DIMSYM
Symmetry determination and
linear differential equation package

James Sherring[1]
Edited by **Geoff Prince**

Version 2.2 (c) January 1996

Contents

Chapter 1

Introduction

Dimsym is primarily a program for determining the Lie point symmetries of a given differential equation, although it is capable of doing much more. It is able to find different sorts of symmetries, isolating special cases, and it incorporates a powerful algorithm for solving systems of linear equations, which can be used without the symmetry structure of the rest of the program.

The user supplies a differential equation or system of equations, Dimsym creates the system of determining equations for the symmetries. The given differential equations may include arbitrary unknown functions or constants, and may also use any user defined special functions with the full power of REDUCE's simplification rules available. Dimsym will attempt to solve the determining equations as far as possible, and then return any unsolved equations in a 'simplified' form. For most equations arising from physical models, Dimsym will solve all of the determining equations, except "constraint" equations whose general solutions cannot be written explicitly in closed form. For the remaining cases, the user must assist in the solution of equations that Dimsym cannot solve.

Given that you want to find symmetries of your differential equation, it is assumed that you know only the basics of REDUCE, such as how to run REDUCE on your machine, and how to compile and load Dimsym (which may vary between different implementations). More advanced usage of Dimsym such as performing manual manipulations of the determining equations if needed will require greater familiarity with REDUCE, while the user who wants to push the program to its limits may need to be familiar with the algorithms involved.

To begin, all you need to do is read section 2 and follow the example through yourself: you will then be set to experiment, more examples are the subject of section 17. I expect that you will be playing with some of the data files supplied with Dimsym and then reading the manual as you feel the need for more precise information, so this is often written with a lot of details for the serious user. Read section 3 on terminology and section 4 on how to best enter you equations and you will be ready to use the program seriously. Read the other sections as you feel the need.

Chapter 2

An example

As an example of what Dimsym does, this is how Dimsym finds the symmetries for the heat equation $u_{xx} = u_t$.

x(1) represents the spatial coordinate x, x(2) represents time t, u(1) represents the heat u, u(1,1,1) represents $u_{xx} = \frac{\partial^2 u(1)}{\partial x(1) \partial x(1)}$, and u(1,2) represents $u_t = \frac{\partial u(1)}{\partial x(2)}$. Input from the user begins on a line preceded with the line number, eg 1:, and is in boldface.

```
REDUCE 3.5, 15-Jul-91
1:    load dimsym;
Dimsym 2.0, 1-Oct-93
Symmetry determination and linear D.E. package
(c) 1992, 1993 James Sherring    J.Sherringlatrobe.edu.au
License #0R0000
Any publication resulting from these calculations must reference
this program.
2:    loaddeq( u(1,1,1) = u(1,2) );
1
3:    mkdets(point);
4:    solvedets(std);
The equation-group numbers of equations remaining are (25)
5:    showdets();
There are 1 determining equations remaining, which are...
deteqn(1)=DF(C(1),X(2)) - DF(C(1),X(1),2)
The remaining dependencies are ...
(C 1) depends on ((X 1) (X 2))
6:    mkgens();
There are 7 symmetries found.
The generators of the finite algebra are:
                                        2                  2
Gen (1) =  - 2*@    *U(1)*X(2) - @    *U(1)*X(1)  + 4*@    *X(2)
                U(1)               U(1)               X(2)

           + 4*@    *X(2)*X(1)
               X(1)

Gen (2) = 2*@    *X(2) + @    *X(1)
            X(2)         X(1)
```

```
Gen (3) =  -  @     *U(1)*X(1) + 2*@    *X(2)
                 U(1)                     X(1)
Gen (4) = @
            X(1)
Gen (5) = @     *U(1)
            U(1)
Gen (6) = @
            X(2)
The generators for the remaining equations are:
(The unknowns in these generators satisfy the remaining
determining equations.)
Gen (7) = @    *C(1)
            U(1)
(C 1) depends on ((X 1) (X 2))
```

So the things that we did were:

load dimsym; to load the Dimsym program. This is implementation dependent.

loaddeq(u(1,2,2)=u(1,1)); to load the original differential equation,

mkdets(point); to make the determining equations for point symmetries,

solvedets(std); to solve the equations as far as possible using the std (standard) algorithm,

showdets(); to show the determining equations remaining,

mkgens(); to make and show the symmetries.

What Dimsym showed us was:

solvedets(std) left it with one equation remaining, this was for c(1) and was just the heat equation back again (a general property of linear equations). It showed us the 6 generators of the finite algebra, and the one generator of the infinite algebra involving an arbitrary solution c(1) to the heat equation. The symbol $_X$ represents $\frac{\partial}{\partial X}$ which is used here to represent the basis vectorfield in the direction of the X coordinate. Its usage here is borrowed from EXCALC.

This example is available as the data file `heat1` supplied with Dimsym. If you are in the right directory (or know how to specify the path) then this file may be loaded in this way:

REDUCE 3.5, 15-Jul-91 1: **in heat1;**

See section 17 for more example files.

Chapter 3

Some terminology

The calculations involved in finding symmetries involve two different systems of differential equations– the first system (often just a single equation) is the system of differential equations for which we want to find the symmetries, these are typically nonlinear. We call these equations the original differential equations. The second system of equations are the ones whose solution gives us the coefficients of the symmetry vectors. These are called the determining equations, and are always linear and homogeneous in the unknown functions.

Dimsym uses `x(j)` for the independent variables x_j and `u(i)` for the dependent variables u^i in the original differential equations. Derivatives are expressed as `u()` variables, with each index after the first representing a derivative by the corresponding `x()` variable, so we have the representation

$$\texttt{u(i,j1,...,jn)} := \frac{\partial^J u^i}{\partial x_{j1} \cdots \partial x_{jn}} = u^i_{x_{j1}\ldots x_{jn}}$$

There is a very important conceptual difference between the original differential equations and the determining equations: in the former case the variables u^i are considered to be dependent variables– functions of the x_j's, and the $u^i_{x_{j1}\ldots x_{jn}}$ are the corresponding derivatives. In the later case, each x_j, u^i and $u^i_{x_{j1}\ldots x_{jn}}$ is considered to be an **independent** variable! The unknown functions in the determining equations are functions of these independent variables, although for the usual case of point symmetries the unknown functions will only depend on the x_i and u^i variables.

There are also several different types of unknown functions we will want to talk about– ones that appear linearly in the determining equations are called determinable unknowns (dunkns) because we expect to determine them. If free unknown functions are used in the original equations, so that we are actually considering a class of differential equations (and possibly a classification problem), then we refer to these as free unknowns (funkns). These must be declared using the command `freeunknown`. These will then also appear in the determining equations, but we don't call them dunkns. See section 9 for more details on funkns.

It is important to have a representation convention for mixed derivatives, for example `u(1,2,1)` and `u(1,1,2)` both represent the same derivative (in the equivalence on smooth jet manifolds). Dimsym uses the convention that all higher-numbered derivatives should be listed first, and so we would write `u(1,2,1)`. Dimsym will change these derivatives if needed if they appear in the original equations or in dependency declarations but only if they appear before `mkdets()` is called. So beware!

9

Chapter 4

`loaddeq`–Specifying the original equations

The original differential equations are handed to Dimsym as arguments to the function `loaddeq`. Each separate differential equation must be given by a separate call of `loaddeq`. Each equation must be in an explicitly solved form, given as *leading derivative = an expression*. The derivatives must all appear using the index notation, explicit usage of the REDUCE differentiation operator `df` is not permitted for `u(i)` variables (this is in anticipation of the needs of the determining equations, we are leaving the `u(i)` variables as independent and so without any statement like `depend u(1),x(1)`, REDUCE would simplify `df(u(1),x(1))` to 0).

The derivative on the left hand side of an equation (the leading derivative) given to `loaddeq` must never appear on the right hand side of the same equation. It is also expected that neither a leading derivative nor any derivative which is a formal derivative of a leading derivative will appear on the right hand side of any other equation given to `loaddeq`, although this requirement can be relaxed provided that the resulting substitutions do not lead to an infinite loop. So Dimsym can only handle equations which can be written in this normal form, which is always valid locally, and only very obscure examples can't be done by making some assumption, such as when there are square-roots involved we restrict to some half-space, and if we are thorough we would check that we got the same answer on the half space with the negative square-roots.

When `mkdets` is called, Dimsym puts the equations given by `loaddeq` into a standard form where each equation (and differential consequence) is substituted into any of the others if the leading derivatives appear in any of the expressions for the RHS's of the given equations. Dimsym will form the differential consequence of any equation if it is needed for substitution into the other equations. Dimsym may give a warning that integrability conditions exist for the equations given to `loaddeq` if there is more than one way of reaching some differential consequence.

For systems of equations where genuine integrability conditions exist which are not differential consequences of given equations then these integrability conditions should also be specified as equations to `loaddeq` as they are required for substitutions. See datafile `sub1` for an example which only gives 7 generators if an integrability condition is omitted, but 10 generators if the integrability condition is included. Dimsym does not always report the possibility that these extra equations exist, even for the rather obvious example `sub1`. Dimsym may also give some further comments about the choice of leading derivatives if they are

not the appropriate ones to place the equations in Standard Form as used by Reid [Rei91]. These comments may be ignored if you are confident there are no integrability conditions and substitutions will not result in a substitution loop.

The importance of the way the determining equations are specified is two-fold. Firstly, it is important that the form of the equations follow the convention which Dimsym expects so that the equations may be substituted back into the determining equations. This is because the infinitesimal symmetry condition need only hold on solution surfaces of the jet space, not the whole jet space.

Secondly, the inclusion of all integrability conditions necessary to reach Standard Form is part of the local solvability condition [Olv86]. This second condition is often overlooked yet is a critical condition for many examples. Note that the Riquier-Janet theory [Jan20], [Rei90a], [Rei90b], [Rei91], [Sch84] is more general for our purposes than the Cauchy-Kovalevskaya theory discussed in [Olv86]. It also implies that a system in Standard Form satisfies the local solvability condition, as well as the maximal rank condition, so that a system in Standard Form is (totally?) nondegenerate, in the sense of [Olv86].

Chapter 5

`mkdets`–Constructing the determining equations

Constructing the determining equations for point symmetries is usually a straight forward matter for the user, performed by calling `mkdets(point)`.

For the more advanced user there is the option available of looking for other types of symmetries by calling `mkdets(`*type*`)`. The current types supported are `point, liebacklund, liebacklund(n), custom1` and `custom2`.

This feature is useful for when the algebra of point symmetries is too large (or too hard to calculate), such as when looking for point symmetries of a first order system of O.D.E.s for which the symmetry algebra is always infinite dimensional and depends on the solution of a system of linear D.E.s. Such systems may grind to a halt when placing them in standard form, and may be too general anyway. In such a case we would probably look for symmetries which were polynomial in at least one of the variables. On the other hand, if we have a system for which we do not get enough interesting point symmetries then we may loosen our restrictions and look for higher order symmetries.

`mkdets` constructs a vectorfield called `symvec` which can be accessed as an algebraic variable. To do this, it first examines the given original differential equations to determine values for `!*p`, the number of independent variables x_j in the equations, `!*q`, the number of dependent variables u^i in the equations, and `!*r`, the (highest) order of the differential equations. These three values are also available as algebraic variables. `symvec` is then (except for type custom2) given by

$$\texttt{symvec} = \sum_{j=1}^{\texttt{!*p}} \xi^j \frac{\partial}{\partial x_j} + \sum_{i=1}^{\texttt{!*q}} \phi_i \frac{\partial}{\partial u^i}.$$

$\xi^j = $ `xi(j)` and $\phi^i = $ `phi(i)` are available for access in algebraic mode, with dependencies set according to the type of symmetry requested by the call of `mkdets`.

The prolongation of `symvec` is available as `prosymvec` in algebraic mode, and (except for type custom2) is set to

$$\texttt{prosymvec} = \texttt{symvec} + \sum_{i=1}^{\texttt{!*q}} \sum_{|J|=1}^{\texttt{!*r}} \phi_i^J \frac{\partial}{\partial u_J^i}.$$

J represents a multi-index $J = (j1, j2, \ldots, jn)$ with $|J|$ representing the number of indicies, and the summation is over all multi-indicies of given lengths. $\phi_i^J = $ `phi(i,j1,j2,...,jn)`

is also available for access in algebraic mode, although the expansion will only have been done by `solvedets()` if this prolongation coefficient has been used and op!*proxpd has been applied.

A call of `mkdets(liebacklund)` or `mkdets(liebacklund(n))` will construct the determining equations for first order or nth order Lie-Backlünd symmetries respectively (so that $\xi^j = 0$ and ϕ^i depends on all variables and derivatives up to n-th order, except for variables which are leading derivatives in the given equations). The choice of leading derivatives can thus effect the number of symmetries found, it is usual practice to choose a time derivative as the leading derivative so that no time derivatives appear in the dependencies of the dunkns. See the data files `bluburgers` and `bluburgers1` for example and comparison.

A call of `mkdets(custom1)` uses current values or dependencies for `xi(j)` and `phi(i)`, which must be set in advance by the user (or else they are assumed constant) and then `symvec` and `prosymvec` are calculated accordingly (using any preset values for higher order `phi(i)`s). Type `custom2` differs from `custom1` in that it expects `symvec` and `prosymvec` to be set rather than `xi(j)` and `phi(i)`, most importantly it does not calculate the prolongations. `mkdets(custom1)` may be more appropriate for finding liebacklund symmetries depending only on some derivatives of a given order, or other generalised symmetries. If `xi(j)` or `phi(i)` are set before calling `mkdets(custom1)`, it is essential to preserve the linearity of these values in dunkns so that the resulting determining equations have this same linearity. See section 7 for more information on maintaining linearity in manipulations on determining equations, and see the data file `burglb3b` for an example of `mkdets(custom1)`.

The prolongation coefficients ϕ_i^J are calculated using the recursive rule

$$\phi_i^{(k,J)} = D_k \phi_i^J - \sum_{m=1}^{\texttt{!*p}} u^i_{(m,J)} D_k \xi^m,$$

where (k, J) means $(k, j1, j2, \ldots, jn)$ and the kth total differential operator D_k is given by

$$D_k = \frac{\partial}{\partial x^k} + \sum_{i=1}^{\texttt{!*q}} u_i^{(k)} \frac{\partial}{\partial u^i} + \sum_{i=1}^{\texttt{!*q}} \sum_{|J|=1}^{\infty} u_i^{(k,J)} \frac{\partial}{\partial u_J^i}.$$

The ϕ_i^J are normally calculated (or expanded in terms of lower order ones) by the operation op!*proexp within `solvedets()` for efficiency, and not by `mkdets()` (their dependence is not set, as it is assumed they will be expanded). D_k is available for algebraic use by a function call $D_k(expression) = \texttt{totder(k,}expression)$.

Once `prosymvec` is properly constructed, then for each original equation E in implicitly zero form, the value `prosymvec`(E) is formed (meaning the vector derivative by `prosymvec` of E) which I call a predetermining equation, for want of better terminology. Each of the original equations are then substituted into this predetermining equation until no more substitutions are possible –this effectively substitutes prolongations of the original equations as well. The condition for `symvec` to be a symmetry is then that this predetermining equation be zero (almost) everywhere, and this is then placed on the internal list `!*deteqns` of determining equations to be solved.

Chapter 6

`solvedets`–Solving the determining equations

Once again, solving the determining equations is usually a straight forward matter for the user, performed by calling `solvedets(`*algorithm* `)`, where *algorithm* is usually `std`; other algorithms are described in section 6.4.

Interaction with the program may be necessary if the `std` algorithm is unable to solve some equations. See section 7 on manipulating the determining equations for more information.

For the more advanced user, it is possible to interact with the `std` algorithm.

6.1 Equation-groups– the internal representation

Each equation is stored in an internal representation called an equation-group, or eqngrp. This representation consists of a standard REDUCE representation of the equation along with other useful information such as a unique number for this equation which serves as a history or time measure, a list of solution algorithms already tried on this equation, and how this equation was generated. Some tracing output may print some of this equation-group information.

6.2 The algorithm and operations used by `solvedets()`

An algorithm to be used as an argument for `solvedets()` is associated with a sequence of operations and some parameters for the overall basic `solvedets()` algorithm. The basic algorithm operates on a list of equation-groups, and successively applies highest priority operations to each equation-group in the list (or to the whole set of equation-groups for some operations) until an operation is successful, in which case it starts over again. An operation might generate new equationsgroups from the old, drop old equation-groups from the list, set values for unknowns in the equations or a combination of these.

The overall aim of the `solvedets(std)` algorithm is to assign values for as many of the detunknowns in `symvec` as possible, so that we have the most explicit representation possible for the symmetry group. Because there is no way of explicitly stating the solution to all given differential equations which might arise, we must content ourselves with an algorithm which

will find the explicit representation for some symmetry groups and give something useful for the other cases.

The overall strategy of the `solvedets(std)` algorithm is to put the determining equations into a standard form while taking as many shortcuts as possible and solving explicitly any equations in the set which the algorithm recognises the solution to. It is surprising how far this approach takes us with a small set of recognised equations. Dimsym is currently able to recognise the solution to equations which can be solved for one unknown in the equation in terms of the others, possibly with term by term integration; first order integrating factor type equations; and second order constant coefficient equations with pure trigonometric or pure exponential solutions. Recognition of these three types seems sufficient for almost all examples I have tried, and can be extended if sufficient interest warrants this. It is an important consideration that the algorithm will complete within a finite number of steps.

The Standard Form representation of the equations involves substituting the equations into themselves and including integrability conditions all in an orderly fashion. This is the subject of much attention, see [Jan20], [Rei90a], [Rei90b], [Rei91], [Sch84] for more details.

For efficiency, there are many short cuts utilised, which involve separating special cases of the operations and applying these operations in a manner in which the equations will simplify as fast as possible and in particular we want to avoid explosive expression growth by leaving the substitution of large expressions until as late as possible, as well as avoiding simplifying or factorising large expressions. In particular, we use the techniques of splitting equations with free variables, and separation of variables when we can recognise this easily.

In order to minimise work in op!*sub2sf, we are careful to solve for the highest (janet) ordered dunkns we can when making assignments within other operations, so that we don't effectively change the ordering too much, which would lead to lots of resubstitution.

The reasoning behind the `std` algorithm is described in Sherring [She93a].

The operations in the `std` algorithm in order of their priority are:

op!*shr1tm   Single term contraction equations. Solves equations where an unknown or first order derivative of an unknown is zero. Solving this equation can only reduce the number of terms in all other equations.

op!*splitEc   Splits an equation by taking coefficients of free terms (ie terms involving variables that none of the detunknowns in the equation depend on). This operation is not applied to equations involving funkns.

op!*simpeq   Simplifies an equation by calling the REDUCE simplifier.

op!*proexp   Expands the ϕ_i^J prolongations.

op!*slvtwo   Solves special two term equations.

op!*get1tm   Forms 1-term equation as if we differentiated m times.

op!*hidefr   Hide equations involving freeunknowns. Any later operations may use these equations to make substitutions, or even worse, substitute them into other equations. We don't want to do this unless we have no other options open.

op!*hidelg   Hide long equations.

op!*exp1tm Single term expansion equations. Solves single term equations not solved by op1, ie a higher order derivative of an unknown is zero, so substitute an appropriate multivariate polynomial. Solving these equations with mixed derivatives can lead to messy expansions and redundancies, and this is not done.

op!*slvspl Solve for an unknown. One unknown can be expressed explicitly in terms of the others, without conflicting with the set dependencies of that unknown and so the number of implicit variables is reduced. So solve for this value and make a REDUCE assignment. Sub-expressions may be created, with new dunkns in the value set and new equations for the value, for efficiency. No derivatives are permitted in the value.

op!*trgexp Second order constant coefficient equations with pure trig solutions or pure exponential solutions.

op!*intfac First order integrating factor method.

op!*intslv Term by term integration. The equation can be integrated term by term until op!*slvall succeeds, so do it.

op!*xdpxpd Separation of variables: if we restrict all dunkns in the equation which do depend on xdeps to new ones which don't, then we can solve for one of the dunkns in the original eqn which didn't depend on xdeps.

op!*slvall Same as op!*slvspl except we relax the requirement that the number of implicit variables is reduced, and sub-expressions are not used.

op!*splitEd Same as op!*splitEc, except it also is used on equations with funkns.

op!*sub2sf Substitute the equations into each-other until the are in normal (orthonomic) form (this is Greg Reid's standard form without integrability conditions).

op!*addintcons Add one level of integrability conditions. After all of these are done we have Greg Reid's standard form.

op!*findlg Find the equations hidden by op!*hidelg.

op!*findfr Find the equations hidden by op!*hidefr. There is nothing else we can do with the other equations.

op!*odetst Check to see if any equations are O.D.E.s, and report.

op!*slvalldfs Same as op!*slvall except it also solves for values with derivatives.

6.3   Tracing and debugging

These are some routines which may be useful.

trace() This provides a step by step description of the operations performed on the equations. Very useful for understanding the algorithm, or for debugging, but very verbose, and is mostly only useful for following a limited number of operations.

traceat(n) Starts tracing at equation n. Useful for when you get an error or warning message and want to know why but don't want to do a full trace from the start. Try starting the trace a couple of equation-groups before the one that gives the message.

tracecute() Gives a limited information trace which is often only a few characters, to indicate what the algorithm is doing. Useful for seeing how the algorithm is proceeding without having to read all the mess generated by a full trace. The characters displayed are:

⟨n m⟩ After n equations (ie. using the number of equations as a history measure) there are m equations remaining on !*deteqns.

B An equation-group is born.

D[n] An equation-group is dropped, leaving n equation-groups (including hidden ones).

S An equation-group is simplified to a new equation-group.

! Reduced the dependence list of an unknown function.

ˆn A polynomial of order n is substituted for an unknown function.

=n/m A value is set for an unknown function as an expression with numerator of length n and denominator of length m.

; A subequation is being used in op!*slvspl.

In/m An expression with numerator of length n and denominator of length m is being integrated.

Pn/m An equation could have been solved to set a value for a dunkn as an expression with numerator of length n and denominator of length m, but we have passed because the expression included DFs of dunkns.

$k%n(m)l An equation with k dunkn terms and length n is factorized, a factor of length m is removed leaving an equation with length l.

$:-(An expression is too long to factorize.

˜n˜ A factor of length n is being divided by.

("n") A factor of length n is being factored from a standard quotient.

? An equation-group is attempting to be substituted into another (or the original differential equations are being substituted into some expression).

+(n) (or +n) An equation-group is successfully substituted into another (or the original differential equations have been substituted into some expression and are being reparsed). n differentiations were required (or n substitutions using the original equations were made)

An equation is not tested for subbing because it has already been done before.

∥ No substitutions could be made.

& An equation makes it onto `!*canonlist`, a list of equations in canonical form.

-n n equations are dropped from `!*canonlist` as we insert a new equation.

The emphasis here is mostly on time consuming factorization, substitution and integration. Using this with `on gc;` (if garbage collection reporting is available in your implementation of REDUCE) will show where all the action (or lack thereof) is happening if things are a bit slow.

`tracecute()` also gives information on the number of functions with any given number of dependencies which appear both in `symvec` and the remaining equations. This information is displayed each time a value is set or a dependency is reduced (ie. each time it changes) and is given as two lists, one for `symvec` and one for the remaining equations, of pairs of numbers indicating first the number of dependent variables and then the number of functions present with that number of dependent variables. This is very useful for tracking just how Dimsym is progressing, and how close it is to finishing. There is also some other miscellaneous information displayed with `tracecute()`.

`traceall()`

Gives over-the-top tracing which is far too verbose to make much sense of.

`stopat(n)`

Stops the solution algorithm at equation n. Use this if things seem to be getting out of control and you want to see what the system looks like by calling `showdets()` or `showeqngrps`.

`verify()`

Attempts to verify parts of various operations. Specifically, if `verify()` is called before `solvedets`, then `solvedets` will attempt to check that `solvedets` splits equations correctly (by remultiplying them by the independent factors, and showing the linearly independent list of factors). Integration operations will substitute the result back into the equation that has been integrated to check that it is zero. If `verify()` is called before `mkgens` then the program will recheck that each generator is actually a symmetry. If `verify()` finds an error or if there are any equations remaining which affect this generator then it will give additional conditions on that generator. No message is given if the generators are correct. All of this is done at some expense of processor time and storage space, and is just a debugging tool which is useful for finding when user defined LET rules are inappropriate (such as incorrect integration rules).

Note the use of `check()` etc described in section 8.

6.4 Other algorithms

The `std` algorithm will solve most of the problems which Dimsym can handle. It incorporates almost all the methods and ideas that I have implemented, and there are only a few examples I have where I need to use any different algorithm.

However, there are certain parts of the `std` algorithm which are useful to access separate from the whole algorithm, and so I here is a full list of other algorithms currently available. If you have a tricky or large problem which the `std` algorithm will not handle, then it is unlikely that you will just be able to choose a different algorithm which will work by itself– you will need to get your hands dirty. The description of these algorithms is relative to the `std` algorithm– they all use different combinations of the same operations used by `std` with only a couple of exceptions.

`simp` This simplifies any equations flagged for simplification.

`proxpnd` This uses the prolongation formula for any unexpanded prolongation coefficients in the equations.

`stdsplit` This algorithm calculates prolongation coefficients and splits the equations to give the equations which are usually considered to be **the** determining equations (except that Dimsym drops multiples of the same equation).

`split` This is the recommended way to get just the determining equations if you have some other use in mind. It solves some single term equations and can be a lot faster than `stdsplit` for large problems.

`alansplit` This algorithm is similar to `split` except that it is designed to give equations more closely resembling the equations given by Alan Head's program Lie [Hea93], which is useful for comparing/debugging.

`std` This is **the** algorithm which Dimsym is based on.

`std1` This tries a bit harder than `std`. It uses a couple more operations early on (which are redundant to later ones), and reports some more info. Just likely to be slower overall, without any expected gain.

`easy` We just solve the easy equations here... One and two term equations, simplification and splitting, easy separation of variables. No messy assignments, integrations, substitutions or integrability conditions.

`nosub` Stops before substitution to Standard Form, and so no integrability conditions either.

`stdform` This is an implementation of Greg Reid's Standard Form algorithm, as described in [Rei90a].

`stdform1` This is a slightly modified version of Greg Reid's Standard form algorithm. We take some short cuts such as solving one term contraction type equations, two term equations without nasty coefficients, splitting equations and easy separation of variables.

`nofxp` The same as `std` except we don't do any hiding or finding of equations with freeunknown functions in them (or of long equations either).

`hidef` Hide any equations with freeunknown functions in them, to be used later. This saves us from making any restrictions on freeunknown functions too early, which is **very** important when considering group classification problems.

`findf` Recover any equations with freeunknown functions in them which have been put aside.

hidel Hide any equations which are too long, to be recovered later. This saves us from making any substitutions into this equation which may increase its length, until later when hopefully we have some substitutions which will decrease its length. This is the place to put equations which have failed to factorize because of length, and usually (but not always) indicates inevitable expression swell.

findl Recover any long equations which have been put aside.

inttbt Integrate equations term by term once if possible. This operation is not part of the **std** algorithm, as it is against the integration strategy which only integrates equations if we can then solve them immediately. However, op!*trgexp which solves second order constant coefficient O.D.E. type equations may need to have the equation integrated term by term before it will recognize and solve the equation (this should be rectified sometime in the future).

rmvred Remove redundancies within the determining equations and **symvec**. The two operations involved here are not part of the **std** algorithm because the integration strategy of Dimsym is designed to **never** introduce redundancies. These operations would be needed if we were to solve first order equations with mixed derivatives.

6.5   Lisp parameters

Here are some Lisp parameters as they appear in the source code. These may be changed by the same command with a different value, after loading Dimsym. I have included these here for completeness, not with the expectation that you will need to make changes. However, some of the examples indicate cases where it is expedient to do so.

```
symbolic (!*keepints:=nil);        % Do we keep explicit integrals
                                   % (not including dunkns)
symbolic (!*keepsubing:=t);        % Do we keep subbing in op!*sub2sf ?
symbolic (!*intfaconfree:=t);
symbolic (!*intcon2bylowsdfdunkn:=nil); % Set to true after testing...
symbolic (!*i2highlong:=nil);
symbolic (!*intcon2onmin:=t);      % We only want to calculate minimal
                                   % int cons, ala Reid.
symbolic (!*intcon1by1:=nil);      % Do we want to calculate more than
                                   % 1 intcon1 at a time?
symbolic (!*intcon2by1:=nil);      % Do we want to calculate more than
                                   % 1 intcon2 at a time?
symbolic (!*intcon2onlowdunkn:=nil);
symbolic (!*op!*sub2sfby1:=t);
symbolic (!*op!*findfrby1:=nil);
symbolic (!*subeqnsbydunkn:=t);
symbolic (!*op!*findlgby1:=t);
symbolic (!*factordivides:=t);     % to pull out only important
                                   % information

symbolic (!*eqnlengthlimit:=100000);% Put aside anything longer than
                                   % this...
```

```
symbolic (!*factorizelimit:=10000);% Factorize equations with fewer
                                   % terms than this.
symbolic (!*liesublimit:=30);      % As high as necessary, not
                                   % infinite...
                                   % Recursion limit for levels of
                                   % subbing
                                   % eqns into themselves when making
                                   % determining eqns.
symbolic (!*exp1tmordlimit:=20;    % Lim on ord of polys made by
                                   % op!*exp1tm
symbolic (!*intnnumlimit:=100);    % Limit on the num.  of what we
                                   % integrate
symbolic (!*intndenlimit:=4);      % Limit on the den.  of what we
                                   % integrate
symbolic (!*asmtnumlimit:=100)     % Limit on the num.  of assignments
                                   % made by op!*slvall
symbolic (!*asmtdenlimit:=5);      % Limit on the den.  of assignments
                                   % made by op!*slvall
symbolic (!*intfacnumlimit:=4);
symbolic (!*intfacdenlimit:=2);
```

Chapter 7

User manipulation of the determining equations

If the program fails to completely solve the determining equations, then it may be necessary to make some manual manipulations, either on the values for the unknowns in the determining equations or on the actual determining equations themselves, before calling `solvedets` again. To manipulate the determining equations, they must be brought into algebraic form by calling `showdets()`, which sets the value of `deteqn(i)` to the i-th determining equation, in implicitly zero form. Note that these values will remain unchanged by the program until a further call of `showdets()`, even if the program has solved all remaining equations. Once these values have been set, they may be manipulated in any way using standard REDUCE manipulations, however the program must be re-fed any changed equations. Unless this is done, Dimsym will just continue to use the same equations it has stored internally. There are three ways to notify Dimsym of these changes.

`readdets()` will cause the values of `deteqn(i)` to be read as the new determining equations, dropping the old ones. Do this when you have changed the deteqn(i)'s.

`adddets()` will cause the values of `deteqn(i)` to be read as new determining equations to be appended to the list of old ones, rather than replacing the existing ones.

`simpdets()` will cause the equations stored internally on the Lisp list !*deteqns to be simplified. Do this when you have set values for the unknown functions.

When setting values for the unknowns, it is important to keep them linear homogeneous in unknown functions, otherwise the equations may no-longer be linear, and symvec will not be linear in the unknowns. (Dimsym assumes that symvec and the determining equations are always linear in the unknowns because it simplifies calculations and makes for easier representations. This is true for all types of symmetry except for conditional symmetries. Some nonlinear solution ansatz are also possible, such as separation of variables techniques, but Dimsym does not use any of these.) To do this, you must be able to create new dunkn (determinable unknown) functions. There are several functions to assist you in doing this, don't just use a `c(i)` function with your own value of `i` or the program may reuse it with dire consequences.

`newconst()` generates a dunkn function depending on nothing, ie a constant.

`newarb(`*dependencies*`)` generates a dunkn function depending on *dependencies*, eg. `a1:=newarb x(1),u(1);` will set `a1` to a new unknown function of `x(1)` and `u(1)`.

newpoly *polyvar, polyord, coeffdeps* generates a polynomial of order *polyord* in the vari-
able *polyvar* whose coefficients are dunkn functions depending on *coeffdeps*.
For example,

```
c(12):= newarb(x(1),u(1))*myfunction(u(2))
        + newpoly x(1),2,x(2),u(1),u(2);
```

Note that the statement

```
c(1):= sin(x(1))*newconst() + newconst();
```

would be simplified by REDUCE to

```
c(1):= (SIN(X(1))+1)*newconst();
```

before newconst() is evaluated, and so would only involve one new unknown function. The
way to avoid this is to construct the new unknown functions first using dummy variables,
eg.

```
a1:=newconst(); a2:=newconst(); c(1):=sin(x(1))*a1 + a2;
```

The datafile liouv3 is a good example of this type of manipulation.

Chapter 8

`mkgens`–Constructing the symmetries

You can examine the algebraic value `symvec` or its prolongation `prosymvec` to see the general form of the symmetry vector, or the more usual way to see the symmetries is to call `mkgens()` which splits `symvec` into its individual generators. After calling `mkgens`, the algebraic values `gen(i)` can be accessed for each of the generators. These generators are vectorfields which can be manipulated using the exterior calculus package EXCALC. Several routines are provided by Dimsym for use with vectorfields: `check(`*vector*`)` will check to see if *vector* is a symmetry vector of the given equations, `prolong`*(n,vector)* calculates the nth prolongation of the vectorfield *vector*. `vecder`*(vector,expression)* calculates the vector derivative of *expression* by the vectorfield *vector*. This last function can also be performed by the EXCALC Lie derivative functions. `showcomms()` will display the commutators of all the generators found by `mkgens()`, and `comm(`*vec1,vec2*`)` will give the commutator of two vectorfields. There are currently some problems with the interaction of Dimsym with EXCALC, see section 13 for details.

Chapter 9

Arbitrary functions, group classification problems and sub-expressions.

Dimsym is capable of performing group classification problems– If you give an unknown function in the original differential equations, then Dimsym will attempt to calculate the symmetries for the most general form of that function. It will report any assumptions it makes on the form of these functions in two ways, both of these serve to exclude certain cases. The simplest condition that the program reports is when it has divided by an expression involving the unknown function(s). There is an obvious condition here that this expression not be zero. The program will factor any such expressions, and keeps a list of them so as not to report them multiple times. The other condition that the program will report comes from splitting the determining equations in op!*splitE, where assumptions are made on the linear independence of certain expressions. If these expressions contain free unknowns, then this operation will exclude certain cases, which can be found by solving the associated (possibly differential) equation. By re-running the program making the assumption that the freeunknowns are of the form previously excluded, it is possible to identify special cases where the given equations have more symmetries. To aid these calculations, the function calls `showindeps()` and `showdivides()` are provided, to neatly report all conditions after `solvedets()` has terminated. It is worth commenting that when a long expression is reported as a divide, there is usually at least one free variable (one on which none of the funkn functions depend) which can be used to split the equation- so we can take the coefficients of linearly independent terms as separate simultaneous conditions. This will probably be implemented as an automatic feature at a later date. See the examples for more details.

Any function used as a free unknown must be declared as such by giving its name as argument of a call of `freeunknown`. The user should set the appropriate dependencies using the REDUCE command `depend`, or if the dependencies are to be given explicitly then the funkn must also be declared an operator in the usual way, and must have an argument included in the `freeunknown` declaration to indicate that it is of operator type. The former method is generably more preferable, but see data file `cent3d` for a useful example of the latter.

Another very practical use for free unknown functions is to use them as sub-expressions when the original differential equations are quite complex. This is most useful for when the original differential equations involve some expression which depends on only one or two

$\mathtt{x}(j)$ variables and that expression is long, involves a denominator, and/or appears repeatedly. Then it is expedient to use a funkn function to represent this expression, especially during the formation of the determining equations. Once the system has stabilised in length, the funkns can have their actual values assigned to them. The danger here is that some assumption may be made on the value of the funkn which contradicts its true value. To avoid this, it is best to only proceed partially into the `std` algorithm. Also, in order to check that any divides will not simplify to zero, the function call `showsimpdivides()` is provided and should be called after the values for the funkns have been set. It is not so easy to do the same for assumptions made about linear independence of expressions involving funkns, and so it is best to avoid the use of op!*splitEd. Once again, see `body3-3d` for a demonstration. This handling of funkns as sub-expressions may be improved in a later version of Dimsym.

Chapter 10

Danger– User beware! Simplification problems

Dimsym is designed to work with arbitrary functions, just as is REDUCE. By declaring an identifier as an operator by the REDUCE call `operator`, you can introduce a name for any function you like, and by utilising `let` rules, you can give REDUCE the structure of this function, specifying special values, expressions for its integral, derivative etc. Dimsym happily uses any function so defined. There is, however, a fundamental problem which you should be aware of if using any special functions. Indeed, Dimsym will use some special functions (namely exp, sin, cos, and log) without you even typing them in.

Consider the equation `c(1)*cos(x(1))**2 + c(1)*sin(x(1))**2 - c(1)=0`, which simplifies to zero using the standard trigonometric identity. However, if REDUCE didn't simplify this to zero because it didn't know the identity, then Dimsym would divide through by the factor `cos(x(1))**2 + sin(x(1))**2 - 1` to get the equation `c(1)=0`!!!

It is the responsibility of the user to ensure that all such identities are known to REDUCE if they are likely to be used. To assist, Dimsym reports any usage of any special function the first time it is used with any given argument. The program assumes that any polynomial, involving this and any other special functions with arguments reported to the user, can not simplify to zero unless the REDUCE simplifier knows about it. Just to be really sure, Dimsym reports any division involving operators (other than `x()`s or `u()`s). Dimsym factorizes divides before reporting them, and it may be useful to inhibit this if you are concerned about polynomials in the special functions present in the system. This is possible by setting the flag `Lisp(!*factordivides:=nil)`, but be prepared for some messy divides.

Dimsym may report that it has been unable to do some integral which you can see how to do... you may then wish to include a `let` rule so that REDUCE knows how to do this integral. However, in the interests of efficiency, it is best not to include too many unnecessary integration rules.

The program includes the identity

$$\texttt{for all x let sin(x)**2 = 1 - cos(x)**2},$$

because trig and exp functions can be introduced by op!*trgexp. `exp()` can also be introduced by op!*intfac and `log()` can be introduced by integration. REDUCE 3.5 includes some simplification rules for these.

Chapter 11

Other features

Dimsym can be used to solve systems of linear P.D.E.s that it hasn't generated itself. This is accomplished by assigning the value of the operator `deteqn(1), deteqn(2)` etc to expressions which are the linear P.D.E.s to solve, given in implicitly zero form. Then by calling `readdets()` and `solvedets()`, the program will proceed as usual. It is essential that the guidelines in section 7 for maintaining linearity are observed. See the data files `exfree` and `firstint` for example. For an example of a program which produces this type of equation which otherwise would be solved by the user, Sarlet and Vanden Bonne [SB92] describe a package for finding adjoint symmetries.

The directory `util` on the distribution disk contains some utility programs for calculations related to symmetry analysis. RESOLVE gives procedures for resolving prolongations of symmetry generators into functional multiples of other prolonged symmetry generators. A standard theorem then tells us that these functional multiples are first integrals of the original system of D.E.s. ODE automates these calculations for systems of O.D.E.s. FORMINT gives procedures for integrating exact and modulo exact 1-forms. This is also related to searches for first integrals as in Sherring and Prince [SP92].

Chapter 12

Function calls and algebraic variables

This is a list of the algebraic values and variables used.

$x(j)$ The j-th independent variable in the original differential equations.

$u(i)$ The i-th dependent variable in the original differential equations.

$u(i,j1,\ldots,jn)$ $u^i_{x_{j1}\ldots x_{jn}}$

$xi(j)$ The coefficient of $\frac{\partial}{\partial x_j}$ in `symvec`.

$phi(i)$ The coefficient of $\frac{\partial}{\partial u^i}$ in `symvec`.

$phi(i,j1,\ldots,jn)$ The coefficient of $\frac{\partial}{\partial u^i_{x_{j1}\ldots x_{jn}}}$ in `symvec`.

`ideq`(i) The i-th original differential equation in implicitly zero form, as used internally.

`deteqn`(i) The i-th determining equation in implicitly zero form, set only after a call to `showdets()`. `deteqn(`*0*`)` gives the number of determining equations remaining.

`c`(i) An unknown function to be determined.

`gen`(i) The i-th symmetry generator, set only after a call to `mkgens()`. `gen(`*0*`)` gives the number symmetry generators.

`symvec` The general form of the symmetry vectorfield.

`prosymvec` The general form of the prolonged symmetry vectorfield.

`!*p` The number of independent variables $x(i)$.

`!*q` The number of dependent variables $u(i)$.

`!*r` The highest order of the original differential equations.

This is a list of the function calls from algebraic mode.

`arbexp()` `arbexp(`*expression*, *coeffdeps*`)` returns a linear combination of exponentials of $\pm expression$ with coefficients `newarb(`*coeffdeps*`)`.

arbexp1() arbexp1(*expression*, *coeffdeps*) is the same as
arbexp(*expression*, *coeffdeps*) + newarb(*coeffdeps*).

arbtrig() arbtrig(*expression*, *coeffdeps*) returns a linear combination of sin and cos of
expression with coefficients newarb(*coeffdeps*).

arbtrig1() arbtrig1(*expression*, *coeffdeps*) is the same as
arbtrig(*expression*, *coeffdeps*) + newarb(*coeffdeps*).

check() check(*vector*) checks if *vector* is a symmetry of the original differential equations.
First *vector* is prolonged and then it is applied to each original differential equation
to give an expression which should be zero. A list of any non-z ero conditions is re-
turned. This could be used as an alternate (but slower) way of forming the determining
equations. The conditions are returned without any processing to remove factors or
split into simpler conditions, to make the check more reliable. T his is automatically
called for each generator in mkgens() if verify() has been called or the symbolic flag
!*verifygens has been set.

comm() comm(*vector1*, *vector2*) calculates the commutator or Lie bracket of two vectors, ie
the Lie derivative of *vector2* along the flow of *vector1*.

freeunknown() freeunknown(*kernel-list*) specifies each kernel on *kernel-list* as a free un-
known function in the original or determining equations. Eg. freeunknown a, b(i),
f(x(1)); Freeunknown operators must also be declared as operators in the usual way.
It is slightly preferable that freeunknowns with dependence have that dependence spec-
ified by a REDUCE depends statement, rather than given explicitly.

liesub() liesub(*expression*) recursively substitutes the original differential equations into
expression.

loaddeq() loaddeq(*solved-equation*) specifies *solved-equation* as (one of) the original dif-
ferential equations. See section 4. This may eventually be replaced by setting values
of deq().

mkdets() mkdets(*type*) will make the determining equations for symmetries of type *type*.
See section 5.

mkgens() This splits *symvec* into independent generators gen(*i*) and lists them. See section
8.

newarb() newarb(*dependence-list*) returns a new function c(*i*) which depends on *dependence-
list*. *i* is an incremented index.

newconst() This returns a new function c(*i*) which depends on nothing. Equivalent to
newarb(*nil*).

newpoly() newpoly(*polyvar*, *polyord*, *dependence-list*) returns an arbitrary polynomial in
polyvar of order *polyord* with coefficients newarb(*dependence-list*).

notrace() Stops the tracing caused by trace().

noverify() Stops the verifying caused by verify().

`prolong()` `prolong(`$n, base\text{-}vector$`)` gives the n−th prolongation of *base-vector*.

`readdets()` Causes the values of `deteqn(`i`)` to be read in as the determining equations, ie. as expressions which implicitly equal zero.

`showcomms()` Shows the commutators of the generators determined by `mkgens()`.

`showdeps()` Shows the dependence of all dunkn functions ever used.

`showdepsused()` Shows the dependence of all dunkn functions used in symvec, ie those of current interest.

`showdets()` Shows the determining equations remaining, and sets the values of `deteqn(`i`)` for manipulation. See section 7.

`showdivides()` Shows all of the divides which have been reported during the program run. This is just a nice way of gathering the relevant information, normally used at the end of the calculations.

`showeqngrps()` Shows the determining equations remaining in their equation-group format.

`simpdets()` Flags each determining equations remaining as requiring simplification, which will lead to a new equation-group as a side effect. Should be used if values have been set manually. See section 7.

`solvedets()` `solvedets(`*alg*`)` uses algorithm *alg* to solve the determining equations. See section 6.

`showindeps()` Shows all of independence conditions which have been reported during the program run. This is just a nice way of gathering the relevant information, normally to be called at the end of the calculations.

`showsimpdivides()` Shows all of the divides which have been reported during the program run, but first we call the simplifier on them, to check if any assignments have been made to freeunknowns which have resulted in division by zero. This is very useful for validating the use of subexpressions. Normally used at the end of the calculations.

`stats()` Gives a short summary of the time used so far, the number of equation-groups and the number of new arbitrary functions (just `c(`i`)`s) used, as well as a list of the number of successful applications of each operation.

`stopat()` `stopat(`n`)` Instructs the solving algorithm to stop once n equation-groups have been used. `stopat(`*0*`)` disables this.

`totder()` `totder(`$i, expression$`)` gives the total derivative of *expression* by `x(`i`)`.

`trace()` Causes various tracing information to be displayed.

`traceat()` Causes various tracing information to be displayed, starting once n equation-groups have been used. `traceat(`*0*`)` disables this.

`tracecute()` Causes some minimal tracing information to be displayed. See section 6.3.

vecder() vecder(*vector*, *expression*) evaluates the vector derivative of *expression* along the flow of *vector*.

verify() Causes various parts of the solving algorithm to attempt to verify their actions, such as verifying integrations by differentiating and comparing. This is extremely useful for detecting typographical errors when giving REDUCE integration rules.

Chapter 13

Release notes, known bugs and limitations of the program.

Dimsym includes the module intpatch written by Francis J. Wright to integrate dependent variables and rational powers. In the acorn implementation of REDUCE this causes the warning message +++ rerror redefined every time that the integrator is called. The standard reduce ordering function has been rewritten for efficiency... no known problems. Modules have not been used for compilation. This program has grown as an experiment, and so there is a large number of global variables which is not the best of programming styles. Most of these are program parameters and are effectively constant.

Known bugs: When using EXCALC with Dimsym there are some problems with vectorfields and shared operators. `symvec, gen(i)` and other vectorfields constructed by Dimsym are incompatible as EXCALC vectorfields, although the values themselves are compatible. This problem is easily overcome by declaring new EXCALC `tvectors` and then assigning their values accordingly. Eg,

3: **mkdets(point)$**

4: **load excalc$**

*** ^redefined

5: **pform x(i)=0,u(i)=0;**

6: **tvector exgen(i);**

7: **for i:=1:6 do exgen(i):=gen(i);**

However we now have the problem that any operator declared as a pform to EXCALC will cause problems to Dimsym if some Dimsym function calls are made. There is a temporary and partial fix for this in the file LEX, which provides routines `usedimsym()` and `useexcalc()`. Yes, this is very messy.

Limitations of the program: The main limitations of the program now appear to lie in two areas: in forming the determining equations we are limited by the expression size that REDUCE can work with. The equations for the Newtonian three body problem in three dimensions- (9 second order equations) are horrendously large if they are all expanded out and put over a common denominator and hence Dimsym grinds to a halt unless one uses

sub-expressions, to be expanded later in which case it can (with some manipulation of the algorithm) solve all of the equations. The other area where the program grinds to a halt is when it is trying to put the equations into standard form and adding integrability conditions. This can cause the size of the equations to grow explosively, rapidly reaching thousands of terms long, especially if there are any freeunknown functions. Dimsym factorizes each equation, and the factorizer grinds to a halt with expressions this large. A factorization limit is set to a default value of 5000 terms, but if this is exceeded and we don't factorize then the expressions grow all the more rapidly... If this happens then there are two choices we have: (1) Look or a less general type of symmetry: If looking for higher order symmetries then look for ones of lower order or point symmetries. If looking for point symmetries then look for ones which are polynomial (or some other explicit dependence) in at least one variable. Use `mkdets(custom1)` to do this- see section 5; (2) Don't put the determining equations into standard form. Avoid this by using `stopat(n)` or another algorithm, eg `solvedets(nosub)`. You then need to interpret the result yourself. This problem may arise from considering a problem with an infinite dimensional symmetry algebra which is dependent on the solution of a system of linear P.D.E.s, putting these (as well as other equations which should simplify eventually) in standard form can bring the calculations to a grinding halt.

A new algorithm which is more efficient in calculating the determining equations (form them without doing full expansion) should be employed but is beyond the scope of the first release of the program. The simplification management is handled carefully by Dimsym, but more intelligent management could lead to significant improvements. A new standard form algorithm employing parallel calculation of integrability conditions is being designed for implementation in a future release, and it is hoped that this will dramatically improve efficiency also. The reason for solving equations explicitly rather placing them in standard form immediately is twofold: firstly we have fewer equations to work with and secondly we are able to split an equation once we have given explicit dependence on some variable for each dunkn function in that equation, which leads to a rapid simplification of the system. It would seem that this first reason is really only an advantage when it leads to the possibility of splitting an equation. It would thus be expedient to focus attention on giving explicit representations in the same variable for all the dunkn functions in some equation. It is likely that an algorithm employing this strategy will give significant performance improvements. Unfortunately, Dimsym will require significant work to move in this direction, but this should certainly be a consideration of any further work in integration strategies.

Chapter 14

Installing Dimsym

Dimsym is supplied a Dos disk, along with various datafiles and utilities. At this time, the distribution disks contain the following directory structure:

ReadMe
Dimsym
mkdimsym
examples ⟨dir⟩
examout ⟨dir⟩
Headfiles ⟨dir⟩
Headout ⟨dir⟩
manual ⟨dir⟩

The program should be compiled and saved as a fastloading file. An example of how to do this is in the file `mkdimsym`, but this is system dependent and you may need to find out more about your system. It is not recommended to run Dimsym uncompiled, as this will be much slower. The file Dimsym in the root directory is the source code for the program Dimsym. This should be copied to the appropriate directory (eg `/usr/local/reduce/src`) if you want to install Dimsym as part of the standard library of fastloading modules (and if you have the access needed for this), otherwise it should be copied into the directory where you wish the compiled version of Dimsym to be placed, along with the file `mkdimsym`. Then enter REDUCE and type `in mkdimsym;` to compile the program, placing the compiled version either in the library for fastloading modules (eg `/usr/local/reduce/fasl`), or the current directory. If you have not compiled Dimsym into the fastloading library, then remember that you will need to either be in the directory where the compiled image is, or specify the path when loading Dimsym. In the event that fastloading files are not implemented in your version of reduce, you may have the ability to compile Dimsym and save it as part of standard REDUCE, although this is not a very nice way to do things as Dimsym upsets some of the standard REDUCE routines (specifically, I have rewritten the standard REDUCE ordering function, there are some clashes with EXCALC, and Dimsym is a rather large program to just have hanging around when you don't really need it). Failing the two above methods, if you cannot save a compiled version of Dimsym, then you should compile it each time you run it by setting the switch `on compiler;` before `in "dimsym";`. It will take around one minute cpu time to compile, depending on the speed of your machine, but this will be well worth it if you are running a difficult problem. Note that only fastloading files use the `load` command, whilst uncompiled ASCII files use the `in` command.

Chapter 15

It seems to be taking a long time...

Try using `tracecute()` to get an idea of where it is slowing down (see section 6.3). I usually use `tracecute()` the first time I run a new problem unless it is very simple, so that I get a feel for how Dimsym is progressing. If Dimsym gets stuck, then I may reduce the generality of the type of symmetry I am looking for, or use `solvedets(nosub)` instead. If you are short on memory then this may make life miserable for you :-(If you are using a PC version of REDUCE then try finding a mainframe installation, you may be pleasantly surprised. If you think the program should be able to do something more intelligently than it seems to be doing now, please let me know :-) Otherwise, see section 13.

Chapter 16

But it gives the wrong answer!

Before you decide that the program is unreliable after your first excursion with it, please check these things. Have you entered the equations correctly? It is very easy to omit an index. Inspect the values of `ideq(i)` after `mkdets()` to see if Dimsym interpreted them the way you expected. If you know what symmetries to expect, try using `check()` to test if Dimsym agrees with you. If it does, then things are looking grim. If Dimsym disagrees with you (by giving some conditions which don't hold true) then odds on you haven't entered the equations correctly. This is the number one cause of all "bugs" reported to me. Inspect the value and dependencies of `symvec` after calling `mkdets()` to see if Dimsym is really looking for the type of symmetries you think it is. Have you ignored some warning messages? It is quite possible that there is some simplification rule that needs to be used which has caused a division by zero or some other such gaff by its absence. If this has happened, then Dimsym will have reported the division or condition to you. Please don't just ignore these messages, they can be very important. My policy is that it is the user's responsibility to include any simplification rules necessary for any special functions that you use (or which might arise in the solution process) and that it is the program's responsibility to report anything that might cause problems without these rules (perhaps tending to be a little paranoid). This gives maximum flexibility and reliability to the program. Have you given some rule which isn't true? It is very easy for typos to slip into integration rules and the like (actually, those should be detected if the verifier is on, so give `verify()` a go). If you are doing manual manipulations of the determining equations, have you maintained the linearity of the equations and the dunkns? Have you called `readdets()` or `adddets()` after making changes to the determining equations? Have you called `simpdets()` after making changes to any dunkns? If you have exhausted all of these possibilities or are unsure what to try next, then please let me know. By doing a `trace()` from the beginning you should be able to find just where the derailment occurred, but this can be very time consuming for larger problems, and is almost never the way I look for gremlins.

Chapter 17

A guide to the examples.

The data files in the examples directory consist of some easy problems and some very hard problems, but almost all have been included to illustrate strategies for how to handle problems which don't go through Dimsym as easily as heat1, either because there is more to specifying the problem or because we need to coax Dimsym through to the solution. Many of the data files have references to where the problem came from. It is my intention to compile a greater list of examples, and any submissions for this compilation are welcomed. This will probably consist of three groups: short data files which are self explanatory and do not require supporting text, medium examples which illustrate features (or bugs) of Dimsym and/or of types of problem with around one page of supporting text (preferably in a TeX format), and longer detailed examples with perhaps several pages of supporting text. If there is something you find intersting then others may also learn from it so please consider submitting it. Such submissions should include the data file(s), text file if applicable, the version of Dimsym you are using and the machine(s) you have tried it on, along with timings and other notable statistics. These are the files found in the examples directory:

blas		
blasf	cev	karpman
blasf1	feder	liouv3
blasf2	fput1	mhd
body3-3d	heat1	mhd1
burglb3	hilbcart	reid1
burglb3a	hilbcart1	subic
burglb3b	hilbcart2	subic1
cent3d	igf1	ure
cent3da	jm	ure1

blasf. blasf, blasf1 and blasf2 are variations of the same problem, looking for symmetries of the third order blasius equation expressed as a system of first order equations. These examples serve to show the importance of placing the original differential equations in Standard Form. File blasf gives the correct input for this problem. It is a well known fact that the algebra of point symmetries of a first order O.D.E. is infinite dimensional, and the file blasf shows this. The infinite dimensional algebra is dependent on the solution to a differential equation which although linear, is much messier than the original equation(s). Just to select a few elements of this algebra, we look for `phi(1)` and `xi(1)` linear in `u(1)`, and find four symmetries: two of which depend on an arbitrary function of `x(1)`. File blasf1 contains the original equations in an equivalent but different expression. This time, they are not strictly

in Standard Form as the right side of the first equation may have `u(2,1)` substituted using the second equation. However, Dimsym can handle this form of the original equations as the resulting substitutions are finite in number and lead to a Standard Form: there are no formal derivatives of leading derivatives in any of the right sides of the same equations or any strange loops. Nor are there any integrability conditions to be specified, and I say that the equations are in Semi-Standard Form. This file gives the output as blasf, as we would expect. File blasf2 also contains the original equations in an equivalent but different expression. This time, they are not even in Semi-Standard Form, as the form of the third equation does not follow the conventions in section 5, and now the leading derivative of the second equation is a formal derivative of the third leading derivative. Dimsym gives good warning of this, and we proceed at our own risk. Sure enough, this time we do not find an infinite dimensional algebra, but only one poor symmetry! To further illustrate this problem, we enter two vectors `V` and `W` which we know previously to be symmetries of this problem. When we use `check()` to check if Dimsym will recognise them as symmetries, we are returned with a condition for each of them. Examination of these conditions shows that we can reduce them to zero with some substitutions of the original equations, but these substitutions do not follow Dimsym's mechanism. Thus the problem here is one of being able to recognise zero: of being able to substitute the original equations in a consistent manner, and this is part of the reason Dimsym expects the given differential equations to follow a set convention. The file blas is just the data file for the same problem properly expressed as a third order O.D.E. Do you recognize the point symmetries here? The vecorfields `V` and `W` which we checked in the other files are just the natural lifts of these two point symmetries. These natural lifts are effectively prolongations. Can you reconstruct the infinite dimensional algebra here using `mkdets(custom1)`?

body3-3d. The file body3-3d is included as an example of the power of Dimsym and some of the tricks I use to coax Dimsym through a problem of this magnitude, although you probably won't want to run it as it takes some 8 hours of cpu time on my machine. The problem being considered is the general three body problem in 3-space, that is, the newtonian dynamics of three point masses (or charges) under the mutual attraction of conservative radial forces. So it is a system of 9 second order O.D.E.s. Even the task of forming the original differential equations here requires some careful thought. Specifying the problem in terms of the arbitrary potential would leave messy denominators in the separations r_{ij}, with each force term involving two different radial separations as denominators, this expression would need to be then placed over a common denominator. Just the original differential equations themselves become long and complicated, and this would be magnified many times by Dimsym while forming the determining equations. The solution I use is to include all the radial separation information, including the true potential P, in a pseudo potential Q, so that $Q(r_{ij}) = P(r_{ij})/r_{ij}$. Next, I have to decide how to specify the pseudo potential Q as a free unknown, depending on the radial separation. There are actually 3 different pseudo potentials $Q_{ij} = Q(r_{ij})$. If we use three different freeunknowns each with dependence on the 6 relevant cartesian variables then Dimsym will report some very messy conditions involving derivatives which are tedious to check and isolate appropriate rules which must them be specified. As with the data file cent3d, I choose instead to declare `Q` a reduce operator, and a free unknown of one argument. With some help, REDUCE will then use the chain rule for derivatives, and conditions on Q will then be given in a much more usable form. Then I use implicit subexpressions for the `Rij`. Although I could give explicit expressions for these, they would involve messy square roots and lead to expression swell. Even if I used dependence on

the squares of the radial separations and gave explicit expressions for those, it would still be very messy. So I declare each of these as freeunknowns, depending on the relevant variables, together with some simple identities relating them (without these identities, REDUCE might make incorrect assumptions). Using subexpressions saves valuable workspace, and I will give the explicit expressions as needed, once the determining equations have contracted to a manageable size. The cost of this is that I must be absolutely fastidious in checking any conditions involving these subexpressions.

burglb3. burglb3, burglb3a and burglb3b are all variations of the same problem, looking for third order Lie-Backlünd symmetries of Burgers equation. Following the spirit of the reference given in burglb3 we solve the equation for `u(1,1):=` u_t, which is an allowable resolution, and form the determining equations by calling `mkdets(liebacklund(3))`. This sets $\xi^1 = \xi^2 = 0$ and ϕ to depend on all derivatives up to third order, except those which are equal to or formal derivates of the leading derivative u_t. Effectively we have the REDUCE command `depend phi(1),x(1),x(2),u(1),u(1,2),u(1,2,2),u(1,2,2,2)` or $\phi = \phi(t, x, u, u_x, u_{xx}, u_{xxx})$. Solving these equations then gives the 9 expected symmetries. If, as in burglb3a, we instead choose to solve the given equation for `u(1,2,2):=` u_{xx}, which is also an allowable resolution, and once again form the determining equations by calling `mkdets(liebacklund(3))` then effectively we have the REDUCE command

$$\text{depend phi(1), x(1),x(2), u(1), u(1,2),u(1,1),}$$
$$\text{u(1,2,1),u(1,1,1), u(1,2,1,1), u(1,1,1,1)}$$

or $\phi = \phi(t, x, u, u_x, u_t, u_{xt}, u_{tt}, u_{xtt}, u_{tttt})$. Solving these equations gives us 27 symmetries! Even using `mkdets(liebacklund(2))` will give us 14 symmetries! Some thought will reveal that if we are to find the symmetries equivalent to those in burglb3 but with the resolution for `u(1,2,2):=` u_{xx}, then the correct dependence for ϕ is $\phi = \phi(t, x, u, u_x, u_t, u_{xt})$ so that there is no u_{tt}, u_{xtt} or u_{ttt} dependence. Then we must either set the dependence manually, or remove the unwanted dependence and then call `mkdets(custom1)`; I choose the former method. Note this requires us also to set the zero values for ξ^j. This is done in burglb3b, and it is reassuring to see that we do indeed find 9 symmetries this time, which are equivalent when we substitute the original equation. Can we recover the 27 symmetries found with burglb3a by setting the dependence for ϕ and using the resolution for burgers equation as in burglb3? Test your hypothesis.

cent3d. In cent3d we consider the central force problem in cartesian coordinates. This is an elegant example of a classification problem and the use of subexpressions, not entirely unlike body3-3d except that it is a much simpler problem to solve. Dimsym only reports two divides: `R` and `G'(R)`. The second of these is equivalent to the condition `G(R)` assumed not to be of the form `a1*r + b1` for any constants `a1` and `b1`. There are 3 sets of linear independence conditions reported:

```
Must have all of

G(R)

1

linearly independent in (U 3)


Must have all of
```

```
G(R)*R

U(3) *G'(R)

      2
G'(R)*R

G'(R)

R

linearly independent in (U 3)


Must have all of

G(R)*U(3)

U(3)

1

linearly independent in (U 3)
```

The first of these conditions just translates to the same condition as before: that `G(R)` is not linear in `R`. The second condition requires some thought. We may think of `R` as being an independent coordinate, and `u(3)` as being a function of `R`, so that this condition is a linear independence condition in `R`. Thus the second condition gives one condition we already have, and two new ones. The second of the new conditions is a subcase of what we already have, the first translates into a differential inequation

$$a_2 R G(R) + b_2 w(R) G'(R) + c_2 R^2 G'(R) + d_2 G'(R) + e_2 R \neq 0$$

where $w(R) = \sqrt{R^2 - u_1^2 - u_2^2}$ and a_2, b_2, c_2, d_2 and e_2 are arbitrary functions of `u(1)` and `u(2)`. Because we know that `G(R)` is just a function of `R`, we can take $w(R) = \sqrt{R^2 - f_2^2}$ with a_2, b_2, c_2, d_2, e_2 and f_2 all constant. The general solution to this inequation is rather messy, and so we might choose to look for any special cases arising from this by specifying the let rule for `G'(R)` like this:

```
freeunknown a2,b2,c2,d2,e2,f2;
for all r let g'(r) = -(a2*G(R)*R + e2*R)/(b2*w(R) + c2*R**2 + d2);
```

Note that if we had not used the let rule for `u(3)**2` in the data file, the second linear independence condition would have translated to a messier differential inequation. The third linear independence condition easily translates into the condition

$$G(R) \neq a_3 + \frac{b_3}{\sqrt{(R^2 - c_3^2)}}$$

for constants a_3, b_3 and c_3. As an alternative to all of this, we can force the use of divides rather than linear independence conditions by not using operation op!*splitEd, which is the splitting operation specifically for equations with free-unknowns in them. The easiest way to do this is just to redefine the operation as the null operation, and this is done in data file cent3da, at some expense of processor time and storage space. We then get the division

reports; unfortunately, some of these involve squares of `r` and all three `u(i)` variables, which are not linearly independent. In order to place these divides into a readable form, and also to check that none of them are zero, we use the let rule `let u(3)**2=r**2 - u(1)**2 - u(2)**2` and then call `simpdivides()`, which simplifies each divide according to the curent system simplification rules before reporting it. This gives us these 8 conditions:

```
Free or special functions found when dividing by

R
which simplifies to

R
Free or special functions found when dividing by

G'(R)
which simplifies to

G'(R)
Free or special functions found when dividing by

     2    2
U(1)  + R
which simplifies to

     2    2
U(1)  + R
Free or special functions found when dividing by

    2              2              2              2
U(3) *G'(R) - U(3) *G"(R)*R + U(2) *G'(R) - U(2) *G"(R)*R

        2              2                  2
 + U(1) *G'(R) - U(1) *G"(R)*R - 6*G'(R)*R
which simplifies to

 2
R *( - 5*G'(R) - G"(R)*R)
Free or special functions found when dividing by

    2              2              2                  2
U(2) *G'(R) - U(2) *G"(R)*R - U(1) *G'(R) - G'(R)*R
which simplifies to

    2              2              2                  2
U(2) *G'(R) - U(2) *G"(R)*R - U(1) *G'(R) - G'(R)*R
Free or special functions found when dividing by

    2              2
U(2) *G"(R) + U(1) *G"(R) + 2*G'(R)*R
which simplifies to
```

```
      2              2
   U(2) *G"(R) + U(1) *G"(R) + 2*G'(R)*R
   Free or special functions found when dividing by

      2              2              2
   U(3) *G"(R) + U(2) *G"(R) + U(1) *G"(R) + 3*G'(R)*R
   which simplifies to

   R*(3*G'(R) + G"(R)*R)
   Free or special functions found when dividing by

            2                  2                      2
   G(R)*U(3) *G'(R) - G(R)*U(3) *G"(R)*R + G(R)*U(2) *G'(R)

              2                   2                    2
    - G(R)*U(2) *G"(R)*R + G(R)*U(1) *G'(R) - G(R)*U(1) *G"(R)*R

                   2     2     2        2     2        2      2
    - 2*G(R)*G'(R)*R  + U(3) *G'(R) *R + U(2) *G'(R) *R + U(1) *G'(R) *R
   which simplifies to

    2                                        2
   R *( - G(R)*G'(R) - G(R)*G"(R)*R + G'(R) *R)
```

The first condition is trivial, the second is once again the condition that G(r) not be linear in r, and the third condition is also trivial. The fourth condition is another differential inequation $RG''(R)+5G'(R) \neq 0$ with general solution $G(R) \neq a_4 \int e^{(-5/2)R^2}dR+b4$. To translate the fifth condition, we note that u(1)**2, u(2)**2 and R are all linearly independent and so we may take coefficients to get the four simultaneous conditions $G'(R) - RG''(R) \neq 0$, $G'(R) \neq 0$ and $RG'(R) \neq 0$. Clearly this has only the solution that G(R) not be constant. The sixth condition translates in the same manner to G(R) not constant, and the seventh has the same solution as the fourth with the constant $-5/2$ replaced by $-3/2$. The eighth condition? This is an interesting looking nonlinear differential inequation, but it is easily solved for $G(R) \neq a_6R^{b6}$. I leave it as an exercise to find any special G(R) for which the problem has more symmetry–you will need to enter the above forms of G(R) which Dimsym has not considered, but there is no guarantee that any of these will give extra symmetries. You may also have to repeat this procedure, becoming more specific at each level. You may want to consider the correlation between these two different ways of obtaining conditions on the free unknowns. And don't forget to declare the new constants as free-unknowns. So which way is best? It is certainly more efficient for Dimsym to solve the determining equations using op!*splitEd, and usually the reported conditions art more tractable in this form. But if you have the inclination and Dimsym doesn't grind to a halt placing the equations in standard form, then it may be worth considering both methods.

cev. The data file cev shows two more features of Dimsym. We set up the equation and look for point symmetries as usual; however there are two determining equations remaining after solvedets(std) is finished. An quick look at these reveals that the second one is a second order constant coefficient type if it is integrated once. Dimsym will not currently recognise the need to integrate this type of equation so that op!*trgexp can solve it, so we

instruct Dimsym to integrate each equation once, if possible, by calling `solvedets inttbt`. This time Dimsym reports

```
   2
   ***In op!*trgexp, can't determine sign of  - B
   Try using a LET rule for SIGN of that expression.
```

with still two equations remaining. As dimsym does not recognise the sign of this expression and we want dimsym to solve this equation with op!*trgexp, we then make the necessary declaration. But we must first call `simpdets()` before calling `solvedets()`, because `solvedets()` has already tried and failed with op!*trgexp on this equation and doesn't yet know that anything has changed the state of the system. So `simpdets()` instructs Dimsym to simplify all determining equations with the REDUCE simplifier, and drop any information stored about those equations. Having done this, Dimsym proceeds smoothly.

feder. This is an example of nonlocal symmetry of the Federbush model, reproducing the results of Kersten [Ker89]. Firstly, in feder we look for point symmetries, and Dimsym unfortunately grinds to a halt. However, we are able to verify that $\omega^1 = (R_1+R_2)\,dx+(-R_1+R_2)\,dt$ and $\omega^2 = (R_3 + R_4)\,dx + (-R_3 + R_4)\,dt$ are conserved currents. In feder1 we look for nonlocal symmetries depending on $p_1 = \int_{-\infty}^x (R_1 + R_2)\,dx$ and $p_2 = \int_{-\infty}^x (R_3 + R_4)\,dx$, where the integral is on solution surfaces. These are potentials for the conserved currents ω^1 and ω^2, and are determined by the differential equations $(p_1)_x = (R_1 + R_2)$, $(p_1)_t = (-R_1 + R_2)$, $(p_2)_x = (R_3 + R_4)$, $(p_2)_t = (-R_3 + R_4)$. Once we include these differential equations to the Federbush system, p_1 and p_2 become local variables and we are now effectively working on a prolonged space. Even though we are looking for a more general type of symmetry than in feder, Dimsym is now able to solve these equations easily.

heat1. This is just the standard example for symmetry analysis.

hilbcart. hibcart, hibcart1 and hibcart2 all calculate the 14 internal symmetries of the Hilbert-Cartan equation. The only difference between hibcart and hibcart1 is the choice of leading derivative; both choices are ok however the square-root in hilbcart1 is valid only for half the equation space. As the equation doesn't possess the discreet symmetry $x \mapsto -x$ we cannot claim a priori that the symmetry algebra we have found is also valid in the other half space without checking with the negative square-root. An interesting way of looking at this problem is to use a freeunknown for the sign of the square-root, as done in hilbcart2, which makes it easier to compare the two algebras in the two different halfspaces. See how the representation of the algebra is not very nice here, and hilbcart is a much preferable way of solving this problem.

igf1. This example for finding the point symmetries of the one dimensional isentropic gas flow solves the determining equations much faster if we place the equations in Standard Form first. It is also an example where we needed to include an integrability condition– try solving the determining equations without specifying this integrability condition...

jm. Another test file.

karpman. Dimsym makes short work of these equations. Compare the time (88s cpu on an IBM 6000) with the 3 hours cpu on a VAX 8650 for the MACSYMA program SYM-MGRP.MAX reported in [CHW91], which leaves 69 determining equations to be solved manually (via the usual feedback process). The resolution of the original equations which we use is allowable, although it may take some thought to see this. We chose this resolution

to minimise denominators in the equations. Just to be sure, you may examine the list of variables that dimsym used to split by, as reported by the call to `stats()`, to see that none of those variables can be substituted by these equations in the resolution used.

liouv3. This example shows an interesting feature of Dimsym which lets us keep explicit integrals of explicit or freeunknown expressions. This is done by setting the lisp flag `!*keepints` to `t` (true). We still need to intervene manually to get the full solution. It is unfortunate that this example requires explicit reference to the `c(i)` functions in the equations. It is better programming style to avoid such explicit references because the actual `c(i)` functions in the equations may change if we use a different algorithm, or with future versions of Dimsym. The ability to express rules for the general solution of equations is a highly desirable feature.

mhd. This example comes from [Her93], with a reported time of 50 minuites cpu time just to create the 222 determining equations which must then be solved manually, using the feedback process. Reid and Wittkopf [RW93] report around 1 hour cpu to place these equation in Standard Form. Dimsym solves the whole problem automatically: forming the determining equations, solving them, and forming the generators, in less than 11 minuites cpu on an IBM 6000. Dimsym reports only one condition whereas [RW93] report another 3 superfluous conditions. Dimsym takes a lot of care to minimise the number of conditions on the result, because of the amount of work involved in checking these conditions. However, there is a lot of room for improvement. To date, there has not been a widespread policy of reporting cpu times with publication of results, making comparison of this aspect of different programs difficult. mhd1 is just the same equations as mhd, with a different resolution. No significant differences.

reid1. Data file reid1 contains the example from [RB91]. We form the determining equations as usual and use `solvedets stdsplit` to get the equations which are usually referred to as **the** determining equations. This gives us nine equations which do not look easy to solve, but once we use `solvedets stdform` to place the equations in standard form (without any integration), we are left with nine easy equations which only require a few simple integrations to solve.

subic. Data file subic shows how we can "lose" symmetries if we do not properly include all necessary integrability conditions. In particular, we can easily see that it is a consequence of these equations that `u(2,2,1)=0`, and yet we see from the `stats()` call that `u(2,2,1)=0` was used to split equations! The 7 symmetries found here are valid, but there may be more. In subic1 we specify this extra equation, and we do indeed find 3 extra symmetries. Also note how we reduce the remaining equation involving more than one function to a simpler equation involving just one unknown function and thus give a more explicit representation of the symmetry algebra.

ure. Here another important aspect of giving the equations in Semi-Standard Form is illustrated. The data file ure finds the point symmetries of Ure's equations, no problem. The data file ure1 contains Ure's equations, but this time they are not given in an allowable resolution, leading to an infinite recursive substitution. Dimsym gives an error message after the substitution limit is reached, and if we look at the equations Dimsym is trying to form, they repeat as Dimsym loops.

Headfiles. The directory Headfiles contains the datafiles circulated by Alan Head with his program Lie, modified for use with Dimsym. They are a good set of test files, and show Dimsym happily performing its duties.

Chapter 18

Literature and References.

There is an extensive literature both on the subject of symmetries of differential equations and finding them using computer algebra. For the former, any one of the contemporary texts [BK89], [Olv86], [Ovs78], [Ste89] provide an excellent treatment of the subject, and most of these have extensive referencing of journal articles. Most of the literature on the calculation of symmetries of differential equations using computer algebra consists of journal articles with specific calculations and only vague references to the code used. There are indeed many other programs available for these calculations, however most of these stop at just finding the determining equations, and perhaps aiding the interactive user with some simplification support. The programs which go the furthest with the automated calculations begin with Schwarz's SPDE [Scw85], which has been an excellent REDUCE program in its time; Head's Lie [Hea93], which makes very good use of the limited memory available in mumath; and Reid's Standard-Form [Rei90a], shows just how much progress can be made without integrating any of the determining equations. Reid also has algorithms and code for determining the dimension of the symmetry algebra. A thorough review and extensive bibliography is given by Hereman [Her93].

REFERENCES

[**BK89**] G. W. Bluman and S. Kumei, Applied Mathematical Sciences 81 (1989), Springer–Verlag New York.

[**CHW91**] B. Champagne, W. Hereman and P. Winternitz, *The Computer Calculation of Lie Point Symmetries of Large Differential Equations.*, Computer Physics Communications **66**, 319–340.

[**Hea93**] A. K. Head, *LIE: A PC Program for Lie Analysis of Differential Equations*, Computer Physics Communications **77**, 241–248

[**Hea91**] A. C. Hearn, *Reduce User's Manual Version 3.5*, RAND, Santa Monica.

[**Her93**] W. Hereman, *Review of Symbolic Software for Computation of Lie Symmetries of Differential Equations*, Euromath Bulletin **2**.

[**Jan20**] M. Janet, *Sur les Systèmes d'Équations aux Dérivées Partielles*, Journal de Mathematique **8III**, 65-151.

[**Ker87**] P. H. M. Kersten, *Infinitesimal Symmetries: A Computational Approach*, CWI Tract 34, Centre for Mathematics and Computer Science, Amsterdam.

[**Ker89**] ——, *Software to compute Infinitesimal Symmetries of Exterior Systems, with Applications.*, Acta Appl. Math. **16**, 207–229.

[**McW91**] M. MacCallum and F. Wright, *Algebraic Computing with REDUCE*, Oxford University Press, Oxford.

[**Olv86**] P. Olver, *Applications of Lie Groups to Differential Equations*, Springer-Verlag, New York.

[**Ovs78**] L. V. Ovsjannikov, *Group Analysis of Differential Equations*, Academic Press, New York, 1982; Translated from, *Russian title?*, Nauka, Moscow, 1978. (Russian)

[**Rei90a**] G. J. Reid, *A Triangularization Algorithm which Determines the Lie Symmetry Algebra of any system of PDEs*, J. Phys. A: Math. Gen. **23**, L853–L859.

[**Rei90b**] ——, *Algorithms for Reducing a System of PDEs to Standard Form, Determining the Dimension of its Solution Space and calculating its Taylor Series Solution*, Preprint, Mathematics Department, University of British Columbia, Vancouver, Canada.

[**Rei91**] ——, *Finding Abstract Lie Symmetry Algebras of Differential Equations Without Integrating Determining Equations*, Euro. Jnl of Applied Mathematics **2**, 319–340.

[**RB91**] G. J. Reid and A. Boulton, *Reduction of Systems of Differential Equations to Standard Form and their Integration using Directed Graphs Without Integrating Determining Equations*, Proceedings of the International Symposium on Symbolic and Algebraic Computation, Bonn.

[**RW93**] G. J. Reid and A. D. Wittkopf, *Long Guide to the Standard Form Package*, Preprint, Mathematics Department, University of British Columbia, Vancouver, Canada.

[**SV92**] W. Sarlet and J. Vanden Bonne, *REDUCE Procedures for the Study of Adjoint Symmetries of Second–Order Differential Equations*, J. Sym. Comp. **13** (1992), 683–693.

[**Sch91**] E. Schrüfer, *EXCALC: A System for Doing Calculations in the Calculus of Modern Differential Geometry*, REDUCE 3.5 Miscellaneous Documentation, The Rand Corporation.

[**Sch84**] F. Schwarz, *The Riquier-Janet Theory and its Application to Nonlinear Evolution Equations*, Physica 11D, 243–251.

[**Sch85**] ——, *Automatically Determining Symmetries of Partial Differential Equations*, Computing **34**, 91–106.

[**Sch86**] ——, *Addendum to: Automatically Determining Symmetries of Partial Differential Equations*, Computing **36**, 279–280.

[**Sch88**] ——, *Symmetries of Differential Equations: From Sophus Lie to Computer Algebra*, Siam Review **30**, 450–481.

[**Sch91**] ——, *The Package SPDE for Determining Symmetries of Partial Differential Equations, User's Manual*, REDUCE 3.5 Miscellaneous Documentation, The Rand Corporation.

[**She93a**] J. Sherring, *Symmetry and Computer Algebra Techniques for Differential Equations, PhD Thesis*, PhD Thesis (1993), LaTrobe University.

[**She93b**] ——, *An EXCALC Program for Integration Modulo Closed One Forms*, La Trobe University Mathematics Department Research Report (1993).

[**SP92**] J. Sherring and G. Prince, *Geometric Aspects of Reduction of Order*, Trans. Amer. Math. Soc. **334**, 433–453.

[**Ste89**] H. Stephani, *Differential Equations: their Solution using Symmetries* (M. MacCallum, ed.), Cambridge University Press.

DEPARTMENT OF MATHEMATICS, LA TROBE UNIVERSITY, BUNDOORA 3083, AUSTRALIA.
E-mail address: J.Sherring@latrobe.edu.au