Using the redcas package

Martin Gregory

November 3, 2026

Preface

redcas is an interface to REDUCE [1], an open source, portable, general-purpose, computer algebra system supporting scalar, vector, matrix and tensor algebra, symbolic differential and integral calculus, arbitrary precision numerical calculations and output in LaTeX format. Using examples, this vignette shows how to configure, start and stop REDUCE sessions and control output format and explains the three different ways to interact with the REDUCE session.

REDUCE is written in Lisp and can run on various Lisp dialects, the most common being CSL and PSL. For these two dialects pre-packaged releases known as snapshots are available for various operating systems. *redcas* can use the CSL and PSL versions.

If you are unfamiliar with REDUCE you should peruse the vignette *Brief Overview of REDUCE* which provides a very short introduction to REDUCE for R users, highlighting differences between the two environments. Full details are available in the REDUCE user manual [3] and detailed examples are available on the tutorials page [8].

In this vignette we use upper case for REDUCE keywords to easily distinguish them from R keywords. Note that REDUCE is, by default, case-insensitive with the exception of the contents of strings.

Contents

1 Configuring, starting and stopping REDUCE **3**
- 1.1 Configuring REDUCE . . . 3
- 1.2 Starting and stopping REDUCE . . . 3

2 Controlling output **5**
- 2.1 Switches . . . 5
- 2.2 Global variables . . . 8
- 2.3 Producing LaTeX output . . . 8

3 Interacting with REDUCE **9**
- 3.1 Common arguments . . . 9
- 3.2 redExec . . . 10
- 3.3 redFunc . . . 11
- 3.4 Special interfaces . . . 12
  - 3.4.1 factorize: Factorizing polynomials, rationals or integers . . . 12
  - 3.4.2 redSolve: Solving systems of equations . . . 15
  - 3.4.3 redDf: Differentiation . . . 19
  - 3.4.4 odesolve: Solving Ordinary Differential Equations . . . 22
  - 3.4.5 redLatex . . . 25
  - 3.4.6 Switch handling . . . 28

4 Conversion functions **29**
- 4.1 R2latex . . . 29
- 4.2 red2Rarray . . . 30
- 4.3 red2Rfn . . . 31
- 4.4 red2Rlist . . . 32
- 4.5 Miscellaneous conversion functions . . . 34

A Appendices **36**
- A.1 Known issues . . . 36
  - A.1.1 MacOS/Intel/REDUCE 6866 . . . 36
  - A.1.2 Passing FACTOR, MODULAR in switches argument . . . 36
  - A.1.3 Error displaying omicron in LaTeX . . . 36
  - A.1.4 Long equations in odesolve . . . 37
  - A.1.5 trode option in odesolve . . . 37
  - A.1.6 REDUCE is mostly case insensitive . . . 37
  - A.1.7 redin, lstlisting and Sweave don't mix . . . 37
- A.2 Testing . . . 38
  - A.2.1 Test categories . . . 38
  - A.2.2 Pre-CRAN testing and results . . . 39
  - A.2.3 Post-installation testing . . . 39
- A.3 Producing this vignette . . . 39

Chapter 1

Configuring, starting and stopping REDUCE

redcas can be installed even if REDUCE is not installed, but to use it requires a REDUCE installation. Details on installing can be found on the REDUCE web site [2].

1.1 Configuring REDUCE

In order to use *redcas* we need to provide the location of the REDUCE executables, *redcsl* or *redpsl*. There are four different ways do do this. These are, in order of precedence:

1. explicitly specifying the path using the *dirpath* argument of *redStart*:

```
sessionId -> redStart(dialect="PSL", dirpath="/usr/local/bin")
```

2. the environment variable *REDUCE_EXEC* exists and contains the path to a directory containing one or both executables

```
Sys.setenv(REDUCE_EXEC="/usr/local/bin")
sessionId -> redStart(dialect="CSL")
```

3. the R option *reduce_exec* exists and contains the path to a directory containing one or both executables

```
options(reduce_exec="/usr/local/bin")
sessionId -> redStart(dialect="CSL")
```

4. the executable is located in a directory listed in the *PATH* environment variable.

1.2 Starting and stopping REDUCE

A REDUCE session is started using the *redStart* function. While the session is open, various functions can be used to send commands to the REDUCE session and retrieve the output. This allows direct handling of the output from a given command or set of commands and easy separation of calculation and display. To close a session the *redClose* function is used. This optionally retrieves the complete transcript.

To start a REDUCE session load the package and call *redStart*:

```
> library(redcas)
> s1 <- redStart(dialect="csl", echo=TRUE)
```

This starts a CSL REDUCE session. CSL is the default dialect and does not need to be specified. Sessions are started using a pipe which means they are interactive, i.e. the INT switch is on. The ECHO switch is on by default to facilitate parsing REDUCE output. *redStart* provides the *echo* argument to turn the switch off at the start of the session but this is discouraged.

redStart

- start a REDUCE session with output directed to a file in the R session's temporary directory;
- loads the *ASSIST* package which provides, among other features, functions which are indispensible for managing lists,
- loads the REDUCE functions which are part of *redcas* and
- sets any switches specified in the switches argument;
- registers the session details (identifier, connection number, REDUCE command, REDUCE log file name, number of calls to redExec, number of lines read from the log file) in a data frame in the package environment.

if successful, *redStart* returns the identifier for the session otherwise it terminates with *stop*. The identifer is required by other *redcas* functions which communicate with the REDUCE session.

The session is closed by calling the *redClose* function. The optional *log* argument saves the entire REDUCE transcript to the specified location.

```
>   redClose(s1, log="controlling-reduce.clg")
[1] TRUE
```

Session information can be viewed using the *showSessions* function and individual details can be retrieved programmatically using the *sessionGet* function. For example, to retrieve the session log one may use

```
readLines(sessionGet("logname"))
```

Chapter 2

Controlling output

REDUCE provides a large number of *switches* for controlling how calculations are performed and how results are presented. These are similar to R options except that, as the name suggests, they are boolean only. For options requiring a non-boolean value, REDUCE uses global variables. Finally there are three different packages for producing LaTeX output, one of which is supported transparently by *redcas*.

2.1 Switches

There are approximately 200 switches available, some are part of the core system while the remainder are specific to user-contributed packages. In this section we show how the switches *ROUNDED*, *EXP*, *MCD*, *NAT* and *DIV* affect how expressions are displayed or simplified. *ROUNDED* affects expressions containing numeric values while the other four affect any expression. See the *Switch Summary* [7] for an alphabetic listing of (most) documented switches including a description of the switch and the context in which it applies. Some additional switches are documented in Appendix A.7 of the REDUCE user manual [3]. The second of the REDUCE interactive lessons [8] also provides useful information. For convenience, the vignette *Switches by Context* presents the list from the switch summary by context.

The function *redExec* communicates directly with the REDUCE session. It returns a list with three named character vector elements: *out* is the output of the executed commands; *cmd* contains the commands executed as displayed in the REDUCE log, including the line numbers; and *raw* contains the log lines including both commands and output. Here we are only interested in the *out* element of the output - we cover this function in more detail in section 3.2.

As a numeric example, consider calculating $(987,654,321/15)^3$. We start a new session and then assign the result to the REDUCE variable XN for later use:

includerangemarker=false]

```
> s1 <- redStart(echo=TRUE)
> o1 <- redExec(s1, "xn := (987654321/15)^3;", drop.blank.lines=TRUE)
> writeLines(o1[["out"]])

       3568216032198131885871043
xn := ---------------------------
                  125
```

As we can see, by default REDUCE uses exact, indefinite precision arithmetic. To display as a float we need to turn on the *ROUNDED* switch:

```
> cmd <- c("ON ROUNDED;", "xn;", "OFF ROUNDED;")
> redExec(s1, cmd, drop.blank.lines=TRUE)[["out"]]

[1] "2.85457282576e+23"
```

As an analytical example consider the function $f(x,y) = (x+y)^2 + 1/(x+y)$ which can be displayed in various ways. Some are:

$$\frac{(x+y)^3+1}{(x+y)} \qquad \frac{x^3+3x^2y+3xy^2+y^3+1}{(x+y)}$$

$$(x+y)^2+(x+y)^{-1} \quad (x^3+3x^2y+3xy^2+y^3+1)(x+y)^{-1}$$

In REDUCE, variables do not need to be declared. A variable can be used with or without assigning it a value. If no value has been assigned the variable is termed unbound and stands for itself. It can, for example, be an independent variable with respect to which an expression is differentiated. Variables with a value are known as bound. In the following REDUCE statement *f* is bound and *x* and *y* are unbound:

```
f := (x + y)^2;
```

First we look at the effect of the *NAT* switch on how REDUCE displays results by assigning the bound variable *f* and displaying it:

```
>   expoff <- redExec(s1, "OFF EXP;")
>   f <- redExec(s1, "f := (x + y)^2;", drop.blank.lines=TRUE)[["out"]]
>   f

[1] "             2" "f := (x + y)"
```

We turned the *EXP* switch off to avoid expanding expressions for now. We have created a variable *f* in both the REDUCE session and in R. We see that the resulting vector in R has two elements, even though *f* is a single entity. This is due to the fact that, even in plain text output, REDUCE by default displays results in a way that is close to the normal maths display. This is controlled by the switch *NAT* (natural) which is on by default. We can see how it would appear in REDUCE if we use *writeLines* to display the variable:

```
>   writeLines(f)

            2
f := (x + y)
```

Alternatively we can turn the switch *NAT* off to get:

```
>   redExec(s1, c("OFF NAT;", "f ; ON NAT;"), drop.blank.lines=TRUE)[["out"]]

[1] "(x + y)**2$" " ON NAT;"
```

Now the expression is written on a single line and is syntactically correct REDUCE code including the dollar terminator. For parsing results generated by redExec in R this is much more suitable than with NAT on. *redcas* functions use NAT off internally to simplify parsing.

We can now assign another bound variable, *h*, as a function of *f*:

```
> writeLines(h <- redExec(s1, "h := f + 1/f;", drop.blank.lines=TRUE)[["out"]])
```

```
             4
      (x + y)  + 1
h := --------------
                2
        (x + y)
```

Again we have chosen to use *writeLines* to display *h* because *NAT* is on. The default and recommended setting of the switch *MCD* (Making Common Denominators) is on and this displays the expression as a fraction, calculating a common denominator when rational functions are added as in this case. Since we turned *EXP* off, the expressions in parentheses are not expanded. If we turn off *MCD* we get a product with a negative exponent for the divisor:

```
> writeLines(redExec(s1, c("OFF MCD ;", "h;"),
+            drop.blank.lines=TRUE)[["out"]])

        4             -2
((x + y)  + 1)*(x + y)
```

If we now turn both *EXP* and *MCD* on we get:

```
> writeLines(redExec(s1, c("ON EXP, MCD ;", "h;"),
+            drop.blank.lines=TRUE)[["out"]])

  4      3        2  2        3    4
 x  + 4*x *y + 6*x *y  + 4*x*y  + y  + 1
----------------------------------------
              2            2
             x  + 2*x*y + y
```

where all sub expressions are expanded and common denominators are separated out.

Finally we consider the *DIV* switch. As an example, consider the indefinite integral of $sin^4 x$ with respect to x:

```
> rf2 <- redExec(s1, "INT(sin(x)**4, x);")
> writeLines(rf2[["out"]])

                    3
  - 2*cos(x)*sin(x)  - 3*cos(x)*sin(x) + 3*x
----------------------------------------------
                       8
```

As we can see, by default, REDUCE displays results using a common denominator for all terms. If we want each term to have its own divisor we can turn on the *DIV* switch which is off by default:

```
> rf3 <- redExec(s1, c("ON DIV;", "INT(sin(x)**4, x);", "OFF DIV;"))
> writeLines(rf3[["out"]])

    1               3    3                    3
 - ---*cos(x)*sin(x)  - ---*cos(x)*sin(x) + ---*x
    4                   8                    8
```

Note that both *EXP* and *MCD* must be on to produce this form.

Further switches which are relevant for controlling simplification are those in the categories *Algebra* and *Output*. Note that the switch *FACTOR* is a mirror image of *EXP*: if one is on the other is automatically off.

2.2 Global variables

Several packages use global variables for different purposes including determining output. Most are fairly specialized packages but a useful example is the *DF* operator (differentiation) which uses the variable *FANCY_PRINT_DF* to determine how to display derivatives for LaTeXoutput. The possible values are

$Setting$	$df(f,x,2,y)$
$PARTIAL$	$\frac{\partial^3 f}{\partial x^2 \partial y}$
$TOTAL$	$\frac{d^3 f}{dx^2 dy}$
$INDEXED$	$f_{x,x,y}$

with *PARTIAL* being the default.

2.3 Producing LaTeX output

The function *redLatex* converts expressions, including the names of variables in the REDUCE session, to LaTeX using the REDUCE package TMPRINT. This package is used because it can be easily applied to fragments, supports using the \frac command and translates variable names which are Greek letters or certain mathematical operators to the corresponding LaTeX command. *redLatex* adds the ability to define maps to convert arguments to raised or lowered indices, to translate object names and to enclose the results in a math environment[1]. It can also be called by any *redcas* function which supports the *outfmt=latex* argument. The function *R2latex* can convert certain R objects to LaTeX.

As an example of an expression, consider the variable *h*

```
f := (x + y)^2; h := f + 1/f;
```

First with the switch *MCD* off:

```
> writeLines(redLatex(s1, "h", mathenv="equation*", switches=c(off="mcd"))[["tex"]])
```

$$\left(x^2+2xy+y^2\right)^{-1}\left(x^4+4x^3y+6x^2y^2+4xy^3+y^4+1\right)$$

the function returns a list with the same three elements as *redExec* and a fourth, named "tex", containing the LaTeX output. Since we only need the LaTeX output here we select the "tex" list element. We use *writeLines* to avoid printing the vector index numbers. The *switches* argument causes *MCD* to be turned off for the duration of the *redLatex* execution. With *MCD* on the same call returns:

$$\frac{x^4+4x^3y+6x^2y^2+4xy^3+y^4+1}{x^2+2xy+y^2}$$

For more detailed examples, including converting matrices, arrays and lists, see section 3.4.5

[1] see the documentation for details

Chapter 3

Interacting with REDUCE

redcas provides three different methods for interacting with REDUCE depending on one's knowledge of REDUCE and on how results are to be returned. These methods are covered in sections 3.2 to 3.4 below. The different methods may be used together in the same session. In brief the methods are:

redExec executes a vector of arbitrary REDUCE code and returns its output. The code to execute may contain any element of the language and so requires a certain level of familiarity with REDUCE programming;

redFunc executes a vector of REDUCE functions including those defined by the user. This requires knowledge of the arguments of the functions to be executed;

Special Interface functions provide access to specific functions. These functions have arguments tailored to the specific function and construct the necessary REDUCE calls. They also provide some level of validation of the inputs. Using these functions is described in detail in the package documentation.

In the following section we first describe some arguments common to each function which calls REDUCE.

3.1 Common arguments

There are a number of arguments common to multiple *redcas* functions.

id: is the session identifier returned by *redStart*. It is required by all functions which interact with the session.

switches: is a character vector with one or both of the following named elements

on: a comma-separated string of switches which should be on

off: a comma-separated string of switches which should be off.

This argument is available on all functions which interact with the session except *redExec* and causes the named switches to be set to the desired state for the duration of execution. When used with *redStart* the *switches* argument is used to set defaults for the session, i.e. the switches are not restored to the previous state after execution of *redStart*.

Note that *swset* prohibits setting of the *FANCY* switch because this formats all output as LaTeX. This could cause issues for *redcas* functions when parsing REDUCE output. Functions which support the *latex* option of the *outfmt* argument selectively turn on *FANCY* when producing output, so there should generally not be any need to explicitly set *FANCY*.

outfmt: is one of the strings *expr*, *func*, *latex* or *text* and determines whether output is an expression, a function, LaTeX or plain text, respectively. The argument is available on all functions which return REDUCE output except *redExec*. The default varies depending on the function and not all options are available on all functions.

verbose: is a logical value controlling printing of additional information.

notify: numeric. If not zero, writes a message to the transcript every *notify* seconds to indicate execution is ongoing. Passed via ... to *redExec*.

timeout: numeric. If not zero and the function is still executing after *timeout* seconds, the function terminates with a message to the transcript. Passed via ... to *redExec*.

vec: logical. Determines whether output of *R2latex* is returned as a (nested) list or a vector. Also available from *odesolve* which calls *R2latex*.

eol: logical. Determines whether a LaTeX end of line character is added to the end of all but the last element of vectors in the output of *factorize*, *odesolve* or*R2latex*.

disallow: a character vector of the names of switches which should not be changed for the called function. This is primarily for use in *redcas* functions which accept the switches argument and require a specific state of some switches, but could also be used in user defined functions.

3.2 redExec

redExec allows for the execution of a vector of abitrary REDUCE commands. This means that the vector may also include conditional statements, compound statements and procedure definitions. All output generated[1] is returned as a list[2] with three character vector elements:

out the output from the commands executed;

cmd the commands as written to the REDUCE transcript. This depends on the state of the ECHO switch which is on by default and generally should not be turned off;

raw the portion of the REDUCE transcript produced by the call. This includes interspersed commands and output.

redExec uses the *drop.blank.lines* option to make the output more compact - the REDUCE transcript uses blank lines generously.

Depending on what further processing will be done in R, we also need to consider that *redExec* returns all output from the commands submitted. So we may want to split execution of REDUCE commands into appropriate separate calls to *redExec*.

All other functions which execute REDUCE code (factorize, odesolve, R2latex, red2Rarray, redDf, redFunc, redLatex, redSolve) do so using *redExec*.

In the following example we define a REDUCE procedure for determining the largest prime smaller than a given integer and use a second call to *redExec* to call the procedure and display the output:

```
> lp <- c("PROCEDURE lastprime (n) ;",
+         "   BEGIN SCALAR k, kfact ;",
+         "      k:=n+1 ;",
+         "      REPEAT",
+         "      << k:=k-1 ;",
+         "         kfact:=factorize(k)",
+         "      >> UNTIL LENGTH(kfact) = 1 and SECOND(FIRST(kfact))=1;",
+         "      RETURN FIRST(FIRST(kfact))",
+         "   end ;")
> deflp <- redExec(s3, lp)
```

[1] intermediate output can be suppressed by turning the OUT switch off but this **must** be turned back on at the latest as the last command in the command vector.

[2] If *redExec* fails it returns *FALSE*.

```
> writeLines(sprintf("largest prime < 10^5: %d",
+                    as.integer(redExec(s3, "lastprime(10e5);",
+                               drop.blank.lines=TRUE)["out"])))

largest prime < 10^5: 999983
```

3.3 redFunc

redFunc provides an alternative method for interfacing with REDUCE. It allows the direct execution of a list of REDUCE operators[3] returning the results to R as a character vector or, if multiple operators are passed, a list of character vectors. *redFunc* does not assign the results to variables in the REDUCE session. An example using numeric operators:

```
> rf1 <- redFunc(s3, func=c('nextprime', 'fix', 'max'),
+                      param=list(10000,   -5/4,  c(2,3,-4,8)))
> str(rf1)

List of 3
 $ nextprime: chr "10007"
 $ fix      : chr "-1"
 $ max      : chr "8"
```

Note that *FIX* returns the integer part of its argument, i.e. is the REDUCE version of the R *as.integer* function.

We can also use *redFunc* to call the *lastprime* defined in the previous section:

```
>   lp10e5 <- as.integer(redFunc(s3, "lastprime", 10e5))
>   lp10e5

[1] 999983
```

See section 4.3 for how to convert expressions returned from REDUCE to R functions.

Some REDUCE functions return a list. For example the *MAP* function is the REDUCE equivalent of *lapply* and can also be used on matrices and operator expressions as well as lists. The first argument to *MAP* is the function to execute on each element. Unlike *lapply* a complete call to the function is required with a placeholder of the form *~identifier* for the argument where the element is referenced. The placeholder identifier may not appear in any of the expressions. The second argument is the data aggregate. Here is an example of integrating a list of expressions:

```
> rf5 <- redFunc(s3, "map", "int(~w,x), {x^2, 2*x^5, cos(x)^4}") ;
> writeLines(rf5)

   3
  x
{----,
  3
   6
  x
 ----,
```

[3] including user-defined procedures

```
  3
                   3
   - 2*cos(x)*sin(x)  + 5*cos(x)*sin(x) + 3*x
-------------------------------------------}
                      8
```

MAP returns an object of the same type as the second argument, in this case a list. We can use *red2Rlist* to convert this to an R list but need to turn off NAT for the *redFunc* call:

```
> rf6 <- red2Rlist(redFunc(s3, "map", "int(~w,x), {x^2, 2*x^5, cos(x)^4}",
+                          switches=c(off="nat")) )
> str(rf6)

List of 3
 $ : chr "x**3/3"
 $ : chr "x**6/3"
 $ : chr "( - 2*cos(x)*sin(x)**3 + 5*cos(x)*sin(x) + 3*x)/8"
```

3.4 Special interfaces

While *redFunc* can call any REDUCE operator, the current version of *redcas* provides specialized interfaces to a small number of interesting functions. These interfaces run checks on the inputs, provide more features for controlling the output and convert output to appropriate R objects. Future versions of *redcas* will provide further such interfaces.

3.4.1 factorize: Factorizing polynomials, rationals or integers

The operator *FACTORIZE* can factorize univariate and multivariate polynomials with integer, rational or complex coefficients. It can also factorize integers and rationals. The numbers e, i and pi may also appear as coefficients. If *ROUNDED* is on, both e and pi are substituted by their values to the current precision level. To factorize polynomials involving i *COMPLEX* must be on. The *redcas* function *factorize*, provides an interface to the REDUCE operator:

factorize(id, x, prime=integer(0), outfmt="text", switches=character(0), eol=FALSE, verbose=FALSE, ...)

For each element in x a list of factor/exponent pairs is returned, for example:

```
> fin <- c("x^2+1",
+          360,
+          "4x^5+30x^4+107x^3+605x^2/2+600x+500",
+          "x^3 + i*x^2 + x + i",
+          "2x^7+17x^6+54x^5+72x^4+12x^3-69x^2-68x-20")
> fotxt <- factorize(s3, fin, outfmt="text")
> for (i in 1:length(fotxt)) writeLines(paste0(fotxt[[i]], collapse=", "))

list("x**2 + 1", 1)
list(2, 3), list(3, 2), list(5, 1)
0.5, list("x**2 + 8", 1), list("2*x + 5", 3)
list("i + x", 1), list("x**2 + 1", 1)
list("2*x + 5", 1), list("x - 1", 1), list("x + 2", 2), list("x + 1", 3)
```

In the case of a common numeric factor, this factor is the first element in the list as can be seen in the result for *fin[3]*. Because values containing only digits are converted by *redcas* to numeric, it is displayed as 0.5 instead of $\frac{1}{2}$.

LaTeX output

We can ask for LaTeX output when factorizing the expressions in the vector *fin* above by specifying outfmt="latex". In this case a single list is returned, each element being a vector of the prime factors raised to the appropriate power. In order to display the factors for each expression on a single line we use the *array* environment. This requires adding an appropriate number of blanks to the expressions with fewer factors, adding column separators (&) and using the *ltx.append.eol* function to append LaTeX end of line commands as in the following:

```
> foltx <- factorize(s3, fin, outfmt="latex")
> foltx.l <- unlist (lapply(foltx, length))
> foltx <- ltx.append.eol(
+     unlist(
+         lapply(1:length(foltx),
+                function(z){paste0(c(foltx[[z]], rep(" ", 4-foltx.l[z])),
+                                   collapse=" & ")}))
+ )
> writeLines(c("\\[", "\\begin{array}{llll}",
+              foltx,
+            "\\end{array}", "\\]"))
```

$$\begin{array}{llll} x^2+1 & & & \\ 2^3 & 3^2 & 5 & \\ \frac{1}{2} & x^2+8 & (2x+5)^3 & \\ i+x & x^2+1 & & \\ 2x+5 & x-1 & (x+2)^2 & (x+1)^3 \end{array}$$

Complex roots

Note that the polynomial x^2+1 is not factorized because neither the *COMPLEX* nor the *ROUNDED* switch is on. Because of the explicit i in x^3+ix^2+x+i, complex roots are returned for this polynomial. If we try x^2+1 with *COMPLEX* on:

```
> foltx1 <- factorize(s3, fin[1], outfmt="latex", switches=c(on="complex"))
> writeLines(c("\\[", paste0(unlist(foltx1), collapse=" \\quad "), "\\]"))
```

$$x+i \quad x-i$$

Furthermore, if *ROUNDED* is on and *COMPLEX* off, the factor x^2+8 of *fin[3]* is further factorized to $(x+\sqrt{8}i)(x-\sqrt{8}i)$:

```
> foltx3 <- factorize(s3, fin[3], outfmt="latex", switches=c(on="rounded"))
> writeLines(c("\\[", paste0(unlist(foltx3), collapse=" \\quad "), "\\]"))
```

$$4 \quad 2.82842712475i+x \quad -(2.82842712475i-x) \quad (x+2.5)^3$$

If both switches are on, this further factorization is not performed.

Using the *ROUNDED* switch

With LaTeX output the numbers e and pi are displayed in the usual mathematical form:

```
> foepi <- factorize(s3, c("2pi^2 + 9pi*x - 5x^2", "x^e*e - x^e*x + e - x"),
+                    outfmt="latex")
> writeLines(c("\\[", "\\begin{array}{llll}",
+              ltx.append.eol(unlist(lapply(foepi, paste0, collapse=" & "))),
+              "\\end{array}", "\\]"))
```

$$\begin{array}{ll} 2\pi - x & \pi + 5x \\ x^{e} + 1 & e - x \end{array}$$

The *ROUNDED* switch replaces e and pi with their values at the current precision:

```
> foepi <- factorize(s3, c("2pi^2 + 9pi*x - 5x^2", "x^e*e - x^e*x + e - x"),
+                    outfmt="latex", switches=c(on="rounded"))
> writeLines(c("\\[", "\\begin{array}{llll}",
+              ltx.append.eol(unlist(lapply(foepi, paste0, collapse=" & "))),
+              "\\end{array}", "\\]"))
```

$$\begin{array}{lll} -5 & x + 0.628318530718 & x - 6.28318530718 \\ -\left(x^{0.718281828459} x^{2} + 1\right) & x - 2.71828182846 & \end{array}$$

Multivariate polynomials

factorize can also handle polynomials of high degree with ease. We show this using $x^{33} + y^{33}$ and because the first factor contains 31 terms we display using LaTeX as this is much easier to read.

```
> q33 <- factorize(s3, "x^33+y^33", outfmt="latex")
> writeLines(c("\\begin{dgroup*}",
+              paste0("\\begin{dmath*}", ltx.append.eol(unlist(q33)), "\\end{dmath*}"),
+              "\\end{dgroup*}"))
```

$$x^{20} + x^{19}y - x^{17}y^{3} - x^{16}y^{4} + x^{14}y^{6} + x^{13}y^{7} - x^{11}y^{9} - x^{10}y^{10} - x^{9}y^{11} + x^{7}y^{13} + x^{6}y^{14} - x^{4}y^{16} - x^{3}y^{17} +$$

$$xy^{19} + y^{20}$$

$$x^{10} - x^{9}y + x^{8}y^{2} - x^{7}y^{3} + x^{6}y^{4} - x^{5}y^{5} + x^{4}y^{6} - x^{3}y^{7} + x^{2}y^{8} - xy^{9} + y^{10}$$

$$x^{2} - xy + y^{2}$$

$$x + y$$

Note that requesting LaTeX output can take up to four times longer than for plain text.

3.4.2 redSolve: Solving systems of equations

The operator *SOLVE* [4] is a powerful tool for solving one or more simultaneous algebraic equations in one or more variables, both linear and non-linear. Equations may contain arbitrary constants. The operator takes two arguments: a list of the equations comprising the system and a list of the unknowns. The latter is optional unless the equations contain arbitrary constants, such as *c* in the following example. First we use *redExec* to call *solve* in order to show the raw REDUCE output:

```
> code <- c("eq1:= x + y + z ;",
+           "eq2:= x^2 + y^2 + z^2 -9 ;",
+           "eq3:= x^2 + c*y^2 - z^2 ;",
+           "on rounded ;")
>           def.eqn <- redExec(s3, code) # to avoid printing the equations again
>           sol1 <- redExec(s3, "sol1:=solve({eq1, eq2, eq3}, {x, y, z});")[["out"]]
```

which produces the following nested (REDUCE) list.

```
             2.12132034356*c - 2.12132034356
sol1 := {{x=---------------------------------,
                          2     0.5
                        (c  + 3)

             4.24264068712
          y=---------------,
                2     0.5
              (c  + 3)

              - 2.12132034356*(c + 1)
          z=--------------------------},
                     2     0.5
                   (c  + 3)

              - 2.12132034356*c + 2.12132034356
         {x=-----------------------------------,
                         2     0.5
                       (c  + 3)

             -4.24264068712
          y=---------------,
                2     0.5
              (c  + 3)

             2.12132034356*(c + 1)
          z=----------------------},
                     2     0.5
                   (c  + 3)

         {x=2.12132034356,y=0,z=-2.12132034356},

         {x=-2.12132034356,y=0,z=2.12132034356}}
```

The top-level list has four elements, one for each solution, each of which is a list containing the values of the unknowns. Note that the arbitrary constant *c* is part of the first two solutions. Even turning NAT off,

this can be very unwieldy so *redcas* provides the *redSolve* function which parses the output and returns an object of class *redcas.solve* containing both inputs and outputs. For convenience, *redSolve* supports the *switches* argument to allow turning switches on or off before calling *solve*:

```
> soln <- redSolve(s3,
+                  eqns=c("x + y + z = 0", "x^2 + c*y^2 + z^2 = 9",
+                         "x^2 + y^2 - z^2 = 0"),
+                  unknowns=c("x", "y", "z"),
+                  switches="on rounded;")
> print(soln)

Equations:
  x + y + z = 0
  x^2 + c*y^2 + z^2 = 9
  x^2 + y^2 - z^2 = 0

Number of solutions: 4

Solutions:
               x                       y                      z
   2.12132034356                       0       -2.12132034356
  -2.12132034356                       0        2.12132034356
               0       3.0/(c + 1)**0.5  ( - 3)/(c + 1)**0.5
               0  ( - 3.0)/(c + 1)**0.5       3/(c + 1)**0.5

Unknowns: x,y,z
Switches: on: rounded
```

As can be seen, *redcas* provides a print method which attempts to display the solutions in as compact a manner as possible but will not always be as neat as in this example. The individual slots of the class can be accessed in the usual way using @. Note that the *solutions* slot has type character since values may contain variable names which may or may not exist in the R session. Even if they do exist, using these values might cause unintended results. The object has another slot *rsolutions* which contains evaluated versions of the *solutions* slot if these are numbers otherwise character. The print method does not display the *rsolutions* slot. In this particular case one could evaluate the results for a specific value of *c* as follows:

```
>     c <- 25
>     sol25 <- lapply(soln@solutions,
+                     function(x){unlist(lapply(x,
+                                 function(y){eval(str2expression(y))}))})
>     print(sol25)

[[1]]
       x        y        z
 2.12132  0.00000 -2.12132

[[2]]
       x        y        z
-2.12132  0.00000  2.12132

[[3]]
          x         y          z
 0.0000000 0.5883484 -0.5883484
```

```
[[4]]
          x           y           z
 0.0000000 -0.5883484  0.5883484
```

An alternative is to use the function *red2Rfn* — see section 4.3

Switches can have a significant effect on the result. As an example consider the following set of equations with the *ROUNDED* switch on and then off:

```
> r.on <- redSolve(id=s3,
+                  eqns=c("x+y+z = 0", "x^2 + y^2 + z^2 = 9", "x^2 + y^2 = z"),
+                  unknowns=c("x", "y", "z"), switches="on rounded;"  )
> r.off <- redSolve(id=s3,
+                   eqns=c("x+y+z = 0", "x^2 + y^2 + z^2 = 9", "x^2 + y^2 = z"),
+                   unknowns=c("x", "y", "z"), switches="off rounded;"  )
```

With *ROUNDED* on, REDUCE returns a set of solutions containing only real and complex constants (we have rounded to 6 digits to show the solutions as side-by-side columns):

```
Equations:
  x+y+z = 0
  x^2 + y^2 + z^2 = 9
  x^2 + y^2 = z

Number of solutions: 4

Solutions:
                        x                          y           z
     1.77069 + 2 .21496i     1.77069 - 2 .21496i    - 3.54138
   - 1.27069 + 0 .58648i   - 1.27069 - 0 .58648i      2.54138
   - 1.27069 - 0 .58648i   - 1.27069 + 0 .58648i      2.54138
     1.77069 - 2 .21496i     1.77069 + 2 .21496i    - 3.54138

Unknowns: x,y,z
Switches: on: rounded
```

and in this case the *rsolutions* slot contains complex or numeric types. With *ROUNDED* off we get the same set of solutions but not simplified:

```
Equations:
  x+y+z = 0
  x^2 + y^2 + z^2 = 9
  x^2 + y^2 = z

Number of solutions: 4

Solution 1:
x:      (2*sqrt( - sqrt(37) - 7)*sqrt(3) + sqrt(74) + sqrt(2))/(4*sqrt(2))
y:   ( - 2*sqrt( - sqrt(37) - 7)*sqrt(3) + sqrt(74) + sqrt(2))/(4*sqrt(2))
z:   ( - (sqrt(37) + 1))/2
```

```
Solution 2:
x:   ( - 2*sqrt( - sqrt(37) - 7)*sqrt(3) + sqrt(74) + sqrt(2))/(4*sqrt(2))
y:      (2*sqrt( - sqrt(37) - 7)*sqrt(3) + sqrt(74) + sqrt(2))/(4*sqrt(2))
z:   ( - (sqrt(37) + 1))/2

Solution 3:
x:         (2*sqrt(sqrt(37) - 7)*sqrt(3) - sqrt(74) + sqrt(2))/(4*sqrt(2))
y:      ( - 2*sqrt(sqrt(37) - 7)*sqrt(3) - sqrt(74) + sqrt(2))/(4*sqrt(2))
z:        (sqrt(37) - 1)/2

Solution 4:
x:      ( - 2*sqrt(sqrt(37) - 7)*sqrt(3) - sqrt(74) + sqrt(2))/(4*sqrt(2))
y:         (2*sqrt(sqrt(37) - 7)*sqrt(3) - sqrt(74) + sqrt(2))/(4*sqrt(2))
z:        (sqrt(37) - 1)/2

Unknowns: x,y,z
Switches: off: rounded
```

Although the expressions contain no variables, they do contain negative square roots which would produce NaNs if evaluated. In this case, the expressions may be evaluated by substituting sqrt(-x) with complex(imaginary=sqrt(x)). A future release of *redcas* may implement handling of such cases.

Underdetermined systems

In cases where there are more variable than unknowns, REDUCE provides solutions which contain arbitrary constants of the form $ARBINT(n)$ for real and $ARBCOMPLEX(n)$ for complex solutions where n is the nth such constant generated in the session. For LaTeX output We can convert this to C_n with the *usermap* argument: the :

```
> arbon <- redSolve(id=s3, eqns=c("x^3+y", "y^2+z"), unknowns=c("x", "y", "z"),
+                   outfmt="latex",
+                   usermap=list(ident=c(arbcomplex="C"), index=c(C="_")))
> writeLines(ltx.append.eol(c("x & y & z",
+                             unlist(lapply(arbon@solutions, paste0,
+                                           collapse=" & "))))
```

$$\begin{array}{lll} x & y & z \\ C_1 & -x^3 & -x^6 \end{array}$$

To avoid arbitrary constants altogether and present the solutions as expressions of one or more of the variables, we can turn off the *ARBVARS* switch:

```
> arboff <- redSolve(id=s3, eqns=c("x^3+y", "y^2+z"), unknowns=c("x", "y", "z"),
+                    outfmt="latex", switches=c(off="arbvars"))
> writeLines(ltx.append.eol(c("y & z",
+                             unlist(lapply(arboff@solutions, paste0,
+                                           collapse=" & "))))
```

$$\begin{array}{ll} y & z \\ -x^3 & -x^6 \end{array}$$

3.4.3 redDf: Differentiation

redDf provides an interface to the *DF* operator [5] which differentiates expressions by one or more variables specifying the degree for each variable. The syntax is

```
redDf(id, expr, depend=character(0), style=character(0), outfmt="func", switches=character(0),
name=character(0), verbose=FALSE, ...)
```

The argument *expr* is a character vector with each element containing the expression to be differentiated, the variable(s) to differentiate by, and the corresponding degree

$$x \quad ,v_1 \quad (,d_1) \quad \ldots \quad (,v_n \quad (,d_n))$$

where

() indicate optional entries,

x is the expression to be differentiated,

v_i are the variables to differentiate by. The order of the v_i determines the order of differentiation, and

d_i is the degree of differentiation by v_i. If d_i is 1 it may be omitted.

By default, *redDf* returns a single function if *expr* contains only one element, otherwise a list of functions. We will use the following expressions to illustrate the use of *redDf*:

$$f1 = vcos(w) \quad f2 = (a+b)^2 \quad f3 = x^3 + x^2/2$$

We start with differentiation by a single variable:

```
> redDf(s1, c("v*cos(w), w", "(a+b)^2, a", "x^3+x^2/2, x, 2"))

$fn1
function (w, v)
{
    return(-sin(w) * v)
}
<environment: 0x37bab968>

$fn2
function (a, b)
{
    return(2 * (a + b))
}
<environment: 0x37b40e20>

$fn3
function (x)
{
    return(6 * x + 1)
}
<environment: 0x37b40f00>
```

Implicit dependencies

The argument *depend* can be used to define dependencies between variables by calling the command *DEPEND* prior to differentiation. If the dependent variable is prefixed by a tilde, the dependency is removed

using the *NODEPEND* command after differentiation. We specify text output to reduce the size of the output.

```
> print(unlist(redDf(s1, c("v*cos(w), w", "(a+b)^2, a"), depend=c("~v, w", "b, a"),
+                    outfmt="text")))
[1] "cos(w)*df(v,w) - sin(w)*v"          "2*(df(b,a)*a + df(b,a)*b + a + b)"
> print(unlist(redDf(s1, c("v*cos(w), w", "(a+b)^2, a"), outfmt="text")))
[1] " - sin(w)*v"                        "2*(df(b,a)*a + df(b,a)*b + a + b)"
```

Since the dependency of v on w is removed after the first call, the result differs for the first element.

Controlling appearance of derivatives

When producing LaTeX we can use the *style* argument to determine how derivatives are to be displayed. There are three possibilities: the default *partial* ($\partial y/\partial x$), *total* (dy/dx) and *indexed* (y_x). In REDUCE this is implemented using a global variable, *FANCY_PRINT_DF*. As a result, the style is applied to all results as we can see:

```
> rdf1 <- c("\\[", "\\begin{array}{lll}",
+           paste0(unlist(redDf(s1, c("v*cos(w), w", "(a+b)^2, a",
+                                     "log(x)*(x+y)^2, y,x"),
+                               depend=c("~v,w", "~b,a"), outfmt="latex")),
+                  collapse="& \\quad"),
+           "\\end{array}",
+           "\\]")
> writeLines(rdf1)
```

$$\cos(w)\,\frac{\partial v}{\partial w} - \sin(w)\,v \qquad 2\left(\frac{\partial b}{\partial a}a + \frac{\partial b}{\partial a}b + a + b\right) \qquad \frac{2(\log(x)x+x+y)}{x}$$

while specifying *style="total"* produces

$$\cos(w)\,\frac{dv}{dw} - \sin(w)\,v \qquad 2\left(\frac{db}{da}a + \frac{db}{da}b + a + b\right) \qquad \frac{2(\log(x)x+x+y)}{x}$$

Note that the style applies to all elements of *expr* so if different styles are required for different elements, it is necessary to use separate calls for each style.

Using function output format

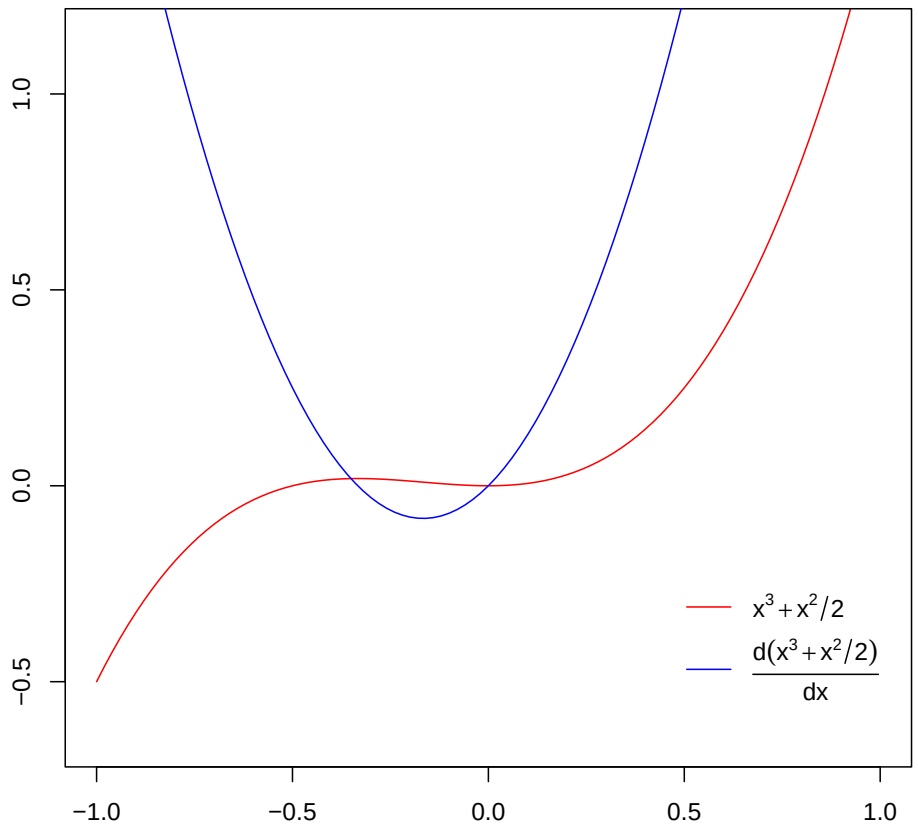
Returning a function can be useful if, for example, the derivative should be plotted. The *curve* function can take a function as its first argument:

```
> pdf(file="Using-redcas-plotdf.pdf") #, width=3, height=3)
> curve(x^3+x^2/2, col="red", xlim=c(-1,1), ylim=c(-0.5,1), asp=1, ann=FALSE)
> dfx <- redDf(s1, "x^3+x^2/2, x")
> curve(dfx, col="blue", add=TRUE)
> legend(x=0.45, y=-0.15, legend=expression(x^3+x^2/2,frac(d(x^3+x^2/2),dx)),
+        col=c("red","blue"), lty=1, bty="n")
> dev.off()
```

```
null device
          1
```

and the result can be seen in Figure 3.1.

Figure 3.1: $x^3 + x^2/2$ and its first derivate



Defining relations between expressions

The *OPERATOR* declaration can be used to implicitly define dependencies because it allows a variable to be treated as a function with an undefined number of arguments. For example

```
1: OPERATOR f, g ;
2: DF(f(x), x) ;
```

returns $df(x)/dx$ because $f(x)$ implies that f depends on x. The *LET* operator goes a step further by defining explict relations between variables or expressions. We could define $g(x)$ to be the derivative of f w.r.t. x:

```
4: LET DF(f(x),x) => g(x) ;
5: DF(f(x),x,2) + DF(f(x),x));
```

which returns $dg(x)/dx + g(x)$. This rule only applies when the name x is used. To generalize to any variable we use:

```
6: LET DF(f(~x),x) => g(x) ;
7: DF(f(y),y,2) + DF(f(y),y));
```

which returns $dg(y)/dy + g(y)$.

In order to define rules through *redcas* we use the *redExec* function.

3.4.4 odesolve: Solving Ordinary Differential Equations

odesolve provides an interface to the REDUCE ODESOLVE operator which solves ordinary differential equations supporting initial and boundary conditions and providing information on the method of solution.

```
odesolve(id, eqn, depvar=character(0), indvar=character(0),
   icond=character(0), bcond=character(0), opt=character(0),
   outfmt="text", vec=FALSE, eol=TRUE,
   switches=character(0), verbose=FALSE, ...)
```

The function returns an object of class *redcas.odesolve* which contains the equation, dependent and independent variables, any specified conditions, options and switches, the solution if any, messages and the generated REDUCE code. If conditions were provided and the option *trode* specified, the general solution is also included.

General solution

Our first example solves the equation $dy/dx - y = x^2$. The solution contains an arbitrary constant rendered by REDUCE as `arbconst(n)` so we use the *usermap* argument to convert this to C_n. We use LaTeX output throughout this section.

```
> ode.1 <- odesolve(s3, eqn="df(y,x) - y = x^2",
+           depvar="y", indvar="x", outfmt="latex",
+           usermap=list(ident=c("arbconst"="C"), index=c("C"="_")))
>           writeLines(ltx.append.eol(unlist(ode.1@solution)))
```

$$y = e^x C_1 - x^2 - 2x - 2$$

Initial and boundary conditions

As an example of initial conditions, consider $y' - y = e^x$ with $y = 0$ when $x = 0$:

```
> ode.2 <- odesolve(s3, eqn="df(y,x,2) - 2*df(y,x) - 3y = 0",
+                   depvar="y", indvar="x", icond="x=0, y=2, df(y,x)=-4",
+                   outfmt="latex")
> writeLines(ltx.append.eol(unlist(ode.2@solution)))
```

$$y = \frac{-e^{4x} + 5}{2e^x}$$

Note that we don't need the *usermap* argument here because there are no arbitrary constants when conditions are specified. The result is:

If we wish to display this as a sum of expressions, we can turn on the *DIV* switch. This, however, does not affect expressions which are equations and so has no effect on the output of *odesolve* which is an equation. To get around this we first call *swset* to turn on *DIV*, then call *odesolve* with text output and finally call *R2latex* on the right hand side of the equation to produce LaTeX output[4]:

```
> ode.2t <- odesolve(s3, eqn="df(y,x,2) - 2*df(y,x) - 3y = 0",
+           depvar="y", indvar="x", icond="x=0, y=2, df(y,x)=-4",
+           outfmt="text")
> div.on <- swset(s3, c(on="div"))
> sol.2t <- paste("y=", R2latex(s3, sub("^y=", "", ode.2t@solution)))
> writeLines(ltx.append.eol(unlist(sol.2t)))
```

$$y = e^{-x}\left(-\frac{1}{2}e^{4x} + \frac{5}{2}\right)$$

As an example of boundary conditions, consider $y'' = y$ with $y = a$ when $x = 0$ and $y = b$ when $x = 1$:

```
> ode.3 <- odesolve(s3, "df(y,x,2) = y",depvar="y", indvar="x",
+                   bcond=c("x = 0, y = a, x = 1, y = b"),
+                   outfmt="latex")
> writeLines(ltx.append.eol(unlist(ode.3@solution)))
```

$$y = \frac{-e^{2x}a + e^{2x}be + ae^2 - be}{e^x e^2 - e^x}$$

Note that we don't need the *usermap* argument here because there are no arbitrary constants when conditions are specified. The result is:

ODESOLVE options

Depending on the solution techniques chosen, *ODESOLVE* produces an explicit, implicit or parametric solution. An *explicit* solution is an equation with y isolated on the left, an *implicit* solution is an equation that determines y as one or more of its solutions and a *parametric* solution expresses both x and y in terms of some additional parameter[5]. *ODESOLVE* provides the following options which request a specific form:

EXPLICIT attempt to convert solutions to explicit form;

IMPLICIT attempt to convert a parametric solution to implicit form;

A further option, *basis*, requests the solution as a list of two elements. The first is a list of basis equations for the solution and the second is a specific solution. If conditions are applied, *basis* is ignored.

The option *TRODE* (TRace ODE) provides diagnostic information: classification of the ODE, intermediate results such as reduced equations. If conditions are specified it also returns the general equation, if any. The *redcas* function *odesolve* returns the diagnostic messages in the slot *msg* and the general solution, if any, in the slot *gen.solution*.

[4] a future release of odesolve may automate this if on="div" is specified

[5] for convenience we use y and x for the dependent and independent variables, respectively.

We can illustrate the use of *basis* and *trode* using the equation $y'' + 9y = sin(2x)$. If we use only the *trode* option and specify initial conditions $x = 0, y = 2$ we get both the specific (3.1) and general (3.2) solutions:

```
> odeopt1 <- odesolve(s3, "df(y,x,2) + 9*y = sin(2*x)", depvar="y", indvar="x",
+                     icond="x=0, y=2, df(y,x)=-4", opt="trode", outfmt="latex",
+                     usermap=list(ident=c("arbconst"="C"), index=c("C"="_")))
> writeLines(c("\\begin{gather}",
+              paste0(unlist(odeopt1@solution), "\\label{eq:ode-opt-1} \\\\"),
+              paste0(unlist(odeopt1@gen.solution), "\\label{eq:ode-opt-2}"),
+              "\\end{gather}"))
```

$$y = \frac{30\cos(3x) - 22\sin(3x) + 3\sin(2x)}{15} \tag{3.1}$$

$$y = \frac{5C_9\sin(3x) + 5C_8\cos(3x) + \sin(2x)}{5} \tag{3.2}$$

If we use the *basis* option we get the following result where

```
> odeopt2 <- odesolve(s3, "df(y,x,2) + 9*y = sin(2*x)", depvar="y", indvar="x",
+                     opt="basis", outfmt="latex")
> writeLines(c("\\begin{gather}",
+              ltx.append.eol(paste0(unlist(odeopt2@solution),
+              "\\label{eq:ode-opt-", 3:5, "}")),
+              "\\end{gather}"))
```

$$\cos(3x) \tag{3.3}$$

$$\sin(3x) \tag{3.4}$$

$$\frac{1}{5}\sin(2x) \tag{3.5}$$

equations (3.3) and (3.4) are the basis and (3.5) is a specific solution.

As we see in the following section, solving the Lagrange equation $v = 2uv' - a(v')^3$ produces a solution in parametric form. We can request the implicit form as follows:

```
> odeopt3 <- odesolve(s3, eqn="v = 2u*df(v,u) - a*df(v,u)^3",
+                     depvar="v", indvar="u", outfmt="latex", opt="implicit",
+                     switches=c(off="allfac, rounded",on="div,intstr"),
+                     usermap=list(ident=c("arbparam"="p", "arbconst"="C"),
+                     index=c("p"="_", "C"="_")), eol=FALSE)
> writeLines(c("\\begin{dmath}\\label{eq:ode-opt-7}",
+              paste0(unlist(odeopt3@solution), collapse=""),
+              "\\end{dmath}"))
```

$$-64C_{10}^3a^2 - 128C_{10}^2au^2 + 144C_{10}auv^2 - 64C_{10}u^4 - 27av^4 + 16u^3v^2 = 0 \tag{3.6}$$

We use the LaTeX *dmath* environment from the package *breqn* in case the solution needs to be broken across multiple lines. Since REDUCE output line size may cause equations to be split over multiple lines and

odesolve uses *eol=TRUE* we need to set the *eol=false* to avoid unnecessary splitting before *dmath* is called.

Switches affecting odesolve

Note that some switches can affect whether *odesolve* finds a solution. As an example, consider the Lagrange equation $v = 2uv' - a(v')^3$. If *DIV* and *INTSTR* are off and *ALLFAC* is on, which are the defaults for these switches, no solution is found. Toggling these switches does produce a solution.

```
> ode.x <- odesolve(s3, eqn="v = 2u*df(v,u) - a*df(v,u)^3",
+                   depvar="v", indvar="u", outfmt="latex",
+                   switches=c(off="allfac, rounded",on="div,intstr"),
+                   usermap=list(ident=c("arbparam"="P", "arbconst"="C"),
+                   index=c("P"="_", "C"="_")))
> writeLines(c("\\begin{eqnarray}",
+              ltx.append.eol(unlist(ode.x@solution)),
+              "\\end{eqnarray}"))
```

$$v = 2C_{11}P_1^{-1} + \frac{1}{2}P_1^3 a \tag{3.7}$$

$$u = C_{11}P_1^{-2} + \frac{3}{4}P_1^2 a \tag{3.8}$$

$$P_1 \tag{3.9}$$

In this case ODESOLVE has returned the solution in parametric form.

3.4.5 redLatex

With *redLatex* we can send expressions or the names of one or more REDUCE variables, arrays or matrices to the REDUCE process for display as LaTeX. The function is also called by other functions to convert their results to LaTeX when using the *outfmt=latex* argument. Note that because REDUCE lists use braces as list delimiters, *redLatex* cannot produce correct LaTeX output for lists. In this case use *red2Rlist* specifying *outfmt=text* to obtain an R list and then use *R2latex* to convert it to LaTeX.

Expressions

redLatex can convert mathematical expressions written in R or REDUCE syntax to LaTeX:

```
>     operators <- redExec(s3, c("operator !G, !T;", "korder kappa, !T;"))
>     writeLines(c("\\begin{eqnarray*}",
+                ltx.append.eol(redLatex(s3, c("!G(i,j)=kappa*!T(i,j)", "(x+y)**2"),
+                                        switches=c(off="exp"))[["tex"]]),
+                "\\end{eqnarray*}"))
```

$$G\left(i,j\right) = \varkappa T\left(i,j\right)$$
$$\left(x+y\right)^2$$

In order to treat G and R as functions it is necessary to first declare them as operators. Turning the EXP switch off ensures that $(x+y)^2$ is not expanded. Because the vector to display contains more than one element we need two provide a math array-type environment.

Arrays

redLatex can also print arrays of any dimension. REDUCE has no print method for arrays - instead a loop over each element of the array is needed. *redcas* provides a simple method which prints each element optionally preceded by the array name and element indices. As an example, consider the tensor

$$g_{ij} = \begin{bmatrix} -\omega_1 u^{-2} & 0 & 0 & 0 \\ 0 & (ux)^2 & 0 & 0 \\ 0 & 0 & g_{1,1} * (sinx)^2 & 0 \\ 0 & 0 & 0 & \omega_2 u^2 \end{bmatrix}$$

where (t, x, y, z) is a coordinate system, u is an arbitrary constant and ω is a vector. In REDUCE we define the tensor as follows:

```
> gm <- c("operator omega;",
+         "array g(3,3) ;",
+         "g(0,0) := -omega(1)*u^(-2);",
+         "g(1,1) := (u*z)^2;",
+         "g(2,2) := g(1,1) * (sin(x))^2;",
+         "g(3,3) := omega(2)*u^2;")
> o4 <- redExec(s3, gm)
```

Here the tensor is represented by an array. We now call *redLatex* to display g as LaTeX. Since we have a sparse tensor, we turn on the NERO switch to suppress printing the zeroes.

```
> g <- redLatex(s3, "g", use.names=TRUE, switches=c(on="nero"))[["tex"]]
> writeLines(paste0(g, collapse=(";\\quad&")))
```

$$g(0,0) = \frac{-\omega(1)}{u^2}; \quad g(1,1) = u^2 z^2; \quad g(2,2) = \sin(x)^2 u^2 z^2; \quad g(3,3) = \omega(2) u^2$$

Here we see that REDUCE variables with Greek letter names are automatically converted to the corresponding LaTeXcommand.

Displaying the indices for a tensor as arguments is not the standard notation. We now show how to define a mapping of the arguments to raised or lowered indices and to change the name of the g tensor to Γ. We construct a named list containing named vectors *ident* and *index*. An *ident* element's name specifies the current name of an identifier while the value is the new name. An *index* element's name specifies the identifier to convert from arguments to indices and the value is a series of underscores and circumflexes to indicate lowered and raised indices, respectively. Since the index map is applied after the ident map, it must use the mapped identifier names. This list is passed as the *usermap* argument to *redLatex*:

```
> gmap <- list(ident=c(g="\\\\Gamma"),
+              index=c("\\\\Gamma"="_^", "\\\\omega"="_"))
> g <- redLatex(s3, "g", use.names=TRUE, switches=c(on="nero"),
+               usermap=gmap)[["tex"]]
> writeLines(paste0(g, collapse=(";\\quad&")))
```

$$\Gamma_0^0 = \frac{-\omega_1}{u^2}; \quad \Gamma_1^1 = u^2 z^2; \quad \Gamma_2^2 = \sin(x)^2 u^2 z^2; \quad \Gamma_3^3 = \omega_2 u^2$$

We need to use four backslashes because the map is used as a regular expression which needs an escaped backslash and a further escaped backslash to get it into the string. Note that the usermap list does not have to have both elements.

Matrices

REDUCE has a print method for matrices so when *redLatex* is passed a REDUCE matrix name this method is called:

```
>   redmat <- c("operator omega;",
+              "gm:=mat((-omega(1)*u^2, u^2z^2),",
+              "        (sin(x)^2u^2z^2, omega(2)u^2));")
>   m1 <- redExec(s3, redmat);
>   writeLines(redLatex(s3, "gm", mathenv="equation*", usermap=gmap)[['tex']])
```

$$\begin{pmatrix} -\omega_1 u^2 & u^2 z^2 \\ \sin(x)^2 u^2 z^2 & \omega_2 u^2 \end{pmatrix}$$

This also works for plain text:

```
>   writeLines(redLatex(s3, "gm", outfmt="text")[['tex']])

[             2      2  2   ]
[ - omega(1)*u       u *z    ]
[                            ]
[      2  2  2              2]
[sin(x) *u *z    omega(2)*u  ]
```

Using LaTeX array-type environments

We may wish to display results from *redLatex* in a LaTeX *array*, *eqnarray*, *align* or similar environment. In our example in section 3.4.5, we have space for multi-column display, but *redLatex* does only single column. We can use the same approach as that in section 3.4.1 to produce multi-column output. Here we will use the *align* environment. It is important to call *redLatex* without the *mathenv* argument and, since we have space, not to set the NERO switch. We then construct a vector for each row, insert column separators and apply the *ltx.append.eol* function:

```
> otex <- redLatex(s3, "g", use.names=FALSE,
+                  usermap=list(ident=c(g="\\\\Gamma"),
+                               index=c("\\\\Gamma"="_^", "\\\\omega"="_")))[["tex"]]
> atex <- t(array(otex, c(4,4)))
> atex <- ltx.append.eol(
+     unlist(lapply(1:4,
+                   function(z){paste0(atex[z,], collapse=" & ")})))
> writeLines(c("\\[", "\\begin{array}{llll}",
+              atex,
+              "\\end{array}", "\\]"))
```

$$\begin{array}{llll} \frac{-\omega_1}{u^2} & 0 & 0 & 0 \\ 0 & u^2 z^2 & 0 & 0 \\ 0 & 0 & \sin(x)^2 u^2 z^2 & 0 \\ 0 & 0 & 0 & \omega_2 u^2 \end{array}$$

3.4.6 Switch handling

All *redcas* functions which support the *switches* argument use the functions *swset* and *swreset* to set desired switch states for the duration of the *redcas* function's execution. Both of these functions are exported and can be called by the user to set switch states for as long as needed and restore them as necessary.

Suppose we wish to make multiple calls to *odesolve* with the switches DIV and INTSTR off and ALLFAC[6] on we call *swset* as follows:

```
> toggled <- swset(s3, c(on="div, intstr", off="allfac"), swlist="demo")
> print(toggled)

[1] "intstr" "allfac"
```

The elements of the vector *toggled* are the names of the switches which were toggled. The argument *swlist* names a REDUCE list in which to store the names of the toggled switches. After completing the calls to *odesolve* we can, if desired, restore the switches to their previous state:

```
> swreset(s3, swlist="demo")
```

If we use the default value for the *swlist* argument we should avoid using the *switches* argument until after calling *swreset* because it uses the default. We should also avoid using *redExec* to call the *ON* or *OFF* statements as mixing the two methods can lead to unexpected results.

[6] odesolve is currently not vectorized and the default states of these switches can lead to no solution be found when one exists.

Chapter 4

Conversion functions

This chapter discusses the *redcas* functions which convert REDUCE output into corresponding R objects. With the exception of *R2latex* and *red2Rarray*, all conversion functions convert or transform REDUCE output using only R code. *R2latex* uses REDUCE as a translation tool from plain text to LaTeX. *red2Rarray* requires some specific formatting of the arrays to be converted and so needs to call REDUCE.

4.1 R2latex

This function uses REDUCE to transform a numeric, character or expression vector, a data frame or a possibly nested list to LaTeX and returns the output in a list of the same structure. List elements of type other than those mentioned are returned unchanged. *R2latex* uses the *redLatex* function to do the transformation. Note that for vectors *redLatex* can be used directly.

```
> op <- redExec(s4, "operator c, f;")
> tmap <- list(ident=c(abc="a"), index=c(f="_"))
> t1 <- R2latex(s4, list("y=c(3) + c(2)*x + e**x",
+                        data.frame(v1=c("f(1)**(-1)", "exp(-i*pi)"),
+                                   v2=c("exp(i*pi)", "f(2)**2"),
+                                   v3=c("pi", "gamma")),
+                        list(expression(f(1)**(-1), f(2)**2),
+                             "1/abc^6+1/abc^5"),
+                        matrix(c("sqrt(x)", "x**y", "gamma", "pi"),
+                        nrow=2, ncol=2)),
+                        usermap=tmap)
> str(t1)

List of 4
 $ : chr "y=c\\left(3\\right)+c\\left(2\\right) x+e^x"
 $ :'data.frame':        2 obs. of  3 variables:
  ..$ v1: chr [1:2] "\\frac{1}{f_{1}}" "-1"
  ..$ v2: chr [1:2] "-1" "f_{2}^2"
  ..$ v3: chr [1:2] "\\pi" "\\gamma"
 $ :List of 2
  ..$ : chr [1:2] "\\frac{1}{f_{1}}" "f_{2}^2"
  ..$ : chr "\\frac{a+1}{a^6}"
 $ : chr [1:2, 1:2] "sqrt(x)" "x**y" "gamma" "pi"
```

Note that the matrix is returned without conversion because this type is not supported by *R2latex*.

We can display the data frame in a math array environment by collapsing its rows to produce a single vector

with separator, and use *ltx.append.eol* to add end of line commands:

```
> tvec <- ltx.append.eol(apply(t1[[2]],1,paste0,collapse=" & "))
> writeLines(c("\\[", "\\begin{array}{lll}", tvec, "\\end{array}", "\\]"))
```

$$\begin{array}{lll}\frac{1}{f_1} & -1 & \pi \\ -1 & f_2^2 & \gamma\end{array}$$

4.2 red2Rarray

This function converts REDUCE arrays to R arrays. The arrays must exist in the REDUCE session because *red2Rarray* uses the REDUCE operator GETTYPE to determine which elements of the input are arrays and the operator LENGTH to get their dimensions. As an example, consider the following array with 3 dimensions of length 2 each:

$i,j,1$		
	$i,1,k$	$i,2,k$
$1,j,k$	111	121
$2,j,k$	211	221

$i,j,2$		
	$i,1,k$	$i,2,k$
$1,j,k$	112	122
$2,j,k$	212	222

First we build this array in REDUCE, convert it to R in plain text format and print.

```
> a3d <- redin('array a3d(1,1,1) ;
+              for i:=1:2 do for j:=1:2 do for k:=1:2 do
+                  a3d(i-1,j-1,k-1) := 100*i + 10*j + 1*k ;')
> a3d.red <- redExec(s4, a3d)
> a3d.R <- red2Rarray(s4, "a3d")[[1]]
> mode(a3d.R) <- "numeric"
> a3d.R

, , 1

     [,1] [,2]
[1,]  111  121
[2,]  211  221

, , 2

     [,1] [,2]
[1,]  112  122
[2,]  212  222
```

With some transformations we can present results of the conversion in LaTeX form with the two sub-arrays side by side. We need to adjust for the fact that REDUCE array indices are zero-based. We also need to use the zero space argument to the LaTeX end of line command in order to avoid the

```
! Illegal unit of measure (pt inserted)
```

error which would otherwise be triggered by the brackets in the R output.

```
> a3d.ltx <- red2Rarray(s4, "a3d", outfmt="latex")[[1]]
> k1 <- gsub(" +", " & ", capture.output(print(a3d.R[,,1]), type="output"))
```

```
> k2 <- gsub(" +", " & ", capture.output(print(a3d.R[,,2]), type="output"))
> writeLines(c("\\[", "\\begin{array}{lrr}",
+              ltx.append.eol(k1, ,eol="\\\\[0pt]"), "\\end{array}",
+              "\\hspace{1cm}",
+              "\\begin{array}{lrr}",
+              ltx.append.eol(k1, ,eol="\\\\[0pt]"), "\\end{array}",
+              "\\]"))
```

$$
\begin{array}{lrr}
 & [,1] & [,2] \\
[1,] & 111 & 121 \\
[2,] & 211 & 221
\end{array}
\qquad
\begin{array}{lrr}
 & [,1] & [,2] \\
[1,] & 111 & 121 \\
[2,] & 211 & 221
\end{array}
$$

4.3 red2Rfn

This function converts algebraic expressions returned by REDUCE into R functions with an argument for each variable in the expression. As an example consider the output of *redFunc* for integration (INT) and differentiation (DF).

```
> flist <- c('int','df')
> plist <-list(c('sin(x)**4','x'), c('y^(3*x)', 'x', 'y'))
> isx <- redFunc(s4, func=flist, param=plist)
> for (i in names(isx)) writeLines(isx[[i]])
                   3
  - 2*cos(x)*sin(x)  - 3*cos(x)*sin(x) + 3*x
-----------------------------------------------
                      8
    3*x
 3*y   *(3*log(y)*x + 1)
-------------------------
            y
```

The output is in the "natural" form which is why we used *writeLines* to display the result. In order to convert to a function we need to execute with NAT off:

```
> (isx <- redFunc(s4, func=flist, param=plist, switches=c(off='nat')))

$int
[1] "( - 2*cos(x)*sin(x)**3 - 3*cos(x)*sin(x) + 3*x)/8"

$df
[1] "(3*y**(3*x)*(3*log(y)*x + 1))/y"

> isx.fn <- red2Rfn(unlist(isx), name=c("xint", "xdf"))
> isx.fn

$xint
function (x)
{
    return((-2 * cos(x) * sin(x)^3 - 3 * cos(x) * sin(x) + 3 *
        x)/8)
}
```

```
<environment: 0x38fc6948>

$xdf
function (y, x)
{
    return((3 * y^(3 * x) * (3 * log(y) * x + 1))/y)
}
<environment: 0x38fc6948>
```

We can assign names to the functions either in the *redFunc* or *red2Rfn* call. Names assigned by *redFunc* are used in *red2Rfn*. To make the functions available in the current environment we can use the following:

```
> for (f in names(isx.fn)) assign(f,isx.fn[[f]])
```

We used the names *xint* and *xdf* to avoid clashing with existing R functions.

4.4 red2Rlist

red2Rlist takes as input a character vector containing REDUCE output and converts any REDUCE lists found in the vector to R lists. REDUCE lists, which may be recursive, are enclosed in braces ({, }). For this reason *red2Rlist* produces only plain text output because of the difficulty distinguishing LaTeX braces from REDUCE list braces. See below for how to obtain LaTeX output.

As an example of converting a REDUCE list, consider the following character vector which contains arbitrary text, a REDUCE informational message and three REDUCE lists of various depths. The lists are outputs from the *redSolve* or *odesolve* functions.

```
>   writeLines(redls)

Some arbitrary text
and list starts here: {{x=2.21*i + 1.77,y= - 2.21*i + 1.77,z= -3.54},
{x=0.58*i - 1.27,y= - 0.58*i - 1.27,z=2.54},
{x= - 0.58*i - 1.27,y=0.58*i - 1.27,z=2.54},
{x= - 2.21*i + 1.77,y=2.21*i + 1.77,z=-3.54}} and we have text at the end$

*** depend v , u

{v=arbconst(1)/e**(a*u)}$

{{q=myfun1(p^2+1,exp(r)), p=cos(r), r=myf2(a,b,c)},{
q=myfun1(p^2+1,exp(-r)), p=cos(-r), r=myf2(a,b,c,d+e,f)}}
```

Calling *red2Rlist* on this input returns an R list with three elements, one for each REDUCE list in the input. Notice that the structure of the R lists is identical to that of the REDUCE lists.

```
> str(Rls <- red2Rlist(redls))

List of 3
 $ :List of 4
  ..$ :List of 3
  .. ..$ : chr "x=2.21*i + 1.77"
```

```
  .. ..$ : chr "y= - 2.21*i + 1.77"
  .. ..$ : chr "z= -3.54"
  ..$ :List of 3
  .. ..$ : chr "x=0.58*i - 1.27"
  .. ..$ : chr "y= - 0.58*i - 1.27"
  .. ..$ : chr "z=2.54"
  ..$ :List of 3
  .. ..$ : chr "x= - 0.58*i - 1.27"
  .. ..$ : chr "y=0.58*i - 1.27"
  .. ..$ : chr "z=2.54"
  ..$ :List of 3
  .. ..$ : chr "x= - 2.21*i + 1.77"
  .. ..$ : chr "y=2.21*i + 1.77"
  .. ..$ : chr "z=-3.54"
 $ :List of 1
  ..$ : chr "v=arbconst(1)/e**(a*u)"
 $ :List of 2
  ..$ :List of 3
  .. ..$ : chr "q=myfun1(p^2+1,exp(r))"
  .. ..$ : chr " p=cos(r)"
  .. ..$ : chr " r=myf2(a,b,c)"
  ..$ :List of 3
  .. ..$ : chr "q=myfun1(p^2+1,exp(-r))"
  .. ..$ : chr " p=cos(-r)"
  .. ..$ : chr " r=myf2(a,b,c,d+e,f)"
```

To produce a list formatted as LaTeX we call *R2latex* on the output of *red2Rlist*. Since we will display the result in an *eqnarray* environment we use the *tochar=TRUE* argument so that the sublists containing only character elements are converted to character vectors and we can add new line commands. For simplicity consider the first sublist of the last element of the previous example with the definitions of q, p and r. The *R2latex* argument *eol* allows us to specify that an end of line command be appended to all but the last element[1]. First we show the LaTeX code:

```
> writeLines(
+     lx <- c("\\begin{eqnarray*}",
+             R2latex(s4, red2Rlist(redls[8:9], tochar=TRUE)[[1]], eol=TRUE),
+             "\\end{eqnarray*}"))

\begin{eqnarray*}
q=myfun1\left(p^2+1,e^r\right)\\
p=\cos\left(r\right)\\
r=myf2\left(a,b,c\right)
\end{eqnarray*}
```

We can display this in typeset form:

```
> writeLines(lx)
```

[1] using *ltx.append.eol*, which adds end of line commands to all but the last element requires a vector as input. Since the default output from red2Rlist contains only vectors of length one, *ltx.append.eol* has no effect without setting *tochar=TRUE*.

$$q = myfun1\left(p^2 + 1, e^r\right)$$
$$p = \cos\left(r\right)$$
$$r = myf2\left(a, b, c\right)$$

4.5 Miscellaneous conversion functions

redcas exports several conversion functions which are used internally as these could be useful in processing output produced directly in REDUCE. For completeness the functions and a brief description of their purpose follows.

dlmMatch : searches a character vector for matching delimiter pairs (brace, bracket, parenthesis) and returns a data frame with the coordinates of matching open/close pairs and the level of nesting.

fnMatch : searches a character vector for function calls and returns a list containing a character vector of the function calls and a character vector of the function calls with commas replaced by a user-specified character. This is used by *red2Rlist* to prevent function calls being interpreted as lists.

redlsExtract : searches a character vector for REDUCE lists and returns a list with one character vector element for each REDUCE list found.

sublist2char : searches a list for elements containing only character types and returns a list with these elements converted to character vectors.

redDropOut : removes blank lines or the submit block end marker from the REDUCE output.

redExpand : a vector of commands to send to REDUCE may contain the name of one or more files containing REDUCE commands. *redExpand* returns a vector containing the commands up to the element containing the file name, the contents of the file and the elements following the file name.

redSplitOut : converts a vector of REDUCE output into a list containing a vector of commands and a vector of command output.

Bibliography

[1] REDUCE Developers. *What is REDUCE?* [Online; accessed 2026-10-10]

[2] REDUCE Developers. *How to Obtain and Run REDUCE* [Online; accessed 2026-10-10]

[3] REDUCE Developers. *REDUCE User's Manual, Revision 7419, September 26, 2026* [Online; accessed 2026-10-10]

[4] REDUCE Developers. *The SOLVE Operator, Revision 7419, September 26, 2026* [Online; accessed 2026-10-10]

[5] REDUCE Developers. *The DF Operator, Revision 7419, September 26, 2026* [Online; accessed 2026-10-10]

[6] REDUCE Developers. *ODESOLVE: Ordinary Differential Equation Solver, Revision 7419, September 26, 2026* [Online; accessed 2026-10-10]

[7] REDUCE Developers. *Switch Summary, Page last modified on Tuesday 2nd January 2024* [Online; accessed 2026-10-10]

[8] David Stoutemyer, Arthur Norman, Francis Wright *Tutorials, Page last modified on Thursday 1st October 2026* [Online; accessed 2026-10-10]

Appendix A

Appendices

A.1 Known issues

A.1.1 MacOS/Intel/REDUCE 6866

For 6866/CSL on MacOS (Intel) some calculations can cause REDUCE to abort because of an issue with garbage collection. This has been reported to the REDUCE developers and is specific to CSL. This issue can be mitigated by using the $-k$ option to allocate more memory to the CSL process. In the case where we found the issue, test program 07-redExecLong.R, allocating 2.5GB worked:

```
sessionID <- redStart("csl", options="-k2.5g")
```

If you encounter this issue you will have to determine what amount of memory suffices.

A.1.2 Passing FACTOR, MODULAR in switches argument

Passing FACTOR in the *switches* argument leads to an error:

```
***** nil invalid as kernel
Stopping.
No list called 'sw!-to!-toggle' exists in REDUCE session.
```

but turning on factor in REDUCE and calling R2latex works. Alternatively set EXP to off.

Passing MODULAR in the *switches* argument leads to an error:

```
***** Zero divisor
Stopping.
dlmMatch: No delimiter '{' found.
redlsExtract: stopping.


***** No switches specified
Stopping.
No list called 'asltx!-to!-toggle' exists in REDUCE session.
No list called 'redltx!-sw!-set' exists in REDUCE session.
```

Curiously, the state of MODULAR is changed. The issue here is that the modulus must first be set using *redExec* to call the SETMOD declaration first:

```
redExec(id, "setmod n;")
```

A.1.3 Error displaying omicron in LaTeX

When producing LaTeX output REDUCE converts variables which are Greek letter names to the corresponding LaTeX command. An exception is *omicron* which produces the following error message:

```
***** unknown print item
```

Any subsequent output as LaTeX also produces this error. The only solution is to close the session.

If PSL is run independently of *redcas* and the package TMPRINT is not loaded, this is not an issue.

A.1.4 Long equations in odesolve

A.1.5 trode option in odesolve

If the option *trode* is specified and *verbose* is not, ODESOLVE writes the first line of *trode* messages on a line starting with a command number.

A.1.6 REDUCE is mostly case insensitive

By default REDUCE ignores case in identifiers and converts them to lower case on output. This can lead to issues when using mixed case for a REDUCE variable name and then, in R, searching the output using the mixed case version. One way to avoid this is to avoid using upper case. However, if upper case letters in REDUCE code are escaped using *!* they are preserved. When referring to these variables in R, for example in the *usermap* argument, do not escape upper case.

Another example is when producing LaTeX to produce Greek upper case letters. REDUCE converts lower case names of Greek letters to the lower case Greek letter, e.g. *gamma* to γ, while capitalized names are converted to the capital letter, eg *Gamma* to Γ. The feature REDUCE provides for using non-standard characters in variable names, escaping the character with an exclamation mark, can be used to keep the case of the following character:

```
13: write gamma, " ", Gamma ;
gamma gamma
14: write gamma, " ", !Gamma ;
gamma Gamma
```

Note that using *redLatex* or *R2latex* to convert mathematical expressions to LaTeX upper case characters and braces in the expression must be escaped.

A.1.7 redin, lstlisting and Sweave don't mix

Using the *redin* function in an *lstlisting* environment which uses text marker line ranges with the *includerangemarker=false* options to drop the *Schunk, Sinput* and *Soutput* begin and end commands causes the exclusion of these markers to fail.

A.2 Testing

The aim of testing is to prove that REDUCE output is correctly transformed to corresponding R objects. No attempt is made to test the correctness of the REDUCE output as REDUCE itself provides extensive testing. Since it is unreasonable to expect CRAN to provide a REDUCE installation, package tests which execute REDUCE code are skipped when no REDUCE executables are found. Prior to CRAN submission all tests are run for various combinations of R and REDUCE versions for both Linux (5.15 and 6.6) and MacOS (Intel and arm M5).

A.2.1 Test categories

The *redcas* package comprises a small number of functions which execute REDUCE code but the majority use only R. Tests of functions which execute REDUCE code are skipped if no REDUCE executable is found. There are 1004 tests grouped as shown in Table A.1. The R column shows the number of tests which do

Table A.1: Tests

Group	Tests	CSL	PSL	R
01 Finding Reduce executables	11	0	0	11
02 Session handling	24	0	0	24
03 Output handling	20	0	0	20
04 Output: file not found	6	3	3	0
05 Execution	123	54	37	32
06 Execution: prompts	5	5	0	0
07 Execution: long-running	4	4	0	0
08 Execution: Many redExec calls	16	8	8	0
09 Arrays	72	36	36	0
10 Expressions	36	18	18	0
11 asltx from REDUCE	52	26	26	0
12 redLatex	109	64	37	8
13 Bug fixes	22	15	7	0
14 redSolve	88	40	31	17
15 redcas Utils	170	44	0	126
16 odesolve	76	39	0	37
17 red2Rfn	24	0	0	24
18 redFunc	68	34	34	0
19 Calculus	45	15	0	30
20 Polynomial	33	25	0	8
All Groups	1004	430	237	337

not require a REDUCE installation. The CSL and PSL columns show the number of tests which would be executed if the corresponding dialect is present and a REDUCE session can be started. Groups for which the number of tests in both the CSL and PSL columns is non-zero, are run for each dialect which is present. Groups for which the number of tests in the CSL column is non-zero and in the PSL column zero, are only executed for one dialect. In this case if both are present only CSL is used.

Some tests require only R for execution and these are always run. Since CRAN does not have a REDUCE installation, these are the only tests which are run there. Therefore we recommend installing the tests and executing locally. In the directory where the tests are run, the file *99-test-results.Rout* provides a summary summary test results for each group and overall:

```
  Group: All Groups
  Tests: 1004
 Passed: 337
```

```
 Failed: 0
Skipped: 667
```

If any tests have failed, this program calls *stop()* after printing the summary to draw attention to the failures. In this case the transcript is named *99-test-results.Rout.fail*.

A.2.2 Pre-CRAN testing and results

Prior to submission to CRAN this release has been tested so that test groups requiring REDUCE are also run to ensure everything passes. Table A.2 shows the combinations of R and REDUCE releases tested for Linux kernel 6.6 (L6), Linux kernel 5.15 (L5), MacOS Intel 12 (Mi) and MacOS arm 26 (Ma). Note that

Table A.2: R and REDUCE releases tested

	7335	7131	6866	6860	6658
R-devel	-	L5	-	L5	L5
R-4.6.1	L6	L5	Mi, Ma	L5	L5, Mi, Ma
R-4.5.3	-	L5	-	L5	L5, -
R-4.5.1	-	-	Mi	L5	L5, Mi

revision 7335 is named as Snapshot 2026-03-08 on the REDUCE website. Prior snapshots are named for the revision. All tests passed except for the following, all of which also fail when run directly in reduce:

6866/CSL on MacOS (Intel) one test, red7ExeLong-001 failed on the issue reported in section A.1.1 for a work-around.

6658 Linux CSL/PSL: redLtxTyp-rok-003/007, redLtxTyp-gen-003/012, 007/016. Variable *gamma* is rendered as Γ in LaTeX. Corrected in later versions of REDUCE.

6658 MacOS Intel and arm CSL: redLtxTyp-rok-003, redLtxTyp-gen-[003,007]. Variable *gamma* is rened-ered as Γ in LaTeX. Corrected in later versions.

A.2.3 Post-installation testing

We recommend executing the tests after installation. While *99-test-results.Rout* shows the expected and actual values if the test fails, it is not very convenient for checking exactly how the two differ. For each set of tests there is an RData file containing two lists, expect and actual. To find this, open the Rout file for the failing category - the RData file name is in the statement which prints and saves the results, for example, in *170-many-input.Rout* the line

```
> print(save.results('redMany',expect,actual))
```

indicates that *redMany.RData* contains the lists.

A.3 Producing this vignette

Since it is unreasonable to expect REDUCE to be installed on CRAN, the Sweave step for this vignette has been executed prior to submission outside the package directory and the resulting LaTeX file copied to the *vignettes* directory as *Using.redcas.Rnw*. The actual *Rnw* file is copied as *Using.redcas.Rnw.src* and the output of *Stangle* as *Using.redcas.R.src*.