```
#############################################################
#####################################
# An R example of curve generation, registration and
classification for JCRC method.
#############################################################
#####################################

######## install and load packages #############
pck <- c("here","refund","mgcv","doParallel")
install.packages(pck)
mypackages<-lapply(pck, require, character.only = TRUE)
# set the path to the current working directory
path = paste(here(),"jcrc",sep = "/")
# set the savepath to save the data generated in this example
savepath = paste(here(),"data",sep = "/")


# generate true mean curves
m<-100  # sampling frequency
t <- seq(0, 1, length = m + 2)[2:(m + 1)]; # equidistant points
x11=0.6*dnorm(t)+0.4*dbeta(t,2,3); x21=sin(2*pi*t+0.5);
x12=0.5*dnorm(t,mean    =    0.5,sd=0.5)+0.5*dbeta(t,3,4);
x22=sin(2*pi*t^1.2+0.5);
N0<-matrix(0, nrow = m, ncol = 3); Ab0<-matrix(0, nrow = m,
ncol = 3);
N0[,1] <- t; N0[,2] <- x11; N0[,3] <- x21; Ab0[,1] <- t; Ab0[,2]
<- x12; Ab0[,3] <- x22;
# save the true curves, k represents the number of simulation
runs
k=1
true_curve<-list(x1=x11,y1=x21,x2=x12,y2=x22)
save(true_curve,file=paste(savepath,paste("true_curve",as
.character(k),".Rda"),sep = "/"))


# B-spline representation of the true mean curves
lenkt<-8 # number of basis functions
kts<-seq(0,1,length=lenkt)[2:(lenkt-1)]
basis_fct<-make_basis_fct(kts = kts, intercept = TRUE) #
basis function
r<-basis_fct(t) # extract basis functions evaluated at t

# Extract the basis coeficients
bcc1<-basis_coe_class(r,N0[,2],Ab0[,2],scale=1);
```

```
beta_at0=bcc1[[1]];                    beta_at1=bcc1[[2]];
beta_at2=bcc1[[3]];
bcc2<-basis_coe_class(r,N0[,3],Ab0[,3],scale=1);
beta_bt0=bcc2[[1]];                    beta_bt1=bcc2[[3]];
beta_bt2=bcc2[[3]];


# Generate the warped curves
n<-120 # sample size
# generate Matern covariance function with parameter
amp_par_t
amp_par_t <- c(100,0.3,3) # parameter used to create the
Matern covariance function
amp_cov1 <- make_cov_fct(Matern, noise = TRUE) # Matern
covariance function

# generate warping function
tw <- seq(0, 1, length = 4)#anchor points for the warping
function
warp_fct <- make_warp_fct(type = 'smooth', tw = tw) #
type="smooth", Hyman filtered
warp_cov_true1 <- matrix(c(10, 4, 4, 8), 2, 2)
warp_cov_true2 <- matrix(c(10, 8, 8, 15), 2, 2)

# generate the warped curves
sigma<-0.02
sigma_w<-sigma/4
sigma_r<-sigma

# parallel computing
#no_cores=detectCores()
#cl<-makeCluster(no_cores-1)
#registerDoParallel(cl)
#foreach (k = 1:100) %dopar% {
w_t1<-replicate(n/2,(t(chol(warp_cov_true1))*rnorm(2,sd=s
igma_w))[,1])
w_t2<-replicate(n/2,(t(chol(warp_cov_true2))*rnorm(2,sd=s
igma_w))[,1])

amp1   <-   lapply(1:(n/2),   function(i)   sigma_r   *
t(chol(amp_cov1(t, amp_par_t))) %*% rnorm(m))
beta_ran1  <-   lapply(1:(n/2),  function(i)  basis_coe(r,
amp1[[i]]))
amp2   <-   lapply(1:(n/2),   function(i)   sigma_r   *
```

```
t(chol(amp_cov1(t, amp_par_t))) %*% rnorm(m))
beta_ran2 <-   lapply(1:(n/2),  function(i)  basis_coe(r,
amp2[[i]]))
amp3   <-   lapply(1:(n/2),   function(i)   sigma_r   *
t(chol(amp_cov1(t, amp_par_t))) %*% rnorm(m))
beta_ran3 <-   lapply(1:(n/2),  function(i)  basis_coe(r,
amp3[[i]]))
amp4   <-   lapply(1:(n/2),   function(i)   sigma_r   *
t(chol(amp_cov1(t, amp_par_t))) %*% rnorm(m))
beta_ran4 <-   lapply(1:(n/2),  function(i)  basis_coe(r,
amp4[[i]]))
ya1 <- lapply(1:(n/2), function(i) {(basis_fct(warp_fct(0,
w_t1[, i], t)) %*% (beta_at0 + beta_at1 + beta_ran1[[i]])
                                    + amp1[[i]] + rnorm(m, sd =
sigma))[,1]})
ya2 <- lapply(1:(n/2), function(i) {(basis_fct(warp_fct(0,
w_t2[, i], t)) %*% (beta_at0 + beta_at2 + beta_ran2[[i]])
                                    + amp2[[i]] + rnorm(m, sd =
sigma))[,1]})
yb1 <- lapply(1:(n/2), function(i) {(basis_fct(warp_fct(0,
w_t1[, i], t)) %*% (beta_bt0 + beta_bt1 + beta_ran3[[i]])
                                    + amp3[[i]] + rnorm(m, sd =
sigma))[,1]})
yb2 <- lapply(1:(n/2), function(i) {(basis_fct(warp_fct(0,
w_t2[, i], t)) %*% (beta_bt0 + beta_bt2 + beta_ran4[[i]])
                                    + amp4[[i]] + rnorm(m, sd =
sigma))[,1]})
tya1 <- lapply(1:(n/2), function(i) {r %*% (beta_at0 +
beta_at1 + beta_ran1[[i]])})
tya2 <- lapply(1:(n/2), function(i) {r %*% (beta_at0 +
beta_at2 + beta_ran2[[i]])})
tyb1 <- lapply(1:(n/2), function(i) {r %*% (beta_bt0 +
beta_bt1 + beta_ran3[[i]])})
tyb2 <- lapply(1:(n/2), function(i) {r %*% (beta_bt0 +
beta_bt2 + beta_ran4[[i]])})

ya = append(ya1,ya2)
yb = append(yb1,yb2)
data_mat=append(ya,yb)
save(data_mat,file=paste(savepath,paste("data_mat",as.cha
racter(k),".Rda"),sep = "/"))

# test data
n0<-n/4
```

```
ind = 1:n0
test_data = list()
t1 <- lapply(1:(2*n0), function(x) t)
for (i in ind){
  test_data[[i]] = list(t0 = t1[[i]], y1 = ya1[[i]], y2 =
yb1[[i]])
  test_data[[i + length(ind)]] = list(t0 = t1[[i]], y1 =
ya2[[i]], y2 = yb2[[i]])
}
save(test_data,file=paste(savepath,paste("test_data",as.c
haracter(k),".Rda"),sep = "/"))

# training data
n1 <- n - 2*n0
t2 <- lapply(1:n1, function(x) t)
ya_tr = append(ya1[(n0+1):(n/2)],ya2[(n0+1):(n/2)])
yb_tr = append(yb1[(n0+1):(n/2)],yb2[(n0+1):(n/2)])
train_data=append(ya_tr,yb_tr)
save(train_data,file=paste(savepath,paste("train_data",as
.character(k),".Rda"),sep = "/"))

#}
#stopCluster(cl)


###### generate the binary outcomes for the model
##############
# true value of the functional coefficients and the scalar
coefficients
betat1 <- cos(2*pi*t);  betat2 <- 2*(t-1)^2;
b0 = 0.1; b1 = -0.5;

aa1 <- 1; aa2 <- 2;
bb1 <- 0.5; bb2 <- 1.5;

# response outcomes
Y <- resp_gene(true_data, t, betat1, betat2, n, b0, b1, b2,
aa1, aa2, bb1, bb2, check = "F")
save(Y,file=paste(savepath,paste("Y_s.Rda"),sep = "/"))

#### curve registration #########
t2 <- lapply(1:n, function(x) t)
warp_cov <- make_cov_fct(unstr_cov, noise = FALSE, param =
c(2, 2, 0))
```

```
Index = Y[[5]]
K = list(ind1 = which(Index==1), ind2 = which(Index==0)) #
class labels
ya = data_mat[1:n] # x-coordinates
yb = data_mat[(n+1):(2*n)] # y-coordinates
res <- pavpop_fixrandwarp(ya, yb, t2, K, basis_fct = basis_fct,
warp_fct = warp_fct,
                      amp_cov  =  amp_cov1,  warp_cov  =
warp_cov) # curve registration procedure
Aligned <- Aligned_curves(ya,yb,t2,K,res)
save(Aligned,file=paste(savepath,paste("Aligned",as.chara
cter(k),".Rda"),sep = "/")) # k is the number of repitition

## estimations of the scalar coeffcients and the functional
coeffcients ###
funcs_tr    <-    list(rbind(Aligned[[1]],Aligned[[3]]),
rbind(Aligned[[2]],Aligned[[4]]))
covt <- as.matrix(Y$x1)
fit.pat <- fpfr(Y$Res, covt, funcs = funcs_tr, kz = 35, kb
= 35,
             family=binomial(link  =  "logit"),  method  =
"REML", smooth.cov=FALSE)
scl_est <-fit.pat$beta.covariates
fun_est <- fit.pat$BetaHat


##### comparison with other methods ########
load(paste(savepath,paste("train_data",as.character(k),".
Rda"),sep = "/"))
Index_train = Y[[k]][[5]][1:(2*n0)]
K   =   list(ind1   =   which(Index_train==1),   ind2   =
which(Index_train==0))
#cat(length(K$ind1),length(K$ind2))
ya_tr = data_mat[1:(2*n0)]
yb_tr = data_mat[(2*n0+1):(4*n0)]
res <- pavpop_fixrandwarp(ya_tr, yb_tr, t2, K, basis_fct =
basis_fct, warp_fct = warp_fct,
                      amp_cov  =  amp_cov1,  warp_cov  =
warp_cov, iter = c(6,6),
                      like_optim_control  =  list(upper  =
rep(Inf, 6),
                                                  lower        =
c(1e-2,1e-2,0.5001,1e-2,1e-2,1e-2)))
Ay_train <- Aligned_curves(ya_tr,yb_tr,t2,K,res)
```

```
save(Ay_train,file=paste(savepath,paste("Ay_train",as.cha
racter(k),".Rda"),sep = "/"))

load(paste(savepath,paste("test_data",as.character(k),".R
da"),sep = "/"))
Index_test = Y[[k]][[5]][(2*n0+1):(4*n0)]
w_ran_te1   <-   test_wran(warp_fct,   res,   test_data,
length(Index_test), basis_fct, amp_cov1, axis = '2D')
save(w_ran_te1,file=paste(savepath,paste("w_ran_te1",as.c
haracter(k),".Rda"),sep = "/"))

y <- Y[[k]]
y_tr <- y$Res[1:(2*n0)]
y_tr <- c(y_tr[K[[1]]],y_tr[K[[2]]])
covt1 <- y$x1[1:(2*n0)]
covt1 <- c(covt1[K[[1]]],covt1[K[[2]]])

# LLR
# use scalar variables only
Ini <- clasf_scal(n0,y)
Cla_scal <- Ini$C1


# JCRC
# use both scalar variable and functional variable
fucoef2 <- func_coef(Ay_train, covat = as.matrix(covt1), kz,
kb, d = 'Aligned')
Ini3 <- classi_fs(y,n0,data_mat,warp_fct,basis_fct, initial
= 1, p= 0, iteration = 5, kz = kz, kb = kb, t2, res, w_ran_te1,
fucoef2)
Cla_fs <- Ini3$C2


# JCRC-f
# use functional variable only
fucoef1 <- func_coef(Ay_train, covat = NULL, kz, kb, d =
'Aligned')
Ini2   <-   clasf_func2(y,n0,dat,warp_fct,basis_fct,   p=0,
iteration = 5, kz, kb, t2,res,w_ran_te1,fucoef1)
Cla_func <- Ini2$C2
```

```
############################### functions required in the
main codes ###################
######## some of them are copied from R package "pavpop4.0"
(Raket 2016) ###############
######## and are changed to fit our approach ##############
###### @refernce: L. L. Raket (2016), “pavpop Version 0.10,”
available at http://github.com/larslau/pavpop/.####


################### make_basis_fct()###################
make_basis_fct <- function(kts = NULL, df = NULL, type =
'B-spline', intercept = FALSE, control = list()) {
  # Match type
  types   <-   c('B-spline',   'increasing',   'intercept',
'Fourier', 'wavelet')
  type <- types[pmatch(type, types)]

  if (is.na(type)) stop('Invalid type of basis.')

  # Check that kts or df is supplied if type is different from
'intercept'
  if (type != 'intercept' & is.null(kts) & is.null(df))
stop('You must supply either list of knots or degrees of
freedom.')



  # Control boundary
  if (!is.null(control$boundary)) {
    boundary = control$boundary
  } else {
    # If kts is supplied, boundary is given by extending the
knot list
    # one "step-length" under the assumption of equidistant
knots
    if (!is.null(kts)) {
      boundary <- range(kts) + diff(range(kts)) * c(-1 /
(length(kts) - 1), 1 / (length(kts) - 1))
    } else if (type != 'intercept') {
      # No boundary supplied, no knots
      warning('No boundary knots or evaluation knots supplied,
assuming observation points are in (0, 1)')
      boundary <- c(0, 1)
    }
  }
```

```
  # Control constaints
  if (is.null(control$constraints) & type != 'increasing')
{
    constraints <- 'none'
  } else {
    constraints <- 'positive'
  }

  #
  # B-spline basis
  #
  if (type == 'B-spline') {
    # Extract order og B-spline basis
    if (is.null(control$order)) {
      order <- 4
    } else {
      order <- control$order
    }
    if (is.null(kts)){
      kts <- seq(boundary[1], boundary[2], length = df - order
+ (3L - intercept))
      kts <- head(tail(kts, -1), -1)
    }
    Aknots <- sort(c(rep(boundary, order), kts))

    # Should sparse matrices be used?
    if (is.null(control$sparse)) {
      sparse <- FALSE
    } else {
      sparse <- control$sparse
    }

    # Basis function to return
    b <- function(t, deriv = FALSE) {
      basis <- splines::splineDesign(Aknots, t, ord = order,
derivs = rep(0,length(t)), outer.ok = TRUE, sparse = sparse)
      if (!intercept) basis <- basis[, -1]
      return(basis[])
    }
    attr(b, 'boundary') <- boundary
  }

  #
```

```
  # Increasing spline basis
  #
  if (type == 'increasing') {
    # Extract order og I-spline basis
    if (is.null(control$order)) {
      order <- 3
    } else {
      order <- control$order
    }

    # Set boundary and B-spline knots (for derivative)
    boundary <- range(kts)
    Aknots <- sort(c(rep(boundary, order - 1), kts))

    b <- function(t, deriv = FALSE) {
      if (!deriv) {
        basis <- ispline(t, knots = kts, d = order)
      } else {
        basis <- t(c(order / 1:(order - 1), rep(1, length(kts)
-    order),    order    /    (order    -    1):1)    *
t(splines::splineDesign(Aknots, t, ord = order, derivs =
rep(0,length(t)), outer.ok = TRUE) * (length(kts) - 1) /
diff(range(kts))))
      }
      if (intercept) {
        basis  <-  cbind(!deriv,  matrix(basis,  nrow  =
length(t)))
      }
      dims <- dim(basis)
      basis <- as.numeric(basis)
      dim(basis) <- dims
      return(basis)
    }
  }

  #
  # Only intercept
  #
  if (type == 'intercept') {
    b <- function(t, deriv = FALSE) {
      t[] <- ifelse(deriv, 0, 1)
      dim(t) <- c(length(t), 1)
      return(t)
    }
```

```
    attr(b, 'df') <- 1
    attr(b, 'intercept') <- TRUE

    return(b)
  }

  attr(b, 'df') <- length(kts) + order - 1 - 1 * (type ==
'increasing') + intercept
  attr(b, 'intercept') <- intercept
  attr(b, 'constraints') <- constraints
  return(b)
}



################### ispline()############################
ispline <- function (t, knots, d) {
  if (is.null(knots) || any(is.na(knots)) || any(diff(knots)
== 0) || length(knots) <= 2)
    return(t)
  m <- length(knots)
  n <- length(t)
  interval <- findInterval(t, knots, all.inside = TRUE)
  M <- sapply(sequence(m - 1), `==`, interval)
  for (i in 2:(d + 1)) {
    tik <- c(knots[-1], rep(knots[m], i - 2))
    ti <- c(rep(knots[1], i - 2), knots[-m])
    M <- M %*% diag(1 / (tik - ti))
    Dx <- Dt <- array(0, dim = c(m + i - 3, m + i - 2))
    Dx[1L + 0L:(m + i - 4L) * (m + i - 2L)] <- -1
    Dx[1L:(m + i - 3L) * (m + i - 2L)] <- 1
    Dt[1L + 0L:(m + i - 4L) * (m + i - 2L)] <- tik
    Dt[1L:(m + i - 3L) * (m + i - 2L)] <- -ti
    M <- (M * t) %*% Dx + M %*% Dt
  }
  M <- M[, -1, drop = FALSE]
  S <- array(1, dim = rep(NCOL(M), 2))
  S[upper.tri(S)] <- 0
  I <- M %*% S
  I[t > max(knots), ] <- 1
  return(I)
}
```

```
######################### basis_coe_class()###########
basis_coe_class <- function(r, x1, x2, scale = 1){
  b1<- basis_coe(r, x1)
  b2<- basis_coe(r, x2)
  beta_t0 = (b1 + b2)/2
  beta_t1 = (b1 - beta_t0) - mean(b1 - beta_t0)
  beta_t2 = (b2 - beta_t0) - mean(b2 - beta_t0)
  return(list(beta_t0 = beta_t0, beta_t1 = beta_t1/scale,
beta_t2 = beta_t2/scale))
}



######################### basis_coe() ################
basis_coe <- function(r,x){
  A = as.matrix(r)
  B = as.matrix(x)
  asvd = svd(A)
  adiag = diag(1/asvd$d)
  return(asvd$v %*% adiag %*% t(asvd$u) %*% B)
  #solution = asvd$v %*% adiag %*% t(asvd$u) %*% B
  #check = A %*% solution
}



#################### make_cov_fct() ###################
make_cov_fct <- function(cov_fct, noise = TRUE, param = NULL,
ns = NULL, ...) {
  # Check ns related things
  if (!is.null(ns)) {
    if (is.null(ns$knots)) stop('ns must be a list with the
argument knots.')
    if (ns$knots < 2) {
      warning('number of knots should at least be 2, ignoring
argument.')
      ns <- NULL
    }
    if (is.null(ns$range)) {
      warning('No range specified using c(0, 1).')
      ns$range <- c(0, 1)
    }
  }
```

```
  if (!is.null(attr(cov_fct, 'discrete'))) {
    # Discrete covariances
    if (attr(cov_fct, 'discrete')) {
      if (noise) {
        f <- function(t, param) cov_fct(t, param) + id_cov(t)
      } else {
        f <- cov_fct
      }
    } else {
      stop('attribute \'discrete\' should be NULL for
non-discrete covariances.')
    }
  } else {
    # Non-discrete covariances
    if (attr(cov_fct, 'stationary')) {
      # Faster construction of stationary covariances
      if (is.null(ns)) {
        # stationary covariance, fill rows and columns
simultaneously
        f <- function (t, param) {
          m <- length(t)
          S <- diag(cov_fct(0, param, ...) + noise, m)
          if (m > 1) {
            for (i in 1:(m - 1)) {
              S[i, (i + 1):m] <- S[(i + 1):m, i] <-
cov_fct(abs(t[i] - t[(1 + i):m]), param, ...)
            }
          }
          return(S)
        }
      } else {
        # Stationary covariance made non-stationary
        knots <- ns$knots
        fixed <- ifelse(!is.null(ns$fixed), ns$fixed,
max(knots))
        m_ns <- knots - length(fixed)
        f <- function (t, param) {
          ns_param <- rep(1, length = knots)
          ns_param[-fixed] <- param[1:m_ns]
          ns <- spline(seq(ns$range[1], ns$range[2], length =
knots), ns_param, xout = t)$y
          m <- length(t)
          S <- diag(cov_fct(0, param[-(1:m_ns)], ...), m)
```

```
        if (m > 1) {
          for (i in 1:(m - 1)) {
            S[i, (i + 1):m] <- S[(i + 1):m, i] <-
cov_fct(abs(t[i] - t[(1 + i):m]), param[-(1:m_ns)], ...)
          }
        }
        S <- ns %*% t(ns) * S
        if (noise) diag(S) <- diag(S) + 1
        return(S)
      }
    }
  } else {
    # Non-stationary covariance, fill in all entries
separately
    if (is.null(ns)) {
      f <- function (t, param) {
        m <- length(t)
        S <- matrix(NA, m, m)
        for (i in 1:m) {
          for (j in i:m) {
            S[i, j] <- S[j, i] <- cov_fct(c(t[i], t[j]),
param, ...)
          }
        }
        if (noise) diag(S) <- diag(S) + 1
        return(S)
      }
    } else {
      knots <- ns$knots
      fixed <- ifelse(!is.null(ns$fixed), ns$fixed,
max(knots))
      m_ns <- knots - length(fixed)
      f <- function (t, param) {
        ns_param <- rep(1, length = knots)
        ns_param[-fixed] <- param[1:m_ns]
        ns <- spline(seq(ns$range[1], ns$range[2], length =
knots), ns_param, xout = t)$y

        m <- length(t)
        S <- matrix(NA, m, m)
        for (i in 1:m) {
          for (j in i:m) {
            S[i, j] <- S[j, i] <- ns[i] * ns[j] *
cov_fct(c(t[i], t[j]), param[-(1:m_ns)], ...)
```

```
          }
        }
        if (noise) diag(S) <- diag(S) + 1
        return(S)
      }
    }
  }
  if (is.null(param)) param <- formals(cov_fct)$param
  attr(f, 'param') <- param

  # Set type
  attr(f, 'type') <- attr(cov_fct, 'type')
  if (attr(f, 'type') == 'Brownian') {
    args <- list(...)
    exist <- "type" %in% names(args)
    if (exist) {
      attr(f,   'type')   <-   attr(cov_fct,   'type')   <-
paste(attr(cov_fct, 'type'), args$type)
    } else {
      attr(f, 'type') <- attr(cov_fct, 'type') <- 'Brownian
motion'
    }
  }
  # Set solve method
  attr(f, 'inv_cov_fct') <- attr(cov_fct, 'inv_cov_fct')
  # Set the scale parameter
  attr(f,                    'scale')                   <-
ifelse(any(names(formals(cov_fct)$param)   ==   'scale'),
which(names(formals(cov_fct)$param) == 'scale') - 1, NA)
  # Include covariance function evaluated with additional
arguments
  eval_cov_fct <- function(t, param) cov_fct(t, param, ...)
  attributes(eval_cov_fct) <- attributes(cov_fct)
  attr(f, 'cov_fct') <- eval_cov_fct
  attr(f, 'noise') <- noise
  # If the covariance has been made non-stationary, modify
  if (!is.null(ns)) {
    fixed    <-    ifelse(!is.null(ns$fixed),    ns$fixed,
max(knots))
    m_ns <- knots - length(fixed)

    attr(f, 'param') <- c(local_scale = rep(1, m_ns), param)
    ns_cov_fct <- function(t, param) {
```

```
      ns_param <- rep(1, length = knots)
      ns_param[-fixed] <- param[1:m_ns]
      prod(spline(seq(ns$range[1], ns$range[2], length =
ns$knots), ns_param, xout = t)$y) *
        cov_fct(abs(diff(t)), param[-(1:m_ns)])
    }
    attr(ns_cov_fct, 'stationary') <- FALSE
    attr(f, 'cov_fct') <- ns_cov_fct
  }
  return(f)
}



################### Matern covariance function########
Matern <- function(d, param = c(scale = 1, range = 1,
smoothness = 2)) {
  scale <- param[1]
  range <- param[2]
  smoothness <- param[3]
  if (any(d < 0))
    stop("distance argument must be nonnegative")
  d <- d / range
  d[d == 0] <- 1e-10
  con <- (2^(smoothness - 1)) * gamma(smoothness)
  con <- 1 / con
  return(scale * con * (d^smoothness) * besselK(d,
smoothness))
}
attr(Matern, 'stationary') <- TRUE
attr(Matern, 'type') <- 'Matern'



################## make_warp_fct() ###################
make_warp_fct <- function(type = c('shift', 'linear',
'piecewise-linear', 'smooth'), tw = NULL, control =
list(wleft = 'fixed', wright = 'fixed')) {
  # Match type argument
  if (length(type) > 1) type <- type[1]
  types <- c('shift', 'linear', 'piecewise-linear',
'smooth')
  type <- types[pmatch(type, types)]
  if (is.null(tw)) tw <- NA
```

```
  # If boundary control is missing, assume fixed
  if (is.null(control$wleft)) control$wleft <- 'fixed'
  if (is.null(control$wright)) control$wright <- 'fixed'


  if (type == 'shift') {
    v <- function(w, t, w_grad = FALSE) {
      if (!w_grad) {
        return(w + t)
      } else {
        return(matrix(1, length(t), 1))
      }
    }
    attr(v, 'mw') <- 1
  } else if (type == 'linear') {
    v <- function(w, t, w_grad = FALSE) {
      if (!w_grad) {
        return(w[1] + (w[2] + 1) * t)
      } else {
        dv <- matrix(1, length(t), 2)
        dv[, 2] <- t
        return(dv)
      }
    }
    attr(v, 'mw') <- 2
  } else if (type == 'piecewise-linear') {
    if (any(is.na(tw))) stop('all anchor points tw should be
supplied for type \'piecewise-linear\'')
    mw <- length(tw)
    v <- function(w1, w2, t, w_grad = FALSE) {
      w <- w1 + w2
      if (!w_grad) {
        vt <- t + approx(tw, c(ifelse(control$wleft == 'fixed',
0, w[1]),
                               w,
                               ifelse(control$wright == 'fixed',
0, w[length(w)])),
                         xout = t, rule = 2)$y
        return(vt)
      } else {
        # Derivative of warp function
        # Note: does not depend on w
        dv <- apply(cbind(tw[1:(mw - 2)], tw[2:(mw - 1)],
```

```
tw[3:mw]), 1, function(x) {
        a <- rep(0, length(t))
        a[t > x[1] & t < x[2]] <- ((t - x[1]) / (x[2] - x[1]))[t >
x[1] & t < x[2]]
        a[t >= x[2] & t < x[3]] <- (1 - ((t - x[2]) / (x[3]
- x[2])))[t >= x[2] & t < x[3]]
        return(a)
       })
       if (control$wleft != 'fixed') dv[t < tw[2], 1] <- 1
       if (control$wright != 'fixed') dv[t > tw[mw - 1], mw
- 2] <- 1
       return(dv)
     }
   }
   attr(v, 'mw') <- mw - 2
  } else if (type == 'smooth') {
   if (any(is.na(tw))) stop('all anchor points tw should be
supplied for type \'smooth\'')
   mw <- length(tw)
   # Hyman spline warping function
   v <- function(w1, w2, t, w_grad = FALSE) {
     w <- w1 + w2
     y <- tw + c(ifelse(control$wleft == 'fixed', 0, w[1]),
w, ifelse(control$wright == 'fixed', 0, w[length(w)]))
     if (!all(diff(y) > 0)) {
       w <- make_homeo(w, tw, epsilon = 0.2)
       y <- tw + c(ifelse(control$wleft == 'fixed', 0, w[1]),
w, ifelse(control$wright == 'fixed', 0, w[length(w)]))
     }
     if (!w_grad){
       return(spline(tw, y, xout = t, method = 'hyman')$y)
     } else {
       # Derivative of warp function
       # Finite difference, could we do better?
       epsilon <- 1e-5
       m <- length(t)
       dv <- matrix(0, m, mw - 2)
       for (j in 1:(mw - 2)) {
         h_tmp <- rep(0, mw)
         if (j == 1 & control$wleft != 'fixed') h_tmp[j] <-
epsilon
         if (j == (mw - 2) & control$wright != 'fixed') h_tmp[j
+ 2] <- epsilon
         h_tmp[j + 1] <- epsilon
```

```
        dv[, j] <- (spline(tw, y + h_tmp, xout = t, method
= 'hyman')$y
                    - spline(tw, y - h_tmp, xout = t, method
= 'hyman')$y) / (2 * epsilon)
       }
       return(dv)
     }
   }
   attr(v, 'mw') <- mw - 2
 }
 attr(v, 'tw') <- tw
 attr(v, 'type') <- type
 return(v)
}



############### make_homeo() ########################
make_homeo <- function(w, tw, epsilon = 0.1) {
mw <- length(w)
ui <- diag(1, nrow = mw)
ui <- rbind(ui, rep(0, mw))
ui[cbind(2:(mw + 1), 1:mw)] <- -1

ci <- (epsilon - 1) * diff(tw)
res <- constrOptim_inf(theta = rep(0, mw), f = function(x)
mean((x - w)^2),
                       grad = function(x) 2 / mw * (x - w), ui
= ui, ci = ci, method = "BFGS")

return(res$par)
}



########################resp_gene()##################
resp_gene <- function(true_data, t, betat1, betat2, ng, b0,
b1, b2, aa1, aa2, bb1, bb2, check = "T"){

 Align_train1 <- true_data[[5]];

 W1 <- Align_train1[[1]]%*% betat1/(length(t))
 W2 <- Align_train1[[2]]%*% betat2/(length(t))
```

```
  x1 <- c(runif((ng/2),aa1,aa2), runif((ng/2),bb1,bb2))
  x2 <- c(runif((ng/2),aa1,aa2), runif((ng/2),bb1,bb2))


  Z <- b0 + x1*b1 + W1 + W2;

  P <- 1/(1+exp(-Z))
  y <- rep(0,ng)
  for (i in 1:ng){
    y[i] <- rbinom(1, 1, P[i])
  }


  return(list(x1 = x1, x2 = x2, Z = Z, P = P, Res = y))
}



#################### pavpop_fixrandwarp() ###############
pavpop_fixrandwarp<- function(ya, yb, t, K, basis_fct,
warp_fct, amp_cov = NULL, warp_cov = NULL, amp_fct = NULL,
                          warped_amp = FALSE, iter = c(5, 5),
parallel = list(n_cores = 1, parallel_likelihood = FALSE),
                          use_warp_gradient    =    FALSE,
warp_optim_method   =   'CG',   homeomorphisms   =   'no',
like_optim_control = list()) {

  ya1 = ya[K[[1]]]
  ya2 = ya[K[[2]]]
  yb1 = yb[K[[1]]]
  yb2 = yb[K[[2]]]
  l1 = length(K[[1]])
  l2 = length(K[[2]])

  if (!is.null(amp_fct) & is.null(amp_cov))
    warning('Amplitude  variation  basis  ignored  when  no
amplitude covariance is specified.')

  warp_centering <- TRUE
  # If the warps are not regularized, we should be careful
when centering.
  if (is.null(warp_cov)) warp_centering <- FALSE
```

```
  nouter <- iter[1] + 1
  if (is.null(amp_cov) & is.null(warp_cov)) nouter <- 1
  ninner <- iter[2]
  halt_iteration <- FALSE

  # Set size parameters
  n <- length(ya)
  m <- sapply(ya, length)

  # Warp parameters
  tw <- attr(warp_fct, 'tw')
  mw <- attr(warp_fct, 'mw')
  if (all(is.na(tw))) tw <- rep(tw, mw + 2)
  tw_int <- tw[2:(mw + 1)]

  warp_type <- attr(warp_fct, 'type')
  if (warp_type != 'piecewise linear' & warp_type != 'smooth')
homeomorphisms <- 'no'

  # Unknown parameters
  amp_cov_par <- eval(attr(amp_cov, 'param'))
  warp_cov_par <- eval(attr(warp_cov, 'param'))
  n_par_amp <- length(amp_cov_par)
  n_par_warp <- length(warp_cov_par)

  # Check for same data structures of y and t
  #if (length(t) != n) stop("y and t must have same length.")
  #if (!all(sapply(t, length) == m)) stop("Observations in
y and t must have same length.")

  # Remove missing values
  #for (i in 1:n) {
  #missing_indices <- is.na(y[[i]])
  #y[[i]] <- y[[i]][!missing_indices]
  #t[[i]] <- t[[i]][!missing_indices]
  #}
  # Stored warped time
  t_warped <- t

  # Update m with cleaned data
  m <- sapply(ya, length)

  # Initialize cluster
```

```
  doParallel::registerDoParallel(cores = parallel$n_cores)

  # Initialize warp parameters
  w_fix1 <- array(0, dim = c(mw, 1))#fixed warp parameters
  w_ran1 <- array(0, dim = c(mw, l1))#random warp parameters

  w_fix2 <- array(0, dim = c(mw, 1))#fixed warp parameters
  w_ran2 <- array(0, dim = c(mw, l2))#random warp parameters

  # Compute amplitude precision matrices
  Sinv <- fill_precision(t, amp_cov, amp_cov_par, amp_fct)


  # Build warp covariance and inverse
  if (!is.null(warp_cov)) {
    C <- warp_cov(tw_int, warp_cov_par)
    Cinv <- solve(C)
  } else {
    C <- Cinv <- matrix(0, mw, mw)
  }

  # Estimate spline weights
  c_a <- spline_weights(ya, t, Sinv, basis_fct)
  d_a1 <- spline_weights_classpecific(ya[K[[1]]], t[K[[1]]],
c_a, Sinv[K[[1]]], basis_fct)
  d_a2 <- spline_weights_classpecific(ya[K[[2]]], t[K[[2]]],
c_a, Sinv[K[[2]]], basis_fct)

  c_b <- spline_weights(yb, t, Sinv, basis_fct)
  d_b1 <- spline_weights_classpecific(yb[K[[1]]], t[K[[1]]],
c_b, Sinv[K[[1]]], basis_fct)
  d_b2 <- spline_weights_classpecific(yb[K[[2]]], t[K[[2]]],
c_b, Sinv[K[[2]]], basis_fct)
  # Construct warp derivative
  dwarp <- list()
  if (warp_type != 'smooth') {
    for (i in 1:n) {
      if (i <= l1)
        dwarp[[i]] <- warp_fct(w_fix1, w_ran1[, i], t[[i]],
w_grad = TRUE)
      else
        dwarp[[i]] <- warp_fct(w_fix2, w_ran2[, i-l1], t[[i]],
w_grad = TRUE)
      if (warp_type == 'piecewise linear') dwarp[[i]] <-
```

```
as(dwarp[[i]], "dgCMatrix")
    }
  }

  # Initialize best parameters
  like_best <- Inf
  w_fix1_best <- w_fix1
  w_ran1_best <- w_ran1
  w_fix2_best <- w_fix2
  w_ran2_best <- w_ran2
  c_a_best <- c_a
  d_a1_best <- d_a1
  d_a2_best <- d_a2
  c_b_best <- c_b
  d_b1_best <- d_b1
  d_b2_best <- d_b2
  amp_cov_par_best <- amp_cov_par
  warp_cov_par_best <- warp_cov_par

  cat('Outer\t:\tInner \t:\tEstimates\n')
  for (iouter in 1:nouter) {
    if (halt_iteration & iouter != nouter) next
    # Outer loop
    if (iouter != nouter) cat(iouter, '\t:\t')
    for (iinner in 1:ninner) {
      # Inner loop
      if (iouter != nouter | nouter == 1) cat(iinner, '\t')

      # Predict warping parameters for all functional samples
      warp_change1 <- c(0, 0)
      warp_change2 <- c(0, 0)
      if (homeomorphisms == 'hard') {
        #TODO: constrainOptim
        stop("Hard homeomorphic constrained optimization for
warps is not implemented.")
      } else {
        # Estimate the fixed warping effect
        ww_fix1 <- optim(par = w_fix1, fn = posterior_fix, gr
= NULL, method = warp_optim_method, warp_fct = warp_fct, t
= t[K[[1]]], y1 = ya[K[[1]]], y2 = yb[K[[1]]], w2 = w_ran1,
c1 = c_a + d_a1, c2 = c_b + d_b1, Sinv = Sinv[K[[1]]], basis_fct
= basis_fct)$par
        ww_fix2 <- optim(par = w_fix2, fn = posterior_fix, gr
= NULL, method = warp_optim_method, warp_fct = warp_fct, t
```

```
= t[K[[2]]], y1 = ya[K[[2]]], y2 = yb[K[[2]]], w2 = w_ran2,
c1 = c_a + d_a2, c2 = c_b + d_b2, Sinv = Sinv[K[[2]]], basis_fct
= basis_fct)$par
       # Parallel prediction of warping parameters
       w_res1 <- foreach(i = 1:l1) %do% {
         gr <- NULL
         ww_ran1 <- optim(par = w_ran1[, i], fn =
posterior_rand, gr = gr, method = warp_optim_method, warp_fct
= warp_fct, t = t[K[[1]]][[i]], y1 = ya[K[[1]]][[i]], y2 =
yb[K[[1]]][[i]], w1 = ww_fix1, c1 = c_a + d_a1, c2 = c_b +
d_b1, Sinv = Sinv[K[[1]]][[i]], Cinv = Cinv, basis_fct =
basis_fct)$par
         if (homeomorphisms == 'soft') ww_ran1 <-
make_homeo(ww_ran1, tw)
         return(ww_ran1)
       }
       w_res2 <- foreach(i = 1:l2) %do% {
         gr <- NULL
         ww_ran2 <- optim(par = w_ran2[, i], fn =
posterior_rand, gr = gr, method = warp_optim_method, warp_fct
= warp_fct, t = t[K[[2]]][[i]], y1 = ya[K[[2]]][[i]], y2 =
yb[K[[2]]][[i]], w1 = ww_fix2, c1 = c_a + d_a2, c2 = c_b +
d_b2, Sinv = Sinv[K[[2]]][[i]], Cinv = Cinv, basis_fct =
basis_fct)$par
         if (homeomorphisms == 'soft') ww_ran2 <-
make_homeo(ww_ran2, tw)
         return(ww_ran2)
       }


       for (i in 1:l1) {
         warp_change1[1] <- warp_change1[1] + sum((w_fix1 +
w_ran1[, i] - ww_fix1 - w_res1[[i]])^2)
         warp_change1[2] <- max(warp_change1[2], abs(w_fix1
+ w_ran1[, i] - ww_fix1 - w_res1[[i]]))
       }
       for (i in 1:l2){
         warp_change2[1] <- warp_change2[1] + sum((w_fix2 +
w_ran2[, i] - ww_fix2 - w_res2[[i]])^2)
         warp_change2[2] <- max(warp_change2[2], abs(w_fix2
+ w_ran2[, i] - ww_fix2 - w_res2[[i]]))
       }

       # Fix warps that are considerably beyond
```

```
basis-function endpoints
       # Pre-compute warped time
       t_warped1      <-      lapply(1:l1,      function(i)
warp_fct(ww_fix1, w_res1[[i]], t[K[[1]]][[i]]))
       t_warped2      <-      lapply(1:l2,      function(i)
warp_fct(ww_fix2, w_res2[[i]], t[K[[2]]][[i]]))
       t_warped <- append(t_warped1, t_warped2)
       max_outside <- 0.75
       outside_frac1   <-   sapply(t_warped1,   function(t)
mean(attr(basis_fct, 'boundary')[1] < t & t < attr(basis_fct,
'boundary')[2]))
       outside_frac2   <-   sapply(t_warped2,   function(t)
mean(attr(basis_fct, 'boundary')[1] < t & t < attr(basis_fct,
'boundary')[2]))
       for  (i  in  which(outside_frac1  <  max_outside))
w_res1[[i]][] <- 0
       for  (i  in  which(outside_frac2  <  max_outside))
w_res2[[i]][] <- 0

       for (i in 1:l1) w_ran1[, i] <- w_res1[[i]]
       for (i in 1:l2) w_ran2[, i] <- w_res2[[i]]
       w_fix1 <- ww_fix1
       w_fix2 <- ww_fix2

       # Center warps
       if (warp_centering & iinner != ninner) {
         w_ran1 <- w_ran1 - rowMeans(w_ran1)
         w_ran2 <- w_ran2 - rowMeans(w_ran2)
       }
     }
     # Update precisions, if
     if (warped_amp){
       Sinv <- fill_precision(t_warped, amp_cov, amp_cov_par,
amp_fct)
     }

     # Update spline weights
     c_a <- spline_weights(ya, t_warped, Sinv, basis_fct)
     d_a1    <-    spline_weights_classpecific(ya[K[[1]]],
t_warped[K[[1]]], c_a, Sinv[K[[1]]], basis_fct)
     d_a2    <-    spline_weights_classpecific(ya[K[[2]]],
t_warped[K[[2]]], c_a, Sinv[K[[2]]], basis_fct)
     c_b <- spline_weights(yb, t_warped, Sinv, basis_fct)
     d_b1    <-    spline_weights_classpecific(yb[K[[1]]],
```

```
t_warped[K[[1]]], c_b, Sinv[K[[1]]], basis_fct)
     d_b2    <-    spline_weights_classpecific(yb[K[[2]]],
t_warped[K[[2]]], c_b, Sinv[K[[2]]], basis_fct)
     if (max(warp_change1[2], warp_change1[2]) < 1e-2 /
sqrt(mw)) break #TODO: Consider other criteria
   }

   #
   # Likelihood estimation of parameters (outer loop)
   #

   # Construct residual vector for given warp prediction
   if (warp_type == 'smooth') {
     for (i in 1:n) #dwarp[[i]] <- warp_fct(w1, w2[, i],
t[[i]], w_grad = TRUE)
     {
       if (i <= l1) dwarp[[i]] <- warp_fct(w_fix1, w_ran1[,
i], t[[i]], w_grad = TRUE)
       else
         dwarp[[i]]  <-  warp_fct(w_fix2,  w_ran2[,  i-l1],
t[[i]], w_grad = TRUE)
     }
   }
   if (!is.null(warp_cov)) {
     Zis_a1   <-   Zis(t_warped[K[[1]]],   dwarp[K[[1]]],
basis_fct, c_a + d_a1)
     Zis_a2   <-   Zis(t_warped[K[[2]]],   dwarp[K[[2]]],
basis_fct, c_a + d_a2)
     Zis_a <- append(Zis_a1, Zis_a2)
     Zis_b1   <-   Zis(t_warped[K[[1]]],   dwarp[K[[1]]],
basis_fct, c_b + d_b1)
     Zis_b2   <-   Zis(t_warped[K[[2]]],   dwarp[K[[2]]],
basis_fct, c_b + d_b2)
     Zis_b <- append(Zis_b1, Zis_b2)
   } else {
     Zis_a <- lapply(m, function(m) Matrix::Matrix(0, m,
mw))
     Zis_b <- lapply(m, function(m) Matrix::Matrix(0, m,
mw))
   }
   ra <- ya
   rb <- yb
   for (i in 1:n) {
     if (i <= l1) {
```

```
      ra[[i]]       <-       as.numeric(ra[[i]]       -
basis_fct(t_warped[[i]]) %*% (c_a + d_a1) + Zis_a1[[i]] %*%
w_ran1[, i])
      rb[[i]]       <-       as.numeric(rb[[i]]       -
basis_fct(t_warped[[i]]) %*% (c_b + d_b1) + Zis_b1[[i]] %*%
w_ran1[, i])
    }
    else {
      ra[[i]]       <-       as.numeric(ra[[i]]       -
basis_fct(t_warped[[i]]) %*% (c_a + d_a2) + Zis_a2[[i-l1]] %*%
w_ran2[, i-l1])
      rb[[i]]       <-       as.numeric(rb[[i]]       -
basis_fct(t_warped[[i]]) %*% (c_b + d_b2) + Zis_b2[[i-l1]] %*%
w_ran2[, i-l1])
    }
  }

  # Check wheter the final outer loop has been reached
  if (iouter != nouter) {
    t_like <- t
    if (warped_amp) t_like <- t_warped
    # Likelihood function
    like_fct <- function(par) {
      if (!is.null(amp_fct)) {
        like_amp(par, n_par = c(n_par_amp, n_par_warp), r =
r, Zis = Zis, amp_cov = amp_cov, warp_cov = warp_cov, amp_fct
= amp_fct, t = t_like, tw = tw_int)
      } else {
        like(par, n_par = c(n_par_amp, n_par_warp), ra = ra,
rb = rb, Zis_a = Zis_a, Zis_b = Zis_b, amp_cov = amp_cov,
warp_cov = warp_cov, t = t_like, tw = tw_int)
      }
    }

    # Likelihood gradient
    like_gr <- NULL
    if (parallel$parallel_likelihood) {
      # Construct parallel gradient
      like_gr <- function(par) {
        epsilon <- 1e-5
        rep(1:length(par), each = 2)
        res <- foreach(i = 1:length(par), .combine = 'c') %:%
          foreach(sign = c(1, -1), .combine= '-') %dopar% {
            h <- rep(0, length(par))
```

```
            h[i] <- sign * epsilon
            return(like_fct(par + h) / (2 * epsilon))
          }
        return(res)
      }
    } else {
      # Construct parallel gradient
      like_gr <- function(par) {
        epsilon <- 1e-5
        rep(1:length(par), each = 2)
        res <- rep(0, length(par))
        for (i in  1:length(par)) {
          for (sign in c(1, -1)) {
            h <- rep(0, length(par))
            h[i] <- sign * epsilon
            res[i] <- res[i] + sign * like_fct(par + h) / (2
* epsilon)
          }
        }
        return(res)
      }
    }

    #  Estimate  parameters  using  locally  linearized
likelihood
    lower    <-  if  (is.null(like_optim_control$lower))
rep(1e-3,      n_par_amp      +      n_par_warp)      else
like_optim_control$lower
    upper    <-  if  (is.null(like_optim_control$upper))
rep(Inf,      n_par_amp      +      n_par_warp)      else
like_optim_control$upper
    method  <-  if  (is.null(like_optim_control$method))
"L-BFGS-B" else like_optim_control$method
    ndeps   <-   if   (is.null(like_optim_control$ndeps))
rep(1e-3,      n_par_amp      +      n_par_warp)      else
like_optim_control$ndeps

    like_optim  <-  optim(c(amp_cov_par,  warp_cov_par),
like_fct, gr = like_gr, method = method, lower = lower, upper
= upper, control = list(ndeps = ndeps, maxit = 20))
    param <- like_optim$par

    if (!is.null(amp_cov)) {
      #  Handle  type  'unstr_cov'  which  can  be  used  in
```

```
combination with an
       # amplitude function. In this case, the found variance
parameters m
       # may not correspond to the nearest positive definite
matrix that
       # was actually used in computations
       if (!is.null(amp_fct) & attr(amp_cov, 'type') ==
'unstr_cov') {
         amp_cov_mat       <-       warp_cov(1:n_par_amp,
param[1:n_par_amp])
         amp_cov_par       <-       c(diag(amp_cov_mat),
warp_cov_mat[upper.tri(amp_cov_mat)])
       } else {
         amp_cov_par <- param[1:n_par_amp]
       }
     }
     if (!is.null(warp_cov)) {
       # Handle type 'unstr_cov' where found parameters may
not correspond to the nearest
       # positive definite matrix that was actually used in
computations
       if (attr(warp_cov, 'type') == 'unstr_cov') {
         warp_cov_mat <- warp_cov(tw_int, param[(n_par_amp +
1):length(param)])
         warp_cov_par       <-       c(diag(warp_cov_mat),
warp_cov_mat[upper.tri(warp_cov_mat)])
       } else {
         warp_cov_par      <-      param[(n_par_amp      +
1):length(param)]
       }
     }


     if (like_optim$value <= like_best) {
       # Save parameters
       like_best <- like_optim$value
       w_fix1_best <- w_fix1
       w_ran1_best <- w_ran1
       w_fix2_best <- w_fix2
       w_ran2_best <- w_ran2
       c_a_best <- c_a
       d_a1_best <- d_a1
       d_a2_best <- d_a2
       c_b_best <- c_b
```

```
      d_b1_best <- d_b1
      d_b2_best <- d_b2
      amp_cov_par_best <- amp_cov_par
      warp_cov_par_best <- warp_cov_par

      # Update covariances
      Sinv <- fill_precision(t, amp_cov, amp_cov_par,
amp_fct)

      if (!is.null(warp_cov)) {
        C <- warp_cov(tw_int, warp_cov_par)
        Cinv <- solve(C)
      }

      cat(':\t', param, '\n')
      cat('Linearized likelihood:\t', like_best, '\n')
    } else {
      cat('.\nLikelihood not improved, returning best
likelihood estimates.\n')
      halt_iteration <- TRUE
    }
  } else {
    # TODO: Should in principle be done before warps are
updated in the final iteration!
    # Estimate of sigma if final iteration is reached
    if (nouter == 1) {
      w_fix1_best <- w_fix1
      w_ran1_best <- w_ran1
      w_fix2_best <- w_fix2
      w_ran2_best <- w_ran2
      c_a_best <- c_a
      d_a1_best <- d_a1
      d_a2_best <- d_a2
      c_b_best <- c_b
      d_b1_best <- d_b1
      d_b2_best <- d_b2
    }

    t_like <- t
    if (warped_amp) t_like <- t_warped

    if (!is.null(amp_fct)) {
      sigma <- sqrt(sigmasq_amp(c(amp_cov_par,
warp_cov_par), c(n_par_amp, n_par_warp), r, Zis, amp_cov,
```

```
warp_cov, amp_fct, t_like, tw_int))
     } else {
       sigma <- sqrt(sigmasq(c(amp_cov_par, warp_cov_par),
c(n_par_amp, n_par_warp), ra, rb, Zis_a, Zis_b, amp_cov,
warp_cov, t_like, tw_int))
     }
    }
  }
  return(list(c_a = c_a_best, d_a1 = d_a1_best, d_a2 =
d_a2_best, c_b = c_b_best, d_b1 = d_b1_best, d_b2 = d_b2_best,
w_fix1 = w_fix1_best, w_ran1 = w_ran1_best, w_fix2 =
w_fix2_best, w_ran2 = w_ran2_best, amp_cov_par =
amp_cov_par_best, warp_cov_par = warp_cov_par_best, sigma =
sigma, like = like_best))
}



################### constrOptim_inf() ###############
constrOptim_inf  <-
  function(theta, f, grad, ui, ci, mu = 0.0001, control =
list(),
           method  =  if(is.null(grad))  "Nelder-Mead"  else
"BFGS",
           outer.iterations = 100, outer.eps = 0.00001, ...,
hessian = FALSE)
  {

    if (!is.null(control$fnscale) && control$fnscale < 0)
      mu <- -mu ##maximizing

    R <- function(theta, theta.old, ...) {
      ui.theta <- ui%*%theta
      gi <-  ui.theta-ci
      if (any(gi<0)) return(sign(mu) * Inf)
      gi.old <- ui%*%theta.old-ci
      bar <- sum(gi.old*log(gi)-ui.theta)
      if (!is.finite(bar)) bar <-  -Inf
      f(theta, ...) -mu*bar
    }

    dR <- function(theta, theta.old, ...) {
      ui.theta <- ui%*%theta
      gi <- drop(ui.theta-ci)
```

```
      gi.old <- drop(ui%*%theta.old-ci)
      dbar <- colSums(ui*gi.old/gi-ui)
      grad(theta, ...) - mu*dbar
    }

    if (any(ui%*%theta-ci <= 0))
      stop("initial value is not in the interior of the
feasible region")
    obj <- f(theta, ...)
    r <- R(theta, theta, ...)
    fun <- function(theta, ...) R(theta, theta.old, ...)
    gradient <- if(method == "SANN") {
      if(missing(grad)) NULL else grad
    } else function(theta, ...) dR(theta, theta.old, ...)
    totCounts <- 0
    s.mu <- sign(mu)

    for(i in seq_len(outer.iterations)) {
      obj.old <- obj
      r.old <- r
      theta.old <- theta

      a <- optim(theta.old, fun, gradient, control = control,
                 method = method, hessian = hessian, ...)
      r <- a$value
      if (is.finite(r) && is.finite(r.old) &&
          abs(r - r.old) < (1e-3 + abs(r)) * outer.eps) break
      theta <- a$par
      totCounts <- totCounts + a$counts
      obj <- f(theta, ...)
      if (s.mu * obj > s.mu * obj.old) break
    }
    if (i == outer.iterations) {
      a$convergence <- 7
      a$message <- gettext("Barrier algorithm ran out of
iterations and did not converge")
    }
    if (mu > 0 && obj > obj.old) {
      a$convergence <- 11
      a$message <- gettextf("Objective function increased at
outer iteration %d", i)
    }
    if (mu < 0 && obj < obj.old) {
      a$convergence <- 11
```

```
     a$message <- gettextf("Objective function decreased at
outer iteration %d", i)
    }
    a$outer.iterations <- i
    a$counts <- totCounts
    a$barrier.value <- a$value
    a$value <- f(a$par, ...)
    a$barrier.value <- a$barrier.value - a$value
    a
  }


################### Aligned_curves() ###############
Aligned_curves <- function(ya,yb,t,K0,res){
  n1 = length(K0[[1]]);
  n2 = length(K0[[2]]);
  Aligned_ya1 <-  Aligned_yb1 <- matrix(0, nrow = n1, ncol =
length(t[[1]]));
  Aligned_ya2 <-  Aligned_yb2 <- matrix(0, nrow = n2, ncol =
length(t[[1]]));

  Raw_ya1 <-   Raw_yb1 <-  matrix(0,  nrow  =  n1,  ncol  =
length(t[[1]]));
  Raw_ya2 <-   Raw_yb2 <-  matrix(0,  nrow  =  n2,  ncol  =
length(t[[1]]));

  for (i in 1:n1) {
    Aligned_ya1[i,]     <-     approx(warp_fct(res$w_fix1,
res$w_ran1[,i], t[[i]]), ya[[K0[[1]][i]]], xout = t[[i]],
rule = 2)$y
    Aligned_yb1[i,]     <-     approx(warp_fct(res$w_fix1,
res$w_ran1[,i], t[[i]]), yb[[K0[[1]][i]]], xout = t[[i]],
rule = 2)$y
    Raw_ya1[i,]   <-   ya[[K0[[1]][i]]];   Raw_yb1[i,]   <-
yb[[K0[[1]][i]]];
  }

  for (i in 1:n2) {
    Aligned_ya2[i,]     <-     approx(warp_fct(res$w_fix2,
res$w_ran2[,i], t[[i]]), ya[[K0[[2]][i]]], xout = t[[i]],
rule = 2)$y
    Aligned_yb2[i,]     <-     approx(warp_fct(res$w_fix2,
res$w_ran2[,i], t[[i]]), yb[[K0[[2]][i]]], xout = t[[i]],
rule = 2)$y
```

```
   Raw_ya2[i,]   <-   ya[[K0[[2]][i]]];   Raw_yb2[i,]   <-
yb[[K0[[2]][i]]];
  }


  return(list(Aligned_ya1  =  Aligned_ya1,  Aligned_yb1  =
Aligned_yb1,  Aligned_ya2  =  Aligned_ya2,  Aligned_yb2  =
Aligned_yb2, Raw_ya1 = Raw_ya1, Raw_yb1 = Raw_yb1, Raw_ya2
= Raw_ya2, Raw_yb2 = Raw_yb2))

}



################### precision function##############
fill_precision <- function(t, cov_fct, param, amp_fct = NULL)
{
  n <- length(t)
  m <- sapply(t, length)

  # Check if covariance function is specified
  if (!is.null(cov_fct)) {
    # Check if functional basis for the amplitude variation
is supplied
    if (!is.null(amp_fct)) {
      df <- attr(amp_fct, 'df')
      SSinv <- chol2inv(chol(cov_fct(1:df, param)))

      # Fill precision matrices
      Sinv <- list()
      for (i in 1:n) {
        A <- amp_fct(t[[i]])
        Sinv[[i]] <- diag(1, m[i]) - A %*% solve(SSinv + t(A) %*%
A, t(A))
      }
      return(Sinv)
    } else {
      # No functional basis specified, invert covariance
matrices

      # Check if inverse method is given
      if (!is.null(attr(cov_fct, 'inv_cov_fct'))) {
        lapply(t,  attr(cov_fct,  'inv_cov_fct'),  param  =
param)
```

```
    } else {
      # No inverse method given, manually invert
      lapply(lapply(t, cov_fct, param = param), function(x)
chol2inv(chol(x)))
    }
   }
  } else {
   # No covariance specified, assuming iid. Gaussian noise
   lapply(m, Matrix::Diagonal, x = 1)
  }
}



##################### spline_weights() #############
spline_weights <- function(y, t, Sinv = NULL, basis_fct,
weights = NULL) {
  if (class(y) != 'list') {
   t <- as.matrix(t)
   y <- as.matrix(y)
   t <- lapply(1:ncol(t), function(i) t[, i])
   y <- lapply(1:ncol(y), function(i) y[, i])
  }

  n <- length(y)
  m <- sapply(y, length)

  nb <- attr(basis_fct, 'df')

  if (is.null(weights)) weights <- rep(1, n)

  Dmat <- matrix(0, nb, nb)
  dvec <- matrix(0, nb, 1)

  # If no precision matrix is supplied, use identity
  if (is.null(Sinv)) Sinv <- lapply(m, Matrix::Diagonal, x
= 1)

  for (i in 1:n) {
   basis <- basis_fct(t[[i]])
   bSinv <- weights[i] * (t(basis) %*% Sinv[[i]])
   Dmat <- Dmat + bSinv %*% basis
   dvec <- dvec + bSinv %*% y[[i]]
  }
```

```
  if (attr(basis_fct, 'constraints') == 'positive') {
    qrDmat <- qr(Dmat)
    indices <- qrDmat$pivot[1:qrDmat$rank]

    c <- rep(0, ncol(basis))
    c[indices] <- quadprog::solve.QP(Dmat = Dmat[indices,
indices],
                                     dvec = dvec[indices,],
                                     Amat    =    diag(nrow    =
length(indices)))$solution
  } else {
    # Faster to use qr?
    c <- as.numeric(MASS::ginv(as.matrix(Dmat)) %*% dvec)
  }

  return(c)
}



############### spline_weights_classpecific() #########
spline_weights_classpecific <- function(y, t, cc, Sinv = NULL,
basis_fct, weights = NULL) {
  if (class(y) != 'list') {
    t <- as.matrix(t)
    y <- as.matrix(y)
    t <- lapply(1:ncol(t), function(i) t[, i])
    y <- lapply(1:ncol(y), function(i) y[, i])
  }

  n <- length(y)
  m <- sapply(y, length)

  nb <- attr(basis_fct, 'df')

  if (is.null(weights)) weights <- rep(1, n)

  Dmat <- matrix(0, nb, nb)
  dvec <- matrix(0, nb, 1)

  # If no precision matrix is supplied, use identity
  if (is.null(Sinv)) Sinv <- lapply(m, Matrix::Diagonal, x
= 1)
```

```
  for (i in 1:n) {
    basis <- basis_fct(t[[i]])
    bSinv <- weights[i] * (t(basis) %*% Sinv[[i]])
    Dmat <- Dmat + bSinv %*% basis
    dvec <- dvec + bSinv %*% (y[[i]] - basis %*% cc)
  }

  if (attr(basis_fct, 'constraints') == 'positive') {
    qrDmat <- qr(Dmat)
    indices <- qrDmat$pivot[1:qrDmat$rank]

    c <- rep(0, ncol(basis))
    c[indices] <- quadprog::solve.QP(Dmat = Dmat[indices,
indices],
                                     dvec = dvec[indices,],
                                     Amat    =    diag(nrow    =
length(indices)))$solution
  } else {
    # Faster to use qr?
    c <- as.numeric(MASS::ginv(as.matrix(Dmat)) %*% dvec)
    c <- c - mean(c)
  }

  return(c)
}



###################### Zis() ########################
Zis <- function(t, dwarp, basis_fct, c) {
  n <- length(t)
  m <- sapply(t, length)
  mw <- ncol(dwarp[[1]])
  Zis <- list()
  for (i in 1:n) {
    Zis[[i]] <- matrix(Zi(t[[i]], dwarp[[i]], basis_fct, c),
m[i], mw)
  }
  return(Zis)
}
```

```
###################### likelihood functions ######
like <- function(param, n_par, ra, rb, Zis_a, Zis_b, amp_cov,
warp_cov, t, tw) {
  amp_cov_par <- param[1:n_par[1]]
  warp_cov_par <- param[(n_par[1] + 1):length(param)]
  if (!is.null(warp_cov)) {
    C <- warp_cov(tw, warp_cov_par)
    Cinv <- chol2inv(chol(C))
  } else {
    C <- Cinv <- matrix(0, length(tw), length(tw))
  }

  n <- length(ra)
  m <- sapply(ra, length)

  sq_a <- logdet_a <- 0
  sq_b <- logdet_b <- 0
  for (i in 1:n) {
    if (!is.null(amp_cov)) {
      S <- amp_cov(t[[i]], amp_cov_par)
      U <- chol(S)
    } else {
      # TODO: SPARSE MATRIX COULD MAKE IT FASTER, THEN 'diag'
cannot be used for logdet
      S <- U <- diag(1, m[i])
    }
    rra <- ra[[i]]
    ZZa <- Zis_a[[i]]
    rrb <- rb[[i]]
    ZZb <- Zis_b[[i]]

    if (!is.null(warp_cov)) {
      A_a <- backsolve(U, backsolve(U, ZZa, transpose = TRUE))
      LR_a <- chol2inv(chol(Cinv + Matrix::t(ZZa) %*% A_a))
      x_a <- t(A_a) %*% rra
      A_b <- backsolve(U, backsolve(U, ZZb, transpose = TRUE))
      LR_b <- chol2inv(chol(Cinv + Matrix::t(ZZb) %*% A_b))
      x_b <- t(A_b) %*% rrb
    } else {
      LR_a <- x_a <- 0
      LR_b <- x_b <- 0
    }
    sq_a <- sq_a + (sum(backsolve(U, rra, transpose = TRUE)^2)
```

```
                - t(x_a) %*% LR_a %*% x_a)
    sq_b <- sq_b + (sum(backsolve(U, rrb, transpose = TRUE)^2)
                - t(x_b) %*% LR_b %*% x_b)
    logdet_tmp_a <- 0
    logdet_tmp_b <- 0
    if (!is.null(warp_cov)){
      logdet_tmp_a <- determinant(LR_a)$modulus[1]
      logdet_tmp_b <- determinant(LR_b)$modulus[1]
    }
    logdet_a   <-   logdet_a   -   (logdet_tmp_a   -   2   *
sum(log(diag(U))))
    logdet_b   <-   logdet_b   -   (logdet_tmp_b   -   2   *
sum(log(diag(U))))
  }
  if (!is.null(warp_cov)) {
    logdet_a <- logdet_a - n * determinant(Cinv)$modulus[1]
    logdet_b <- logdet_b - n * determinant(Cinv)$modulus[1]
  }
  sigmahat_a <- as.numeric(sq_a / sum(m))
  sigmahat_b <- as.numeric(sq_b / sum(m))
  res_a <- sum(m) * log(sigmahat_a) + logdet_a
  res_b <- sum(m) * log(sigmahat_b) + logdet_b
  return(res_a + res_b)
}


#' @describeIn like Likelihood function with amplitude
variation in a functional basis.

like_amp <- function(param, n_par, r, Zis, amp_cov, warp_cov,
amp_fct, t, tw) {
  amp_cov_par <- param[1:n_par[1]]
  warp_cov_par <- param[(n_par[1] + 1):length(param)]

  # Compute warp cov
  if (!is.null(warp_cov)) {
    Cinv <- chol2inv(chol(warp_cov(tw, warp_cov_par)))
  } else {
    Cinv <- matrix(0, length(tw), length(tw))
  }

  # Compute amp cov
  df <- attr(amp_fct, 'df')
  if (!is.null(amp_cov)) {
```

```
    Sinv <- chol2inv(chol(amp_cov(1:df, amp_cov_par)))
  }

  n <- length(r)
  m <- sapply(r, length)

  sq <- logdet <- 0
  for (i in 1:n) {
    A <- amp_fct(t[[i]])
    ZZ <- Zis[[i]]
    logdet_tmp <- 0

    if (!is.null(amp_cov)) {
      SLR <- diag(m[i]) - A %*% solve(Sinv + t(A) %*% A, t(A))
    } else {
      SLR <- diag(m[i], x = 1)
    }

    if (!is.null(warp_cov)) {
      LR <- chol2inv(chol(as.matrix(Cinv + t(ZZ) %*% SLR %*%
ZZ)))
      logdet_tmp <- determinant(LR)$modulus[1]
    } else {
      LR <- Matrix::Matrix(data = 0, nrow = nrow(Cinv), ncol
= nrow(Cinv))
    }

    rr <- SLR %*% r[[i]]
    rz <- as.numeric(t(ZZ) %*% rr)
    sq <- sq + t(r[[i]]) %*% rr - t(rz) %*% LR %*% rz

    logdet      <-      logdet      -      logdet_tmp      -
determinant(SLR)$modulus[1]
  }
  if  (!is.null(warp_cov))  logdet  <-  logdet  -  n  *
determinant(Cinv)$modulus[1]
  sigmahat <- as.numeric(sq / sum(m))
  res <- sum(m) * log(sigmahat) + logdet
  return(res)
}



#################### fpfr() ######################
```

```
fpfr <- function(Y1, covariates, funcs, kz,
kb,family=binomial(link = "logit"), method = "REML",
smooth.cov=FALSE){
  kb = min(kz,kb)
  n = dim(funcs[[1]])[1]
  p = ifelse(is.null(covariates), 0, dim(covariates)[2])
  #N_subj = length(unique(subj))

  if(is.matrix(funcs)){
    Funcs = list(length=1)
    Funcs[[1]] = funcs
  }else{
    Funcs = funcs
  }


  # functional predictors
  N.Pred = length(Funcs) #N.Pred=2

  t = phi = psi = CJ = list(length = N.Pred)

  for (i in 1:N.Pred){
    t[[i]] = seq(0,1,length = dim(Funcs[[i]])[2])
    N_obs = length(t[[i]]) #N_obs=m

    #de-mean the functions
    meanFunc = apply(Funcs[[i]], 2, mean, na.rm = TRUE)
    resd = sapply(1:length(t[[i]]), function(u)
Funcs[[i]][,u] - meanFunc[u])
    Funcs[[i]] = resd

    # construct and smooth covariance matrices
    G.sum <- matrix(0, N_obs, N_obs)
    G.count <- matrix(0, N_obs, N_obs)

    for(j in 1:dim(resd)[1]){
      row.ind = j
      temp = resd[row.ind, ] %*% t(resd[row.ind, ])
      G.sum <- G.sum + replace(temp, which(is.na(temp)), 0)
      G.count <- G.count + as.numeric(!is.na(temp))
    }

    G <- ifelse(G.count == 0, NA, G.sum/G.count)
```

```
   ## get the eigen decomposition of the smoothed variance
matrix
   if(smooth.cov){
     G2 <- G
     M <- length(t[[i]])
     diag(G2) = rep(NA, M)
     g2 <- as.vector(G2)
     ## define a N*N knots for bivariate smoothing
     N <- 10

     ## bivariate smoothing using the gamm function
     t1 <- rep(t[[i]], each = M)
     t2 <- rep(t[[i]], M)
     newdata <- data.frame(t1 = t1, t2 = t2)
     K.0 <- matrix(predict(gam(as.vector(g2) ~ te(t1, t2, k
= N)), newdata), M, M) # smooth K.0
     K.0 <- (K.0 + t(K.0))/2

     eigenDecomp <- eigen(K.0)
   }else{
     eigenDecomp <- eigen(G)
   }

   psi[[i]] = eigenDecomp$vectors[,1:kz]

   # set the basis to be used for beta(t)
   num = kb - 2
   qtiles <- seq(0, 1, length = num + 2)[-c(1, num + 2)]
   knots <- quantile(t[[i]], qtiles)
   phi[[i]] = cbind(1, t[[i]], sapply(knots, function(k)
((t[[i]] - k > 0) * (t[[i]] - k))))

   C = matrix(0, nrow = dim(resd)[1], ncol = kz)

   for(j in 1:dim(resd)[1]){
     C[j,] <-  replace(resd[j,],  which(is.na(resd[j,])),
0) %*% psi[[i]][,1:kz]
   }

   J = t(psi[[i]]) %*% phi[[i]]/N_obs
   CJ[[i]] = C %*% J
 }

 X = cbind(rep(1,n), covariates)
```

```
  for (i in 1:N.Pred){
    X = cbind(X,CJ[[i]])
  }

  D = list(length = N.Pred)
  for(i in 1:N.Pred){
    D[[i]] = diag(c(rep(0, 1+p), rep(0, kb*(i-1)), c(rep(0,2),
rep(1, kb-2)), rep(0, kb*(N.Pred-i))))
  }



  ## fit the model
  fit = gam(Y1~X-1, paraPen = list(X=D), family=family,
method = method) #

  ## get the coefficient and betaHat estimates
  coefs = fit$coef
  fitted.vals <- as.matrix(X[,1:length(coefs)]) %*% coefs

  beta.covariates = coefs[1:(p+1)]

  BetaHat = varBeta = varBetaHat = Bounds =
list(length(N.Pred))
  for(i in 1:N.Pred){
    BetaHat[[i]]          =          phi[[i]]          %*%
coefs[(2+p+kb*(i-1)):(1+p+kb*(i))]

    ## get the covariance matrix of the estimated functional
coefficient

varBeta[[i]]=fit$Vp[(2+p+kb*(i-1)):(1+p+kb*(i)),(2+p+kb*(
i-1)):(1+p+kb*(i))]
    varBetaHat[[i]]=phi[[i]]%*%varBeta[[i]]%*%t(phi[[i]])

    ## construct upper and lower bounds for betahat
    Bounds[[i]]        =        cbind(BetaHat[[i]]        +
1.96*(sqrt(diag(varBetaHat[[i]]))),
                      BetaHat[[i]]                          -
1.96*(sqrt(diag(varBetaHat[[i]]))))
  }

  ret <- list(fit, fitted.vals, BetaHat, beta.covariates, X,
phi, psi, varBetaHat, Bounds)
```

```
  names(ret) <- c("fit", "fitted.vals", "BetaHat",
"beta.covariates", "X", "phi",
               "psi", "varBetaHat", "Bounds")
  ret
}



################ test_wran() ####################
test_wran <-function(warp_fct, res, dat, n, basis_fct,
amp_cov, axis){
  tw <- attr(warp_fct, 'tw')
  mw <- attr(warp_fct, 'mw')
  if (all(is.na(tw)))
    tw <- rep(tw, mw + 2)
  tw_int <- tw[2:(mw + 1)]
  # Compute amplitude precision matrices
  Sinv <- fill_precision(dat[[1]][1], amp_cov,
res$amp_cov_par, amp_fct)
  # Build warp covariance and inverse
  if (!is.null(warp_cov)) {
    C <- warp_cov(tw_int, res$warp_cov_par)
    Cinv <- solve(C)
  } else {
    C <- Cinv <- matrix(0, mw, mw)
  }

  w_ran = matrix(0, nrow = 7, ncol = n)

  for (i in 1:n){
    if (axis == '2D') w_ran[,i] <-test_warp_2D(dat[[i]][[1]],
dat[[i]][[2]], dat[[i]][[3]], res, basis_fct, amp_cov =
amp_cov, warp_fct = warp_fct, amp_fct = NULL,
warp_optim_method = 'CG', Sinv, Cinv)
    else if (axis == 'x') w_ran[,i]
<-pred_warp_1D(dat[[i]][[1]], dat[[i]][[2]], lt, res,
basis_fct, amp_cov = amp_cov, warp_fct = warp_fct, amp_fct
= NULL, warp_optim_method = 'CG', Sinv, Cinv)
    else w_ran[,i] <-pred_warp_1D(dat[[i]][[1]],
dat[[i]][[3]], lt, res, basis_fct, amp_cov = amp_cov,
warp_fct = warp_fct, amp_fct = NULL, warp_optim_method = 'CG',
Sinv, Cinv)
  }
  return(w_ran)
```

```
}


################ test_warp_2D() ##################
test_warp_2D <- function(t0, y1, y2, res, basis_fct, amp_cov
= amp_cov, warp_fct = warp_fct, amp_fct = NULL,
                     warp_optim_method = 'CG', Sinv, Cinv){
  ww_ran1 <- optim(par = res$w_ran1[,1], fn = posterior_rand,
gr = NULL, method = warp_optim_method, warp_fct = warp_fct,
t = t0, y1 = y1, y2 = y2, w1 = res$w_fix1, c1 = res$c_a + res$d_a1,
c2 = res$c_b + res$d_b1, Sinv = Sinv[[1]], Cinv = Cinv,
basis_fct = basis_fct)$par
  ww_ran2 <- optim(par = res$w_ran2[,1], fn = posterior_rand,
gr = NULL, method = warp_optim_method, warp_fct = warp_fct,
t = t0, y1 = y1, y2 = y2, w1 = res$w_fix2, c1 = res$c_a + res$d_a2,
c2 = res$c_b + res$d_b2, Sinv = Sinv[[1]], Cinv = Cinv,
basis_fct = basis_fct)$par
  post1 <- posterior_classifi_2D(w1 = res$w_fix1, w2 =
ww_ran1, warp_fct = warp_fct, t = t0, y1 = y1, y2 = y2, basis_fct
= basis_fct, c1 = res$c_a + res$d_a1, c2 = res$c_b + res$d_b1,
Sinv = Sinv[[1]], Cinv = Cinv)
  post2 <- posterior_classifi_2D(w1 = res$w_fix2, w2 =
ww_ran2, warp_fct = warp_fct, t = t0, y1 = y1, y2 = y2, basis_fct
= basis_fct, c1 = res$c_a + res$d_a2, c2 = res$c_b + res$d_b2,
Sinv = Sinv[[1]], Cinv = Cinv)
  p1                                                    <-
exp(-post1/(res$sigma)^2)/(exp(-post1/(res$sigma)^2)    +
exp(-post2/(res$sigma)^2))
  if (is.nan(p1)|p1 <= 1e-10) p1 <- 0
  return(c(ww_ran1,ww_ran2,post1,post2,p1))
}

################ pred_warp_1D() ##################
pred_warp_1D <- function(t0, y,lt, res, basis_fct, amp_cov
= amp_cov, warp_fct = warp_fct,
                     amp_fct = NULL, warp_optim_method =
'CG', Sinv, Cinv){
  ww_ran1   <-   optim(par   =   res$w_ran1[,1],   fn   =
posterior_rand_1D, gr = NULL, method = warp_optim_method,
                warp_fct = warp_fct, t = t0, y = y, w1 =
res$w_fix1, c = res$c + res$c1, Sinv = Sinv[[1]],
                Cinv = Cinv, basis_fct = basis_fct)$par
  ww_ran2   <-   optim(par   =   res$w_ran2[,1],   fn   =
posterior_rand_1D, gr = NULL, method = warp_optim_method,
```

```
                warp_fct = warp_fct, t = t0, y = y, w1 =
res$w_fix2, c = res$c + res$c2, Sinv = Sinv[[1]],
                Cinv = Cinv, basis_fct = basis_fct)$par
  post1 <- posterior_classifi_1D(w1 = res$w_fix1, w2 =
ww_ran1, warp_fct = warp_fct, t = t0, y = y,
                               basis_fct = basis_fct, c = res$c
+ res$c1, Sinv = Sinv[[1]], Cinv = Cinv)
  post2 <- posterior_classifi_1D(w1 = res$w_fix2, w2 =
ww_ran2, warp_fct = warp_fct, t = t0, y = y,
                               basis_fct = basis_fct, c = res$c
+ res$c2, Sinv = Sinv[[1]], Cinv = Cinv)
  p1                                                          <-
exp(-post1/(res$sigma)^2)/(exp(-post1/(res$sigma)^2)    +
exp(-post2/(res$sigma)^2))
  if (is.nan(p1)|p1 <= 1e-10) p1 <- 0
  return(c(ww_ran1,ww_ran2,post1,post2,p1))

}


################ posterior_rand() ##################
posterior_rand <- function(w1, w2, warp_fct, t, y1, y2,
basis_fct, c1, c2, Sinv, Cinv, weights = NULL) {
  if (is.null(weights)) weights <- 1
  vt1 <- warp_fct(w1, w2, t)
  basis1 <- basis_fct(vt1)
  r1 <- weights*(y1 - basis1 %*% c1)
  vt2 <- warp_fct(w1, w2, t)
  basis2 <- basis_fct(vt2)
  r2 <- weights*(y2 - basis2 %*% c2)
  return((t(r1) %*% Sinv %*% r1 + t(r2) %*% Sinv %*% r2 +
2*weights*weights*t(w2) %*% Cinv %*% w2)[1])
}



################ posterior_classifi_1D() ##################
posterior_classifi_1D <- function(w1, w2, warp_fct, t, y,
basis_fct, c, Sinv, Cinv) {
  vt <- warp_fct(w1, w2, t)
  basis <- basis_fct(vt)
  r <- y - basis %*% c
  return((t(r) %*% Sinv %*% r)[1])
}
```

```
############                          posterior_classifi_2D()
#######################
posterior_classifi_2D <- function(w1, w2, warp_fct, t, y1,
y2, basis_fct, c1, c2, Sinv, Cinv) {
  vt1 <- warp_fct(w1, w2, t)
  basis1 <- basis_fct(vt1)
  r1 <- y1 - basis1 %*% c1
  vt2 <- warp_fct(w1, w2, t)
  basis2 <- basis_fct(vt2)
  r2 <- y2 - basis2 %*% c2
  return((t(r1) %*% Sinv %*% r1 + t(r2) %*% Sinv %*% r2)[1])
}



################ func_coef() ###################
func_coef <- function(Ay_train, covat, kz, kb, d = 'raw'){
  if (d == 'raw'){
    funcs_tr  <-  list(rbind(Ay_train[[5]],Ay_train[[7]]),
rbind(Ay_train[[6]],Ay_train[[8]]))
  }else{
    funcs_tr  <-  list(rbind(Ay_train[[1]],Ay_train[[3]]),
rbind(Ay_train[[2]],Ay_train[[4]]))
  }
  #funcs_te    <-    list(Align_Curve_data[[k]][[4]][[1]],
Align_Curve_data[[k]][[4]][[2]])
  X_tr <- predictor(covariates = covat, funcs = funcs_tr, kz
= kz, kb = kb, smooth.cov = FALSE)
  #attach(X_tr)
  fit = gam(y_tr~X_tr$X-1, paraPen = list(X = X_tr$D), family
= binomial(link = "logit"), method = "REML")
  return(fit$coef)
}



################## predictor() ###############
predictor   <-   function(covariates,   funcs,   kz,   kb,
smooth.cov=FALSE){
  kb = min(kz,kb)
  n = dim(funcs[[1]])[1]
```

```
  p = ifelse(is.null(covariates), 0, dim(covariates)[2])
  #N_subj = length(unique(subj))

  if(is.matrix(funcs)){
    Funcs = list(length=1)
    Funcs[[1]] = funcs
  }else{
    Funcs = funcs
  }

  # functional predictors
  N.Pred = length(Funcs)

  t = phi = psi = CJ = list(length = N.Pred)

  for (i in 1:N.Pred){
    t[[i]] = seq(0,1,length = dim(Funcs[[i]])[2])
    N_obs = length(t[[i]])

    #de-mean the functions
    meanFunc = apply(Funcs[[i]], 2, mean, na.rm = TRUE)
    resd     =     sapply(1:length(t[[i]]),     function(u)
Funcs[[i]][,u] - meanFunc[u])
    Funcs[[i]] = resd

    # construct and smooth covariance matrices
    G.sum <- matrix(0, N_obs, N_obs)
    G.count <- matrix(0, N_obs, N_obs)

    for(j in 1:dim(resd)[1]){
      row.ind = j
      temp = resd[row.ind, ] %*% t(resd[row.ind, ])
      G.sum <- G.sum + replace(temp, which(is.na(temp)), 0)
      G.count <- G.count + as.numeric(!is.na(temp))
    }

    G <- ifelse(G.count == 0, NA, G.sum/G.count)

    ## get the eigen decomposition of the smoothed variance
matrix
    if(smooth.cov){
      G2 <- G
      M <- length(t[[i]])
      diag(G2) = rep(NA, M)
```

```
      g2 <- as.vector(G2)
      ## define a N*N knots for bivariate smoothing
      N <- 10

      ## bivariate smoothing using the gamm function
      t1 <- rep(t[[i]], each = M)
      t2 <- rep(t[[i]], M)
      newdata <- data.frame(t1 = t1, t2 = t2)
      K.0 <- matrix(predict(gam(as.vector(g2) ~ te(t1, t2, k
= N)), newdata), M, M) # smooth K.0
      K.0 <- (K.0 + t(K.0))/2

      eigenDecomp <- eigen(K.0)
    }else{
      eigenDecomp <- eigen(G)
    }

    psi[[i]] = eigenDecomp$vectors[,1:kz]

    # set the basis to be used for beta(t)
    num = kb - 2
    qtiles <- seq(0, 1, length = num + 2)[-c(1, num + 2)]
    knots <- quantile(t[[i]], qtiles)
    phi[[i]] = cbind(1, t[[i]], sapply(knots, function(k)
((t[[i]] - k > 0) * (t[[i]] - k))))

    C = matrix(0, nrow = dim(resd)[1], ncol = kz)

    for(j in 1:dim(resd)[1]){
      C[j,] <-  replace(resd[j,],  which(is.na(resd[j,])),
0) %*% psi[[i]][,1:kz]
    }

    J = t(psi[[i]]) %*% phi[[i]]/N_obs
    CJ[[i]] = C %*% J
  }

  X = cbind(rep(1,n), covariates)
  for (i in 1:N.Pred){
    X = cbind(X,CJ[[i]])
  }

  D = list(length = N.Pred)
  for(i in 1:N.Pred){
```

```
    D[[i]] = diag(c(rep(0, 1+p), rep(0, kb*(i-1)), c(rep(0,2),
rep(1, kb-2)), rep(0, kb*(N.Pred-i))))
  }
  return(list(X=X, D=D))
}


########### clasf_scal() ######################
clasf_scal<-function(n,y){
  X0 <- cbind(rep(1,2*n),y$x1[1:(2*n)])
  D0 <- diag(rep(1, 2))
  y0 <- y$Res[1:(2*n)]
  fit = gam(y0~X0-1, paraPen = list(D0), family=binomial(link
= "logit"), method = "REML")#list(D0)
  coefs1 = fit$coef
  X1 <- cbind(rep(1,2*n),y$x1[(2*n + 1):(4*n)])
  fitted.vals1 <- as.matrix(X1[, 1:length(coefs1)]) %*%
coefs1
  P0 <- 1/(1+exp(-fitted.vals1))
  Re0 <- as.numeric(P0>0.5)

  fiter1 <- mean(abs(y$P[(2*n + 1):(4*n)]-P0))
  er1 <- sum(as.numeric(y$Res[(2*n + 1):(4*n)] == Re0)/(2*n))
  ri1 <- randIndex(table(y$Res[(2*n + 1):(4*n)], Re0),
correct = F)[[1]]
  ari1 <-randIndex(table(y$Res[(2*n + 1):(4*n)], Re0),
correct = T)[[1]]

  C1 = c(fiter1, er1, ri1, ari1)
  return(list(C1 = C1, P0 = P0))
}


########### classi_fs() ######################
classi_fs <- function(y,ng,dat,warp_fct,basis_fct, initial,
p, iteration, kz, kb, t2,res,w_ran_te1,coefs2){
  ya_test <- yb_test <- list()
  for (i in 1:(2*ng)){
    ya_test[[i]] = dat[[i]][[2]]
    yb_test[[i]] = dat[[i]][[3]]
  }
  P <- sample(0:1,2*ng, replace = T)

  #er1 <- rep(0,20)
```

```
  iter <- 1
  while (iter <= iteration){
    ss <- P
    Ay_test                                                    <-
Test_curves(ya_test,yb_test,t2,res,ss,w_ran_te1,warp_fct,
basis_fct)

    K0 = list(ind1 = which(ss>=0.5), ind2 = which(ss<0.5))
    Re0            <-            c(rep(1,length(K0[[1]])),
rep(0,length(K0[[2]])))
    K00<-c(K0[[1]],K0[[2]])

    ##test the logistic model
    covt2 <- y$x1[(2*ng + 1):(4*ng)]
    #covt2 <- rbind(covt2[K0[[1]],],covt2[K0[[2]],])
    covt2 <- c(covt2[K0[[1]]],covt2[K0[[2]]])
    funcs_te   <-   list(rbind(Ay_test[[1]],Ay_test[[3]]),
rbind(Ay_test[[2]],Ay_test[[4]]))
    X_te <- predictor(covariates = as.matrix(covt2), funcs =
funcs_te, kz, kb, smooth.cov = FALSE)
    fitted.vals2 <- as.matrix(X_te$X[, 1:length(coefs2)]) %*%
coefs2
    P <- 1/(1+exp(-fitted.vals2))
    Re <- as.numeric(P>0.5)

    ##check the disagreements
    iter <- iter + 1
    #er1[iter] <- mean(abs(Re0-Re))
  }

  fiter2 <- mean(abs(y$P[(2*ng + 1):(4*ng)][K00]-P))
  er2 <- sum(as.numeric(y$Res[(2*ng + 1):(4*ng)][K00] ==
Re)/(2*ng))
  ri2 <- randIndex(table(y$Res[(2*ng + 1):(4*ng)][K00], Re),
correct = F)[[1]]
  ari2 <-randIndex(table(y$Res[(2*ng + 1):(4*ng)][K00], Re),
correct = T)[[1]]
  C2 = c(fiter2, er2, ri2, ari2)
  return(list(C2 = C2, Ay_test = Ay_test))
}


################### Test_curves() ############
Test_curves                                                <-
```

```
function(ya,yb,t,res,ss,w_ran,warp_fct,basis_fct){

  #ss <- w_ran_te1[7,]
  K0 = list(ind1 = which(ss>=0.5), ind2 = which(ss<0.5))
  n1 = length(K0[[1]]);
  n2 = length(K0[[2]]);
  Aligned_ya1 <-  Aligned_yb1 <- matrix(0, nrow = n1, ncol =
length(t[[1]]));
  Aligned_ya2 <-  Aligned_yb2 <- matrix(0, nrow = n2, ncol =
length(t[[1]]));
  Raw_ya1 <-   Raw_yb1 <-  matrix(0,  nrow  =  n1,  ncol  =
length(t[[1]]));
  Raw_ya2 <-   Raw_yb2 <-  matrix(0,  nrow  =  n2,  ncol  =
length(t[[1]]));

  for (i in 1:n1) {
    Aligned_ya1[i,]     <-     approx(warp_fct(res$w_fix1,
w_ran[,K0[[1]][i]][1:2], t[[i]]), ya[[K0[[1]][i]]], xout =
t[[i]], rule = 2)$y
    Aligned_yb1[i,]     <-     approx(warp_fct(res$w_fix1,
w_ran[,K0[[1]][i]][1:2], t[[i]]), yb[[K0[[1]][i]]], xout =
t[[i]], rule = 2)$y
    Raw_ya1[i,]   <-   ya[[K0[[1]][i]]];   Raw_yb1[i,]   <-
yb[[K0[[1]][i]]];
  }

  for (i in 1:n2) {
    Aligned_ya2[i,]     <-     approx(warp_fct(res$w_fix2,
w_ran[,K0[[2]][i]][3:4], t[[i]]), ya[[K0[[2]][i]]], xout =
t[[i]], rule = 2)$y
    Aligned_yb2[i,]     <-     approx(warp_fct(res$w_fix2,
w_ran[,K0[[2]][i]][3:4], t[[i]]), yb[[K0[[2]][i]]], xout =
t[[i]], rule = 2)$y
    Raw_ya2[i,]   <-   ya[[K0[[2]][i]]];   Raw_yb2[i,]   <-
yb[[K0[[2]][i]]];
  }

  return(list(Aligned_ya1  =  Aligned_ya1,  Aligned_yb1  =
Aligned_yb1,  Aligned_ya2  =  Aligned_ya2,  Aligned_yb2  =
Aligned_yb2,
              Raw_ya1 = Raw_ya1, Raw_yb1 = Raw_yb1, Raw_ya2 =
Raw_ya2, Raw_yb2 = Raw_yb2))

}
```

```
################### clasf_func2() ############
clasf_func2   <-   function(y,ng,dat,warp_fct,basis_fct,p,
iteration, kz, kb, t2,res,w_ran_te1,coefs2){
  ya_test <- yb_test <- list()
  for (i in 1:(2*ng)){
    ya_test[[i]] = dat[[i]][[2]]
    yb_test[[i]] = dat[[i]][[3]]
  }

  P <- sample(0:1,2*ng, replace = T)

  iter <- 1
  while (iter <= iteration){
    ss <- P
    Ay_test                                                <-
Test_curves(ya_test,yb_test,t2,res,ss,w_ran_te1,warp_fct,
basis_fct)
    K0 = list(ind1 = which(ss>=0.5), ind2 = which(ss<0.5))
    Re0            <-            c(rep(1,length(K0[[1]])),
rep(0,length(K0[[2]])))
    K00<-c(K0[[1]],K0[[2]])

    funcs_te   <-   list(rbind(Ay_test[[1]],Ay_test[[3]]),
rbind(Ay_test[[2]],Ay_test[[4]]))

    X_te <- predictor(covariates = NULL, funcs = funcs_te, kz,
kb, smooth.cov = FALSE)
    fitted.vals2 <- as.matrix(X_te$X[, 1:length(coefs2)]) %*%
coefs2
    P <- 1/(1+exp(-fitted.vals2))
    Re <- as.numeric(P>0.5)

    iter = iter + 1
  }

  fiter2 <- mean(abs(y$P[(2*ng + 1):(4*ng)][K00]-P))
  er2  <-  sum(as.numeric(y$Res[(2*ng  +  1):(4*ng)][K00]  ==
Re)/(2*ng))
  ri2 <- randIndex(table(y$Res[(2*ng + 1):(4*ng)][K00], Re),
correct = F)[[1]]
  ari2 <-randIndex(table(y$Res[(2*ng + 1):(4*ng)][K00], Re),
```

```
correct = T)[[1]]
  C2 = c(fiter2,er2, ri2, ari2)
  return(list(C2 = C2, Ay_test = Ay_test))
}
```