Vignette for `R` package `rodeo`

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

February 3, 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/web/packages/deSolve/index.html`) 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 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`).
- The generated source code is applicable to an arbitrary number of boxes (control volumes). 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 best illustrated by an example. In this document, we consider a continuous flow stirred tank reactor (CFSTR) containing a degradable organic matter (concentration `c_z`) and dissolved oxygen, `c_do` (Fig. 1). Due to mixing, the reactors contents is spatially homogeneous, hence the concentrations are scalars. Changes in concentrations are triggered by

1. the loads in the reactor's inflow,
2. aerobic degradation of the organic matter (1st order) which consumes dissolved oxygen,
3. exchange of oxygen with the atmosphere.

Some specific aspects of the model are as follows:

- The dependence of aerobic degradation on available oxygen is described with a Monod term (Michaelis-Menten model).
- The rate of oxygen exchange between water and atmosphere is described as an empirical function of water depth and wind speed.

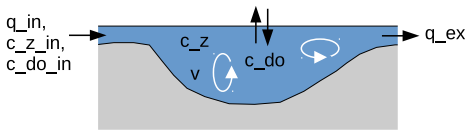


Figure 1: Sketch of considered system.

Table 1: Declaration of identifiers (data set 'identifiers'). The type column indicates whether an item is a state variable (v), parameter (p), or function (f).

name	type	unit	description	tex	html
c_do	v	mg/l	dissolved O2	c_{DO}	c_{DO}
c_z	v	mg/l	degradable org. matter Z	c_Z	c_Z
v	v	m3	water volume in reactor	v	v
q_in	p	m3/s	inflow rate	q_{in}	q_{in}
q_ex	p	m3/s	outflow rate	q_{ex}	q_{ex}
kd	p	1/d	decay rate	k_d	k_d
s_do_z	p	g/g	stoichiometry (O2/Z)	s_{DO,Z}	s_{DO,Z}
h_do	p	g/m3	half-saturation O2	h_{DO}	h_{DO}
temp	p	degC	temperature	T	T
wind	p	m/s	wind speed	W	W
depth	p	m	water depth	D	D
O2sat	f	mg/l	O2 saturation, f(temp)	O2_{sat}	O2_{sat}
ka	f	1/d	aeration, f(wind, depth)	k_a	k_a
monod	f	-	monod model	monod	monod
c_z_in	f	g/m3	Z in inflow, f(time)	c_{Z,in}	c_{Z,in}
c_do_in	f	g/m3	O2 in inflow, f(time)	c_{DO,in}	c_{DO,in}

- The model does *not* assume steady flow conditions, hence the volume of the reactor is a state variable too.

Using `rodeo`, the model can be described using just tabular text files (Tables 1 – 3). These files are shipped with the package and can be loaded with `R`'s `data` method.

Table 2: Specification of processes (data set 'processes').

name	unit	description	expression
flow	m3/s	water balance	q_in - q_ex
flushing	1/s	flushing rate	q_in / v
decay	g/m3/s	decay of z	kd * c_z * monod(c_do, h_do)
aeration	g/m3/s	O2-exchange	ka(wind,depth) * (O2sat(temp) - c_do)

Table 3: Specification of stoichiometric factors (data set 'stoichiometry').

variable	process	expression
v	flow	1
c_z	flushing	c_z_in(time) - c_z
c_z	decay	-1
c_do	flushing	c_do_in(time) - c_do
c_do	decay	-s_do_z
c_do	aeration	1

3 Basic use

3.1 Creating and inspecting a model object

We start by creating a new object with `new`. This requires us to supply the name of the class as well as a set of data frames for initialization.

```
library(rodeo, quietly=TRUE)
# Load sample data frames (contents shown above)
data(identifiers, processes, stoichiometry)
# Instantiate new object
model= new("rodeo", vars=subset(identifiers,type=="v"),
  pars=subset(identifiers,type=="p"),
  funs=subset(identifiers,type=="f"),
  pros=processes, stoi=stoichiometry)
```

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

```
# Built-in method
model$show()
# Show stoichiometry information as a matrix
print(model$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 model's mathematical expressions. This includes the forcings which are functions of a special argument with the reserved name 'time'. See Sect. 4.3 for details.

```
# 'normal' functions
O2sat= function(t) {
  14.652 - 0.41022*t + 0.007991*t^2 - 0.000077774*t^3
}
ka= function(u, d) {
  (0.728*sqrt(u) - 0.317*u + 0.0372*u^2) / d / 86400
}
monod= function(s,h) {
  s / (s + h)
}

# forcings are functions of special variable 'time'
c_z_in= function(seconds) {
  0.1 * seconds/(7*86400 + seconds)
}
c_do_in= function(seconds) {
  10.  # taken as constant
}
```

We also need to set the values of parameters and state variables (initial values). We first define both parameters and initial values as lists. These lists are then coerced into named vectors using the dedicated methods `arrangePars` and `arrangeVars`. Use of these methods guarantees that the vector elements appear in proper order. Proper order is essential when the vector elements are accessed *by position* rather than *by name*.

```
pars= list(kd=5.78e-7, h_do=0.5, s_do_z=2.76, wind=1, depth=2,
 temp=20, q_in=1, q_ex=1)
vars= list(c_z=1, c_do=9.022, v=1.e6)
p= model$arrangePars(pars)
v= model$arrangeVars(vars)
```

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 values of all state variables, parameters, as well as a time value (in the case of non-autonomous models).

```
m= model$stoichiometry(c(v, p, time=0))
print(signif(m, 3))
```

```
           c_do c_z v
flow      0.000   0 1
flushing  0.978  -1 0
decay    -2.760  -1 0
aeration  1.000   0 0
```

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 for 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).

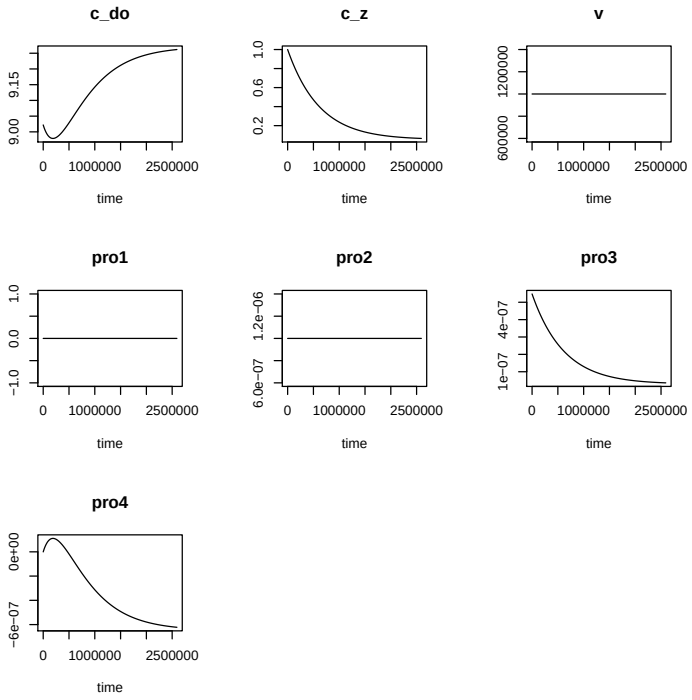
```
code= model$generate(name="derivs",lang="r")
derivs= eval(parse(text=code))
```

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. At this point, the proper order of the elements in the vectors `v` and `p` is essential since the generated function code uses *by-index* access! Therefore, `v` and `p` should always be set by calls to the `arrangeVars` and `arrangePars` method, respectively.

In addition to the dynamics of the state variables, we also get the dynamics of the process rates.

```
library(deSolve)
t= seq(0, 30*86400, 3600)
out= ode(y=v, times=t, func=derivs, parms=p, NLVL=1)
layout(matrix(1:9, ncol=3, byrow=TRUE))
plot(out, mfrow=NULL)
layout(1)
```



4 Advanced topics

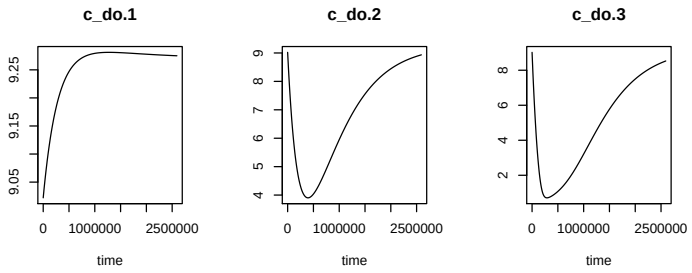
4.1 Spatially distributed systems (multi-box models)

A zero-dimensional case (single reactor) has been considered so far. We will now extend the model for multiple spatial levels, i. e. for a collection of (isolated) reactors.

First, we need to provide vectors (instead of scalars) for one variable and one parameter, at least. Variables and parameters with scalar values will be replicated to vectors of the required length. In the example, we initialize every modeled reactor with a different concentration of organic matter.

Second, we need to pass the actual number of reactors to the derivatives function in the `NLVL` argument.

```
nbox= 3
pars= list(kd=rep(5.78e-7, nbox), h_do=0.5, s_do_z=2.76, wind=1,
  depth=2, temp=20, q_in=1, q_ex=1)
vars= list(c_z=seq(from=0, to=50, length.out=nbox), c_do=9.022,
  v=1.e6)
p= model$arrangePars(pars)
v= model$arrangeVars(vars)
out= ode(y=v, times=t, func=derivs, parms=p, NLVL=nbox)
layout(matrix(1:nbox, nrow=1))
plot(out, which=paste("c_do",1:nbox,sep="."), mfrow=NULL)
```



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`.

The next statement generates `Fortran` code and exports it to a file for later compilation.

```
code= model$generate(name="derivs",lang="f95")
file_fdrvs= "derivatives.f95"
write(x=code, file=file_fdrvs)
# Optionally display generated code
#cat(code)
```

Inspection of the generated `Fortran` code reveals that the interface of the central subroutine `derivs` is

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

However, according to the `deSolve` vignette `http://cran.r-project.org/web/packages/deSolve/vignettes/compiledCode.pdf`, page 6, the following 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. Consequently, we need to provide the following `Fortran` code:

1. a wrapper around the generated derivatives function to make its interface compatible with `deSolve`,
2. the additional subroutine `initmod` for parameter initialization,
3. any functions being referenced in the particular model's mathematical expressions (consult Sect. 5 for guidelines).

A suitable Fortan code for (1) and (2) can be generated by the function call `solverInterface(NLVL)` where the integer argument `NLVL` specifies the desired number of spatial levels (see Sect. 4.1; default is `NLVL=1`). The generated wrapper and initialization code must be written to disk for later use by the compiler.

```
# Generate wrapper and initialization routine
fwrap= solverInterface(nbox, "derivs")
# Optionally display the generated code
#cat(fwrap)
# Export generated code to file for later compilation
file_fwrap= "fwrap.f95"
write(fwrap, file_fwrap)
```

The model-specific functions (number 3 in above list) could be defined as:

```
module functions
   implicit none
   contains

   double precision function O2sat(t)
     double precision, intent(in):: t
     O2sat= 14.652d0 - 0.41022d0*t + 7.991d-3*(t**2d0) - &
       7.7774d-5*(t**3d0)
   end function

   double precision function ka(u, d)
     double precision, intent(in):: u, d
     ka= (0.728d0*sqrt(u) - 0.317d0*u + 0.0372d0*(u**2.d0)) / &
       d / 86400.d0
   end function

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

   double precision function c_z_in(seconds)
     double precision, intent(in):: seconds
     c_z_in= 0.1d0 * seconds/(7d0*86400d0 + seconds)
   end function

   double precision function c_do_in(seconds)
     double precision, intent(in):: seconds
     c_do_in= 10.d0
   end function

 end module
```

Note that all 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.

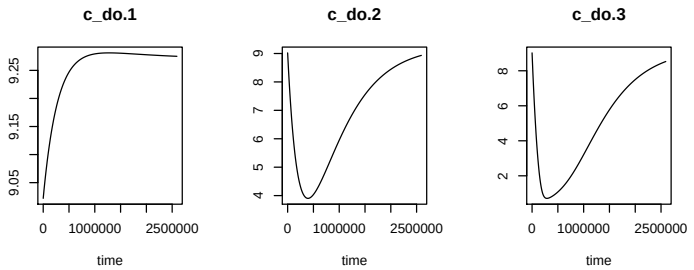
As a next step, one needs to compile the generated code and the hand-coded functions into a shared library. For that purpose, we make use of `R`'s command line utility `R CMD SHLIB`. The 3 files passed to the command are the following:

`file_fdrvs`	File containing the derivatives function (automatically created with the `generate` class method, see above)
`file_fwrap`	Source file containing the wrapper and initialization routine (automatically created with the `solverIterface` method, see above)
`file_ffuns`	Hand-coded source file with the model-specific function (contents shown above)

```
dllname= "mySharedLib"
dllfile= paste0(dllname,.Platform$dynlib.ext)
command= paste0("R CMD SHLIB ",file_ffuns," ",file_fdrvs," ",
  file_fwrap," --preclean --clean -o ",dllfile)
if (system(command) != 0)
  stop(paste0("Error running '",command,"'"))
```

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). Note that setting a wrong value for `nout` easily makes `R` crash.

```
nbox= 3
pars= list(kd=rep(5.78e-7, nbox), h_do=0.5, s_do_z=2.76, wind=1,
  depth=2, temp=20, q_in=1, q_ex=1)
vars= list(c_z=seq(from=0, to=50, length.out=nbox), c_do=9.022,
  v=1.e6)
p= model$arrangePars(pars)
v= model$arrangeVars(vars)
dyn.load(dllfile)
out= ode(y=v, times=t, func="derivs_wrapped", parms=p, NLVL=nbox,
  dllname=dllname, initfunc="initmod", nout=model$lenPros()*nbox)
layout(matrix(1:nbox, nrow=1))
dyn.unload(dllfile)
plot(out, which=paste("c_do",1:nbox,sep="."), mfrow=NULL)
```



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 rescrict 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   17    16
2     2   17    10
3     3   17    50
4     4   25    70
```

```
5     5   18     6
6     6   24    81
7     7   21    68
8     8   22    32
9     9   18    52
10   10   20    89
```

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)
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$getVars()[,c("tex","unit","description")]
# Define formatting functions
bold= function(x){paste0("\\textbf{",x,"}")}
mathmode= function(x) {paste0("$",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 holding tabular information on the model's state variables.

```
\begin{tabular}{lll}\hline
  \textbf{symbol} & \textbf{unit} & \textbf{description} \\ \hline
  $c_{DO}$ & mg/l & dissolved O2 \\
  $c_Z$ & mg/l & degradable org. matter Z \\
  $v$ & m3 & water volume in reactor \\ \hline
\end{tabular}
```

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 code section

```
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$getVars()[,c("tex", "unit", "description")]
names(ids)= c("Symbol", "Unit", "Description")
kable(to_markdown(ids, which_cols=c("Symbol")),
  caption = "State variables")
```

generates the following `markdown` code:

```
|Symbol   |Unit |Description              |
|:--------|:----|:------------------------|
|$c_{DO}$ |mg/l |dissolved O2             |
|$c_Z$    |mg/l |degradable org. matter Z |
|$v$      |m3   |water volume in reactor  |
```

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

```
pars= list(kd=5.78e-7, h_do=0.5, s_do_z=2.76, wind=1, depth=2,
 temp=20, q_in=1, q_ex=1)
vars= list(c_z=1, c_do=9.022, v=1.e6)
p= model$arrangePars(pars)
v= model$arrangeVars(vars)
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$stoichiometry(c(v, p, time=0))
tbl= cbind(data.frame(process=rownames(m), stringsAsFactors=FALSE),
  as.data.frame(m))
tex= exportDF(x=tbl, tex=TRUE,
  colnames= setNames(c("",model$getVars()$tex[match(colnames(m),
    model$getVars()$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:

	c_{DO}	c_Z	v
flow			▲
flushing	▲	▼	
decay	▼	▼	
aeration	▲		

Option 2: 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(c(v, p, 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$getVars()$html[match(colnames(m),
    model$getVars()$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 3: Markdown A `markdown` compatible version looks like this (contributed by Andrew Dolman):

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

and the generated `markdown` output is:

```
|Process  |$c_{DO}$             |$c_Z$                |$v$              |
|:--------|:--------------------|:--------------------|:----------------|
|flow     |                     |                     |$\blacktriangle$ |
|flushing |$\blacktriangle$     |$\blacktriangledown$ |                 |
|decay    |$\blacktriangledown$ |$\blacktriangledown$ |                 |
|aeration |$\blacktriangle$     |                     |                 |
```

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.