

Recognising and Solving Special Function ODEs

Francis J. Wright
School of Mathematical Sciences
Queen Mary and Westfield College
University of London
F.J.Wright@qmw.ac.uk
<http://www.maths.qmw.ac.uk/~fjw/>

Abstract

Code to recognise and hence solve the second-order linear ODEs that define the special functions of mathematical physics is being developed for the REDUCE ODE solver. It allows an arbitrary, possibly symbolic, affine transformation of the independent variable. At first sight, the problem appears to be one of pattern matching or table look-up, but simple special-purpose code written for each class of ODEs turns out to be more appropriate than use of a general pattern matcher. The technique used is successive analysis of the coefficient functions after dividing through by the coefficient function of the leading derivative. This canonicalizes the ODE and allows the ODE class and the transformation to be determined together.

1 Introduction

By the term “special function ODE” I mean an ordinary differential equation that defines one of the so-called special functions of mathematical physics (e.g. [5, 2, 1]), such as Bessel functions or classical orthogonal polynomials. These are almost always linear second-order ODEs, and at present I consider only the linear second-order ODEs listed in [1].

Such ODEs might appear at first sight to be solvable by table look-up, but it is not quite that simple. The names used for variables must be allowed to be arbitrary and textbooks do not agree on the overall sign or indeed the precise formulation of equivalent ODEs, but this can be avoided by some form of canonicalization of the ODE before it is looked up. However, special functions fall into families described by one or more parameters, which will be countable if the parameters are restricted to be rational (usually integer). Hence, any table look-up is for a family of ODEs rather than a specific ODE.

The classification of ODEs into families is ambiguous in many cases because there are multi-parameter “super-families” such as hypergeometric ODEs that include many simpler families with fewer parameters as special cases. For example, Legendre, Chebyshev and Gegenbauer (or Ultraspherical) polynomials are all special cases of Jacobi polynomials, which are themselves special cases of hypergeometric

functions. In such hierarchical cases it seems desirable to select the most specific rather than the most general class.

Some special function ODEs are invariant under rescaling or shifting of the independent variable but most are not, although the change to the form of the ODE under such transformations is essentially trivial. It therefore seems desirable to recognise classes of special function ODEs that are related by at least an *affine transformation of the independent variable*. Some such transformed ODEs are listed in textbooks, thereby giving them some kind of “standard” status, although most are not.

It would of course be desirable to allow much more general transformations, but that changes the nature of the problem considerably and is not what I want to pursue here. However, the approach described here could form part of a more general solution technique.

This paper has arisen out of work to enhance the ODE solver in REDUCE [11, 10]. The solver (`odesolve`) currently distributed with REDUCE was written a long time ago [8], [10, §8.6, page 229] and I have been developing a solver (`ODESolve 1+`) [13] that is intended to replace it. The latest release, `ODESolve 1.06` at the time of writing, is always freely available [15] and runs under the latest release (version 3.7) of REDUCE. However, in this paper I will mainly describe code under development for `ODESolve 1.07`, which is not yet released. The general structure of `ODESolve 1.06+` is described in [14], with particular emphasis on the reasons for major changes from the previous structure, which was essentially as described in [13].

My primary purpose in this paper is to compare two different approaches to the solution of special function ODEs: the use of a general pattern matcher and the use of problem-specific code. I conclude that the latter is better, at least in the context of REDUCE. I describe the classes of ODEs that the current prototype can solve and present some of the code, and compare its capabilities very briefly with those of Maple. This paper is primarily about the practical implementation rather than the theoretical algorithmic or mathematical background.

Details of the algorithms and their implementations used in computer algebra systems (CASs) are often not well documented. Hence, there is little guidance available on how to go about solving many classes of problems in a CAS that does not already contain appropriate code. For example, I understand [4, 6] that there is no published description of the special function ODE solver in Maple (although of course the source code is essentially available for inspection). There is a very useful general description [7] of the linear ODE solver

in Maple V Release 3, but it does not give any details of the special function code. Whilst a standard reference text such as [17] discusses “look up techniques” [17, §§40,41], it gives no indication of how to perform them algorithmically.

2 Significance of this Problem

By definition, special function ODEs arises commonly in mathematical physics and therefore it is important that CASs can solve them. Currently, REDUCE has some well developed packages for manipulating and numerically evaluating special functions but no code at all for solving the related ODEs, so it is important for REDUCE to add this capability. But is this problem completely solved in other CASs? I suspect a fair answer is that it is partly solved. I tried running some of my test files through Maple V Release 5.0 to see what it would make of them. Generally, it copes with Bessel and related functions well, although it does not seem to like to use the modified Bessel functions. It returns

```
> dsolve(x^2*diff(y(x),x$2) +
> x*diff(y(x),x) - (a^2*x^2+n^2)*y(x), y(x),
> output=basis);
```

$$\left[\text{BesselJ}\left(n, \sqrt{-a} x\right), \text{BesselY}\left(n, \sqrt{-a} x\right) \right]$$

whereas a user might well expect the tidier solution in terms of I and K , although in the more explicit case with $a = 1$ it does return

```
[BesselI(n, x), BesselK(n, x)]
```

It may well be a consequence of this reluctance to use I and K that, for Bessel’s equation with a Struve function as a particular integral

```
> dsolve((x+k)^2*diff(y(x),x$2) +
> (x+k)*diff(y(x),x) + (a^2*(x+k)^2-n^2)*y(x) =
> C*4*(a*(x+k)/2)^(n+1)/(sqrt(Pi)*Gamma(n+1/2)),
> y(x), output=basis);
```

Maple returns a perfectly reasonable solution, but if the sign of the a^2 term is changed then it returns a horribly complicated and very different-looking solution.

The ODEsolve 1.07 prototype solves these particular examples neatly, e.g. (with basis output format turned on globally)

```
odesolve((x+k)^2*df(y,x,2) +
(x+k)*df(y,x) - (a^2*(x+k)^2+n^2)*y =
C*4*(a*(x+k)/2)^(n+1)/(sqrt(Pi)*Gamma(n+1/2)),
y, x);
```

```
{besseli(n,a*k + a*x), besselk(n,a*k + a*x)},
struvel(n,a*k + a*x)*c}
```

Maple also solves ODEs defining orthogonal polynomials, but does not always return the solution in the most elegant form. Here are two very simple examples, picked from one of my test files essentially at random:

```
> dsolve(diff(y(x),x$2) - 2*x*diff(y(x),x) +
> 2*n*y(x), y(x), output=basis);
```

$$\left[\frac{\text{WhittakerM}\left(\frac{1}{2}n + \frac{1}{4}, \frac{1}{4}, x\right) \exp\left(\frac{1}{2}x\right)}{\sqrt{x}}, \frac{\text{WhittakerW}\left(\frac{1}{2}n + \frac{1}{4}, \frac{1}{4}, x\right) \exp\left(\frac{1}{2}x\right)}{\sqrt{x}} \right]$$

```
> dsolve((x^2-1)*diff(y(x),x$2) +
> (alpha-beta+(alpha+beta+2)*x)*diff(y(x),x) -
> n*(n+alpha+beta+1)*y(x), y(x), output=basis);
```

```
[LegendreP(n + 1/2 alpha + 1/2 beta,
1/2 sqrt(beta^2 + 3 alpha^2), x)
(- 1/2 alpha) (- 1/2 beta)
(x - 1) (x + 1) ,
LegendreQ(n + 1/2 alpha + 1/2 beta,
1/2 sqrt(beta^2 + 3 alpha^2), x)
(- 1/2 alpha) (- 1/2 beta)
(x - 1) (x + 1) ]
```

The output has been reformatted slightly by hand to fit the page.

The ODEsolve 1.07 prototype solves these examples rather more neatly, e.g. (with basis output format turned on globally)

```
odesolve(df(y,x,2) - 2x*df(y,x) + 2n*y, y, x);
```

```
{hermitep(n,x)}
```

```
odesolve((x^2-1)*df(y,x,2) +
(alpha-beta+(alpha+beta+2)*x)*df(y,x) -
n*(n+alpha+beta+1)*y, y, x);
```

```
{jacobip(n,alpha,beta,x)}
```

However, these ODEs have only one “neat” solution. In such cases the second solution is computed by another module, as described below – only the solutions shown above are computed by the special function solver itself.

It is reasonable for the solution of a complicated problem to be complicated, but one of my design goals is that users should get the form of solution that they expect. In the case of an ODE that defines a classical orthogonal polynomial the expected solution is likely to be that polynomial, i.e. the solution returned should have the most specific form possible. It may be that it would be hard to make the algorithm used by Maple meet this requirement, whereas mine was designed from the beginning with this in mind.

However, my purpose here is neither to criticize Maple nor to attempt any detailed comparison, but only to indicate that the problem is not completely solved and there is scope for considering alternative approaches.

3 Internal Representation for Linear ODEs

It is necessary to outline the internal representations used for linear ODEs in ODEsolve 1+ in order to set the scene for

the algorithms described below. These use lists, rather than the algebraic expressions used in previous versions, for both input and output.

One of the major changes in `ODESolve 1.06` (the latest released version) was to allow the option to output the solution of a linear ODE in the form of a list instead of a linear combination of basis functions. I therefore changed the code to use the basis representation internally, which is generally more convenient and efficient, and a linear combination is now normally constructed only at the final output stage.

The other major change was to represent all linear ODEs internally as follows. The first step is to extract the coefficient functions of the derivatives of the dependent variable (which I will refer to as y) and the “driver”, the term independent of y . The ODE is then represented by a list of these coefficient functions, which removes any further direct involvement of the dependent variable. The independent variable (which I will refer to as x) is used explicitly in recognising the special function class of the ODE. The coefficient functions are then all divided by the “leading” coefficient, namely the coefficient of the highest (in this case second) order derivative, thereby making the representation “monic”. This removes any common factor, fixes the overall sign and canonicalizes the representation except for the name of the independent variable, which is used explicitly throughout the solution process. In fact, this pre-processing is done by the top-level routines of the linear ODE solver and is not specific to special function ODEs. The special function ODE solver receives as input an ordered list of “monic” coefficient functions, the driver term and the independent variable.

4 Pattern Matching

The difficulty in solving special function ODEs lies in recognising them, after which the solution follows trivially. The recognition problem is one of pattern matching, which is essentially a flexible form of table look-up. The flexibility is required to match variables and parameters by context rather than name or type, although the type and value of parameters may be important in determining the precise class of an ODE.

`REDUCE` makes considerable use of pattern matching in its simplifier (e.g. see [10, §2.8, page 56]), which is therefore very well developed and tested. (Some computer algebra systems, such as Maple, do not use pattern matching very much, and certainly not in the very general and automatic way that `REDUCE` does.) It therefore seemed appropriate at first sight to use the standard `REDUCE` pattern matching facilities to recognise special function ODEs. All versions of `ODESolve 1+` released so far have used explicit pattern matching to attempt to solve Airy and Bessel equations (and no other special function ODEs), allowing rescaling of the independent variable but shifting only for Airy equations. Hence, extending special function ODE solving is one of the main areas on which I have been working for version 1.07.

Explicit pattern matching turns out to be awkward to extend to larger classes of ODEs for a number of reasons. The `REDUCE` pattern matching facilities (rule lists) are intended for simplification rather than recognition. It was therefore necessary to make the recognition problem masquerade as a simplification problem. I did this by applying a set of rule lists to the ODE so that if a rule matched then the ODE was “simplified” into its solution. If the ODE simplified to

itself then no pattern had matched and so the solver failed.

This approach seemed to work reasonably well at first although a few technical tricks were necessary. The `REDUCE` pattern matcher focuses on one term in a sum, taking all other terms to the other side of the rewrite rule, which is not at all appropriate for this rather contrived simplification problem. However, this splitting can be avoided by wrapping an inert function around the ODE, which is thereby packaged as the function argument. (This is analogous to wrapping a Maple list around an expression sequence to avoid a function call seeing the elements of the sequence as separate arguments.) It was also necessary to treat the ODE variables as known rather than arbitrary match variables in the rule lists, which meant that the rule lists had to be constructed dynamically. (To be precise, they were defined within the body of a procedure whose arguments included the ODE variables, so that they were replaced in the rule list by the actual variables when the procedure was called.) This is a minor inefficiency, because `ODESolve 1+` is recursive and could in principle call this sub-solver many times in the course of solving an arbitrary ODE (especially one that it will eventually fail to solve).

With the change to list representation in `ODESolve 1.06`, the pattern matcher was required to rewrite lists, which proved impossible to do directly. This problem is overcome by converting the input list to an inert function (as described above but now for a different reason), “simplifying” one inert function to a different inert function and then converting the result back into a list, so that lists appear explicitly only outside the context of the pattern matcher. The need to do this made the code rather more convoluted than I would wish and led me to wonder whether using a general-purpose pattern matcher was the right approach.

Another problem was that it was necessary to include a number of special-case rules in addition to the most general match rule. The pattern matcher is not good at matching an arbitrary match variable to either of the explicit numbers 1 or 0, as may be required when a factor in a product or a term in a sum is missing, because the numbers 1 or 0 are implicit. (They are implied by the match pattern rather than the data structure being matched.) This capability was improved in recent versions of `REDUCE` with the introduction of the “`~`” match operator, which works well in some situations, but I was still unable to avoid the need for a lot of special-case rules for this ODE problem. When I began to extend the range of special function ODEs to be recognised and to allow the independent variable to be shifted, the number of special-case rules began to proliferate alarmingly and to become unmanageable.

This difficulty may well arise from limitations in the current `REDUCE` pattern matcher, which is not intended for such multi-variate matching; it may not be a fundamental problem with using a general-purpose pattern matcher (such as that in Mathematica) to solve special-function ODEs. Nevertheless, it does lead me to believe that this is probably not the best approach. There is another pattern matcher, the “PM” package, distributed with `REDUCE`, which was modelled on the early Mathematica pattern matcher. However, I ran into essentially the same problems with this, at which point I gave up attempting to use a general-purpose pattern matcher in favour of special-purpose code that performs more implicit pattern matching and which I wrote to solve this particular ODE recognition problem.

4.1 The Airy Equation Pattern Matcher

As an example of the general pattern-matching approach, the following code, which is explained below, is the main special function ODE solver from `ODESolve 1.06`. Note that in `REDUCE` the character “%” introduces a comment, “!” is the escape character necessary to include special characters in identifiers and the keyword “scalar” introduces local variables.

```
algebraic procedure
ODESolve!-Specfn(odecoeffs1, driver1, x);
%% Using monic coeffs for uniqueness.
begin scalar ode, rules, soln;
ode := odesolve!-specfn!(first odeccoeffs1,
                        second odeccoeffs1);

rules := {
%% odesolve(df(y,x,2) - x*y, y, x);
odesolve!-specfn!(-x, 0) =>
odesolve!-solns(Airy_Ai(x), Airy_Bi(x)),
%% odesolve(df(y,x,2) - a3*x*y, y, x);
odesolve!-specfn!(-~a3*x, 0) =>
odesolve!-solns(Airy_Ai(x), Airy_Bi(x),
                x=a3^(1/3)*x),
%% odesolve(df(y,x,2) - (a3*x+a2b)*y, y, x);
odesolve!-specfn!(-~a3*x+~a2b, 0) =>
odesolve!-solns(Airy_Ai(x), Airy_Bi(x),
                x=a3^(1/3)*x+a2b/a3^(2/3)) };

soln := (ode where rules);
if soln neq ode then <<
soln := part(soln, 1);
return if driver1 then
{ soln, ODESolve!-PI(soln, driver1, x) }
else { soln }
>>
end;
```

The first step in this procedure is to convert the representation of the ODE from a *list* of coefficient functions to an *inert function* (called `odesolve!-specfn!`) whose arguments are the coefficient functions (discarding the leading coefficient, which is always 1). The reason for this conversion is simply that the pattern matcher is intended for simplifying algebraic expressions and so does not accept lists.

The heart of the procedure is a long rule list assigned to the local variable `rules`, which does all the work by relying on the standard `REDUCE` pattern matcher. However, for purposes of exposition, I have removed the Bessel equation rules leaving only the Airy equation rules. Each rule in the list has the syntax

source pattern => target expression

An identifier preceded by a `~` operator matches anything whereas all other symbols match only themselves. The actual variable x is built into the rule list when the procedure is called, which avoids unnecessary arbitrary match variables and seems to be essential for unambiguous matching. Each rule in the list is preceded by a comment indicating the form of ODE that it is intended to recognise. The first rule recognises the standard Airy equation, which involves no parameters so the rule does not involve any arbitrary match variables. The second rule allows the independent variable x to be scaled by a quantity a , which appears as a factor a^3

in the (monic) coefficient of y and so is represented by an arbitrary match variable called `~a3`. The third rule allows the independent variable x to be scaled by a and shifted by b and so involves two arbitrary match variables.

The target of each rule is a call of the function `odesolve!-solns`, which requires two arguments (the two independent solutions). It accepts a third argument in the form of a substitution equation that, if provided, is substituted into the solution to effect any scaling and shifting of x . The solution is returned as a list packaged as the single argument of another inert function, which is extracted and returned. As before, this is done to avoid a naked list resulting from a rule-list simplification, which is invalid.

The rule-list simplification is effected by the infix operator `where`. If the ODE “simplifies” then it has been recognised and solved, otherwise the procedure returns nothing, i.e. `nil` representing false and meaning failure.

The general pattern-matching technique indicated here might be quite elegant if the most general rule discussed above covered all cases, but that would require an arbitrary match variable to be able to match an implicit 0 in a sum or 1 in a product. I have been unable to make this work reliably, even using the `^^` operator, and so it seems to be necessary to include explicit rules with 0, 1 and 2 arbitrary match variables, as described.

The point of this example is that the Airy equation is almost trivial but still requires three match rules. The standard Bessel equations, allowing the independent variable to be scaled but *not shifted*, require six rules. These rules occur in pairs, one for the normal and one for the modified equation; there is a pair for the unscaled general-order case, a pair for the scaled general-order case, and a pair for the scaled zero-order case. That was the minimum number of rules that I could get to work. Allowing the independent variable to be shifted would probably double the number of rules.

It becomes clear that recognising one of the multi-parameter families of ODEs would involve an unmanageable number of rules, which becomes difficult to write, debug and maintain, and seems unlikely to be efficient. Moreover, the need to disguise the problem as algebraic simplification just to be able to use the pattern matcher is inelegant. For comparison, the Airy sub-solver from the `ODESolve 1.07` prototype, which does not use general pattern matching, is shown below. A similar but slightly more complicated sub-solver is explained in detail in the next section.

```
algebraic procedure ODESolve!-Airy(c0, x);
%% y" + x*y
%% c0 = a^2(ax+b)
begin scalar coeffs, a;
coeffs := coeff(c0, x); % {a^2b, a^3}
if high_pow neq 1 or depends(coeffs, x)
then return;
a := (second coeffs)^(1/3);
x := -(a*x + (first coeffs)/a^2);
return {Airy_Ai(x), Airy_Bi(x)}
end;
```

Whilst this one procedure is not quite the whole story, it is most of it. The top-level procedure that calls this one has already determined that the y' term is missing and the denominator of the (monic) coefficient of y does not depend on x , and has extracted the coefficient of y to be passed to

this procedure as the argument `c0`, which all requires very little additional code.

I contend that this comparison demonstrates that use of a general pattern matcher is not the way to proceed!

5 Analysis of the ODE Coefficient Functions

My current approach, which is under development for version 1.07, is based on analysis of the coefficient functions using specific code written for each class of ODEs to be solved, although the basic approach is the same in every case. This is almost certainly more efficient than any general pattern-matching approach could be, because it can take full advantage of the precise structure of each class of ODE. The analysis is based on the algebraic structure of the ODE and hence is currently mostly implemented using the algebraic mode of REDUCE. This does not require a huge amount of code and it is much easier to debug than the previous code, because it no longer relies on a large “black box” pattern matcher. (This description is not quite fair since REDUCE comes with full source code, but nevertheless the pattern matcher is much harder to understand than the simple polynomial operations that I now use.)

The classification of special function ODEs proceeds via the (monic) coefficients of firstly y' and secondly y . The first test is whether the coefficient of y' is zero. Any non-trivial coefficient function of y'' in the original input ODE is recognised as part of the denominator of the monic coefficient functions of y' and y . The x shift is normally found first, followed by other parameters including the x scaling, as part of the classification process. Reliably extracting potentially rational parameter values from rational coefficient expressions requires care!

The solver currently recognises the following special function equation classes, although the grouping I have used here for exposition is different from that used in recognising them, for which other structural relationships are relevant:

Bessel: Airy, generalised Bessel, generalised Riccati-Bessel, Struve;

Hypergeometric: Gauss (${}_2F_1$), Whittaker and Kummer confluent (${}_1F_1$);

Orthogonal polynomial and related: Hermite, Weber-Hermite, Laguerre, associated Laguerre;

Jacobi and special cases: Legendre, associated Legendre, Chebyshev, Gegenbauer, Jacobi.

6 An Outline of the Algorithm

The top-level special function solver calls a sequence of sub-solver procedures in turn, each of which tries to recognise a particular class of ODE. If a sub-solver succeeds then it returns the solution, otherwise it returns “false” (which, since REDUCE is Lisp-based, is represented as the atom `nil`). This sequence of calls is implemented as a logical disjunction (“or”) of sub-solver predicates (as is most of `ODESolve 1+`). The structure of the top-level solver is therefore essentially

class_1.solver or class_2.solver or ...

If all sub-solvers fail then the top-level solver fails.

The structure of the top-level special function solver is explained in more detail in Table 1. This shows each class

First-order derivative present:

$y'' - 2xy' + 2ny$	Hermite polynomial
$xy'' + (m + 1 - x)y' + ny$	Laguerre polynomial
$xy'' + (\beta - x)y' - \alpha y$	Kummer confluent
$x^2y'' + xy' + (\pm x^2 - n^2)y$	Bessel / Struve
$x^2y'' + (1 - 2\alpha)xy' + (\pm\beta^2\gamma^2x^{2\gamma} + (\alpha^2 - n^2\gamma^2))y$	QuasiBessel / Struve
$(x^2 - 1)y'' + (\alpha - \beta + (\alpha + \beta + 2)x)y' - n(n + \alpha + \beta + 1)y$	Jacobi polynomial etc.
$x(x - 1)y'' + ((\alpha + \beta + 1)x - \gamma)y' + \alpha\beta y$	Gauss hypergeometric

First-order derivative missing:

Denominator of “monic” coefficient of y depends on x	
$x^2y'' + (\pm\beta^2\gamma^2x^{2\gamma} + (\frac{1}{4} - n^2\gamma^2))y$	QuasiRiccatiBessel
$y'' + (-\frac{1}{4} + k/x + (\frac{1}{4} - m^2)/x^2)y$	Whittaker confluent
Denominator of “monic” coefficient of y independent of x	
$y'' + xy$	Airy
$y'' + ((2n + 1) - x^2)y$	Weber-Hermite
$y'' - (\frac{1}{4}x^2 + A)y$	Parabolic-Cylinder

Table 1: ODE Classes Recognised

of ODE recognised in one of its conventional forms and the name for the class. This is essentially the name of the procedure that is called to recognise it, except that all classes between a pair of horizontal rules are handled by the same procedure. ODE classes are listed in the order in which they are currently tested, which can be important to avoid repeated tests. (But note that the solver actually works with the ODE coefficient functions of y' and y after dividing through by the coefficient function of y'' , and the actual name of the dependent variable y is not used.) Any symbols other than x and y represent parameters, which may be either symbolic or numeric, and if they are numeric then they may determine the class of ODE (as well as the precise member of a class), e.g. it may be significant whether they are positive or negative, and/or integer or rational.

Some of the nomenclature and choice of equations that I use is taken from [2]. Distinct sub-classes (handled by distinct sub-solvers) are separated by horizontal lines. For example, the Laguerre equation (satisfied by Laguerre polynomials) is handled as a special case of the associated Laguerre equation, which is handled as a special case of the Kummer confluent hypergeometric equation.

The Bessel equation is handled as a special case of an equation reducible to the Bessel equation [2, Theorem 4.12, page 109], which I call “QuasiBessel”. Depending on the sign of the $x^{2\gamma}y$ term, it is solved in terms of either normal or modified Bessel functions, rather than introducing the imaginary number in the latter case. The solver also recognises spherical Bessel functions, as reported by its tracing,

although it currently returns the solution in terms of Bessel functions with fractional order, mainly because the `specfn` package does not currently support spherical Bessel functions explicitly. The solver does not return Hankel functions, which provide alternative pairs of independent solutions to Bessel equations. A priori, there is no reason to prefer Hankel functions and their relevance depends on the broader context of the problem, although a user option could be added to output them. A driver term (independent of y) of the appropriate form for the QuasiBessel equation is recognised and a suitable particular integral is constructed in terms of a Struve function. A modified Struve function is used in the case of a modified QuasiBessel equation.

Legendre, Chebyshev, Gegenbauer and Jacobi equations are handled together since the former are all special cases of the Jacobi equation (hence the label “Jacobi polynomial etc.” in Table 1), and the Legendre equation is treated as a special case of the associated Legendre equation. One or two related equations listed in [1] that have simple solutions are also recognised and treated as special cases (which requires very little extra code).

The Riccati-Bessel equation [1, §10.3] is a special case of the QuasiBessel equation for which the first derivative vanishes, which I generalize as the QuasiRiccatiBessel equation. It is solved in terms of spherical Bessel functions. The Airy equation is another special case of Bessel’s equation and could probably be handled as such, although I currently treat it separately since it does not fall into the QuasiRiccati-Bessel class. The Weber-Hermite equation is very similar to the Parabolic-Cylinder equation, so I treat them both as special cases of a more general class. (The Weber-Hermite function of order n is $e^{-x^2/2}H_n(x)$, where $H_n(x)$ is the Hermite polynomial of degree n .)

Some (second-order) special function ODEs define a pair of independent standard functions (e.g. the Bessel function pairs J, Y and I, K or the Legendre function pair P, Q) that provide a natural basis for the solution space, but others, in particular most of the ODEs satisfied by orthogonal polynomials, define only one standard function. In such cases, a second “non-standard” solution is required to construct a basis (which is necessary in all except a few special cases with conditions that obviate the need for a second solution). One way to construct a second solution is to use reduction of order (e.g. [17, chapter 81]): given a solution $z(x)$ of a second-order linear ODE, seeking another solution of the form $z(x)v(x)$ leads to a first-order linear ODE for v' which can be trivially solved using an integrating factor.

If there is a driver term (that has not already been handled as a special case, such as in the case of Struve functions as particular integrals of Bessel ODEs) then variation of parameters (e.g. [17, chapter 91]) is used to attempt to construct a complete solution from the solution basis for the reduced ODE.

Neither the construction of a second independent solution nor the construction of a complete solution in the presence of a driver term is specific to special function ODEs. Both problems are handled by other independent modules of `ODESolve 1.07`, and will not be discussed further here. Whilst the techniques described for constructing a second solution or particular integral always work in principle, their efficacy in practice has not yet been well tested, but experience suggests that they do not always lead to the most tractable form of solution. The currently distributed version of `odesolve` does not use variation of parameters at

all to solve linear constant-coefficient (or any other) ODEs for precisely this reason [9], although I have re-instated it in `ODESolve 1+` and it is used automatically if the alternative “inverse D-operator” technique produces intractable integrals.

6.1 The Hermite Sub-Solver

The sub-solvers all have a similar structure and for purposes of exposition I will describe one of the simplest, that which recognises (affine transformations of) the Hermite ODE

$$\frac{d^2y}{dx^2} - 2x \frac{dy}{dx} + 2ny = 0.$$

There are essentially two ways to represent an affine transformation of the independent variable, either $x \rightarrow ax+b$ or $x \rightarrow a(x+k)$, and I use whichever seems to be more convenient, usually the latter. But in the simple case of the Hermite ODE, which is naturally monic and denominator-free, applying the former transformation leads to the more general (monic) ODE

$$\frac{d^2y}{dx^2} - 2(a^2x+ab) \frac{dy}{dx} + 2a^2ny = 0.$$

Hence the problem is to recognise this form of ODE and in the process extract the values of the transformation parameters a, b and the ODE parameter n . The ODE is represented by its (monic) coefficient functions

$$c_1 = -2(a^2x+ab), \quad c_0 = 2a^2n.$$

In this case, denominators can arise only from the transformation parameters a, b (since n must be either an integer or, as a generalization, a symbol) and so are automatically picked up from the polynomial structure of c_1 without needing any special care, which is not the case in general!

Here is the REDUCE algebraic-mode procedure that I currently use to solve this problem, which I will explain below. Recall that in REDUCE the character “%” introduces a comment and “!” is the escape character necessary to include special characters in identifiers, and note that function application can be implied by juxtaposition. The keyword “scalar” introduces local variables and the functions `traceode` and `traceode1` are part of the `ODESolve 1+` algorithm tracing mechanism, which I will not discuss further.

```
algebraic procedure ODESolve!-Hermite(c1, c0, x);
%% y" - 2x*y' + 2n*y
%% c1 = -2a(ax+b) = -2(a^2x+ab)
if den c1 freeof x then      % y''
begin scalar a;
  traceode1 "Hermite equation?";
  c1 := coeff(-c1/2, x);      % {ab, a^2}
  if high_pow neq 1 or depends(c1, x)
  then return;
  a := second c1;            % a^2
  if ODESolve!<0 a then return;
  if not ODESolve!-NonNegIntp!(c0 := c0/(2a))
  then return;
  traceode "Hermite equation.";
  a := sqrt a;
  x := a*x + first c1 / a;
  return {HermiteP(c0, x)}
end;
```

The code first checks the form of c_1 , the coefficient function of y' . This must have a denominator independent of x , which is equivalent to the coefficient function of y'' when denominators are cleared being independent of x ; if not then the procedure fails (by returning `nil`) immediately. The code then checks that c_1 is a linear polynomial in x . It does this by assuming that it is polynomial and extracting the coefficients (back into the variable `c1` – procedure arguments in REDUCE are proper local variables). Now if either the highest power of x found is not 1 or the coefficients depend on x then c_1 was not a linear polynomial and the procedure fails (by returning `nil`). Having divided out a factor of -2 , the coefficient of x (which should be a^2) is extracted into the local variable `a`. I allow only real affine transformations of x so a^2 must be positive; if it is not then the procedure fails. The variable `a` has already been checked to be non-zero, otherwise the polynomial would not have degree 1.

I want to allow the parameter a to be either numerical or symbolic, so I regard a^2 as positive if it is either numerical and positive, or symbolic and not explicitly negative, i.e. in this context I accept z^2 as positive but not $-z^2$. This definition of negativity is checked by the predicate `ODESolve!<0`. (The standard REDUCE relational operators, “ $<$ ” etc., require numeric operands.)

Having got this far, the code now divides $2a^2$ out of c_0 , the (monic) coefficient function of y , to give a candidate for the ODE parameter n . (The variable `a` still holds the value of a^2 .) If n is a non-negative integer then this really is a Hermite ODE. I also accept an identifier as a non-negative integer, in the same spirit as described above. Whilst this may end up not being strictly correct mathematically, I take the view that it is what most users (of REDUCE) would expect and prefer. This laxness could, and perhaps should, be controlled by a switch.

If the ODE is accepted as a Hermite ODE then the code computes a (as the square root of a^2) and returns the correct Hermite polynomial with its argument appropriately transformed. The special functions returned by `ODESolve 1+` are all as defined by the standard REDUCE `specfn` package [3]. The solution is returned as a list of one element. If there were two independent solutions in the list then this would form a basis for the solution space. The fact that there is only one element is detected by the top-level special function solver and appropriate action taken, as described above.

6.2 Other Sub-Solvers

Most of the other special function sub-solvers are more complicated, but they essentially just use the same techniques repeatedly to recognise the form of an ODE and extract parameters in the process. They all fail as early as possible, in which case the task is passed on to the next sub-solver in the chain by the top-level special function solver.

In many cases, such as the Kummer-Laguerre sub-solver, most of the processing is common but then the solver branches, in this particular example depending on whether a particular parameter is a non-negative integer – if so then the ODE is satisfied by a Laguerre polynomial, otherwise it is satisfied by a Kummer hypergeometric function, the former being a special case of the latter. The solver always tries to pick the most specific special function solution.

7 Known Problems and Further Work

This paper describes work in progress. I intend to add a few of the more obscure special function ODEs and the code needs much more testing. It is much easier to accept the right ODE than to reject wrong ones, hence it is important to run *all* test examples through a prospective final version of the *complete* solver.

One problem that has emerged in testing is that care is required to extract correctly possibly rational parameters from rational expressions. Such rational parameters might be either rational numbers or symbolic rational expressions. Where necessary, I am revising the code to make polynomials monic for uniqueness and generally avoid extracting numerators and denominators during the decomposition of ODE coefficients. The general strategy is *first* to find any shift k from some polynomial in $(x + k)$ after making it monic, and *then* to find other parameters including any scaling a involved in the transformation $x \rightarrow a(x + k)$. This complication does not arise in the simple case of the Hermite solver presented above.

8 Conclusions and Comparison with Maple

My algorithm does not perform a great deal of canonicalization, it uses the (monic) coefficient of y' as the primary key to recognising the ODE and it matches against a fairly large number of standard ODEs, essentially one per special function. By contrast, I understand [6] (although I have not looked at the code) that Maple first reduces the ODE to an invariant form with no y' term. (Various possible transformations to effect this are described in [17], among which that in §32 is probably the one to use in general.) It then matches the resulting single coefficient function of the y term against those corresponding to the “primary classical equations”, in particular hypergeometric, Bessel, Whittaker and Legendre, via a partial fraction decomposition. If necessary, it tries a transformation of the form $x \rightarrow ax^b$.

It would be useful to compare the pros and cons of these two approaches carefully, which I have not (yet) done. The very cursory comparison in §2 of this paper suggests that my approach may be better for very simple text-book examples. I would expect the Maple approach to work better for non-trivially transformed special function ODEs.

There is no algebraic complexity in my approach, so the matching should either succeed or fail rapidly (and preliminary testing bears this out). By contrast, a transformation to invariant form might involve algebraic complexity that could slow down the matching considerably. I have no evidence that this does happen in practice, but the obvious transformation [17, §32] involves integrals over the (monic) y' coefficient function. CASs can spend a long time trying to evaluate complicated integrals before giving up and returning them unevaluated. Maybe the way forward is to combine both approaches, using transformation to invariant form only if more direct matching fails.

Special function ODEs is not an area that has been well covered by recent benchmarks of CASs in general (e.g. [12]) or of ODE solvers in particular (e.g. [16]). Perhaps it is time for a benchmark of special function ODE solvers, but not until I have released `ODESolve 1.07!`

Acknowledgements

I am very grateful to Edgardo Cheb-Terrab and George Labahn for providing me with some inside information about the ODE solver in Maple. In particular, I thank George for providing me with an outline of his code in Maple V Release 5 for solving special function ODEs.

References

- [1] M. Abramowitz and I. A. Stegun (eds.), *Handbook of Mathematical Functions*, National Bureau of Standards, Washington, DC, 1964; Dover Publications, NY, 1965.
- [2] W. W. Bell, *Special Functions for Scientists and Engineers*, Van Nostrand, London, 1968.
- [3] C. Cannam and W. Neun, *SPECFN: Package for special functions*, online documentation accessible via <http://www.rrz.uni-koeln.de/REDUCE/>.
- [4] E. Cheb-Terrab, private communication.
- [5] H. Hochstadt, *The Functions of Mathematical Physics*, Wiley-Interscience, NY, 1971; Dover Publications, NY, 1986.
- [6] G. Labahn, private communication.
- [7] G. Labahn, Solving Linear Differential Equations in Maple, *MapleTech*, 2(1), 1995, 20–28.
- [8] 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.
- [9] M. A. H. MacCallum, private communication.
- [10] M. A. H. MacCallum and F. J. Wright, *Algebraic Computing with REDUCE*. Oxford University Press (1991).
- [11] REDUCE web home page, <http://www.rrz.uni-koeln.de/REDUCE/>.
- [12] M. Wester, A Review of CAS Mathematical Capabilities. *Computer Algebra Nederland Nieuwsbrief 13* (Dec. 1994), 41–48. The latest version of this review is available from http://math.unm.edu/~wester/cas_review.html.
- [13] F. J. Wright, An Enhanced ODE Solver for REDUCE. *Programirovanie* No 3 (1997), 5–22, in Russian, and *Programming and Computer Software* No 3 (1997), in English.
- [14] F. J. Wright, *Design and Implementation of ODESolve 1+ : An Enhanced REDUCE ODE Solver*, CATHODE Workshop, Marseilles, May 1999. Available as a PDF file via <http://www.maths.qmw.ac.uk/~fjw/>.
- [15] F. J. Wright, ODESolve 1+. The latest release is available in the directory <http://reduce.maths.qmw.ac.uk/packages/odesolv1/> which contains full source code, test and documentation files.
- [16] 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/>.
- [17] D. Zwillinger, *Handbook of Differential Equations*, Academic Press. (Second edition 1992.)