

Package ‘QPot’

August 29, 2016

Version 1.1

Date 2016-04-03

Title Quasi-Potential Analysis for Stochastic Differential Equations

Depends R (>= 3.0.2)

Imports MASS

Maintainer Christopher Stieha <stieha@hotmail.com>

Description Tools to 1) simulate and visualize stochastic differential equations and 2) determine stability of equilibria using the ordered-upwind method to compute the quasi-potential.

License GPL-2

URL <http://www.r-project.org>, <https://github.com/bmarkslash7/QPot>

BugReports <https://github.com/bmarkslash7/QPot/issues>

RoxygenNote 5.0.0

NeedsCompilation yes

Author Christopher Moore [aut],
Christopher Stieha [aut, cre],
Ben Nolting [aut],
Maria Cameron [aut],
Karen Abbott [aut],
James Gregson [cph] (author of expression_parser library:
https://github.com/jamesgregson/expression_parser)

Repository CRAN

Date/Publication 2016-04-04 01:07:18

R topics documented:

Model2String	2
QPContour	3
QPGlobal	6
QPIinterp	7

QPotential	8
TSDensity	10
TSPlot	11
TSTraj	12
VecDecomAll	13
VecDecomGrad	15
VecDecomPlot	16
VecDecomRem	18
VecDecomVec	19

Index	21
--------------	-----------

Model2String	<i>Inserts parameter values into equations</i>
--------------	------------------------------------------------

Description

Converts differential equations from string-format (or function-format) with parameters (e.g. "a*x+b) to string-format with parameter values (e.g. 2*x+3). Specifically, Model2String reads in the equations, searches for the differential equations within the function (if required), and replaces the parameters with numerical values given by the user. Returns an array of strings containing the differential equations.

Usage

```
Model2String(model, parms = "NULL", deSolve.form = FALSE,
  x.lhs.term = "dx", y.lhs.term = "dy", supress.print = FALSE,
  width.cutoff = 500)
```

Arguments

model	contains the differential equations as given to TSTraj . Can either be a string or a function used by the package deSolve (see third example).
parms	a named vector of paramters and their respective values for the deterministic equations. If inputing a function and parms is empty, Model2String will return the equation as a string.
deSolve.form	if FALSE (default) interprets model as a string containing the rhs of the equation. If TRUE, treats model as the function input into the package deSolve.
x.lhs.term	string containing the left hand side of the first equation to search for if deSolve.form is TRUE. Default is 'dx'.
y.lhs.term	string containing the left hand side of the second equation to search for if deSolve.form is TRUE. Default is 'dy'.
supress.print	if FALSE (default), suppress output. TRUE prints out equations converted to strings with parameters replaced with values.
width.cutoff	parameter width.cutoff from deparse in package:base. Determines the cutoff in bytes at which line breaking is tried. Default is 500 with possible range of 2 to 500.

Value

equations: an array with strings of the equations with substituted parameter values. If `deSolve.form` is `TRUE`, the first is the x equation, the second is the y equation.

Examples

```
# First example with the right hand side of an equation
test.eqn.x = "(alpha*x)*(1-(x/beta)) - ((delta*(x^2)*y)/(kappa + (x^2)))"
model.parms <- c(alpha=1.54, beta=10.14, delta=1, kappa=1)
equations.as.strings.x <- Model2String(test.eqn.x, parms = model.parms)

# Second example with individual strings with left and right hand sides
# Note the use deSolve.form = TRUE to remove the dx and dy
test.eqn.x = "dx = (alpha*x)*(1-(x/beta)) - ((delta*(x^2)*y)/(kappa + (x^2)))"
test.eqn.y = "dy = ((gamma*(x^2)*y)/(kappa + (x^2))) - mu*(y^2)"
model.parms <- c(alpha=1.54, beta=10.14, delta=1, kappa=1, gamma=0.476, mu=0.112509)
equations.as.strings.x <- Model2String(test.eqn.x, parms = model.parms,
  deSolve.form = TRUE, x.lhs.term = 'dx', y.lhs.term = 'dy')
equations.as.strings.y <- Model2String(test.eqn.y, parms = model.parms,
  deSolve.form = TRUE, x.lhs.term = 'dx', y.lhs.term = 'dy')

# Third example with deSolve-style function call:
model.parms <- c(alpha=1.54, beta=10.14, delta=1, kappa=1, gamma=0.476, mu=0.112509)
ModelEquations <- function(t, state, parms) {
  with(as.list(c(state, parms)), {
    dx <- (alpha*x)*(1-(x/beta)) - ((delta*(x^2)*y)/(kappa + (x^2)))
    dy <- ((gamma*(x^2)*y)/(kappa + (x^2))) - mu*(y^2)
    list(c(dx,dy))
  })
}

Model2String(ModelEquations, parms = model.parms, deSolve.form = TRUE,
  x.lhs.term = 'dx', y.lhs.term = 'dy')
```

QPContour

Contour plot of quasi-potential surfaces

Description

This function allows users to create a contour plot of quasi-potential surfaces from [QPGlobal](#)

Usage

```
QPContour(surface, dens, x.bound, y.bound, xlim = "NULL", ylim = "NULL",
  n.filled.contour = 25, n.contour.lines = 25, c.parm = 1, col.contour,
  contour.lines = TRUE, ...)
```

Arguments

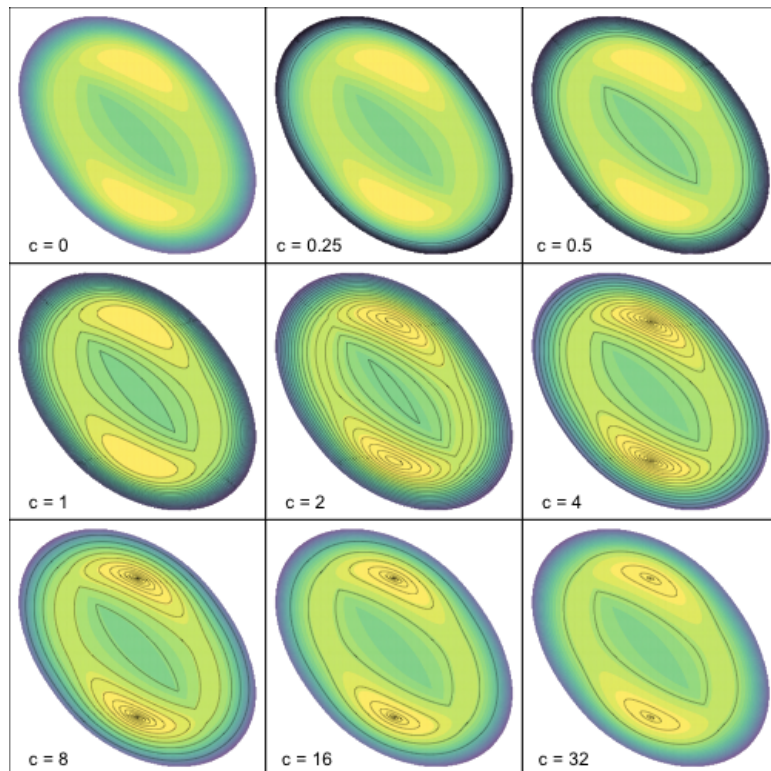
<code>surface</code>	the surface to be plotted, from QPGlobal .
<code>dens</code>	vector respectively for the number of x and y points to be plotted.
<code>x.bound</code>	a two-element vector with the minimum and maximum x values used for computing the quasi-potential.
<code>y.bound</code>	a two-element vector with the minimum and maximum y values used for computing the quasi-potential.
<code>xlim</code>	numeric vectors of length 2, giving the x coordinate range.
<code>ylim</code>	numeric vectors of length 2, giving the y coordinates range.
<code>n.filled.contour</code>	numeric value for the nubmber of breaks in the filled contour.
<code>n.contour.lines</code>	numeric value for the number of breaks in the contour lines.
<code>c.parm</code>	contour line adjustment (see details).
<code>col.contour</code>	colors to interpolate; must be a valid argument to col2rgb .
<code>contour.lines</code>	if TRUE, then contour lines plotted over filled contour; vice versa if FALSE.
<code>...</code>	passes arguments to plot .

Details

Because, in general, capturing the topological features of a surface can be subtle, we implemented a feature in [QPContour](#) to keep the filled contour region while changing the contour lines. Specifically, `filled.contour` takes the range of the surface values (ϕ), divides by the number of the specified contours (i.e., `n.filled.contour`), and creates a contour at each break, which happens to be equal across the range. But because visualizing some topology may (i) require looking between contour breaks and (ii) adding contour lines would overload the plot with lines, we use an equation to modify the distribution of contour lines. Namely, adjusting the `c` argument in the [QPContour](#) function adjusts the `c` paramter in the following equation:

$$\max_{\phi} \times \left(\frac{x}{n-1} \right)^c.$$

This allows the user to keep the same number of contour lines (i.e., specified with `n.contour.lines`), but focus them toward the troughs or peaks of the surfaces. At $c = 1$, the contour lines correspond to the `filled.contour` breaks. If $c > 1$, then the contour lines become more concentrated towards the trough. Similarly, if $c < 1$, then the contour lines are more focused at the peaks of the surface. As an example, we change `c`:



Examples

```
# First, System of equations
equationx <- "1.54*x*(1.0-(x/10.14)) - (y*x*x)/(1.0+x*x)"
equationy <- "((0.476*x*x*y)/(1+x*x)) - 0.112590*y*y"

# Second, shared parameters for each quasi-potential run
xbounds <- c(-0.5, 10.0)
ybounds <- c(-0.5, 10.0)
xstepnumber <- 150
ystepnumber <- 150

# Third, first local quasi-potential run
xinit1 <- 1.40491
yinit1 <- 2.80808
storage.eq1 <- QPotential(x.rhs = equationx, x.start = xinit1,
x.bound = xbounds, x.num.steps = xstepnumber, y.rhs = equationy,
y.start = yinit1, y.bound = ybounds, y.num.steps = ystepnumber)

# Fourth, second local quasi-potential run
xinit2 <- 4.9040
yinit2 <- 4.06187
storage.eq2 <- QPotential(x.rhs = equationx, x.start = xinit2,
x.bound = xbounds, x.num.steps = xstepnumber, y.rhs = equationy,
y.start = yinit2, y.bound = ybounds, y.num.steps = ystepnumber)
```

```
# Fifth, determine global quasi-potential
unst.x <- c(0, 4.2008)
unst.y <- c(0, 4.0039)
ex1.global <- QPGlobal(local_surfaces = list(storage.eq1, storage.eq2),
  unstable.eq.x = unst.x, unstable.eq.y = unst.y, x.bound = xbounds,
  y.bound = ybounds)

# Sixth, contour of the quasi-potential
QPContour(ex1.global, dens = c(100,100), x.bound = xbounds,
  y.bound = ybounds, c.parm = 5)
```

 QPGlobal

Finding the global quasi-potential

Description

This function allows you to find the global quasi-potential values for several local quasi-potential surfaces

Usage

```
QPGlobal(local_surfaces, unstable.eq.x, unstable.eq.y, x.bound, y.bound)
```

Arguments

<code>local_surfaces</code>	a list of local quasi-potential surfaces, each of which is stored in discretized form as a matrix.
<code>unstable.eq.x</code>	a vector of the x-coordinates of the unstable equilibria. Must be in the same order as <code>unstable.eq.y</code> .
<code>unstable.eq.y</code>	a vector of the y-coordinates of the unstable equilibria. Must be in the same order as <code>unstable.eq.x</code> .
<code>x.bound</code>	a two-element vector with the minimum and maximum x values used for computing the quasi-potential.
<code>y.bound</code>	a two-element vector with the minimum and maximum y values used for computing the quasi-potential.

Examples

```
# First, System of equations
equationx <- "1.54*x*(1.0-(x/10.14)) - (y*x*x)/(1.0+x*x)"
equationy <- "((0.476*x*x*y)/(1+x*x)) - 0.112590*y*y"

# Second, shared parameters for each quasi-potential run
xbounds <- c(-0.5, 10.0)
ybounds <- c(-0.5, 10.0)
xstepnumber <- 100
ystepnumber <- 100
```

```

# Third, first local quasi-potential run
xinit1 <- 1.40491
yinit1 <- 2.80808
storage.eq1 <- QPotential(x.rhs = equationx, x.start = xinit1,
  x.bound = xbounds, x.num.steps = xstepnumber, y.rhs = equationy,
  y.start = yinit1, y.bound = ybounds, y.num.steps = ystepnumber)

# Fourth, second local quasi-potential run
xinit2 <- 4.9040
yinit2 <- 4.06187
storage.eq2 <- QPotential(x.rhs = equationx, x.start = xinit2,
  x.bound = xbounds, x.num.steps = xstepnumber, y.rhs = equationy,
  y.start = yinit2, y.bound = ybounds, y.num.steps = ystepnumber)

# Fifth, determine global quasi-potential
unst.x <- c(0, 4.2008)
unst.y <- c(0, 4.0039)
ex1.global <- QPGlobal(local-surfaces = list(storage.eq1, storage.eq2),
  unstable.eq.x = unst.x, unstable.eq.y = unst.y, x.bound = xbounds,
  y.bound = ybounds)

```

 QPInterp

Quasi-potential interpolation

Description

This function estimates the quasi-potential value for any x- and y-values

Usage

```
QPInterp(X, Y, x.bound, y.bound, surface)
```

Arguments

X	the value of the x to interpolate.
Y	the value of the y to interpolate.
x.bound	a two-element vector with the minimum and maximum x values used for computing the quasi-potential.
y.bound	a two-element vector with the minimum and maximum y values used for computing the quasi-potential.
surface	the surface to interpolated, from QPGlobal .

Details

this function uses bilinear interpolation for estimation of any x- and y-value.

QPotential	<i>Computes the quasi-potential for a system of stochastic differential equations using the upwind ordered method.</i>
------------	------------------------------------------------------------------------------------------------------------------------

Description

Computes the quasi-potential for a system of stochastic differential equations using the upwind ordered method.

Usage

```
QPotential(x.rhs = "NULL", x.start = "NULL", x.bound = "NULL",
           x.num.steps = "NULL", y.rhs = "NULL", y.start = "NULL",
           y.bound = "NULL", y.num.steps = "NULL", filename = "NULL",
           save.to.R = TRUE, save.to.HD = FALSE, bounce = "d",
           bounce.edge = 0.01, verboseR = FALSE, verboseC = TRUE, debugC = FALSE,
           k.x = 20, k.y = 20)
```

Arguments

x.rhs	a string containing the right hand side of the equation for x.
x.start	the starting value of x, usually the x value of the current equilibrium.
x.bound	the x boundaries denoted as c(minimum, maximum).
x.num.steps	the number of steps between the minimum and maximum x value defined in x range.
y.rhs	a string containing the right hand side of the equation for y.
y.start	the starting value of y, usually the y value of the current equilibrium.
y.bound	the y boundaries denoted as c(minimum, maximum).
y.num.steps	the number of steps between the minimum and maximum y value defined in y range.
filename	string for the name of the file saved to the hard drive. If save.to.HD=TRUE and filename is left blank, output file saved as defaultname-xX.STARTyY.START.txt, where X.START and Y.START are values in x.start and y.start, respectively. Matrix stored as a tab-delimited file.
save.to.R	boolean to output the matrix of results for the upwind-ordered method to the current R session. The default is to write the matrix to the R session. save.to.R=FALSE prevents the output from being written to the R session.
save.to.HD	boolean to write the matrix of results for the upwind-ordered method to the hard drive in a file named filename. Default is FALSE.
bounce	by default, the upwind-ordered method stops when the boundaries are reached (x.bound and y.bound). The bounce parameter allows the default action (bounce = 'd'), only positive values to be tested (bounce = 'p'), or reflection near the boundaries (bounce = 'b').

bounce.edge	if bounce = 'b', then to prevent the upwind-ordered method from reaching the boundaries, temporary boundaries are created inside the boundaries defined by x.bound and y.bound. The boundary edge is bounce.edge of the total range. Default is 0.01
verboseR	NOT IMPLEMENTED: Flag (default = FALSE) for printing out information in QPotential R wrapper.
verboseC	flag (default = TRUE) for printing out useful-for-everyone information from C code implementing the upwind-ordered method (quasipotential.C).
debugC	NOT IMPLEMENTED: Flag (default = FALSE) for printing out debugging C code
k.x	integer anisotropic factor for x. See journal article. Default is 20.
k.y	integer anisotropic factor for y. See journal article. Default is 20.

Value

filetoHD if save.to.HD enabled, then saves a file in the current directory as either filename or as defaultname-xXSTARTyYSTART.txt

filetoR if save.to.R enabled, then the function QPotential returns a matrix containing the upwind-ordered results to be used for plotting. Requires a variable to catch the returned matrix, i.e. storage <- QPotential(parameters...)

Examples

```
# First, System of equations
equationx <- "1.54*x*(1.0-(x/10.14)) - (y*x*x)/(1.0+x*x)"
equationy <- "((0.476*x*x*y)/(1+x*x)) - 0.112590*y*y"

# Second, shared parameters for each quasi-potential run
xbounds <- c(-0.5, 8.0)
ybounds <- c(-0.5, 8.0)
xstepnumber <- 200
ystepnumber <- 200

# Third, a local quasi-potential run
xinit1 <- 1.40491
yinit1 <- 2.80808
storage.eq1 <- QPotential(x.rhs = equationx, x.start = xinit1,
x.bound = xbounds, x.num.steps = xstepnumber, y.rhs = equationy,
y.start = yinit1, y.bound = ybounds, y.num.steps = ystepnumber)
# Visualize the quasi-potential
QPContour(storage.eq1, dens = c(xstepnumber, ystepnumber),
x.bound = xbounds, y.bound = ybounds, c.parm = 5)
```

TSDensity	<i>Density plot from simulation of two-dimensional stochastic differential equations</i>
-----------	------------------------------------------------------------------------------------------

Description

This function creates density plots for the simulation of two-dimensional stochastic differential equations from [TSTraj](#)

Usage

```
TSDensity(mat, dim = 1, xlim = "NULL", ylim = "NULL",
  contour.levels = 15, col2d = c("blue", "yellow", "orange", "red"),
  contour.lwd = 0.5, contour.lines = TRUE, kde2d.n = 100, ...)
```

Arguments

mat	a matrix output from TSTraj .
dim	dimensions of the plot; dim = 1 plots simple density histogram or dim = 2 plots the density in state space (i.e., X and Y respectively on the abscissa and ordinate axes).
xlim	numeric vectors of length 2, giving the x coordinate range. Default = 'NULL' automatically sizes plot window.
ylim	numeric vectors of length 2, giving the y coordinate range. Default = 'NULL' automatically sizes plot window.
contour.levels	the number of contour levels for the two-dimensional plots (i.e., when dim = 2).
col2d	vector of colors to be used in the plot.
contour.lwd	line width of contour lines if contour.lines = TRUE.
contour.lines	if TRUE, then black countour lines added to the graph.
kde2d.n	number of grid points in each direction. Can be scalar or a length-2 integer vector. Passes to argument n in kde2d .
...	passes arguments to plot .

Examples

```
## Not run:
# First, the parameter values, as found in TSTraj
model.state <- c(x = 3, y = 3)
model.sigma <- 0.2
model.deltat <- 0.005
model.time <- 100

# Second, write out the deterministic skeleton of the equations to be simulated,
# as found in TSTraj
#Example 1 from article
```

```

equationx <- "1.54*x*(1.0-(x/10.14)) - (y*x*x)/(1.0 + x*x)"
equationy <- "((0.476*x*x*y)/(1 + x*x)) - 0.112590*y*y"

# Third, run it, as found in TSTraj
ModelOut <- TSTraj(y0 = model.state, time = model.time, deltat = model.deltat,
x.rhs = equationx, y.rhs = equationy, sigma = model.sigma)
# Fourth, plot it:
# in 1D
TSDensity(ModelOut, dim = 1)
# in 2D
TSDensity(ModelOut, dim = 2, kde2d.n = 20, xlab = "")

## End(Not run)

```

TSPlot

Plot simulation of two-dimensional stochastic differential equations

Description

This function plots the simulation of two-dimensional stochastic differential equations from [TSTraj](#)

Usage

```

TSPlot(mat, deltat, dim = 1, xlim = "NULL", ylim = "NULL",
x.lab = "time", dens = TRUE, lwd = 2, line.alpha = 130,
zero.axes = TRUE, ...)

```

Arguments

mat	a matrix output from TSTraj .
deltat	numeric value indicating the frequency of stochastic perturbation, as Δt , used in the function to recalculate axes if applicable.
dim	dimensions of the plot; $\text{dim} = 1$ to plot a timeseries with X and Y on the ordinate axis or $\text{dim} = 2$ to plot the trajectories in state space (i.e., X and Y respectively on the abscissa and ordinate axes).
xlim	numeric vectors of length 2, giving the x coordinate range. Default = 'NULL' automatically sizes plot window.
ylim	numeric vectors of length 2, giving the y coordinate range. Default = 'NULL' automatically sizes plot window.
x.lab	for $\text{dim} = 1$, allows user to specify the axis as "time" or "steps," with steps being $\text{time} \times \Delta t$
dens	if $\text{dens} = \text{TRUE}$, plots a horizontal one-dimensional density plot adjacent to the timeseries.
lwd	line width. Defaults to 1.
line.alpha	transparency of lines from 0–255.
zero.axes	if TRUE, then axes plotted at $X = 0$ and $Y = 0$.
...	passes arguments to plot .

Examples

```

# First, the parameter values, as found in TSTraj
model.state <- c(x = 3, y = 3)
model.sigma <- 0.2
model.deltat <- 0.05
model.time <- 100

# Second, write out the deterministic skeleton of the equations to be simulated,
# as found in TSTraj
#Example 1 from article
equationx <- "1.54*x*(1.0-(x/10.14)) - (y*x*x)/(1.0 + x*x)"
equationy <- "((0.476*x*x*y)/(1 + x*x)) - 0.112590*y*y"

# Third, run it, as found in TSTraj
ModelOut <- TSTraj(y0 = model.state, time = model.time, deltat = model.deltat,
x.rhs = equationx, y.rhs = equationy, sigma = model.sigma)
# Fourth, plot it:
# in 1D
TSPlot(ModelOut, deltat = model.deltat, dim = 1)
# in 2D
TSPlot(ModelOut, deltat = model.deltat, dim = 2)

```

TSTraj

*Simulate two-dimensional stochastic differential equations***Description**

This function allows you to simulate two-dimensional stochastic differential equations.

Usage

```
TSTraj(y0, time, deltat, x.rhs, y.rhs, parms = NA, sigma, lower.bound = NA,
upper.bound = NA)
```

Arguments

<code>y0</code>	a two-element vector of the initial conditions for the state variables. Elements must be assigned as objects (see Example below).
<code>time</code>	numeric value indicating the total time over which the simulation is to be run.
<code>deltat</code>	numeric value indicating the frequency of stochastic perturbation, as Δt .
<code>x.rhs</code>	a string containing the right hand side of the equation for x.
<code>y.rhs</code>	a string containing the right hand side of the equation for y.
<code>parms</code>	n-element vector of objects representing unvalued parameters in the equation. If parameter values are values in the equation, then default is <code>parms = NA</code> .
<code>sigma</code>	numeric value specifying the noise intensity.
<code>lower.bound</code>	numeric value specifying a lower bound in the simulation.
<code>upper.bound</code>	numeric value specifying an upper bound in the simulation.

Value

returns a matrix with three columns (timestep, x values, and y values) with a length of time/deltat (2×10^4 in the examples below).

Examples

```
# First, the parameter values
model.state <- c(x = 3, y = 3)
model.sigma <- 0.2
model.deltat <- 0.1
model.time <- 100

# Second, write out the deterministic skeleton of the equations to be simulated
equationx <- "1.54*x*(1.0-(x/10.14)) - (y*x*x)/(1.0 + x*x)"
equationy <- "((0.476*x*x*y)/(1 + x*x)) - 0.112590*y*y"

# Third, Run it
ModelOut <- TSTraj(y0 = model.state, time = model.time, deltat = model.deltat,
x.rhs = equationx, y.rhs = equationy, sigma = model.sigma)

# Can also input x.rhs and y.rhs as strings that contain parameter names
# and include parms with names and values of parameters
model.state <- c(x = 1, y = 2)
model.parms <- c(alpha = 1.54, beta = 10.14, delta = 1, kappa = 1, gamma = 0.476, mu = 0.112509)
model.sigma <- 0.2
model.time <- 100
model.deltat <- 0.1

test.eqn.x = "(alpha*x)*(1-(x/beta)) - ((delta*(x^2)*y)/(kappa + (x^2)))"
test.eqn.y = "((gamma*(x^2)*y)/(kappa + (x^2))) - mu*(y^2)"

ModelOut.parms <- TSTraj(y0 = model.state, time = model.time, deltat = model.deltat,
x.rhs = test.eqn.x, y.rhs = test.eqn.y, parms = model.parms, sigma = model.sigma)
```

VecDecomAll

Vector decomposition and remainder fields

Description

This function calculates the vector, gradient, and remainder fields.

Usage

```
VecDecomAll(surface, x.rhs, y.rhs, x.bound, y.bound)
```

Arguments

surface	matrix output from QPGlobal or QPotential .
x.rhs	a string containing the right hand side of the equation for x.
y.rhs	a string containing the right hand side of the equation for y.
x.bound	the x boundaries denoted at c(minimum, maximum).
y.bound	the y boundaries denoted at c(minimum, maximum).

Value

returns an array of all three vector fields: the deterministic skeleton, the negative gradient of the quasi-potential, and the remainder. The array has three dimensions with the respective lengths of `xstepnumber`, `ystepnumber`, and 6. The six are the x- and y-values for each of the three vector fields, as x-deterministic skeleton, y-deterministic skeleton, x-negative gradient of the quasi-potential, y-negative gradient of the quasi-potential, x-remainder, and y-remainder.

Examples

```
# First, the system of equations
equationx <- "1.54*x*(1.0-(x/10.14)) - (y*x*x)/(1.0+x*x)"
equationy <- "((0.476*x*x*y)/(1+x*x)) - 0.112590*y*y"

# Second, shared parameters for each quasi-potential run
xbounds <- c(-0.5, 10.0)
ybounds <- c(-0.5, 10.0)
xstepnumber <- 100
ystepnumber <- 100

# Third, first local quasi-potential run
xinit1 <- 1.40491
yinit1 <- 2.80808
storage.eq1 <- QPotential(x.rhs = equationx, x.start = xinit1,
x.bound = xbounds, x.num.steps = xstepnumber, y.rhs = equationy,
y.start = yinit1, y.bound = ybounds, y.num.steps = ystepnumber)

# Fourth, second local quasi-potential run
xinit2 <- 4.9040
yinit2 <- 4.06187
storage.eq2 <- QPotential(x.rhs = equationx, x.start = xinit2,
x.bound = xbounds, x.num.steps = xstepnumber, y.rhs = equationy,
y.start = yinit2, y.bound = ybounds, y.num.steps = ystepnumber)

# Fifth, determine global quasi-potential
unst.x <- c(0, 4.2008)
unst.y <- c(0, 4.0039)
ex1.global <- QPGlobal(local_surfaces = list(storage.eq1, storage.eq2),
unstable.eq.x = unst.x, unstable.eq.y = unst.y, x.bound = xbounds,
y.bound = ybounds)

# Sixth, decompose the global quasi-potential into the
# deterministic skeleton, gradient, and remainder vector fields
```

```
VDA11 <- VecDecomAll(surface = ex1.global, x.rhs = equationx, y.rhs = equationy,
x.bound = xbounds, y.bound = ybounds)
```

VecDecomGrad *Vector decomposition and remainder fields*

Description

This function calculates the gradient field.

Usage

```
VecDecomGrad(surface)
```

Arguments

surface matrix output from [QPGlobal](#) or [QPotential](#).

Value

returns an array of the gradient vector field. The array has three dimensions with the respective lengths of the columns of the surface, the rows of the surface, and the number of variables (always 2). The two variables are the x-negative gradient of the quasi-potential surface and the y-negative gradient of the quasi-potential surface.

Examples

```
# First, the system of equations
equationx <- "1.54*x*(1.0-(x/10.14)) - (y*x*x)/(1.0+x*x)"
equationy <- "((0.476*x*x*y)/(1+x*x)) - 0.112590*y*y"

# Second, shared parameters for each quasi-potential run
xbounds <- c(-0.5, 10.0)
ybounds <- c(-0.5, 10.0)
xstepnumber <- 100
ystepnumber <- 100

# Third, first local quasi-potential run
xinit1 <- 1.40491
yinit1 <- 2.80808
storage.eq1 <- QPotential(x.rhs = equationx, x.start = xinit1,
x.bound = xbounds, x.num.steps = xstepnumber, y.rhs = equationy,
y.start = yinit1, y.bound = ybounds, y.num.steps = ystepnumber)

# Fourth, second local quasi-potential run
xinit2 <- 4.9040
yinit2 <- 4.06187
storage.eq2 <- QPotential(x.rhs = equationx, x.start = xinit2,
x.bound = xbounds, x.num.steps = xstepnumber, y.rhs = equationy,
y.start = yinit2, y.bound = ybounds, y.num.steps = ystepnumber)
```

```
# Fifth, determine global quasi-potential
unst.x <- c(0, 4.2008)
unst.y <- c(0, 4.0039)
ex1.global <- QPGlobal(local_surfaces = list(storage.eq1, storage.eq2),
  unstable.eq.x = unst.x, unstable.eq.y = unst.y, x.bound = xbounds,
  y.bound = ybounds)

# Sixth, creat the gradient vector field
VDG <- VecDecomGrad(surface = ex1.global)
```

VecDecomPlot

Plotting function for vector decomposition and remainder fields

Description

This function plots various vector fields

Usage

```
VecDecomPlot(x.field, y.field, dens, x.bound, y.bound, xlim = "NULL",
  ylim = "NULL", arrow.type = "equal", tail.length = 0.25,
  head.length = 0.25, ...)
```

Arguments

x.field	a two-dimensional array containing the x-values for the vector field, generated from VecDecomAll , VecDecomVec , VecDecomGrad , or VecDecomRem .
y.field	a two-dimensional array containing the y-values for the vector field, generated from VecDecomAll , VecDecomVec , VecDecomGrad , or VecDecomRem .
dens	two-element vector respectively specifying the number of respective arrows in the x and y directions.
x.bound	the x boundaries denoted at c(minimum, maximum) for the quasi-potential simulation.
y.bound	the y boundaries denoted at c(minimum, maximum) for the quasi-potential simulation.
xlim	numeric vectors of length 2, giving the x coordinate range.
ylim	numeric vectors of length 2, giving the y coordinate range.
arrow.type	sets the type of line segments plotted. If set to "proportional" the length of the line segments reflects the magnitude of the derivative. If set to "equal" the line segments take equal lengths, simply reflecting the gradient of the derivative(s). Defaults to "equal".
tail.length	multiplies the current length of the tail (both proportional and equal arrow.types) by the specified factor. The argument defaults to 1, which is length of the longest vector within the domain boundaries (i.e., the entire field).
head.length	length of the edges of the arrow head (in inches).
...	passes arguments to both plot .

Details

If `arrow.type = "proportional"`, a common warning, passed from `arrows`, will appear: "The direction of a zero-length arrow is indeterminate, and hence so is the direction of the arrowheads. To allow for rounding error, arrowheads are omitted (with a warning) on any arrow of length less than 1/1000 inch." Either increase `tail.length` or increase the plot window to avoid this warning.

Examples

```
# First, system of equations
equationx <- "1.54*x*(1.0-(x/10.14)) - (y*x*x)/(1.0+x*x)"
equationy <- "((0.476*x*x*y)/(1+x*x)) - 0.112590*y*y"

# Second, shared parameters for each quasi-potential run
xbounds <- c(-0.5, 10.0)
ybounds <- c(-0.5, 10.0)
xstepnumber <- 100
ystepnumber <- 100

# Third, first local quasi-potential run
xinit1 <- 1.40491
yinit1 <- 2.80808
storage.eq1 <- QPotential(x.rhs = equationx, x.start = xinit1,
x.bound = xbounds, x.num.steps = xstepnumber, y.rhs = equationy,
y.start = yinit1, y.bound = ybounds, y.num.steps = ystepnumber)

# Fourth, second local quasi-potential run
xinit2 <- 4.9040
yinit2 <- 4.06187
storage.eq2 <- QPotential(x.rhs = equationx, x.start = xinit2,
x.bound = xbounds, x.num.steps = xstepnumber, y.rhs = equationy,
y.start = yinit2, y.bound = ybounds, y.num.steps = ystepnumber)

# Fifth, determine global quasi-potential
unst.x <- c(0, 4.2008)
unst.y <- c(0, 4.0039)
ex1.global <- QPGlobal(local-surfaces = list(storage.eq1, storage.eq2),
unstable.eq.x = unst.x, unstable.eq.y = unst.y, x.bound = xbounds,
y.bound = ybounds)

# Sixth, decompose the global quasi-potential into the
# deterministic skeleton, gradient, and remainder vector fields
VDAll <- VecDecomAll(surface = ex1.global, x.rhs = equationx, y.rhs = equationy,
x.bound = xbounds, y.bound = ybounds)

# Seventh, plot all three vector fields
# The deterministic skeleton vector field
VecDecomPlot(x.field = VDAll[,1], y.field = VDAll[,2], dens = c(25,25),
x.bound = xbounds, y.bound = ybounds, tail.length = 0.25, head.length = 0.05)
# The gradient vector field
VecDecomPlot(x.field = VDAll[,3], y.field = VDAll[,4], dens = c(25,25),
x.bound = xbounds, y.bound = ybounds, tail.length = 0.15, head.length = 0.05)
# The remainder vector field
```

```
VecDecomPlot(x.field = VDA11[, ,5], y.field = VDA11[, ,6], dens = c(25,25),
x.bound = xbounds, y.bound = ybounds, tail.length = 0.15, head.length = 0.05)
```

VecDecomRem

Vector decomposition and remainder fields

Description

This function calculates the remainder field.

Usage

```
VecDecomRem(surface, x.rhs, y.rhs, x.bound, y.bound)
```

Arguments

surface	matrix output from QPGlobal or QPotential .
x.rhs	a string containing the right hand side of the equation for x.
y.rhs	a string containing the right hand side of the equation for y.
x.bound	the x boundaries denoted at c(minimum, maximum).
y.bound	the y boundaries denoted at c(minimum, maximum).

Value

returns an array of the remainder vector field. The array has three dimensions with the respective lengths of the columns of the surface, the rows of the surface, and the number of variables (always 2). The two variables are the x-remainder and y-remainder.

Examples

```
# First, the system of equations
equationx <- "1.54*x*(1.0-(x/10.14)) - (y*x*x)/(1.0+x*x)"
equationy <- "((0.476*x*x*y)/(1+x*x)) - 0.112590*y*y"

# Second, shared parameters for each quasi-potential run
xbounds <- c(-0.5, 10.0)
ybounds <- c(-0.5, 10.0)
xstepnumber <- 150
ystepnumber <- 150

# Third, first local quasi-potential run
xinit1 <- 1.40491
yinit1 <- 2.80808
storage.eq1 <- QPotential(x.rhs = equationx, x.start = xinit1,
x.bound = xbounds, x.num.steps = xstepnumber, y.rhs = equationy,
y.start = yinit1, y.bound = ybounds, y.num.steps = ystepnumber)

# Fourth, second local quasi-potential run
```

```

xinit2 <- 4.9040
yinit2 <- 4.06187
storage.eq2 <- QPotential(x.rhs = equationx, x.start = xinit2,
x.bound = xbounds, x.num.steps = xstepnumber, y.rhs = equationy,
y.start = yinit2, y.bound = ybounds, y.num.steps = ystepnumber)

# Fifth, determine global quasi-potential
unst.x <- c(0, 4.2008)
unst.y <- c(0, 4.0039)
ex1.global <- QPGlobal(local-surfaces = list(storage.eq1, storage.eq2),
unstable.eq.x = unst.x, unstable.eq.y = unst.y, x.bound = xbounds,
y.bound = ybounds)

# Sixth, create remainder field
VDR <- VecDecomRem(surface = ex1.global, x.rhs = equationx, y.rhs = equationy,
x.bound = xbounds, y.bound = ybounds)

```

VecDecomVec

Vector decomposition and remainder fields

Description

This function calculates the vector field.

Usage

```
VecDecomVec(x.num.steps, y.num.steps, x.rhs, y.rhs, x.bound, y.bound)
```

Arguments

x.num.steps	the number of steps between the minimum and maximum x value defined in x range.
y.num.steps	the number of steps between the minimum and maximum y value defined in y range.
x.rhs	a string containing the right hand side of the equation for x.
y.rhs	a string containing the right hand side of the equation for y.
x.bound	the x boundaries denoted at c(minimum, maximum).
y.bound	the y boundaries denoted at c(minimum, maximum).

Value

returns an array of the deterministic skeleton vector field. The array has three dimensions with the respective lengths of x.num.steps, y.num.steps, and the number of variables (always 2). The two variables are the x-deterministic skeleton and the y-deterministic skeleton.

Examples

```
# First, the system of equations
equationx <- "1.54*x*(1.0-(x/10.14)) - (y*x*x)/(1.0+x*x)"
equationy <- "(0.476*x*x*y)/(1+x*x) - 0.112590*y*y"

# Second, shared parameters for each quasi-potential run
xbounds <- c(-0.5, 20.0)
ybounds <- c(-0.5, 20.0)
xstepnumber <- 1000
ystepnumber <- 1000

# Third, create the deterministic skeleton vector field
VDV <- VecDecomVec(x.num.steps = xstepnumber, y.num.steps = ystepnumber, x.rhs = equationx,
y.rhs = equationy, x.bound = xbounds, y.bound = ybounds)
```

Index

- *Topic **Global**
 - QPGlobal, 6
- *Topic **Stochastic**
 - TSTraj, 12
- *Topic **decomposition,**
 - VecDecomRem, 18
- *Topic **decomposition,**
 - VecDecomGrad, 15
- *Topic **detrministic**
 - VecDecomPlot, 16
- *Topic **field**
 - VecDecomGrad, 15
 - VecDecomPlot, 16
 - VecDecomRem, 18
- *Topic **gradient**
 - VecDecomGrad, 15
 - VecDecomPlot, 16
- *Topic **interpolation**
 - QPInterp, 7
- *Topic **plot,**
 - VecDecomPlot, 16
- *Topic **plot**
 - TSDensity, 10
 - TSPlot, 11
 - VecDecomPlot, 16
- *Topic **quasi-potential**
 - QPGlobal, 6
- *Topic **remainder**
 - VecDecomPlot, 16
 - VecDecomRem, 18
- *Topic **simulations**
 - TSDensity, 10
 - TSPlot, 11
- *Topic **simulation**
 - TSTraj, 12
- *Topic **skeleton**
 - VecDecomPlot, 16
- *Topic **stochastic**
 - TSDensity, 10
 - TSPlot, 11
- *Topic **vector**
 - VecDecomGrad, 15
 - VecDecomPlot, 16
 - VecDecomRem, 18
- arrows, 17
- col2rgb, 4
- filled.contour, 4
- kde2d, 10
- Model2String, 2
- plot, 4, 10, 11, 16
- QPContour, 3, 4
- QPGlobal, 3, 4, 6, 7, 14, 15, 18
- QPInterp, 7
- QPotential, 8, 14, 15, 18
- TSDensity, 10
- TSPlot, 11
- TSTraj, 2, 10, 11, 12
- VecDecomAll, 13, 16
- VecDecomGrad, 15, 16
- VecDecomPlot, 16
- VecDecomRem, 16, 18
- VecDecomVec, 16, 19