Package 'KGode'

June 23, 2020

Title Kernel Based Gradient Matching for Parameter Inference in Ordinary Differential Equations

Version 1.0.3

Author Mu Niu [aut, cre]

Maintainer Mu Niu <mu.niu@glasgow.ac.uk>

Description The kernel ridge regression and the gradient matching algorithm proposed in Niu et al. (2016) <http://jmlr.org/proceedings/papers/v48/niu16.html> and the warping algorithm proposed in Niu et al. (2017) <DOI:10.1007/s00180-017-0753-z> are implemented for parameter inference in differential equations. Four schemes are provided for improving parameter estimation in odes by using the odes regularisation and warping.

Depends R (>= 3.2.0)

License GPL (>= 2)

Imports R6,pracma,pspline,mvtnorm,graphics

Encoding UTF-8

LazyData true

RoxygenNote 7.1.0

NeedsCompilation no

Repository CRAN

Date/Publication 2020-06-23 05:31:07 UTC

R topics documented:

`bootstrap` *The 'bootstrap' function*

Description

This function is used to perform bootstrap procedure to estimate parameter uncertainty.

Usage

```
bootstrap(kkk, y_no, ktype, K, ode_par, intp_data, www = NULL)
```

Arguments

`kkk`	ode class object.
`y_no`	matrix(of size n_s*n_o) containing noisy observations. The row(of length n_s) represent the ode states and the column(of length n_o) represents the time points.
`ktype`	character containing kernel type. User can choose 'rbf' or 'mlp' kernel.
`K`	the number of bootstrap replicates to collect.
`ode_par`	a vector of ode parameters estimated using gradient matching.
`intp_data`	a list of interpolations produced by gradient matching for each ode state.
`www`	an optional warping object (if warping has been performed using warpfun).

Details

Arguments of the 'bootstrap' function are 'ode' class, noisy observation, kernel type, the set of parameters that have been estimated before using gradient matching, a list of interpolations for each of the ode state from gradient matching, and the warping object (if warping has been performed). It returns a vector of the median absolute standard deviations for each ode state, computed from the bootstrap replicates.

Value

return a vector of the median absolute deviation (MAD) for each ode state.

Author(s)

Mu Niu <mu.niu@glasgow.ac.uk>

Examples

```
## Not run:
require(mvtnorm)
noise = 0.1  ## set the variance of noise
SEED = 19537
set.seed(SEED)
## Define ode function, we use lotka-volterra model in this example.
## we have two ode states x[1], x[2] and four ode parameters alpha, beta, gamma and delta.
LV_fun = function(t,x,par_ode){
  alpha=par_ode[1]
  beta=par_ode[2]
  gamma=par_ode[3]
  delta=par_ode[4]
  as.matrix( c( alpha*x[1]-beta*x[2]*x[1] , -gamma*x[2]+delta*x[1]*x[2] ) )
}
## Define the gradient of ode function against ode parameters
## df/dalpha,  df/dbeta, df/dgamma, df/ddelta where f is the differential equation.
LV_grlNODE= function(par,grad_ode,y_p,z_p) {
alpha = par[1]; beta= par[2]; gamma = par[3]; delta = par[4]
dres= c(0)
dres[1] = sum( -2*( z_p[1,]-grad_ode[1,])*y_p[1,]*alpha )
dres[2] = sum( 2*( z_p[1,]-grad_ode[1,])*y_p[2,]*y_p[1,]*beta)
dres[3] = sum( 2*( z_p[2,]-grad_ode[2,])*gamma*y_p[2,] )
dres[4] = sum( -2*( z_p[2,]-grad_ode[2,])*y_p[2,]*y_p[1,]*delta)
dres
}

## create a ode class object
kkk0 = ode$new(2,fun=LV_fun,grfun=LV_grlNODE)
## set the initial values for each state at time zero.
xinit = as.matrix(c(0.5,1))
## set the time interval for the ode numerical solver.
tinterv = c(0,6)
## solve the ode numerically using predefined ode parameters. alpha=1, beta=1, gamma=4, delta=1.
kkk0$solve_ode(c(1,1,4,1),xinit,tinterv)

## Add noise to the numerical solution of the ode model and use it as the noisy observation.
n_o = max( dim( kkk0$y_ode) )
t_no = kkk0$t
y_no =  t(kkk0$y_ode) + rmvnorm(n_o,c(0,0),noise*diag(2))

## Create a ode class object by using the simulation data we created from the ode numerical solver.
## If users have experiment data, they can replace the simulation data with the experiment data.
## Set initial value of ode parameters.
init_par = rep(c(0.1),4)
init_yode = t(y_no)
init_t = t_no
kkk = ode$new(1,fun=LV_fun,grfun=LV_grlNODE,t=init_t,ode_par= init_par, y_ode=init_yode )

## The following examples with CPU or elapsed time > 10s
```

```
##Use function 'rkg' to estimate the ode parameters. The standard gradient matching method is coded
##in the the 'rkg' function. The parameter estimations are stored in the returned vector of 'rkg'.
## Choose a kernel type for 'rkhs' interpolation. Two options are provided 'rbf' and 'mlp'.
ktype ='rbf'
rkgres = rkg(kkk,y_no,ktype)
## show the results of ode parameter estimation using the standard gradient matching
kkk$ode_par

## Perform bootstrap procedure to estimate the median absolute deviations of ode parameters
# here we get the resulting interpolation from gradient matching using 'rkg' for each ode state
bbb = rkgres$bbb
nst = length(bbb)
intp_data = list()
for( i in 1:nst) {
    intp_data[[i]] = bbb[[i]]$predictT(bbb[[i]]$t)$pred
}
K = 12 # the number of bootstrap replicates
mads = bootstrap(kkk, y_no, ktype, K, ode_par, intp_data)

## show the results of ode parameter estimation and its uncertainty
## using the standard gradient matching
ode_par
mads

############## gradient matching + ODE regularisation
crtype='i'
lam=c(10,1,1e-1,1e-2,1e-4)
lamil1 = crossv(lam,kkk,bbb,crtype,y_no)
lambdai1=lamil1[[1]]
res = third(lambdai1,kkk,bbb,crtype)
oppar = res$oppar

### do bootstrap here for gradient matching + ODE regularisation
ode_par = oppar
K = 12
intp_data = list()
for( i in 1:nst) {
    intp_data[[i]] = res$rk3$rk[[i]]$predictT(bbb[[i]]$t)$pred
}
mads = bootstrap(kkk, y_no, ktype, K, ode_par, intp_data)
ode_par
mads

############## gradient matching + ODE regularisation + warping
###### warp state
peod = c(6,5.3) #8#9.7     ## the guessing period
eps= 1          ## the standard deviation of period
fixlens=warpInitLen(peod,eps,rkgres)
kkkrkg = kkk$clone()
www = warpfun(kkkrkg,bbb,peod,eps,fixlens,y_no,kkkrkg$t)

### do bootstrap here for gradient matching + ODE regularisation + warping
nst = length(bbb)
```

```
K = 12
ode_par = www$wkkk$ode_par
intp_data = list()
for( i in 1:nst) {
    intp_data[[i]] = www$bbbw[[i]]$predictT(www$wtime[i, ])$pred
}
mads = bootstrap(kkk, y_no, ktype, K, ode_par, intp_data,www)
ode_par
mads


## End(Not run)
```

---

crossv *The 'crossv' function*

Description

This function is used to estimate the weighting parameter for ode regularisation using cross validation.

Usage

```
crossv(lam, kkk, bbb, crtype, y_no, woption, resmtest, dtilda, fold)
```

Arguments

lam	vector containing different choices of the weighting parameter of ode regularisation.
kkk	'ode' class object containing all information about the odes.
bbb	list of 'rkhs' class object containing the interpolation for all ode states.
crtype	character containing the optimisation scheme type. User can choose 'i' or '3'. 'i' is for fast iterative scheme and '3' for optimising the ode parameters and interpolation coefficients simultaneously.
y_no	matrix(of size n_s*n_o) containing noisy observations. The row(of length n_s) represent the ode states and the column(of length n_o) represents the time points.
woption	character containing the indication of using warping. If the warping scheme is done before using the ode regularisation, user can choose 'w' otherwise just leave this option empty.
resmtest	vector(of length n_o) containing the warped time points. This variable is only used if user want to combine warping and the ode regularisation.
dtilda	vector(of length n_o) containing the gradient of warping function. This variable is only used if user want to combine warping and the ode regularisation.
fold	scalar indicating the folds of cross validation.

Details

Arguments of the 'crossv' function are list of weighting parameter for ode regularisation, 'ode' class objects, 'rkhs' class objects, noisy observation, type of regularisation scheme, option of warping and the gradient of warping function. It return the interpolation for each of the ode states. The ode parameters are estimated using gradient matching, and the results are stored in the 'ode' class as the ode_par attribute.

Value

return list containing :

- lam - scalar containing the optimised weighting parameter.
- ress -vector containing the cross validation error for all choices of weighting parameter.

Author(s)

Mu Niu <mu.niu@glasgow.ac.uk>

Examples

```
## Not run:
require(mvtnorm)
noise = 0.1
SEED = 19537
set.seed(SEED)
## Define ode function, we use lotka-volterra model in this example.
## we have two ode states x[1], x[2] and four ode parameters alpha, beta, gamma and delta.
LV_fun = function(t,x,par_ode){
  alpha=par_ode[1]
  beta=par_ode[2]
  gamma=par_ode[3]
  delta=par_ode[4]
  as.matrix( c( alpha*x[1]-beta*x[2]*x[1] , -gamma*x[2]+delta*x[1]*x[2] ) )
}
## Define the gradient of ode function against ode parameters
## df/dalpha,  df/dbeta, df/dgamma, df/ddelta where f is the differential equation.
LV_grlNODE= function(par,grad_ode,y_p,z_p) {
alpha = par[1]; beta= par[2]; gamma = par[3]; delta = par[4]
dres= c(0)
dres[1] = sum( -2*( z_p[1,]-grad_ode[1,])*y_p[1,]*alpha )
dres[2] = sum( 2*( z_p[1,]-grad_ode[1,])*y_p[2,]*y_p[1,]*beta)
dres[3] = sum( 2*( z_p[2,]-grad_ode[2,])*gamma*y_p[2,] )
dres[4] = sum( -2*( z_p[2,]-grad_ode[2,])*y_p[2,]*y_p[1,]*delta)
dres
}

## create a ode class object
kkk0 = ode$new(2,fun=LV_fun,grfun=LV_grlNODE)
## set the initial values for each state at time zero.
xinit = as.matrix(c(0.5,1))
```

```
## set the time interval for the ode numerical solver.
tinterv = c(0,6)
## solve the ode numerically using predefined ode parameters. alpha=1, beta=1, gamma=4, delta=1.
kkk0$solve_ode(c(1,1,4,1),xinit,tinterv)

## Add noise to the numerical solution of the ode model and use it as the noisy observation.
n_o = max( dim( kkk0$y_ode) )
t_no = kkk0$t
y_no =  t(kkk0$y_ode) + rmvnorm(n_o,c(0,0),noise*diag(2))

## create a ode class object by using the simulation data we created from the Ode numerical solver.
## If users have experiment data, they can replace the simulation data with the experiment data.
## set initial value of Ode parameters.
init_par = rep(c(0.1),4)
init_yode = t(y_no)
init_t = t_no
kkk = ode$new(1,fun=LV_fun,grfun=LV_grlNODE,t=init_t,ode_par= init_par, y_ode=init_yode )

## The following examples with CPU or elapsed time > 10s

## Use function 'rkg' to estimate the Ode parameters.
ktype ='rbf'
rkgres = rkg(kkk,y_no,ktype)
bbb = rkgres$bbb

############## gradient matching + third step
crtype='i'
## using cross validation to estimate the weighting parameters of the ode regularisation
lam=c(1e-4,1e-5)
lamil1 = crossv(lam,kkk,bbb,crtype,y_no)
lambdai1=lamil1[[1]]

## End(Not run)
```

`diagnostic` *The 'diagnostic' function*

Description

This function is used to perform diagnostic procedure to compute the residual and make diagnostic plots.

Usage

```
diagnostic(infer_list, index, type, qq_plot)
```

Arguments

`infer_list`	a list of inference results including ode objects and inference objects.
`index`	the index of the ode states which the user want to do the diagnostic analysis.

type	character containing the type of inference methods. User can choose 'rkg', 'third', or 'warp'.
qq_plot	boolean variable, enable or disable the plotting function.

Details

Arguments of the 'diagnostic' function are inference list , inference type, a list of interpolations for each of the ode state from gradient matching, and . It returns a vector of the median absolute standard deviations for each ode state.

Value

return list containing :

- residual - vector containing residual.
- interp - vector containing interpolation.

Author(s)

Mu Niu <mu.niu@glasgow.ac.uk>

Examples

```
## Not run:
require(mvtnorm)
set.seed(SEED);  SEED = 19537
FN_fun <- function(t, x, par_ode) {
a = par_ode[1]
b = par_ode[2]
c = par_ode[3]
as.matrix(c(c*(x[1]-x[1]^3/3 + x[2]),-1/c*(x[1]-a+b*x[2])))
}

solveOde = ode$new(sample=2,fun=FN_fun)
xinit = as.matrix(c(-1,-1))
tinterv = c(0,10)
solveOde$solve_ode(par_ode=c(0.2,0.2,3),xinit,tinterv)

n_o = max(dim(solveOde$y_ode))
noise = 0.01
y_no = t(solveOde$y_ode)+rmvnorm(n_o,c(0,0),noise*diag(2))
t_no = solveOde$t

odem = ode$new(fun=FN_fun,grfun=NULL,t=t_no,ode_par=rep(c(0.1),3),y_ode=t(y_no))
ktype = 'rbf'
rkgres = rkg(odem,y_no,ktype)
rkgdiag = diagnostic( rkgres,1,'rkg',qq_plot=FALSE )

## End(Not run)
```

Kernel *The 'Kernel' class object*

Description

This a abstract class provide the kernel function and the 1st order derivative of rbf kernel function.

Format

R6Class object.

Value

an R6Class object which can be used for the rkhs interpolation.

Methods

`kern(t1,t2)` This method is used to calculate the kernel function given two one dimensional real inputs.

`dkd_kpar(t1,t2)` This method is used to calculate the gradient of kernel function against the kernel hyper parameters given two one dimensional real inputs.

`dkdt(t1,t2)` This method is used to calculate the 1st order derivative of kernel function given two one dimensional real inputs.

Public fields

`k_par` vector(of length n_hy) containing the hyper-parameter of kernel. n_hy is the length of kernel hyper parameters.

Methods

Public methods:

- `Kernel$new()`
- `Kernel$greet()`
- `Kernel$kern()`
- `Kernel$dkd_kpar()`
- `Kernel$dkdt()`
- `Kernel$clone()`

Method `new()`:

Usage:

```
Kernel$new(k_par = NULL)
```

Method `greet()`:

Usage:

```
Kernel$greet()
```

Method `kern()`**:**

Usage:

```
Kernel$kern(t1, t2)
```

Method `dkd_kpar()`**:**

Usage:

```
Kernel$dkd_kpar(t1, t2)
```

Method `dkdt()`**:**

Usage:

```
Kernel$dkdt(t1, t2)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Kernel$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Mu Niu, <mu.niu@glasgow.ac.uk>

`MLP` *The 'MLP' class object*

Description

This a R6 class. It inherits from 'kernel' class. It provides the mlp kernel function and the 1st order derivative of mlp kernel function.

Format

`R6Class` object.

Value

an `R6Class` object which can be used for the rkhs interpolation.

Super class

`KGode::Kernel` -> `MLP`

Methods

Public methods:

- MLP$greet()
- MLP$set_k_par()
- MLP$kern()
- MLP$dkd_kpar()
- MLP$dkdt()
- MLP$clone()

Method `greet()`**:**

Usage:

```
MLP$greet()
```

Method `set_k_par()`**:**

Usage:

```
MLP$set_k_par(val)
```

Method `kern()`**:**

Usage:

```
MLP$kern(t1, t2)
```

Method `dkd_kpar()`**:**

Usage:

```
MLP$dkd_kpar(t1, t2)
```

Method `dkdt()`**:**

Usage:

```
MLP$dkdt(t1, t2)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MLP$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Mu Niu, <mu.niu@glasgow.ac.uk>

ode *The 'ode' class object*

Description

This class provide all information about odes and methods for numerically solving odes.

Format

`R6Class` object.

Value

an `R6Class` object which can be used for gradient matching.

Methods

`solve_ode(par_ode,xinit,tinterv)` This method is used to solve ode numerically.

`optim_par(par,y_p,z_p)` This method is used to estimate ode parameters by standard gradient matching.

`lossNODE(par,y_p,z_p)` This method is used to calculate the mismatching between gradient of interpolation and gradient from ode.

Public fields

`ode_par` vector(of length n_p) containing ode parameters. n_p is the number of ode parameters.

`ode_fun` function containing the ode function.

`t` vector(of length n_o) containing time points of observations. n_o is the length of time points.

Methods

Public methods:

- `ode$new()`
- `ode$greet()`
- `ode$solve_ode()`
- `ode$rmsfun()`
- `ode$gradient()`
- `ode$lossNODE()`
- `ode$grlNODE()`
- `ode$loss32NODE()`
- `ode$grl32NODE()`
- `ode$optim_par()`
- `ode$clone()`

Method `new()`:

Usage:

```
ode$new(
  sample = NULL,
  fun = NULL,
  grfun = NULL,
  t = NULL,
  ode_par = NULL,
  y_ode = NULL
)
```

Method `greet()`**:**

Usage:

```
ode$greet()
```

Method `solve_ode()`**:**

Usage:

```
ode$solve_ode(par_ode, xinit, tinterv)
```

Method `rmsfun()`**:**

Usage:

```
ode$rmsfun(par_ode, state, M1, true_par)
```

Method `gradient()`**:**

Usage:

```
ode$gradient(y_p, par_ode)
```

Method `lossNODE()`**:**

Usage:

```
ode$lossNODE(par, y_p, z_p)
```

Method `grlNODE()`**:**

Usage:

```
ode$grlNODE(par, y_p, z_p)
```

Method `loss32NODE()`**:**

Usage:

```
ode$loss32NODE(par, y_p, z_p)
```

Method `grl32NODE()`**:**

Usage:

```
ode$grl32NODE(par, y_p, z_p)
```

Method `optim_par()`**:**

Usage:

```
ode$optim_par(par, y_p, z_p)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ode$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Mu Niu, < mu.niu@glasgow.ac.uk>

Examples

```
noise = 0.1  ## set the variance of noise
SEED = 19537
set.seed(SEED)
## Define ode function, we use lotka-volterra model in this example.
## we have two ode states x[1], x[2] and four ode parameters alpha, beta, gamma and delta.
LV_fun = function(t,x,par_ode){
  alpha=par_ode[1]
  beta=par_ode[2]
  gamma=par_ode[3]
  delta=par_ode[4]
  as.matrix( c( alpha*x[1]-beta*x[2]*x[1] , -gamma*x[2]+delta*x[1]*x[2] ) )
}
## Define the gradient of ode function against ode parameters
## df/dalpha,  df/dbeta, df/dgamma, df/ddelta where f is the differential equation.
LV_grlNODE= function(par,grad_ode,y_p,z_p) {
alpha = par[1]; beta= par[2]; gamma = par[3]; delta = par[4]
dres= c(0)
dres[1] = sum( -2*( z_p[1,]-grad_ode[1,])*y_p[1,]*alpha )
dres[2] = sum( 2*( z_p[1,]-grad_ode[1,])*y_p[2,]*y_p[1,]*beta)
dres[3] = sum( 2*( z_p[2,]-grad_ode[2,])*gamma*y_p[2,] )
dres[4] = sum( -2*( z_p[2,]-grad_ode[2,])*y_p[2,]*y_p[1,]*delta)
dres
}

## create a ode class object
kkk0 = ode$new(2,fun=LV_fun,grfun=LV_grlNODE)
## set the initial values for each state at time zero.
xinit = as.matrix(c(0.5,1))
## set the time interval for the ode numerical solver.
tinterv = c(0,6)
## solve the ode numerically using predefined ode parameters. alpha=1, beta=1, gamma=4, delta=1.
kkk0$solve_ode(c(1,1,4,1),xinit,tinterv)

## Create another ode class object by using the simulation data from the ode numerical solver.
## If users have experiment data, they can replace the simulation data with the experiment data.
## set initial values for ode parameters.
init_par = rep(c(0.1),4)
init_yode = kkk0$y_ode
init_t = kkk0$t
kkk = ode$new(1,fun=LV_fun,grfun=LV_grlNODE,t=init_t,ode_par= init_par, y_ode=init_yode )
```

RBF *The 'RBF' class object*

Description

This a R6 class. It inherits from 'kernel' class. It provides the rbf kernel function and the 1st order derivative of rbf kernel function.

Format

`R6Class` object.

Value

an `R6Class` object which can be used for the rkhs interpolation.

Super class

`KGode::Kernel` -> RBF

Methods

Public methods:

- `RBF$greet()`
- `RBF$set_k_par()`
- `RBF$kern()`
- `RBF$dkd_kpar()`
- `RBF$dkdt()`
- `RBF$clone()`

Method `greet()`**:**

Usage:

```
RBF$greet()
```

Method `set_k_par()`**:**

Usage:

```
RBF$set_k_par(val)
```

Method `kern()`**:**

Usage:

```
RBF$kern(t1, t2)
```

Method `dkd_kpar()`**:**

Usage:

```
RBF$dkd_kpar(t1, t2)
```

Method `dkdt()`**:**

Usage:

```
RBF$dkdt(t1, t2)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
RBF$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Mu Niu, <mu.niu@glasgow.ac.uk>

`rkg` *The 'rkg' function*

Description

This function is used to create 'rkhs' class object and estimate ode parameters using standard gradient matching.

Usage

```
rkg(kkk, y_no, ktype)
```

Arguments

`kkk`	ode class object.
`y_no`	matrix(of size n_s*n_o) containing noisy observations. The row(of length n_s) represent the ode states and the column(of length n_o) represents the time points.
`ktype`	character containing kernel type. User can choose 'rbf' or 'mlp' kernel.

Details

Arguments of the 'rkg' function are 'ode' class, noisy observation, and kernel type. It return the interpolation for each of the ode states. The Ode parameters are estimated using gradient matching, and the results are stored in the 'ode' class as the ode_par attribute.

Value

return list containing :

- intp - list containing interpolation for each ode state.
- bbb - rkhs class objects for each ode state.

Author(s)

Mu Niu <mu.niu@glasgow.ac.uk>

Examples

```
## Not run:
require(mvtnorm)
noise = 0.1  ## set the variance of noise
SEED = 19537
set.seed(SEED)
## Define ode function, we use lotka-volterra model in this example.
## we have two ode states x[1], x[2] and four ode parameters alpha, beta, gamma and delta.
LV_fun = function(t,x,par_ode){
  alpha=par_ode[1]
  beta=par_ode[2]
  gamma=par_ode[3]
  delta=par_ode[4]
  as.matrix( c( alpha*x[1]-beta*x[2]*x[1] , -gamma*x[2]+delta*x[1]*x[2] ) )
}
## Define the gradient of ode function against ode parameters
## df/dalpha,  df/dbeta, df/dgamma, df/ddelta where f is the differential equation.
LV_grlNODE= function(par,grad_ode,y_p,z_p) {
alpha = par[1]; beta= par[2]; gamma = par[3]; delta = par[4]
dres= c(0)
dres[1] = sum( -2*( z_p[1,]-grad_ode[1,])*y_p[1,]*alpha )
dres[2] = sum( 2*( z_p[1,]-grad_ode[1,])*y_p[2,]*y_p[1,]*beta)
dres[3] = sum( 2*( z_p[2,]-grad_ode[2,])*gamma*y_p[2,] )
dres[4] = sum( -2*( z_p[2,]-grad_ode[2,])*y_p[2,]*y_p[1,]*delta)
dres
}

## create a ode class object
kkk0 = ode$new(2,fun=LV_fun,grfun=LV_grlNODE)
## set the initial values for each state at time zero.
xinit = as.matrix(c(0.5,1))
## set the time interval for the ode numerical solver.
tinterv = c(0,6)
## solve the ode numerically using predefined ode parameters. alpha=1, beta=1, gamma=4, delta=1.
kkk0$solve_ode(c(1,1,4,1),xinit,tinterv)

## Add noise to the numerical solution of the ode model and use it as the noisy observation.
n_o = max( dim( kkk0$y_ode) )
t_no = kkk0$t
y_no =  t(kkk0$y_ode) + rmvnorm(n_o,c(0,0),noise*diag(2))

## Create a ode class object by using the simulation data we created from the ode numerical solver.
## If users have experiment data, they can replace the simulation data with the experiment data.
## Set initial value of ode parameters.
init_par = rep(c(0.1),4)
init_yode = t(y_no)
init_t = t_no
kkk = ode$new(1,fun=LV_fun,grfun=LV_grlNODE,t=init_t,ode_par= init_par, y_ode=init_yode )

## The following examples with CPU or elapsed time > 10s
```

```
##Use function 'rkg' to estimate the ode parameters. The standard gradient matching method is coded
##in the the 'rkg' function. The parameter estimations are stored in the returned vector of 'rkg'.
## Choose a kernel type for 'rkhs' interpolation. Two options are provided 'rbf' and 'mlp'.
ktype ='rbf'
rkgres = rkg(kkk,y_no,ktype)
## show the results of ode parameter estimation using the standard gradient matching
kkk$ode_par

## End(Not run)
```

---

rkg3 *The 'rkg3' class object*

Description

This class provides advanced gradient matching method by using the ode as a regularizer.

Format

R6Class object.

Value

an R6Class object which can be used for improving ode parameters estimation by using ode as a regularizer.

Methods

`iterate(iter,innerloop,lamb)` Iteratively updating ode parameters and interpolation regression coefficients.

`witerate(iter,innerloop,dtilda,lamb)` Iteratively updating ode parameters and the warped interpolation regression coefficients.

`full(par,lam)` Updating ode parameters and rkhs interpolation regression coefficients simultaneously. This method is slow but guarantee convergence.

Public fields

`rk` the 'rkhs' class object containing the interpolation information for each state of the ode.

`ode_m` the 'ode' class object containing the information about the odes.

Active bindings

`ode_m` the 'ode' class object containing the information about the odes.

Methods

Public methods:

- rkg3$new()
- rkg3$greet()
- rkg3$add()
- rkg3$iterate()
- rkg3$witerate()
- rkg3$full()
- rkg3$wfull()
- rkg3$opfull()
- rkg3$wopfull()
- rkg3$cross()
- rkg3$fullos()
- rkg3$clone()

Method new():

Usage:

```
rkg3$new(rk = NULL, odem = NULL)
```

Method greet():

Usage:

```
rkg3$greet()
```

Method add():

Usage:

```
rkg3$add(x)
```

Method iterate():

Usage:

```
rkg3$iterate(iter, innerloop, lamb)
```

Method witerate():

Usage:

```
rkg3$witerate(iter, innerloop, dtilda, lamb)
```

Method full():

Usage:

```
rkg3$full(par, lam)
```

Method wfull():

Usage:

```
rkg3$wfull(par, lam, dtilda)
```

Method opfull():

Usage:

```
rkg3$opfull(lam)
```

Method `wopfull()`**:**

Usage:

```
rkg3$wopfull(lam, dtilda)
```

Method `cross()`**:**

Usage:

```
rkg3$cross(lam, testX, testY)
```

Method `fullos()`**:**

Usage:

```
rkg3$fullos(par)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
rkg3$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Mu Niu, <mu.niu@glasgow.ac.uk>

`rkhs` *The 'rkhs' class object*

Description

This class provide the interpolation methods using reproducing kernel Hilbert space.

Format

`R6Class` object.

Value

an `R6Class` object which can be used for doing interpolation using reproducing kernel Hilbert space.

Methods

`predict()` This method is used to make prediction on given time points

`skcross()` This method is used to do cross-validation to estimate the weighting parameter lambda of L^2 norm.

Public fields

`y` matrix(of size n_s*n_o) containing observation.

`t` vector(of length n_o) containing time points for observation.

`b` vector(of length n_o) containing coefficients of kernel or basis functions.

`lambda` scalar containing the weighting parameter for L2 norm of the reproducing kernel Hilbert space.

`ker` kernel class object containing kernel.

Methods

Public methods:

- `rkhs$new()`
- `rkhs$greet()`
- `rkhs$showker()`
- `rkhs$predict()`
- `rkhs$predictT()`
- `rkhs$lossRK()`
- `rkhs$grlossRK()`
- `rkhs$numgrad()`
- `rkhs$skcross()`
- `rkhs$mkcross()`
- `rkhs$loss11()`
- `rkhs$grloss11()`
- `rkhs$clone()`

Method `new()`:

Usage:

```
rkhs$new(y = NULL, t = NULL, b = NULL, lambda = NULL, ker = NULL)
```

Method `greet()`:

Usage:

```
rkhs$greet()
```

Method `showker()`:

Usage:

```
rkhs$showker()
```

Method `predict()`:

Usage:

```
rkhs$predict()
```

Method `predictT()`:

Usage:

```
rkhs$predictT(testT)
```

Method `lossRK()`**:**

Usage:

```
rkhs$lossRK(par, tl1, y_d, jitter)
```

Method `grlossRK()`**:**

Usage:

```
rkhs$grlossRK(par, tl1, y_d, jitter)
```

Method `numgrad()`**:**

Usage:

```
rkhs$numgrad(par, tl1, y_d, jitter)
```

Method `skcross()`**:**

Usage:

```
rkhs$skcross(init, bounded)
```

Method `mkcross()`**:**

Usage:

```
rkhs$mkcross(init)
```

Method `loss11()`**:**

Usage:

```
rkhs$loss11(par, tl1, y_d, jitter)
```

Method `grloss11()`**:**

Usage:

```
rkhs$grloss11(par, tl1, y_d, jitter)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
rkhs$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Mu Niu, <mu.niu@glasgow.ac.uk>

Examples

```
## Not run:
require(mvtnorm)
noise = 0.1  ## set the variance of noise
SEED = 19537
set.seed(SEED)
## Define ode function, we use lotka-volterra model in this example.
## we have two ode states x[1], x[2] and four ode parameters alpha, beta, gamma and delta.
LV_fun = function(t,x,par_ode){
  alpha=par_ode[1]
  beta=par_ode[2]
  gamma=par_ode[3]
  delta=par_ode[4]
  as.matrix( c( alpha*x[1]-beta*x[2]*x[1] , -gamma*x[2]+delta*x[1]*x[2] ) )
}
## Define the gradient of ode function against ode parameters
## df/dalpha,  df/dbeta, df/dgamma, df/ddelta where f is the differential equation.
LV_grlNODE= function(par,grad_ode,y_p,z_p) {
alpha = par[1]; beta= par[2]; gamma = par[3]; delta = par[4]
dres= c(0)
dres[1] = sum( -2*( z_p[1,]-grad_ode[1,])*y_p[1,]*alpha )
dres[2] = sum( 2*( z_p[1,]-grad_ode[1,])*y_p[2,]*y_p[1,]*beta)
dres[3] = sum( 2*( z_p[2,]-grad_ode[2,])*gamma*y_p[2,] )
dres[4] = sum( -2*( z_p[2,]-grad_ode[2,])*y_p[2,]*y_p[1,]*delta)
dres
}

## create a ode class object
kkk0 = ode$new(2,fun=LV_fun,grfun=LV_grlNODE)
## set the initial values for each state at time zero.
xinit = as.matrix(c(0.5,1))
## set the time interval for the ode numerical solver.
tinterv = c(0,6)
## solve the ode numerically using predefined ode parameters. alpha=1, beta=1, gamma=4, delta=1.
kkk0$solve_ode(c(1,1,4,1),xinit,tinterv)

## Add noise to the numerical solution of the ode model and use it as the noisy observation.
n_o = max( dim( kkk0$y_ode) )
t_no = kkk0$t
y_no =  t(kkk0$y_ode) + rmvnorm(n_o,c(0,0),noise*diag(2))

## Create a ode class object by using the simulation data we created from the ode numerical solver.
## If users have experiment data, they can replace the simulation data with the experiment data.
## Set initial value of ode parameters.
init_par = rep(c(0.1),4)
init_yode = t(y_no)
init_t = t_no
kkk = ode$new(1,fun=LV_fun,grfun=LV_grlNODE,t=init_t,ode_par= init_par, y_ode=init_yode )

## The following examples with CPU or elapsed time > 5s
####### rkhs interpolation for the 1st state of ode using 'rbf' kernel
```

```
### set initial value of length scale of rbf kernel
initlen = 1
aker = RBF$new(initlen)
bbb = rkhs$new(t(y_no)[1,],t_no,rep(1,n_o),1,aker)
## optimise lambda by cross-validation
## initial value of lambda
initlam = 2
bbb$skcross( initlam )

## make prediction using the 'predict()' method of 'rkhs' class and plot against the time.
plot(t_no,bbb$predict()$pred)

## End(Not run)
```

`third` *The 'third' function*

Description

This function is used to create 'rk3g' class objects and estimate ode parameters using ode regularised gradient matching.

Usage

```
third(lam, kkk, bbb, crtype, woption, dtilda)
```

Arguments

`lam`	scalar containing the weighting parameter of ode regularisation.
`kkk`	'ode' class object containing all information about the odes.
`bbb`	list of 'rkhs' class object containing the interpolation for all ode states.
`crtype`	character containing the optimisation scheme type. User can choose 'i' or '3'. 'i' is for fast iterative scheme and '3' for optimising the ode parameters and interpolation coefficients simultaneously.
`woption`	character containing the indication of using warping. If the warping scheme is done before using the ode regularisation, user can choose 'w' otherwise just leave this option empty.
`dtilda`	vector(of length n_o) containing the gradient of warping function. This variable is only used if user want to combine warping and the ode regularisation.

Details

Arguments of the 'third' function are ode regularisation weighting parameter, 'ode' class objects, 'rkhs' class objects, noisy observation, type of regularisation scheme, option of warping and the gradient of warping function. It return the interpolation for each of the ode states. The ode parameters are estimated using gradient matching, and the results are stored in the ode_par attribute of 'ode' class.

Value

return list containing :

- oppar - vector(of length n_p) containing the ode parameters estimation. n_p is the length of ode parameters.
- rk3 - list of 'rkhs' class object containing the updated interpolation results.

Author(s)

Mu Niu <mu.niu@glasgow.ac.uk>

Examples

```
## Not run:
require(mvtnorm)
noise = 0.1
SEED = 19537
set.seed(SEED)
## Define ode function, we use lotka-volterra model in this example.
## we have two ode states x[1], x[2] and four ode parameters alpha, beta, gamma and delta.
LV_fun = function(t,x,par_ode){
  alpha=par_ode[1]
  beta=par_ode[2]
  gamma=par_ode[3]
  delta=par_ode[4]
  as.matrix( c( alpha*x[1]-beta*x[2]*x[1] , -gamma*x[2]+delta*x[1]*x[2] ) )
}
## Define the gradient of ode function against ode parameters
## df/dalpha,  df/dbeta, df/dgamma, df/ddelta where f is the differential equation.
LV_grlNODE= function(par,grad_ode,y_p,z_p) {
alpha = par[1]; beta= par[2]; gamma = par[3]; delta = par[4]
dres= c(0)
dres[1] = sum( -2*( z_p[1,]-grad_ode[1,])*y_p[1,]*alpha )
dres[2] = sum( 2*( z_p[1,]-grad_ode[1,])*y_p[2,]*y_p[1,]*beta)
dres[3] = sum( 2*( z_p[2,]-grad_ode[2,])*gamma*y_p[2,] )
dres[4] = sum( -2*( z_p[2,]-grad_ode[2,])*y_p[2,]*y_p[1,]*delta)
dres
}

## create a ode class object
kkk0 = ode$new(2,fun=LV_fun,grfun=LV_grlNODE)
## set the initial values for each state at time zero.
xinit = as.matrix(c(0.5,1))
## set the time interval for the ode numerical solver.
tinterv = c(0,6)
## solve the ode numerically using predefined ode parameters. alpha=1, beta=1, gamma=4, delta=1.
kkk0$solve_ode(c(1,1,4,1),xinit,tinterv)

## Add noise to the numerical solution of the ode model and use it as the noisy observation.
n_o = max( dim( kkk0$y_ode) )
t_no = kkk0$t
```

```
y_no =  t(kkk0$y_ode) + rmvnorm(n_o,c(0,0),noise*diag(2))

## create a ode class object by using the simulation data we created from the ode numerical solver.
## If users have experiment data, they can replace the simulation data with the experiment data.
## set initial value of Ode parameters.
init_par = rep(c(0.1),4)
init_yode = t(y_no)
init_t = t_no
kkk = ode$new(1,fun=LV_fun,grfun=LV_grlNODE,t=init_t,ode_par= init_par, y_ode=init_yode )

## The following examples with CPU or elapsed time > 10s

## Use function 'rkg' to estimate the ode parameters.
ktype ='rbf'
rkgres = rkg(kkk,y_no,ktype)
bbb = rkgres$bbb

############# gradient matching + ode regularisation
crtype='i'
## using cross validation to estimate the weighting parameters of the ode regularisation
lam=c(1e-4,1e-5)
lamil1 = crossv(lam,kkk,bbb,crtype,y_no)
lambdai1=lamil1[[1]]

## estimate ode parameters using gradient matching and ode regularisation
res = third(lambdai1,kkk,bbb,crtype)
## display the ode parameter estimation.
res$oppar

## End(Not run)
```

`Warp` *The 'Warp' class object*

Description

This class provide the warping method which can be used to warp the original signal to sinusoidal like signal.

Format

`R6Class` object.

Value

an `R6Class` object which can be used for doing interpolation using reproducing kernel Hilbert space.

Methods

`warpsin(len ,lop,p0,eps)` This method is used to warp the initial interpolation into a sinusoidal shape.

`slowWarp(lens,peod,eps)` This method is used to find the optimised initial hyper parameters for the sigmoid basis function for each ode states.

`warpLossLen(par,lam,p0,eps)` This method is used to implement the loss function for warping. It is called by the 'warpSin' function.

Public fields

`y` matrix(of size n_s*n_o) containing observation.

`t` vector(of length n_o) containing time points for observation.

`b` vector(of length n_o) containing coefficients of kernel or basis functions.

`lambda` scalar containing the weighting parameter for penalising the length of warped time span.

`ker` kernel class object containing sigmoid basis function.

Methods

Public methods:

- `Warp$new()`
- `Warp$greet()`
- `Warp$showker()`
- `Warp$warpLoss()`
- `Warp$warpLossLen()`
- `Warp$warpSin()`
- `Warp$slowWarp()`
- `Warp$clone()`

Method `new()`:

Usage:

```
Warp$new(y = NULL, t = NULL, b = NULL, lambda = NULL, ker = NULL)
```

Method `greet()`:

Usage:

```
Warp$greet()
```

Method `showker()`:

Usage:

```
Warp$showker()
```

Method `warpLoss()`:

Usage:

```
Warp$warpLoss(par, len, p0, eps)
```

Method `warpLossLen()`**:**

Usage:

```
Warp$warpLossLen(par, lam, p0, eps)
```

Method `warpSin()`**:**

Usage:

```
Warp$warpSin(len, lop, p0, eps)
```

Method `slowWarp()`**:**

Usage:

```
Warp$slowWarp(lens, p0, eps)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Warp$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Mu Niu, <mu.niu@glasgow.ac.uk>

`warpfun` *The 'warpfun' function*

Description

This function is used to produce the warping function and learning the interpolation in the warped time domain.

Usage

```
warpfun(kkkrkg, bbb, peod, eps, fixlens, y_no, testData, witer)
```

Arguments

`kkkrkg`	'ode' class object.
`bbb`	list of 'rkhs' class object.
`peod`	vector(of length n_s) containing the period of warped signal. n_s is the length of the ode states.
`eps`	vector(of length n_s) containing the uncertainty level of the period. n_s is the length of the ode states.
`fixlens`	vector(of length n_s) containing the initial values of the hyper parameters of sigmoid basis function.

y_no	matrix(of size n_s*n_o) containing noisy observations. The row(of length n_s) represent the ode states and the column(of length n_o) represents the time points.
testData	vector(of size n_x) containing user defined time points which will be warped by the warping function.
witer	scale containing the number of iterations for optimising the hyper parameters of warping.

Details

Arguments of the 'warpfun' function are 'ode' class, 'rkhs' class, period of warped signal, uncertainty level of the period, initial values of the hyper parameters for sigmoid basis function, noisy observations and the time points that user want to warped. It return the interpolation for each of the ode states. The ode parameters are estimated using gradient matching, and the results are stored in the 'ode' class as the ode_par attribute.

Value

return list containing :

- dtilda - vector(of length n_x) containing the gradients of warping function at user defined time points.
- bbbw - list of 'rkhs' class object containing the interpolation in warped time domain.
- wtime - vector(of length n_x) containing the warped time points.
- wfun - list of 'rkhs' class object containing information about warping function.
- wkkk - 'ode' class object containing the result of parameter estimation using the warped signal and gradient matching.

Author(s)

Mu Niu <mu.niu@glasgow.ac.uk>

Examples

```
## Not run:
require(mvtnorm)
noise = 0.1
SEED = 19537
set.seed(SEED)
## Define ode function, we use lotka-volterra model in this example.
## we have two ode states x[1], x[2] and four ode parameters alpha, beta, gamma and delta.
LV_fun = function(t,x,par_ode){
  alpha=par_ode[1]
  beta=par_ode[2]
  gamma=par_ode[3]
  delta=par_ode[4]
  as.matrix( c( alpha*x[1]-beta*x[2]*x[1] , -gamma*x[2]+delta*x[1]*x[2] ) )
}
## Define the gradient of ode function against ode parameters
```

```
## df/dalpha,  df/dbeta, df/dgamma, df/ddelta where f is the differential equation.
LV_grlNODE= function(par,grad_ode,y_p,z_p) {
alpha = par[1]; beta= par[2]; gamma = par[3]; delta = par[4]
dres= c(0)
dres[1] = sum( -2*( z_p[1,]-grad_ode[1,])*y_p[1,]*alpha )
dres[2] = sum( 2*( z_p[1,]-grad_ode[1,])*y_p[2,]*y_p[1,]*beta)
dres[3] = sum( 2*( z_p[2,]-grad_ode[2,])*gamma*y_p[2,] )
dres[4] = sum( -2*( z_p[2,]-grad_ode[2,])*y_p[2,]*y_p[1,]*delta)
dres
}

## create a ode class object
kkk0 = ode$new(2,fun=LV_fun,grfun=LV_grlNODE)
## set the initial values for each state at time zero.
xinit = as.matrix(c(0.5,1))
## set the time interval for the ode numerical solver.
tinterv = c(0,6)
## solve the ode numerically using predefined ode parameters. alpha=1, beta=1, gamma=4, delta=1.
kkk0$solve_ode(c(1,1,4,1),xinit,tinterv)

## Add noise to the numerical solution of the ode model and use it as the noisy observation.
n_o = max( dim( kkk0$y_ode) )
t_no = kkk0$t
y_no =  t(kkk0$y_ode) + rmvnorm(n_o,c(0,0),noise*diag(2))

## create a ode class object by using the simulation data we created from the Ode numerical solver.
## If users have experiment data, they can replace the simulation data with the experiment data.
## set initial value of Ode parameters.
init_par = rep(c(0.1),4)
init_yode = t(y_no)
init_t = t_no
kkk = ode$new(1,fun=LV_fun,grfun=LV_grlNODE,t=init_t,ode_par= init_par, y_ode=init_yode )

## The following examples with CPU or elapsed time > 10s

## Use function 'rkg' to estimate the Ode parameters.
ktype ='rbf'
rkgres = rkg(kkk,y_no,ktype)
bbb = rkgres$bbb

###### warp all ode states
peod = c(6,5.3) ## the guessing period
eps= 1          ## the uncertainty level of period

###### learn the initial value of the hyper parameters of the warping basis function
fixlens=warpInitLen(peod,eps,rkgres)

kkkrkg = kkk$clone() ## make a copy of ode class objects
##learn the warping function, warp data points and do gradient matching in the warped time domain.
www = warpfun(kkkrkg,bbb,peod,eps,fixlens,y_no,kkkrkg$t)

dtilda= www$dtilda  ## gradient of warping function
bbbw = www$bbbw      ## interpolation in warped time domain
```

```
resmtest = www$wtime  ## warped time points
##display the results of parameter estimation using gradient matching in the warped time domain.
www$wkkk$ode_par

## End(Not run)
```

---

warpInitLen *The 'warpInitLen' function*

Description

This function is used to find the optmised initial value of the hyper parameter for the sigmoid basis function which is used for warping.

Usage

```
warpInitLen(peod, eps, rkgres, lens)
```

Arguments

peod	vector(of length n_s) containing the period of warped signal. n_s is the length of the ode states.
eps	vector(of length n_s) containing the uncertainty level of the period. n_s is the length of the ode states.
rkgres	list containing interpolation and 'rkhs' class objects for all ode states.
lens	vector(of length n_l) containing a list of hyper parameters of sigmoid basis function. n_l is the length of user defined hyper parameters of the sigmoid basis functino.

Details

Arguments of the 'warpfun' function are 'ode' class, 'rkhs' class, period of warped signal, uncertainty level of the period, initial values of the hyper parameters for sigmoid basis function, noisy observations and the time points that user want to warped. It return the interpolation for each of the ode states. The ode parameters are estimated using gradient matching, and the results are stored in the 'ode' class as the ode_par attribute.

Value

return list containing :

- wres- vector(of length n_s) contaning the optimised initial hyper parameters of sigmoid basis function for each ode states.

Author(s)

Mu Niu <mu.niu@glasgow.ac.uk>

Examples

```
## Not run:
require(mvtnorm)
noise = 0.1
SEED = 19537
set.seed(SEED)
## Define ode function, we use lotka-volterra model in this example.
## we have two ode states x[1], x[2] and four ode parameters alpha, beta, gamma and delta.
LV_fun = function(t,x,par_ode){
  alpha=par_ode[1]
  beta=par_ode[2]
  gamma=par_ode[3]
  delta=par_ode[4]
  as.matrix( c( alpha*x[1]-beta*x[2]*x[1] , -gamma*x[2]+delta*x[1]*x[2] ) )
}
## Define the gradient of ode function against ode parameters
## df/dalpha,  df/dbeta, df/dgamma, df/ddelta where f is the differential equation.
LV_grlNODE= function(par,grad_ode,y_p,z_p) {
alpha = par[1]; beta= par[2]; gamma = par[3]; delta = par[4]
dres= c(0)
dres[1] = sum( -2*( z_p[1,]-grad_ode[1,])*y_p[1,]*alpha )
dres[2] = sum( 2*( z_p[1,]-grad_ode[1,])*y_p[2,]*y_p[1,]*beta)
dres[3] = sum( 2*( z_p[2,]-grad_ode[2,])*gamma*y_p[2,] )
dres[4] = sum( -2*( z_p[2,]-grad_ode[2,])*y_p[2,]*y_p[1,]*delta)
dres
}

## create a ode class object
kkk0 = ode$new(2,fun=LV_fun,grfun=LV_grlNODE)
## set the initial values for each state at time zero.
xinit = as.matrix(c(0.5,1))
## set the time interval for the ode numerical solver.
tinterv = c(0,6)
## solve the ode numerically using predefined ode parameters. alpha=1, beta=1, gamma=4, delta=1.
kkk0$solve_ode(c(1,1,4,1),xinit,tinterv)

## Add noise to the numerical solution of the ode model and use it as the noisy observation.
n_o = max( dim( kkk0$y_ode) )
t_no = kkk0$t
y_no =  t(kkk0$y_ode) + rmvnorm(n_o,c(0,0),noise*diag(2))

## create a ode class object by using the simulation data we created from the Ode numerical solver.
## If users have experiment data, they can replace the simulation data with the experiment data.
## set initial value of Ode parameters.
init_par = rep(c(0.1),4)
init_yode = t(y_no)
init_t = t_no
kkk = ode$new(1,fun=LV_fun,grfun=LV_grlNODE,t=init_t,ode_par= init_par, y_ode=init_yode )

## The following examples with CPU or elapsed time > 10s
```

```
## Use function 'rkg' to estimate the Ode parameters.
ktype ='rbf'
rkgres = rkg(kkk,y_no,ktype)
bbb = rkgres$bbb

###### warp all ode states
peod = c(6,5.3) ## the guessing period
eps= 1          ## the uncertainty level of period

###### learn the initial value of the hyper parameters of the warping basis function
fixlens=warpInitLen(peod,eps,rkgres)

## End(Not run)
```

Index