

An Enhanced ODE Solver for REDUCE

Francis J. Wright
School of Mathematical Sciences
Queen Mary and Westfield College, University of London
Mile End Road, London E1 4NS, UK.
E-mail: F.J.Wright@QMW.ac.uk
WWW: <http://Maths.QMW.ac.uk/~fjw/>

1 May 1996

Abstract

Work is in progress to extend the REDUCE package ODESOLVE for solving ordinary differential equations (ODEs). The latest version has a more user-friendly and flexible interface, allows conditions to be imposed on solutions and can solve some classes of nonlinear ODEs. It performs well on recently published test examples. The interface supports systems of ODEs.

1 Introduction

REDUCE [8, 15] is one of the longest-established general-purpose computer algebra systems. Nevertheless, there is no “standard” REDUCE differential equation solver, although a number of “user-contributed packages” for dealing with various aspects of differential equations are included in the standard distribution. User-contributed packages are maintained primarily by their authors rather than by the main REDUCE developers, and most of them must be explicitly loaded before they can be used, but apart from that they behave like standard REDUCE facilities. REDUCE itself is written in Standard Lisp.

A user-contributed package called ODESOLVE [13] has been distributed with REDUCE for several years. It was written primarily by Malcolm MacCallum, and was described at the ISSAC '88 conference [12]. The version currently distributed (with REDUCE 3.6) was last revised in November 1990 for REDUCE 3.4. It was intended to be the first step toward a general ordinary differential equation (ODE) solver for REDUCE. This paper describes a second step toward that goal. I will refer to the distributed package as ODESOLVE 1. This paper describes an experimental prototype of an enhanced version of this package which I will refer to as ODESolve 1+, versions of which are available via my WWW home page. (To be precise, this paper describes ODESolve 1.03.)

Part of the original motivation for ODESOLVE 1 was to be able to use it in undergraduate courses [29], for which it was appropriate to keep both the underlying algorithms and the implementation fairly simple. Hence, the techniques implemented were essentially those that were being taught to first-

year undergraduates, although it was always the intention to develop the package to be much more sophisticated [12].

The current enhanced version consists essentially of additional layers built around a revised version of the original package. There remains a great deal more to be done and it still does not implement any sophisticated algorithms; nevertheless ODESolve 1+ can solve a lot of ODEs that the distributed ODESOLVE cannot.

1.1 An outline of the original ODESOLVE

The distributed package, ODESOLVE 1, provides a function called `odesolve` (REDUCE is by default not case sensitive) that requires precisely three arguments: an expression or equation representing an ODE (in any form), the dependent variable, and the independent variable. Following the documentation for ODESOLVE 1 [13], I will usually refer to the dependent and independent variables simply as y and x . I will also use y' to denote the first derivative of y , i.e. dy/dx .

REDUCE provides an elegant mechanism that allows variables to depend implicitly in some unspecified way on other variables (where actually “variable” can be interpreted rather loosely), which is established by a declaration of the form “`depend y,x`”. This is the natural way in REDUCE to set up the dependence of the variable (y), for which an ODE is to be solved, on the independent variable (x). ODESOLVE 1 requires suitable `depend` declarations to have been made explicitly by the user before `odesolve` is called, otherwise derivatives will simplify to zero and so disappear.

REDUCE also supports explicit variable dependence via an operator or functional notation of the form $y(x)$. (In Maple, this is the *only* way to specify dependence, which is sometimes hidden by the trick of declaring y to be an alias for $y(x)$, but this can become confusing!) Whilst in principle ODESOLVE 1 also accepts input in this explicit functional form, it does not always handle it correctly.

The `odesolve` interface was designed to be compatible with the interface used by `solve`, the standard REDUCE algebraic equation solver. The function `odesolve` returns a list containing a single equation that represents the solution of the ODE, either explicitly in the form $y = f(x)$ or implicitly in the form $f(x, y) = 0$, which depends on the type of ODE and is not under user control. If `odesolve` cannot solve the ODE then it simply returns it (in the form of an implicit solution where the function f involves derivatives). This was the convention used formerly by `solve`, but in recent versions of REDUCE `solve` returns unsolved components of a solution in terms of the function `root_of`. (This aspect of `odesolve` should probably be re-considered.) The solution returned by `odesolve` contains instances of the operator `arbconst` to represent the arbitrary constants required in the general solution of an ODE, where the argument of `arbconst` is a positive integer used to distinguish arbitrary constants.

The solution strategy adopted in ODESOLVE 1 is first to determine a number of properties of the ODE, such as its order, degree and linearity, by calling a procedure called `sortoutode`. Appropriate solvers are then called depending on whether the ODE is linear and whether it is first order. ODESOLVE 1 has no code to handle any nonlinear ODE that is not first order, which is perhaps

its major weakness. The solution strategy is entirely linear, following a branch of the decision tree from root to leaf with no recursion or backtracking.

A linear first-order ODE is solved using an integrating factor if direct integration (quadrature) is not possible. In the case of a nonlinear first-order ODE a number of tests are applied and if a test succeeds then an appropriate solver is called. There is support for ODEs of separable, homogeneous (in the sense of trivially scale-invariant) and nearly so, Bernoulli, and exact type.

A linear ODE of order higher than one is solved if it has constant coefficients or is of Euler type (equidimensional in x), which is trivially reduced to the former case. In the case of linear ODEs (in particular) it is useful to distinguish those terms that depend on y from those that do not by writing the ODE in the form $L(x, y) = R(x)$. The solution of a linear constant-coefficient ODE is computed as the sum of a “complementary solution” and a “particular integral”. The complementary solution is constructed from the roots of the auxiliary equation for $L(x, y) = 0$ in the form of a sum of exponentials, with polynomial coefficients if the corresponding root of the auxiliary equation is multiple. In the case of complex conjugate roots, an attempt is made to keep the solution manifestly real and re-express the exponentials in terms of sine and cosine functions. The particular integral is computed using an “inverse D-operator” method that involves repeatedly integrating $R(x)$ multiplied by an exponential for each root of the auxiliary equation (contrary to the comments in the code). A little more detail is given in [12].

Algorithmic tracing is supported and is activated by turning on the switch `trode`, which causes ODESOLVE to describe its solution strategy (cf. [19, page 14]).

1.2 Motivation to enhance the original ODESOLVE

MacCallum [12, 13] described plans for the development of ODESOLVE that were clearly desirable (if not essential) but, whilst some further code has been written [14], it has not been released. In the meantime, the differential equation solvers provided in other computer algebra systems (CASs) have moved on. Also, a number of independent specialized differential equation solvers have been written in REDUCE.

The following developments have been made within the QMW computer algebra group. A first-order ODE solver using the Prellé-Singer algorithm [20] has been implemented and further developed by Yiu-Kwong Man [16, 17], as part of a PhD project supervised by Malcolm MacCallum. Thomas Wolf and Andreas Brand¹ have developed the `CRACK` package [27], intended primarily for solving systems of differential equations, from its predecessor `CRACKSTAR`, and Wolf has developed the `APPLYSYM` package [25, 26, 28] that uses `CRACK` to apply symmetries to the solution of differential equations. Versions of both `CRACK` and `APPLYSYM` are distributed with REDUCE. The set of primitives for operations on linear ordinary differential operators (LODOs) specified by the CATHODE collaboration [7, 11, 18] has been ported to REDUCE [30].

In addition, REDUCE now provides within the library of user-contributed packages [8] good support for series solution of linear homogeneous ODEs (`DE-SIR`), Laplace transforms (`LAPLACE`), linear operator algebra (`NCPOLY`),

¹At Universität Jena, Germany.

numerical solution of (first-order) ODEs (NUMERIC), Lie symmetry analysis (SPDE) and special functions (SPECFN). It would be helpful, especially to naive users, if a package such as ODESOLVE could automatically call other such packages as appropriate, as was clearly part of the original intention [12].

Recently it has become fashionable to publish comparisons among CASs. One by Wester [24] included the solution of very simple ODEs as one of its components, and a more recent comparison by Postel and Zimmermann [19] dealt exclusively with the solution of ODEs and systems of ODEs, taking most of its examples from the handbook of solution techniques by Zwillinger [31]. REDUCE faired rather badly on these test sets.

It could be argued that these test examples shed very little light on the ability of a CAS to solve an arbitrary ODE, because they are all very special examples with simple structures that were designed to illustrate special textbook solution techniques. Small perturbations of such special-case ODEs would almost certainly not be solvable by most currently distributed CASs, and the main thrust of research and development within the REDUCE community has been toward general techniques that could handle such perturbed examples. Nevertheless, such test sets do shed light on user expectations and perceptions of CASs.

One aim of my enhancements to ODESOLVE is to make the REDUCE ODE solver more user friendly and better meet user expectations (without sacrificing formal correctness). I claim that REDUCE 3.6, patched up to date and running ODESolve 1+, now performs as well on average as any CAS on the test set of Postel and Zimmermann [19] for single ODEs. Further details are given in Table 1.² Although this only means that it can now solve more special cases than it could previously, this is not entirely trivial because often more general cases are solved by transforming them into special cases, which then need to be solved.

Problem class:	Linear	Nonlinear	Special	Initial conditions	Total
Max:	15	27	2	4	48
Best:	12	23	1	3	–
Old:	6	5	1	0	12
New:	11	27	1	4	43

Table 1: Comparison of the number of test examples from [19] solved out of the maximum possible (Max) by the best distributed solver in each class (Best), the distributed ODESOLVE (Old), and ODESolve 1.03 (New). Tests on systems of ODEs are not included.

My intention is that ODESolve should contain code to solve a large number of simple types of ODEs and when given an example that it cannot solve should automatically call the best external package. For the expert user there will be facilities to control the operation of ODESolve. At present, most of the special-case code has been implemented but not the interfaces to external packages. The biggest difficulty in calling external packages is that they have mostly been written to be called directly by the user, so that they often output messages to the user that are not appropriate when they are being called by another

²For reasons discussed later, I have not accepted that any system tested succeeds in solving the differential-delay equation in the “special” section.

routine; they have different conventions (switches, global variables, procedure arguments, etc.) for controlling their operation; and they may disturb the global environment. Not all of these problems can be entirely resolved by writing suitable interface code, and there may be a need to modify some parts of the packages.

2 The solutions returned by ODEsolve

The solutions returned by ODEsolve should be as complete and informative as possible. It should be clear both to the interactive user and to a program that has called ODEsolve what the form of the solution is. Additional information about the solution *process* should be available by selecting various levels of algorithmic tracing. Additional information about the solution itself may be available in appropriate situations by inspecting global variables, e.g. at present, when conditions are imposed on a solution, the general solution is still available as the value of the global algebraic variable `ode_solution`.

A *solution* may not involve derivatives or integrals of any *dependent* variables, so that an integral equation is *not* an acceptable solution. (This disagrees with the view taken in [19] of their differential-delay equation (43) – ODEsolve 1+ at least recognises that it is not a simple ODE!) However, I accept solutions involving integral representations (Liouvillian or more general, regardless of whether they have standard names), or equivalently I regard reduction to quadratures as a solution of the differential equation. (Of course, in the case of an ODE of the form $y' = f(x)$ the solution $y = \int f(x) dx$ is trivial, but I regard evaluating integrals as the task of an integrator, not of an ODE solver, although this is open to debate.)

The complete solution of an ODE (system) will generally consist of a number of independent components (each of which I will refer to as “a solution”). This is clear from the fact that an ODE may factorize (algebraically), in which case the solution must be the union of the solutions of the factors. Each solution may be a general solution, in which case it will contain a number of arbitrary constants, or it may be a singular solution, in which case it will contain (typically) one fewer arbitrary constants than the general solution.

Each solution of a single ODE may be expressed explicitly in the form $y = f(x)$, implicitly in the form $f(y, x) = 0$, or parametrically in the form $\{y = f(p), x = g(p)\}$. Depending on the type of ODE and the solution technique used, one of these solution forms is most natural. It may or may not be easy (or possible) to convert between any two forms. By default, ODEsolve 1+ returns solutions in the forms that arise naturally, but it provides a switch (implicit) that causes it to attempt to convert parametric solutions to implicit form and a switch (explicit) that causes it to attempt to convert all solutions to explicit form. If conversion fails, then an unconverted or partially converted solution is returned.

A single explicit or implicit solution is returned as an equation; a single parametric solution is returned as a REDUCE list of the form $\{y = f(p), x = g(p), p\}$. The name of the parameter is currently returned as the third element of the list in order to facilitate further manipulation. Each new distinct arbitrary constant or parameter is represented by a REDUCE operator called `arbconst` or `arbparam` respectively with a suitable positive integer argument. A program

calling ODESolve can distinguish the type of each solution (component) as follows: if it is a list then it is parametric, otherwise it must be an equation, in which case if the equation has 0 on the right then the solution is implicit, otherwise it is explicit (and has y isolated on the left). Tests of this nature are used internally.

The complete solution of a single ODE returned by ODESolve 1+ consists of a REDUCE list of one or more solutions, each of the above form, thereby generalizing the list of one equation returned by ODESOLVE 1. Each solution (component) of a *system* of ODEs will consist of a list of solutions, each of the above form, one for each dependent variable, and so the complete solution will always consist of a list of lists. This is completely analogous to the behaviour of `solve` (and similar to that of `CRACK`).

3 The structure of ODESolve 1+

ODESolve 1+ currently consists of a completely new user interface module that includes code to impose conditions on solutions of (single) ODEs, a module of fairly general transformations that are intended to “simplify” an ODE, a module of solution or transformation techniques for a number of special types of ODE, a module of patches and extensions to other parts of REDUCE, and four modules that comprise a modified version of the original ODESOLVE package, which still ultimately does most of the actual solving. The newly implemented solution algorithms are all described in [31, chapter II.A]. Changes of variable are all implemented using recursive rule lists, which seem to work well and avoid the kind of minor errors that appear in one or two of the transformation tables listed in [31, at least in the first edition].

Three distinct types of operation can be identified in the solution of ODEs. There are transformations that will never make the problem more difficult and so should always be applied, which I will refer to as *simplifications*. The simplifications are implemented by passing the ODE through a linear sequence of routines, each of which calls the next. There are transformations that will simplify some ODEs but make others more complicated, in which case it is necessary to backtrack out of them. There are operations that lead directly to an algorithmic solution. My intention in the design of ODESolve 1+ is that any routine that can fail returns either a solution or a failure message (the Lisp constant `nil` representing both the empty list and the Boolean constant `false`). Such failure messages are used by the calling routine to backtrack and try a different strategy.

This is a different convention from that used in ODESOLVE 1, and as a temporary expedient I have introduced a global variable (`!*odesolve_failed`) that is used in some places to indicate that a solution routine has failed. The strategy used in ODESOLVE 1 is to have a test routine and a solution routine, the latter being called if the former succeeds. However, I prefer to combine the two routines wherever possible.

The solution of an ODE proceeds in outline as follows; further details are given below. It is first simplified, and then various techniques are applied in sequence to try to solve it. If the simplification splits the ODE into several factors then the next stage is applied to each factor independently. As soon as a solution technique succeeds then the solution is returned (for that factor).

The solution sequence begins with attempts to recognise ODE types that have a direct algorithmic solution and later proceeds with attempts at transformations that may or may not lead to solutions. The transformations typically re-enter the main simplifier, so that in general the solution algorithm is recursive, and care must be taken to break potentially endless loops. This is a major difference from ODESOLVE 1, which involves a strictly linear decision process.

The algorithmic tracing provided by ODESOLVE 1 has been extended to ODESolve 1+, and is particularly useful to show what the re-entrant code is doing. A second level of tracing is also provided (currently by assigning a number to the shared variable `!*tmode`), which shows what operation is currently being considered, even if it is rejected and ODESolve backtracks.

3.1 The user interface module (`odeintfc`)

The principal user interface is via an algebraic-mode operator called `odesolve` that, in ODESolve 1+, accepts a variable number of arguments. It requires at least one argument, `ode`, which should evaluate to either an ODE or a list of ODEs, where an ODE can be either an expression that is implicitly equated to zero, or an explicit equation. If the optional second argument is either an identifier or a list of identifiers then these identifiers are used as the dependent variable(s). If the optional third argument is an identifier then it is used as the independent variable. An empty list can be used as a place holder. If the independent and/or dependent variables are not specified then `odesolve` attempts to parse them out of `ode`, which it does primarily by examining the derivatives, and it outputs a message to indicate what variables it has chosen.

Dependent variables can be either simple variables (e.g. y) or explicit functions of the independent variable and/or other parameters (e.g. $y(x)$). Any dependent variable that is not simple is checked to ensure that it appears everywhere with the same arguments. If it does then it is replaced by a generated simple variable (a Lisp “gensym”), to which the original variable name is attached as a property. This replacement makes subsequent code much simpler, because it avoids an expression depending (explicitly) on x via its dependence on y . (Implicit dependence is much easier to control.) The replacement is later undone in the solution. If any dependent variable depends explicitly on different arguments in different places in the ODE then this is currently trapped as an error; later it will be used as an interface to a solver for differential-delay equations (probably using the Laplace transform package).

If any dependent variable does not already depend (implicitly) on x then this dependence is automatically declared and a message to this effect output. Only now is the first argument, `ode`, fully evaluated. Earlier in the interface routine it is evaluated partially, treating the derivative operator as inert. (This is easy to do, because the code that implements differentiation is attached to the operator name `df` via its `simpfn` property, which can easily be changed. Provided the partial evaluation is performed with REDUCE’s internal simplification caching turned off, `ode` can later be re-evaluated with differentiation active.) However, this automatic dependence declaration mechanism fails if `ode` is evaluated outside the context of `odesolve`, such as happens if the first argument to `odesolve` is an assignment (which is allowed in REDUCE).

Conditions can be imposed on the solution of a single ODE by supplying suitable additional arguments. (There is not yet any support for imposing con-

ditions on systems of ODEs.) The principal mechanism for specifying conditions is to give an optional argument that consists of a list of lists, each of which contains a number of equations with x , y or a derivative of y on the left and a constant on the right. For example, Wester's [24] test example 110, namely

$$d^2y/dx^2 + k^2y = 0, \quad y(0) = 0, \quad y'(1) = 0,$$

can be entered as

```
odesolve(df(y,x,2) + k^2*y = 0, y, x,
  {{x = 0, y = 0}, {x = 1, df(y,x) = 0}});
```

Each inner list is treated as a set, which *must* contain precisely one equation for x and not more than one equation for y and each derivative, giving its value at the specified value of x . This allows boundary and/or initial conditions to be specified, but only on y or *single* derivatives separately; specification of more general conditions may be supported later. The optional condition argument must come after any optional arguments specifying the dependent or independent variables. Simple boundary conditions can alternatively, or additionally, be specified by giving the dependent and independent variables explicitly in the form of equations with the variable identifiers on the left and single values or lists of matching values on the right.

Any remaining arguments are treated as options, which currently are restricted to keywords that control the operation of ODESolve. There are a number of REDUCE switches, with names of the form `odesolve_option`, that can be turned on and off to control ODESolve globally. These switches can alternatively be turned on locally by specifying the keyword *option* as a final argument to `odesolve`. In a later version it is intended that options will exist to call external packages, such as a numerical solver, in which case arguments after the option name will be passed on to the external package. (It should perhaps also be possible to control algorithmic tracing locally.)

The interface routines establish a suitable environment for the rest of the package by enforcing various switch settings internally. However, the code is very careful not to disturb its global environment, and so avoids using the standard `on` and `off` commands. ODESolve resimplifies its return value in the global context if necessary. This has the slightly odd consequence that tracing output may have different formatting from the final return value. ODESolve currently uses the default number domain, but later it may control that internally also.

Finally, the interface code attempts to make all solutions explicit if this option was requested. This is done after imposing any conditions because explicit solutions may be much more complicated than implicit solutions from which they were computed, so it may be much more difficult to impose conditions on the explicit forms. However, it is possible that not all branches of an explicit solution satisfy the conditions imposed on the implicit solution [19, example 48], so this is re-checked, which is quite easy and fast since the solution is now explicit. (It is interesting to note that different solution techniques can produce different implicit solutions to the same problem, caused by different spurious factors.)

The ODESolve 1+ interface is (intended to be) fully compatible with that provided by ODESOLVE 1, and to that end a symbolic-mode interface routine called `odesolve` is also provided that takes precisely three arguments for the

benefit of any external programs that call `odesolve` in symbolic mode. (ODE-SOLVE 1 provided exactly the same interface to both symbolic and algebraic modes.)

3.1.1 Imposing conditions

If there are no conditions then `odesolve` calls `odesolve!*0` (defined in module `odegenr1`); otherwise it calls the routine `odesolve!-with!-conds`, which then calls `odesolve!*0`. This is done because `odesolve!-with!-conds` needs access to the indices of the `arbconst` operator used in the general solution, which it finds by examining the values of the (global) index variable before and after computing the solution. The condition code assumes that solutions are implicit. An explicit solution is acceptable as a special case, but the code does not take advantage of the fact that a slightly simpler algorithm could be used – the gain in speed is probably not worth the extra code. (The condition code does not yet accept parametric solutions.)

The interface routine checks the consistency of any conditions and regroups them into a single list of lists, each containing one equation specifying an x value followed by one or more equations specifying values for y and its derivatives, sorted by increasing order of derivative.

The strategy for imposing conditions is, at each x value and for each derivative on which a condition is specified, to differentiate the solution equation the required number of times (using the REDUCE 3.6 `map` operator), substitute into it the current condition and also the conditions on all lower-order derivatives, and finally substitute any condition on y itself and the condition on x . This should give one algebraic equation per condition for the arbitrary constants; if any derivatives of y remain then the attempt to impose conditions directly on an implicit solution has failed, which can happen only if there are “gaps” in the sequence of derivatives on which conditions are imposed, e.g. if a condition were imposed on y' but not on y . I currently believe that this is unlikely to happen in practice. If it did happen then the code could try to solve for any missing derivatives, or it could try to compute explicit solutions before imposing the conditions, and some such strategy may be added later. (It should be possible to use whatever conditions are specified to return a solution that still contains some, but fewer, arbitrary constants, although the present code is not designed to support this.)

Arbitrary (algebraic) conditions on y and its derivatives at each specified value of x could be accommodated fairly easily by a small generalization of the algorithm sketched above, but in general a set of coupled polynomial equations would result, from which y and its derivatives would need to be eliminated to leave a set of equations in the arbitrary constants alone. The condition code should at least allow conditions to be imposed on linear combinations of y and y' , which is important for some second-order ODEs that arise from physical problems. I have already implemented (in prototype) some rather similar code intended to check solutions of an ODE by using the REDUCE GROEBNER package to perform the elimination, which seems to work.

3.2 The general transformation module (odegenrl)

The main entry point to this module is the routine `odesolve!*0`, which is the only routine in this module called by the user interface module. This routine initiates a sequence of simplifications that ends with a call of the routine `odesolve!*1`. This routine attempts to solve the simplified ODE directly by calling `odesolve!*2`, which is essentially the top-level routine from ODESOLVE 1. If this fails then various transformation techniques are tried.

Currently, the simplification routines are not called directly, but via a small set of general routines that call the simplification routines named in a list in the order specified. This makes it much easier to experiment with changes to the sequence of simplification routines without the need to edit a large number of calls. (A similar technique is used in the latest version of `CRACK`.)

The strategy used in ODESOLVE 1 was to convert an input equation into an expression and determine its type once and for all, both of which were effected by the routine `sortoutode`, which assigns the type information to global variables and returns the ODE as an expression. The re-entrant nature of ODESolve 1+ requires a more general strategy. Several solution techniques require the solution of a simpler ODE to be converted back to another ODE and re-solved, and since ODE solutions are always returned as equations (which are convenient for use in substitutions) it is necessary to convert equations to expressions in several places. Therefore, many routines accept an ODE as either an equation or an expression. Similarly, the type of ODE being solved at various points varies and so needs to be determined locally rather than globally. In fact, a compromise between purely global and purely local determination of ODE type should be possible, perhaps by making the type information part of an extended ODE data structure and transforming it as the ODE is transformed, and this improvement will be considered later.

However, better use could be made of the primary classification of ODEs as first or higher order and linear or nonlinear, as was done in ODESOLVE 1. First-order linear ODEs are trivial, leaving three non-trivial classes of problems. Among these, for example, Lie symmetry techniques are useful *in practice* only for higher-order nonlinear ODEs [12, 16, 23]. (This observation probably dates back to Lie himself! [14]).

The routine `odesolve!*0` initializes the whole solver and *must not be called recursively*. It then enters the simplification sequence. If that does not lead to a solution then it calls the exact ODE solver on the original input ODE before returning failure, in case the simplification has destroyed exactness. Since exactness is “non-generic”, this call is made last rather than first, so that it imposes an overhead, albeit small, only for ODEs that cannot be solved. (It should perhaps be called only if simplification has actually changed the ODE.) It is included mainly to meet user expectations in artificial tests; in principle all exact ODEs should probably be solved as special cases of the use of integrating factors.

3.2.1 The simplification sequence

The first simplification attempts to factorize the ODE algebraically over the integers. It discards any repeated factors and any factors that do not depend

(explicitly) on y ; unless they are simply numerical a warning message is output.³ Note that this is not the same as factorizing the differential operator, a technique that is not currently implemented (except in a sense in the special cases of exact and autonomous ODEs and the following trivial case).

The second simplification determines the lowest-order derivative present, treating y itself as the zeroth-order derivative. If the lowest-order derivative is not y then the routine replaces it by y , attempts to solve the resulting ODE, replaces y in the solution by the true lowest-order derivative and attempts to solve the resulting lower-order ODE. This reduction is not performed if it would produce a purely algebraic equation at any stage.

The third simplification looks for a shift of the independent variable that would simplify the ODE. A shift is very easy to handle because it does not change any of the derivatives, although in the longer term it would be worth considering more general transformations. (This is not the same as looking for symmetry transformations [23]; the transformations considered here are intended to simplify the ODE rather than leave it invariant.) The current strategy is to consider the coefficient function of each derivative in turn, by decreasing order. When one is found that is a linear polynomial or a (polynomial) function of a linear polynomial (found using the standard REDUCE operator `decompose`), the ODE is shifted, solved, and the solution shifted back (taking care to keep x isolated on the left in parametric solutions). This simplification strategy is heuristic; there is no guarantee that it will make an ODE easier to solve, but in practice it seems to work in all the examples that have been considered so far. Since a shift is a very mild transformation there is currently no facility to backtrack out of it. (However, it *could* make a sparse high-degree polynomial coefficient function significantly more complex, and a more sophisticated and careful algorithm is really needed!)

3.2.2 The solution sequence

The main simplification sequence ends with a call of the routine `odesolve!*1`. This routine first checks whether it has seen the ODE before. If it has then a recursive call somewhere has generated an endless transformation loop which is broken at this point. One must be careful where to insert this loop check so as to detect loops as early as possible. The ODE should be in a canonical form before it is checked, and the previous three simplification routines help to ensure this. (Further canonicalization may be necessary, such as canonicalization of the overall sign, which is currently not done here.) The interrupt routine works by building a list of all the ODEs that it has seen during one call of the top-level solver (`odesolve!*0`) and comparing each new ODE against the list. For this, it further canonicalizes each ODE by converting it to true (Lisp) prefix representation and replacing the variables by standardized variables (each of which is a Lisp “gensym” generated when the package is loaded). This variable canonicalization is necessary because some ODESolve 1+ routines generate new ODEs with new (gensym) variables.

After ensuring that the solver is not looping, `odesolve!*1` calls procedure `odesolve!*2`, which is an enhanced version of ODESOLVE 1. If this fails, it then calls a sequence of routines that try either to solve the ODE directly or

³Discarding *may* not be appropriate, such as after an interchange of variable roles!

to further transform it into a solveable form, until one of the routines succeeds. The first routine in this sequence is the exact ODE solver in module `odespecl`, described below. (Note that this routine may be called again by `odesolve!*0` on the original *unsimplified* ODE if all other attempts to solve it have failed.)

The next routine differentiates the ODE and if the derivative factorizes then attempts to solve each factor. This technique could be considered to be an inverse of (partially) solving an exact ODE by integrating once, but it is essentially heuristic and its use is experimental. If a factor is purely algebraic then it is retained provided it satisfies the undifferentiated ODE; if a factor represents a differential equation then `odesolve!*2` is called to try to solve it. (Calling this low-level solver is a temporary expedient to avoid deep or indefinite recursion.) It is then necessary to ensure that each solution returned by `odesolve!*2` satisfies the undifferentiated ODE, which generally requires solving for one of the arbitrary constants. It is tricky to implement this solution strategy generally, and the current implementation is not complete. Experiment suggests that this strategy is unstable, in the sense that there is no *a priori* way to detect suitable candidate ODEs and it can take a great deal of computation before it finally fails on inappropriate examples, although it works well in some cases.

The next sequence of routines detects special types of ODE – the code is in module `odespecl`. If they all fail then a further attempt is made to algebraically factorize the ODE, but this time allowing restricted algebraic extensions of the underlying polynomial ring. More precisely, if the ODE involves only a single-order derivative and is not simply a linear polynomial in that derivative then the routine tries to solve for it algebraically. If this succeeds then it results in a “more linear” ODE that does not involve powers of the single derivative and so may be solvable. In practice, this strategy works well in some simple cases, but it seems best to try most other techniques first that do not introduce extension rings.

If all attempts at solution so far have failed then the roles of the variables are interchanged so that the ODE for $y(x)$ is transformed into an ODE for $x(y)$ and the whole simplification process is restarted. This sometimes leads to a solution, but the transformation is an involution, and if it does not lead to a solution then it is an obvious source of infinite recursion!

3.2.3 Roots of unity

There are currently two situations in which `ODESolve 1+` may need to solve an algebraic equation of the form $z^n = c^n$ for z : as a special case of the algebraic factorization over an extension ring described above, and when returning explicit solutions of first-order ODEs of Bernoulli type. The standard `solve` operator does not currently express the solution compactly in terms of symbolic roots of unity ($\exp(2\pi mi/n)$, $0 \leq m < n$) unless n is symbolic; in other cases it either expands the solution into a list of explicit solutions or returns a solution involving the `root_of` operator. It is convenient to treat all these cases uniformly, both to retain the essential structure of the solutions as clearly as possible and to minimize the apparent complexity. I have therefore introduced a symbolic `root_of_unity` operator. This operator takes two arguments: the first is n , indicating that it represents an n^{th} root of unity, and the second is a unique tag to show which roots are related, as used by `root_of` for the same reason. The common special case `root_of_unity(2,tag)` is represented as

`plus_or_minus(tag)`, and the two representations are treated identically. The use of the symbolic `root_of_unity` operator is particularly important in the solutions of ODEs that have been generated internally and which will generate new ODEs to be solved: it avoids solving essentially the same ODE more than once.

An operator `expand_roots_of_unity`, with `expand_plus_or_minus` as a synonym, is provided to expand an expression containing a `root_of_unity` operator into a list of expressions in each of which the symbolic root of unity takes one of its possible explicit values. (It is analogous to the standard operator `expand_cases` in the `solve` package.) Expansion is currently effected by a simple recursive multi-pass linear search-and-replace operation until there are no symbolic roots of unity left. It could be done more efficiently in a single pass at the expense of slightly more complicated code, but the current inefficiency does not seem to be significant. (Re-evaluation of any symbolic integrals can be much more significant!) There should perhaps be an option to expand roots of unity automatically in solutions returned by `ODESolve`.

3.3 The special solution module (`odespecl`)

This module provides a number of routines that are called from module `odegenr1`, primarily by routine `odesolve!*1`, to try to either solve or simplify the following special types of ODE.

3.3.1 Exact ODEs

An exact ODE is one that has the form $d\phi(x, y)/dx = 0$, where the function ϕ may itself involve derivatives. An exact ODE solver is provided that currently calls one of three separate routines to handle general first, general second and *linear* higher order ODEs. I propose later to merge the techniques used in these routines to provide a completely general exact ODE solver. However, first and higher-order exact ODEs are inherently different in that an exact first-order ODE can be solved directly, whereas exactness of a higher-order ODE leads only to a first integral. There is no guarantee that the first integral ODE can itself be solved, in which case an alternative solution strategy may produce a better solution. The higher-order exact ODE solvers call the general ODE simplifier hierarchy to attempt to solve the first integral. The routines all test both the numerator of the ODE expression alone and the complete ODE for exactness. Apart from that addition, the first-order exact ODE solver is essentially the same as that in `ODESOLVE 1`.

3.3.2 Special cases of Lie symmetry

The remaining transformation techniques in this module are special cases of Lie symmetry methods [23].

An *autonomous* ODE is one that involves the independent variable x only via the dependent variable y . Formally, it is invariant under a transformation of the form $x \rightarrow x + a$. It can be reduced to an ODE of order one lower by regarding y as the new independent variable and y' as the new dependent variable. If the reduced-order ODE can be solved then its solution leads to a first-order ODE that must still be solved; hence this reduction is not useful for

a first-order ODE. It is also probably not useful for linear ODEs – certainly not if they have constant coefficients.

An ODE is *equidimensional in y* if it is invariant under a transformation of the form $y \rightarrow ay$. In this case a transformation of the form $y \rightarrow e^u$ simplifies it by removing any nonlinearity caused by y appearing explicitly in coefficient functions, at the expense generally of introducing new powers of derivatives. A linear ODE may be trivially equidimensional in y but the transformation generally makes it nonlinear, so linear ODEs are not treated as equidimensional in y .

An ODE is *equidimensional in x* if it is invariant under a transformation of the form $x \rightarrow ax$. In this case a transformation of the form $x \rightarrow e^t$ removes x from the coefficient functions to leave an autonomous ODE, which can immediately be reduced to an ODE of lower order as described above. The linear case needs to be treated separately for two reasons. Firstly, the transformation produces a linear ODE with constant coefficients, which is not usefully regarded as autonomous. Secondly, it is generally advantageous to make the transformation if the ODE apart from terms independent of y is equidimensional, because there are simple algorithmic solutions of linear constant-coefficient ODEs with terms independent of y . Linear equidimensional ODEs are said to be of Euler type and ODESOLVE 1 provided a routine for solving these which is used. Euler equations are detected before equidimensional equations.

An ODE is *scale-invariant* if there exists a constant p such that it is invariant under a transformation of the form $x \rightarrow ax, y \rightarrow a^p y$, which is a generalization of equidimensionality. Then the transformation $y \rightarrow x^p u$ reduces it to an ODE that is equidimensional in x , which can be solved by calling the routine described above. The code for solving scale-invariant ODEs is new and not yet fully tested. It may be sensible to merge the code for solving, or at least detecting, equidimensional and scale-invariant ODEs, but it will require some experimentation to determine whether a merged general routine is more or less efficient than several special routines.

3.3.3 ODEs satisfied by “special functions”

The REDUCE 3.6 SPECN package supports quite a large set of “special functions”, most of which are solutions of particular ODEs. My intention is that ODESolve 1+ will be able to solve all such ODEs and minor perturbations of them, by which I mean ODEs satisfied by special functions with arguments of the form $ax+b$. This problem is unavoidably one of pattern matching: searching for ODEs in a table of ODEs and their independent solutions. Unfortunately, this requires rather large tables, especially because it may be necessary to include a large number of special cases explicitly (although it ought to be possible to compress some of these using the “double tilde” match operator introduced in REDUCE 3.6 [8]). However, the table size can be reduced by handling the perturbations algorithmically. It may also be advantageous to use a hierarchy of tables distinguished by properties such as the order of the ODE or the nature of the coefficient of its highest-order derivative. I have experimented with code to perform such pre-discrimination of ODE types, but it is not included in the current version of ODESolve 1+.

The current implementation will solve ODEs satisfied by Bessel functions (J and Y) and modified Bessel functions (I and K) of any order, and Airy

integral functions (Ai and Bi). Inclusion of other special functions primarily requires extension of the tables and should be straightforward. Translation of the arguments is handled by the general ODE simplification code, but rescaling of the arguments is handled by specific patterns. The pattern matching is done using standard REDUCE rule lists. At present these are (partly) constructed dynamically to have the same variables as the input ODE, which avoids one source of ambiguity in the matching, but it might be better to canonicalize the actual ODE variables instead, as in the loop interrupt routine described earlier. The input and match ODEs are each made the single argument of a dummy operator, which forces the ODE to be matched entirely. The right-hand side of each match is a list of independent solutions as the argument of a procedure. This avoids having a naked list on the right of a rule (which is currently invalid) and the procedure is used to convert the list into a linear combination with `arbconst` coefficients.

The pattern matching assumes that the ODE to be matched has been canonicalized at least to remove any common factors and “overall” sign. Common factors are removed by the general ODE simplification code, but (as a temporary expedient) the sign is currently canonicalized by applying the standard REDUCE operator `abs`. This remains symbolic when applied to a non-numeric argument, but it nevertheless simplifies its argument, which is then extracted from the operator.

3.4 The original ODESOLVE modules

The original modules to solve first-order ODEs (`ode1`) and linear ODEs with constant coefficients (`lccode`) have been modified; the modules to determine the type of an ODE (`testdf`) and to solve Euler-type ODEs (`linearn`) currently remain unchanged (except for minor changes to the tracing).

3.4.1 Modifications to the original ODESOLVE

The original code to detect and solve ODEs of Bernoulli type ($dy/dx = P(x)y + Q(x)y^n$) was restricted to (explicit) integer values of n . The new Bernoulli solver works (I believe) for arbitrary n and coefficient functions (P, Q) that depend explicitly or *implicitly* on x . This is slightly tricky, because the REDUCE representation of y^n depends on the form of n : any explicit integer component is extracted and used in a polynomial representation as a numerator or denominator, whilst the remainder is used in a symbolic representation as a “kernel” that forms a factor of the coefficient of the numerator polynomial. Thus if $n = n_i + n_f$, where n_i is the explicit integer part of n and n_f is either fractional or symbolic, then the general form of the representation is essentially $\text{expt}(y, n_f) * y^{n_i}$ if $n_i \geq 0$ or $\text{expt}(y, n_f)/y^{-n_i}$ otherwise. The possible forms of the sum $P(x)y + Q(x)y^n$ are a bit more complicated; it cannot in general be treated simply as a polynomial or even a rational function in y .

ODESOLVE 1 took a very strict view of the meaning of “exact” differential equation, namely an exact differential form equated to zero. It contained code to solve an exact first-order ODE of the form $N(x, y)dy/dx + M(x, y) = 0$ if $N(x, y)dy/dx + M(x, y) = d\phi(x, y)/dx$. The revised code will also detect and solve this ODE if it is presented in the form $dy/dx + M(x, y)/N(x, y) = 0$. The differential form in the latter is strictly not exact, and becomes exact only after

multiplying by the integrating factor $N(x, y)$. However, it is a useful and simple heuristic, which meets users' likely expectations, to try the denominator as an integrating factor, and I do this in my routines for solving "exact" (or perhaps "nearly exact") ODEs.

It is important that an ODE solver does not overlook the fact that, in REDUCE, any variable can be declared to depend implicitly on any others, and if any variable other than x or y that appears in an ODE has been declared to depend on x or y then it must be regarded not as a symbolic constant but as a symbolic *function*, which can change the apparent type of the ODE. I have modified the code that detects a separable ODE to avoid detecting a Bernoulli equation, of the form $dy/dx = Py + Qy^n$ where P, Q depend *implicitly* on x , as separable when it is not (unless $n = 1$). However, I doubt that either the old or the new code is completely reliable in this respect. This raises a question about how far along a chain of implicit dependence one should look. It could be very slow to follow all such chains to their ends, yet not to do so could lead to incorrect solutions. The differentiator follows all dependence chains to their ends, whereas the advertised predicate `freeof` only checks immediate dependence and does not follow dependence chains at all. ODESolve 1+ is currently rather inconsistent about this, and it is an issue that needs to be resolved. (Dependence predicates are discussed briefly in [15, appendix D.2]; however, ODESolve 1+ uses the internal REDUCE predicates `smember` and `depends` rather than the expository code presented there.)

When replacing complex conjugate pairs of exponentials by sines and cosines in the complementary solution of a linear ODE with constant coefficients, the original code relied on an assumption about the order in which the REDUCE `solve` algebraic equation solver returned solutions, which is no longer valid. It also assumed that the constant coefficients would always be purely real so that complex conjugate roots of the auxiliary equation would always be paired. The revised code does not rely on either of these assumptions, and explicitly matches complex conjugate pairs of roots of the auxiliary equation, taking account of their multiplicities, to return solutions that are "as manifestly real as possible". Complex numbers in REDUCE are treated as single numbers or as polynomials in i depending on whether the `complex` switch is on or off respectively. The original code assumed the latter, which is the default, whereas the revised code works in either mode.

3.4.2 Additions to the original ODESOLVE

First-order ODEs are a natural special case; I have followed the philosophy of ODESOLVE 1 and added routines for solving the following special types of first-order ODE to the original module `ode1`. They can all be recognised by testing for the appropriate polynomial structures.

ODEs of the form $x = f(y, y')$ or $y = f(x, y')$ are called "*solvable for x* " or "*solvable for y* " respectively. They can be recast as first-order ODEs for the derivative $p = y'$, which may be easier to solve than the original ODE; if they can be solved they give parametric solutions of the original ODEs in terms of the parameter p . The general first-order ODE solver is currently called recursively via `odesolve!*2`, and an internal switch is used to prevent looping by allowing only one recursive call. An ODE of Lagrange type has the form $y = xF(y') + G(y')$, which is a special case of the "solvable for y " type. It

has the advantage that it leads to a *linear* first-order ODE for $x(p)$ which has a simple algorithmic solution (using an integrating factor).

An ODE of Clairaut type has the form $f(xy' - y) = g(y')$, of which the general solution is $f(xC - y) = g(C)$ where C is an arbitrary constant. It is also easy to construct a singular solution to this type of ODE if it has one. Clairaut ODEs are currently recognised as a special case of the “solveable for x or y ” type for convenience, even though the actual solution technique is different.

An ODE of Riccati type has the form $y' = a(x)y^2 + b(x)y + c(x)$, and can be transformed into a *linear* second-order ODE, which may be easier to solve than the original ODE. This is currently the only type of first-order ODE recognised by ODEsolve that involves higher-order ODEs in its solution, and so requires a call of another module. It is possible for this solution technique to lead to infinite recursion, because the second-order ODE may be reduced back to the original first-order ODE, but the general interrupt routine described earlier will detect and prevent this.

3.4.3 Remaining problems with the original ODESOLVE

As explained earlier, ODESOLVE uses a method to compute a particular integral for a linear constant-coefficient ODE of order n that involves an n -fold multiple integral. This method was chosen out of four possible techniques that were tried, because experience suggested [14] that it frequently gave the result in a more satisfactory form; in particular it gave the least obvious duplication with the complementary function. It is also faster because it avoids the need to compute the Wronskian in the method of “variation of parameters” [12]. However, there are cases (e.g. [19, equation (12)]) where an inner integral cannot be evaluated, and in such cases, whilst formally correct, the result is not very satisfactory. (The solution can also be unnecessarily slow – see below.) In such cases, it is quite possible that another technique, such as “variation of parameters”, might lead to integrals that can be evaluated. It would therefore be useful to implement (or restore previous implementations [12] of) other such techniques. They could be selected under user control, although I believe that a selection strategy that is at least semi-automatic could be implemented that aims to minimize the number of unevaluated integrals, which should give better results.

4 Simplifying arbitrary constants and parameters

One of the design goals of ODEsolve 1+ is to return solutions in a convenient format, as close as possible to what a mathematician would produce by hand. A simplification that is desirable in this context is to replace functions of arbitrary constants by new arbitrary constants. Functions of arbitrary constants arise unavoidably when an ODE is decomposed in some way into a lower-order ODE whose solution represents another ODE (e.g. when the differential operator is factorized in some way). This latter ODE will itself contain arbitrary constants, so its solution will generally contain functions of these arbitrary constants. If an arbitrary constant appears only once in a solution then the largest sub-expression containing that arbitrary constant and other fixed constants can

be simply replaced by the arbitrary constant itself (to avoid generating a new arbitrary constant). However, if the arbitrary constant appears more than once then it is necessary to replace the largest *common* sub-expression that contains that arbitrary constant and other fixed constants. This is not trivial. It is related to the problem solved by optimising compilers, and it is discussed in the context of REDUCE in [9]. (It is possible that the SCOPE package [8], or at least ideas used in it, could be relevant here.)

My current code to perform the required simplification of arbitrary constants is partly successful. It uses the standard REDUCE operator `structr` to determine the common sub-expression structure of a solution. It then rebuilds the solution except for any sub-expression that is a candidate for replacing by a single arbitrary constant. This approach has been a useful first step, but it is not currently completely reliable. Moreover, it is necessary to consider more subtle sub-expression structure than that picked up by `structr`, which will probably require special-purpose code perhaps based on `structr`. The arbitrary constant simplifier is applied to any solution of an ODE that was itself constructed from an ODE solution, namely when solving exact or autonomous ODEs or after the trivial order-reduction simplification.

A related simplification problem arises with parametric solutions. The parameter typically corresponds to y' , but if (as is likely) this correspondence is not important, and provided one is happy to allow the parameter to take arbitrary complex values, then a parametric solution can sometimes be simplified considerably by replacing common sub-expressions that depend only on the parameter and fixed constants by the parameter alone. This simplification is currently effected by code similar to that used for arbitrary constants, with the exception that it is necessary to consider the intersection of the structure sets of the two expressions that represent x and y parametrically. In some cases, the parametrization is also simplified considerably by replacing the parameter by its reciprocal. Unfortunately, the approach currently used works well sometimes, but in other cases it increases the complexity, and it requires further work to determine what simplification to make when. For that reason it is currently turned off by default.

Simplification of parametric solutions is important not only for aesthetic reasons, but also when it is desired to eliminate the parameter to give an implicit or fully explicit solution. When the parametric solution is polynomial in the parameter then the resultant⁴ can be used to eliminate the parameter. (The result benefits from a factorization to remove irrelevant factors.) Otherwise, the only option is to try to solve one of the equations for the parameter and substitute it into the other equation, which can be very messy. Simplification of the parametric solution can make it much easier to eliminate the parameter, especially if it produces a polynomial parametrization.

⁴The standard REDUCE `resultant` operator does not check that its arguments are actually polynomial in the specified variable, and so if used incautiously it can give erroneous results!

5 Problems with the integrator; the patch module (odepatch)

The patch module is used for corrections and extensions to REDUCE that are not specific to solving ODEs. They mostly relate to the integrator, and a number of problems that manifested themselves as errors in ODESOLVE 1 were in fact errors in the integrator that have now been corrected. Some patches have been transferred into the standard REDUCE 3.6 patch file (and more into the development version of REDUCE); what currently remains is primarily a rule-list correction that deals with integrals that lead to error functions. ODESolve 1+ should be run only under REDUCE 3.6 with the latest patch file.

There are a number of situations in which solutions of ODEs can be returned only in terms of integrals that cannot be evaluated in closed form. In some of these cases the standard REDUCE integration operator `int` can be very slow to return essentially unevaluated. It seems to be trying too hard! In some cases, it is simply trying a lot of strategies, none of which work, and it is not clear how to avoid this in general. As a temporary measure, I have provided a switch `odesolve_noint` that turns off evaluation of the integrals arising in certain solution algorithms, in particular the solution of linear first-order ODEs. But this is not a satisfactory resolution! (Becken [4] currently uses an inert operator instead of `int` for the same reason.)

However, a similar problem that I believe *can* be satisfactorily resolved involves multiple integrals that cannot be evaluated because an inner integral cannot be evaluated. In this situation, the outer integrations cannot be evaluated beyond performing some minor simplifications, but the integrator currently tries nevertheless. It can try to re-evaluate the inner integral a prodigious number of times, as is clearly shown by tracing the integrator. This can be prevented by replacing any unevaluated integrals in an integrand by some inert operator, integrating the result, and then undoing the replacement, being careful to avoid further algebraic evaluation. This allows the integrator to perform its normal simplifications (removing constant factors, etc). The modification is effected by a patch routine that is installed when ODESolve 1+ is loaded. It can be dynamically uninstalled or reinstalled under the control of a switch called `NoIntInt`, which is on by default. This modification could probably safely be incorporated into the standard integrator. It speeds up some symbolic multiple integrals considerably. Note that it needs to be effective outside the context of ODESolve so that solutions returned by ODESolve can be re-evaluated if required without undue delay.

6 Systems of ODEs

The ODESolve 1+ user interface support for systems of ODEs is essentially complete, but it currently calls a dummy routine. I have written some code that solves simple triangular and linear systems, but my intention is primarily that ODESolve will apply `CRACK` [27] to systems of ODEs. However, it may be advantageous for ODESolve to handle the simpler cases itself. There is also still a need to implement code to impose conditions on solutions of systems of ODEs.

A small enhancement to REDUCE that relates primarily to systems of ODEs

extends the syntax of the `depend` declaration. This enhancement was initially in the `odepatch` module, but is now in the standard REDUCE 3.6 patch file. It allows a list of variables all to be declared to depend on the same sequence of variables in one declaration of the form

$$\text{depend } \{y_1, y_2, \dots\}, x_1, x_2, \dots$$

Formerly, this command would have declared the list itself to depend on the specified variables, which is unlikely to be useful since normally REDUCE does not accept lists as algebraic objects. (If this declaration really is desired then the list must be put inside a list.) Dependence declarations of this form can be concatenated by following the sequence of x_i by another list of y_j and sequence of x_k , etc.

7 ODEs that depend on parameters

The problem with these is that the nature of the solution may depend on the value of the parameters, and it may be possible to satisfy imposed conditions only for certain values of the parameters, leading to eigenvalue problems. Wester's [24] test example 110, namely $d^2y/dx^2 + k^2y = 0, y(0) = y'(1) = 0$, used earlier to illustrate the syntax for specifying conditions, is a simple example. Wester claims that no system tested did better than to produce the trivial solution. Some progress has been made in handling this problem by using the fact that the standard REDUCE 3.6 `solve` operator can formally solve a parametrized system of algebraic equations and return any assumptions that it has made about the parameters, which can be used to compute the eigenvalues. However, the code for this is experimental and not yet included in ODESolve.

8 Conclusion and future plans

The currently distributed version of ODESOLVE [13] has been updated for use with REDUCE 3.6, some restrictions have been removed and some of the developments proposed in [13] have been realized. Some errors and inefficiency in the standard REDUCE integrator have been corrected or improved. The ODESOLVE user interface has been generalized and made more user-friendly, and allows conditions to be imposed on ODE solutions. A number of solution techniques have been added, primarily for nonlinear ODEs. This has increased the success rate on the single equations in the Postel-Zimmermann test set [19] (excluding systems) from about 12/48 to about 43/48 (depending on precisely how one defines "success"). (The latest version of `CRACK`, which will be called by a later version of ODESolve, performs at least as well as any CAS on the systems in this test set.)

ODESolve now needs to be tested on a larger set of ODEs. Experience suggests that the current design is rather sensitive to the order in which solution techniques are tried. Some techniques can take a long time before they fail and ODESolve backtracks, and there is a need for better early discrimination in the choice of techniques to apply to a particular ODE.

The ability of ODESolve to find singular solutions needs to be improved. At present there is explicit code for this only for ODEs of Clairaut type. Some

solution techniques, such as those that cause the ODE to factorize algebraically after some transformations, may automatically find some (perhaps partial) singular solutions, whereas a different technique that also finds the general solution of the same ODE may not find any singular solution. (Some obvious singular solutions are omitted in [19], e.g. from the solutions of equations (41) and (42).)

My intention is that ODEsolve will solve an ODE “as much as possible”. As a trivial example, if an ODE factorizes into two simpler ODEs, then it should be possible for ODEsolve to return the solution of one of them together with an indication that it cannot solve the other. If this splitting takes place after a number of transformations of the original ODE, should one return the result as a successful solution, or should one regard it as a failure and try another solution technique on the original ODE? This issue requires further consideration. Moreover, what should ODEsolve return to represent an ODE (component) that it cannot solve? Currently, it returns essentially just an unadorned ODE, analogous to what the standard `solve` operator formerly did with algebraic equations that it could not solve. But now `solve` uses the `root_of` operator, which is more consistent and makes it easier to detect the failure. Perhaps ODEsolve should do something similar.

ODEsolve is currently coded in a mixture of algebraic and symbolic modes, using whichever seemed easier. Some of the algebraic-mode code would benefit from converting to symbolic mode, which would allow finer control, particularly over algebraic evaluation of expressions, and allow some computations to be performed more efficiently. Other parts, particularly those using rule lists, are probably best left in algebraic mode.

A number of “back end” interfaces need to be implemented to solve ODEs that are not solved by the current special-case code, starting with the following: an interface to Man’s Prelle-Singer solver PSODE [16, 17] for solving first-order ODEs; interfaces to `CRACK` [27] for solving systems of ODEs – the user interface is already running – and nonlinear ODEs [25, 28].

ODEsolve currently provides no code at all specifically for solving linear ODEs with polynomial coefficients, and two of the Postel-Zimmerman examples that it cannot solve (plus others that it does not solve very elegantly) are in this class. A lot of work has been done on this and related problems over recent years [1, 2, 3, 5, 6, 21, 22, for example] and some of the algorithms developed have been implemented in `REDUCE`, so they could be incorporated into, or called from, ODEsolve. The `CATHODE LODO` primitives [30] provide a potential vehicle for implementing some of the recent algorithms. An algorithm that decomposes the ODE using recurrence equations has been developed and implemented in `REDUCE` by Olaf Becken [4] for the homogeneous case (as a program called `hloodesolve`) and generalized to the inhomogeneous case by Wolfram Koepf and Winfried Neun [10]. This seems to work well in practice and I am considering calling a version of this code from ODEsolve. (Becken’s code already calls `ODESOLVE`.)

A neat way to provide flexible “back-end” interfaces would be via a “hook” mechanism (such as that used in GNU Emacs, another large Lisp program), whereby the routines (if any) named in a list assigned to a “hook variable” are called. This allows the behaviour of some aspects of the program to be changed (by a user) without any need to change the main source code. Essentially this technique is used at present for the simplifier routines in module `odegenrl`. The current back-end interface for an ODE system solver works differently by

providing a dummy routine that is intended to be redefined.

Acknowledgements

I thank Olaf Becken, Tony Hearn, Wolfram Koepf, Malcolm MacCallum and Thomas Wolf for helpful discussions and their general support of this project.

References

- [1] S. A. Abramov, M. Bronstein and M. Petkovšek, On Polynomial Solutions of Linear Operator Equations, *Proc. ISSAC '95*, ed. A. H. M. Levelt, ACM Press (1995), 290–296.
- [2] S. A. Abramov and K. Yu. Kvashenko, Fast Algorithm to Search for the Rational Solutions of Linear Differential Equations, *Proc. ISSAC '91*, ed. S. M. Watt, ACM Press (1991), 267–270.
- [3] S. A. Abramov and M. Petkovšek, D'Alembertian Solutions of Linear Differential and Difference Equations, *Proc. ISSAC '94*, ed. J. von zur Gathen and M. Giesbrecht, ACM Press (1994), 169–174.
- [4] O. Becken, Algorithmen zum Lösen einfacher Differentialgleichungen, Rostocker Informatik-Berichte 17 (1995). Reports and source code available from <http://www.informatik.uni-rostock.de/~obecken/>. (An English version is to appear in SIGSAM Bulletin.)
- [5] M. Bronstein, An Improved Algorithm for Factoring Linear Ordinary Differential Operators, *Proc. ISSAC '94*, ed. J. von zur Gathen and M. Giesbrecht, ACM Press (1994), 336–340.
- [6] M. Bronstein, On Radical Solutions of Linear Ordinary Differential Equations, in [11].
- [7] CATHODE (Computer Algebra Tools for Handling Ordinary Differential Equations) <http://www-lmc.imag.fr/CATHODE/cathode.html>
- [8] A. C. Hearn and J. P. Fitch (ed.), *REDUCE User's Manual 3.6*, RAND Publication CP78 (Rev. 7/95), RAND, Santa Monica, CA 90407-2138, USA (1995).
- [9] A. C. Hearn, Structure: The key to improved algebraic computation, *Symbolic and Algebraic Computation by Computers*, ed. N. Inada and T. Soma, World Scientific Publishing Co. (1985), 215–230.
- [10] W. Koepf (koepf@ZIB-Berlin.DE); W. Neun (neun@ZIB-Berlin.DE): private communication (March 1996).
- [11] A. H. M. Levelt (ed.), *Cathode Workshop, Nijmegen, 9–12 January 1995*. The proceedings and related CATHODE source code are available from <ftp://ftp.inf.ethz.ch/org/cathode/>.

- [12] M. A. H. MacCallum, An Ordinary Differential Equation Solver for REDUCE, *Proc. ISSAC '88*, ed. P. Gianni, *Lecture Notes in Computer Science* **358**, Springer-Verlag (1989), 196–205.
- [13] M. A. H. MacCallum, ODESOLVE, L^AT_EX file `reduce/doc/odesolve.tex` distributed with REDUCE 3.6. The first part of this document is included in the printed REDUCE User's Manual 3.6 [8], 345–346.
- [14] M. A. H. MacCallum, private communication.
- [15] M. A. H. MacCallum and F. J. Wright, *Algebraic Computing with REDUCE*. Oxford University Press (1991).
- [16] Y.-K. Man, *Algorithmic Solution of ODEs and Symbolic Summation using Computer Algebra*, PhD Thesis, School of Mathematical Sciences, Queen Mary and Westfield College, University of London (July 1994).
- [17] Y.-K. Man and M. A. H. MacCallum, A Rational Approach to the Preller-Singer Algorithm, *J. Symbolic Computation*, to appear.
- [18] T. Mulders, Primitives: Orepoly and Lodo, in [11].
- [19] F. Postel and P. Zimmermann, A Review of the ODE Solvers of AXIOM, DERIVE, MAPLE, MATHEMATICA, MACSYMA, MUPAD and REDUCE, *Proceedings of the 5th Rhine Workshop on Computer Algebra, April 1-3, 1996, Saint-Louis, France*. The latest version of this review, together with log files for each of the systems, is available from <http://www.loria.fr/~zimmerma/ComputerAlgebra/>. Specific references are to the version dated April 11, 1996.
- [20] M. J. Preller and M. F. Singer, Elementary First Integrals of Differential Equations, *Trans. AMS* **279** (1983), 215–229.
- [21] F. Schwarz, Efficient Factorization of Linear ODEs, *SIGSAM Bulletin* **28**, No. 1 (1994), 9–17.
- [22] M. F. Singer and F. Ulmer, Liouvillian and Algebraic Solutions of Second and Third Order Linear Differential Equations, *Journal of Symbolic Computation* **16** (1993), 37–73.
- [23] H. Stephani, *Differential Equations: Their Solution using Symmetries* (ed. M. A. H. MacCallum), Cambridge University Press (1989).
- [24] M. Wester, A Review of CAS Mathematical Capabilities, *Computer Algebra Nederland Nieuwsbrief 13* (December 1994) ISBN 1380-1260, 41–48.
- [25] T. Wolf, Programs for Applying Symmetries of PDEs, *Proc. ISSAC '95*, ed. A. H. M. Levelt, ACM Press (1995), 7–15.
- [26] T. Wolf, APPLYSYM: Applying Infinitesimal Symmetries of Differential Equations, L^AT_EX file `reduce/doc/applysym.tex` distributed with REDUCE 3.6. A shorter document is included in the printed REDUCE User's Manual 3.6 [8], 193–195.

- [27] T. Wolf and A. Brand, The Computer Algebra Package **CRACK** for Investigating PDEs, L^AT_EX file `reduce/doc/crack.tex` distributed with REDUCE 3.6. A shorter document is included in the printed REDUCE User's Manual 3.6 [8], 241–244.
- [28] T. Wolf and A. Brand, Approaches to Solving Nonlinear ODEs, *Mathematics and Computers in Simulation*, Elsevier, to appear.
- [29] F. J. Wright, Teaching with Computer Algebra: the QMC experience, *Proc. Undergraduate Mathematics Teaching Conference, University of Nottingham, Sept. 1988*, ed. D. Towers. Shell Centre for Mathematical Education (1989) ISBN 0 906126 58 4, 3–15.
- [30] F. J. Wright, REDUCE Implementation of Primitives for Univariate Skew Polynomials and Linear Ordinary Differential Operators: A progress report, in [11]. A longer report and source code are also available from <http://Maths.QMW.ac.uk/~fjw/>.
- [31] D. Zwillinger, *Handbook of Differential Equations*, Academic Press (1989). (Second edition available.)