Vignette for `R` package `rodeo`

David Kneis, david.kneis [at] tu-dresden.de

October 11, 2016

Contents

1 When to use this package

The `rodeo` package facilitates the implementation of ODE-based models. These are models that describe the dynamics of a set of n state variables by integrating a set of n ordinary differential equations. The package is particularly useful in conjunction with the `deSolve` package (http://cran.r-project.org/package=deSolve) providing numerical solvers for initial value problems. The advantages from using `rodeo` are:

- Models are defined using plain tabular text files or spreadsheets. Thus, the model is formulated independent from source code. This facilitates documentation, portability, and re-use.
- You are forced to provide the model in stoichiometry matrix notation (see http://en.wikipedia.org/wiki/Petersen_matrix). Although this is a restriction, it is a very useful one and benefit is almost guaranteed.
- Owing to the matrix notation, redundant terms are largely eliminated from the differential equations. This contributes to comprehensibility and increases computational efficiency. The stoichiometry matrix can also be visualized to better communicate the model to users or non-modelers.
- `rodeo` provides a code generator which supports `R` and `Fortran` as target languages. Using compiled `Fortran` can speed up numerical integration by 1 or 2 orders of magnitude (compared to plain `R`).
- Code can be generated for an arbitrary number of sections (e.g. control volumes in a spatially discretized model). This allows even partial differential equations (e. g. reactive transport problems) to be tackled by means of semi-discretization (see http://en.wikipedia.org/wiki/Method_of_lines).

2 Example problem

The functioning of the package is illustrated with simple model of bacteria growth in a continuous flow stirred tank reactor (CFSTR) also known as chemostat. In is assuemed that the bacteria grow on a single resource (e. g. a source of organic carbon) which is imported via the reactor's inflow (Fig. 1). Due to mixing, the reactors contents is spatially homogeneous, hence the density of bacteria as well as the concentration of the substrate are scalars.

Changes in bacteria density are due to (1) resource-limited growth and (2) wash-out from the reactor. The substrate concentration is controlled by (1) the inflow as well as (2) the consumption by bacteria. A classical Monod term was used to model the resource dependency of bacteria growth. For the sake of simplicity, the external forcings (i. e. flow rate and substrate load) are held constant.

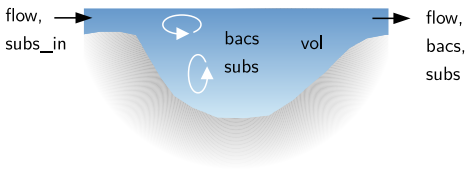


Figure 1: Sketch of considered system.

Table 1: Declaration of state variables.

name	unit	description
bacs	cells/ml	bacteria density
subs	mg/ml	substrate concentration

Using `rodeo`, the model can be described using just tabular text files (Tables 1 – 5). These files are shipped with the package and can be loaded with `R`'s `data` method.

Table 2: Declaration of parameters.

name	unit	description
mu	1/hour	intrinsic bacteria growth rate
half	mg/ml	substrate level where bacteria grow at 50% of max. rate
yield	cells/mg	cells produced per amount of substrate
vol	ml	volume of reactor
flow	ml/hour	rate of through-flow
subs_in	mg/ml	substrate concentration in inflow

Table 3: Declaration of functions being reference in process rate expressions and/or stoichiometric factor expressions.

```
 name unit                              description
monod    - monod expression for resource limitation
```

Table 4: Specification of processes.

```
   name       unit           description                    expression
growth  cells/hour bacteria growth       mu * monod(subs, half) * bacs
renewal 1/hour     rate of water renewal flow/vol
```

Table 5: Specification of stoichiometric factors.

```
variable process       expression
    bacs growth  1
    bacs renewal -bacs
    subs growth  -1 / yield
    subs renewal (subs_in - subs)
```

3 Basic use

3.1 Creating and inspecting a model object

We start by loading the package and the example data.

```
library(rodeo, quietly=TRUE)
# Load sample data frames (contents shown above)
data(vars, pars, pros, funs, stoi)
```

Then, a new object is created with `new`. This requires us to supply the name of the class, data frames for initialization, as well as the spatial dimensions. Here, we create a single-box model (one dimension with no sub-units).

```
# Instantiate new object
model <- rodeo$new(vars=vars, pars=pars, funs=funs,
  pros=pros, stoi=stoi, dim=c(1))
```

To inspect the object's contents, we can use the following:

```
# Automatic print method
print(model)
# Show stoichiometry information as a matrix
print(model$stoichiometry())
```

3.2 Defining functions and supplying data

In order to work with the object, we need to define functions that are referenced in the process rate expressions or stoichiometric factors (i. e. the ODEs' right hand sides). For non-autonomous models, this includes the definition of forcings which are functions of a special argument with the reserved name 'time'. See Sect. 4.3 for details.

For the example, we need just one simple function (see https://en.wikipedia.org/wiki/Monod_equation).

```
monod <- function(c,h) { c / (c + h) }
```

We also need to set the values of parameters and state variables (initial values) using the dedicated methods `setPars` and `setVars`. Since we deal with a single-box model, parameters and initial values can be stored in ordinary vectors.

```
model$setVars(c(bacs=0.01, subs=0))
model$setPars(c(mu=0.8, half=0.1, yield= 0.1,
  vol=1000, flow=50, subs_in=1))
```

3.3 Computing the stoichiometry matrix

Having defined all functions and having set the values of variables and parameters, one can compute the stoichiometric factors. In general, explicitly computing these factors is not necessary, it may be helpful in debugging however. To do so, the `stoichiometry` method needs to be supplied with the index of the spatial box (only relevant for multi-box models) as well as a time value (in the case of non-autonomous models).

```
m <- model$stoichiometry(box=1, time=0)
print(signif(m, 3))
```

```
         bacs subs
growth   1.00  -10
renewal -0.01    1
```

The stoichiometry matrix is also a good means to communicate a model because it shows the interactions between processes and variables in a concise way. How the stoichiometry matrix can be visualized graphically is demonstrated in Sect. 4.4.2.

3.4 Translating the model into source code

In order to use the model for simulation, we need to transfer it into source code. This is also known as *code generation*. Specifically, we want the code generator to create a function that returns the derivatives of the state variables with respect to time. In addition to the derivatives, the generated function also returns the values of all process rates (as diagnostic variables).

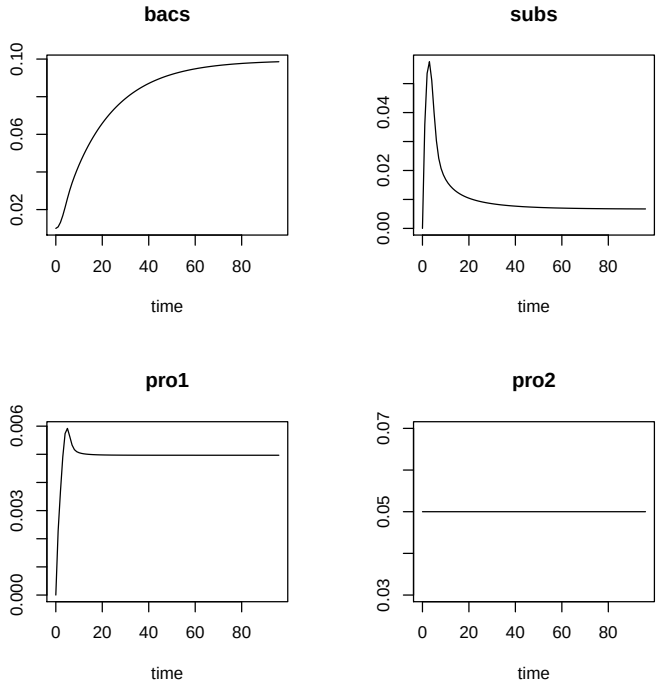
After generating the code, we need to make it executable. In `R`, we can use a combination of `eval` and `parse`. Alternatively, the generated code could be loaded with `source` after exporting it to a file (e.g. using `write`). The latter method is needed if one wants to inspect the generated code (or even modify it, which rarely makes sense).

```
code <- model$generate(name="derivs",lang="r")
derivs <- eval(parse(text=code))
```

3.5 Solving the ODE system

We are now ready to compute the evolution of the state variables over time by means of numerical integration. Note that the initial values (argument `y` of `deSolve::ode`) and parameters (argument `p`) are set to the output of the model object's dedicated 'get' methods. This is to make sure that the order of values in the two arrays is consistent with the generated code.

```
library(deSolve)
out <- deSolve::ode(y=model$getVars(useNames=TRUE), times=0:96, func=derivs,
  parms=model$getPars())
layout(matrix(1:4, ncol=2, byrow=TRUE))
plot(out, mfrow=NULL)
layout(1)
```



In addition to the dynamics of the state variables, the numerical solver also outputs the dynamics of the process rates. This is valuable information in general and it also facilitates debugging.

4 Advanced topics

4.1 Spatially distributed systems (multi-section models)

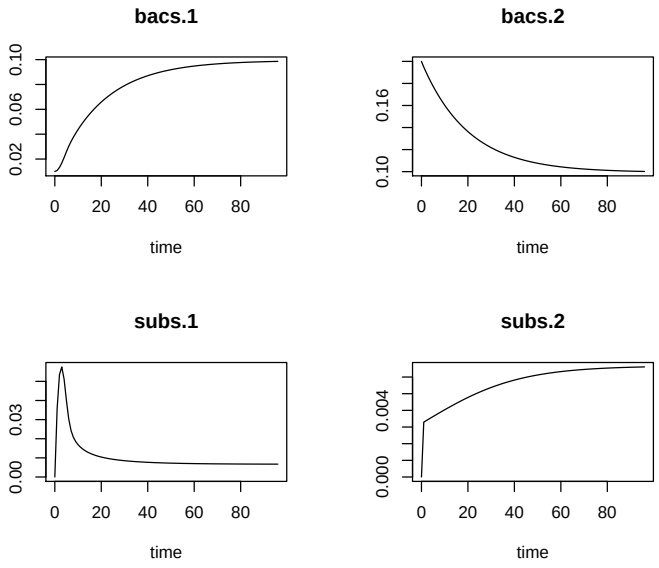
A single-box case has been considered so far. We will now extend the model for multiple boxes, i. e. for a collection of (isolated) reactors. To keep it simple, we use just a few boxes in a single spatial dimension. First, we need to create a model object with the desired dimensions and resolution. Second, the code needs to be re-generated to reflect the altered dimension. Third, initial values and parameters need to be specified as arrays to account for the spatial discretization. For a model with one spatial dimension, we must use two-dimensional arrays, i. e. matrices. Here, we initialize every modeled reactor with a different bacteria density.

```
nBox <- 2
model <- rodeo$new(vars=vars, pars=pars, funs=funs,
  pros=pros, stoi=stoi, dim=c(nBox))


code <- model$generate(name="derivs",lang="r")
derivs <- eval(parse(text=code))


v <- cbind(
  bacs= seq(from=0.01, to=0.2, length.out=nBox),
  subs= rep(0, nBox)
)
model$setVars(v)
p <- c(mu=0.8, half=0.1, yield= 0.1,
  vol=1000, flow=50, subs_in=1)
model$setPars(matrix(rep(p, each=nBox), nrow=nBox,
  dimnames=list(NULL, names(p))))


out <- ode(y=model$getVars(useNames=TRUE), times=0:96, func=derivs,
  parms=model$getPars())
layout(matrix(1:(model$lenVars()*nBox), nrow=nBox, byrow=TRUE))
plot(out, which=1:(model$lenVars()*nBox), mfrow=NULL)
layout(1)
```



4.2 Increasing performance by means of `Fortran`

Real-world models usually consist of many and lengthy mathematical expressions. Also, depending on the studied problem, the ODE solver may need to use (very) short time steps. Then, computation times become of serious concern. In those time-critical cases, it is recommended to generate source code for a fast, compilable language rather than for (slower) `R`. The compilable language supported by `rodeo` is `Fortran`.

To generate code to compute the state variables' derivatives in `Fortran`, one would use:

```
code <- model$generate(name="derivs",lang="f95")
# Optionally display generated code
#cat(code)
```

The generated `Fortran` subroutine with assumed name `derivs` has a simple, quite universal interface

```
subroutine derivs(time, var, par, NLVL, dydt, pro)
```

In order to use the numerical solvers from the packages http://cran.r-project.org/package=deSolve or http://cran.r-project.org/package=rootSolve, however, a different interface is required

```
subroutine derivs (neq, t, y, ydot, yout, ip)
```

and an additional subroutine for parameter initialization (`initmod`) must to be supplied as well (see the `deSolve` vignette http://cran.r-project.org/web/packages/deSolve/vignettes/compiledCode.pdf, page 6). Consequently, a suitable wrapper code must be written.

In order to make the use of `Fortran` as simple as possible, the `rodeo` package provides a high-level class method `compile` that combines

1. generation of the basic `Fortran` code via the `generate` method (see above),
2. generation of wrapper code for compatibility with `deSolve` and `rootSolve`,
3. compilation of all `Fortran` sources into a shared library using the `R CMD SHLIB` command,
4. clean-up of any intermediate files from compilation.

The `compile` method takes as argument the name of a file holding the `Fortran` implementation of functions being referenced in the particular model's mathematical expressions (consult Sect. 5 for guidelines). This can actually be a vector of file names if the source code is split.

```
lib <- model$compile(sources="functionsCode.f95")
```

The return value of `compile` is a vector of character strings holding the name of the generated library (in element `libName`), the full file path of the library (in element `libFile`) as well as the name of the callable subroutine within that library (in element `libFunc`).

A suitable `Fortran` implementation of the functions used in the example (contents of file 'functionsCode.f95') is shown below. Note that all the functions are collected in a single `Fortran` module with implicit typing turned off. The name of this module (`functions`) is mandatory and cannot be changed. Note that a module can import other modules which helps to structure the source code. Also note that the user-supplied source files need to reside in directories with write-access to allow the creation of intermediate files during compilation.

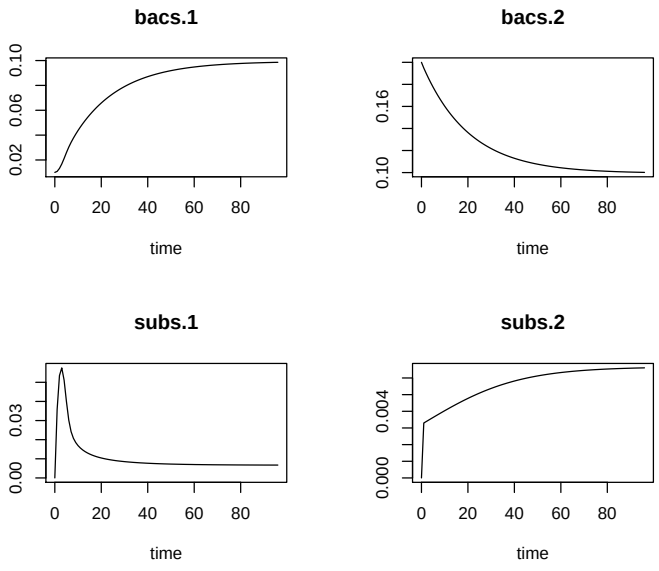
```
module functions
   implicit none
   contains

   double precision function monod(c, h)
     double precision, intent(in):: c, h
     monod= c / (c + h)
   end function

 end module
```

We are now prepared to load the shared library and run the simulation based on the `Fortran` code. Note the additional arguments `dllname`, `initfunc`, and `nout` being passed to the numerical solver (open the help page for `lsoda` to see the documentation for them). Setting a wrong value for `nout` easily makes `R` crash.

```
v <- cbind(
  bacs= seq(from=0.01, to=0.2, length.out=nBox),
  subs= rep(0, nBox)
)
model$setVars(v)
p <- c(mu=0.8, half=0.1, yield= 0.1,
  vol=1000, flow=50, subs_in=1)
model$setPars(matrix(rep(p, each=nBox), nrow=nBox,
  dimnames=list(NULL, names(p))))
dyn.load(lib["libFile"])
out <- ode(y=model$getVars(useNames=TRUE), times=0:96,
  func=lib["libFunc"], parms=model$getPars(), dllname=lib["libName"],
  initfunc="initmod", nout=model$lenPros()*prod(model$getDim()))
layout(matrix(1:(model$lenVars()*nBox), nrow=nBox, byrow=TRUE))
plot(out, which=1:(model$lenVars()*nBox), mfrow=NULL)
layout(1)
dyn.unload(lib["libFile"])
invisible(file.remove(lib["libFile"]))
```



4.3 Forcings (time-varying parameters)

4.3.1 Two alternative options

In general, there are two options for dealing with time-variable forcings:

functions-of-time: In this approach one needs to define the forcings as functions of a single argument representing time. In `rodeo` this argument must have the reserved name `time`. Use of this approach is most convenient if the forcings are easily described as parametric functions of time (e.g. seasonal change of solar radiation). It can also be used with tabulated time series data, but this requires some extra coding. In any case, it is essential for restrict the integration step size of the solver (e.g. using the `hmax` argument of `deSolve::lsoda`) so that short-term variations in the forcings cannot be 'missed'.

stop-and-go: In this approach forcings are implemented as normal parameters. To allow for their variation in time, the ODE solver is interrupted every time when the forcing data change. The solver is then re-started with the updated parameters (i.e. forcing data) using the states computed in the previous call as initial values. Hence, the calls to the ODE solver must be embedded within a time-loop. With this approach, setting a limit on the solver's integration step size (through argument `hmax`) is not required since the solver is interrupted at the 'critical times' anyway.

In real-world applications, the 'stop-and-go' approach is often simpler to use and the overhead due to interruption and re-start of the solvers seems to be rather small. It also facilitates the generation of useful traceback information in case of exceptions (e.g. due to corrupt time series data).

4.3.2 The 'functions-of-time' approach with `Fortran` models

This section demonstrates how the 'functions-of-time' approach can be used in `Fortran`-based models assuming that information on forcings is stored in delimited text files. Such files can be created, for example, with any spreadsheet software, data base system, or `R`. Assume that we have time series of two meteorological variables exported to a text file 'meteo.txt':

```
dat <- data.frame(time=1:10, temp=round(rnorm(n=10, mean=20, sd=3)),
  humid=round(runif(10)*100))
write.table(x=dat, file="meteo.txt", col.names=TRUE,
  row.names=FALSE, sep="\t", quote=FALSE)
print(dat)
```

```
   time temp humid
1     1   12    19
2     2   18    20
3     3   21    32
```

```
4     4   23    74
5     5   21    13
6     6   20    51
7     7   19    41
8     8   23    38
9     9   22    27
10   10   20    97
```

We can now call `forcingFunctions` to generate the appropriate forcing function in `Fortran`. In this example, we request linear interpolation via the method's `mode` argument.

```
dat <- data.frame(name=c("temp","humid"),
  column=c("temp","humid"), file="meteo.txt", mode=-1, default=FALSE)
code <- forcingFunctions(dat)
write(x=code, file="forc.f95")
# Optionally inspect generated code
# cat(code)
```

In order to use the generated code, it is necessary to

1. write it to disk (e. g. using `write` as above),
2. declare all forcings as functions in `rodeo`'s respective input table,
3. insert the statement `use forcings` at the top (e. g. line 2) of the `Fortran` module `functions`,
4. pass the generated file to the compiler along with all other `Fortran` source files.

The following `Fortran` code demonstrates how the user-defined forcings can be tested/debugged outside of the `rodeo` environment. The shown utility program can be compiled, for example, using a command like

```
gfortran <generated_module_file> <file_with_program> -o test
```

Note that the subroutines `rwarn` and `rexit` are available automatically if the code is used to build a shared library with `R CMD SHLIB`, i. e. the subroutines must not be defined then.

```
! auxiliary routines for testing outside R
subroutine rwarn(x)
  character(len=*),intent(in):: x
  write(*,*)x
end subroutine
```

```
subroutine rexit(x)
  character(len=*),intent(in):: x
  write(*,*)x
  stop
end subroutine

! test program
program test
use forcings  ! imports generated module with forcing functions

implicit none

integer:: i
double precision, dimension(5):: times= &
  dble((/ 1., 1.5, 2., 2.5, 3. /))

do i=1, size(times)
  write(*,*) times(i), temp(times(i)), humid(times(i))
end do
end program
```

4.4 Generating model documentation

4.4.1 Exporting formatted tables

One can use e.g. `exportDF` to export the object's basic information in a format which is suitable for inclusion in HTML or LaTeX documents. The code section

```
# Select columns to export
df <- model$getVarsTable()[,c("tex","unit","description")]
# Define formatting functions
bold <- function(x){paste0("\\textbf{",x,"}")}
mathmode <- function(x) {paste0("$",x,"$")}
# Export
tex <- exportDF(x=df, tex=TRUE,
  colnames=c(tex="symbol"),
  funHead=setNames(replicate(ncol(df),bold),names(df)),
  funCell=list(tex=mathmode)
)
cat(tex)
```

generates the following LaTeX code

```
\begin{tabular}{lll}\hline
  \textbf{symbol} & \textbf{unit} & \textbf{description} \\ \hline
  $bacs$ & cells/ml & bacteria density \\
  $subs$ & mg/ml & substrate concentration \\ \hline
\end{tabular}
```

holding tabular information on the model's state variables. To include the result in a document one needs to write the generated LaTeX code to a file for import with either the `input` or `include` directive. Things are even simpler if the `Sweave` pre-processor is used (as in the case of this vignette file). The above `R` code can then be embedded in the LaTeX code between the special markers `<<echo=FALSE, results=tex>>=` and `@` resulting in the following output.

symbol	**unit**	**description**
bacs	cells/ml	bacteria density
subs	mg/ml	substrate concentration

Alternatively, a `markdown` compatible dataframe can be generated and used with the `knitr` function `kable`. This will work with html, pdf or even Word (.docx) output. The following code section would create a table of the model's state variables (output not shown).

```
to_markdown <- function(dat, which_cols){
  cols <- which(names(dat) %in% which_cols)
  for(i in cols){
    dat[, i] <- ifelse(dat[, i] != "", paste0("$", dat[, i], "$"), "")
    }
  return(dat)
}
ids <- model$getVarsTable()[,c("tex", "unit", "description")]
names(ids) <- c("Symbol", "Unit", "Description")
kable(to_markdown(ids, which_cols=c("Symbol")),
  caption= "State variables")
```

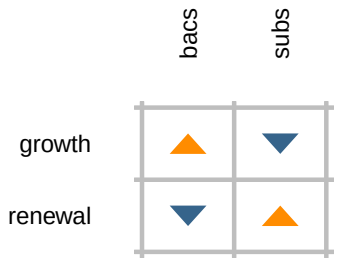
Thanks to Andrew Dolman for the latter example.

4.4.2 Visualizing the stoichiometry matrix

A graphical representation of the stoichiometry matrix is often a good means to communicate a model. To create such a graphics, one typically wants to replace the stoichiometry factors' numeric values by symbols encoding their sign only.

Option 1: Plain R graphics One can use the class method `plotStoichiometry` to visualize the matrix using standard R graphic facilities as demonstrated below. In practice, one needs to fiddle around a bit with the dimensions of the plot and the font size to get an acceptable scaling of symbols and text. Also, it is hardly possible to nicely display row and column names containing special things like sub- or superscripts.

```
model$plotStoichiometry(box=1, time=0, cex=0.3)
```



Option 2: TEX The following example generates suitable code for inclusion in LaTeX documents.

```
signsymbol <- function(x) {
  if (as.numeric(x) > 0) return("\\textcolor{orange}{$\\blacktriangle$}")
  if (as.numeric(x) < 0) return("\\textcolor{cyan}{$\\blacktriangledown$}")
```

```
  return("")
}
rot90 <- function(x) { paste0("\\rotatebox{90}
  {$",gsub(pattern="*", replacement="\\cdot ", x=x, fixed=TRUE),"$}") }
m <- model$stoichiometry(box=1, time=0)
tbl <- cbind(data.frame(process=rownames(m), stringsAsFactors=FALSE),
  as.data.frame(m))
tex <- exportDF(x=tbl, tex=TRUE,
  colnames= setNames(c("",model$getVarsTable()$tex[match(colnames(m),
    model$getVarsTable()$name)]), names(tbl)),
  funHead= setNames(replicate(ncol(m),rot90), colnames(m)),
  funCell= setNames(replicate(ncol(m),signsymbol), colnames(m)),
  lines=TRUE
)
tex <- paste0("%\n% THIS IS A GENERATED FILE\n%\n", tex)
# write(tex, file="/home/dkneis/temp/stoichiometry.tex")
```

The contents of the variable `tex` must be written to a text file and this file is then imported in LaTeX with the `input` directive. The result looks as follows:

	bacs	*subs*
growth	▲	▼
renewal	▼	▲

Option 3: HTML The following example generates suitable code for inclusion in HTML documents.

```
signsymbol <- function(x) {
  if (as.numeric(x) > 0) return("&#9651;")
  if (as.numeric(x) < 0) return("&#9661;")
  return("")
}
m <- model$stoichiometry(box=1, time=0)
tbl <- cbind(data.frame(process=rownames(m), stringsAsFactors=FALSE),
  as.data.frame(m))
html <- exportDF(x=tbl, tex=FALSE,
  colnames= setNames(c("Process",model$getVarsTable()$html[match(colnames(m),
    model$getVarsTable()$name)]), names(tbl)),
  funCell= setNames(replicate(ncol(m),signsymbol), colnames(m))
)
html <- paste("<html>", html, "</html>", sep="\n")
# write(html, file="/home/dkneis/temp/stoichiometry.html")
```

To test this, one needs to write the contents of the variable `html` to a text file and open that file in a web browser. In some cases, automatic conversion of the generated HTML into true graphics formats may be possible, e. g. using auxiliary tools like `html2ps` and `convert` (on Linux systems).

Option 4: Markdown A `markdown` compatible can be generated with the `knitr` function `kable` as shown below (contributed by Andrew Dolman; output not displayed).

```
signsymbol <- function(x) {
  if (as.numeric(x) > 0) return("$\\blacktriangle$")
  if (as.numeric(x) < 0) return("$\\blacktriangledown$")
  return("")
}
stoi_mat <- model$stoichiometry(box=1, time=0)
stoi_mat <- data.frame(apply(stoi_mat, MARGIN = c(1, 2), signsymbol))
stoi_mat <- setNames(stoi_mat, paste0("$",
  model$getVarsTable()$tex[match(colnames(stoi_mat),
  model$getVarsTable()$name)], "$"))
stoi_mat <- cbind(Process=rownames(stoi_mat), stoi_mat)
kable(stoi_mat, row.names= FALSE, caption= "Stoichiometric matrix")
```

5 Writing `rodeo`-compatible `Fortran` functions

5.1 Reference example

As a reference, the following example code can be used which declares a function of two arguments. Comments have been added to explain the individual statements. In `Fortran`, comments are generally initiated with the exclamation mark (!). They may appear right after statements or on separate lines.

```
1  double precision function FUNCNAME (ARG1, ARG2) ! declare the function
2  implicit none                                   ! force declarations
3  double precision, intent(in):: ARG1, ARG2       ! declare arguments
4  double precision:: LOCAL                        ! declare local var.
5  double precision, parameter:: CONST=1.d0        ! declare local const.
6  LOCAL= ARG1 * CONST + ARG2                      ! local computation(s)
7  FUNCNAME= LOCAL                                 ! set return value
8  end function                                    ! closes the function
```

For compatibility with `rodeo`, the function result must be a scalar of type `double precision` (a floating point number of typically 8 byte). There are several ways to achieve this but the simplest and recommended syntax is put the type declaration `double precision` right before the function's name (line 1). Then, the return value must be set by an assignment to the function's name (line 7). This is best done at a single location in the body code, typically at the very end.

It is a good habit to always put `implicit none` in the first line of the function body (line 2). This is to disable so-called implicit typing (a rather

dangerous technique of automatic data type assignment). With this statement, all arguments (line 3) and local variables or constants (lines 4 and 5) need to be explicitly declared. The repetition of the argument's names in lines 1 and 3 may be a bit annoying (but one can use copy and paste). All declarations need to be made at the top of the function's body (right after the `implicit none`) before any other statements.

In `Fortran`, identifier names are not case-sensitive (as opposed to `R`). This applies to the name of the function itself as well as to the names of arguments and local variables or parameters.

Note: It is actually sufficient to use the `implicit none` statement at the beginning of the module that contains all function declarations (see example in Sect. 4.2). Repetition of the statement in the individual functions does not do any harm, however.

5.2 Common pitfalls

5.2.1 Double precision variables and constants

`Fortran` has several types to represent floating point numbers that vary in precision but `rodeo` generally uses the type `double precision`. Thus, any local variables and parameters should also be declared as `double precision`. To declare a numeric constant of this type, e. g. 'pi', one needs to use the syntax `3.1415d0`, i. e. the conventional 'e' in scientific notation is replaced by 'd'. An alternative but less portable syntax exists but it is not mentioned here.

```
double precision, parameter:: pi= 3.1415d0, e= 2.7183d0   ! math constants
double precision, parameter:: kilograms_per_gram = 1.d-3  ! 1/1000
double precision, parameter:: distance_to_moon = 3.844d+5 ! 384400 km
```

Note the `parameter` keyword used to inform the compiler that the declared item(s) are constants rather than variables.

5.2.2 Integers in numeric expressions

It is recommended to avoid integers in arithmetic expressions as the result may be unexpected. Use `double precision` constants instead of `integer` constants or, alternatively, explicitly convert `integer` constants to `double precision` by means of the `dble` intrinsic function.

```
average= (value1 + value2) / 2d0       ! does not use an integer at all
average= (value1 + value2) / dble(2)   ! explicit type conversion
```

It is often even better not to use any literal constants, leading to a code like

```
double precision, parameter:: TWO= 2.d0
! possibly other statements
average= (value1 + value2) / TWO
```

Using uppercase names for constants is a widespread habit but this is a matter of style only.

5.2.3 Continuation lines

Source code lines should not exceed 80 characters (though some `Fortran` compilers support longer lines). If an expression does not fit on a single line, the ampersand (`&`) must be used to indicate continuation lines. It is recommended to put the `&` at the end of any unfinished line as in the following example:

```
a = term1 + term2 + &
    term3 + term4 + &
    term5
```

Missing `&` characters are a frequent cause of compile time errors sometimes being rather obscure.

5.3 More information on `Fortran` programming

The examples in Sect. 4.2 may serve as a starting point. The website http://fortranwiki.org/fortran/show/HomePage is a good source of additional information, providing links to standard documents, books, etc.

6 Practical issues

6.1 Managing tabular input data

The tabular input data can be held in either plain text files or spreadsheets. The two alternatives have their own pros and cons as described below:

Spreadsheet software is specialized on displaying tabular data in a convenient way. Nevertheless, there are some practical issues. For example, spreadsheet programs allow for cell formatting. Formatting rules and automatisms often lead to unwanted results (e.g. in the case of logical columns). This also depends on the used format (e.g. 'xlsx' or 'odt'). As of now, R does not have native support for spreadsheets in 'odt' format. Conversion between 'odt' and 'xlsx' is possible, but the exact result seems to be unpredictable.

Delimited text files are much simpler in concept. They can be edited with any editor and platform dependence is a minor issue because line endings and character encodings are easy to change. Good programs provide powerful editing commands, e.g. using regular expressions, and they can highlight matching parenthesis. Text files are perfect for use with version control systems. Nevertheless, normal editors are unable to display tabular data in a nice way.

It is a good compromise to store the tabular data in delimited text files (e.g. separated by TAB or semicolon) and to open them either in a spreadsheet program or editor, depending on the actual task. Note that the conventional 'csv' is not recommended since mathematical expressions involving multi-argument functions and text descriptions may contain commas (hence, they need to be quoted). Using TAB-delimited text is probably the best option.