Introduction to the redcas package

Martin Gregory

October 21, 2024

Contents

1 Overview

redcas is an interface to REDUCE , a 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 LaTeXformat. This document shows some basic examples of using *redcas*. The examples are trivial in order to concentrate on the use of the package but explanation of some aspects of REDUCE are provided for those not familiar with it. REDUCE is written in Lisp and can run on various Lisp dialects, the most common being CSL and PSL. *redcas* can use either of these two. We show how *redcas* finds the REDUCE executables (section 2), how to start a session, define and display expressions and close the session (section 3), control output (section 4), how to solve a system of equations (section 6) and how to produce LaTeX output (section 7). We also describe some known issues (appendix A) and the tests supplied with the package (appendix C).

2 Finding 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 [1]. There are four different ways that *redcas* can find the REDUCE executables (*redcsl, redpsl*). These are, in order of precedence:

1. explicitly specifying the path using the *dirpath* argument of *redStart*;
2. the environment variable *REDUCE_EXEC* exists and contains the path to a directory containing one or both executables;
3. the R option *reduce_exec* exists and contains the path to a directory containing one or both executables;
4. the executable is located in a directory listed in the *PATH* environment variable.

3 Controlling a REDUCE session from R

A REDUCE session is started using the *redStart* function. While the session is open, the *redExec*, *redSolve* or *asltx* 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. Finally, the *redClose* function shuts down the REDUCE session and optionally retrieves the complete transcript. *redSolve* and *asltx* will be covered in sections 6 and 7, respectively.

3.1 Starting REDUCE

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 off by default as this is the REDUCE default. *redStart* provides the *echo* argument to turn the switch on at the start of the session so that commands will appear in the REDUCE transcript. *redStart* returns the identifier for the session if successful, otherwise it terminates with *stop*. The identifer is required by other *redcas* functions which communicate with the REDUCE session.

3.2 Executing REDUCE code

To execute REDUCE code we use the *redExec* function which takes as required arguments the identifier returned by *redStart* and a character vector containing REDUCE commands. As an example consider a call with a single REDUCE command:

```
> o1 <- redExec(s1, "r2d2 := (987654321/15)^3;", drop.blank.lines=TRUE)
```

If *redExec* fails it returns *FALSE*. Otherwise it returns a list 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;

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

We used the *drop.blank.lines* option to make the output more compact - the REDUCE transcript uses blank lines generously. We can see the result by displaying the *out* element of *o1*:

```
> writeLines(o1[["out"]])
           35682160321981318855871043
r2d2 := -----------------------------
                       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;",
+          "r2d2;",
+          "off rounded;")
> redExec(s1, cmd, drop.blank.lines=TRUE)[["out"]]
[1] "2.85457282576e+23"
```

3.3 Closing REDUCE

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
```

See Appendix B for a discussion of the REDUCE transcript for this section.

4 Controlling output

In this section we consider some REDUCE features which control how expressions are displayed or simplified. As an example consider the function $f(x, y) = (x + y)^2 + 1/(x + y)$ which can be displayed in various ways. Some are:

$$\begin{aligned} f(x,y) &= \frac{(x+y)^3+1}{(x+y)} \\ &= \frac{x^3+3x^2y+3xy^2+y^3+1}{(x+y)} \\ &= (x+y)^2+(x+y)^{-1} \\ &= (x^3+3x^2y+3xy^2+y^3+1)(x+y)^{-1} \end{aligned}$$

In addition to ROUNDED, the relevant REDUCE switches are EXP, FACTOR[1], MCD, GCD, ALLFAC, RAT, DIV and LIST. 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*.

In this section we cover the EXP and MCD switches. For details of the others, see the Switch Summary [2], the REDUCE Manual [3] or the second of the REDUCE interactive lessons [4]. In the previous section we used a numeric example. In this and the next sections we deal with symbolic entities, the primary reason why REDUCE is of interest. 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 indeterminate 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 indeterminate:

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

First we start a new REDUCE session and assign the bound variable *f*:

[1]turning FACTOR on turns EXP off automatically without notification

```
>   s2 <- redStart(echo=TRUE)
>   expoff <- redExec(s2, "off exp;")
>   f <- redExec(s2, "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 by default REDUCE 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 also turn the switch NAT off to get

```
>   redExec(s2, 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. For parsing results generated by redExec in R this is much more suitable than with NAT on. Specialized functions such as *asltx* and *redSolve* handle this internally.

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

```
> writeLines(h <- redExec(s2, "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* as 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:

```
> writeLines(redExec(s2, 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(s2, 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.

5 Sourcing REDUCE files: IN

The REDUCE command *IN* is the equivalent of the R *source* function. In contrast to the R function, the REDUCE command has only two modes of operation: whether the commands are echoed or not:

```
  in "some.file.red" ;  %% commands from some.file.red are echoed
  in "some.file.red" $  %% commands from some.file.red are not echoed
```

The IN statement suppresses the printing of statement numbers. This means that the *redExec* with the *split* option cannot distinguish commands from output. If we need to split we can use the *file:* syntax in the input vector:

```
  commands <- c("q1:=(x+y);", "file:some.file.red", "q2:=q1^2")
  redExec(s2, x=commands)
```

Using this approach redExec inserts the contents of *some.file.red* into the vector between the assignments of *q1* and *q2* before submitting to REDUCE , effectively bypassing the use of the IN statement.

6 Solving a system of equations

The REDUCE operator *solve* [5] 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. Here we first use *redExec* to call *solve*:

```
> 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:

```
> 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 is 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 not be what is intended. The object has another slot *rsolutions* which contains evaluated versions of the *solutions* slot, if these are numbers including complex, 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
```

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. For convenience, *redSolve* has a *switches* argument to allow turning switches on or off before calling *solve*:

```
> 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.

7 Producing LaTeX output

The function *asltx* converts an array or expression previously defined in the REDUCE session to LaTeXusing REDUCE features. If provided with appropiate mappings it can translate object names or convert array arguments to indices and can enclose the result in a math environment specified by the user.

As an example of an expression, consider the variable h which we used in section 4.

```
>   s4 <- redStart()
>   f <- redExec(s4, "f := (x + y)^2;", drop.blank.lines=TRUE)[["out"]]
>   h <- redExec(s4, "h := f + 1/f;", drop.blank.lines=TRUE)[["out"]]
```

First with MCD off:

```
> dummy <- redExec(s4, "off mcd;")
> writeLines(asltx(s4, "h", mathenv="equation*")[["tex"]])
```

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

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. With MCD on the same call returns:

$$h = \frac{x^4 + 4\,x^3\,y + 6\,x^2\,y^2 + 4\,x\,y^3 + y^4 + 1}{x^2 + 2\,x\,y + y^2}$$

Note that *asltx* is currently not vectorized.

asltx can also print arrays of any dimension. Unlike matrix printing in REDUCE it does not attempt to display arrays of two dimensions as a matrix. Instead it prints each element of the array along with its 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("on nero; off exp;",
+         "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(s4, gm)$out
```

where the tensor is represented by an array. Since we have a sparse tensor, we have turned on the NERO switch to suppress the zeroes. We now call *asltx* to display g as LATEXby using the *mode* argument:

```
> writeLines(asltx(s4, "g", mathenv="dmath*", mode="fancy")[["tex"]])
```

$$g(0,0) = \frac{-\omega\,(1)}{u^2}$$

$$g(1,1) = u^2\,z^2$$

$$g(2,2) = \sin\,(x)^2\;u^2\,z^2$$

$$g(3,3) = \omega\,(2)\;u^2$$

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

Finally, displaying the indices for a tensor using arguments is not the standard. We now show how to define a mapping of the arguments to raised or lowered indices. We construct a list[2] containing a named vector called *index* with name being the identifier and the value a series of underscores and circumflexes to indicate lowered and raised indices, respectively. This list is passed as the *usermap* argument to *asltx*:

[2]because it may also contain a map for changing variable names

```
>     writeLines(asltx(s4, "g", usermap=list(index=c(g="_^", "\\\\omega"="_")),
+                         mathenv="dmath*", mode="fancy")[["tex"]])
```

$$g_0^0 = \frac{-\omega_1}{u^2}$$
$$g_1^1 = u^2\, z^2$$
$$g_2^2 = \sin{(x)}^2\; u^2\, z^2$$
$$g_3^3 = \omega_2\, u^2$$

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.

Suppose we now want to change the name of g to Γ: we add a named vector called *ident* to the list with the old name as element name and new name as element value:

```
> writeLines(asltx(s4, "g", mathenv="dmath*", usermap=list(ident=c(g="\\\\Gamma"),
+                              index=c("\\\\Gamma"="_^", "\\\\omega"="_")))[["tex"]])
```

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

Note that the index element names must use the mapped identifier name.

In our example, we have space for multi-column display, but *asltx* does only single column. We can do multi-column, for example, using the *align* environment, by calling *asltx* without a mathenv value and use R to format the output appropriately. We add alignment marks before the equals sign and add some space:

```
> otex <- asltx(s4, "g",
+               usermap=list(ident=c(g="\\\\Gamma"),
+                           index=c("\\\\Gamma"="_^", "\\\\omega"="_")))[["tex"]]
> otex <- sub("=", "&=", otex)
> writeLines(c("\\begin{align}",
+             sprintf("%s      &%s\\\\", otex[1], otex[2]),
+             sprintf("%s      &%s",   otex[3], otex[4]),
+             "\\end{align}"))
```

$$\begin{align}
\Gamma_0^0 &= \frac{-\omega_1}{u^2} \qquad & \Gamma_1^1 &= u^2\, z^2 \tag{1}\\
\Gamma_2^2 &= \sin{(x)}^2\; u^2\, z^2 \qquad & \Gamma_3^3 &= \omega_2\, u^2 \tag{2}
\end{align}$$

References

[1] REDUCE Developers. *How to Obtain and Run REDUCE* [Online; accessed 2024-08-18]

[2] REDUCE Developers. *Switch Summary* [Online; accessed 2024-08-29]

[3] REDUCE Developers. *REDUCE Manual* [Online; accessed 2024-08-29]

[4] David Stoutemyer, Arthur Norman, Francis Wright *Introductory tutorial* [Online; accessed 2024-08-29]

[5] REDUCE Developers. *The SOLVE Operator* [Online; accessed 2024-09-05]

A Known issues

A.1 redExec hangs

A.1.1 Operator prompt

Because REDUCE is started via a pipe, the session is interactive so if you forget to declare an operator, REDUCE prompts and waits for the user to respond. redExec traps this, terminates the REDUCE session, informs of the problem and writes the complete REDUCE transcript to the current working directory.

A.1.2 Missing statement terminators

If you forget to insert a statement terminator[3] on the last statement, the *redExec* function might[4] not return. To avoid this situation, *redExec* checks whether the last non-blank, non-comment element of the input vector has a terminator. If not it stops with an appropriate message and returns FALSE.

Forgetting a terminator on a statement other than the last will most likely cause an error rather than hanging, but hanging may still happen. You will know if the function will not return if it writes a message

```
x seconds elapsed, reduce commands still executing.
```

every 10 seconds (by default). In this case you must interrupt the function and call *redClose*. Against this eventuality you can set the *timeout* argument to a value other than zero.

A.2 Parsing REDUCE output

Splitting the REDUCE transcript into commands and output is not trivial. Some issue are described in the following sections

A.2.1 Using the OUT statement

The REDUCE OUT statement redirects output to a named file. *redExec* does not currently check for this. If you use this option, none of the commands or output sent to the file will be returned. Furthermore, if you do not turn off the redirection in the same call to *redExec*, the end of submission marker[5] will not be seen and *redExec* with fail to return.

A.2.2 Using the OUTPUT switch

If the REDUCE OUTPUT switch is turned off, all output is suppressed. In order to prevent *redExec* hanging, be sure to turn it back on at the latest in the last statement in the call.

A.2.3 Using the IN statement

The REDUCE IN statement suppresses the printing of statement numbers. This means that the *redExec split* option will write both command and output to the *out* element of the returned list and only comments will be written to the *cmd* element. Using the *file:* syntax in the input vector instead of an IN statement avoids this problem.

A.2.4 Commands with trailing comments

If a command has a comment after the terminator and ECHO is on, REDUCE prints it on a separate line. If the command produces output, the comment comes after the output, for example

[3] semi-colon or dollar sign

[4] for example a LET statement without terminator hangs, but ON or OFF without terminator just generates an error message

[5] see appendix **??** for details

```
df(x^2, x) ; % a demonstrative comment
```

produces

```
2: df(x^2, x) ;
2*x
 % a demonstrative comment
```

This is handled correctly by *redExec* but may be confusing when viewing the transcript.

A.3 asltx issues

asltx may cause LaTeX compilation to fail with either of the following messages:

```
! Double subscript.
! Double superscript.
```

LaTeXdoes not permit more than one subscript or superscript command on a single item. For example,

```
\Gamma_{1}^{2}_{3}
```

will fail. One situation where this could occur is when a variable with a raised index is also raised to a power, for example when the argument in `x(2)^3` is converted to a raised index. *asltx* handles this situation by converting to {x{^2}}^3. There may be other situations which cause this error.

B The REDUCE transcript

In this section we describe the full REDUCE transcript from section 3.

Session start

```
 1 Reduce (CSL, rev 6860), 11-Aug-2024 ...
 2
 3 1:
 4 2:
 5 lisp_dialect
 6
 7 swget
 8
 9 swtoggle
10
11 asltx
12
13 exprltx
14
15 itoa
16
17 array2flatls
18
19 arrayltx
20
21 asltx_marker
22
23 3:
24
25 4: write "=====> submit number ", 0, " done <=====" ;
26 =====> submit number 0 done <=====
```

The first part of the transcript, up to the line containing *submit number 0 done* is generated by the call to *redStart* which calls *redExec* to define some REDUCE functions required by the *redcas* package. Lines 5-21 contain the names of these procedures. For reasons related to the use of a pipe to control the

REDUCE session, all code submitted writes a marker line to the transcript to indicate the execution is complete. The marker command and output are lines 25 and 26, respectively and these are not returned by *redExec*.

First redExec call

```
30
31 6: r2d2 := (987654321/15)^3;
32          35682160321981318855871043
33 r2d2 := ----------------------------
34                     125
35
36
37 7: write "=====> submit number ", 1, " done <=====" ;
38 =====> submit number 1 done <=====
```

The results of the first (explicit) *redExec* call are shown in lines 30-38. *redExec* splits these into commands and output by checking whether there is a command number or not. The marker command and output are not returned.

Second redExec call

```
42
43 9: on rounded;
44
45 10: r2d2;
46 2.85457282576e+23
47
48
49 11: off rounded;
50
51 12: write "=====> submit number ", 2, " done <=====" ;
52 =====> submit number 2 done <=====
```

The results of the second *redExec* call are shown in lines 42-52.

C Testing

The packages *redcas* comes with 328 tests grouped as shown in Table 1. The 46 tests in the “R” column

Table 1: Tests

Group	Tests	CSL	PSL	R
01 Finding Reduce executables	11	0	0	11
02 Session handling	15	0	0	15
03 Output handling	17	0	0	17
04 Output: file not found	6	3	3	0
05 Execution	78	39	39	0
06 Execution: prompts	3	3	0	0
07 Execution: long-running	3	3	0	0
08 Execution: Many redExec calls	16	8	8	0
09 Arrays	34	17	17	0
10 Expressions	8	4	4	0
11 asltx from REDUCE	40	20	20	0
12 asltx from R	21	9	9	3
13 Bug fixes	22	15	7	0
14 redSolve	54	27	27	0
All Groups	328	148	134	46

do not require a REDUCE installation. Of the remaining 282 tests, 134 are executed for both CSL and PSL if they are both present while 14 tests are executed for CSL only. If either or both CSL and

PSL executables are absent, the relevant tests are skipped. In the directory where the tests are run, the file *99-test-results.Rout* provides a summary of the results of these tests.

This release has been tested with R 4.4.1 using REDUCE releases 6860 and 6558 for both CSL and PSL on Linux, and 6558 and 6339 for CSL on MacOS.

For 6339, 13 tests failed because of white space differences and, because the expression used is rational, differences in the number of dashes in the line separating numerator and denominator:

```
6860 and 6558:            6339:
           -1                          - 1
g(0,0) := ----            g(0,0) := ------
            2                          2
           u                          u
```

While the testing program checks for white space only differences, the dashes are a problem. Possibly turning the MCD switch off might solve the issue.

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.