```
# TwoFold cross validation (Royal of statistics: Wavelet Shrinkage Using
Cross-Validation, 1996)
# out contribution: irregular spaced data + all sample sizes
source("LiftingSchemeJPEG.R")

test.MCP<-function(Signal, X.Grid, Level=1, lambda, gamma=Inf){
#  if (is.nan(coeff[1])){
    temp = main.filter.X.Grid(Signal, X.Grid, Level)
    coeff=temp$y
    my.X.Grid = temp$X.Grid
  n = length(Signal)
  object.fun = function(coeff){
    yhat = inv.main.filter.X.Grid(coeff, my.X.Grid, Level)$y
    residual = Signal - yhat
    obj.fun=1/(2*n)*sum(residual^2) +
sum(MCP.penalty(abs(coeff),lambda=lambda, gamma=gamma))
  #  print(obj.fun)
    return(obj.fun)
  }

  #coeff=Denoised.func(Signal,X.Grid,Penalty.type = "Hard")
#  print(object.fun(coeff))
  optim(coeff,object.fun)
}
# example:
# test.MCP(1:15,1:15,lambda=0.5,gamma=1)
# compare with:

#
t=Global.Penalization_nir(1:15,X.Grid=1:15,lambda=0.5,gamma=1,alpha=0.001
)

MCP.penalty<-function(params, lambda, alpha=1, gamma=Inf){
  params.out = params
  Indicator = (params<=lambda*gamma)*1
  out=rep(0.5*lambda^2*gamma, length(params))
  if ((sum(Indicator) > 0) & (lambda > 0)) {
    p = params[Indicator==1]
    out[Indicator==1] = lambda*p-p^2/(2*gamma)
  }
  return(out)
}

#-----------------------------------------------------------
S.alpha.Threshold<-function(params, lambda, alpha=1, gamma=Inf){
  gamma = max(gamma,1.0000000001)
  maxRow<-function(x1,x2)unlist(sapply(1:max(length(x1),length(x2)),
function(i)max(x1[min(i,length(x1))],x2[min(i,length(x2))])))
  sign.vec<-function(x)(x>0)*1 + (x<0)*(-1)
  Indicator = (abs(params)<=lambda*gamma)*1

  if ((sum(Indicator) > 0) & (lambda > 0)) {
    params2 = params[Indicator==1]
```

```
    p=1/(1-1/(alpha*gamma))*sign.vec(params2)*maxRow(abs(params2)-
lambda/alpha,0)
    params[Indicator==1] = p
  }
  return(params)
}

#----------- Global.penalization choice -------------
Global.Penalization_nir<-function(Signal, X.Grid=NaN, coeff=NaN, Level=1,
UpToLevel=NaN, alpha=0.001, gamma=Inf, lambda = NaN) {

  if (is.nan(Level)) {
    Level = ceiling(log(length(coeff)/log(2)))
  }
  if (is.nan(UpToLevel)) {
    UpToLevel = Level
  }

  if (is.nan(X.Grid[1])){
    X.Grid = 1:length(Signal)
  }

  if (is.nan(coeff[1])){
    temp = main.filter.X.Grid(Signal, X.Grid, Level)
    coeff=temp$y
    my.X.Grid = temp$X.Grid
  } else {
    my.X.Grid = forward.X.Grid(X.Grid = X.Grid, Level=Level)
  }


  #res=Build_res(length(coeff), Level)

  iter = 0
  gap = Inf
  n = length(Signal)
  new.alpha=NULL
  max.coeff=max(abs(coeff))
  min.coeff=min(abs(coeff))
  print(sprintf("max.coeff=%f, lambda/alpha=%f,
gamma*lambda=%f",max.coeff, lambda/alpha, gamma*lambda))

#  print(coeff)


  min.gamma = min.coeff / lambda

  #gamma = max(gamma, min.gamma)

  OK = TRUE
  gap = Inf
while ((iter < 100) & OK & (gap>0.0001)) {
  gap = Inf
  Xt.res = NULL
```

```
  old.Xt.res = NULL
  iter.inner=0

  yhat = inv.main.filter.X.Grid(coeff, my.X.Grid, Level)$y
  residual = Signal - yhat
  Xt.res = inv.t(residual, X.Grid=X.Grid, Level = Level)

  object.fun = 1/(2*n)*sum(residual^2) +
sum(MCP.penalty(abs(coeff),lambda, alpha, gamma))

    print(sprintf("obj=%f",object.fun))

    old.object.fun = object.fun
    old.coeff = coeff
    old.Xt.res = Xt.res
    old.yhat = yhat

  inner.iter = 0
  if (iter==0){
    alpha = alpha
  } else{
    alpha = sqrt(new.alpha)
  }

  # ?
  #alpha = max(lambda/max(abs(coeff))+0.001, alpha)
  # ?

  repeat {
    Update = coeff+rep(1/(alpha*n),length(Xt.res))*Xt.res

#    if ((lambda/alpha>=max.coeff) & (gamma*lambda>=min.coeff)) { # do
lasso and replace the coefficients with zeros
#      coeff = rep(0,n)
#      yhat = rep(0,n)
#    } else if (gamma*lambda < min.coeff) { # do hard thresholding and
keep the origional coefficients
#      coeff = coeff
#      yhat = old.yhat # inv.main.filter.X.Grid(coeff, my.X.Grid,
Level)$y
#    } else { # do an iterative update
      coeff = S.alpha.Threshold(Update,lambda, alpha,gamma)
      yhat = inv.main.filter.X.Grid(coeff, my.X.Grid, Level)$y

 #   }

    residual = Signal - yhat

    object.fun = 1/(2*n)*sum(residual^2) +
sum(MCP.penalty(abs(coeff),lambda, alpha, gamma))

    gap = max(abs(coeff-old.coeff))

    iter.inner = iter.inner + 1
```

```
    OK = (object.fun < old.object.fun)
    if ((inner.iter > 100)| OK) {
      Xt.res = inv.t(residual, X.Grid=X.Grid, Level = Level)
      break
    } else {
      inner.iter = inner.iter + 1
      alpha = alpha * 1.2;
      coeff = old.coeff
      object.fun = old.object.fun
      Xt.res = old.Xt.res
      yhat = old.yhat
    }
  }

  if (OK){
    g=(-1/n*Xt.res)-(-1/n*old.Xt.res)
    #print("g=")
    #print((sprintf("length(g)=%f, length(old.coeff)=%f,
length(coeff)=%f", length(g), length(old.coeff),length(coeff))))
    new.alpha=(t(as.matrix(g))%*%as.matrix(coeff-
old.coeff))/(t(as.matrix(coeff-old.coeff))%*%as.matrix(coeff-old.coeff))
  } else {
    new.alpha = NULL
  }
  print(sprintf("ok==%d, obj=%f, alpha=%f, inner.iter=%f,
new.alpha=%f",OK*1, object.fun, alpha, inner.iter, new.alpha))
  print(sprintf("max.coeff=%f, lambda/alpha=%f,
gamma*lambda=%f",max.coeff, lambda/alpha, gamma*lambda))

  iter = iter + 1
} # ouer while alpha
  #print(sprintf("iter=%f, alpha=%f, new.alpha=%f",iter, alpha,
new.alpha))


  #print("?")
  print(sprintf("min.gamma=%f, min.coeff=%f", min.gamma, min.coeff))
  return(list(coeff=old.coeff, X.Grid=my.X.Grid, obj.fun=old.object.fun,
yhat=old.yhat))
}

# example:
# S.alpha.Threshold(c(1,250,25),lambda=10, alpha=1, gamma=1)

# smaller alpha values decrease running time
#
Global.Penalization_nir(1:4000,X.Grid=1:4000,lambda=0.05,gamma=Inf,alpha=
0.001)

#t=Global.Denoised.func(1:10,X.Grid=1:10,Level=1,lambda=0.05,gamma=Inf,al
pha=1)

#-------------- implementation ------------------
```

```
Global.Denoised.func<-function(y, X.Grid = NaN, Level = NaN, UpToLevel =
NaN,  lambda = NaN, alpha=0.001, gamma=1){
  n = length(y)
  MaxLevel = ceiling(log(n)/log(2))
  if (is.nan(Level)) {
    Level = MaxLevel
  }
  else {
    Level = min(Level, MaxLevel)
  }

  if (is.nan(UpToLevel)){
    UpToLevel = min(Level-1, 7+(n>=2000))
  }

  # in global denoising, the penalization and forward transform are
combined and not separated
  #coeff=main.filter.X.Grid(y,X.Grid = X.Grid, Level = Level)
  coeff=Global.Penalization_nir(y,Level = Level, UpToLevel = UpToLevel,
lambda = lambda, alpha=alpha, gamma=gamma)
  yhat=coeff$yhat#inv.main.filter.X.Grid(coeff$y,X.Grid = coeff$X.Grid,
Level = Level)
  return(list(yhat=yhat, obj.fun=coeff$obj.fun))
}

Global.Eval.Lambda<-function(Signal, X.Grid = NaN, Level = NaN, UpToLevel
= NaN, lambda = NaN, alpha=0.001, gamma=Inf) {
  n=length(Signal)

  if (is.nan(X.Grid[1])) {
    X.Grid = 1:n
  }

  m = ceiling(n/2)

  X1 = Signal[seq(2,n,by=2)]

  X0 = Signal[seq(1,n,by=2)]

  if (n%%2 > 0) { # extrapolation
  #---------------------------------------------------------------
    n = n + 1; # enter a new data point
    # X0(M1-1,:,:)*ExtrapolateOdd(1) + X(N1-1,1:N2,:)*ExtrapolateOdd(2) +
X0(M1,:,:)*ExtrapolateOdd(3)]
    new.even = X0[length(X0)-1]*ExtrapolateOdd[1] +
X1[length(X1)]*ExtrapolateOdd[2] + X0[length(X0)]*ExtrapolateOdd[3]
    X1 = c(X1, new.even)
    X.Grid=c(X.Grid,X.Grid[length(X.Grid)]-X.Grid[length(X.Grid)-
1]+X.Grid[length(X.Grid)])

    #weights=-
2*c(LiftFilter[3]+2*LiftFilter[1]*LiftFilter[2]*LiftFilter[3],LiftFilter[
2]*LiftFilter[3])/(1+2*LiftFilter[2]*LiftFilter[3])
```

```
    t0=X.Grid[length(X.Grid)-1]
    t0.lag=X.Grid[length(X.Grid)-3]
    t1.lag=X.Grid[length(X.Grid)-2]

    w0=(t0-t1.lag)/(t0-t0.lag); we = 0.5

    weights=-2*c(4*LiftFilter[1]*LiftFilter[2]*LiftFilter[3]*w0*we,
2*LiftFilter[2]*LiftFilter[3]*we,LiftFilter[1]+LiftFilter[3]+4*LiftFilter
[1]*LiftFilter[2]*LiftFilter[3]*((1-we)+(1-
w0)*we))/(1+4*LiftFilter[2]*LiftFilter[3]*(1-we))
    #    ExtrapolateOdd = -
2*cbind(S1*S2*S3,S2*S3,S1+S3+3*S1*S2*S3)/(1+2*S2*S3);
    X1[length(X1)]=weights[1]*X0[length(X0)-1]+weights[2]*X1[length(X1)-
1]+weights[3]*X0[length(X0)]
   # print(sprintf("new.x=%f",X1[length(X1)]))
    flag = 1
  #-----------------------------------------------------------------

  } else {
    flag = 0
  }

  Even.Grid = X.Grid[seq(2,n,by=2)]

  Odd.Grid = X.Grid[seq(1,n,by=2)]

  X1.hat = Global.Denoised.func(X1, X.Grid = Even.Grid, Level = Level,
UpToLevel =  UpToLevel, lambda = lambda, alpha=alpha, gamma=gamma)$yhat
  X0.hat = Global.Denoised.func(X0, X.Grid = Odd.Grid, Level = Level,
UpToLevel =  UpToLevel, lambda = lambda, alpha=alpha, gamma=gamma)$yhat

  Interp.X0 = filter(X1.hat, TRUE, 0.5, X.Grid = X.Grid)

  Interp.X1 = filter(X0.hat, FALSE, 0.5, X.Grid = X.Grid)

  sse = sum((X1 - Interp.X1)^2) + sum((X0 - Interp.X0)^2)
  # see filter(c(2,4,6,8), TRUE, 0.5, X.Grid = c(1:8))

  if (flag==1) {
    X1 = X1[1:(length(X1)-1)]
    X.Grid = X.Grid[1:(length(X.Grid)-1)]
    n = n - 1
  }

  return(sse)
}

# this function utilizes the orinigal noisy signal
Global.eval.Lambda.inv.main.filter.X.Grid<-function(y, X.Grid=NaN,
Level=NaN, UpToLevel=NaN, Penalty.type=Inf, lambda.list = NULL,
penalty.list=NULL) {
  MaxLevel = ceiling(log(length(y))/log(2));
  if (is.nan(Level)) {
    Level = MaxLevel
```

```
  } else {
    Level=min(Level, MaxLevel);
  }

  if (is.nan(UpToLevel)) {
    UpToLevel = floor(log(length(y))/log(2));#Level
  } else {
    UpToLevel=min(UpToLevel, floor(log(length(y))/log(2)));
  }

  n=length(y)

  M = ceiling(n/2^(Level-1))
  Q = ceiling(M/2);

  sig=median(abs(y[1:M]))/0.6745
  univ.lambda=sig*sqrt(2*log(M))

  max.lambda = max(abs(y[1:M]))
  print(sprintf("maxLambda=%f",max.lambda))
  #print(sprintf("universal Lambda=%f",univ.lambda))


  # ----------- evaluate SSE (resoultion-dependent) -------------
  # I. Denoise Y using penalization (while sse is not optimized do... )
  # while .... # search for Lambda
  #sse = Denoised.Penalized.sse(Noisy.Signal=y[1:M], X.Grid=X.Grid[1:M],
Penalty.type = Penalty.type, Lambda=Lambda)
  # end of while
  # ----------- end of SSE evaluation -------------


  X0 = y[1:Q]/ScaleFactor
  X1 = y[(Q+1):M]*ScaleFactor

  X0.Grid = X.Grid[1:Q]
  X1.Grid = X.Grid[(Q+1):M]

  # Undo lifting stages: shift only the first M elements
  X.Grid[c(seq(1,M,by=2),seq(2,M,by=2))] = c(X0.Grid, X1.Grid)

  if (M%%2 > 0){
    X1[Q] = 0; # an additional ("fake") even data point
    new.t = X.Grid[M]-X.Grid[M-1]+X.Grid[M]
    last.X.Grid=X.Grid
    X.Grid = c(X.Grid[1:M], new.t, X.Grid[(M+1):length(X.Grid)])
    flag = 1
  } else {
    flag = 0
    #X.Grid[c(seq(1,M,by=2),seq(2,M,by=2))] = c(X0.Grid, X1.Grid)
  }


  X0 = X0 - filter(X1, TRUE, LiftFilter[4], X.Grid)
```

```
  X1 = X1 - filter(X0, FALSE, LiftFilter[3], X.Grid)
  X0 = X0 - filter(X1, TRUE, LiftFilter[2], X.Grid)
  X1 = X1 - filter(X0, FALSE, LiftFilter[1], X.Grid)
  if (flag==1) {
    X1 = X1[-Q];
    X.Grid=last.X.Grid
  }

  y[c(seq(1,M,by=2),seq(2,M,by=2))] = c(X0, X1)
  # myobj = list(y=y, X.Grid = X.Grid)

  # while sse can be decreased change Lambda:

  if (UpToLevel>=Level) { # denoise if Level in {1,..,UpToLevel}
    sse = function(lambda, gamma)Global.Eval.Lambda(Signal = y[1:M],
X.Grid = X.Grid[1:M], Level = 1, UpToLevel = 1, lambda = lambda,
gamma=gamma)

    types=Penalty.type ##c("hyperbolic", "hard", "scad","adapt")

fmin=lapply(1:length(types),function(i)optimize(function(x)sse(x,types[i]
), c(0,10), tol=10^-12))

#fmin=lapply(1:length(types),function(i)Median.Optimizer(function(x)sse(x
,types[i]), c(0,0.5)))

#fmin=lapply(1:length(types),function(i)Median.Optimizer(function(x)sse(x
,types[[i]]), c(1,5)))

    #print("?")

    s=sapply(1:length(types),function(i)fmin[[i]]$minimum)

    opt.gamma = types[which.min(s)]

    fmin=fmin[[which.min(s)]] # select the optimal
    #Lambda=fmin$minimum

    #print(sprintf("UpToLevel = %d, Level=%d, Lambda=%f, sse=%f",
UpToLevel, Level, Lambda, fmin$objective))

    #----------
    #fmin=Median.Optimizer(sse, c(0,5))
    opt.Lambda=fmin$minimum * (1-log(2)/log(M))^(-1/2)
    print(sprintf("UpToLevel = %d, Level=%d, opt.Lambda=%f, sse=%f,
opt.gamma=%s", UpToLevel, Level, opt.Lambda, fmin$objective, opt.gamma))


    # end of while

    # Denoising using optimal lambda which is the arg min of sse function
    y[1:M] =Global.Denoised.func(y[1:M], X.Grid = X.Grid[1:M], Level = 1,
UpToLevel = 1, lambda = opt.Lambda, gamma = opt.gamma)$yhat
  } else {
```

```
    opt.Lambda = 0
    opt.gamma = NaN
  }

  myobj = list(y=y, X.Grid = X.Grid, Lambda=c(opt.Lambda, lambda.list),
Penalty=c(opt.gamma, penalty.list))

  if (Level > 1) {
    myobj = Global.eval.Lambda.inv.main.filter.X.Grid(myobj$y,
myobj$X.Grid, Level - 1, UpToLevel = UpToLevel, Penalty.type =
Penalty.type, lambda.list = myobj$Lambda, penalty.list = myobj$Penalty)
  }

  return(myobj)
}
```