, so as long as **u** can be present in a pivot it will be handled through case splitting rather than nonlinear equation handling (see `rifsimp[cases]`). Conversely, the equation **u[tt]^2-u[t] = 0** is leading nonlinear, and is always handled through the nonlinear methods described here.

During the course of the computation, nonlinear equations are always differentially simplified with respect to the linear equations of the system, but they are also algebraically simplified with respect to each other. This simplification process resembles an algebraic Groebner basis, but takes the inequations (pivots) of the system into account, and does not compute algebraic S-polynomials. Use of this set of equations allows recognition of, and elimination of redundant equations, and produces a more compact representation (due to the use of pivots and lack of S-polynomials).

In order to perform the algebraic simplification of these nonlinear equations, a monomial ordering must be imposed for all possible nonlinear terms. This is simply chosen to be pure lexicographical ordering with respect to the ordering imposed on the linear derivatives.

For example, the equation **u[xx]^2 u[x]+u[xx]^2 u-u[x]^2-u^2** for a system in which **u[x]+u<>0** holds is algebraically solved for **u[xx]^2** giving the nonlinear relation:

u[xx]^2=(u[x]^2+u^2)/(u[x]+u).

Note: during the course of the algorithm, when nonlinear equations are encountered, case splitting is performed on the coefficient of the highest degree of the leading derivative each equation. This coefficient is called the *initial* of the equation. So for the example above, the initial is **u[x]+u**. If there was no condition on **u[x]+u**, then **rifsimp** would split into the cases **u[x]+u<>0** and **u[x]+u=0**. This can increase the number of cases in the output, but is required for the proper functioning of the more compact nonlinear equation handling algorithm, so if this splitting is not desired, pure Groebner basis methods can be used instead with the **grobonly** option.

If leading linear equations with usable pivots are found in the algebraically simplified set, **rifsimp** removes them from the set, and treats them using leading linear methods (i.e. isolation of the leading derivative and pivoting, differential substitution, and linear integrability conditions).

Differential consequences are enforced by the spawning (differentiation with respect to each independent variable) of each equation, which naturally results in leading linear equations.

Before algorithm completion, **rifsimp** forms an algebraic Groebner basis over the leading nonlinear equations, spawning any new relations that appear as a result of S-polynomials. This step is necessary to ensure that all differential consequences of the leading non-linear equations are accounted for, so that the set of nonlinear constraint equations returned by the algorithm can be viewed as algebraic (not differential) constraints.

Of course, Groebner basis computations require a ranking of monomials in a system. By default the ranking is chosen as total-degree order with ties broken by inverse lexicographical order (otherwise known as grevlex). The allowed rankings for **rifsimp** are quite flexible, but require some knowledge of rankings (please see `rifsimp[ranking]`, and look at the Groebner Rankings description below).

Control options

There are four options used to control how **rifsimp** handles leading nonlinear equations. The **spoly** and **spawn** options limit what is done in the algorithm, and thus they generally return an incomplete answer. Unfortunately, these options are needed in some cases as highly nonlinear ODE/PDE systems may have to perform a Groebner basis of a large number of equations in a large number of indeterminates. Though the process is finite, it can be extremely time consuming, and may not return the result for days (weeks, months, etc.). These options allow **rifsimp** to return a result that can be further processed using other tools, such as hand integration or resultants, to obtain the final result.

1) **grobonly**

This option tells **rifsimp** to only use Groebner basis simplification of the nonlinear equations in the course of the algorithm instead of using the algebraic simplification described above. This has the potential to decrease the number of cases in the output, but caution should be used with this option, as **rifsimp** works with only part of the system at any one time, so use of full a Groebner basis on only a partial system can often lead to inefficiency.

2) **checkempty**

This option tells **rifsimp** to attempt to determine, on the completion of the calculation (or the completion of each calculation when dealing with cases) whether the resulting system contains any solutions. These empty systems do not occur for linear problems, but when algebraic constraints and pivots are both present for a system, then there is a possibility that the resulting output has no solution. For a simple example, see the last problem in the `examples` section below. Caution should be used for this option also, as the checking process can be quite expensive.

3) **spoly=[true,false]**

This option indicates whether to generate S-polynomials during the Groebner basis algorithm (default is **true**). Setting **spoly=false** has some risk associated with it, as if nonlinear equations are present in the output, then it is possible that not all consequences of these equations have been obtained (hence they are placed in the **DiffConstraint** list rather than the **Constraint** list). If an orderly ranking is used (see `rifsimp[ranking]`), and spawn is set to **true** (see below), then only algebraic simplifications remain, but these may still result in an inconsistent system when considered with the Pivot equations.

Note: By algebraic simplifications we mean that there are no further integrability conditions that can result once the system is algebraically simplified. Differential reduction of leading linear equations can still occur.

4) **spawn=[true,false]**

Indicates whether to perform a differential spawn of nonlinear equations (default is **true**). A differential spawn simply means taking the derivative of an equation with respect to all independent variables. Setting this to **false** allows **rifsimp** to ignore differential consequences of the polynomially nonlinear equations. This is only useful if **rifsimp** is being used as a single step of a different calculation, since an incomplete answer may be obtained. When **spawn=false**, all nonlinear equations are placed in the **DiffConstraint** list rather than the **Constraint** list to indicate that the answer may be incomplete.

Groebner rankings 1

A Groebner ranking is a ranking defined on the set of all monomials in the indeterminates of a system of equations. For example, if the indeterminates were x, y, and z, a ranking would be able to order the monomials **x*y^2*z, x^3*y^2, x^2*y*z^2, x^2*y*z^4**, and determine a leader. The determination of this leader is vital to the completion of a Groebner basis, and different rankings can give drastically different bases for the same input system.

Algebraic monomial rankings

There are a number of standard Groebner basis rankings available for algebraic systems of equations. The rankings described below can be used in **rifsimp**. Each ranking is described as a comparison of two monomials:

1) Total-degree ranking

This only defines a partial ranking; that is, there can be ties for different terms. It looks at the degree of each monomial in all indeterminates (the total degree), and if the total degrees of the monomials are different, the one of larger degree is ranked higher.

2) Lexicographic ranking

Given an ordered list of indeterminates, lexicographic ranking compares the degrees of the two monomials in each indeterminate in turn. If the degree of one monomial is greater than the other in that indeterminate, then it is ranked higher.

3) Inverse Lexicographic ranking

This ranking looks at the ordered list of indeterminates in reverse order, and compares the degrees of the two monomials in each indeterminate in turn. If the degree of one monomial is lower than the other in that indeterminate, then it is ranked higher.

It is important that a ranking is deterministic. That is, given two different monomials it is always possible to rank one higher than the other. Since a total-degree ranking is not deterministic, it must be followed by either a lexicographic or an inverse lexicographic ranking.

As an example we look at all monomials in x, y, and z up to total degree 3 and rank them using the rankings described above.

Total Degree followed by Lexicographic on [x,y,z]

```
x^3 > x^2*y > x^2*z > x*y^2 > x*y*z > x*z^2 > y^3 > y^2*z > y*z^2
 > z^3 > x^2 > x*y > x*z > y^2 > y*z > z^2 > x > y > z > 1
```

Total Degree followed by Lexicographic on [z,y,x]

```
z^3 > y*z^2 > x*z^2 > y^2*z > x*y*z > x^2*z > y^3 > x*y^2 > x^2*y
 > x^3 > z^2 > y*z > x*z > y^2 > x*y > z^2 > z > y > x > 1
```

Total Degree followed by Inverse Lexicographic on [x,y,z]

```
x^3 > x^2*y > x*y^2 > y^3 > x^2*z > x*y*z > y^2*z > x*z^2 > y*z^2
 > z^3 > x^2 > x*y > y^2 > x*z > y*z > z^2 > x > y > z > 1
```

Pure Lexicographic on [x,y,z]

```
x^3 > x^2*y > x^2*z > x^2 > x*y^2 > x*y*z > x*y > x*z^2 > x*z > x
 > y^3 > y^2*z > y^2 > y*z^2 > y*z > y > z^3 > z^2 > z > 1
```

Simple algebraic rankings in rifsimp

The rankings for Groebner basis computations in **rifsimp** are specified by the **grob_rank = [criterion list]** option. If all that is needed is an algebraic ranking as described above (for the indeterminates in the order described by the linear ranking) the criterion list can simply be given as below:

grob_rank=[[1,deg,none],[1,ilex]]

This says to use algebraic total-degree ranking followed by inverse lexicographic ranking (this is the default).

grob_rank=[[1,deg,none],[1,lex]]

Use algebraic total-degree ranking followed by lexicographic ranking.

grob_rank=[[1,lex]]

Use pure lexicographic ranking.

The criterion lists may seem a bit verbose for what they need to accomplish, but far more detailed rankings are available (see Groebner Rankings 2 below).

Though the pure lexicographic ranking produces a more desirable result, total-degree / inverse lexicographic ranking was chosen as the default as it was in general (by experiment on a number of ODE/PDE systems) the least expensive in time and memory.

Groebner Rankings 2: a different point of view

The main thing to note is that all the types of monomial rankings previously discussed can be viewed as looking at the degree of one or more indeterminates between the two monomials being compared.

For example, total degree followed by lexicographic ranking (for the x,y,z example) looks at the degree of the monomials in **x,y,z**, then in **x**, then in **y**, then finally in **z**. Total degree followed by inverse lexicographic ranking looks at the degree of the monomials in **x,y,z**, then **x,y**, then **x,z**, then **y,z**.

So now that we are dealing with differential systems, it may be desirable to rank indeterminates based upon other criteria such as their differential degree, dependent variable, differentiations, etc. This can be accomplished in **rifsimp**.

Classification of linear ranking criteria

As a first step, all linear differential ranking criteria are classified.

```
diffdeg   any criterion comparing differentiations of more than one
          independent variable

diffvar   any criterion involving differentiations of a single independent
          variable

depvar    any criterion only involving dependent variables

other     any criterion which mixes independent and dependent variables

all       all possible criteria

none      no criteria
```

For more detail about criteria, please see `rifsimp[ranking]`. In that help page, an example of the **diffdeg** classification is given by criterion 2, **diffvar** by criterion 3, and **depvar** by criterion 1 and 4.

These classifications, which actually refer to a specific part of the linear ranking, allow the definition of equivalence classes for the derivatives. The monomial ranking then takes the degree of each monomial in derivatives of the same equivalence class.

Definition of the criteria for grob_rank

With the above information we can now construct a differential monomial ranking as an example. Suppose we define our first criterion as **[1,deg,diffdeg]**. This describes equivalence classes based upon the differential degree of the derivatives. Consider a system containing the dependent variable f(x,y) and all derivatives up to second order, and using the default linear ranking for differential degree (all differentiations have equal weight). This defines the following equivalence classes:

```
{f[xx], f[xy], f[yy]}, {f[x], f[y]}, {f}
```

Now when comparing two monomials, the degree of the monomials in each equivalence class is considered in turn, and if they are different, the one with greater degree is ranked higher.

Here are a few examples:

```
f[xx]^2*f[xy]*f[y]*f < f[yy]^4    Degree in second order derivatives
                                  is 3 < 4
f[xy]*f[yy] < f[xx]^3             Degree in second order derivatives
                                  is 2 < 3
f[xy]^2*f[yy]*f[y]*f^20 <         Degree in first order derivatives
 f[yy]^3*f[x]^2*f^3               is 1 < 2.
```

This does not fully determine a ranking, since if there were (for example) two terms that were fourth degree in second order derivatives, this criterion would regard them as equal.

To fully specify the monomial ranking, we could add another criterion of the lex or ilex type. If we added a lex criterion, namely [1,lex], then we would be looking at the degree of the monomials in the indeterminates in their linearly ranked order. Again using the default for the linear ranking, and considering the f(x,y) system up to second order we would get the following:

```
{f[xx]},{f[xy]},{f[yy]},{f[x]},{f[y]},{f}
```

This is presented in the same manner as the equivalence classes for the degree-type monomial ranking.

```
f[xx]^2 f[xy] f[y] f < f[xx]^3 f[x] f  Degree in f[xx] is 2 < 3.
f[xx]^2 f[yy] < f[xx]^2 f[xy]          Degree in f[xy] is 0 < 1.
f[xy]^2 f[yy] f[x] f[y] f <            Degree in f[x] is
 f[xy]^2 f[yy] f[x]^2 f                1 < 2.
```

In summary, one should consider all the criteria as a list of equivalence classes, for which the degree of the monomial in the indeterminates of the equivalence class determines the relative monomial ranking. For **grob_rank=[[1,deg,diffdeg],[1,lex]]**, and the above examples, the list of equivalence classes is given as follows:

```
[{f[xx], f[xy], f[yy]}, {f[x], f[y]}, {f}, {f[xx]}, {f[xy]}, {f[yy]},
 {f[x]}, {f[y]}, {f}]}
```

One final detail – a bit more flexibility

You may have noticed that there is always a 1 present as the first element of each criterion, but it is not discussed. This allows a bit more flexibility in the specification of the ranking, as it allows for nesting of the criteria.

Consider again the earlier example where the criterion was specified as [[1,deg,diffdeg],[1,lex]]. Based on the above description, if the higher derivatives are of equal degree in each order of derivative, then the degree of the lower order derivatives may be used to break the tie. This may not be what is desired. Suppose one wants to examine the degree in the higher-order derivatives, and if they are equal, to compare the higher-order derivatives lexicographically. This can be accomplished through nesting. Any increase in the first element (integer) of a criterion from the prior criterion indicates that the new criterion should be nested inside the prior.

Consider our earlier example system with the new nested ranking **[[1,deg,diffdeg],[2,lex]]**. This would have the effect of changing the order of the combined equivalence classes to the following:

```
[{f[xx], f[xy], f[yy]}, {f[xx]}, {f[xy]}, {f[yy]}, {f[x], f[y]},
 {f[x]}, {f[y]}, {f}, {f}]
```

This tells **rifsimp** to consider the degree of the monomial in second order derivatives, then in each second order derivative in turn, then do the same for first order derivatives, etc.

Here are a few examples:

```
f[xx]^2 f[xy] f[y] f < f[yy]^4     Degree in second order derivatives
                                   is 3 < 4.
f[xy]^2 f[yy] f[y]^6 < f[xx]^3     Degree in f[xx] is 0 < 3*.
f[yy]^3 f[y]^2 f^20 <              Degree in first order derivative
 f[yy]^3 f[x]^2 f^3                f[x] is 0 < 2*.
```

The comparisons with * will give different results for the non-nested ranking.

Examples

The following example arises from the parameterization of an ellipse (where we have used `PDEtools[declare]` to compact the output and display the derivatives in primed notation.

```
> with(DEtools): with(PDEtools):
> declare(x(t),y(t),z(t),prime=t);
```

x(t) will now be displayed as x

y(t) will now be displayed as y

*derivatives with respect to : t of functions of one variable will now be displayed wi\
th '*

z(t) will now be displayed as z

```
> sys1:=[diff(y(t),t)-x(t)^2-x(t),
> y(t)^4-2*y(t)^2*diff(z(t),t)+diff(z(t),t)^2
> -y(t)^2+2*diff(z(t),t),
> diff(x(t),t)^2+2*y(t)*diff(x(t),t)+y(t)^4];
```

$$sys1 := [y' - x^2 - x,\, y^4 - 2\, y^2\, z' + z'^2 - y^2 + 2\, z',\, x'^2 + 2\, y\, x' + y^4]$$

Calling with the default ranking gives the following:

```
> ans1:=rifsimp(sys1);
```

$$\begin{aligned}
&ans1 := \mathrm{table}([Case = [[x' + y \neq 0,\, x''],\, [-y^2 + z' + 1 \neq 0,\, z'']],\\
&Pivots = [x' + y \neq 0,\, -y^2 + z' + 1 \neq 0],\, Constraint = [x'^4 + 2\,z'^2\,x'^2 + z'^4 + 4\,x'^3\,y\\
&+ 4\,z'^2\,y\,x' - 8\,y^3\,x' + 6\,z'\,x'^2 + 6\,z'^3 + 12\,z'\,y\,x' + 2\,x'^2 + 11\,z'^2 + 4\,y\,x' - 3\,y^2\\
&+ 6\,z' = 0,\\
&4\,y^2\,x'^2 + 8\,y^3\,x' - 6\,z'\,x'^2 - 2\,z'^3 - 12\,z'\,y\,x' - x'^2 - 7\,z'^2 - 2\,y\,x' + 3\,y^2 - 6\,z' = 0\\
&,\, x'^2 + 2\,y\,x' + y^4 = 0,\, 2\,y^2\,z' + x'^2 - z'^2 + 2\,y\,x' + y^2 - 2\,z' = 0],\\
&Solved = [x'' = -\frac{x\,(2\,x+2)\,y^3 + x\,(x+1)\,x'}{x' + y},\\
&z'' = -\frac{-x\,(2\,x+2)\,y^3 - x\,(-2\,x-2)\,z'\,y - x\,(-x-1)\,y}{y^2 - z' - 1},\, y' = x\,(x+1)]\\
&])
\end{aligned}$$

So we have isolated ODE for **y'**, **x"** and **z"**, and four constraints involving **x**, **y**,**x'**, and **z'**.

```
> nops(ans1[Constraint]);
```

$$4$$

If instead we want to perform an elimination of **x**, then **z**, then **y**, we can specify this through use of a lex ranking for the algebraic problem.

```
> ans2:=rifsimp(sys1,[[x],[z],[y]],grob_rank=[[1,lex]]);
```

$$\begin{aligned}
&ans2 := \mathrm{table}([Case = [[-2\,x - 1 \neq 0,\, x'],\, [y''\,y \neq 0,\, x],\, [-y^2 + z' + 1 \neq 0,\, z''],\\
&[(1 + 4\,y')\,(4\,y^4\,y' + y^4 - y''^2) \neq 0,\, y''']],\\
&Pivots = [-y''^2 - y^4 - 4\,y^4\,y' \neq 0,\, y'' \neq 0,\, -y^2 + z' + 1 \neq 0,\, 4\,y^4\,y' + y^4 - y''^2 \neq 0],\\
&Constraint = [y^4 - 2\,y^2\,z' + z'^2 - y^2 + 2\,z' = 0,\\
&-16\,y'\,y''^2\,y^2 - 4\,y''^2\,y^2 + y''^4 + 2\,y''^2\,y^4 + 8\,y''^2\,y^4\,y' + y^8 + 8\,y^8\,y' + 16\,y^8\,y'^2 =\\
&0],\\
&Solved = [x = -\frac{1}{4}\,\frac{2\,y''\,y + y''^2 + y^4 + 4\,y^4\,y'}{y''\,y},\, z'' = -\frac{-2\,y^3\,y' + 2\,y\,z'\,y' + y\,y'}{y^2 - z' - 1},\\
&y''' = -\frac{-12\,y''\,y^4\,y'^2 - 3\,y''\,y^4\,y' + y'\,y''^3 - 8\,y^9\,y' - 6\,y''^2\,y^5 - 2\,y^9 + 8\,y''^2\,y^3}{4\,y^5\,y' + y^5 - y''^2\,y}]\\
&])
\end{aligned}$$

so now we have isolated ODE for **x** and **z"**, an ODE in **y"'** in terms of **y** alone, and two constraints involving **z'**,**y"**,**y'** and **y**.

For an example of the use of the **grobonly** and **checkempty** options we consider the following algebraic system:

```
> sys3:=Groebner[gbasis]([(b+a^2)^2,a^3-c],tdeg(a,b,c));
```

$$sys3 := [-3\,b\,a\,c + b^3 - 2\,c^2,\, 2\,b\,c + a^2\,c + a\,b^2,\, a\,c + b^2 + 2\,b\,a^2,\, a^3 - c]$$

If we call rifsimp with this system, and the inequation **a^2+b<>0** we get:

```
> ans3_1:=rifsimp([op(sys3),a^2+b<>0],nopiv=[b],casesplit,grobonly);
```

$ans3\_1 :=$ table([$Pivots = [b + a^2 \neq 0]$,
$Constraint =$
$[a^3 - c = 0,\, a\,c + b^2 + 2\,b\,a^2 = 0,\, 2\,b\,c + a^2\,c + a\,b^2 = 0,\, -3\,b\,a\,c + b^3 - 2\,c^2 = 0]$
])

But with the additional option 'checkempty' we get

```
> ans3_2:=rifsimp([op(sys3),a^2+b<>0],nopiv=[b],casesplit,grobonly,
> checkempty);
```

$ans3\_2 :=$ table([$status =$ "system is inconsistent"])

because **ans3_1** represents an empty case.

A.10 maxdimsystems - determination of maximum dimensional subsystems of systems of PDE

Calling Sequences

maxdimsystems(**system**, **maxdimvars**)

maxdimsystems(**system**, **maxdimvars**, **options**)

Parameters

system - list or set of ODE or PDE (may contain inequations)

maxdimvars - list of the dependent variables used for the dimension count

options - (optional) sequence of options to control the behavior of **maxdimsystems**

Description

The default output of **maxdimsystems** (modulo *** below) consists of the set of cases having maximal dimension in **maxdimvars**, where each member has the form:

```
[Solved = list, Constraint = list, Pivots = list, dimension = integer]
```

Solved is a list of the PDE which are in solved form with respect to their highest derivative (with respect to the given ranking of derivatives), **Constraint** is a list of the PDEs in the classvars (any dependent variables not in **maxdimvars**) which are nonlinear in their leading derivatives, **Pivots** is a list of the inequations in the classvars and/or maxdimvars, and **dimension** is the maximal degree of arbitrariness in maxdimvars (i.e. the number of arbitrary constants on which maxdimvars depends).

*** The result is valid provided that the **Constraint** and **Pivots** equations have a solution, and also that **maxdimvars** do not occur in the constraint equations. The user must check the first condition algebraically (e.g. see `Groebner[gsolve]`, and possibly `diffalg`). If the **maxdimvars** are present in the pivots, then the returned system will have lower dimension that indicated, and must be handled manually.

Each of the cases is in `rifsimp` form (a generalization of Gauss normal form to systems of PDE) and also contains a system of differential equations and inequations in the classvars characterizing the case. In the same way that the Gauss normal form is determined by a ordering of variables, **rifsimp** form is determined by a ranking of derivatives.

The default ranking is determined by **maxdimvars**. It can be inferred by the application of `checkrank`, and always ranks the classvars and their derivatives lower than all **maxdimvars**.

The maximal cases can be determined in many applications where a complete analysis of all cases is not possible. These cases are often the most interesting, with the richest properties, when compared to the relatively uninteresting (but often difficult to compute) lower dimensional cases.

The options for **maxdimsystems** consist of the **output** option (described below), and a subset of those for `rifsimp`, the main subroutine used by maxdimsystems. Of the subset of **rifsimp** options available, two that are often used are **indep** which specifies the independent variables, **ezcriteria** which controls the flow of equations from a fast (ultra-hog) setting (that can readily lead to memory explosion but sometimes faster calculations), to a conservative (one equation at a time) setting that can overcome some expression swell problems. The complete list is given by:

`tosolve, indep, arbitrary, clean, fullclean`	`See rifsimp[options]`
`ezcriteria, faclimit`	`See rifsimp[adv_options]`
`mindim, pivselect`	`See rifsimp[cases]`
`grob_rank,grobonly,checkempty`	`See rifsimp[nonlinear]`

Many **rifsimp** options are not supported by **maxdimsystems** as their use could produce misleading results. For example, the **rifsimp** option **ctl=time**, places a time limit on rifsimp case computations, and hence could cause **maxdimsystems** to miss the maximal case if it takes too long to compute. Conversely, specification of other options are not supported as they must have a specific value for the **maxdimsystems** command to function properly (for example, **casesplit** must always be on). Other unsupported **rifsimp** options include **itl**, **stl**, **nopiv**, **unsolved**, **pivsub**, **store**, **storeall**, **spawn**, **spoly**, and **casecount**.

output = type

The output type can either be **set** (default) or **rif**. Note that **set** output is easier to read and manipulate, but cannot be used with the `caseplot` visualization tool. The **rif** output can be used with **caseplot**, but the output is somewhat more complicated, and is described in `rifsimp[output]`. Note that for the class of problems that **maxdimsystems** simplifies, many of the outputs from **rifsimp** described in the output page (such as **DiffConstraint** and **UnSolve**) will never be present in the output.

The **maxdimsystems** command allows a subset of the rankings available for **rifsimp**. Specifically those rankings that are controlled by a partition of the dependent variables in **maxdimvars**, and those available by the order of the independent variables in **indep**. See `rifsimp[ranking]` and `checkrank` for an explanation of how to specify rankings based on these options.

The application of `caseplot` to the **rif**-form output of **maxdimsystems** with the second argument given as the **maxdimvars** displays the tree of maximal cases and their dimensions, also indicating which branches of the tree have less than maximal dimension.

Note that **maxdimsystems** is essentially a simplified version of the **rifsimp** interface that also has the ability to automatically determine the maximum finite dimensional cases or all infinite dimensional cases for a classification ODE/PDE system. If the dimension of the solution space is known, multiple conditions on the dimension of the solution space are needed, or much finer control of the computation is desired, then `rifsimp` could be used with the **mindim** option described in `rifsimp[cases]`.

References:

G.J. Reid and A.D. Wittkopf, "Determination of Maximal Symmetry Groups of Classes of Differential Equations", *Proc. ISSAC 2000*, pp.272-280

G.W. Bluman and S. Kumei, "Symmetries and Differential Equations", *Springer-Verlag*, vol. 81.

Examples

```
> with(DEtools):
```

Algebraic systems can be examined with **maxdimsystems**

```
> maxdimsystems([ u - a*v = 0, u - c*v = 0],  [u,v]);
```

$$\{[Solved = [u = c\,v,\, a = c],\, Constraint = [\,],\, Pivots = [\,],\, dimension = 1]\}$$

In the below example, inequations are given in the **Pivots** list, and the dimension is 2

```
> maxdimsystems([ (a(x)-d(x))*diff(u(x),x,x) +
> b(x)*diff(u(x),x) + u(x) = 0],
> [u]);
```

$$\left\{\left[\mathit{Solved} = \left[\frac{\partial^2}{\partial x^2}\,\mathrm{u}(x) = \frac{-\mathrm{b}(x)\,(\frac{\partial}{\partial x}\,\mathrm{u}(x)) - \mathrm{u}(x)}{\mathrm{a}(x) - \mathrm{d}(x)}\right],\ \mathit{Pivots} = [\mathrm{a}(x) - \mathrm{d}(x) \neq 0],\right.\right.$$
$$\left.\left.\mathit{dimension} = 2,\ \mathit{Constraint} = [\,]\right]\right\}$$

This next example has a constraint on the classvars present in the result

```
> maxdimsystems([diff(u(x,y),x) - f(x,y)*u(x,y) = 0,
> diff(u(x,y),y) - g(x,y)*u(x,y) = 0,
> f(x,y)^3 - 1 = 0],  [u]);
```

$$\{[\mathit{Solved} = [\tfrac{\partial}{\partial x}\,\mathrm{u}(x,\,y) = \mathrm{f}(x,\,y)\,\mathrm{u}(x,\,y),\ \tfrac{\partial}{\partial y}\,\mathrm{u}(x,\,y) = \mathrm{g}(x,\,y)\,\mathrm{u}(x,\,y),\ \tfrac{\partial}{\partial x}\,\mathrm{f}(x,\,y) = 0,$$
$$\tfrac{\partial}{\partial x}\,\mathrm{g}(x,\,y) = 0,\ \tfrac{\partial}{\partial y}\,\mathrm{f}(x,\,y) = 0],\ \mathit{Constraint} = [\mathrm{f}(x,\,y)^3 - 1 = 0],\ \mathit{Pivots} = [\,],$$
$$\mathit{dimension} = 1]\}$$

In each of the above examples **maxdimsystems** picked out the maximum dimensional case. This is in contrast to **rifsimp**, with its **casesplit** option, which obtains all cases.

For the second example, computation with **rifsimp** using casesplit gives 2-d, 1-d and 0-d cases, of which the 2-d case is the maximal case from the above example.

```
> ans := rifsimp([ (a(x)-d(x))*diff(u(x),x,x) +
> b(x)*diff(u(x),x) + u(x) = 0],
> [u], casesplit);
```

$$
\begin{aligned}
&\mathit{ans} := \mathrm{table}([1 = \mathrm{table}([\mathit{Case} = [[\mathrm{a}(x) - \mathrm{d}(x) \neq 0,\ \tfrac{\partial^2}{\partial x^2}\,\mathrm{u}(x)]],\\
&\mathit{Solved} = \left[\tfrac{\partial^2}{\partial x^2}\,\mathrm{u}(x) = \frac{-\mathrm{b}(x)\,(\frac{\partial}{\partial x}\,\mathrm{u}(x)) - \mathrm{u}(x)}{\mathrm{a}(x) - \mathrm{d}(x)}\right],\\
&\mathit{Pivots} = [\mathrm{a}(x) - \mathrm{d}(x) \neq 0]\\
&]), 2 = \mathrm{table}([\mathit{Case} = [[\mathrm{a}(x) - \mathrm{d}(x) = 0,\ \tfrac{\partial^2}{\partial x^2}\,\mathrm{u}(x)],\ [\mathrm{b}(x) \neq 0,\ \tfrac{\partial}{\partial x}\,\mathrm{u}(x)]],\\
&\mathit{Solved} = [\tfrac{\partial}{\partial x}\,\mathrm{u}(x) = -\frac{\mathrm{u}(x)}{\mathrm{b}(x)},\ \mathrm{a}(x) = \mathrm{d}(x)],\\
&\mathit{Pivots} = [\mathrm{b}(x) \neq 0]\\
&]), 3 = \mathrm{table}([\mathit{Case} = [[\mathrm{a}(x) - \mathrm{d}(x) = 0,\ \tfrac{\partial^2}{\partial x^2}\,\mathrm{u}(x)],\ [\mathrm{b}(x) = 0,\ \tfrac{\partial}{\partial x}\,\mathrm{u}(x)]],\\
&\mathit{Solved} = [\mathrm{u}(x) = 0,\ \mathrm{a}(x) = \mathrm{d}(x),\ \mathrm{b}(x) = 0]\\
&]),\\
&\mathit{casecount} = 3\\
&])
\end{aligned}
$$

from which the `caseplot` with initial data counts could be viewed by issuing the command:

```
> caseplot(ans,[u]);
```

One example of practical use of maxdimsystems is the determination of the forms of **f(y')** (where **f(y')** is not identically zero) for which the ode **y"=f(y')** has the maximum dimension.

```
> ODE:=diff(diff(y(x),x),x)=f(diff(y(x),x));
```

$$\mathit{ODE} := \frac{\partial^2}{\partial x^2}\,\mathrm{y}(x) = \mathrm{f}(\frac{\partial}{\partial x}\,\mathrm{y}(x))$$

```
> symeq:=[DEtools[odepde](ODE, [xi(x,y),eta(x,y)])];
```

$$
\begin{aligned}
&\mathit{symeq} := [(-3\,\_y1\,(\tfrac{\partial}{\partial y}\,\xi(x,\,y)) - 2\,(\tfrac{\partial}{\partial x}\,\xi(x,\,y)) + (\tfrac{\partial}{\partial y}\,\eta(x,\,y)))\,\mathrm{f}(\_y1) +\\
&(-\_y1\,(\tfrac{\partial}{\partial y}\,\eta(x,\,y)) + \_y1^2\,(\tfrac{\partial}{\partial y}\,\xi(x,\,y)) + \_y1\,(\tfrac{\partial}{\partial x}\,\xi(x,\,y)) - (\tfrac{\partial}{\partial x}\,\eta(x,\,y)))\\
&(\tfrac{\partial}{\partial \_y1}\,\mathrm{f}(\_y1)) + (\tfrac{\partial^2}{\partial x^2}\,\eta(x,\,y)) + (\tfrac{\partial^2}{\partial y^2}\,\eta(x,\,y))\,\_y1^2 - (\tfrac{\partial^2}{\partial y^2}\,\xi(x,\,y))\,\_y1^3\\
&- 2\,(\tfrac{\partial^2}{\partial y\,\partial x}\,\xi(x,\,y))\,\_y1^2 + 2\,(\tfrac{\partial^2}{\partial y\,\partial x}\,\eta(x,\,y))\,\_y1 - \_y1\,(\tfrac{\partial^2}{\partial x^2}\,\xi(x,\,y))]
\end{aligned}
$$

where the above **symeq** is an overdetermined system of PDE in the infinitesimal symmetries **xi(x,y),eta(x,y)**.

So we compute the maximum dimensional case which, is not displayed as it is quite large:

```
> t1 := time():
> msys:= maxdimsystems([op(symeq),f(_y1)<>0],[xi,eta]):
> time()-t1;
```

$$3.471$$

```
> nops(msys);
```

$$1$$

```
> length(msys[1]);
```

$$5362$$

```
> msys[1][4];
```

$$dimension = 8$$

```
> remove(has,rhs(msys[1][1]),[xi,eta]);
```

$$[\frac{\partial^4}{\partial \_y1^4} \mathrm{f}(\_y1) = 0]$$

from which we see that we have one case, of dimension 8, and it occurs when the displayed ODE in **f(_y1)** is satisfied. This classical result is also explored in the ISSAC Proc. article listed in the references (the primary reference for this command).

The timing above should be compared to the **rifsimp** full case analysis that takes significantly longer, and has a great number of lower dimensional cases:

```
> t2 := time():
> rsys:= rifsimp([op(symeq),f(_y1)<>0],[xi,eta],'casesplit'):
> time()-t2;
```

$$88.840$$

```
> rsys['casecount'];
```

$$11$$

Consider the nonlinear Heat equation

```
> diff(u(x,t),t) = diff( k(u(x,t))*diff(u(x,t),x), x);
```

$$\frac{\partial}{\partial t} \mathrm{u}(x, t) = \mathrm{D}(k)(\mathrm{u}(x, t)) \left(\frac{\partial}{\partial x} \mathrm{u}(x, t)\right)^2 + \mathrm{k}(\mathrm{u}(x, t)) \left(\frac{\partial^2}{\partial x^2} \mathrm{u}(x, t)\right)$$

The overdetermined system for it's symmetries **X,T,U** is given by

```
> sys:= [
> -2*diff(T(u,t,x),x),
> diff(diff(U(u,t,x),u),u)-2*diff(diff(X(u,t,x),u),x)+U(u,t,x)
> *diff(diff(k(u),u),u)/k(u)+1/k(u)*diff(k(u),u)*diff(U(u,t,x),u)
> -U(u,t,x)/k(u)^2*diff(k(u),u)^2,
> 1/k(u)*diff(X(u,t,x),u)*diff(k(u),u)-diff(diff(X(u,t,x),u),u),
> diff(diff(U(u,t,x),x),x)-1/k(u)*diff(U(u,t,x),t),
> 1/k(u)*diff(T(u,t,x),t)+U(u,t,x)/k(u)^2*diff(k(u),u)
> -diff(diff(T(u,t,x),x),x)-2/k(u)*diff(X(u,t,x),x),
> -diff(diff(X(u,t,x),x),x)+2*diff(diff(U(u,t,x),u),x)+1/k(u)
> *diff(X(u,t,x),t)+2*diff(k(u),u)/k(u)*diff(U(u,t,x),x),
> -2*diff(k(u),u)/k(u)*diff(T(u,t,x),x)-2/k(u)*diff(X(u,t,x),u)
> -2*diff(diff(T(u,t,x),u),x),
> -1/k(u)*diff(k(u),u)*diff(T(u,t,x),u)-diff(diff(T(u,t,x),u),u),
> -2*diff(T(u,t,x),u)]:
```

At the same time as determining maximum dimensional cases, one can request, through a partitioning of the **maxdimvars**, that the dependent variables be `ranked` in a specific manner. To always isolate derivatives of **U(u,t,x)** in terms of the other dependent variables, one could specify **maxdimvars=[[U],[X,T]]**. For this problem we would obtain

```
> maxdimsystems([op(sys),diff(k(u),u)<>0],[[U],[X,T]]);
```

$$\left\{\left[Solved = \left[\mathrm{U}(u,\,t,\,x) = \frac{\mathrm{k}(u)\,(-(\frac{\partial}{\partial t}\,\mathrm{T}(u,\,t,\,x)) + 2\,(\frac{\partial}{\partial x}\,\mathrm{X}(u,\,t,\,x)))}{\frac{\partial}{\partial u}\,\mathrm{k}(u)},\ \frac{\partial^3}{\partial x^3}\,\mathrm{X}(u,\,t,\,x) = 0,\right.\right.\right.$$

$$\frac{\partial^2}{\partial t^2}\,\mathrm{T}(u,\,t,\,x) = 0,\ \frac{\partial}{\partial u}\,\mathrm{X}(u,\,t,\,x) = 0,\ \frac{\partial}{\partial u}\,\mathrm{T}(u,\,t,\,x) = 0,\ \frac{\partial}{\partial t}\,\mathrm{X}(u,\,t,\,x) = 0,$$

$$\left.\frac{\partial}{\partial x}\,\mathrm{T}(u,\,t,\,x) = 0,\ \frac{\partial^2}{\partial u^2}\,\mathrm{k}(u) = \frac{7}{4}\,\frac{(\frac{\partial}{\partial u}\,\mathrm{k}(u))^2}{\mathrm{k}(u)}\right],\ Constraint = [\,],\ Pivots = [\frac{\partial}{\partial u}\,\mathrm{k}(u) \neq 0],$$

$$\left.\left.dimension = 5\right]\right\}$$

Alternatively we could try to obtain PDE only in **U(u,t,x)** and the classifying function **k(u)** by an elimination ranking:

```
> maxdimsystems([op(sys),diff(k(u),u)<>0],[[X,T],[U]]);
```

$$\left\{\left[\mathit{Solved} = \left[\frac{\partial^2}{\partial x^2}\mathrm{X}(u,t,x) = \frac{1}{2}\,\frac{(\frac{\partial}{\partial u}\mathrm{k}(u))\,(\frac{\partial}{\partial x}\mathrm{U}(u,t,x))}{\mathrm{k}(u)},\ \frac{\partial}{\partial u}\mathrm{X}(u,t,x) = 0,\ \frac{\partial}{\partial u}\mathrm{T}(u,t,x) = 0,\right.\right.\right.$$

$$\frac{\partial}{\partial t}\mathrm{X}(u,t,x) = 0,\ \frac{\partial}{\partial t}\mathrm{T}(u,t,x) = \frac{-\mathrm{U}(u,t,x)\,(\frac{\partial}{\partial u}\mathrm{k}(u)) + 2\,(\frac{\partial}{\partial x}\mathrm{X}(u,t,x))\,\mathrm{k}(u)}{\mathrm{k}(u)},$$

$$\frac{\partial}{\partial x}\mathrm{T}(u,t,x) = 0,\ \frac{\partial^2}{\partial x^2}\mathrm{U}(u,t,x) = 0,\ \frac{\partial}{\partial u}\mathrm{U}(u,t,x) = -\frac{3}{4}\,\frac{\mathrm{U}(u,t,x)\,(\frac{\partial}{\partial u}\mathrm{k}(u))}{\mathrm{k}(u)},$$

$$\left.\frac{\partial}{\partial t}\mathrm{U}(u,t,x) = 0,\ \frac{\partial^2}{\partial u^2}\mathrm{k}(u) = \frac{7}{4}\,\frac{(\frac{\partial}{\partial u}\mathrm{k}(u))^2}{\mathrm{k}(u)}\right],\ \mathit{Constraint} = [\,],\ \mathit{Pivots} = [\frac{\partial}{\partial u}\,\mathrm{k}(u) \neq 0],$$

$$\left.\left.\mathit{dimension} = 5\right]\right\}$$

where you can see from the above that we have three PDE for **U(u,t,x)** alone (which can be quite easily solved for **U(u,t,x)** in terms of a function that is linear in **x** with coefficients as integrals in **u** depending on **k(u)**).

Both computations show that maximal group is of dimension 5, and occurs when **k(u)** satisfies **k" = 7/4 (k')^2/k**. This is a known result (see the Bluman/Kumei reference).

In determining how a specific ranking will be used in a system, one should consult the `checkrank` help page.

The final example demonstrates how to find near-maximal cases, and how to obtain **rif**-style output. Say we want all cases having dimension 3 or greater, and we want the rifsimp output (so we can display the information in a case plot). This can be done as follows:

```
> rifans:=maxdimsystems([op(sys),diff(k(u),u)<>0],[X,T,U],mindim=4,
> output=rif);
```

$rifans := \mathrm{table}([1 = \mathrm{table}([\mathit{dimension} = 3, \mathit{UnClass} = [0 = \mathrm{U}(u,t,x)(\frac{\partial}{\partial u}\,\mathrm{k}(u))^2$
$(\frac{\partial^2}{\partial u^2}\,\mathrm{k}(u)) - 2\,\mathrm{U}(u,t,x)\,(\frac{\partial^2}{\partial u^2}\,\mathrm{k}(u))^2\,\mathrm{k}(u) + \mathrm{U}(u,t,x)\,(\frac{\partial}{\partial u}\,\mathrm{k}(u))\,\mathrm{k}(u)\,(\frac{\partial^3}{\partial u^3}\,\mathrm{k}(u))],$
$\mathit{Case} = [[\%4 \neq 0,\ \mathrm{U}(u,t,x)]],\ \mathit{status} =$ "free count fell below mindim",

$$\mathit{Solved} = \left[\frac{\partial^2}{\partial x^2}\,\mathrm{X}(u,t,x) = \frac{1}{2}\,\frac{(\frac{\partial}{\partial u}\,\mathrm{k}(u))\,(\frac{\partial}{\partial x}\,\mathrm{U}(u,t,x))}{\mathrm{k}(u)},\ \frac{\partial}{\partial u}\,\mathrm{U}(u,t,x) = \right.$$

$$\left. -\frac{\mathrm{U}(u,t,x)\,(\%2 - (\frac{\partial}{\partial u}\,\mathrm{k}(u))^2)}{(\frac{\partial}{\partial u}\,\mathrm{k}(u))\,\mathrm{k}(u)},\ \frac{\partial}{\partial t}\,\mathrm{X}(u,t,x) = \frac{1}{2}\,\frac{(\frac{\partial}{\partial x}\,\mathrm{U}(u,t,x))\,\%3}{\frac{\partial}{\partial u}\,\mathrm{k}(u)},\ \%1\right],$$

$\mathit{DiffConstraint} = [0 = 8\,(\frac{\partial}{\partial x}\,\mathrm{U}(u,t,x))\,(\frac{\partial^2}{\partial u^2}\,\mathrm{k}(u))\,\mathrm{k}(u)\,(\frac{\partial}{\partial u}\,\mathrm{k}(u))^2$
$+\,4\,(\frac{\partial}{\partial x}\,\mathrm{U}(u,\,t,\,x))\,(\frac{\partial}{\partial u}\,\mathrm{k}(u))\,(\frac{\partial^3}{\partial u^3}\,\mathrm{k}(u))\,\mathrm{k}(u)^2$
$-8\,(\frac{\partial}{\partial x}\,\mathrm{U}(u,\,t,\,x))\,(\frac{\partial^2}{\partial u^2}\,\mathrm{k}(u))^2\,\mathrm{k}(u)^2 - 7\,(\frac{\partial}{\partial x}\,\mathrm{U}(u,\,t,\,x))\,(\frac{\partial}{\partial u}\,\mathrm{k}(u))^4],$
$\mathit{Pivots} = [\frac{\partial}{\partial u}\,\mathrm{k}(u) \neq 0]]),$
$2 = \mathrm{table}([\mathit{dimension} = 4,\ \mathit{Case} = [[\%4 = 0,\ \mathrm{U}(u,t,x)],\ [\%3 \neq 0,\ \frac{\partial}{\partial x}\,\mathrm{U}(u,t,x)]],$

$$\mathit{Solved} = \left[\frac{\partial^2}{\partial x^2}\,\mathrm{X}(u,t,x) = 0,\ \frac{\partial}{\partial u}\,\mathrm{X}(u,t,x) = 0,\ \frac{\partial}{\partial u}\,\mathrm{T}(u,t,x) = 0, \frac{\partial}{\partial u}\,\mathrm{U}(u,t,x) = \right.$$

$$\frac{\mathrm{U}(u,t,x)\,(-\%2 + (\frac{\partial}{\partial u}\,\mathrm{k}(u))^2)}{(\frac{\partial}{\partial u}\,\mathrm{k}(u))\,\mathrm{k}(u)},\ \frac{\partial}{\partial t}\,\mathrm{X}(u,t,x) = 0,\ \%1,\ \frac{\partial}{\partial t}\,\mathrm{U}(u,\,t,\,x) = 0,$$

$$\left. \frac{\partial}{\partial x}\mathrm{T}(u,t,x) = 0,\ \frac{\partial}{\partial x}\mathrm{U}(u,t,x) = 0,\ \frac{\partial^3}{\partial u^3}\mathrm{k}(u) = -\frac{(\frac{\partial^2}{\partial u^2}\mathrm{k}(u))((\frac{\partial}{\partial u}\mathrm{k}(u))^2 - 2\,\%2)}{(\frac{\partial}{\partial u}\mathrm{k}(u))\mathrm{k}(u)}\right],$$

$\mathit{Pivots} = [\%3 \neq 0]]),$
$3 = \mathrm{table}([\mathit{dimension} = 5,\ \mathit{Case} = [[\%4 = 0,\ \mathrm{U}(u,t,x)],\ [\%3 = 0,\ \frac{\partial}{\partial x}\,\mathrm{U}(u,t,x)]],$

$$\mathit{Solved} = \left[\frac{\partial^2}{\partial x^2}\,\mathrm{X}(u,t,x) = \frac{1}{2}\,\frac{(\frac{\partial}{\partial u}\,\mathrm{k}(u))\,(\frac{\partial}{\partial x}\,\mathrm{U}(u,t,x))}{\mathrm{k}(u)},\ \frac{\partial^2}{\partial x^2}\,\mathrm{U}(u,t,x) = 0,\right.$$

$$\frac{\partial}{\partial u}\,\mathrm{X}(u,t,x) = 0,\ \frac{\partial}{\partial u}\,\mathrm{T}(u,t,x) = 0,\ \frac{\partial}{\partial u}\,\mathrm{U}(u,t,x) = -\frac{3}{4}\,\frac{\mathrm{U}(u,\,t,\,x)\,(\frac{\partial}{\partial u}\,\mathrm{k}(u))}{\mathrm{k}(u)},$$

$$\frac{\partial}{\partial t}\,\mathrm{X}(u,t,x) = 0,\ \%1,\ \frac{\partial}{\partial t}\,\mathrm{U}(u,\,t,\,x) = 0,\ \frac{\partial}{\partial x}\,\mathrm{T}(u,\,t,\,x) = 0,$$

$$\left.\frac{\partial^2}{\partial u^2}\,\mathrm{k}(u) = \frac{7}{4}\,\frac{(\frac{\partial}{\partial u}\,\mathrm{k}(u))^2}{\mathrm{k}(u)}\right],\mathit{Pivots} = [\frac{\partial}{\partial u}\,\mathrm{k}(u) \neq 0]]),$$

$\mathit{casecount} = 3])$

$$\%1 := \frac{\partial}{\partial t}\,\mathrm{T}(u,\,t,\,x) = \frac{-\mathrm{U}(u,t,x)\,(\frac{\partial}{\partial u}\,\mathrm{k}(u)) + 2\,(\frac{\partial}{\partial x}\,\mathrm{X}(u,t,x))\,\mathrm{k}(u)}{\mathrm{k}(u)}$$

$$\%2 := (\frac{\partial^2}{\partial u^2}\,\mathrm{k}(u))\,\mathrm{k}(u)$$

$$\%3 := -7\,(\frac{\partial}{\partial u}\,\mathrm{k}(u))^2 + 4\,\%2$$

$$\%4 := (\frac{\partial}{\partial u}\,\mathrm{k}(u))^2\,(\frac{\partial^2}{\partial u^2}\,\mathrm{k}(u)) - 2\,(\frac{\partial^2}{\partial u^2}\,\mathrm{k}(u))^2\,\mathrm{k}(u) + (\frac{\partial}{\partial u}\,\mathrm{k}(u))\,\mathrm{k}(u)\,(\frac{\partial^3}{\partial u^3}\,\mathrm{k}(u))$$

All cases where the initial data was smaller than the bound are tagged with a status of "free count fell below mindim". This calculation can then be displayed in a caseplot with the command:

```
> caseplot(rifans,[X,T,U]);
```

A.11 initialdata - find initial data for a solved-form ODE or PDE system

Calling Sequence

initialdata(**system**)

initialdata(**system**,**vars**)

initialdata(**rifresult**)

initialdata(**rifresult**,**vars**)

Parameters

system - list or set of polynomially nonlinear PDEs or ODEs in solved form

vars - the dependent variables to compute the initial data with respect to

rifresult - a single case result as returned by `rifsimp`

Description

The **initialdata** function determines the initial data required to obtain a formal power series solution for a PDE or ODE system at a point (see `rtaylor`). The input system must be in standard or involutive form, so all equations must be of the form **derivative = expression**, and there must be no additional relations between the dependent variables that are not accounted for by the equations of the system. It can be used on any PDE or ODE system in isolated form, as well as for a single case result output by `rifsimp`. In general, all integrability conditions (equality of mixed partials) must be satisfied to obtain an accurate answer, so it is best to use **rifsimp** to simplify the system before calculation of the initial data.

The optional **vars** argument allows specification of the variables to compute the initial data of. This option is useful for *classification* problems, where different dependent variables play different roles in the system. This option also provides a way to specify the variables of a problem and their dependencies (in the event that one of the dependent variables does not appear in the **Solved** list, for example).

The output is given in table form with the **Finite** entry containing the required zero dimensional initial data (constants), and the **Infinite** entry containing the required higher dimensional data (functions).

When the input to **initialdata** is a **rifresult**, then additional information is provided on output. As well as the infinite and finite initial data, the resulting table may contain a **Pivots** and/or **Constraint** entry (if these entries are present in the **rifresult**). These returned lists of equations are given in terms of the initial data.

When **Constraint** equations are present in the result, these equations must be satisfied by the initial data for the underlying system to be 'well posed'. These **Constraint**s must be accounted for manually.

Notes:

When not using the **rifresult** form of the **initialdata** call, any nonlinear constraints must be converted to initial data form and imposed manually.

The initial data required for a system is independent of the right hand side of the solved form equations, so for the examples below we may simply ignore the right hand side.

Background Information: Formal Power Series and the **initialdata** algorithm

Being able to obtain a formal power series solution of a PDE or ODE system at a point essentially means that the values of all derivatives of the solution at that point are known, or can be found through differential relations from the system.

As a simple example, consider the ODE in **u(x)**, **u" = u u'** (where prime notation is used to denote differentiation). Here the initial data would simply be **u(x0)** and **u'(x0)**. With this data and the ODE, we could then determine **u"(x0) = u(x0) u'(x0)**, and from the derivative of the ODE, we could obtain **u"'(x0) = u'(x0)^2+u(x0) u"(x0)**, and so on.

Knowledge of all derivatives then allows us to write **u(x) = u(x0)+(x-x0) u'(x0)+(x-x0)^2 u(x0) u'(x0)/2+'...'** to any desired order.

As an example of the **initialdata** algorithm in two dimensions, consider the following system:

```
> sys := [diff(f(x,y),x,x,x)=0,diff(f(x,y),x,y)=0];
```

$$sys := [\frac{\partial^3}{\partial x^3}\,\mathrm{f}(x,\,y) = 0,\ \frac{\partial^2}{\partial y\,\partial x}\,\mathrm{f}(x,\,y) = 0]$$

```
> initialdata(sys);
```

table([*Infinite* = [f(x_0, y) = _F1(y)],
Finite = [$\mathrm{D}_1(f)(x_0, y_0)$ = _*C1*, $\mathrm{D}_{1,1}(f)(x_0, y_0)$ = _*C2*]
])

In this case we have one arbitrary function and two arbitrary constants. Plots can be used to visualize how the **initialdata** algorithm calculates this data. Consider an integer-valued graph where the x-axis represents differentiation by **x**, and the y-axis represents differentiation by **y**. Looking at **f[xxx]** from our original system, we can place dots at that value and at all of its derivatives:

```
> d1:=[3,0],[4,0],[3,1],[4,1],[3,2],[4,2],[3,3],[4,3],[3,
> plot([d1], x=-0.1..4.1, y=-0.1..4.1, style=point,
> xtickmarks=[0,1,2,3,4], ytickmarks=[0,1,2,3,4]);
```

```
4 +                          H         H
  |
  |
3 +                          H         H
  |
  |
2 +                          H         H
  |
  |
1 +                          H         H
  |
  |
 -+--------+--------+--------*--------*-
  |        1        2        3        4
```

Note, for example, that the point (4,1) corresponds to **f[xxxxy]**, which can be obtained from the differentiation of **f[xxx]** with respect to **x** and **y**. Adding in all derivatives that are known from the **f[xy]** equation gives the following graph:

```
> d2:=[1,1],[2,1],[3,1],[4,1],[1,2],[2,2],[3,2],[4,2],
> [1,3],[2,3],[3,3],[4,3],[1,4],[2,4],[3,4],[4,4]:
> plot([d1,d2], x=-0.1..4.1, y=-0.1..4.1, style=point,
> xtickmarks=[0,1,2,3,4], ytickmarks=[0,1,2,3,4]);
```

```
4 +        A        A        A        A
  |
  |
3 +        A        A        A        A
  |
  |
2 +        A        A        A        A
  |
  |
1 +        A        A        A        A
  |
  |
 -+--------+--------+--------*--------*-
  |        1        2        3        4
```

Now the objective is simply to fill in all missing points. From our graph we can see that all points of the form **(0,i)** are missing, which explains the arbitrary function of **y** in the answer. This function describes the data for all **y** derivatives of **f**. We still require the points (1,0) and (2,0), which correspond to the arbitrary constants present in the answer for the values of **f[x]** and **f[xx]**, as given in the **Finite** list.

Examples

```
> with(DEtools):
> sys1:=[diff(f(x,y,z),x,x)=0,diff(f(x,y,z),y)=0,diff(f(x,y,z),z)=0];
```

$$sys1 := [\frac{\partial^2}{\partial x^2}\,\mathrm{f}(x,\,y,\,z) = 0,\ \frac{\partial}{\partial y}\,\mathrm{f}(x,\,y,\,z) = 0,\ \frac{\partial}{\partial z}\,\mathrm{f}(x,\,y,\,z) = 0]$$

This system is fully specified if the following derivatives are known.

```
> initialdata(sys1);
```

$$\mathrm{table}([\mathit{Finite} = [\mathrm{f}(x_0,\,y_0,\,z_0) = \_C1,\ \frac{\partial}{\partial x}\,\mathrm{f}(x_0,\,y_0,\,z_0) = \_C2],\ \mathit{Infinite} = []])$$

For this next system, the initial data contains a number of arbitrary functions of one variable.

```
> sys2:=[diff(f(x,y,z),x,y)=0,diff(f(x,y,z),x,z)=0,
> diff(f(x,y,z),y,z)=0];
```

$$sys2 := [\frac{\partial^2}{\partial y\,\partial x}\,\mathrm{f}(x,\,y,\,z) = 0,\ \frac{\partial^2}{\partial z\,\partial x}\,\mathrm{f}(x,\,y,\,z) = 0,\ \frac{\partial^2}{\partial z\,\partial y}\,\mathrm{f}(x,\,y,\,z) = 0]$$

```
> initialdata(sys2);
```

$$\mathrm{table}([\mathit{Finite} = [],\\ \mathit{Infinite} = [\mathrm{f}(x,\,y_0,\,z_0) = \_\mathrm{F1}(x),\ \frac{\partial}{\partial y}\,\mathrm{f}(x_0,\,y,\,z_0) = \_\mathrm{F2}(y),\ \frac{\partial}{\partial z}\,\mathrm{f}(x_0,\,y_0,\,z) = \_\mathrm{F3}(z)]\\ ])$$

For the next system, the initial data contains a number of arbitrary functions of two variables.

```
> sys3:=[diff(f(x,y,z),x,y,z)=0];
```

$$sys3 := [\frac{\partial^3}{\partial z\,\partial y\,\partial x}\,\mathrm{f}(x,\,y,\,z) = 0]$$

```
> initialdata(sys3);
```

$$\mathrm{table}([\mathit{Finite} = [],\\ \mathit{Infinite} =\\ [\mathrm{f}(x,\,y,\,z_0) = \_\mathrm{F1}(x,\,y),\ \frac{\partial}{\partial z}\,\mathrm{f}(x,\,y_0,\,z) = \_\mathrm{F2}(x,\,z),\ \frac{\partial^2}{\partial z\,\partial y}\,\mathrm{f}(x_0,\,y,\,z) = \_\mathrm{F3}(y,\,z)]\\ ])$$

Of course, we must include the one-dimensional heat equation.

```
> sys4:=[diff(u(x,t),t)=diff(u(x,t),x,x)];
```

$$sys4 := [\frac{\partial}{\partial t}\,\mathrm{u}(x,\,t) = \frac{\partial^2}{\partial x^2}\,\mathrm{u}(x,\,t)]$$

```
> initialdata(sys4);
```

$$\mathrm{table}([\mathit{Finite} = [],\ \mathit{Infinite} = [\mathrm{u}(x,\,t_0) = \_\mathrm{F1}(x)]])$$

This example is a system that contains mixed data:

```
> sys5:=[diff(f(x,y,z),x,y,y)=0,diff(f(x,y,z),x,z)=0,diff(f(x,y,z),y,z,
> z)=0];
```

$$sys5 := [\frac{\partial^3}{\partial y^2\,\partial x}\,\mathrm{f}(x,\,y,\,z) = 0,\ \frac{\partial^2}{\partial z\,\partial x}\,\mathrm{f}(x,\,y,\,z) = 0,\ \frac{\partial^3}{\partial z^2\,\partial y}\,\mathrm{f}(x,\,y,\,z) = 0]$$

```
> initialdata(sys5);
```

$$\begin{aligned}&\mathrm{table}([\mathit{Finite} = [\tfrac{\partial^2}{\partial z\,\partial y}\,\mathrm{f}(x_0,\,y_0,\,z_0) = \_C1],\\&\mathit{Infinite} = [\mathrm{f}(x,\,y_0,\,z_0) = \_\mathrm{F1}(x),\ \tfrac{\partial}{\partial y}\,\mathrm{f}(x,\,y_0,\,z_0) = \_\mathrm{F2}(x),\ \tfrac{\partial}{\partial z}\,\mathrm{f}(x_0,\,y_0,\,z) = \_\mathrm{F3}(z),\\&\tfrac{\partial^2}{\partial y^2}\,\mathrm{f}(x_0,\,y,\,z_0) = \_\mathrm{F4}(y),\ \tfrac{\partial^3}{\partial z\,\partial y^2}\,\mathrm{f}(x_0,\,y,\,z_0) = \_\mathrm{F5}(y)]\\&])\end{aligned}$$

And an example of use of initialdata with a nonlinear rif result:

```
> sys6:=[diff(u(x),x,x)^2*u(x)+diff(u(x),x)^3];
```

$$\mathit{sys6} := [(\tfrac{\partial^2}{\partial x^2}\,\mathrm{u}(x))^2\,\mathrm{u}(x) + (\tfrac{\partial}{\partial x}\,\mathrm{u}(x))^3]$$

```
> rif6:=rifsimp(sys6);
```

$$\begin{aligned}&\mathit{rif6} := \mathrm{table}([\mathit{Solved} = \left[\tfrac{\partial^3}{\partial x^3}\,\mathrm{u}(x) = \frac{1}{2}\,\frac{-(\tfrac{\partial^2}{\partial x^2}\,\mathrm{u}(x))\,(\tfrac{\partial}{\partial x}\,\mathrm{u}(x)) - 3\,(\tfrac{\partial}{\partial x}\,\mathrm{u}(x))^2}{\mathrm{u}(x)}\right],\\&\mathit{Pivots} = [\tfrac{\partial^2}{\partial x^2}\,\mathrm{u}(x) \neq 0],\ \mathit{Constraint} = [(\tfrac{\partial^2}{\partial x^2}\,\mathrm{u}(x))^2\,\mathrm{u}(x) + (\tfrac{\partial}{\partial x}\,\mathrm{u}(x))^3 = 0],\\&\mathit{Case} = [[\mathrm{u}(x) \neq 0,\ (\tfrac{\partial^2}{\partial x^2}\,\mathrm{u}(x))^2],\ [\tfrac{\partial^2}{\partial x^2}\,\mathrm{u}(x) \neq 0,\ \tfrac{\partial^3}{\partial x^3}\,\mathrm{u}(x)]]\\&])\end{aligned}$$

```
> initialdata(rif6);
```

$$\begin{aligned}&\mathrm{table}([\mathit{Pivots} = [\_C3 \neq 0],\ \mathit{Constraint} = [\_C3^2\,\_C1 + \_C2^3 = 0],\\&\mathit{Finite} = [\mathrm{u}(x_0) = \_C1,\ \tfrac{\partial}{\partial x}\,\mathrm{u}(x_0) = \_C2,\ \tfrac{\partial^2}{\partial x^2}\,\mathrm{u}(x_0) = \_C3],\\&\mathit{Infinite} = []\\&])\end{aligned}$$

where it is understood that the selected initial data must obey the nonlinear constraint (so in truth we only have two free parameters) and must not violate the pivots (so **_C3<>0**, and as a result **_C2<>0**).

A.12 rtaylor - obtain the Taylor series for an ODE or PDE system

Calling Sequences

rtaylor(**solved**, **options**)

rtaylor(**solved**, **vars**, **options**)

Parameters

solved - system in solved form

vars - (optional) solving variables of the system

options - (optional) sequence of options to specify the ranking for the solved form, initial data, and the order of the Taylor series

Description

The **rtaylor** function uses an output **rifsimp** form to obtain local Taylor series expansions for all dependent variables in the ODE or PDE system simultaneously. The Taylor series output is a list containing equations of the form **depvar(indepvars)=Taylor series**.

The ranking related options that are accepted by **rtaylor** include the specification of the **vars** as a ranking, and the **ranking** and **indep** options described in `rifsimp[ranking]`, `rifsimp[options]`, and `rifsimp[adv_options]`.

Note: specification of different **vars** than those used to obtain the result from `rifsimp` can give incomplete results.

The **order=n** option specifies the order that the Taylor series should be computed to, and must be a non-negative integer. The default value is **2**.

The table resulting from a call to `initialdata` can be given as an option to **rtaylor**, in which case the Taylor series will be given in terms of the functions present in the initial data.

In general, any **Constraint** or **DiffConstraint** relations (see `rifsimp[nonlinear]`) in the **rif** form cannot be used in an automatic way, so they are ignored. These relations must be accounted for manually after the Taylor series calculation. Special care must be taken when **DiffConstraint** relations are present, because **all** derivatives of these relations must be manually accounted for. This is not the case for **Constraint** relations, as they are purely algebraic.

The requirement that the input solved form be in **rif** form can be relaxed mildly, but **rtaylor** still requires that the equations are in a valid solved form that matches the input ranking (given in the options), and have no integrability conditions remaining. Only when these conditions hold is the resulting Taylor series an accurate representation of the local solution.

Examples

```
> with(DEtools):
```

A simple ODE

```
> rtaylor([diff(f(x),x,x)=-f(x)],order=4);
```

$$[\mathrm{f}(x) = \mathrm{f}(x_0) + (\tfrac{\partial}{\partial x}\,\mathrm{f}(x_0))\,(x - x_0) - \frac{1}{2}\,\mathrm{f}(x_0)\,(x - x_0)^2 - \frac{1}{6}\,(\tfrac{\partial}{\partial x}\,\mathrm{f}(x_0))\,(x - x_0)^3 + \frac{1}{24}\,\mathrm{f}(x_0)\,(x - x_0)^4]$$

A PDE system with a single dependent variable

```
> rtaylor([diff(f(x,y),y,y)=diff(f(x,y),x)*f(x,y),
> diff(f(x,y),x,x)=2*f(x,y)], order=3);
```

$$\begin{aligned}[\mathrm{f}(x,\,y) &= \mathrm{f}(x_0,\,y_0) + \%1\,(x - x_0) + (\tfrac{\partial}{\partial y}\,\mathrm{f}(x_0,\,y_0))\,(y - y_0) + \mathrm{f}(x_0,\,y_0)\,(x - x_0)^2\\ &+ (\tfrac{\partial^2}{\partial y\,\partial x}\,\mathrm{f}(x_0,\,y_0))\,(x - x_0)\,(y - y_0) + \frac{1}{2}\,\%1\,\mathrm{f}(x_0,\,y_0)\,(y - y_0)^2 + \frac{1}{3}\,\%1\,(x - x_0)^3\\ &+ (\tfrac{\partial}{\partial y}\,\mathrm{f}(x_0,\,y_0))\,(x - x_0)^2\,(y - y_0) + \frac{1}{2}\,(2\,\mathrm{f}(x_0,\,y_0)^2 + \%1^2)\,(x - x_0)\,(y - y_0)^2\\ &+ \frac{1}{6}\,((\tfrac{\partial^2}{\partial y\,\partial x}\,\mathrm{f}(x_0,\,y_0))\,\mathrm{f}(x_0,\,y_0) + \%1\,(\tfrac{\partial}{\partial y}\,\mathrm{f}(x_0,\,y_0)))\,(y - y_0)^3]\end{aligned}$$

$$\%1 := \tfrac{\partial}{\partial x}\,\mathrm{f}(x_0,\,y_0)$$

A PDE system with two dependent variables

```
> rtaylor([diff(f(x,y),x,x)=diff(g(x,y),y),
> diff(f(x,y),y,y)=diff(g(x,y),x),
> diff(g(x,y),x)=diff(g(x,y),y)]);
```

$$[\mathrm{f}(x,y)=\mathrm{f}(x_0,y_0)+(\frac{\partial}{\partial x}\mathrm{f}(x_0,y_0))(x-x_0)+(\frac{\partial}{\partial y}\mathrm{f}(x_0,y_0))(y-y_0)+\frac{1}{2}(\frac{\partial}{\partial y}\mathrm{g}(x_0,y_0))(x-x_0)^2$$
$$+(\frac{\partial^2}{\partial y\,\partial x}\mathrm{f}(x_0,y_0))(x-x_0)(y-y_0)+\frac{1}{2}(\frac{\partial}{\partial x}\mathrm{g}(x_0,y_0))(y-y_0)^2,\mathrm{g}(x,y)=\mathrm{g}(x_0,y_0)$$
$$+(\frac{\partial}{\partial y}\mathrm{g}(x_0,y_0))(x-x_0)+(\frac{\partial}{\partial y}\mathrm{g}(x_0,y_0))(y-y_0)+\frac{1}{2}(\frac{\partial^2}{\partial y^2}\mathrm{g}(x_0,y_0))(x-x_0)^2$$
$$+(\frac{\partial^2}{\partial y^2}\mathrm{g}(x_0,y_0))(x-x_0)(y-y_0)+\frac{1}{2}(\frac{\partial^2}{\partial y^2}\mathrm{g}(x_0,y_0))(y-y_0)^2]$$

An example using initial data

```
> sys := {diff(f(x,y),x,x)=0,diff(f(x,y),x,y)=0};
```

$$sys:=\{\frac{\partial^2}{\partial y\,\partial x}\mathrm{f}(x,\,y)=0,\ \frac{\partial^2}{\partial x^2}\mathrm{f}(x,\,y)=0\}$$

```
> id := initialdata(sys);
```

$$id:=\mathrm{table}([\mathit{Finite}=[\frac{\partial}{\partial x}\mathrm{f}(x_0,\,y_0)=\_C1],\ \mathit{Infinite}=[\mathrm{f}(x_0,\,y)=\_\mathrm{F1}(y)]])$$

```
> rtaylor(sys, id, order=3);
```

$$[\mathrm{f}(x,\,y)=\_\mathrm{F1}(y_0)+\_C1\,(x-x_0)+(\frac{\partial}{\partial y}\_\mathrm{F1}(y_0))(y-y_0)+\frac{1}{2}(\frac{\partial^2}{\partial y^2}\_\mathrm{F1}(y_0))(y-y_0)^2$$
$$+\frac{1}{6}(\frac{\partial^3}{\partial y^3}\_\mathrm{F1}(y_0))(y-y_0)^3]$$

A.13 caseplot - provide a graphical display of binary case tree output from rifsimp

Calling Sequences

caseplot(**rifresult**)

caseplot(**rifresult**, **options**)

Parameters

rifresult - output from **rifsimp** with **casesplit** active

Description

The **caseplot** function provides a graphical display of the binary case tree resulting from a `rifsimp` calculation, or a `maxdimsystems` calculation using the **output=rif** option.

Note that **rifsimp** must have been called with the **casesplit** option to obtain multiple cases.

For the graphical display of the case structure to be meaningful, the computation must have obtained more than a single case. For more information on cases and **casesplit** options, see `rifsimp[cases]`. For information on the output format for **rifsimp**, see `rifsimp[output]`.

When the **rifsimp** algorithm splits a system into cases, depending on the complexity of the original system, the number of cases can be quite large. The **caseplot** command has been provided for quick examination of the case tree to allow for more efficient use of the results of **rifsimp**.

The case tree is plotted with the cases corresponding to **pivot** <> **0** on the left, and those corresponding to **pivot** = **0** on the right. Numbers are displayed for all cases at the leaves of the tree, corresponding to each the solution case in the input **rifsimp** solution.

When the **caseplot** function is called with the optional argument **vars**, a list of dependent variable names, initial data for the system with respect to those dependent variables is also computed, and displayed on the case tree below each case number. For more information on computation of initial data for ODE and PDE systems, please see `initialdata`.

The optional argument **pivots** tells caseplot to display information on the pivots that represent the case splittings in the displayed case structure. On the plot each case splitting will have a reference of the form **p<i>**, and a legend will be printed giving the correspondence between the pivot references (**p<i>**) and the actual pivots.

The branches of the tree are color coded, using the following conventions:

```
Black     Regular case

Red       Inconsistent case: no solution exists for this case.

Yellow    Error or timeout case: the error is typically caused by a Maple
          "object too large" error, while the timeouts are controlled

          using the ctl,stl,itl options (see rifsimp[options]).
Blue      Ignored case: use of the casesplit=[...] option told rifsimp
          not to explore this case (see rifsimp[adv_options])..
Green     Ignored case: specification of mindata=[...] was violated,

          so rifsimp ignored this case (see rifsimp[adv_options])..
```

The yellow cases are important, as they indicate an incomplete calculation. When yellow cases are present in the plot, a message is printed indicating the status of each of these cases (for example, 'time expired' or 'object too large').

Messages are also produced prior to the display of the plot when an initial data computation is specified through use of the **vars** argument. The case and computed dimension of the initial data is displayed.

Examples

```
> with(DEtools):
```

As a first example, we choose the simplest system with a case split:

```
> sys1:=[f(x)*g(x)=0];
```

$$sys1 := [\mathrm{f}(x)\,\mathrm{g}(x) = 0]$$

```
> ans1:=rifsimp(sys1,casesplit);
```

$ans1 := \mathrm{table}([1 = \mathrm{table}([\mathit{Solved} = [\mathrm{f}(x) = 0], \mathit{Pivots} = [\mathrm{g}(x) \neq 0],$
$\mathit{Case} = [[\mathrm{g}(x) \neq 0, \mathrm{f}(x)]]]), 2 = \mathrm{table}([\mathit{Solved} = [\mathrm{g}(x) = 0], \mathit{Case} = [[\mathrm{g}(x) = 0, \mathrm{f}(x)]]]),$
$\mathit{casecount} = 2$
$])$

```
> caseplot(ans1);
```

So if you were to run the command (it cannot be run in a help page), you would notice that we have two cases (cases 1 and 2), and that both give answers (the tree branches are black).

A.14 checkrank - illustrate ranking to be used for a rifsimp calculation

Calling Sequences

checkrank(**system**, **options**)

checkrank(**system**, **vars**, **options**)

Parameters

system - list or set of polynomially nonlinear PDEs or ODEs (may contain inequations)

vars - (optional) list of the dependent variables to solve for

options - (optional) sequence of options to control the behavior of **checkrank**

Description

To simplify systems of PDEs or ODEs, a ranking must be defined over all indeterminates present in the system. The ranking allows the algorithm to select an indeterminate for which to solve algebraically when looking at an equation. The **checkrank** function can be used to understand how the ranking-associated options define a ranking in **rifsimp**. For more detailed information about rankings, please see `rifsimp[ranking]`.

The **checkrank** function takes in the system as input along with the options:

```
vars                List of dependent variables (See Following)

indep=[indep vars]  List of independent variables (See Following)

ranking=[...]       Specification of exact ranking (See rifsimp[ranking])

degree=n            Use all derivatives to differential order n.
```

The output is a list that contains the derivatives in the system ordered from highest to lowest rank. If **degree** is given, all possible derivatives of all dependent variables up to the specified degree are used; otherwise, only the derivatives present in the input are used.

Default Ranking

When simplifying a system of PDEs or ODEs, you may want to eliminate higher order derivatives in favor of lower order derivatives. Do this by using a ranking by differential order, as is the default for **rifsimp**. Unfortunately, this says nothing about how ties are broken, for example, between two third order derivatives.

The breaking of ties is accomplished by first looking at the differentiations of the derivative with respect to each independent variable in turn. If they are of equal order, then the dependent variable itself is examined. The independent variable differentiations are examined in the order in which they appear in the dependency lists of the dependent variables, and the dependent variables are ordered alphabetically.

So, for example, given an input system containing **f(x,y,z),g(x,y,z),h(x,z)**, the following will hold:

```
[x,y,z]           Order of independent variables
[f,g,h]           Order of dependent variables
f[x] < g[xx]      By differential order
g[xy] < f[xxz]    By differential order
f[xy] < g[xx]     By differentiation with respect to x (x>y)
                     Note: differential order is equal
f[xzz] < g[xyz]   By differentiation with respect to y
g[xx] < f[xx]     By dependent variable
                     Note: differentiations are exactly equal
h[xz] < f[xz]     By dependent variable
```

Note that, in the above example, the only time the dependent variable comes into play is when all differentiations are equal.

Changing the Default

To change the default ranking, use the **vars**, **indep=[...]**, or **ranking=[...]** options. The **vars** can be specified in two distinct ways:

1. Simple List

If the **vars** are specified as a simple list, this option overrides the alphabetical order of the dependent variables described in the default ordering section.

2. Nested List

This option gives a solving order for the dependent variables. For example, if **vars** were specified as [[**f**],[**g**,**h**]], this would tell **rifsimp** to rank any derivative of **f** greater than all derivatives of **g** and **h**. Then, and when comparing **g** and **h**, the solving order would be differential order, then differentiations, and then dependent variable name as specified by the input [**g**,**h**]. This would help in obtaining a subset of the system that is independent of **f**; that is, a smaller PDE system in **g** and **h** only.

The **indep=[...]** option provides for the specification of the independent variables for the problem, as well as the order in which differentiations are examined. So if the option **indep=[x,y]** were used, then **f[x]** would be ranked higher than **f[y]**, but if **indep=[y,x]** were specified, then the opposite would be true.

Before using the **ranking=[...]** option, please read `rifsimp[ranking]`.

Examples

```
> with(DEtools):
```

The first example uses the default ranking for a simple system.

```
> sys:=[diff(g(x),x,x)-g(x)=0,diff(f(x),x)^3-diff(g(x),x)=0];
```

$$sys := [(\frac{\partial^2}{\partial x^2}\,\mathrm{g}(x)) - \mathrm{g}(x) = 0,\ (\frac{\partial}{\partial x}\,\mathrm{f}(x))^3 - (\frac{\partial}{\partial x}\,\mathrm{g}(x)) = 0]$$

```
> checkrank(sys);
```

$$[\frac{\partial^2}{\partial x^2}\,\mathrm{g}(x),\ \frac{\partial}{\partial x}\,\mathrm{f}(x),\ \frac{\partial}{\partial x}\,\mathrm{g}(x),\ \mathrm{g}(x)]$$

By default, the first equation would be solved for the second order derivative in **g(x)**, while the second equation would be solved for the first order derivative in f(x). Suppose instead that we always want to solve for **g(x)** before **f(x)**. We can use **vars**.

```
> checkrank(sys,[[g],[f]]);
```

$$[\frac{\partial^2}{\partial x^2}\,\mathrm{g}(x),\ \frac{\partial}{\partial x}\,\mathrm{g}(x),\ \mathrm{g}(x),\ \frac{\partial}{\partial x}\,\mathrm{f}(x)]$$

So here **g(x)** and all derivatives are ranked higher than **f(x)**.

The next example shows the default for a PDE system in **f(x,y), g(x,y), h(y)** (where we use the **degree=2** option to get all second order derivatives):

```
> checkrank([f(x,y),g(x,y),h(y)],degree=2);
```

$$[\frac{\partial^2}{\partial x^2}\mathrm{f}(x,\,y),\ \frac{\partial^2}{\partial x^2}\mathrm{g}(x,\,y),\ \frac{\partial^2}{\partial y\,\partial x}\mathrm{f}(x,\,y),\ \frac{\partial^2}{\partial y\,\partial x}\mathrm{g}(x,\,y),\ \frac{\partial^2}{\partial y^2}\mathrm{f}(x,\,y),\ \frac{\partial^2}{\partial y^2}\mathrm{g}(x,\,y),\ \frac{\partial^2}{\partial y^2}\mathrm{h}(y),\ \frac{\partial}{\partial x}\mathrm{f}(x,\,y),\ \frac{\partial}{\partial x}\mathrm{g}(x,\,y),\ \frac{\partial}{\partial y}\mathrm{f}(x,\,y),\ \frac{\partial}{\partial y}\mathrm{g}(x,\,y),\ \frac{\partial}{\partial y}\mathrm{h}(y),\ \mathrm{f}(x,\,y),\ \mathrm{g}(x,\,y),\ \mathrm{h}(y)]$$

All second order derivatives are first (first 7 entries), then the first derivatives with respect to **x** ahead of the first derivatives with respect to **y**, and finally **f(x,y)**, then **g(x,y)**, then **h(y)**.

Suppose we want to eliminate higher derivatives involving **y** before **x**. We can use **indep** for this as follows:

```
> checkrank([f(x,y),g(x,y),h(y)],indep=[y,x],degree=2);
```

$$[\frac{\partial^2}{\partial y^2}\mathrm{f}(x,\,y),\ \frac{\partial^2}{\partial y^2}\mathrm{g}(x,\,y),\ \frac{\partial^2}{\partial y^2}\mathrm{h}(y),\ \frac{\partial^2}{\partial y\,\partial x}\mathrm{f}(x,\,y),\ \frac{\partial^2}{\partial y\,\partial x}\mathrm{g}(x,\,y),\ \frac{\partial^2}{\partial x^2}\mathrm{f}(x,\,y),\ \frac{\partial^2}{\partial x^2}\mathrm{g}(x,\,y),\ \frac{\partial}{\partial y}\mathrm{f}(x,\,y),\ \frac{\partial}{\partial y}\mathrm{g}(x,\,y),\ \frac{\partial}{\partial y}\mathrm{h}(y),\ \frac{\partial}{\partial x}\mathrm{f}(x,\,y),\ \frac{\partial}{\partial x}\mathrm{g}(x,\,y),\ \mathrm{f}(x,\,y),\ \mathrm{g}(x,\,y),\ \mathrm{h}(y)]$$

Now to eliminate **f(x,y)** and derivatives in terms of **g(x,y)** and **h(y)**, and to rank **y** derivatives higher than **x**, we can combine the options to obtain the following.

```
> checkrank([f(x,y),g(x,y),h(y)],[[f],[g,h]],indep=[y,x],degree=2);
```

$$[\frac{\partial^2}{\partial y^2}\mathrm{f}(x,\,y),\ \frac{\partial^2}{\partial y\,\partial x}\mathrm{f}(x,\,y),\ \frac{\partial^2}{\partial x^2}\mathrm{f}(x,\,y),\ \frac{\partial}{\partial y}\mathrm{f}(x,\,y),\ \frac{\partial}{\partial x}\mathrm{f}(x,\,y),\ \mathrm{f}(x,\,y),\ \frac{\partial^2}{\partial y^2}\mathrm{g}(x,\,y),\ \frac{\partial^2}{\partial y^2}\mathrm{h}(y),\ \frac{\partial^2}{\partial y\,\partial x}\mathrm{g}(x,\,y),\ \frac{\partial^2}{\partial x^2}\mathrm{g}(x,\,y),\ \frac{\partial}{\partial y}\mathrm{g}(x,\,y),\ \frac{\partial}{\partial y}\mathrm{h}(y),\ \frac{\partial}{\partial x}\mathrm{g}(x,\,y),\ \mathrm{g}(x,\,y),\ \mathrm{h}(y)]$$

A.15 rifread - load intermediate computation as saved using the 'store' options of rifsimp

Calling Sequences

rifread(**filename**)

Parameters

filename - (optional) name of the file containing the partial **rifsimp** computation.

Description

The **rifread** command loads a partial **rifsimp** computation that was run using the **store** or **storeall** options (see `rifsimp[adv_options]`).

The storage options are most useful for large or complex computations, where the resources required to complete the computation may exceed the capability of the machine. The state of the system at the last iteration (using **store**), or at all previous iterations and/or cases (using **storeall**), can be retrieved using this command.

When called with no arguments, the file "RifStorage.m" is used. This is the default file name for use with the **store** option of **rifsimp**. If a file name was specified for the **rifsimp** run, or if the **storeall** option was used (storing all iterations and/or cases in separate files), the file name must be included in the **rifread** command.

Please note that the system obtained using **rifread** is not in final form and may have redundant equations or unresolved integrability conditions.

Examples

```
> with(DEtools):
```

Consider the following system, run with **rifsimp**.

```
> sys1 := [diff(xi(x,y),y,y),
> diff(eta(x,y),y,y)-2*diff(xi(x,y),x,y),
> -3*y^2*diff(xi(x,y),y)+2*diff(eta(x,y),x,y)-diff(xi(x,y),x,x),
> -2*eta(x,y)*y-2*y^2*diff(xi(x,y),x)
> +y^2*diff(eta(x,y),y)+diff(eta(x,y),x,x)];
```

$$sys1 := [\frac{\partial^2}{\partial y^2}\xi(x, y), (\frac{\partial^2}{\partial y^2}\eta(x, y)) - 2(\frac{\partial^2}{\partial y\,\partial x}\xi(x, y)),$$
$$-3y^2(\frac{\partial}{\partial y}\xi(x, y)) + 2(\frac{\partial^2}{\partial y\,\partial x}\eta(x, y)) - (\frac{\partial^2}{\partial x^2}\xi(x, y)),$$
$$-2\eta(x, y)\,y - 2y^2(\frac{\partial}{\partial x}\xi(x, y)) + y^2(\frac{\partial}{\partial y}\eta(x, y)) + (\frac{\partial^2}{\partial x^2}\eta(x, y))]$$

```
> ans1:=rifsimp(sys1,[xi,eta],store);
```

$$ans1 := \text{table}([$$
$$Solved =$$
$$[\frac{\partial}{\partial x}\xi(x, y) = -\frac{1}{2}\frac{\eta(x, y)}{y}, \frac{\partial}{\partial x}\eta(x, y) = 0, \frac{\partial}{\partial y}\xi(x, y) = 0, \frac{\partial}{\partial y}\eta(x, y) = \frac{\eta(x, y)}{y}]$$
$$])$$

Partial results can be obtained using **rifread**, even if the full computation did not succeed (notice the presence of the redundant equation **eta[yy]=0**).

```
> temp1 := rifread();
```

$$temp1 := \text{table}([$$
$$Solved = [\frac{\partial}{\partial y}\xi(x, y) = 0, \frac{\partial^2}{\partial y^2}\eta(x, y) = 0, \frac{\partial}{\partial x}\eta(x, y) = 0, \frac{\partial}{\partial x}\xi(x, y) = -\frac{1}{2}\frac{\eta(x, y)}{y},$$
$$\frac{\partial}{\partial y}\eta(x, y) = \frac{\eta(x, y)}{y}]$$
$$])$$

Note that the results obtained above were stored in the file "RifStorage.m".

The same example can have results stored under a different name. Here the temp results are stored in "tmpstore.m":

```
> ans1:=rifsimp(sys1,[xi,eta],store="tmpstore");
```

$$ans1 := \text{table}([$$
$$Solved =$$
$$[\frac{\partial}{\partial x}\xi(x, y) = -\frac{1}{2}\frac{\eta(x, y)}{y}, \frac{\partial}{\partial x}\eta(x, y) = 0, \frac{\partial}{\partial y}\xi(x, y) = 0, \frac{\partial}{\partial y}\eta(x, y) = \frac{\eta(x, y)}{y}]$$
$$])$$

```
> temp1 := rifread("tmpstore");
```

$temp1$:= table([

$$Solved = [\frac{\partial}{\partial y}\xi(x, y) = 0, \frac{\partial^2}{\partial y^2}\eta(x, y) = 0, \frac{\partial}{\partial x}\eta(x, y) = 0, \frac{\partial}{\partial x}\xi(x, y) = -\frac{1}{2}\frac{\eta(x, y)}{y},$$

$$\frac{\partial}{\partial y}\eta(x, y) = \frac{\eta(x, y)}{y}]$$

])

It is also possible to store the results at the end of each iteration and for each separate case using **storeall**. Increasing the **infolevel** will display the file name as each partial calculation is saved.

```
> sys2 := [-diff(xi(x,y),y)*y-diff(xi(x,y),y,y)*y^2,
> eta(x,y)-2*diff(xi(x,y),x,y)*y^2+diff(eta(x,y),y,y)*y^2
> -diff(eta(x,y),y)*y-2*diff(xi(x,y),y)*y
> +2*diff(xi(x,y),y)*a*y^2,
> -2*y*diff(eta(x,y),x)-diff(xi(x,y),x,x)*y^2
> +2*diff(eta(x,y),x,y)*y^2+eta(x,y)-diff(xi(x,y),x)*y
> +diff(xi(x,y),x)*a*y^2+3*y^3*diff(xi(x,y),y)*a^2
> -3*y^4*diff(xi(x,y),y)*b^2-3*y^5*diff(xi(x,y),y)*b^2
> +3*y^2*diff(xi(x,y),y)*a^2,
> diff(eta(x,y),x,x)*y^2+y^2*diff(eta(x,y),x)*a
> -y*diff(eta(x,y),x)+2*y^3*diff(xi(x,y),x)*a^2
> -2*y^4*diff(xi(x,y),x)*b^2-y^3*diff(eta(x,y),y)*a^2
> +y^4*diff(eta(x,y),y)*b^2+eta(x,y)*y^2*a^2
> -2*eta(x,y)*y^3*b^2-y^2*diff(eta(x,y),y)*a^2
> -3*eta(x,y)*y^4*b^2-2*y^5*diff(xi(x,y),x)*b^2
> +2*y^2*diff(xi(x,y),x)*a^2+y^5*diff(eta(x,y),y)*b^2];
```

$$sys2 := [-(\frac{\partial}{\partial y}\,\xi(x,\,y))\,y - (\frac{\partial^2}{\partial y^2}\,\xi(x,\,y))\,y^2, \eta(x,\,y) - 2\,(\frac{\partial^2}{\partial y\,\partial x}\,\xi(x,\,y))\,y^2 + (\frac{\partial^2}{\partial y^2}\,\eta(x,\,y))\,y^2$$
$$-\,(\frac{\partial}{\partial y}\,\eta(x,\,y))\,y - 2\,(\frac{\partial}{\partial y}\,\xi(x,\,y))\,y + 2\,(\frac{\partial}{\partial y}\,\xi(x,\,y))\,a\,y^2, -2\,y\,(\frac{\partial}{\partial x}\,\eta(x,\,y))$$
$$-\,(\frac{\partial^2}{\partial x^2}\,\xi(x,\,y))\,y^2 + 2\,(\frac{\partial^2}{\partial y\,\partial x}\,\eta(x,\,y))\,y^2 + \eta(x,\,y) - (\frac{\partial}{\partial x}\,\xi(x,\,y))\,y + (\frac{\partial}{\partial x}\,\xi(x,\,y))\,a\,y^2$$
$$+\,3\,y^3\,(\frac{\partial}{\partial y}\,\xi(x,\,y))\,a^2 - 3\,y^4\,(\frac{\partial}{\partial y}\,\xi(x,\,y))\,b^2 - 3\,y^5\,(\frac{\partial}{\partial y}\,\xi(x,\,y))\,b^2$$
$$+\,3\,y^2\,(\frac{\partial}{\partial y}\,\xi(x,\,y))\,a^2, (\frac{\partial^2}{\partial x^2}\,\eta(x,\,y))\,y^2 + y^2\,(\frac{\partial}{\partial x}\,\eta(x,\,y))\,a - y\,(\frac{\partial}{\partial x}\,\eta(x,\,y))$$
$$+\,2\,y^3\,(\frac{\partial}{\partial x}\,\xi(x,\,y))\,a^2 - 2\,y^4\,(\frac{\partial}{\partial x}\,\xi(x,\,y))\,b^2 - y^3\,(\frac{\partial}{\partial y}\,\eta(x,\,y))\,a^2 + y^4\,(\frac{\partial}{\partial y}\,\eta(x,\,y))\,b^2$$
$$+\,\eta(x,\,y)\,y^2\,a^2 - 2\,\eta(x,\,y)\,y^3\,b^2 - y^2\,(\frac{\partial}{\partial y}\,\eta(x,\,y))\,a^2 - 3\,\eta(x,\,y)\,y^4\,b^2$$
$$-\,2\,y^5\,(\frac{\partial}{\partial x}\,\xi(x,\,y))\,b^2 + 2\,y^2\,(\frac{\partial}{\partial x}\,\xi(x,\,y))\,a^2 + y^5\,(\frac{\partial}{\partial y}\,\eta(x,\,y))\,b^2]$$

```
> infolevel[rifsimp]:=1;
```

$$infolevel_{rifsimp} := 1$$

```
> rr:=rifsimp(sys2,[xi,eta],storeall,casesplit);
```

```
Warning, the following denominator was added to ineqns:
```

$$1$$

The system has been identified as follows :

The system has the following dependent variables :

$$\xi,\, \eta$$

The system has the following independent variables :

$$x,\, y$$

The system has the following constants :

$$a,\, b$$

The following are to be treated as solve variables :

$$\xi,\, \eta,\, a,\, b$$

```
Storing current system in RifStorage_1_1.m
Storing current system in RifStorage_1_2.m
Storing current system in RifStorage_1_3.m
Storing current system in RifStorage_1_4.m
Storing current system in RifStorage_1_5.m
Storing current system in RifStorage_1_6.m
Storing current system in RifStorage_1_7.m
Storing current system in RifStorage_1_8.m
Storing current system in RifStorage_1_9.m
Storing current system in RifStorage_1_10.m
```

```
Storing current system in RifStorage_1_11.m
Storing current system in RifStorage_1_12.m
Storing current system in RifStorage_1_13.m
Storing current system in RifStorage_1_14.m
Storing current system in RifStorage_2_13.m
Storing current system in RifStorage_2_14.m
Storing current system in RifStorage_2_15.m
Storing current system in RifStorage_3_14.m
Storing current system in RifStorage_3_15.m
Storing current system in RifStorage_3_16.m
Storing current system in RifStorage_4_15.m
```

$$
\begin{aligned}
&rr := \mathrm{table}([1 = \mathrm{table}([\mathit{Solved} = [\tfrac{\partial}{\partial x}\,\xi(x,\,y) = 0,\ \tfrac{\partial}{\partial y}\,\xi(x,\,y) = 0,\ \eta(x,\,y) = 0],\\
&\mathit{Case} = [[b \neq 0,\ \eta(x,\,y)]],\\
&\mathit{Pivots} = [b \neq 0]\\
&]), 2 = \mathrm{table}([\mathit{Solved} = [\tfrac{\partial}{\partial x}\,\xi(x,\,y) = 0,\ \tfrac{\partial}{\partial y}\,\xi(x,\,y) = 0,\ \eta(x,\,y) = 0,\ b = 0],\\
&\mathit{Case} = [[b = 0,\ \eta(x,\,y)],\ [a\,(a-1) \neq 0,\ \eta(x,\,y)]],\\
&\mathit{Pivots} = [a \neq 0,\ a - 1 \neq 0]\\
&]), 3 = \mathrm{table}([\mathit{Solved} = [\tfrac{\partial}{\partial x}\,\xi(x,\,y) = 0,\ \tfrac{\partial}{\partial y}\,\xi(x,\,y) = 0,\ \eta(x,\,y) = 0,\ a = 1,\ b = 0],\\
&\mathit{Case} = [[b = 0,\ \eta(x,\,y)],\ [a\,(a-1) = 0,\ \eta(x,\,y)],\ [a \neq 0,\ \eta(x,\,y)]]\\
&]), 4 = \mathrm{table}([\mathit{Solved} = [\\
&\tfrac{\partial}{\partial x}\,\xi(x,\,y) = \frac{\eta(x,\,y)}{y},\ \tfrac{\partial}{\partial x}\,\eta(x,\,y) = 0,\ \tfrac{\partial}{\partial y}\,\xi(x,\,y) = 0,\ \tfrac{\partial}{\partial y}\,\eta(x,\,y) = \frac{\eta(x,\,y)}{y},\ a = 0,\ b = 0],\\
&\mathit{Case} = [[b = 0,\ \eta(x,\,y)],\ [a\,(a-1) = 0,\ \eta(x,\,y)],\ [a = 0,\ \eta(x,\,y)]]\,]),\\
&\mathit{casecount} = 4])
\end{aligned}
$$

```
> rifread("RifStorage_2_13");
```

$$
\begin{aligned}
&\mathrm{table}([\mathit{Solved} = [\tfrac{\partial}{\partial y}\,\xi(x,\,y) = 0,\ b = 0,\ \tfrac{\partial}{\partial x}\,\xi(x,\,y) = \frac{\eta(x,\,y)}{y},\\
&\tfrac{\partial}{\partial x}\,\eta(x,\,y) = a\,(3\,a - 2)\,\eta(x,\,y), \tfrac{\partial}{\partial y}\,\eta(x,\,y) = \frac{\eta(x,\,y)}{y}],\\
&\mathit{DiffConstraint} = [0 = \eta(x,\,y)\,a\,(a-1),\ 0 = \eta(x,\,y)\,a\,(a-1),\ 0 = \eta(x,\,y)\,a\,(a-1),\\
&0 = \eta(x,\,y)\,a\,(9\,a^3\,y + 4\,y\,a - 9\,y\,a^2 - 2\,a + 2)]\\
&])
\end{aligned}
$$

A description of the meaning of each entry appearing in the output tables can be found on the `rifsimp[output]` page.

Bibliography

[1] A. Aho, J. Hopcroft, and J. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983.

[2] T. Becker, V. Weispfenning, and H. Kredel, *Gröbner Bases: A Computational Approach to Commutative Algebra*, vol. 141 of Graduate Texts in Mathematics, Springer, 1993.

[3] G. W. Bluman and S. Kumei, *Symmetries and Differential equations*, vol. 81 of Applied Mathematical Sciences, Springer, 1989.

[4] A. Bocharov and M. Bronstein, *Efficiently implementing two methods of the geometrical theory of differential equations*, Acta. Appl. Math., 16 (1989), pp. 143–166.

[5] F. Boulier, *Some improvements of a lemma of Rosenfeld.* Preprint, 1998.

[6] F. Boulier, D. Lazard, F. Ollivier, and M. Petitot, *Representation for the radical of a finitely generated differential ideal*, in Proc. ISSAC 1995, ACM Press, 1995, pp. 158–166.

[7] F. Boulier, F. Lemaire, and M. Maza, *Pardi!*, in Proc. ISSAC 2001, ACM Press, 2001, pp. 38–47.

[8] W. Brown, *On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors*, J. of the ACM, 18 (1971), pp. 476–504.

[9] B. Buchberger, *An Algorithm for Finding a Basis for the Residue Class Ring of a Zero-Dimensional Polynomial Ideal*, PhD thesis, Univ. of Innsbruck, Math. Inst., 1965.

[10] ——, *Gröbner Bases: An algorithmic method in polynomial ideal theory*, in Multidimensional Systems Theory, D. Reidel Publ. Company, 1985, pp. 184–232.

[11] G. Carrà Ferro, *Gröbner bases and differential ideals*, in Proc. AAECC5, Menorca, Spain, Springer, 1987, pp. 129–140.

[12] G. Carrà Ferro and W. Sit, *On term-orderings and rankings*, in Computational Algebra, vol. 151 of Lecture Notes in Pure and Applied Mathematics, Marcel Dekker, 1994, pp. 31–77.

[13] E. Cartan, *Sur la structure des groupes infinis de transformations*, in Oeuvres Complètes Part II, vol. 2, Gauthier-Villars, 1904, pp. 571–714.

[14] Y. Chen and X. Gao, *Involutive characteristic sets of algebraic partial differential equation systems*, Science in China: Series A, 46 (2003), pp. 469–487.

[15] S. C. Chou, *Mechanical Geometry Theorem Proving*, Reidel, 1988.

[16] C. Collins, *Complex Potential Equations I*, Math. Proc. Cambridge Philos. Soc., 80 (1976), pp. 165–187.

[17] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, MIT Press, 1990.

[18] D. Cox, J. Little, and D. O'Shea, *Ideals, Varieties, and Algorithms*, Undergraduate Texts in Mathematics, Springer, 1992.

[19] K. Geddes, S. Czapor, and G. Labahn, *Algorithms for Computer Algebra*, Kluwer, 1992.

[20] V. Gerdt, *Gröbner bases and involutive methods for algebraic and differential equations*, in Comp. Alg. in Sci. and Eng., J. Fleischer, J. Grabmeier, F. Hehl, and W. Küchlin, eds., World Sci., 1995, pp. 117–137.

[21] V. Gerdt and A. Zharkov, *Computer generation of necessary integrability conditions for polynomial-nonlinear evolution systems*, in Proc. ISSAC 1990, ACM Press, 1990, pp. 250–254.

[22] H. Goldschmidt, *Integrability criteria for systems of partial differential equations*, J. Diff. Geom., 1 (1967), pp. 269–307.

[23] D. Hartley and R. Tucker, *A constructive implementation of the Cartan–Kähler theory of exterior differential systems*, J. Symbolic Comp., 12 (1991), pp. 655–667.

[24] W. Hereman, *Review of symbolic software for the computation of Lie symmetries of differential equations*, Euromath Bull., 1 (1994), pp. 45–79.

[25] M. Hickman, *Symmetry: A maple package for jet bundle computations and lie symmetry computations.* Available at `http://www.math.canterbury.ac.nz/~mathmsh/Symmetry.html`.

[26] L. Hsu and N. Kamran, *Classification of second-order ordinary differential equations admitting Lie groups of fibre-preserving symmetries*, Proc. Lond. Math. Soc., 58 (1989), pp. 387–416.

[27] E. Hubert, *Factorization free decomposition algorithms in differential algebra*, J. Symbolic Comp., 29 (2000), pp. 641–662.

[28] ——, *Notes on triangular sets and triangulation-decomposition algorithms I: Polynomial systems*, in Symbolic and Numerical Scientific Computing, vol. 2630 of Lecture Notes in Computer Science, Springer Verlag, 2003, pp. 1–39.

[29] ——, *Notes on triangular sets and triangulation-decomposition algorithms II: Differential systems*, in Symbolic and Numerical Scientific Computing, vol. 2630 of Lecture Notes in Computer Science, Springer Verlag, 2003, pp. 40–87.

[30] N. Ibragimov, ed., *CRC Handbook of Lie Group Analysis of Differential Equations*, vol. I,II,III, CRC Press, Boca Raton, 1994,1995,1996.

[31] M. Janet, *Sur les Systèmes d'équations aux dérivées partielles*, J. de Math. (8), 3 (1920), pp. 65–151.

[32] E. Kaltofen and W. Lee, *Early termination in sparse interpolation algorithms*, J. Symbolic Comp., 36 (2003), pp. 365–400.

[33] E. Kaltofen and B. Trager, *Computing with polynomials given by black boxes for their evaluations: Greatest common divisors, factorization, separation of numerators and denominators*, J. Symbolic Comp., 9 (1990), pp. 301–320.

[34] D. Knuth, *The Art of Computer Programming: Fundamental Algorithms*, vol. 1, Addison Wesley Longman, 3rd ed., 1997.

[35] ——, *The Art of Computer Programming: Seminumerical Algorithms*, vol. 2, Addison Wesley Longman, 3rd ed., 1997.

[36] E. Kolchin, *Differential Algebra and Algebraic Groups*, Academic Press, 1973.

[37] E. Mansfield, *Differential Gröbner Bases*, PhD thesis, Univ. of Sydney, 1991.

[38] ——, *The differential algebra package diffgrob2*, Maple Tech., 3 (1998), pp. 33–37.

[39] ——, *Algorithms for symmetric differential systems.* Technical Report UKC/IMS/99/35, University of Kent at Canterbury, 1999.

[40] E. Mansfield, P. Clarkson, and G. Reid, *Nonclassical reductions of a coupled nonlinear Schrödinger system*, Comp. Phys. Comm., 115 (1998), pp. 460–488.

[41] A. Miola and D. D. Y. Yun, *Computational aspects of Hensel-type univariate polynomial Greatest Common Divisor algorithms*, in Proc. EUROSAM '74, ACM Press, 1974, pp. 46–54.

[42] B. Mishra, *Algorithmic Algebra*, Texts and Monographs in Computer Science, Springer-Verlag, 1993.

[43] M. Monagan and A. Wittkopf, *On the design and implementation of Brown's algorithm over the integers and number fields*, in Proc. ISSAC '00, St. Andrews, ACM Press, 2000, pp. 215–223.

[44] J. Moses and D. Yun, *The EZ-GCD Algorithm*, in Proc. ACM '73, ACM Press, 1973, pp. 159–166.

[45] J. D. Murray, *Mathematical Biology*, Springer-Verlag, 1989.

[46] P. Olver, *Applications of Lie Groups to Differential Equations*, vol. 107 of Graduate Texts in Mathematics, Springer, 2nd ed., 1993.

[47] ——, *Equivalence, Invariants, and Symmetry*, Graduate Texts in Mathematics, Cambridge University Press, 1995.

[48] J.-F. Pommaret, *Systems of Partial Differential Equations and Lie Pseudogroups*, Gordon and Breach, 1978.

[49] ——, *Partial Differential Equations and Group Theory: New Perspectives for Applications*, Kluwer, 1994.

[50] G. Reid, *Algorithms for reducing a system of PDEs to standard form, determining the dimension of its solution space and calculating its Taylor series solution*, European J. of Appl. Math., 2 (1991), pp. 293–318.

[51] G. Reid, P. Lin, and A. Wittkopf, *Differential elimination-completion algorithms for dae and pdae*, Studies in Appl. Maths, 106 (2001), pp. 1–45.

[52] G. Reid, I. Lisle, A. Boulton, and A. Wittkopf, *Algorithmic determination of commutation relations for Lie symmetry algebras of PDEs*, in Proc. ISSAC '92, Berkeley, P. Wang, ed., ACM Press, 1992, pp. 63–68.

[53] G. Reid and A. Wittkopf, *Determination of maximal symmetry groups of classes of differential equations*, in Proc. ISSAC '2000, St. Andrews, ACM Press, 2000, pp. 264–272.

[54] G. Reid, A. Wittkopf, and A. Boulton, *Reduction of systems of nonlinear partial differential equations to simplified involutive forms*, European J. of Appl. Math., 7 (1996), pp. 635–666.

[55] C. Riquier, *Les systèmes d'équations aux dérivées partielles*, Gauthier-Villars, 1910.

[56] J. Ritt, *Differential Algebra*, vol. 33 of Colloquium Publications, AMS, 1950.

[57] A. Rosenfeld, *Specializations in differential algebra*, AMS Trans., 90 (1959), pp. 394–407.

[58] C. Rust, *Rankings of Derivatives for Elimination Algorithms and Formal Solvability of Analytic Partial Differential Equations*, PhD thesis, Univ. Chicago, 1998. www-address: www.cecm.sfu.ca/˜rust.

[59] C. Rust and G. Reid, *Rankings of partial derivatives*, in Proc. ISSAC '97, Maui, W. Küchlin, ed., ACM Press, 1997, pp. 9–16.

[60] C. Rust, G. Reid, and A. Wittkopf, *Existence and uniqueness theorems for formal power series solutions of analytic differential systems*, in Proc. ISSAC '99, Vancouver, S. Dooley, ed., ACM Press, 1999, pp. 105–112.

[61] J. Schü, W. Seiler, and J. Calmet, *Algorithmic Methods for Lie Pseudogroups*, in Proc. Modern Group Analysis, N. Ibragimov, M. Torrisi, and A. Valenti, eds., Kluwer, 1993, pp. 337–344.

[62] F. Schwarz, *Reduction and completion algorithms for partial differential equations*, in Proc. ISSAC '92, Berkeley, P. Wang, ed., ACM Press, 1992, pp. 49–56.

[63] ——, *Janet Bases of 2nd Order Differential Equations*, in Proc. ISSAC '96, Zurich, Y. N. Lakshman, ed., ACM Press, 1996, pp. 179–188.

[64] W. Seiler, *Analysis and Application of the Formal Theory of Partial Differential Equations*, PhD thesis, Lancaster University, 1994.

[65] ——, *Applying AXIOM to Partial Differential Equations.* Technical Report #17, Universität Karlsruhe: Institut für Algorithmen und Kognitive Systeme, 1995.

[66] C. Sulem and P. L. Sulem, *The Nonlinear Schrödinger Equation, Self-Focussing and Wave Collapse*, vol. 139 of Applied Mathematical Sciences, Springer, 1999.

[67] V. Talanov, *Focusing of light in cubic media*, JETP Lett, 11 (1970), pp. 199–201.

[68] A. Tresse, *Sur les invariants différentiels des groupes continus de transformations*, Acta Math., 18 (1894), pp. 1–88.

[69] E. Varley and B. Seymour, *Exact solutions for large amplitude waves in dispersive and dissipative systems*, Studies in Applied Math., 72 (1985), pp. 241–262.

[70] D. Wang, *Elimination Methods*, Springer-Verlag, 2001.

[71] P. S. Wang, *The EEZ-GCD algorithm*, ACM SIGSAM Bull., 14 (1980), pp. 50–60.

[72] A. Wittkopf and G. Reid, *Benchmark problems for differential elimination algorithms.* Available at `http://www.cecm.sfu.ca/~wittkopf/systems.html`, 2000.

[73] ——, *Fast differential elimination algorithms.* Technical Report TR-00-06, Ontario Research Centre for Computer Algebra, 2000.

[74] ——, *Fast differential elimination in C: The CDiffElim environment*, Comp. Phys. Comm., 139 (2001), pp. 192–217.

[75] T. Wolf, *A package for the analytic investigation and exact solutions of differential equations*, in Proc. EUROCAL '87, vol. 378, Springer, 1987, pp. 479–491.

[76] W. Wu, *Basic principles of mechanical theorem proving in elementary geometries*, J. Syst. Sci. Math. Sci., 4 (1984), pp. 207–235.

[77] ——, *Mechanical Theorem Proving In Geometries: Basic Principles*, Springer-Verlag, 1994. translated by Xiaofan Jin And Dongming Wang.

[78] R. Zippel, *Probabilistic Algorithms for Sparse Polynomials*, PhD thesis, Massachusetts Inst. of Technology, 1979.

[79] ——, *Interpolating polynomials from their values*, J. Symbolic Comp., 9 (1990), pp. 375–403.

[80] ——, *Effective Polynomial Computation*, Kluwer, 1993.

[81] W. Zulehner and W. F. Ames, *Group Analysis of a semilinear vector diffusion equation*, Nonlinear Analysis Theory Methods and Applications, 7 (1983), pp. 945–969.