Package 'ReacTran'

August 16, 2017

Version 1.4.3.1

Title Reactive Transport Modelling in 1d, 2d and 3d

Author Karline Soetaert <karline.soetaert@nioz.nl>,
Filip Meysman <filip.meysman@nioz.nl>

Maintainer Karline Soetaert <karline.soetaert@nioz.nl>

Depends R (>= 2.10), rootSolve, deSolve, shape

Imports stats, graphics

Description Routines for developing models that describe reaction and advective-diffusive transport in one, two or three dimensions.
Includes transport routines in porous media, in estuaries, and in bodies with variable shape.

License GPL (>= 3)

LazyData yes

Repository CRAN

Repository/R-Forge/Project reactran

Repository/R-Forge/Revision 100

Repository/R-Forge/DateTimeStamp 2017-08-15 06:57:39

Date/Publication 2017-08-15 22:13:03 UTC

NeedsCompilation yes

R topics documented:

`ReacTran-package` *Reactive transport modelling in 1D, 2D and 3D*

Description

R-package ReacTran contains routines that enable the development of reactive transport models in aquatic systems (rivers, lakes, oceans), porous media (floc aggregates, sediments,...) and even idealized organisms (spherical cells, cylindrical worms,...).

The geometry of the model domain is either one-dimensional, two-dimensional or three-dimensional.

The package contains:

- Functions to setup a finite-difference grid (1D or 2D)
- Functions to attach parameters and properties to this grid (1D or 2D)
- Functions to calculate the advective-diffusive transport term over the grid (1D, 2D, 3D)
- Utility functions

Details

Package:	ReacTran
Type:	Package
Version:	1.4.3
Date:	2017-08-14
License:	GNU Public License 2 or above

Author(s)

Karline Soetaert (Maintainer)

Filip Meysman

See Also

Functions ode.1D, ode.2D, ode.3D from package `deSolve` to integrate the reactive-transport model

Functions steady.1D, steady.2D, steady.3D from package `rootSolve` to find the steady-state solution of the reactive-transport model

tran.1D, tran.2D, tran.3D for a discretisation of the general transport equations

tran.volume.1D for discretisation of the 1-D transport equations using finite volumes

tran.cylindrical, tran.spherical for a discretisation of 3-D transport equations in cylindrical and spherical coordinates

tran.polar, for a discretisation of 2-D transport equations in polar coordinates

setup.grid.1D, setup.grid.2D for the creation of grids in 1-D and in 2-D

setup.prop.1D, setup.prop.2D for defining properties on these grids.

Examples

```
## Not run:

## show examples (see respective help pages for details)
## 1-dimensional transport
example(tran.1D)
example(tran.volume.1D)

## 2-dimensional transport
example(tran.2D)
example(tran.polar)

## 3-dimensional transport
example(tran.3D)
example(tran.cylindrical)
example(tran.spherical)

## open the directory with documents
browseURL(paste(system.file(package="ReacTran"), "/doc", sep=""))

## open the directory with fortran codes of the transport functions
browseURL(paste(system.file(package="ReacTran"), "/doc/fortran", sep=""))

## show package vignette with how to use ReacTran and how to solve PDEs
## + source code of the vignettes
vignette("ReacTran")
vignette("PDE")
edit(vignette("ReacTran"))

## a directory with fortran implementations of the transport
browseURL(paste(system.file(package="ReacTran"), "/doc/fortran", sep=""))

## End(Not run)
```

---

advection.1D *One-Dimensional Advection Equation*

Description

Estimates the advection term in a one-dimensional model of a liquid (volume fraction constant and equal to one) or in a porous medium (volume fraction variable and lower than one).

The interfaces between grid cells can have a variable cross-sectional area, e.g. when modelling spherical or cylindrical geometries (see example).

TVD (total variation diminishing) slope limiters ensure monotonic and positive schemes in the presence of strong gradients.

`advection.1-D`: uses finite differences.

This implies the use of velocity (length per time) and fluxes (mass per unit of area per unit of time).

`advection.volume.1D` Estimates the volumetric advection term in a one-dimensional model of an aquatic system (river, estuary). This routine is particularly suited for modelling channels (like rivers, estuaries) where the cross-sectional area changes, and hence the velocity changes.

Volumetric transport implies the use of flows (mass per unit of time).

When solved dynamically, the euler method should be used, unless the first-order upstream method is used.

Usage

```
advection.1D(C, C.up = NULL, C.down = NULL,
  flux.up = NULL, flux.down = NULL, v, VF = 1, A = 1, dx,
  dt.default = 1, adv.method = c("muscl", "super", "quick", "p3", "up"),
  full.check = FALSE)

advection.volume.1D(C, C.up = C[1], C.down = C[length(C)],
  F.up = NULL, F.down = NULL, flow, V,
  dt.default = 1, adv.method = c("muscl", "super", "quick", "p3", "up"),
  full.check = FALSE)
```

Arguments

`C`	concentration, expressed per unit of phase volume, defined at the centre of each grid cell. A vector of length N [M/L3].
`C.up`	concentration at upstream boundary. One value [M/L3]. If `NULL`, and `flux.up` is also `NULL`, then a zero-gradient boundary is assumed, i.e. `C.up = C[1]`.
`C.down`	concentration at downstream boundary. One value [M/L3]. If `NULL`, and `flux.down` is also `NULL`, then a zero-gradient boundary is assumed, i.e. `C.down = C[length(C)]`.
`flux.up`	flux across the upstream boundary, positive = INTO model domain. One value, expressed per unit of total surface [M/L2/T]. If `NULL`, the boundary is prescribed as a concentration boundary.
`flux.down`	flux across the downstream boundary, positive = OUT of model domain. One value, expressed per unit of total surface [M/L2/T]. If `NULL`, the boundary is prescribed as a concentration boundary.
`F.up`	total input across the upstream boundary, positive = INTO model domain; used with `advection.volume.1D`. One value, expressed in [M/T]. If `NULL`, the boundary is prescribed as a concentration boundary.

F.down	total input across the downstream boundary, positive = OUT of model domain; used with advection.volume.1D. One value, expressed in [M/T]. If NULL, the boundary is prescribed as a concentration boundary.
v	advective velocity, defined on the grid cell interfaces. Can be positive (downstream flow) or negative (upstream flow). One value, a vector of length N+1 [L/T], or a 1D property list; the list contains at least the element int (see setup.prop.1D) [L/T]. Used with advection.1D.
flow	water flow rate, defined on grid cell interfaces. One value, a vector of length N+1, or a list as defined by setup.prop.1D [L^3/T]. Used with advection.volume.1D.
VF	Volume fraction defined at the grid cell interfaces. One value, a vector of length N+1, or a 1D property list; the list contains at least the elements int and mid (see setup.prop.1D) [-].
A	Interface area defined at the grid cell interfaces. One value, a vector of length N+1, or a 1D grid property list; the list contains at least the elements int and mid (see setup.prop.1D) [L^2].
dx	distance between adjacent cell interfaces (thickness of grid cells). One value, a vector of length N, or a 1D grid list containing at least the elements dx and dx.aux (see setup.grid.1D) [L].
dt.default	timestep to be used, if it cannot be estimated (e.g. when calculating steady-state conditions.
V	volume of cells. One value, or a vector of length N [L^3].
adv.method	the advection method, slope limiter used to reduce the numerical dispersion. One of "quick","muscl","super","p3","up" - see details.
full.check	logical flag enabling a full check of the consistency of the arguments (default = FALSE; TRUE slows down execution by 50 percent).

Details

This implementation is based on the GOTM code

The **boundary conditions** are either

- zero-gradient.
- fixed concentration.
- fixed flux.

The above order also shows the priority. The default condition is the zero gradient. The fixed concentration condition overrules the zero gradient. The fixed flux overrules the other specifications.

Ensure that the boundary conditions are well defined: for instance, it does not make sense to specify an influx in a boundary cell with the advection velocity pointing outward.

Transport properties:

The *advective velocity* (v), the *volume fraction* (VF), and the *interface surface* (A), can either be specified as one value, a vector, or a 1D property list as generated by setup.prop.1D.

When a vector, this vector must be of length N+1, defined at all grid cell interfaces, including the upper and lower boundary.

The **finite difference grid** (dx) is specified either as one value, a vector or a 1D grid list, as generated by setup.grid.1D.

Several slope limiters are implemented to obtain monotonic and positive schemes also in the presence of strong gradients, i.e. to reduce the effect of numerical dispersion. The methods are (Pietrzak, 1989, Hundsdorfer and Verwer, 2003):

- "quick": third-order scheme (TVD) with ULTIMATE QUICKEST limiter (quadratic upstream interpolation for convective kinematics with estimated stream terms) (Leonard, 1988)
- "muscl": third-order scheme (TVD) with MUSCL limiter (monotonic upstream centered schemes for conservation laws) (van Leer, 1979).
- "super": third-order scheme (TVD) with Superbee limiter (method=Superbee) (Roe, 1985)
- "p3": third-order upstream-biased polynomial scheme (method=P3)
- "up": first-order upstream (method=UPSTREAM) - this is the same method as implemented in tran.1D or tran.volume.1D

where "TVD" means a total variation diminishing scheme

Some schemes may produce artificial steepening. Scheme "p3" is not necessarily monotone (may produce negative concentrations!).

If during a certain time step the maximum Courant number is larger than one, a split iteration will be carried out which guarantees that the split step Courant numbers are just smaller than 1. The maximal number of such iterations is set to 100.

These limiters are supposed to work with explicit methods (euler). However, they will also work with implicit methods, although less effectively. Integrate ode.1D only if the model is stiff (see first example).

Value

dC	the rate of change of the concentration C due to advective transport, defined in the centre of each grid cell. The rate of change is expressed per unit of (phase) volume [M/L^3/T].
adv.flux	advective flux across at the interface of each grid cell. A vector of length N+1 [M/L2/T] - only for advection.1D.
flux.up	flux across the upstream boundary, positive = INTO model domain. One value [M/L2/T] - only for advection.1D.
flux.down	flux across the downstream boundary, positive = OUT of model domain. One value [M/L2/T] - only for advection.1D.
adv.F	advective mass flow across at the interface of each grid cell. A vector of length N+1 [M/T] - only for advection.volume.1D.
F.up	mass flow across the upstream boundary, positive = INTO model domain. One value [M/T] - only for advection.volume.1D.
F.down	flux across the downstream boundary, positive = OUT of model domain. One value [M/T] - only for advection.volume.1D.
it	number of split time iterations that were necessary.

Note

The advective equation is not checked for mass conservation. Sometimes, this is not an issue, for instance when v represents a sinking velocity of particles or a swimming velocity of organisms.

In others cases however, mass conservation needs to be accounted for.

To ensure mass conservation, the advective velocity must obey certain continuity constraints: in essence the product of the volume fraction (VF), interface surface area (A) and advective velocity (v) should be constant. In sediments, one can use `setup.compaction.1D` to ensure that the advective velocities for the pore water and solid phase meet these constraints.

In terms of the units of concentrations and fluxes we follow the convention in the geosciences. The concentration `C`, `C.up`, `C.down` as well at the rate of change of the concentration `dC` are always expressed per unit of phase volume (i.e. per unit volume of solid or liquid).

Total concentrations (e.g. per unit volume of bulk sediment) can be obtained by multiplication with the appropriate volume fraction. In contrast, fluxes are always expressed per unit of total interface area (so here the volume fraction is accounted for).

Author(s)

Karline Soetaert <karline.soetaert@nioz.nl>

References

Pietrzak J (1998) The use of TVD limiters for forward-in-time upstream-biased advection schemes in ocean modeling. Monthly Weather Review 126: 812 .. 830

Hundsdorfer W and Verwer JG (2003) Numerical Solution of Time-Dependent Advection-Diffusion-Reaction Equations. Springer Series in Computational Mathematics, Springer-Verlag, Berlin, 471 pages

Burchard H, Bolding K, Villarreal MR (1999) GOTM, a general ocean turbulence model. Theory, applications and test cases. Tech Rep EUR 18745 EN. European Commission

Leonard BP (1988) Simple high accuracy resolution program for convective modeling of discontinuities. Int. J. Numer. Meth.Fluids 8: 1291–1318.

Roe PL (1985) Some contributions to the modeling of discontinuous flows. Lect. Notes Appl. Math. 22: 163-193.

van Leer B. (1979) Towards the ultimate conservative difference scheme V. A second order sequel to Godunov's method. J. Comput. Phys. 32: 101-136

See Also

`tran.1D`, for a discretisation of the general transport equations

Examples

```
## ============================================================================
## EXAMPLE 1: Testing the various methods - moving a square pulse
## use of advection.1D
## The tests as in Pietrzak
```

```
## =============================================================================

#--------------------#
# Model formulation  #
#--------------------#
model <- function (t, y, parms,...) {

  adv <- advection.1D(y, v = v, dx = dx,
     C.up = y[n], C.down = y[1], ...)  # out on one side -> in at other
  return(list(adv$dC))

}

#--------------------#
# Parameters         #
#--------------------#

n     <- 100
dx    <- 100/n
y     <- c(rep(1, 5), rep(2, 20), rep(1, n-25))
v     <- 2
times <- 0:300   # 3 times out and in

#--------------------#
# model solution     #
#--------------------#

## a plotting function
plotfun <- function (Out, ...) {
  plot(Out[1, -1], type = "l", col = "red", ylab = "y", xlab = "x", ...)
  lines(Out[nrow(Out), 2:(1+n)])
}

# courant number = 2
pm   <- par(mfrow = c(2, 2))

## third order TVD, quickest limiter
out <- ode.1D(y = y, times = times, func = model, parms = 0, hini = 1,
              method = "euler", nspec = 1, adv.method = "quick")

plotfun(out, main = "quickest, euler")

## third-order ustream-biased polynomial
out2 <- ode.1D(y = y, times = times, func = model, parms = 0, hini = 1,
              method = "euler", nspec = 1, adv.method = "p3")

plotfun(out2, main = "p3, euler")

## third order TVD, superbee limiter
out3 <- ode.1D(y = y, times = times, func = model, parms = 0, hini = 1,
              method = "euler", nspec = 1, adv.method = "super")

plotfun(out3, main = "superbee, euler")
```

```
## third order TVD, muscl limiter
out4 <- ode.1D(y = y, times = times, func = model, parms = 0, hini = 1,
               method = "euler", nspec = 1, adv.method = "muscl")

plotfun(out4, main = "muscl, euler")

## ================================================================================
## upstream, different time-steps , i.e. different courant number
## ================================================================================
out <- ode.1D(y = y, times = times, func = model, parms = 0,
               method = "euler", nspec = 1, adv.method = "up")

plotfun(out, main = "upstream, courant number = 2")

out2 <- ode.1D(y = y, times = times, func = model, parms = 0, hini = 0.5,
                method = "euler", nspec = 1, adv.method = "up")
plotfun(out2, main = "upstream, courant number = 1")


## Now muscl scheme, velocity against x-axis
y     <- rev(c(rep(0, 5), rep(1, 20), rep(0, n-25)))
v     <- -2.0
out6 <- ode.1D(y = y, times = times, func = model, parms = 0, hini = 1,
                method = "euler", nspec = 1, adv.method = "muscl")

plotfun(out6, main = "muscl, reversed velocity, , courant number = 1")

image(out6, mfrow = NULL)
par(mfrow = pm)


## ================================================================================
## EXAMPLE 2: moving a square pulse in a widening river
## use of advection.volume.1D
## ================================================================================

#--------------------#
# Model formulation  #
#--------------------#

river.model <- function (t=0, C, pars = NULL, ...)  {

 tran <- advection.volume.1D(C = C, C.up = 0,
                     flow = flow, V = Volume,...)
 return(list(dCdt = tran$dC, F.down = tran$F.down, F.up = tran$F.up))
}

#--------------------#
# Parameters         #
#--------------------#

# Initialising morphology river:
```

```
nbox          <- 100                # number of grid cells
lengthRiver   <- 100000             # [m]
BoxLength     <- lengthRiver / nbox # [m]

Distance      <- seq(BoxLength/2, by = BoxLength, len = nbox)   # [m]

# Cross sectional area: sigmoid function of distance [m2]
CrossArea <- 4000 + 72000 * Distance^5 /(Distance^5+50000^5)

# Volume of boxes                          (m3)
Volume    <- CrossArea*BoxLength

# Transport coefficients
flow      <- 1000*24*3600   # m3/d, main river upstream inflow

#--------------------#
# Model solution     #
#--------------------#

pm   <- par(mfrow=c(2,2))

# a square pulse
yini <-  c(rep(10, 10), rep(0, nbox-10))

## third order TVD, muscl limiter
Conc <- ode.1D(y = yini, fun = river.model,  method = "euler", hini = 1,
              parms = NULL, nspec = 1, times = 0:40, adv.method = "muscl")

image(Conc, main = "muscl", mfrow = NULL)
plot(Conc[30, 2:(1+nbox)], type = "l", lwd = 2, xlab = "x", ylab = "C",
     main = "muscl after 30 days")

## simple upstream differencing
Conc2<- ode.1D(y = yini, fun = river.model, method = "euler", hini = 1,
              parms = NULL, nspec = 1, times = 0:40, adv.method = "up")

image(Conc2, main = "upstream", mfrow = NULL)
plot(Conc2[30, 2:(1+nbox)], type = "l", lwd = 2, xlab = "x", ylab = "C",
     main = "upstream after 30 days")

par(mfrow = pm)


# Note: the more sophisticated the scheme, the more mass lost/created
# increase tolerances to improve this.

CC <- Conc[ , 2:(1+nbox)]
MASS <- t(CC)*Volume
colSums(MASS)

## ================================================================================
## EXAMPLE 3: A steady-state solution
```

```
## use of advection.volume.1D
## ============================================================================

Sink <- function (t, y, parms, ...) {
  C1 <- y[1:N]
  C2 <- y[(N+1):(2*N)]
  C3 <- y[(2*N+1):(3*N)]
  # Rate of change= Flux gradient and first-order consumption

  # upstream can be implemented in two ways:

  dC1  <- advection.1D(C1, v = sink, dx = dx,
          C.up = 100, adv.method = "up", ...)$dC - decay*C1

# same, using tran.1D
#  dC1  <- tran.1D(C1, v = sink, dx = dx,
#        C.up = 100, D = 0)$dC -
#           decay*C1

  dC2  <- advection.1D(C2, v = sink, dx = dx,
        C.up = 100, adv.method = "p3", ...)$dC -
           decay*C2

  dC3  <- advection.1D(C3, v = sink, dx = dx,
        C.up = 100, adv.method = "muscl", ...)$dC -
           decay*C3

  list(c(dC1, dC2, dC3))
}

N     <- 10
L     <- 1000
dx    <- L/N                          # thickness of boxes
sink  <- 10
decay <- 0.1
out <- steady.1D(runif(3*N), func = Sink, names = c("C1", "C2", "C3"),
        parms = NULL, nspec = 3, bandwidth = 2)
matplot(out$y, 1:N, type = "l", ylim = c(10, 0), lwd = 2,
  main = "Steady-state")
legend("bottomright", col = 1:3, lty = 1:3,
  c("upstream", "p3", "muscl"))
```

`fiadeiro` *Advective Finite Difference Weights*

Description

Weighing coefficients used in the finite difference scheme for advection calculated according to Fiadeiro and Veronis (1977).

This particular AFDW (advective finite difference weights) scheme switches from backward differencing (in advection dominated conditions; large Peclet numbers) to central differencing (under diffusion dominated conditions; small Peclet numbers).

This way it forms a compromise between stability, accuracy and reduced numerical dispersion.

Usage

```
fiadeiro(v, D, dx.aux = NULL, grid = list(dx.aux = dx.aux))
```

Arguments

v	advective velocity; either one value or a vector of length N+1, with N the number of grid cells [L/T]
D	diffusion coefficient; either one value or a vector of length N+1 [L2/T]
dx.aux	auxiliary vector containing the distances between the locations where the concentration is defined (i.e. the grid cell centers and the two outer interfaces); either one value or a vector of length N+1 [L]
grid	discretization grid as calculated by setup.grid.1D

Details

The Fiadeiro and Veronis (1977) scheme adapts the differencing method to the local situation (checks for advection or diffusion dominance).

Finite difference schemes are based on following rationale:

- When using forward differences (AFDW = 0), the scheme is first order accurate, creates a low level of (artificial) numerical dispersion, but is highly unstable (state variables may become negative).
- When using backward differences (AFDW = 1), the scheme is first order accurate, is universally stable (state variables always remain positive), but the scheme creates high levels of numerical dispersion.
- When using central differences (AFDW = 0.5), the scheme is second order accurate, is not universally stable, and has a moderate level of numerical dispersion, but state variables may become negative.

Because of the instability issue, forward schemes should be avoided. Because of the higher accuracy, the central scheme is preferred over the backward scheme.

The central scheme is stable when sufficient physical dispersion is present, it may become unstable when advection is the only transport process.

The Fiadeiro and Veronis (1977) scheme takes this into account: it uses central differencing when possible (when physical dispersion is high enough), and switches to backward differing when needed (when advection dominates). The switching is determined by the Peclet number

```
Pe = abs(v)*dx.aux/D
```

- the higher the diffusion D (Pe > 1), the closer the AFDW coefficients are to 0.5 (central differencing)
- the higher the advection v (Pe < 1), the closer the AFDW coefficients are to 1 (backward differencing)

Value

the Advective Finite Difference Weighing (AFDW) coefficients as used in the transport routines `tran.1D` and `tran.volume.1D`; either one value or a vector of length N+1 [-]

Note

- If the state variables (concentrations) decline in the direction of the 1D axis, then the central difference scheme will be stable. If this is known a prioiri, then central differencing is preferred over the fiadeiro scheme.
- Each scheme will always create some numerical diffusion. This principally depends on the resolution of the grid (i.e. larger `dx.aux` values create higher numerical diffusion). In order to reduce numerical dispersion, one should increase the grid resolution (i.e. decrease `dx.aux`).

Author(s)

Filip Meysman <filip.meysman@nioz.nl>,

Karline Soetaert <karline.soetaert@nioz.nl>

References

- Fiadeiro ME and Veronis G (1977) Weighted-mean schemes for finite-difference approximation to advection-diffusion equation. Tellus 29, 512-522.
- Boudreau (1997) Diagnetic models and their implementation. Chapter 8: Numerical Methods. Springer.

Examples

```
#==========================================
# Model formulation (differential equations)
#==========================================

# This is a test model to evaluate the different finite difference schemes
# and evaluate their effect on munerical diffusion. The model describes the
# decay of organic carbon (OC) as it settles through the ocean water column.

model <- function (time, OC, pars, AFDW = 1) {
  dOC <- tran.1D(OC, flux.up = F_OC, D = D.eddy,
                 v = v.sink, AFDW = AFDW, dx = dx)$dC - k*OC
  return(list(dOC))
}
#==========================================
# Parameter set
#==========================================

L <- 1000         # water depth model domain [m]
x.att <- 200      # attenuation depth of the sinking velocity [m]
v.sink.0 <- 10    # sinking velocity at the surface [m d-1]
D.eddy <- 10      # eddy diffusion coefficient [m2 d-1]
F_OC <- 10        # particle flux [mol m-2 d-1]
k <- 0.1          # decay coefficient [d-1]
```

```
## ================================================================================
## Model solution for a coarse grid (10 grid cells)
## ================================================================================

# Setting up the grid
N <- 10                               # number of grid layers
dx <- L/N                             # thickness of boxes [m]
dx.aux <- rep(dx, N+1)                # auxilliary grid vector
x.int <- seq(from = 0, to = L, by = dx)    # water depth at box interfaces [m]
x.mid <- seq(from = dx/2, to = L, by = dx) # water depth at box centres [m]

# Exponentially declining sink velocity
v.sink <- v.sink.0 * exp(-x.int/x.att) # sink velocity [m d-1]
Pe <- v.sink * dx/D.eddy               # Peclet number

# Calculate the weighing coefficients
AFDW <- fiadeiro(v = v.sink, D = D.eddy, dx.aux = dx.aux)

par(mfrow = c(2, 1), cex.main = 1.2, cex.lab = 1.2)

# Plot the Peclet number over the grid

plot(Pe, x.int, log = "x", pch = 19, ylim = c(L,0), xlim = c(0.1, 1000),
     xlab = "", ylab = "depth [m]",
     main = "Peclet number", axes = FALSE)
abline(h = 0)
axis(pos = NA, side = 2)
axis(pos = 0, side = 3)

# Plot the AFDW coefficients over the grid

plot(AFDW, x.int, pch = 19, ylim = c(L, 0), xlim = c(0.5, 1),
     xlab = "", ylab = "depth [m]", main = "AFDW coefficient", axes = FALSE)
abline(h = 0)
axis(pos = NA, side = 2)
axis(pos = 0, side = 3)

# Three steady-state solutions for a coarse grid based on:
# (1) backward differences (BD)
# (2) central differences (CD)
# (3) Fiadeiro & Veronis scheme (FV)

BD <- steady.1D(y = runif(N), func = model, AFDW = 1.0, nspec = 1)
CD <- steady.1D(y = runif(N), func = model, AFDW = 0.5, nspec = 1)
FV <- steady.1D(y = runif(N), func = model, AFDW = AFDW, nspec = 1)

# Plotting output - use rootSolve's plot method
plot(BD, CD, FV, grid = x.mid, xyswap = TRUE, mfrow = c(1,2),
     xlab = "", ylab = "depth [m]", main = "conc (Low resolution grid)")

legend("bottomright", col = 1:3, lty = 1:3,
       legend = c("backward diff", "centred diff", "Fiadeiro&Veronis"))
```

```
## ============================================================================
## Model solution for a fine grid (1000 grid cells)
## ============================================================================

# Setting up the grid
N <- 1000                           # number of grid layers
dx <- L/N                           # thickness of boxes[m]
dx.aux <- rep(dx, N+1)             # auxilliary grid vector
x.int <- seq(from = 0, to = L, by = dx)      # water depth at box interfaces [m]
x.mid <- seq(from = dx/2, to = L, by = dx)   # water depth at box centres [m]

# Exponetially declining sink velocity
v.sink <- v.sink.0 * exp(-x.int/x.att) # sink velocity [m d-1]
Pe <- v.sink * dx/D.eddy               # Peclet number

# Calculate the weighing coefficients
AFDW <- fiadeiro(v = v.sink, D = D.eddy, dx.aux = dx.aux)

# Three steady-state solutions for a coarse grid based on:
# (1) backward differences (BD)
# (2) centered differences (CD)
# (3) Fiadeiro & Veronis scheme (FV)

BD <- steady.1D(y = runif(N), func = model, AFDW = 1.0, nspec = 1)
CD <- steady.1D(y = runif(N), func = model, AFDW = 0.5, nspec = 1)
FV <- steady.1D(y = runif(N), func = model, AFDW = AFDW, nspec = 1)

# Plotting output
plot(BD, CD, FV, grid = x.mid, xyswap = TRUE, mfrow = NULL,
     xlab = "", ylab = "depth [m]", main = "conc (High resolution grid)")

legend("bottomright", col = 1:3, lty = 1:3,
       legend = c("backward diff", "centred diff", "Fiadeiro&Veronis"))

# Results and conclusions:
# - For the fine grid, all three solutions are identical
# - For the coarse grid, the BD and FV solutions show numerical dispersion
#   while the CD provides more accurate results
```

`g.sphere` *Surface Area and Volume of Geometrical Objects*

Description

- `g.sphere` the surface and volume of a sphere
- `g.spheroid` the surface and volume of a spheroid
- `g.cylinder` the surface and volume of a cylinder; note that the surface area calculation ignores the top and bottom.

Usage

```
g.sphere(x)
g.spheroid (x, b = 1)
g.cylinder (x, L = 1)
```

Arguments

x	the radius
b	the ratio of long/short radius of the spheroid; if b<1: the spheroid is oblate.
L	the length of the cylinder

Value

A list containing:

surf	the surface area
vol	the volume

Author(s)

Filip Meysman <filip.meysman@nioz.nl>, Karline Soetaert <karline.soetaert@nioz.nl>

Examples

```
mf <- par(mfrow = c(3, 2))
x  <- seq(from = 0, to = 1, length = 10)
plot(x, g.sphere(x)$surf, main = "sphere surface")
plot(x, g.sphere(x)$vol, main = "sphere volume")
plot(x, g.spheroid(x, b = 0.5)$surf, main = "spheroid surface")
plot(x, g.spheroid(x, b = 0.5)$vol, main = "spheroid volume")
plot(x, g.cylinder(x, L = 1)$surf, main = "cylinder surface")
plot(x, g.cylinder(x, L = 1)$vol, main = "cylinder volume")
par("mfrow" = mf)
```

---

p.exp *Common Properties with Distance, to be used with setup.prop.1D*

Description

Functions that define an y-property as a function of the one-dimensional x-coordinate. These routines can be used to specify properties and parameters as a function of distance, e.g. depth in the water column or the sediment.

They make a transition from an upper (or upstream) zone, with value `y.0` to a lower zone with a value `y.inf`.

Particularly useful in combination with setup.prop.1D

- `p.exp`: exponentially decreasing transition

$$y = y_{\inf} + (y_0 - y_{\inf}) \exp(-\max(0, x - x_0)/x_a)$$

- `p.lin`: linearly decreasing transition

$$y = y_0; y = y_0 - (y_0 - y_{inf}) * (x - x_L)/x_{att}); y = y_{inf}$$

  for $0 \leq x \leq x_L$, $x_L \leq x \leq x_L + x_{att}$ and $(x \geq x_L + x.att)$ respectively.

- `p.sig`: sigmoidal decreasing transition

$$y = y_{inf} + (y_0 - y_{inf}) \frac{\exp(-(x - x_L)/(0.25 x_{att}))}{(1 + \exp(-(x - x_L))/(0.25 x_{att}))})$$

Usage

```
p.exp(x, y.0 = 1, y.inf = 0.5, x.L = 0, x.att = 1)
p.lin(x, y.0 = 1, y.inf = 0.5, x.L = 0, x.att = 1)
p.sig(x, y.0 = 1, y.inf = 0.5, x.L = 0, x.att = 1)
```

Arguments

`x`	the x-values for which the property has to be calculated.
`y.0`	the y-value at the origin
`y.inf`	the y-value at infinity
`x.L`	the x-coordinate where the transition zone starts; for `x <= x.0`, the value will be equal to `y.0`. For `x >> x.L + x.att` the value will tend to `y.inf`
`x.att`	attenuation coefficient in exponential decrease, or the size of the transition zone in the linear and sigmoid decrease

Details

For `p.lin`, the width of the transition zone equals `x.att` and the depth where the transition zone starts is `x.L`.

For `p.sig`, `x.L` is located the middle of the smooth transition zone of approaximate width `x.att`.

For `p.exp`, there is no clearly demarcated transition zone; there is an abrupt change at `x.L` after which the property exponentially changes from `y.0` towards `y.L` with attenuation coefficient `x.att`; the larger `x.att` the less steep the change.

Value

the property value, estimated for each x-value.

Author(s)

Filip Meysman <filip.meysman@nioz.nl>, Karline Soetaert <karline.soetaert@nioz.nl>

Examples

```
x <- seq(0, 5, len = 100)
plot(x, p.exp(x, x.L = 2),
     xlab = "x.coordinate", ylab = "y value", ylim = c(0, 1))
lines(x, p.lin(x, x.L = 2), col = "blue")
lines(x, p.sig(x, x.L = 2), col = "red")
```

---

setup.compaction.1D *Calculates Advective Velocities of the Pore Water and Solid Phase in a Water Saturated Sediment assuming Steady State Compaction*

Description

This function calculates the advective velocities of the pore water and the solid phase in a sediment based on the assumption of steady state compaction.

The velocities of the pore water (`u`) and the solid phase (`v`) are calculated in the middle (`mid`) of the grid cells and the interfaces (`int`).

One needs to specify the porosity at the interface (`por.0`), the porosity at infinite depth (`por.inf`), the porosity profile (`por.grid`) encoded as a 1D grid property (see `setup.prop.1D`, as well as the advective velocity of the solid phase at one particular depth (either at the sediment water interface (`v.0`) or at infinite depth (`v.inf`)).

Usage

```
setup.compaction.1D(v.0 = NULL, v.inf = NULL, por.0, por.inf,
                    por.grid)
```

Arguments

`v.0`	advective velocity of the solid phase at the sediment-water interface (also referred to as the sedimentation velocity); if `NULL` then `v.inf` must not be `NULL` [L/T]
`v.inf`	advective velocity of the solid phase at infinite depth (also referred to as the burial velocity); if `NULL` then `v.0` must not be `NULL` [L/T]
`por.0`	porosity at the sediment-water interface
`por.inf`	porosity at infinite depth
`por.grid`	porosity profile specified as a 1D grid property (see `setup.prop.1D` for details on the structure of this list)

Value

A list containing:

u	list with pore water advective velocities at the middle of the grid cells (`mid`) and at the grid cell interfaces (`int`).
v	list with solid phase advective velocities at the middle of the grid cells (`mid`) and at the grid cell interfaces (`int`).

Author(s)

Filip Meysman <filip.meysman@nioz.nl>, Karline Soetaert <karline.soetaert@nioz.nl>

References

Meysman, F. J. R., Boudreau, B. P., Middelburg, J. J. (2005) Modeling Reactive Transport in Sediments Subject to Bioturbation and Compaction. Geochimica Et Cosmochimica Acta 69, 3601-3617

Examples

```
# setup of the 1D grid

L <-10
grid <- setup.grid.1D(x.up = 0, L = L, N = 20)

# attaching an exponential porosity profile to the 1D grid
# this uses the "p.exp" profile function

por.grid <- setup.prop.1D(func = p.exp, grid = grid,
                          y.0 = 0.9, y.inf = 0.5, x.att = 3)

# calculate the advective velocities

dummy <- setup.compaction.1D(v.0 = 1, por.0 = 0.9, por.inf = 0.5,
                             por.grid = por.grid)
u.grid <- dummy$u
v.grid <- dummy$v

# plotting the results

par(mfrow = c(2, 1), cex.main = 1.2, cex.lab = 1.2)

plot(por.grid$int, grid$x.int, pch = 19, ylim = c(L,0), xlim = c(0,1),
     xlab = "", ylab = "depth [cm]", main = expression("porosity"),
     axes = FALSE)
abline(h = 0)
axis(pos = 0, side = 2)
axis(pos = 0, side = 3)

plot(u.grid$int, grid$x.int, type = "l", lwd = 2, col = "blue",
     ylim = c(L, 0), xlim = c(0, max(u.grid$int,v.grid$int)),
```

```
     xlab = "", ylab = "depth [cm]",
     main = "advective velocity [cm yr-1]", axes = FALSE)
abline(h = 0)
axis(pos = 0, side = 2)
axis(pos = 0, side = 3)

lines(v.grid$int, grid$x.int, lwd = 2, col = "red")

legend(x = "bottomright", legend = c("pore water","solid phase"),
       col = c("blue", "red"), lwd = 2)
```

`setup.grid.1D` *Creates a One-Dimensional Finite Difference Grid*

Description

Subdivides the one-dimensional model domain into one or more zones that are each sub-divided into grid cells. The resulting grid structure can be used in the other `ReacTran` functions.

The grid structure is characterized by the position of the middle of the grid cells (`x.mid`) and the position of the interfaces between grid cells (`x.int`).

Distances are calculated between the interfaces (`dx`), i.e. the thickness of the grid cells. An auxiliary set of distances (`dx.aux`) is calculated between the points where the concentrations are specified (at the center of each grid cell and the two external interfaces).

A more complex grid consisting of multiple zones can be constructed when specifying the endpoints of ech zone (`x.down`), the interval length (`L`), and the number of layers in each zone (`N`) as vectors. In each zone, one can control the grid resolution near the upstream and downstream boundary.

The grid resolution at the upstream interface changes according to the power law relation `dx[i+1] = min(max.dx.1,p.dx.1*`, where `p.dx.1` determines the rate of increase and `max.dx.1` puts an upper limit on the grid cell size.

A similar formula controls the resolution at the downstream interface. This allows refinement of the grid near the interfaces.

If only `x.up, N` and `dx.1` are specified, then the grid size is taken constant = `dx.1` (and `L=N*dx.1`)

Usage

```
setup.grid.1D(x.up = 0, x.down = NULL, L = NULL, N = NULL, dx.1 = NULL,
              p.dx.1 = rep(1, length(L)), max.dx.1 = L, dx.N = NULL,
              p.dx.N = rep(1, length(L)), max.dx.N = L)

## S3 method for class 'grid.1D'
plot(x, ...)
```

Arguments

x.up	position of the upstream interface; one value [L]
x.down	position of the endpoint of each zone; one value when the model domain covers only one zone (x.down = position of downstream interface), or a vector of length M when the model domain is divided into M zones (x.down[M] = position of downstream interface) [L]
L	thickness of zones; one value (model domain = one zone) or a vector of length M (model domain = M zones) [L]
N	number of grid cells within a zone; one value or a vector of length M [-]
dx.1	size of the first grid cell in a zone; one value or a vector of length M [L]
p.dx.1	power factor controlling the increase in grid cell size near the upstream boundary; one value or a vector of length M. The default value is 1 (constant grid cell size) [-]
max.dx.1	maximum grid cell size in the upstream half of the zone; one value or a vector of length M [L]
dx.N	size of the last grid cell in a zone; one value or a vector of length M [L]
p.dx.N	power factor controlling the increase in grid cell size near the downstream boundary; one value or a vector of length M. The default value is 1 (constant grid cell size) [-]
max.dx.N	maximum grid cell size in the downstream half of the zone; one value or a vector of length M [L]
x	the object of class grid.1D that needs plotting
...	additional arguments passed to the function plot

Value

a list of type grid.1D containing:

N	the total number of grid cells [-]
x.up	position of the upstream interface; one value [L]
x.down	position of the downstream interface; one value [L]
x.mid	position of the middle of the grid cells; vector of length N [L]
x.int	position of the interfaces of the grid cells; vector of length N+1 [L]
dx	distance between adjacent cell interfaces (thickness of grid cells); vector of length N [L]
dx.aux	auxiliary vector containing the distance between adjacent cell centers; at the upper and lower boundary calculated as (x[1]-x.up) and (x.down-x[N]) respectively; vector of length N+1 [L]

Author(s)

Filip Meysman <filip.meysman@nioz.nl>, Karline Soetaert <karline.soetaert@nioz.nl>

See Also

tran.1D, for a discretisation of the general transport equation in 1-D

setup.grid.2D for the creation of grids in 2-D

setup.prop.1D, for defining properties on the 1-D grid.

Examples

```
# one zone, constant resolution
(GR <- setup.grid.1D(x.up = 0, L = 10, N = 10))
(GR <- setup.grid.1D(x.up = 0, L = 10, dx.1 = 1))
(GR <- setup.grid.1D(x.up = 0, L = 10, dx.N = 1))
plot(GR)

# one zone, constant resolution, origin not zero
(GR <- setup.grid.1D(x.up = 5, x.down = 10, N = 10))
plot(GR)

# one zone, variable resolution
(GR <- setup.grid.1D(x.up = 0, L = 10, dx.1 = 1, p.dx.1 = 1.2))
(GR <- setup.grid.1D(x.up = 0, L = 10, dx.N = 1, p.dx.N = 1.2))
plot(GR)

# one zone, variable resolution, imposed number of layers
(GR <- setup.grid.1D(x.up = 0, L = 10, N = 6, dx.1 = 1, p.dx.1 = 1.2))
(GR <- setup.grid.1D(x.up = 0, L = 10, N = 6, dx.N = 1, p.dx.N = 1.2))
plot(GR)

# one zone, higher resolution near upstream and downstream interfaces
(GR <- setup.grid.1D(x.up = 0, x.down = 10, dx.1 = 0.1,
                     p.dx.1 = 1.2, dx.N = 0.1, p.dx.N = 1.2))
plot(GR)

# one zone, higher resolution near upstream and downstream interfaces
# imposed number of layers
(GR <- setup.grid.1D(x.up = 0, x.down = 10, N = 20,
                     dx.1 = 0.1, p.dx.1 = 1.2,
                     dx.N = 0.1, p.dx.N = 1.2))
plot(GR)

# two zones, higher resolution near the upstream
# and downstream interface
(GR<-setup.grid.1D(x.up = 0, L = c(5, 5),
         dx.1 = c(0.2, 0.2), p.dx.1 = c(1.1, 1.1),
         dx.N = c(0.2, 0.2), p.dx.N = c(1.1, 1.1)))
plot(GR)

# two zones, higher resolution near the upstream
# and downstream interface
# the number of grid cells in each zone is imposed via N
(GR <- setup.grid.1D(x.up = 0, L = c(5, 5), N = c(20, 10),
         dx.1 = c(0.2, 0.2), p.dx.1 = c(1.1, 1.1),
```

```
          dx.N = c(0.2, 0.2), p.dx.N = c(1.1, 1.1)))
plot(GR)
```

`setup.grid.2D` *Creates a Finite Difference Grid over a Two-Dimensional Rectangular Domain*

Description

Creates a finite difference grid over a rectangular two-dimensional model domain starting from two separate one-dimensional grids (as created by `setup.grid.1D`).

Usage

```
setup.grid.2D(x.grid = NULL, y.grid = NULL)
```

Arguments

`x.grid`	list containing the one-dimensional grid in the vertical direction - see `setup.grid.1D` for the structure of the list
`y.grid`	list containing the one-dimensional grid in the horizontal direction - see `setup.grid.1D` for the structure of the list

Value

a list of type `grid.2D` containing:

`x.up`	position of the upstream interface in x-direction (i.e. if x is vertical, the upper boundary); one value
`x.down`	position of the downstream interface in x-direction (i.e. if x is vertical, the lower boundary); one value
`x.mid`	position of the middle of the grid cells in x-direction; vector of length `x.N`
`x.int`	position of the interfaces of the grid cells in x-direction; vector of length `x.N+1`
`dx`	distance between adjacent cell interfaces in x-direction (thickness of grid cells); vector of length `x.N`
`dx.aux`	auxiliary vector containing the distance between adjacent cell centers; at the upstream and downstream boundary calculated as (`x[1]-x.up`) and (`x.down-x[x.N]`) respectively; vector of length `x.N+1`
`x.N`	total number of grid cells in the x direction; one value
`y.left`	position of the upstream interface in y-direction (i.e. if y us the horizontal, the left boundary); one value
`y.right`	position of the downstream interface in y-direction (i.e. if y us the horizontal, the right boundary); one value
`y.mid`	position of the middle of the grid cells in y-direction; vector of length `y.N`
`y.int`	position of the interfaces of the grid cells in y-direction; vector of length `y.N+1`

dy	distance between adjacent cell interfaces in y-direction (thickness of grid cells); vector of length y.N
dy.aux	auxiliary vector containing the distance between adjacent cell centers; at the upstream and downstream boundary calculated as (y[1]-y.up) and (y.down-y[y.N]) respectively; vector of length y.N+1
y.N	total number of grid cells in the y direction; one value

Author(s)

Filip Meysman <filip.meysman@nioz.nl>, Karline Soetaert <karline.soetaert@nioz.nl>

See Also

tran.2D, for a discretisation of the general transport equation in 2-D

setup.grid.1D, for the creation of grids in 1-D

setup.prop.2D for defining properties on the 2-D grid.

Examples

```
# test of the setup.grid.2D functionality
x.grid  <- setup.grid.1D(x.up = 0, L = 10, N = 5)
y.grid  <- setup.grid.1D(x.up = 0, L = 20, N = 10)
(grid2D <- setup.grid.2D(x.grid, y.grid))
```

---

setup.prop.1D *Attaches a Property to a One-Dimensional Grid*

Description

This routine calculates the value of a given property at the middle of the grid cells (mid) and at the interfaces of the grid cells (int).

Two possibilities are available: either specifying a mathematical function (func) that describes the spatial dependency of the property, or obtaining the property from interpolation of a data series (via the input of the data matrix xy).

For example, in a sediment model, setup.prop.1D can be used to specify the porosity, the mixing intensity or some other parameter over the one-dimensional grid. Similarly, in a vertical water column model, setup.prop.1D can be used to specify the sinking velocity of particles or other model parameters changing with water depth.

Usage

```
setup.prop.1D(func = NULL, value = NULL, xy = NULL,
              interpolate = "spline", grid, ...)

## S3 method for class 'prop.1D'
plot(x, grid, xyswap = FALSE, ...)
```

Arguments

func	function that describes the spatial dependency. For example, one can use the functions provided in p.exp
value	constant value given to the property (no spatial dependency)
xy	a two-column data matrix where the first column (x) provides the position, and the second column (y) provides the values that needs interpolation over the grid
interpolate	specifies how the interpolation should be done, one of "spline" or "linear"; only used when xy is present
grid	list specifying the 1D grid characteristics, see setup.grid.1D for details on the structure of this list
x	the object of class prop.1D that needs plotting
xyswap	if TRUE, then x- and y-values are swapped and the y-axis is oriented from top to bottom. Useful for drawing vertical depth profiles
...	additional arguments that are passed on to func or to the S3 method

Details

There are two options to carry out the data interpolation:

- "spline" gives a smooth profile, but sometimes generates strange profiles - always check the result!
- "linear" gives a segmented profile

Value

A list of type prop.1D containing:

mid	property value in the middle of the grid cells; vector of length N (where N is the number of grid cells)
int	property value at the interface of the grid cells; vector of length N+1

Author(s)

Karline Soetaert <karline.soetaert@nioz.nl>, Filip Meysman <filip.meysman@nioz.nl>

See Also

tran.1D, for a discretisation of the general transport equation in 1-D

setup.grid.1D, the creation of grids in 1-D

setup.prop.2D for defining properties on 2-D grids.

Examples

```
# Construction of the 1D grid

grid <- setup.grid.1D(x.up = 0, L = 10, N = 10)

# Porosity profile via function specification

P.prof <- setup.prop.1D(func = p.exp, grid = grid, y.0 = 0.9,
                        y.inf = 0.5, x.att = 3)

# Porosity profile via data series interpolation

P.data <- matrix(ncol = 2, data = c(0,3,6,10,0.9,0.65,0.55,0.5))
P.spline <- setup.prop.1D(xy = P.data, grid = grid)
P.linear <- setup.prop.1D(xy = P.data, grid = grid, interpolate = "linear")

# Plot different profiles

plot(P.prof, grid = grid, type = "l",
     main = "setup.prop, function evaluation")
points(P.data, cex = 1.5, pch = 16)
lines(grid$x.int, P.spline$int, lty = "dashed")
lines(grid$x.int, P.linear$int, lty = "dotdash")
```

`setup.prop.2D` *Attaches a Property to a Two-Dimensional Grid*

Description

Calculates the value of a given property at the middle of grid cells (`mid`) and at the interfaces of the grid cells (`int`).

Two possibilities are available: either specifying a mathematical function (`func`) that describes the spatial dependency of the property, or asssuming a constant value (`value`). To allow for anisotropy, the spatial dependency can be different in the x and y direction.

For example, in a sediment model, the routine can be used to specify the porosity, the mixing intensity or other parameters over the grid of the reactangular sediment domain.

Usage

```
setup.prop.2D(func = NULL, value = NULL, grid, y.func = func,
              y.value = value, ...)

## S3 method for class 'prop.2D'
contour(x, grid, xyswap = FALSE, filled = FALSE, ...)
```

Arguments

func	function that describes the spatial dependency in the x-direction; defined as func <- function (x,y, ...); it should return as many elements as in x or y
value	constant value given to the property in the x-direction
grid	list specifying the 2D grid characteristics, see setup.grid.2D for details on the structure of this list
y.func	function that describes the spatial dependency in the y-direction; defined as y.func <- function (x, y, ...); it should return as many elements as in x or y. By default the same as in the x-direction.
y.value	constant value given to the property in the y-direction. By default the same as in the x-direction.
x	the object of class prop.2D that needs plotting
filled	if TRUE, uses filled.contour, else contour
xyswap	if TRUE, then x- and y-values are swapped and the y-axis is oriented from top to bottom. Useful for drawing vertical depth profiles
...	additional arguments that are passed on to func or to the method

Details

- When the property is isotropic, the x.mid and y.mid values are identical. This is for example the case for sediment porosity.
- When the property is anisotropic, the x.mid and y.mid values can differ. This can be for example the case for the velocity, where in general, the value will differ between the x and y direction.

Value

A list of type prop.2D containing:

x.mid	property value in the x-direction defined at the middle of the grid cells; Nx * Ny matrix (where Nx and Ny = number of cells in x, y direction)
y.mid	property value in the y-direction at the middle of the grid cells; Nx * Ny matrix
x.int	property value in the x-direction defined at the x-interfaces of the grid cells; (Nx+1)*Ny matrix
y.int	property value in the y-direction at the y-interfaces of the grid cells; Nx*(Ny+1) matrix

Note

For some properties, it does not make sense to use y.func different to func. For instance, for volume fractions, AFDW.

For other properties, it may be usefull to have y.func or y.value different from func or value, for instance for velocities, surface areas, ...

Author(s)

Filip Meysman <filip.meysman@nioz.nl>, Karline Soetaert <karline.soetaert@nioz.nl>

Examples

```
# Inverse quadratic function
inv.quad <- function(x, y, a = NULL, b = NULL)
   return(1/((x-a)^2+(y-b)^2))


# Construction of the 2D grid
x.grid <- setup.grid.1D (x.up = 0, L = 10, N = 10)
y.grid <- setup.grid.1D (x.up = 0, L = 10, N = 10)
grid2D <- setup.grid.2D (x.grid, y.grid)

# Attaching the inverse quadratic function to the 2D grid
(twoD <- setup.prop.2D (func = inv.quad, grid = grid2D, a = 5, b = 5))

# show
contour(log(twoD$x.int))
```

---

tran.1D *General One-Dimensional Advective-Diffusive Transport*

Description

Estimates the transport term (i.e. the rate of change of a concentration due to diffusion and advection) in a one-dimensional model of a liquid (volume fraction constant and equal to one) or in a porous medium (volume fraction variable and lower than one).

The interfaces between grid cells can have a variable cross-sectional area, e.g. when modelling spherical or cylindrical geometries (see example).

Usage

```
tran.1D(C, C.up = C[1], C.down = C[length(C)],
        flux.up = NULL, flux.down = NULL,
        a.bl.up = NULL, a.bl.down = NULL,
        D = 0, v = 0, AFDW = 1, VF = 1, A = 1, dx,
        full.check = FALSE, full.output = FALSE)
```

Arguments

C	concentration, expressed per unit of phase volume, defined at the centre of each grid cell. A vector of length N [M/L3]
C.up	concentration at upstream boundary. One value [M/L3]
C.down	concentration at downstream boundary. One value [M/L3]

flux.up	flux across the upstream boundary, positive = INTO model domain. One value, expressed per unit of total surface [M/L2/T]. If NULL, the boundary is prescribed as a concentration or a convective transfer boundary.
flux.down	flux across the downstream boundary, positive = OUT of model domain. One value, expressed per unit of total surface [M/L2/T]. If NULL, the boundary is prescribed as a concentration or a convective transfer boundary.
a.bl.up	convective transfer coefficient across the upstream boundary layer. Flux = a.bl.up*(C.up-C0). One value [L/T]
a.bl.down	convective transfer coefficient across the downstream boundary layer (L). Flux = a.bl.down*(CL-C.dow One value [L/T]
D	diffusion coefficient, defined on grid cell interfaces. One value, a vector of length N+1 [L2/T], or a 1D property list; the list contains at least the element int (see setup.prop.1D) [L2/T]
v	advective velocity, defined on the grid cell interfaces. Can be positive (downstream flow) or negative (upstream flow). One value, a vector of length N+1 [L/T], or a 1D property list; the list contains at least the element int (see setup.prop.1D) [L/T]
AFDW	weight used in the finite difference scheme for advection, defined on grid cell interfaces; backward = 1, centred = 0.5, forward = 0; default is backward. One value, a vector of length N+1, or a 1D property list; the list contains at least the element int (see setup.prop.1D) [-]
VF	Volume fraction defined at the grid cell interfaces. One value, a vector of length N+1, or a 1D property list; the list contains at least the elements int and mid (see setup.prop.1D) [-]
A	Interface area defined at the grid cell interfaces. One value, a vector of length N+1, or a 1D grid property list; the list contains at least the elements int and mid (see setup.prop.1D) [L2]
dx	distance between adjacent cell interfaces (thickness of grid cells). One value, a vector of length N, or a 1D grid list containing at least the elements dx and dx.aux (see setup.grid.1D) [L]
full.check	logical flag enabling a full check of the consistency of the arguments (default = FALSE; TRUE slows down execution by 50 percent)
full.output	logical flag enabling a full return of the output (default = FALSE; TRUE slows down execution by 20 percent)

Details

The **boundary conditions** are either

- (1) zero-gradient.
- (2) fixed concentration.
- (3) convective boundary layer.
- (4) fixed flux.

The above order also shows the priority. The default condition is the zero gradient. The fixed concentration condition overrules the zero gradient. The convective boundary layer condition overrules the fixed concentration and zero gradient. The fixed flux overrules all other specifications.

Ensure that the boundary conditions are well defined: for instance, it does not make sense to specify an influx in a boundary cell with the advection velocity pointing outward.

Transport properties:

The *diffusion coefficient* (D), the *advective velocity* (v), the *volume fraction* (VF), the *interface surface* (A), and the *advective finite difference weight* (AFDW) can either be specified as one value, a vector or a 1D property list as generated by setup.prop.1D.

When a vector, this vector must be of length N+1, defined at all grid cell interfaces, including the upper and lower boundary.

The **finite difference grid** (dx) is specified either as one value, a vector or a 1D grid list, as generated by setup.grid.1D.

Value

dC	the rate of change of the concentration C due to transport, defined in the centre of each grid cell. The rate of change is expressed per unit of phase volume [M/L3/T]
C.up	concentration at the upstream interface. One value [M/L3] only when (full.output = TRUE)
C.down	concentration at the downstream interface. One value [M/L3] only when (full.output = TRUE)
dif.flux	diffusive flux across at the interface of each grid cell. A vector of length N+1 [M/L2/T] only when (full.output = TRUE)
adv.flux	advective flux across at the interface of each grid cell. A vector of length N+1 [M/L2/T] only when (full.output = TRUE)
flux	total flux across at the interface of each grid cell. A vector of length N+1 [M/L2/T]. only when (full.output = TRUE)
flux.up	flux across the upstream boundary, positive = INTO model domain. One value [M/L2/T]
flux.down	flux across the downstream boundary, positive = OUT of model domain. One value [M/L2/T]

Note

The advective equation is not checked for mass conservation. Sometimes, this is not an issue, for instance when v represents a sinking velocity of particles or a swimming velocity of organisms. In others cases however, mass conservation needs to be accounted for. To ensure mass conservation, the advective velocity must obey certain continuity constraints: in essence the product of the volume fraction (VF), interface surface area (A) and advective velocity (v) should be constant. In sediments, one can use setup.compaction.1D to ensure that the advective velocities for the pore water and solid phase meet these constraints.

In terms of the units of concentrations and fluxes we follow the convention in the geosciences. The concentration C, C.up, C.down as well at the rate of change of the concentration dC are always expressed per unit of phase volume (i.e. per unit volume of solid or liquid).

Total concentrations (e.g. per unit volume of bulk sediment) can be obtained by multiplication with the appropriate volume fraction. In contrast, fluxes are always expressed per unit of total interface area (so here the volume fraction is accounted for).

Author(s)

Filip Meysman <filip.meysman@nioz.nl>, Karline Soetaert <karline.soetaert@nioz.nl>

References

Soetaert and Herman (2009). A practical guide to ecological modelling - using R as a simulation platform. Springer

See Also

tran.volume.1D for a discretisation the transport equation using finite volumes.

tran.2D, tran.3D

advection.1D, for more sophisticated advection schemes

Examples

```
## =============================================================================
## EXAMPLE 1: O2 and OC consumption in sediments
## =============================================================================

# this example uses only the volume fractions
# in the reactive transport term

#====================#
# Model formulation  #
#====================#

# Monod consumption of oxygen (O2)

O2.model <- function (t = 0, O2, pars = NULL) {

  tran <- tran.1D(C = O2, C.up = C.ow.O2, D = D.grid,
                  v = v.grid, VF = por.grid, dx = grid)$dC
  reac <- - R.O2*(O2/(Ks+O2))
  return(list(dCdt = tran + reac))
}

# First order consumption of organic carbon (OC)

OC.model <- function (t = 0, OC, pars = NULL) {

  tran <- tran.1D(C = OC, flux.up = F.OC, D = Db.grid,
                  v = v.grid, VF = svf.grid, dx = grid)$dC
  reac <- - k*OC
  return(list(dCdt = tran + reac))
}
```

```
#======================#
# Parameter definition #
#======================#

# Parameter values

F.OC    <- 25    # input flux organic carbon [micromol cm-2 yr-1]
C.ow.O2 <- 0.25  # concentration O2 in overlying water [micromol cm-3]
por     <- 0.8   # porosity
D       <- 400   # diffusion coefficient O2 [cm2 yr-1]
Db      <- 10    # mixing coefficient sediment [cm2 yr-1]
v       <- 1     # advective velocity [cm yr-1]
k       <- 1     # decay constant organic carbon [yr-1]
R.O2    <- 10    # O2 consumption rate [micromol cm-3 yr-1]
Ks      <- 0.005 # O2 consumption saturation constant

# Grid definition

L <- 10   # depth of sediment domain [cm]
N <- 100  # number of grid layers
grid <- setup.grid.1D(x.up = 0, L = L, N = N)

# Volume fractions

por.grid <- setup.prop.1D(value = por, grid = grid)
svf.grid <- setup.prop.1D(value = (1-por), grid = grid)
D.grid   <- setup.prop.1D(value = D, grid = grid)
Db.grid  <- setup.prop.1D(value = Db, grid = grid)
v.grid   <- setup.prop.1D(value = v, grid = grid)

#====================#
# Model solution     #
#====================#

# Initial conditions + simulation O2

yini <- rep(0, length.out = N)
O2   <- steady.1D(y = yini, func = O2.model, nspec = 1)

# Initial conditions + simulation OC

yini <- rep(0, length.out = N)
OC   <- steady.1D(y = yini, func = OC.model, nspec = 1)

# Plotting output, using S3 plot method of package rootSolve"

plot(O2, grid = grid$x.mid, xyswap = TRUE, main = "O2 concentration",
     ylab = "depth [cm]", xlab = "", mfrow = c(1,2), type = "p", pch = 16)

plot(OC, grid = grid$x.mid, xyswap = TRUE, main = "C concentration",
     ylab = "depth [cm]", xlab = "", mfrow = NULL)
```

```
## =============================================================================
## EXAMPLE 2: O2 in a cylindrical and spherical organism
## =============================================================================

# This example uses only the surface areas
# in the reactive transport term

#====================#
# Model formulation  #
#====================#

# the numerical model - rate of change = transport-consumption
Cylinder.Model <- function(time, O2, pars)
  return (list(
    tran.1D(C = O2, C.down = BW, D = Da, A = A.cyl, dx = dx)$dC - Q
    ))

Sphere.Model <- function(time, O2, pars)
  return (list(
    tran.1D(C = O2, C.down = BW, D = Da, A = A.sphere, dx = dx)$dC - Q
    ))

#======================#
# Parameter definition #
#======================#

# parameter values

BW     <- 2      # mmol/m3,  oxygen conc in surrounding water
Da     <- 0.5    # cm2/d     effective diffusion coeff in organism
R      <- 0.0025 # cm        radius of organism
Q      <- 250000 # nM/cm3/d  oxygen consumption rate/ volume / day
L      <- 0.05   # cm        length of organism (if a cylinder)

# the numerical model

N  <- 40                              # layers in the body
dx <- R/N                             # thickness of each layer
x.mid <- seq(dx/2, by = dx, length.out = N) # distance of center to mid-layer
x.int <- seq(0, by = dx, length.out = N+1)  # distance to layer interface

# Cylindrical surfaces
A.cyl   <- 2*pi*x.int*L  # surface at mid-layer depth

# Spherical surfaces
A.sphere <- 4*pi*x.int^2 # surface of sphere, at each mid-layer

#====================#
# Model solution     #
#====================#

# the analytical solution of cylindrical and spherical model
cylinder <- function(Da, Q, BW, R, r)  BW + Q/(4*Da)*(r^2-R^2)
```

```
sphere   <- function(Da, Q, BW, R, r)  BW + Q/(6*Da)*(r^2-R^2)

# solve the model numerically for a cylinder
O2.cyl <- steady.1D (y = runif(N), name = "O2",
      func = Cylinder.Model, nspec = 1, atol = 1e-10)

# solve the model numerically for a sphere
O2.sphere <- steady.1D (y = runif(N), name = "O2",
      func = Sphere.Model, nspec = 1, atol = 1e-10)

#====================#
# Plotting output    #
#====================#
# Analytical solution - "observations"
Ana.cyl   <- cbind(x.mid, O2 = cylinder(Da, Q, BW, R, x.mid))
Ana.spher <- cbind(x.mid, O2 = sphere(Da, Q, BW, R, x.mid))

plot(O2.cyl, O2.sphere, grid = x.mid, lwd = 2, lty = 1, col = 1:2,
     xlab = "distance from centre, cm",
     ylab = "mmol/m3", main = "tran.1D",
     sub = "diffusion-reaction in a cylinder and sphere",
     obs = list(Ana.cyl, Ana.spher), obspar = list(pch = 16, col =1:2))

legend ("topleft", lty = c(1, NA), pch = c(NA, 18),
        c("numerical approximation", "analytical solution"))
legend ("bottomright", pch = 16, lty = 1, col = 1:2,
        c("cylinder", "sphere"))

## =============================================================================
## EXAMPLE 3: O2 consumption in a spherical aggregate
## =============================================================================

# this example uses both the surface areas and the volume fractions
# in the reactive transport term

#====================#
# Model formulation  #
#====================#

Aggregate.Model <- function(time, O2, pars) {

  tran <- tran.1D(C = O2, C.down = C.ow.O2,
                  D = D.grid, A = A.grid,
                  VF = por.grid, dx = grid )$dC

  reac <- - R.O2*(O2/(Ks+O2))*(O2>0)
  return(list(dCdt = tran + reac, consumption = -reac))

}

#======================#
# Parameter definition #
#======================#
```

```
# Parameters

C.ow.O2 <- 0.25     # concentration O2 water [micromol cm-3]
por     <- 0.8      # porosity
D       <- 400      # diffusion coefficient O2 [cm2 yr-1]
v       <- 0        # advective velocity [cm yr-1]
R.O2    <- 1000000  # O2 consumption rate [micromol cm-3 yr-1]
Ks      <- 0.005    # O2 saturation constant [micromol cm-3]

# Grid definition
R <- 0.025           # radius of the agggregate [cm]
N <- 100             # number of grid layers
grid <- setup.grid.1D(x.up = 0, L = R, N = N)

# Volume fractions

por.grid <- setup.prop.1D(value = por, grid = grid)
D.grid   <- setup.prop.1D(value = D, grid = grid)

# Surfaces

A.mid <- 4*pi*grid$x.mid^2  # surface of sphere at middle of grid cells
A.int <- 4*pi*grid$x.int^2  # surface of sphere at interface
A.grid <- list(int = A.int, mid = A.mid)

#====================#
# Model solution     #
#====================#

# Numerical solution: staedy state

O2.agg <- steady.1D (runif(N), func = Aggregate.Model, nspec = 1,
                     atol = 1e-10, names = "O2")

#====================#
# Plotting output    #
#====================#

par(mfrow = c(1,1))

plot(grid$x.mid, O2.agg$y, xlab = "distance from centre, cm",
     ylab = "mmol/m3",
     main = "Diffusion-reaction of O2 in a spherical aggregate")
legend ("bottomright", pch = c(1, 18), lty = 1, col = "black",
        c("O2 concentration"))

# Similar, using S3 plot method of package rootSolve"
plot(O2.agg, grid = grid$x.mid, which = c("O2", "consumption"),
     xlab = "distance from centre, cm", ylab = c("mmol/m3","mmol/m3/d"))
```

tran.2D *General Two-Dimensional Advective-Diffusive Transport*

Description

Estimates the transport term (i.e. the rate of change of a concentration due to diffusion and advection) in a two-dimensional model domain.

Usage

```
tran.2D (C, C.x.up = C[1,], C.x.down = C[nrow(C),],
         C.y.up = C[,1], C.y.down = C[ ,ncol(C)],
         flux.x.up = NULL, flux.x.down = NULL,
         flux.y.up = NULL, flux.y.down = NULL,
         a.bl.x.up = NULL, a.bl.x.down = NULL,
         a.bl.y.up = NULL, a.bl.y.down = NULL,
         D.grid = NULL, D.x = NULL, D.y = D.x,
         v.grid = NULL, v.x = 0, v.y = 0,
         AFDW.grid = NULL, AFDW.x = 1, AFDW.y = AFDW.x,
         VF.grid = NULL, VF.x = 1, VF.y = VF.x,
         A.grid = NULL, A.x = 1, A.y = 1,
         grid = NULL, dx = NULL, dy = NULL,
         full.check = FALSE, full.output = FALSE)
```

Arguments

C	concentration, expressed per unit volume, defined at the centre of each grid cell; Nx*Ny matrix [M/L3].
C.x.up	concentration at upstream boundary in x-direction; vector of length Ny [M/L3].
C.x.down	concentration at downstream boundary in x-direction; vector of length Ny [M/L3].
C.y.up	concentration at upstream boundary in y-direction; vector of length Nx [M/L3].
C.y.down	concentration at downstream boundary in y-direction; vector of length Nx [M/L3].
flux.x.up	flux across the upstream boundary in x-direction, positive = INTO model domain; vector of length Ny [M/L2/T].
flux.x.down	flux across the downstream boundary in x-direction, positive = OUT of model domain; vector of length Ny [M/L2/T].
flux.y.up	flux across the upstream boundary in y-direction, positive = INTO model domain; vector of length Nx [M/L2/T].
flux.y.down	flux across the downstream boundary in y-direction, positive = OUT of model domain; vector of length Nx [M/L2/T].
a.bl.x.up	transfer coefficient across the upstream boundary layer. in x-direction; Flux=a.bl.x.up*(C.x.up-C[1,]). One value [L/T].
a.bl.x.down	transfer coefficient across the downstream boundary layer in x-direction; Flux=a.bl.x.down*(C[Nx,]-C.x.down). One value [L/T].

a.bl.y.up	transfer coefficient across the upstream boundary layer. in y-direction; Flux=a.bl.y.up*(C.y.up-C[,1]). One value [L/T].
a.bl.y.down	transfer coefficient across the downstream boundary layer in y-direction; Flux=a.bl.y.down*(C[,Ny]-C.y.down). One value [L/T].
D.grid	diffusion coefficient defined on all grid cell interfaces. A prop.2D list created by setup.prop.2D [L2/T]. See last example for creating spatially-varying diffusion coefficients.
D.x	diffusion coefficient in x-direction, defined on grid cell interfaces. One value, a vector of length (Nx+1), a prop.1D list created by setup.prop.1D, or a (Nx+1)* Ny matrix [L2/T].
D.y	diffusion coefficient in y-direction, defined on grid cell interfaces. One value, a vector of length (Ny+1), a prop.1D list created by setup.prop.1D, or a Nx*(Ny+1) matrix [L2/T].
v.grid	advective velocity defined on all grid cell interfaces. Can be positive (downstream flow) or negative (upstream flow). A prop.2D list created by setup.prop.2D [L/T].
v.x	advective velocity in the x-direction, defined on grid cell interfaces. Can be positive (downstream flow) or negative (upstream flow). One value, a vector of length (Nx+1), a prop.1D list created by setup.prop.1D, or a (Nx+1)*Ny matrix [L/T].
v.y	advective velocity in the y-direction, defined on grid cell interfaces. Can be positive (downstream flow) or negative (upstream flow). One value, a vector of length (Ny+1), a prop.1D list created by setup.prop.1D, or a Nx*(Ny+1) matrix [L/T].
AFDW.grid	weight used in the finite difference scheme for advection in the x- and y- direction, defined on grid cell interfaces; backward = 1, centred = 0.5, forward = 0; default is backward. A prop.2D list created by setup.prop.2D [-].
AFDW.x	weight used in the finite difference scheme for advection in the x-direction, defined on grid cell interfaces; backward = 1, centred = 0.5, forward = 0; default is backward. One value, a vector of length (Nx+1), a prop.1D list created by setup.prop.1D, or a (Nx+1)*Ny matrix [-].
AFDW.y	weight used in the finite difference scheme for advection in the y-direction, defined on grid cell interfaces; backward = 1, centred = 0.5, forward = 0; default is backward. One value, a vector of length (Ny+1), a prop.1D list created by setup.prop.1D, or a Nx*(Ny+1) matrix [-].
VF.grid	Volume fraction. A prop.2D list created by setup.prop.2D [-].
VF.x	Volume fraction at the grid cell interfaces in the x-direction. One value, a vector of length (Nx+1), a prop.1D list created by setup.prop.1D, or a (Nx+1)*Ny matrix [-].
VF.y	Volume fraction at the grid cell interfaces in the y-direction. One value, a vector of length (Ny+1), a prop.1D list created by setup.prop.1D, or a Nx*(Ny+1) matrix [-].
A.grid	Interface area. A prop.2D list created by setup.prop.2D [L2].

A.x	Interface area defined at the grid cell interfaces in the x-direction. One value, a vector of length (Nx+1), a prop.1D list created by setup.prop.1D, or a (Nx+1)*Ny matrix [L2].
A.y	Interface area defined at the grid cell interfaces in the y-direction. One value, a vector of length (Ny+1), a prop.1D list created by setup.prop.1D, or a Nx*(Ny+1) matrix [L2].
dx	distance between adjacent cell interfaces in the x-direction (thickness of grid cells). One value or vector of length Nx [L].
dy	distance between adjacent cell interfaces in the y-direction (thickness of grid cells). One value or vector of length Ny [L].
grid	discretization grid, a list containing at least elements dx, dx.aux, dy, dy.aux (see setup.grid.2D) [L].
full.check	logical flag enabling a full check of the consistency of the arguments (default = FALSE; TRUE slows down execution by 50 percent).
full.output	logical flag enabling a full return of the output (default = FALSE; TRUE slows down execution by 20 percent).

Details

The **boundary conditions** are either

- (1) zero-gradient
- (2) fixed concentration
- (3) convective boundary layer
- (4) fixed flux

This is also the order of priority. The zero gradient is the default, the fixed flux overrules all other.

Value

a list containing:

dC	the rate of change of the concentration C due to transport, defined in the centre of each grid cell, a Nx*Ny matrix. [M/L3/T].
C.x.up	concentration at the upstream interface in x-direction. A vector of length Ny [M/L3]. Only when full.output = TRUE.
C.x.down	concentration at the downstream interface in x-direction. A vector of length Ny [M/L3]. Only when full.output = TRUE.
C.y.up	concentration at the the upstream interface in y-direction. A vector of length Nx [M/L3]. Only when full.output = TRUE.
C.y.down	concentration at the downstream interface in y-direction. A vector of length Nx [M/L3]. Only when full.output = TRUE.
x.flux	flux across the interfaces in x-direction of the grid cells. A (Nx+1)*Ny matrix [M/L2/T]. Only when full.output = TRUE.
y.flux	flux across the interfaces in y-direction of the grid cells. A Nx*(Ny+1) matrix [M/L2/T]. Only when full.output = TRUE.

`flux.x.up`	flux across the upstream boundary in x-direction, positive = INTO model domain. A vector of length Ny [M/L2/T].
`flux.x.down`	flux across the downstream boundary in x-direction, positive = OUT of model domain. A vector of length Ny [M/L2/T].
`flux.y.up`	flux across the upstream boundary in y-direction, positive = INTO model domain. A vector of length Nx [M/L2/T].
`flux.y.down`	flux across the downstream boundary in y-direction, positive = OUT of model domain. A vector of length Nx [M/L2/T].

Note

It is much more efficient to use the *grid* input rather than vectors or single numbers.

Thus: to optimise the code, use setup.grid.2D to create the `grid`, and use setup.prop.2D to create `D.grid`, `v.grid`, `AFDW.grid`, `VF.grid`, and `A.grid`, even if the values are 1 or remain constant.

There is no provision (yet) to deal with *cross-diffusion*. Set `D.x` and `D.y` different only if cross-diffusion effects are unimportant.

Author(s)

Filip Meysman <filip.meysman@nioz.nl>, Karline Soetaert <karline.soetaert@nioz.nl>

References

Soetaert and Herman, 2009. a practical guide to ecological modelling - using R as a simulation platform. Springer

See Also

`tran.polar` for a discretisation of 2-D transport equations in polar coordinates

`tran.1D`, `tran.3D`

Examples

```
## =============================================================================
## Testing the functions
## =============================================================================
# Parameters
F        <- 100             # input flux [micromol cm-2 yr-1]
por      <- 0.8             # constant porosity
D        <- 400             # mixing coefficient [cm2 yr-1]
v        <- 1               # advective velocity [cm yr-1]

# Grid definition
x.N <- 4   # number of cells in x-direction
y.N <- 6   # number of cells in y-direction
x.L <- 8   # domain size x-direction [cm]
y.L <- 24  # domain size y-direction [cm]
dx  <- x.L/x.N            # cell size x-direction [cm]
dy  <- y.L/y.N            # cell size y-direction [cm]
```

```
# Intial conditions
C <- matrix(nrow = x.N, ncol = y.N, data = 0, byrow = FALSE)

# Boundary conditions: fixed concentration
C.x.up   <- rep(1, times = y.N)
C.x.down <- rep(0, times = y.N)
C.y.up   <- rep(1, times = x.N)
C.y.down <- rep(0, times = x.N)

# Only diffusion
tran.2D(C = C, D.x = D, D.y = D, v.x = 0, v.y = 0,
  VF.x = por, VF.y = por, dx = dx, dy = dy,
  C.x.up = C.x.up, C.x.down = C.x.down,
  C.y.up = C.y.up, C.y.down = C.y.down, full.output = TRUE)

# Strong advection, backward (default), central and forward
#finite difference schemes
tran.2D(C = C, D.x = D, v.x = 100*v,
  VF.x = por, dx = dx, dy = dy,
  C.x.up = C.x.up, C.x.down = C.x.down,
  C.y.up = C.y.up, C.y.down = C.y.down)

tran.2D(AFDW.x = 0.5, C = C, D.x = D, v.x = 100*v,
  VF.x = por, dx = dx, dy = dy,
  C.x.up = C.x.up, C.x.down = C.x.down,
  C.y.up = C.y.up, C.y.down = C.y.down)

tran.2D(AFDW.x = 0, C = C, D.x = D, v.x = 100*v,
  VF.x = por, dx = dx, dy = dy,
  C.x.up = C.x.up, C.x.down = C.x.down,
  C.y.up = C.y.up, C.y.down = C.y.down)

# Boundary conditions: fixed fluxes

flux.x.up <- rep(200, times = y.N)
flux.x.down <- rep(-200, times = y.N)
flux.y.up <- rep(200, times = x.N)
flux.y.down <- rep(-200, times = x.N)
tran.2D(C = C, D.x = D, v.x = 0,
  VF.x = por, dx = dx, dy = dy,
  flux.x.up = flux.x.up, flux.x.down = flux.x.down,
  flux.y.up = flux.y.up, flux.y.down = flux.y.down)

# Boundary conditions: convective boundary layer on all sides

a.bl <- 800   # transfer coefficient
C.x.up <- rep(1, times = (y.N)) # fixed conc at boundary layer
C.y.up <- rep(1, times = (x.N)) # fixed conc at boundary layer
tran.2D(full.output = TRUE, C = C, D.x = D, v.x = 0,
  VF.x = por, dx = dx, dy = dy,
  C.x.up   = C.x.up, a.bl.x.up = a.bl,
  C.x.down = C.x.up, a.bl.x.down = a.bl,
```

```
  C.y.up   = C.y.up, a.bl.y.up = a.bl,
  C.y.down = C.y.up, a.bl.y.down = a.bl)

# Runtime test with and without argument checking

n.iterate <-500

test1 <- function() {
  for (i in 1:n.iterate )
    ST <- tran.2D(full.check = TRUE, C = C, D.x = D,
      v.x = 0, VF.x = por, dx = dx, dy = dy,
      C.x.up = C.x.up, a.bl.x.up = a.bl, C.x.down = C.x.down)
}
system.time(test1())

test2 <- function() {
  for (i in 1:n.iterate )
    ST <- tran.2D(full.output = TRUE, C = C, D.x = D,
      v.x = 0, VF.x = por, dx = dx, dy = dy,
      C.x.up = C.x.up, a.bl.x.up = a.bl, C.x.down = C.x.down)
}
system.time(test2())

test3 <- function() {
  for (i in 1:n.iterate )
    ST <- tran.2D(full.output = TRUE, full.check = TRUE, C = C,
      D.x = D, v.x = 0, VF.x = por, dx = dx, dy = dy,
      C.x.up = C.x.up, a.bl.x.up = a.bl, C.x.down = C.x.down)
}
system.time(test3())

## ============================================================================
## A 2-D model with diffusion in x- and y direction and first-order
## consumption - unefficient implementation
## ============================================================================

N     <- 51          # number of grid cells
XX    <- 10           # total size
dy    <- dx <- XX/N  # grid size
Dy    <- Dx <- 0.1   # diffusion coeff, X- and Y-direction
r     <- 0.005       # consumption rate
ini   <- 1           # initial value at x=0

N2  <- ceiling(N/2)
X   <- seq (dx, by = dx, len = (N2-1))
X   <- c(-rev(X), 0, X)

# The model equations

Diff2D <- function (t, y, parms)  {

 CONC  <- matrix(nrow = N, ncol = N, y)
 dCONC <- tran.2D(CONC, D.x = Dx, D.y = Dy, dx = dx, dy = dy)$dC + r * CONC
```

```
 return (list(dCONC))

}

# initial condition: 0 everywhere, except in central point
y <- matrix(nrow = N, ncol = N, data = 0)
y[N2, N2] <- ini  # initial concentration in the central point...

# solve for 10 time units
times <- 0:10
out <- ode.2D (y = y, func = Diff2D, t = times, parms = NULL,
               dim = c(N,N), lrw = 160000)

pm <- par (mfrow = c(2, 2))

# Compare solution with analytical solution...
for (i in seq(2, 11, by = 3))  {
  tt   <- times[i]
  mat  <-  matrix(nrow = N, ncol = N,
                  data = subset(out, time == tt))
  plot(X, mat[N2,], type = "l", main = paste("time=", times[i]),
       ylab = "Conc", col = "red")
  ana <- ini*dx^2/(4*pi*Dx*tt)*exp(r*tt-X^2/(4*Dx*tt))
  points(X, ana, pch = "+")
}

legend ("bottom", col = c("red","black"), lty = c(1, NA),
  pch = c(NA, "+"), c("tran.2D", "exact"))
par("mfrow" = pm )



## =============================================================================
## A 2-D model with diffusion in x- and y direction and first-order
## consumption - more efficient implementation, specifying ALL 2-D grids
## =============================================================================

N     <- 51          # number of grid cells
Dy    <- Dx <- 0.1   # diffusion coeff, X- and Y-direction
r     <- 0.005       # consumption rate
ini   <- 1           # initial value at x=0

x.grid    <- setup.grid.1D(x.up = -5, x.down = 5, N = N)
y.grid    <- setup.grid.1D(x.up = -5, x.down = 5, N = N)
grid2D    <- setup.grid.2D(x.grid, y.grid)

D.grid    <- setup.prop.2D(value = Dx, y.value = Dy, grid = grid2D)
v.grid    <- setup.prop.2D(value = 0, grid = grid2D)
A.grid    <- setup.prop.2D(value = 1, grid = grid2D)
AFDW.grid <- setup.prop.2D(value = 1, grid = grid2D)
VF.grid   <- setup.prop.2D(value = 1, grid = grid2D)
```

```
# The model equations - using the grids

Diff2Db <- function (t, y, parms)  {

   CONC  <- matrix(nrow = N, ncol = N, data = y)

   dCONC <- tran.2D(CONC, grid = grid2D, D.grid = D.grid,
      A.grid = A.grid, VF.grid = VF.grid, AFDW.grid = AFDW.grid,
      v.grid = v.grid)$dC + r * CONC

  return (list(dCONC))
}

# initial condition: 0 everywhere, except in central point
y <- matrix(nrow = N, ncol = N, data = 0)
y[N2,N2] <- ini  # initial concentration in the central point...

# solve for 8 time units
times <- 0:8
outb <- ode.2D (y = y, func = Diff2Db, t = times, parms = NULL,
                dim = c(N, N), lrw = 160000)

image(outb, ask = FALSE, mfrow = c(3, 3), main = paste("time", times))

## ================================================================================
## Same 2-D model, but now with spatially-variable diffusion coefficients
## ================================================================================

N     <- 51          # number of grid cells
r     <- 0.005       # consumption rate
ini   <- 1           # initial value at x=0
N2    <- ceiling(N/2)

D.grid <- list()

# Diffusion on x-interfaces
D.grid$x.int <- matrix(nrow = N+1, ncol = N, data = runif(N*(N+1)))

# Diffusion on y-interfaces
D.grid$y.int <- matrix(nrow = N, ncol = N+1, data = runif(N*(N+1)))

dx <- 10/N
dy <- 10/N

# The model equations

Diff2Dc <- function (t, y, parms)  {

   CONC  <- matrix(nrow = N, ncol = N, data = y)

   dCONC <- tran.2D(CONC, dx = dx, dy = dy, D.grid = D.grid)$dC + r * CONC

  return (list(dCONC))
```

```
}

# initial condition: 0 everywhere, except in central point
y <- matrix(nrow = N, ncol = N, data = 0)
y[N2, N2] <- ini  # initial concentration in the central point...

# solve for 8 time units
times <- 0:8
outc <- ode.2D (y = y, func = Diff2Dc, t = times, parms = NULL,
                dim = c(N, N), lrw = 160000)

outtimes <- c(1, 3, 5, 7)
image(outc, ask = FALSE, mfrow = c(2, 2), main = paste("time", outtimes),
      legend = TRUE, add.contour = TRUE, subset = time %in% outtimes)
```

`tran.3D` *General Three-Dimensional Advective-Diffusive Transport*

Description

Estimates the transport term (i.e. the rate of change of a concentration due to diffusion and advection) in a three-dimensional rectangular model domain.

Do not use with too many boxes!

Usage

```
tran.3D (C, C.x.up = C[1,,], C.x.down = C[dim(C)[1],,],
         C.y.up = C[ ,1, ],  C.y.down=C[ ,dim(C)[2], ],
         C.z.up = C[ , ,1],  C.z.down=C[ , ,dim(C)[3]],
         flux.x.up = NULL, flux.x.down = NULL,
         flux.y.up = NULL, flux.y.down = NULL,
         flux.z.up = NULL, flux.z.down = NULL,
         a.bl.x.up = NULL, a.bl.x.down = NULL,
         a.bl.y.up = NULL, a.bl.y.down = NULL,
         a.bl.z.up = NULL, a.bl.z.down = NULL,
         D.grid = NULL, D.x = NULL, D.y = D.x, D.z = D.x,
         v.grid = NULL, v.x = 0, v.y = 0, v.z = 0,
         AFDW.grid = NULL, AFDW.x = 1, AFDW.y = AFDW.x, AFDW.z = AFDW.x,
         VF.grid = NULL, VF.x = 1, VF.y = VF.x, VF.z = VF.x,
         A.grid = NULL, A.x = 1, A.y = 1, A.z = 1,
         grid = NULL, dx = NULL, dy = NULL, dz = NULL,
         full.check = FALSE, full.output = FALSE)
```

Arguments

`C`	concentration, expressed per unit volume, defined at the centre of each grid cell; Nx*Ny*Nz array [M/L3].

C.x.up	concentration at upstream boundary in x-direction; matrix of dimensions Ny*Nz [M/L3].
C.x.down	concentration at downstream boundary in x-direction; matrix of dimensions Ny*Nz [M/L3].
C.y.up	concentration at upstream boundary in y-direction; matrix of dimensions Nx*Nz [M/L3].
C.y.down	concentration at downstream boundary in y-direction; matrix of dimensions Nx*Nz [M/L3].
C.z.up	concentration at upstream boundary in z-direction; matrix of dimensions Nx*Ny [M/L3].
C.z.down	concentration at downstream boundary in z-direction; matrix of dimensions Nx*Ny [M/L3].
flux.x.up	flux across the upstream boundary in x-direction, positive = INTO model domain; matrix of dimensions Ny*Nz [M/L2/T].
flux.x.down	flux across the downstream boundary in x-direction, positive = OUT of model domain; matrix of dimensions Ny*Nz [M/L2/T].
flux.y.up	flux across the upstream boundary in y-direction, positive = INTO model domain; matrix of dimensions Nx*Nz [M/L2/T].
flux.y.down	flux across the downstream boundary in y-direction, positive = OUT of model domain; matrix of dimensions Nx*Nz [M/L2/T].
flux.z.up	flux across the upstream boundary in z-direction, positive = INTO model domain; matrix of dimensions Nx*Ny [M/L2/T].
flux.z.down	flux across the downstream boundary in z-direction, positive = OUT of model domain; matrix of dimensions Nx*Ny [M/L2/T].
a.bl.x.up	transfer coefficient across the upstream boundary layer. in x-direction `Flux=a.bl.x.up*(C.x.up-C[1,,])`. One value [L/T].
a.bl.x.down	transfer coefficient across the downstream boundary layer in x-direction; `Flux=a.bl.x.down*(C[Nx,,]-C.x.down)`. One value [L/T].
a.bl.y.up	transfer coefficient across the upstream boundary layer. in y-direction `Flux=a.bl.y.up*(C.y.up-C[,1,])`. One value [L/T].
a.bl.y.down	transfer coefficient across the downstream boundary layer in y-direction; `Flux=a.bl.y.down*(C[,Ny,]-C.y.down)`. One value [L/T].
a.bl.z.up	transfer coefficient across the upstream boundary layer. in y-direction `Flux=a.bl.y.up*(C.y.up-C[,,1])`. One value [L/T].
a.bl.z.down	transfer coefficient across the downstream boundary layer in z-direction; `Flux=a.bl.z.down*(C[,,Nz]-C.z.down)`. One value [L/T].
D.grid	diffusion coefficient defined on all grid cell interfaces. Should contain elements x.int, y.int, z.int, arrays with the values on the interfaces in x, y and z-direction, and with dimensions (Nx+1)*Ny*Nz, Nx*(Ny+1)*Nz and Nx*Ny*(Nz+1) respectively. [L2/T].
D.x	diffusion coefficient in x-direction, defined on grid cell interfaces. One value, a vector of length (Nx+1), or a (Nx+1)* Ny *Nz array [L2/T].

D.y	diffusion coefficient in y-direction, defined on grid cell interfaces. One value, a vector of length (Ny+1), or a Nx*(Ny+1)*Nz array [L2/T].
D.z	diffusion coefficient in z-direction, defined on grid cell interfaces. One value, a vector of length (Nz+1), or a Nx*Ny*(Nz+1) array [L2/T].
v.grid	advective velocity defined on all grid cell interfaces. Can be positive (downstream flow) or negative (upstream flow). Should contain elements x.int, y.int, z.int, arrays with the values on the interfaces in x, y and z-direction, and with dimensions (Nx+1)*Ny*Nz, Nx*(Ny+1)*Nz and Nx*Ny*(Nz+1) respectively. [L/T].
v.x	advective velocity in the x-direction, defined on grid cell interfaces. Can be positive (downstream flow) or negative (upstream flow). One value, a vector of length (Nx+1), or a (Nx+1)*Ny*Nz array [L/T].
v.y	advective velocity in the y-direction, defined on grid cell interfaces. Can be positive (downstream flow) or negative (upstream flow). One value, a vector of length (Ny+1), or a Nx*(Ny+1)*Nz array [L/T].
v.z	advective velocity in the z-direction, defined on grid cell interfaces. Can be positive (downstream flow) or negative (upstream flow). One value, a vector of length (Nz+1), or a Nx*Ny*(Nz+1) array [L/T].
AFDW.grid	weight used in the finite difference scheme for advection in the x-direction, defined on grid cell interfaces; backward = 1, centred = 0.5, forward = 0; default is backward. Should contain elements x.int, y.int, z.int, arrays with the values on the interfaces in x, y and z-direction, and with dimensions (Nx+1)*Ny*Nz, Nx*(Ny+1)*Nz and Nx*Ny*(Nz+1) respectively. [-].
AFDW.x	weight used in the finite difference scheme for advection in the x-direction, defined on grid cell interfaces; backward = 1, centred = 0.5, forward = 0; default is backward. One value, a vector of length (Nx+1), a prop.1D list created by setup.prop.1D, or a (Nx+1)*Ny*Nz array [-].
AFDW.y	weight used in the finite difference scheme for advection in the y-direction, defined on grid cell interfaces; backward = 1, centred = 0.5, forward = 0; default is backward. One value, a vector of length (Ny+1), a prop.1D list created by setup.prop.1D, or a Nx*(Ny+1)*Nz array [-].
AFDW.z	weight used in the finite difference scheme for advection in the z-direction, defined on grid cell interfaces; backward = 1, centred = 0.5, forward = 0; default is backward. One value, a vector of length (Nz+1), a prop.1D list created by setup.prop.1D, or a Nx*Ny*(Nz+1) array [-].
VF.grid	Volume fraction. A list. Should contain elements x.int, y.int, z.int, arrays with the values on the interfaces in x, y and z-direction, and with dimensions (Nx+1)*Ny*Nz, Nx*(Ny+1)*Nz and Nx*Ny*(Nz+1) respectively. [-].
VF.x	Volume fraction at the grid cell interfaces in the x-direction. One value, a vector of length (Nx+1), a prop.1D list created by setup.prop.1D, or a (Nx+1)*Ny*Nz array [-].
VF.y	Volume fraction at the grid cell interfaces in the y-direction. One value, a vector of length (Ny+1), a prop.1D list created by setup.prop.1D, or a Nx*(Ny+1)*Nz array [-].

VF.z	Volume fraction at the grid cell interfaces in the z-direction. One value, a vector of length (Nz+1), a prop.1D list created by setup.prop.1D, or a Nx*Ny*(Nz+1) array [-].
A.grid	Interface area, a list. Should contain elements x.int, y.int, z.int, arrays with the values on the interfaces in x, y and z-direction, and with dimensions (Nx+1)*Ny*Nz, Nx*(Ny+1)*Nz and Nx*Ny*(Nz+1) respectively. [L2].
A.x	Interface area defined at the grid cell interfaces in the x-direction. One value, a vector of length (Nx+1), a prop.1D list created by setup.prop.1D, or a (Nx+1)*Ny*Nz array [L2].
A.y	Interface area defined at the grid cell interfaces in the y-direction. One value, a vector of length (Ny+1), a prop.1D list created by setup.prop.1D, or a Nx*(Ny+1)*Nz array [L2].
A.z	Interface area defined at the grid cell interfaces in the z-direction. One value, a vector of length (Nz+1), a prop.1D list created by setup.prop.1D, or a Nx*Ny*(Nz+1) array [L2].
dx	distance between adjacent cell interfaces in the x-direction (thickness of grid cells). One value or vector of length Nx [L].
dy	distance between adjacent cell interfaces in the y-direction (thickness of grid cells). One value or vector of length Ny [L].
dz	distance between adjacent cell interfaces in the z-direction (thickness of grid cells). One value or vector of length Nz [L].
grid	discretization grid, a list containing at least elements dx, dx.aux, dy, dy.aux, dz, dz.aux (see setup.grid.2D) [L].
full.check	logical flag enabling a full check of the consistency of the arguments (default = FALSE; TRUE slows down execution by 50 percent).
full.output	logical flag enabling a full return of the output (default = FALSE; TRUE slows down execution by 20 percent).

Details

Do not use this with too large grid.

The **boundary conditions** are either

- (1) zero-gradient
- (2) fixed concentration
- (3) convective boundary layer
- (4) fixed flux

This is also the order of priority. The zero gradient is the default, the fixed flux overrules all other.

Value

a list containing:

dC	the rate of change of the concentration C due to transport, defined in the centre of each grid cell, an array with dimension Nx*Ny*Nz [M/L3/T].

C.x.up	concentration at the upstream interface in x-direction. A matrix of dimension Ny*Nz [M/L3]. Only when full.output = TRUE.
C.x.down	concentration at the downstream interface in x-direction. A matrix of dimension Ny*Nz [M/L3]. Only when full.output = TRUE.
C.y.up	concentration at the upstream interface in y-direction. A matrix of dimension Nx*Nz [M/L3]. Only when full.output = TRUE.
C.y.down	concentration at the downstream interface in y-direction. A matrix of dimension Nx*Nz [M/L3]. Only when full.output = TRUE.
C.z.up	concentration at the upstream interface in z-direction. A matrix of dimension Nx*Ny [M/L3]. Only when full.output = TRUE.
C.z.down	concentration at the downstream interface in z-direction. A matrix of dimension Nx*Ny [M/L3]. Only when full.output = TRUE.
x.flux	flux across the interfaces in x-direction of the grid cells. A (Nx+1)*Ny*Nz array [M/L2/T]. Only when full.output = TRUE.
y.flux	flux across the interfaces in y-direction of the grid cells. A Nx*(Ny+1)*Nz array [M/L2/T]. Only when full.output = TRUE.
z.flux	flux across the interfaces in z-direction of the grid cells. A Nx*Ny*(Nz+1) array [M/L2/T]. Only when full.output = TRUE.
flux.x.up	flux across the upstream boundary in x-direction, positive = INTO model domain. A matrix of dimension Ny*Nz [M/L2/T].
flux.x.down	flux across the downstream boundary in x-direction, positive = OUT of model domain. A matrix of dimension Ny*Nz [M/L2/T].
flux.y.up	flux across the upstream boundary in y-direction, positive = INTO model domain. A matrix of dimension Nx*Nz [M/L2/T].
flux.y.down	flux across the downstream boundary in y-direction, positive = OUT of model domain. A matrix of dimension Nx*Nz [M/L2/T].
flux.z.up	flux across the upstream boundary in z-direction, positive = INTO model domain. A matrix of dimension Nx*Ny [M/L2/T].
flux.z.down	flux across the downstream boundary in z-direction, positive = OUT of model domain. A matrix of dimension Nx*Ny [M/L2/T].

Author(s)

Filip Meysman <filip.meysman@nioz.nl>, Karline Soetaert <karline.soetaert@nioz.nl>

References

Soetaert and Herman, a practical guide to ecological modelling - using R as a simulation platform, 2009. Springer

See Also

tran.cylindrical, tran.spherical for a discretisation of 3-D transport equations in cylindrical and spherical coordinates

tran.1D, tran.2D

Examples

```
## ========================================================================
## Diffusion in 3-D; imposed boundary conditions
## ========================================================================
diffusion3D <- function(t, Y, par) {

  yy    <- array(dim = c(n, n, n), data = Y)  # vector to 3-D array
  dY    <- -r * yy                            # consumption
  BND   <- matrix(nrow = n, ncol = n, 1)      # boundary concentration

  dY <- dY + tran.3D(C = yy,
      C.x.up = BND, C.y.up = BND, C.z.up = BND,
      C.x.down = BND, C.y.down = BND, C.z.down = BND,
      D.x = Dx, D.y = Dy, D.z = Dz,
      dx = dx, dy = dy, dz = dz, full.check = TRUE)$dC
  return(list(dY))
}

# parameters
dy   <- dx <- dz <- 1   # grid size
Dy   <- Dx <- Dz <- 1   # diffusion coeff, X- and Y-direction
r    <- 0.025     # consumption rate

n  <- 10
y  <- array(dim = c(n, n, n), data = 10.)

print(system.time(
  ST3 <- steady.3D(y, func = diffusion3D, parms = NULL,
                pos = TRUE, dimens = c(n, n, n),
                lrw = 2000000, verbose = TRUE)
))

pm <- par(mfrow = c(1,1))
y <- array(dim = c(n, n, n), data = ST3$y)
filled.contour(y[ , ,n/2], color.palette = terrain.colors)

# a selection in the x-direction
image(ST3, mfrow = c(2, 2), add.contour = TRUE, legend = TRUE,
      dimselect = list(x = c(1, 4, 8, 10)))

par(mfrow = pm)
```

`tran.cylindrical` *Diffusive Transport in cylindrical (r, theta, z) and spherical (r, theta, phi) coordinates.*

Description

Estimates the transport term (i.e. the rate of change of a concentration due to diffusion) in a cylindrical (r, theta, z) or spherical (r, theta, phi) coordinate system.

Usage

```
tran.cylindrical (C, C.r.up = NULL, C.r.down = NULL,
                  C.theta.up = NULL, C.theta.down = NULL,
                  C.z.up = NULL, C.z.down = NULL,
                  flux.r.up = NULL, flux.r.down = NULL,
                  flux.theta.up = NULL, flux.theta.down = NULL,
                  flux.z.up = NULL, flux.z.down = NULL,
                  cyclicBnd = NULL,
                  D.r = NULL, D.theta = D.r, D.z = D.r,
                  r = NULL, theta = NULL, z = NULL)

tran.spherical (C, C.r.up = NULL, C.r.down = NULL,
                C.theta.up = NULL, C.theta.down = NULL,
                C.phi.up = NULL, C.phi.down = NULL,
                flux.r.up = NULL, flux.r.down = NULL,
                flux.theta.up = NULL, flux.theta.down = NULL,
                flux.phi.up = NULL, flux.phi.down = NULL,
                cyclicBnd = NULL,
                D.r = NULL, D.theta = D.r, D.phi = D.r,
                r = NULL, theta = NULL, phi = NULL)
```

Arguments

C	concentration, expressed per unit volume, defined at the centre of each grid cell; Nr*Nteta*Nz (cylindrica) or Nr*Ntheta*Nphi (spherical coordinates) array [M/L3].
C.r.up	concentration at upstream boundary in r(x)-direction; one value [M/L3].
C.r.down	concentration at downstream boundary in r(x)-direction; one value [M/L3].
C.theta.up	concentration at upstream boundary in theta-direction; one value [M/L3].
C.theta.down	concentration at downstream boundary in theta-direction; one value [M/L3].
C.z.up	concentration at upstream boundary in z-direction (cylindrical coordinates); one value [M/L3].
C.z.down	concentration at downstream boundary in z-direction(cylindrical coordinates); one value [M/L3].
C.phi.up	concentration at upstream boundary in phi-direction (spherical coordinates); one value [M/L3].
C.phi.down	concentration at downstream boundary in phi-direction(spherical coordinates); one value [M/L3].
flux.r.up	flux across the upstream boundary in r-direction, positive = INTO model domain; one value [M/L2/T].
flux.r.down	flux across the downstream boundary in r-direction, positive = OUT of model domain; one value [M/L2/T].
flux.theta.up	flux across the upstream boundary in theta-direction, positive = INTO model domain; one value [M/L2/T].

`flux.theta.down`
: flux across the downstream boundary in theta-direction, positive = OUT of model domain; one value [M/L2/T].

`flux.z.up`
: flux across the upstream boundary in z-direction(cylindrical coordinates); positive = INTO model domain; one value [M/L2/T].

`flux.z.down`
: flux across the downstream boundary in z-direction, (cylindrical coordinates); positive = OUT of model domain; one value [M/L2/T].

`flux.phi.up`
: flux across the upstream boundary in phi-direction(spherical coordinates); positive = INTO model domain; one value [M/L2/T].

`flux.phi.down`
: flux across the downstream boundary in phi-direction, (spherical coordinates); positive = OUT of model domain; one value [M/L2/T].

`cyclicBnd`
: If not `NULL`, the direction in which a cyclic boundary is defined, i.e. `cyclicBnd = 1` for the `r` direction, `cyclicBnd = 2` for the `theta` direction and `cyclicBnd = c(1,2)` for both the `r` and `theta` direction.

`D.r`
: diffusion coefficient in r-direction, defined on grid cell interfaces. One value or a vector of length (Nr+1), [L2/T].

`D.theta`
: diffusion coefficient in theta-direction, defined on grid cell interfaces. One value or or a vector of length (Ntheta+1), [L2/T].

`D.z`
: diffusion coefficient in z-direction, defined on grid cell interfaces for cylindrical coordinates. One value or a vector of length (Nz+1) [L2/T].

`D.phi`
: diffusion coefficient in phi-direction, defined on grid cell interfaces for cylindrical coordinates. One value or a vector of length (Nphi+1) [L2/T].

`r`
: position of adjacent cell interfaces in the r-direction. A vector of length Nr+1 [L].

`theta`
: position of adjacent cell interfaces in the theta-direction. A vector of length Ntheta+1 [L]. Theta should be within [0,2 pi]

`z`
: position of adjacent cell interfaces in the z-direction (cylindrical coordinates). A vector of length Nz+1 [L].

`phi`
: position of adjacent cell interfaces in the phi-direction (spherical coordinates). A vector of length Nphi+1 [L]. Phi should be within [0,2 pi]

Details

`tran.cylindrical` performs (diffusive) transport in cylindrical coordinates

`tran.spherical` performs (diffusive) transport in spherical coordinates

The **boundary conditions** are either

- (1) zero gradient
- (2) fixed concentration
- (3) fixed flux
- (4) cyclic boundary

This is also the order of priority. The cyclic boundary overrules the other. If fixed concentration, fixed flux, and cyclicBnd are NULL then the boundary is zero-gradient

A cyclic boundary condition has concentration and flux at upstream and downstream boundary the same. It is useful mainly for the theta and phi direction.

** Do not expect too much of this equation: do not try to use it with many boxes **

Value

a list containing:

dC	the rate of change of the concentration C due to transport, defined in the centre of each grid cell, a Nr*Nteta*Nz (cylindrical) or Nr*Ntheta*Nphi (spherical coordinates) array. [M/L3/T].
flux.r.up	flux across the upstream boundary in r-direction, positive = INTO model domain. A matrix of dimension Nteta*Nz (cylindrical) or Ntheta*Nphi (spherical) [M/L2/T].
flux.r.down	flux across the downstream boundary in r-direction, positive = OUT of model domain. A matrix of dimension Nteta*Nz (cylindrical) or Ntheta*Nphi (spherical) [M/L2/T].
flux.theta.up	flux across the upstream boundary in theta-direction, positive = INTO model domain. A matrix of dimension Nr*Nz (cylindrical) or or Nr*Nphi (spherical) [M/L2/T].
flux.theta.down	flux across the downstream boundary in theta-direction, positive = OUT of model domain. A matrix of dimension Nr*Nz (cylindrical) or Nr*Nphi (spherical) [M/L2/T].
flux.z.up	flux across the upstream boundary in z-direction, for cylindrical coordinates; positive = OUT of model domain. A matrix of dimension Nr*Nteta [M/L2/T].
flux.z.down	flux across the downstream boundary in z-direction for cylindrical coordinates; positive = OUT of model domain. A matrix of dimension Nr*Nteta [M/L2/T].
flux.phi.up	flux across the upstream boundary in phi-direction, for spherical coordinates; positive = OUT of model domain. A matrix of dimension Nr*Nteta [M/L2/T].
flux.phi.down	flux across the downstream boundary in phi-direction, for spherical coordinates; positive = OUT of model domain. A matrix of dimension Nr*Nteta [M/L2/T].

See Also

tran.polar for a discretisation of 2-D transport equations in polar coordinates

tran.1D, tran.2D, tran.3D

Examples

```
## =============================================================================
## Testing the functions
## =============================================================================
# Grid definition
r.N     <- 4   # number of cells in r-direction
```

```
theta.N <- 6   # number of cells in theta-direction
z.N     <- 3   # number of cells in z-direction

D       <- 100 # diffusion coefficient

r      <- seq(0,   8, len = r.N+1)       # cell size r-direction [cm]
theta  <- seq(0,2*pi, len = theta.N+1)   # theta-direction - theta: from 0, 2pi
phi    <- seq(0,2*pi, len = z.N+1)       # phi-direction (0,2pi)
z      <- seq(0,5, len = z.N+1)          # cell size z-direction [cm]

# Intial conditions
C <- array(dim = c(r.N, theta.N, z.N), data = 0)

# Concentration boundary conditions
tran.cylindrical (C = C, D.r = D, D.theta = D,
  C.r.up = 1, C.r.down = 1,
  C.theta.up = 1, C.theta.down = 1,
  C.z.up = 1, C.z.down = 1,
  r = r, theta = theta, z = z )

tran.spherical (C = C, D.r = D, D.theta = D,
  C.r.up = 1, C.r.down = 1, C.theta.up = 1, C.theta.down = 1,
  C.phi.up = 1, C.phi.down = 1,
  r = r, theta = theta, phi = phi)

# Flux boundary conditions
tran.cylindrical(C = C, D.r = D, r = r, theta = theta, z = z,
  flux.r.up = 10, flux.r.down = 10,
  flux.theta.up = 10, flux.theta.down = 10,
  flux.z.up = 10, flux.z.down = 10)

tran.spherical(C = C, D.r = D, r = r, theta = theta, phi = phi,
  flux.r.up = 10, flux.r.down = 10,
  flux.theta.up = 10, flux.theta.down = 10,
  flux.phi.up = 10, flux.phi.down = 10)

# cyclic boundary conditions
tran.cylindrical(C = C, D.r = D, r = r, theta = theta, z = z,
  cyclicBnd = 1:3)
tran.spherical(C = C, D.r = D, r = r, theta = theta, phi = phi,
  cyclicBnd = 1:3)

# zero-gradient boundary conditions
tran.cylindrical(C = C, D.r = D, r = r, theta = theta, z = z)
tran.spherical(C = C, D.r = D, r = r, theta = theta, phi = phi)

## ============================================================================
## A model with diffusion and first-order consumption
## ============================================================================

N     <- 10          # number of grid cells
rr    <- 0.005       # consumption rate
D     <- 400
```

```
r       <- seq (2, 4, len = N+1)
theta   <- seq (0, 2*pi, len = N+1)
z       <- seq (0, 3, len = N+1)
phi     <- seq (0, 2*pi, len = N+1)

# The model equations
Diffcylin <- function (t, y, parms)  {
  CONC  <- array(dim = c(N, N, N), data = y)
  tran  <- tran.cylindrical(CONC,
        D.r = D, D.theta = D, D.z = D,
        r = r, theta = theta, z = z,
        C.r.up = 0,  C.r.down = 1,
        cyclicBnd = 2)
  dCONC <- tran$dC  - rr * CONC
  return (list(dCONC))
}

Diffspher <- function (t, y, parms)  {
  CONC  <- array(dim = c(N, N, N), data = y)
  tran  <- tran.spherical (CONC,
        D.r = D, D.theta = D, D.phi = D,
        r = r, theta = theta, phi = phi,
        C.r.up = 0,  C.r.down = 1,
        cyclicBnd = 2:3)
  dCONC <- tran$dC  - rr * CONC
  return (list(dCONC))
}

# initial condition: 0 everywhere, except in central point
y   <- array(dim = c(N, N, N), data = 0)
N2  <- ceiling(N/2)

y[N2, N2, N2] <- 100  # initial concentration in the central point...

# solve to steady-state; cyclicBnd = 2,
outcyl <- steady.3D (y = y, func = Diffcylin, parms = NULL,
                  dim = c(N, N, N), lrw = 1e6, cyclicBnd = 2)

STDcyl <- array(dim = c(N, N, N), data = outcyl$y)
image(STDcyl[,,1])

# For spherical coordinates, cyclic Bnd = 2, 3
outspher <- steady.3D (y = y, func = Diffspher, parms = NULL, pos=TRUE,
                  dim = c(N, N, N), lrw = 1e6, cyclicBnd = 2:3)

#STDspher <- array(dim = c(N, N, N), data = outspher$y)
#image(STDspher[,,1])

## Not run:
  image(outspher)

## End(Not run)
```

tran.polar *Diffusive Transport in polar (r, theta) coordinates.*

Description

Estimates the transport term (i.e. the rate of change of a concentration due to diffusion) in a polar (r, theta) coordinate system

Usage

```
tran.polar (C, C.r.up = NULL, C.r.down = NULL,
            C.theta.up = NULL, C.theta.down = NULL,
            flux.r.up = NULL, flux.r.down = NULL,
            flux.theta.up = NULL, flux.theta.down = NULL,
            cyclicBnd = NULL, D.r = 1, D.theta = D.r,
            r = NULL, theta = NULL, full.output = FALSE)

polar2cart (out, r, theta, x = NULL, y = NULL)
```

Arguments

C	concentration, expressed per unit volume, defined at the centre of each grid cell; Nr*Nteta matrix [M/L3].
C.r.up	concentration at upstream boundary in r(x)-direction; vector of length Nteta [M/L3].
C.r.down	concentration at downstream boundary in r(x)-direction; vector of length Nteta [M/L3].
C.theta.up	concentration at upstream boundary in theta-direction; vector of length Nr [M/L3].
C.theta.down	concentration at downstream boundary in theta-direction; vector of length Nr [M/L3].
flux.r.up	flux across the upstream boundary in r-direction, positive = INTO model domain; vector of length Ntheta [M/L2/T].
flux.r.down	flux across the downstream boundary in r-direction, positive = OUT of model domain; vector of length Ntheta [M/L2/T].
flux.theta.up	flux across the upstream boundary in theta-direction, positive = INTO model domain; vector of length Nr [M/L2/T].
flux.theta.down	flux across the downstream boundary in theta-direction, positive = OUT of model domain; vector of length Nr [M/L2/T].
cyclicBnd	If not NULL, the direction in which a cyclic boundary is defined, i.e. cyclicBnd = 1 for the r direction, cyclicBnd = 2 for the theta direction and cyclicBnd = c(1,2) for both the r and theta direction.

D.r	diffusion coefficient in r-direction, defined on grid cell interfaces. One value, a vector of length (Nr+1), a prop.1D list created by setup.prop.1D, or a (Nr+1)* Nteta matrix [L2/T].
D.theta	diffusion coefficient in theta-direction, defined on grid cell interfaces. One value, a vector of length (Ntheta+1), a prop.1D list created by setup.prop.1D, or a Nr*(Ntheta+1) matrix [L2/T].
r	position of adjacent cell interfaces in the r-direction. A vector of length Nr+1 [L].
theta	position of adjacent cell interfaces in the theta-direction. A vector of length Ntheta+1 [L]. Theta should be within [0,2 pi]
full.output	logical flag enabling a full return of the output (default = FALSE; TRUE slows down execution by 20 percent).
out	output as returned by tran.polar, and which is to be mapped from polar to cartesian coordinates
x	The cartesian x-coordinates to whicht the polar coordinates are to be mapped
y	The cartesian y-coordinates to whicht the polar coordinates are to be mapped

Details

tran.polar performs (simplified) transport in polar coordinates

The **boundary conditions** are either

- (1) zero gradient
- (2) fixed concentration
- (3) fixed flux
- (4) cyclic boundary

This is also the order of priority. The cyclic boundary overrules the other. If fixed concentration, fixed flux, and cyclicBnd are NULL then the boundary is zero-gradient

A cyclic boundary condition has concentration and flux at upstream and downstream boundary the same.

polar2cart maps the polar coordinates to cartesian coordinates

If x and y is not provided, then it will create an (x,y) grid based on r : seq(-maxr, maxr, length.out=Nr), where maxr is the maximum value of r, and Nr is the number of elements in r.

Value

a list containing:

dC	the rate of change of the concentration C due to transport, defined in the centre of each grid cell, a Nr*Nteta matrix. [M/L3/T].
C.r.up	concentration at the upstream interface in r-direction. A vector of length Nteta [M/L3]. Only when full.output = TRUE.
C.r.down	concentration at the downstream interface in r-direction. A vector of length Nteta [M/L3]. Only when full.output = TRUE.

C.theta.up	concentration at the the upstream interface in theta-direction. A vector of length Nr [M/L3]. Only when full.output = TRUE.
C.theta.down	concentration at the downstream interface in theta-direction. A vector of length Nr [M/L3]. Only when full.output = TRUE.
r.flux	flux across the interfaces in x-direction of the grid cells. A (Nr+1)*Nteta matrix [M/L2/T]. Only when full.output = TRUE.
theta.flux	flux across the interfaces in y-direction of the grid cells. A Nr*(Nteta+1) matrix [M/L2/T]. Only when full.output = TRUE.
flux.r.up	flux across the upstream boundary in r-direction, positive = INTO model domain. A vector of length Nteta [M/L2/T].
flux.r.down	flux across the downstream boundary in r-direction, positive = OUT of model domain. A vector of length Nteta [M/L2/T].
flux.theta.up	flux across the upstream boundary in theta-direction, positive = INTO model domain. A vector of length Nr [M/L2/T].
flux.theta.down	flux across the downstream boundary in theta-direction, positive = OUT of model domain. A vector of length Nr [M/L2/T].

References

Soetaert and Herman, 2009. a practical guide to ecological modelling - using R as a simulation platform. Springer

See Also

tran.cylindrical, tran.spherical for a discretisation of 3-D transport equations in cylindrical and spherical coordinates

tran.1D, tran.2D, tran.3D

Examples

```
## =============================================================================
## Testing the functions
## =============================================================================
# Parameters
F        <- 100            # input flux [micromol cm-2 yr-1]
D        <- 400            # mixing coefficient [cm2 yr-1]

# Grid definition
r.N   <- 4     # number of cells in r-direction
theta.N <- 6   # number of cells in theta-direction
r.L <- 8       # domain size r-direction [cm]
r      <- seq(0, r.L,len = r.N+1)      # cell size r-direction [cm]
theta  <- seq(0, 2*pi,len = theta.N+1) # theta-direction - theta: from 0, 2pi

# Intial conditions
C <- matrix(nrow = r.N, ncol = theta.N, data = 0)
```

```
# Boundary conditions: fixed concentration
C.r.up       <- rep(1, times = theta.N)
C.r.down     <- rep(0, times = theta.N)
C.theta.up   <- rep(1, times = r.N)
C.theta.down <- rep(0, times = r.N)

# Concentration boundary conditions
tran.polar(C = C, D.r = D, D.theta = D,
  r = r, theta = theta,
  C.r.up = C.r.up, C.r.down = C.r.down,
  C.theta.up = C.theta.up, C.theta.down = C.theta.down)

# Flux boundary conditions
flux.r.up <- rep(200, times = theta.N)
flux.r.down <- rep(-200, times = theta.N)
flux.theta.up <- rep(200, times = r.N)
flux.theta.down <- rep(-200, times = r.N)

tran.polar(C = C, D.r = D, r = r, theta = theta,
  flux.r.up = flux.r.up, flux.r.down = flux.r.down,
  flux.theta.up = flux.theta.up, flux.theta.down = flux.theta.down,
  full.output = TRUE)


## ================================================================================
## A model with diffusion and first-order consumption
## ================================================================================
N     <- 50          # number of grid cells
XX    <- 4           # total size
rr    <- 0.005       # consumption rate
ini   <- 1           # initial value at x=0
D     <- 400

r     <- seq (2, 4, len = N+1)
theta   <- seq(0, 2*pi, len = N+1)
theta.m <- 0.5*(theta[-1]+theta[-(N+1)])

# The model equations

Diffpolar <- function (t, y, parms)  {
  CONC  <- matrix(nrow = N, ncol = N, data = y)
  tran  <- tran.polar(CONC, D.r = D, D.theta = D, r = r, theta = theta,
        C.r.up = 0, C.r.down = 1*sin(5*theta.m),
        cyclicBnd = 2, full.output=TRUE )
  dCONC <- tran$dC  - rr * CONC
  return (list(dCONC))
}

# solve to steady-state; cyclicBnd = 2, because of C.theta.up, C.theta.down
out <- steady.2D (y = rep(0, N*N), func = Diffpolar, parms = NULL,
                  dim = c(N, N), lrw = 1e6, cyclicBnd = 2)

image(out)
```

```
cart <- polar2cart(out, r = r, theta = theta,
                        x = seq(-4, 4, len = 100),
                        y = seq(-4, 4, len = 100))
image(cart)
```

`tran.volume.1D` *1-D, 2-D and 3-D Volumetric Advective-Diffusive Transport in an Aquatic System*

Description

Estimates the volumetric transport term (i.e. the rate of change of the concentration due to diffusion and advection) in a 1-D, 2-D or 3-D model of an aquatic system (river, estuary).

Volumetric transport implies the use of flows (mass per unit of time) rather than fluxes (mass per unit of area per unit of time) as is done in `tran.1D`, `tran.2D` or `tran.3D`.

The `tran.volume.xD` routines are particularly suited for modelling channels (like rivers, estuaries) where the cross-sectional area changes, but where this area change needs not to be explicitly modelled as such.

Another difference with `tran.1D` is that the `tran.volume.1D` routine also allows lateral water or lateral mass input (as from side rivers or diffusive lateral ground water inflow).

The `tran.volume.2D` routine can check for water balance and assume an in- or efflux in case the net flows in and out of a box are not = 0

Usage

```
tran.volume.1D(C, C.up = C[1], C.down = C[length(C)],
               C.lat = C, F.up = NULL, F.down = NULL, F.lat = NULL,
               Disp,flow = 0, flow.lat = NULL, AFDW = 1,
               V = NULL, full.check = FALSE, full.output = FALSE)

tran.volume.2D(C, C.x.up = C[1, ], C.x.down = C[nrow(C), ],
               C.y.up = C[, 1], C.y.down = C[, ncol(C)],
               C.z = C, masscons = TRUE,
               F.x.up = NULL, F.x.down = NULL,
               F.y.up = NULL, F.y.down = NULL,
               Disp.grid = NULL, Disp.x = NULL, Disp.y = Disp.x,
               flow.grid = NULL, flow.x = NULL, flow.y = NULL,
               AFDW.grid = NULL, AFDW.x = 1, AFDW.y = AFDW.x,
               V = NULL, full.check = FALSE, full.output = FALSE)

tran.volume.3D(C,  C.x.up = C[1, , ], C.x.down = C[dim(C)[1], , ],
               C.y.up = C[, 1, ], C.y.down = C[, dim(C)[2], ],
               C.z.up = C[, , 1], C.z.down = C[, , dim(C)[3]],
               F.x.up = NULL, F.x.down = NULL,
```

```
F.y.up = NULL, F.y.down = NULL,
F.z.up = NULL, F.z.down = NULL,
Disp.grid = NULL,
Disp.x = NULL, Disp.y = Disp.x, Disp.z = Disp.x,
flow.grid = NULL, flow.x = 0, flow.y = 0, flow.z = 0,
AFDW.grid = NULL, AFDW.x = 1, AFDW.y = AFDW.x,
AFDW.z = AFDW.x,
V = NULL, full.check = FALSE, full.output = FALSE)
```

Arguments

C	tracer concentration, defined at the centre of the grid cells. A vector of length N [M/L3] (tran.volume.1D), a matrix of dimension Nr*Nc (tran.volume.2D) or an Nx*Ny*Nz array (tran.volume.3D) [M/L3].
C.up	tracer concentration at the upstream interface. One value [M/L3].
C.down	tracer concentration at downstream interface. One value [M/L3].
C.lat	tracer concentration in the lateral input, defined at grid cell centres. One value, a vector of length N, or a list as defined by setup.prop.1D [M/L3]. The default is C.lat = C, (a zero-gradient condition). Setting C.lat=0, together with a positive F.lat will lead to dilution of the tracer concentration in the grid cells.
C.x.up	concentration at upstream boundary in x-direction; vector of length Ny (2D) or matrix of dimensions Ny*Nz (3D) [M/L3].
C.x.down	concentration at downstream boundary in x-direction; vector of length Ny (2D) or matrix of dimensions Ny*Nz (3D) [M/L3].
C.y.up	concentration at upstream boundary in y-direction; vector of length Nx (2D) or matrix of dimensions Nx*Nz (3D) [M/L3].
C.y.down	concentration at downstream boundary in y-direction; vector of length Nx (2D) or matrix of dimensions Nx*Nz (3D) [M/L3].
C.z.up	concentration at upstream boundary in z-direction; matrix of dimensions Nx*Ny [M/L3].
C.z.down	concentration at downstream boundary in z-direction; matrix of dimensions Nx*Ny [M/L3].
C.z	concentration at boundary in z-direction for 2-D models where masscons = TRUE. Matrix of dimensions Nx*Ny [M/L3].
masscons	When TRUE, will check flow balance in 2D model. The flow in the third direction will then be estimated.
F.up	total tracer input at the upstream interface. One value [M/T].
F.down	total tracer input at downstream interface. One value [M/T].
F.lat	total lateral tracer input, defined at grid cell centres. One value, a vector of length N, or a 1D list property as defined by setup.prop.1D,[M/T].
F.x.up	total tracer input at the upstream interface in x-direction. positive = INTO model domain. A vector of length Ny (2D) or a matrix of dimensions Ny*Nz (3D) [M/T].

F.x.down — total tracer input at downstream interface in x-direction. positive = INTO model domain. A vector of length Ny (2D) or a matrix of dimensions Ny*Nz (3D) [M/T].

F.y.up — total tracer input at the upstream interface in y-direction. positive = INTO model domain. A vector of length Nx (2D) or a matrix of dimensions Nx*Nz (3D) [M/T].

F.y.down — total tracer input at downstream interface in y-direction. positive = INTO model domain. A vector of length Nx (2D) or a matrix of dimensions Nx*Nz (3D) [M/T].

F.z.up — total tracer input at the upstream interface in z-direction. positive = INTO model domain. A matrix of dimensions Nx*Ny [M/T].

F.z.down — total tracer input at downstream interface in z-direction. positive = INTO model domain. A matrix of dimensions Nx*Ny [M/T].

Disp.grid — BULK dispersion coefficients defined on all grid cell interfaces. For tran.volume.2D, should contain two matrices, x.int (dimension (Nx+1)*Ny) and y.int (dimension Nx * (Ny+1)). For tran.volume.3D should contain three arrays x.int (dim = (Nx+1)*Ny*Nz), y.int (dim = Nx*(Ny+1)*Nz), and z.int (dim = Nx*Ny*(Nz+1))

Disp — BULK dispersion coefficient, defined on grid cell interfaces. One value, a vector of length N+1, or a 1D list property as defined by setup.prop.1D [L3/T].

Disp.x — BULK dispersion coefficient in x-direction, defined on grid cell interfaces. One value, a vector of length (Nx+1), a prop.1D list created by setup.prop.1D, a (Nx+1)* Ny matrix (2D) or a Nx*(Ny+1)*Nz array (3D) [L3/T].

Disp.y — BULK dispersion coefficient in y-direction, defined on grid cell interfaces. One value, a vector of length (Ny+1), a prop.1D list created by setup.prop.1D, or a Nx*(Ny+1) matrix (2D) or a Nx*(Ny+1)*Nz array (3D)[L3/T].

Disp.z — BULK dispersion coefficient in z-direction, defined on grid cell interfaces. One value, a vector of length (Nz+1), or a Nx*Ny*(Nz+1) array [L3/T].

flow — water flow rate, defined on grid cell interfaces. One value, a vector of length N+1, or a list as defined by setup.prop.1D [L3/T]. If flow.lat is not NULL the flow should be one value containing the flow rate at the upstream boundary. If flow.lat is NULL then flow can be either one value, a vector or a list.

flow.lat — lateral water flow rate [L3/T] into each volume box, defined at grid cell centres. One value, a vector of length N, or a list as defined by setup.prop.1D. If flow.lat has a value, then flow should be the flow rate at the upstream interface (one value). For each grid cell, the flow at the downstream side of a grid cell is then estimated by water balance (adding flow.lat in the cell to flow rate at the upstream side of the grid cell). If flow.lat is NULL, then it is determined by water balance from flow.

flow.grid — flow rates defined on all grid cell interfaces. Can be positive (downstream flow) or negative (upstream flow). Should contain elements x.int, y.int, z.int (3-D), arrays with the values on the interfaces in x, y and z-direction [L3/T].

flow.x — flow rates in the x-direction, defined on grid cell interfaces. Can be positive (downstream flow) or negative (upstream flow). One value, a vector of length (Nx+1), a prop.1D list created by setup.prop.1D (2D), a (Nx+1)*Ny matrix (2D) or a (Nx+1)*Ny*Nz array (3D) [L3/T].

flow.y	flow rates in the y-direction, defined on grid cell interfaces. Can be positive (downstream flow) or negative (upstream flow). One value, a vector of length (Ny+1), a prop.1D list created by setup.prop.1D (2D), a Nx*(Ny+1) matrix (2D) or a Nx*(Ny+1)*Nz array [L3/T].
flow.z	flow rates in the z-direction, defined on grid cell interfaces. Can be positive (downstream flow) or negative (upstream flow). One value, a vector of length (Nz+1), or a Nx*Ny*(Nz+1) array [L3/T].
AFDW	weight used in the finite difference scheme for advection, defined on grid cell interfaces; backward = 1, centred = 0.5, forward = 0; default is backward. One value, a vector of length N+1, or a list as defined by setup.prop.1D [-].
AFDW.grid	weight used in the finite difference scheme for advection in the x-direction, defined on grid cell interfaces; backward = 1, centred = 0.5, forward = 0; default is backward. For tran.volume.3D should contain elements x.int, y.int, z.int (3D), for tran.volume.2D should contain elements x.int and y.int. [-].
AFDW.x	weight used in the finite difference scheme for advection in the x-direction, defined on grid cell interfaces; backward = 1, centred = 0.5, forward = 0; default is backward. One value, a vector of length (Nx+1), a prop.1D list created by setup.prop.1D, a (Nx+1)*Ny matrix (2D) or a (Nx+1)*Ny*Nz array (3D) [-].
AFDW.y	weight used in the finite difference scheme for advection in the y-direction, defined on grid cell interfaces; backward = 1, centred = 0.5, forward = 0; default is backward. One value, a vector of length (Ny+1), a prop.1D list created by setup.prop.1D, a Nx*(Ny+1) matrix (2D) or a Nx*(Ny+1)*Nz array [-].
AFDW.z	weight used in the finite difference scheme for advection in the z-direction, defined on grid cell interfaces; backward = 1, centred = 0.5, forward = 0; default is backward. One value, a vector of length (Nz+1), a prop.1D list created by setup.prop.1D, or a Nx*Ny*(Nz+1) array [-].
V	grid cell volume, defined at grid cell centres [L3]. One value, a vector of length N, or a list as defined by setup.prop.1D.
full.check	logical flag enabling a full check of the consistency of the arguments (default = FALSE; TRUE slows down execution by 50 percent).
full.output	logical flag enabling a full return of the output (default = FALSE; TRUE slows down execution by 20 percent).

Details

The **boundary conditions** are of type

- 1. zero-gradient (default)
- 2. fixed concentration
- 3. fixed input

The *bulk dispersion coefficient* (Disp) and the *flow rate* (flow) can be either one value or a vector of length N+1, defined at all grid cell interfaces, including upstream and downstream boundary.

The spatial discretisation is given by the volume of each box (V), which can be one value or a vector of length N+1, defined at the centre of each grid cell.

The water flow is mass conservative. Over each volume box, the routine calculates internally the downstream outflow of water in terms of the upstream inflow and the lateral inflow.

Value

dC	the rate of change of the concentration C due to transport, defined in the centre of each grid cell [M/L3/T].
F.up	mass flow across the upstream boundary, positive = INTO model domain. One value [M/T].
F.down	mass flow across the downstream boundary, positive = OUT of model domain. One value [M/T].
F.lat	lateral mass input per volume box, positive = INTO model domain. A vector of length N [M/T].
flow	water flow across the interface of each grid cell. A vector of length N+1 [L3/T]. Only provided when (full.output = TRUE
flow.up	water flow across the upstream (external) boundary, positive = INTO model domain. One value [L3/T]. Only provided when (full.output = TRUE)
flow.down	water flow across the downstream (external) boundary, positive = OUT of model domain. One value [L3/T]. Only provided when (full.output = TRUE)
flow.lat	lateral water input on each volume box, positive = INTO model domain. A vector of length N [L3/T]. Only provided when (full.output = TRUE)
F	mass flow across at the interface of each grid cell. A vector of length N+1 [M/T]. Only provided when (full.output = TRUE)

Author(s)

Filip Meysman <filip.meysman@nioz.nl>, Karline Soetaert <karline.soetaert@nioz.nl>

References

Soetaert and Herman (2009) A practical guide to ecological modelling - using R as a simulation platform. Springer.

See Also

tran.1D

advection.volume.1D, for more sophisticated advection schemes

Examples

```
## =============================================================================
##  EXAMPLE : organic carbon (OC) decay in a widening estuary
## =============================================================================

# Two scenarios are simulated: the baseline includes only input
# of organic matter upstream. The second scenario simulates the
# input of an important side river half way the estuary.

#===================#
# Model formulation  #
```

```
#====================#

river.model <- function (t = 0, OC, pars = NULL) {

  tran <- tran.volume.1D(C = OC, F.up = F.OC, F.lat = F.lat,
          Disp = Disp, flow = flow.up, flow.lat = flow.lat,
          V = Volume, full.output = TRUE)

  reac <- - k*OC
  return(list(dCdt = tran$dC + reac, Flow = tran$flow))
}

#======================#
# Parameter definition #
#======================#

# Initialising morphology estuary:

nbox          <- 500     # number of grid cells
lengthEstuary <- 100000  # length of estuary [m]
BoxLength     <- lengthEstuary/nbox # [m]
Distance      <- seq(BoxLength/2, by = BoxLength, len =nbox) # [m]
Int.Distance  <- seq(0, by = BoxLength, len = (nbox+1))      # [m]

# Cross sectional area: sigmoid function of estuarine distance [m2]
CrossArea <- 4000 + 72000 * Distance^5 /(Distance^5+50000^5)

# Volume of boxes                          (m3)
Volume  <- CrossArea*BoxLength

# Transport coefficients
Disp    <- 1000   # m3/s, bulk dispersion coefficient
flow.up  <- 180    # m3/s, main river upstream inflow
flow.lat.0  <- 180    # m3/s, side river inflow

F.OC    <- 180               # input organic carbon [mol s-1]
F.lat.0 <- 180              # lateral input organic carbon [mol s-1]

k       <- 10/(365*24*3600)  # decay constant organic carbon [s-1]


#====================#
# Model solution     #
#====================#
#scenario 1: without lateral input
F.lat    <- rep(0, length.out = nbox)
flow.lat <- rep(0, length.out = nbox)

Conc1 <- steady.1D(runif(nbox), fun = river.model, nspec = 1, name = "OC")

#scenario 2: with lateral input
F.lat <- F.lat.0 * dnorm(x =Distance/lengthEstuary,
                         mean = Distance[nbox/2]/lengthEstuary,
```

```
                          sd = 1/20, log = FALSE)/nbox
flow.lat <- flow.lat.0 * dnorm(x = Distance/lengthEstuary,
                               mean = Distance[nbox/2]/lengthEstuary,
                               sd = 1/20, log = FALSE)/nbox

Conc2 <- steady.1D(runif(nbox), fun = river.model, nspec = 1, name = "OC")

#====================#
# Plotting output    #
#====================#
# use S3 plot method
plot(Conc1, Conc2, grid = Distance/1000, which = "OC",
     mfrow = c(2, 1), lwd = 2, xlab = "distance [km]",
     main = "Organic carbon decay in the estuary",
     ylab = "OC Concentration [mM]")

plot(Conc1, Conc2, grid = Int.Distance/1000, which = "Flow",
     mfrow = NULL, lwd = 2, xlab = "distance [km]",
     main = "Longitudinal change in the water flow rate",
     ylab = "Flow rate [m3 s-1]")

legend ("topright", lty = 1:2, col = 1:2, lwd = 2,
        c("baseline", "+ side river input"))
```

Index