

# GENERALIZED LINEAR DENSITY RATIO MODEL (GLDRM) WITH FREQUENCY WEIGHTS

ANIKO SZABO

The code is based on the `gldrm` package, with minor changes to incorporate frequency weights. The code is simplified a bit as well by removing some options:

- sampling weights are removed
- there is no variance/standard error calculation, and no inference against the null

```
"../R/wgldrm.R" 1≡
```

```
#' Main optimization function
#
#' This function is called by the main \code{gldrm} function.
#
#' @keywords internal
#' @importFrom stats lm.fit lm.wfit weighted.mean
gldrm.control <- function(eps=1e-10, maxiter=100, returnfTiltMatrix=TRUE,
                          returnfOScoreInfo=FALSE, print=FALSE,
                          betaStart=NULL, f0Start=NULL)
{
  gldrmControl <- as.list(environment())
  class(gldrmControl) <- "gldrmControl"
  gldrmControl
}

gldrmFit <- function(x, y, linkfun, linkinv, mu.eta, mu0=NULL, offset=NULL, weights=NULL,
                    gldrmControl=gldrm.control(), thetaControl=theta.control(),
                    betaControl=beta.control(), f0Control=f0.control())
{
  ## Extract control arguments
  if (class(gldrmControl) != "gldrmControl")
    stop("gldrmControl must be an object of class \'gldrmControl\' returned by
         gldrm.control() function.")
  eps <- gldrmControl$eps
  maxiter <- gldrmControl$maxiter
  returnfTiltMatrix <- gldrmControl$returnfTiltMatrix
  returnfOScoreInfo <- gldrmControl$returnfOScoreInfo
  print <- gldrmControl$print
  betaStart <- gldrmControl$betaStart
  f0Start <- gldrmControl$f0Start

  ## Tabulation and summary of responses used in estimating f0
  n <- length(y)
  spt <- sort(unique(y)) # observed support
  ySptIndex <- match(y, spt) # index of each y value within support
  # sptFreq <- table(ySptIndex)
  # attributes(sptFreq) <- NULL

  # create a weight matrix for score.logT1 calculation
  weightsMatrix <- matrix(0, nrow=n, ncol=length(spt))
```

```

weightsMatrix[cbind(1:n, ySptIndex)] <- weights

# weighted version of "sptFreq"
sptFreq.weighted <- colSums(weightsMatrix)

## Initialize offset
if (is.null(offset))
  offset <- rep(0, n)

## Initialize mu0 if not provided by user
if (is.null(mu0)) {
  mu0 <- weighted.mean(y, weights)
} else if (mu0 <= min(spt) || mu0 >= max(spt)) {
  stop(paste0("mu0 must lie within the range of observed values. Choose a different ",
    "value or set mu0=NULL to use the default value, weighted.mean(y,weights)."))
}

## Initialize f0
if (is.null(f0Start)) {
  # weighted version of initial value for baseline distribution
  f0 <- sptFreq.weighted / sum(sptFreq.weighted)
  if (mu0 != weighted.mean(y, weights))
    f0 <- getTheta(spt=spt, f0=f0, mu=mu0, weights=weights, ySptIndex=1, thetaStart=0,
      thetaControl=thetaControl)$fTilt[, 1]
} else {
  if (length(f0Start) != length(spt))
    stop("Length of f0Start should equal number of unique values in the response.")
  if (any(f0Start <= 0))
    stop("All values in f0Start should be strictly positive.")
  f0 <- f0Start / sum(f0Start)
  f0 <- getTheta(spt=spt, f0=f0, mu=mu0, weights=weights, ySptIndex=1, thetaStart=0,
    thetaControl=thetaControl)$fTilt[, 1]
}

## Initialize beta
## The starting values returned by lm.fit guarantee that all mu values are
## within the support range, even if there is no intercept.
## Offset could still create problems.
lmcoef <- stats::lm.wfit(x=x, y=linkfun(mu0) - offset, weights)$coef
if (is.null(betaStart)) {
  beta <- lmcoef
} else {
  if (length(betaStart) != ncol(x))
    stop("Length of betaStart should equal the number of columns in the model matrix.")
  beta <- betaStart
}

## Drop coefficients if x is not full rank (add NA values back at the end)
naID <- is.na(lmcoef)
beta <- beta[!naID]
x <- x[, !naID, drop=FALSE]
eta <- c(x %*% beta + offset)
mu <- linkinv(eta)
if (ncol(x) >= n)
  stop("gldrm requires n > p.")
if (any(mu < min(spt) | mu > max(spt)))
  stop("Unable to find beta starting values that do not violate convex hull condition.")

## Get initial theta and log likelihood

```

```

th <- getTheta(spt=spt, f0=f0, mu=mu, weights=weights, ySptIndex=ySptIndex,
               thetaStart=NULL, thetaControl=thetaControl)
llik <- th$llik

conv <- FALSE
iter <- 0
while (!conv && iter <= maxiter)
{
  iter <- iter+1
  betaold <- beta
  f0old <- f0
  llikold <- llik

  ## update beta (mu) and theta, with fixed f0:
  bb <- getBeta(x=x, y=y, spt=spt, ySptIndex=ySptIndex, f0=f0,
                linkinv=linkinv, mu.eta=mu.eta, offset=offset, weights=weights,
                betaStart=beta, thStart=th,
                thetaControl=thetaControl, betaControl=betaControl)

  th <- bb$th
  llik <- bb$llik
  mu <- bb$mu
  beta <- bb$beta

  ## update f0 and theta, with fixed beta (mu)
  ff <- getf0(y=y, spt=spt, ySptIndex=ySptIndex,
              weights=weights, sptFreq.weighted=sptFreq.weighted, mu=mu, mu0=mu0, f0Start=f0, thSta
              thetaControl=thetaControl, f0Control=f0Control, trace=FALSE)

  th <- ff$th
  llik <- ff$llik
  f0 <- ff$f0

  ## Check convergence
  del <- abs((llik - llikold) / llik)
  if (llik == 0) del <- 0
  conv <- del < eps

  if (print) {
    cat("iteration ", iter,
        "\nrelative change in log-likelihood = ", del,
        " (eps = ", eps, ")\n")
  }
}

## Final values
eta <- linkfun(mu)
dmudeta <- mu.eta(eta)
llik <- ff$llik
theta <- th$theta
bPrime <- th$bPrime
bPrime2 <- th$bPrime2
fTilt <- th$fTilt[cbind(ySptIndex, seq_along(ySptIndex))]]

## Add NA values back into beta vector and varbeta if covariate matrix is not full rank
nBeta <- length(beta) + sum(naID)
betaTemp <- rep(NA, nBeta)
betaTemp[!naID] <- beta
beta <- betaTemp

```

```

## Return gldrm object
attributes(beta) <- NULL
attributes(f0) <- NULL

fit <- list(conv=conv, iter=iter, llik=llik,
            beta=beta, mu=mu, eta=eta, f0=f0, spt=spt, mu0=mu0,
            theta=theta, bPrime=bPrime, bPrime2=bPrime2, fTilt=fTilt, weights=weights)

if (returnfTiltMatrix)
  fit$fTiltMatrix <- t(th$fTilt)

if (returnf0ScoreInfo) {
  fit$score.logf0 <- ff$score.log
  fit$info.logf0 <- ff$info.log
}

class(fit) <- "gldrm"
fit
}
◇

```

File defined by 1, 4, 6, 11.

"../R/wgldrm.R" 4≡

```

#' Beta optimization routing
#'
#' @param x Covariate matrix.
#' @param y Response vector.
#' @param spt Vector of unique observed support points in the response.
#' @param ySptIndex Index of each \code{y} value within the \code{spt} vector.
#' @param f0 Current values of f0.
#' @param linkinv Inverse link function.
#' @param mu.eta Derivative of inverse link function.
#' @param offset Vector of known offset values to be added to the linear
#' combination (x' beta) for each observation. Mostly intended for likelihood ratio
#' and score confidence intervals.
#' @param sampprobs Optional matrix of sampling probabilities.
#' @param betaStart Starting values for beta (typically the estimates from the
#' previous iteration).
#' @param thStart Starting theta values. Needs to be a list of values matching
#' the output of the \code{getTheta} function.
#' @param thetaControl A "thetaControl" object returned from the \code{theta.control}
#' function.
#' @param betaControl A "betaControl" object returned from the \code{beta.control}
#' function.
#'
#' @return A list containing the following:
#' \itemize{
#' \item \code{beta} Updated values.
#' \item \code{mu} Updated mean for each observation.
#' \item \code{th} Updated list returned from the \code{getTheta} function.
#' \item \code{llik} Updated log-likelihood.
#' \item \code{iter} Number of iterations until convergence. (Will always be
#' one unless \code{maxiter} is increased to something greater than one using the
#' \code{betaControl} argument.)
#' \item \code{conv} Convergence indicator. (Will always be FALSE unless
#' \code{maxiter} is increased to something greater than one using the
#' \code{betaControl} argument.)

```

```

#' }
#'
#' @keywords internal
beta.control <- function (eps = 1e-10, maxiter = 1, maxhalf = 10)
{
  betaControl <- as.list(environment())
  class(betaControl) <- "betaControl"
  betaControl
}

getBeta <- function(x, y, spt, ySptIndex, f0, linkinv, mu.eta, offset, weights,
  betaStart, thStart,
  thetaControl=theta.control(), betaControl=beta.control())
{
  ## Extract control arguments
  if (class(betaControl) != "betaControl")
    stop("betaControl must be an object of class betaControl returned by betaControl() function.")
  eps <- betaControl$eps
  maxiter <- betaControl$maxiter
  maxhalf <- betaControl$maxhalf

  sptMin <- min(spt)
  sptMax <- max(spt)
  beta <- betaStart
  th <- thStart
  llik <- th$llik

  conv <- FALSE
  maxhalfreached <- FALSE
  iter <- 0
  while (!conv && !maxhalfreached && iter < maxiter)
  {
    iter <- iter+1

    ## Update mean vector and related quantities
    eta <- c(x %*% beta + offset)
    mu <- linkinv(eta)
    dmudeta <- mu.eta(eta)
    betaold <- beta
    muold <- mu
    thold <- th
    llikold <- llik

    ## Compute weighted least squares update
    w <- weights * dmudeta^2 / th$bPrime2
    ymm <- y - mu
    r <- ymm / dmudeta

    yeqmu <- which(abs(ymm) < 1e-15)
    w[yeqmu] <- 0 # prevent 0/0
    r[yeqmu] <- 0 # prevent 0/0

    if (any(w==Inf)) break

    betastep <- unname(coef(lm.wfit(x, r, w)))
    betastep[is.na(betastep)] <- 0

    ## Let q = b''*(theta) / b''(theta)
    ## W = diag{dmudeta^2 / b''(theta) * q}

```

```

## r = (y - b'*(theta)) / (q * dmudeta)
## We need to solve for beta such that I(betaHat) %*% beta = Score(betaHat),
## or equivalently, X'WX = X'Wr, or equivalently  $W^{1/2}X = W^{1/2}r$ .
## The linear system can be solved using qr.coef().

### Update beta and take half steps if log-likelihood does not improve
beta <- beta + betastep
eta <- c(x %*% beta + offset)
mu <- linkinv(eta)
if (min(mu)<sptMin || max(mu)>sptMax) {
  llik <- -Inf
} else {
  th <- getTheta(spt=spt, f0=f0, mu=mu, weights=weights, ySptIndex=ySptIndex,
    thetaStart=thold$theta, thetaControl=thetaControl)
  llik <- th$llik
}

nhalf <- 0
while ((llik<llikold) && (nhalf<maxhalf)) {
  nhalf <- nhalf + 1
  beta <- (beta + betaold) / 2
  eta <- c(x %*% beta + offset)
  mu <- linkinv(eta)
  if (min(mu)<sptMin || max(mu)>sptMax) {
    llik <- -Inf
  } else {
    th <- getTheta(spt=spt, f0=f0, mu=mu, weights=weights, ySptIndex=ySptIndex,
      thetaStart=thold$theta, thetaControl=thetaControl)
    llik <- th$llik
  }
}

if (llik < llikold) {
  beta <- betaold
  mu <- muold
  th <- thold
  llik <- llikold
  conv <- FALSE
  maxhalfreached <- TRUE
} else {
  del <- (llik - llikold) / llik
  if (llik == 0) del <- 0 # consider converged if model fit is perfect
  conv <- del < eps
}

}

return(list(beta=beta, mu=mu, th=th, llik=llik, iter=iter, conv=conv))
}
◇

```

File defined by [1](#), [4](#), [6](#), [11](#).

"../R/wgldrm.R" 6≡

```

## usethis namespace: start
#' @useDynLib exchreg, .registration = TRUE
#' @importFrom Rcpp sourceCpp
## usethis namespace: end

```

```

## Computes log(sum(exp(x))) with better precision
logSumExp <- function(x)
{
  i <- which.max(x)
  m <- x[i]
  lse <- log1p(sum(exp(x[-i]-m))) + m
  lse
}

## g function (logit transformation from appendix)
g <- function(mu, m, M) log(mu-m) - log(M-mu)

#' Control arguments for \eqn{\theta} update algorithm
#'
#' This function returns control arguments for the \eqn{\theta} update algorithm.
#' Each argument has a default value, which will be used unless a different
#' value is provided by the user.
#'
#'@param eps Convergence threshold for theta updates. Convergence is
#' evaluated separately for each observation. An observation has converged when
#' the difference between \eqn{b'(\theta)} and \eqn{\mu} is less than \code{epsTheta}.
#'@param maxiter Maximum number of iterations.
#'@param maxhalf Maximum number of half steps allowed per iteration if the
#' convergence criterion does not improve.
#'@param maxtheta Absolute value of theta is not allowed to exceed \code{maxtheta}.
#'@param logit Logical for whether logit transformation should be used. Use of
#' this stabilizing transformation appears to be faster in general. Default is TRUE.
#'@param logsumexp Logical argument for whether log-sum-exp trick should be used.
#' This may improve numerical stability at the expense of computational time.
#'
#' @return Object of S3 class "thetaControl", which is a list of control arguments.
#'
#' @keywords internal
theta.control <- function(eps=1e-10, maxiter=100, maxhalf=20, maxtheta=500,
                          logit=TRUE, logsumexp=FALSE)
{
  thetaControl <- as.list(environment())
  class(thetaControl) <- "thetaControl"
  thetaControl
}

#' getTheta
#' Updates theta. Vectorized but only updates observations that have not converged.
#'
#'@param spt Support of the observed response variable. (This is the set of
#' unique values observed, not the set of all possible values.)
#'@param f0 Values of the baseline distribution corresponding to the values of spt
#'@param mu The fitted mean for each observation. Note these values must lie
#' strictly within the range of the support.
#'@param sampprobs Matrix of sampling probabilities. The number of rows should
#' equal the number of observations, and the number of columns should equal
#' the number of unique observed support points.
#'@param ySptIndex Vector containing index of each observation's response value
#' within the \code{spt} vector. This is only needed to calculate the log-likelihood
#' after each update.
#'@param thetaStart Vector of starting values. One value per observation. If
#' \code{NULL}, zero is used as the starting value for each observation.
#'@param thetaControl Object of class \code{thetaControl}, which is a list of
#' control arguments returned by the \code{thetaControl} function.

```

```

#'
#' @return List containing the following:
#' \itemize{
#' \item \code{theta} Updated values.
#' \item \code{fTilt} Matrix containing the exponentially tilted distribution for each
#' observation, i.e.  $f(y|X=x)$ . Each column corresponds to an observation and sums to one.
#' \item \code{bPrime} Vector containing the mean of the exponentially tilted distribution
#' for each observation. Should match {mu} argument very closely.
#' \item \code{bPrime2} Vector containing the variance of the exponentially tilted
#' distribution for each observation.
#' matches {bPrime2}.
#' \item \code{llik} Semiparametric log-likelihood, evaluated at the current beta
#' and f0 values. If sampling weights are used, then the log-likelihood is conditional
#' on each observation being sampled.
#' \item \code{conv} Convergence indicator.
#' \item \code{iter} Number of iterations until convergence was reached.
#' }
#'
#' @keywords internal
getTheta <- function(spt, f0, mu, weights, ySptIndex, thetaStart=NULL, thetaControl=theta.control())
{
  ## Extract control arguments
  if (class(thetaControl) != "thetaControl")
    stop("thetaControl must be an object of class \'thetaControl\' returned by
    thetaControl() function.")

  ## Define value from inputs
  sptN <- length(spt)
  m <- min(spt)
  M <- max(spt)
  n <- length(mu)

  ## Format arguments
  spt <- as.vector(spt)
  f0 <- as.vector(f0)
  mu <- as.vector(mu)
  if (!is.null(thetaStart)) {
    thetaStart <- as.vector(thetaStart)
  } else {
    thetaStart <- rep(0, n)
  }

  ## Argument checks
  if (length(f0) != sptN)
    stop("spt and f0 must be vectors of equal length.")
  if (any(f0 < 0))
    stop("f0 values cannot be negative.")
  if (min(mu)<m || max(mu)>M)
    stop("mu starting values must lie within the range of spt.")
  if (length(thetaStart) != n)
    stop("thetaStart must be a vector with length equal length(mu)")

  res <- getThetaC(spt, f0, mu, thetaStart, thetaControl)

  ## Calculate log-likelihood
  fTilt.extracted <- res$fTilt[cbind(ySptIndex, seq_along(ySptIndex))]
  llik <- sum(weights[fTilt.extracted>0] * log(fTilt.extracted[fTilt.extracted>0]))

  list(theta=c(res$theta), fTilt=res$fTilt, bPrime=c(res$bPrime), bPrime2=c(res$bPrime2),

```



```

        llik=llik, conv=c(res$conv), iter=res$iter)
    }
    ◇

```

File defined by 1, 4, 6, 11.

The computationally intensive part of `getTheta` is implemented in C++. For now, only the `logsumexp = FALSE` and `logit = FALSE` settings are implemented.

"../src/getTheta.cpp" 9≡

```

#include <RcppArmadillo.h>
using namespace Rcpp;
using namespace arma;
// [[Rcpp::depends(RcppArmadillo)]]

// Computes log(sum(exp(x))) with better precision
double logSumExp(arma::vec x){
    uword i = x.index_max();
    double m = x(i);
    x.shed_row(i);
    double lse = log1p(sum(exp(x-m))) + m;
    return lse;
}

// g function (logit transformation from appendix)
arma::vec g(arma::vec mu, double m, double M){
    arma::vec res = log(mu-m) - log(M-mu);
    return res;
}

// [[Rcpp::export]]
List getThetaC(arma::vec spt, arma::vec f0, arma::vec mu,
               arma::vec thetaStart, List thetaControl){
    // bool logit = thetaControl["logit"];
    double eps = thetaControl["eps"];
    int maxiter = thetaControl["maxiter"];
    double maxhalf = thetaControl["maxhalf"];
    double maxtheta = thetaControl["maxtheta"];
    // bool logsumexp = thetaControl["logsumexp"];

    int sptN = spt.size();
    // double m = min(spt);
    // double M = max(spt);
    int n = mu.size();

    //initialize values
    arma::vec theta = thetaStart;
    arma::vec thetaOld(n);
    arma::vec bPrimeErrOld(n);
    uvec conv(n, fill::zeros);
    uvec maxedOut(n, fill::zeros);

    arma::rowvec oo(sptN, fill::ones);

    mat fUnstd = exp(spt * theta.t());
    fUnstd.each_col() %= f0; // |spt| x n matrix of tilted f0 values
    rowvec b = oo * fUnstd; //column sums;

```

```

mat fTilt = fUnstd.each_row() / b; // normalized

colvec bPrime = fTilt.t() * spt; // mean as a function of theta
colvec bPrime2(n); // variance as a function of theta
for (int j=0; j<n; j++){ //iterate over the columns of fTilt
    bPrime2(j) = 0;
    for (int i=0; i<sptN; i++){ //iterate over the rows of fTilt
        bPrime2(j) += pow(spt(i) - bPrime(j), 2.0) * fTilt(i,j);
    }
}
colvec bPrimeErr = bPrime - mu; // used to assess convergence

conv = (abs(bPrimeErr) < eps) || (theta==maxtheta && bPrimeErr<0) ||
        (theta==maxtheta && bPrimeErr>0);
uvec s = find(conv == 0);
int iter = 0;

while(s.size() > 0 && iter < maxiter) {
    iter++;
    bPrimeErrOld(s) = bPrimeErr(s); // used to assess convergence

// 1) Update theta
    thetaOld(s) = theta(s);
    colvec thetaS = theta(s) - bPrimeErr(s) / bPrime2(s);
    thetaS(find(thetaS > maxtheta)).fill(maxtheta);
    thetaS(find(thetaS < -maxtheta)).fill(-maxtheta);
    theta(s) = thetaS;

// 2) Update fTilt, bPrime, and bPrime2 and take half steps if bPrimeErr not improved
    uvec ss = s;
    int nhalf = 0;
    while(ss.size() > 0 && nhalf < maxhalf) {
// 2a) Update fTilt, bPrime, and bPrime2
        fUnstd.cols(ss) = exp(spt * theta(ss).t());
        fUnstd.each_col(ss) %= f0; // |spt| x n matrix of tilted f0 values
        b = oo * fUnstd.cols(ss); //column sums;
        mat tmp = fUnstd.cols(ss);
        fTilt.cols(ss) = tmp.each_row() / b; // normalized

        bPrime(ss) = fTilt.cols(ss).t() * spt; // mean as a function of theta
        for (uword j=0; j<ss.size(); j++){ //iterate over the columns of fTilt[,ss]
            bPrime2(ss(j)) = 0;
            for (int i=0; i<sptN; i++){ //iterate over the rows of fTilt[,ss]
                bPrime2(ss(j)) += pow(spt(i) - bPrime(ss(j)), 2.0) * fTilt(i,ss(j));
            }
        }
        bPrimeErr(ss) = bPrime(ss) - mu(ss);

// 2b) Take half steps if necessary
        ss = ss(find(abs(bPrimeErr(ss)) > abs(bPrimeErrOld(ss))));
        if (ss.size() > 0){
            nhalf++;
        }
        theta(ss) = (theta(ss) + thetaOld(ss)) / 2;
    }
}
// If maximum half steps are exceeded, set theta to previous value
maxedOut(ss).fill(1);
theta(ss) = thetaOld(ss);

```

```

// 3) Check convergence
conv(s) = (abs(bPrimeErr(s)) < eps) || (theta(s)==maxtheta && bPrimeErr(s) < 0) ||
  (theta(s)==-maxtheta && bPrimeErr(s) > 0);
s = s(find(conv(s) == 0 && maxedOut(s) == 0));
}

List res;
res["theta"] = theta;
res["fTilt"] = fTilt;
res["bPrime"] = bPrime;
res["bPrime2"] = bPrime2;
res["conv"] = conv;
res["iter"] = iter;
return res;
}

◇

"../R/wgldrm.R" 11≡

#' Control arguments for f0 update algorithm
#'
#' This function returns control arguments for the  $f_0$  update algorithm.
#' Each argument has a default value, which will be used unless a different
#' value is provided by the user.
#'
#' @param eps Convergence threshold. The update has converged when the relative
#' change in log-likelihood between iterations is less than {eps}.
#' absolute change is less than {thesh}.
#' @param maxiter Maximum number of iterations allowed.
#' @param maxhalf Maximum number of half steps allowed per iteration if
#' log-likelihood does not improve between iterations.
#' @param maxlogstep Maximum optimization step size allowed on the
#' {log(f0)} scale.
#'
#' @return Object of S3 class "f0Control", which is a list of control arguments.
#'
#' @keywords internal
f0.control <- function(eps=1e-10, maxiter=1000, maxhalf=20, maxlogstep=2)
{
  f0Control <- as.list(environment())
  class(f0Control) <- "f0Control"
  f0Control
}

#' f0 optimization routine
#'
#' @param y Vector of response values.
#' @param spt Vector of unique observed support points in the response.
#' @param ySptIndex Index of each {y} value within {spt}.
#' @param sptFreq.weighted Vector containing weighted frequency of each {spt} value.
#' @param mu Fitted mean for each observation. Only used if {sampprobs=NULL}.
#' @param mu0 Mean constraing for f0.
#' @param f0Start Starting f0 values. (Typically the estimate from the previous
#' iteration.)
#' @param thStart Starting theta values. Needs to be a list of values matching

```

```

#' the output of the \code{getTheta} function.
#' @param thetaControl A "thetaControl" object returned from the \code{theta.control}
#' function.
#' @param f0Control An "f0Control" object returned from the \code{f0.control}
#' function.
#' trace Logical. If TRUE, then progress is printed to terminal at each iteration.
#'
#' @return A list containing the following:
#' \itemize{
#' \item \code{f0} Updated values.
#' \item \code{llik} Updated log-likelihood.
#' \item \code{th} Updated list returned from the \code{getTheta} function.
#' \item \code{conv} Convergence indicator.
#' \item \code{iter} Number of iterations until convergence.
#' \item \code{nhalf} The number of half steps taken on the last iteration if the
#' initial BFGS update did not improve the log-likelihood.
#' \item \code{score.log} Score function with respect to  $\log(f_0)$  at convergence.
#' \item \code{info.log} Information matrix with respect to  $\log(f_0)$  at convergence.
#' }
#'
#' @keywords internal
getf0 <- function(y, spt, ySptIndex, weights, sptFreq.weighted, mu, mu0, f0Start, thStart,
                  thetaControl=theta.control(), f0Control=f0.control(), trace=FALSE)
{
  ## Extract theta control arguments
  if (class(f0Control) != "f0Control")
    stop("f0Control must be an object of class f0Control returned by f0Control() function.")
  eps <- f0Control$eps
  maxiter <- f0Control$maxiter
  maxhalf <- f0Control$maxhalf
  maxlogstep <- f0Control$maxlogstep

  f0 <- f0Start # assumes  $\text{sum}(f0Start) = 1$  and  $\text{sum}(f0Start * spt) = \mu_0$ 
  th <- thStart
  llik <- th$llik
  score.log <- NULL
  smm <- outer(spt, mu, "-")
  ymm <- y - mu
  yeqmu <- which(abs(ymm) < 1e-15)

  conv <- FALSE
  iter <- 0
  while (!conv && iter < maxiter) {
    iter <- iter + 1

    # Score calculation
    score.logOld <- score.log

    fTiltSums <- rowSums(th$fTilt)
    fTiltSumsWeighted <- apply(th$fTilt, MARGIN=1, function(x) sum(weights * x))
    smmfTilt <- smm * th$fTilt
    ystd <- ymm / th$bPrime2
    ystdWeighted <- weights * ystd
    ystd[yeqmu] <- 0 # prevent 0/0
    ystdWeighted[yeqmu] <- 0 # prevent 0/0

    score.logT1 <- sptFreq.weighted

```

```

score.logT2 <- fTiltSumsWeighted
score.logT3 <- c(smmfTilt %*% ystdWeighted)
score.log <- score.logT1 - score.logT2 - score.logT3

# Inverse info, score step, and f0 step are on the log scale (score is not)
if (iter == 1) {
  d1 <- min(fTiltSumsWeighted) # max inverse diagonal of first information term, on log scale
  d2 <- max(abs(score.log)) / maxlogstep
  d <- max(d1, d2)
  infoinvBFGS.log <- diag(1/d, nrow=length(f0))
} else {
  scorestep.log <- score.log - score.logOld
  # f0step.log <- log(f0) - log(f0old)
  # to prevent the 0/0 situation
  ratioOf0f0old <- f0 / f0old
  ratioOf0f0old[is.na(ratioOf0f0old)] <- 1
  f0step.log <- log(ratioOf0f0old)

  sy <- sum(f0step.log * scorestep.log)
  yiy <- c(crossprod(scorestep.log, infoinvBFGS.log %*% scorestep.log))
  iys <- tcrossprod(infoinvBFGS.log %*% scorestep.log, f0step.log)
  infoinvBFGS.log <- infoinvBFGS.log + ((yiy - sy) / sy^2) * tcrossprod(f0step.log) - (1 / sy)
}
logstep <- c(infoinvBFGS.log %*% score.log)

# Cap log(f0) step size
logstep.max <- max(abs(logstep))
if (logstep.max > maxlogstep)
  logstep <- logstep * (maxlogstep / logstep.max)

# Save values from previous iteration
f0old <- f0
thold <- th
llikold <- llik

# Take update step
f0 <- exp(log(f0) + logstep)
# Scale and tilt f0
f0 <- f0 / sum(f0)
f0 <- getTheta(spt=spt, f0=f0, mu=mu0, weights=weights, ySptIndex=1,
               thetaStart=0, thetaControl=thetaControl)$fTilt[, 1]
# Update theta and likelihood
thold <- th
llikold <- llik
th <- getTheta(spt=spt, f0=f0, mu=mu, weights=weights, ySptIndex=ySptIndex,
               thetaStart=th$theta, thetaControl=thetaControl)
llik <- th$llik
conv <- abs((llik - llikold) / (llik + 1e-100)) < eps

# If log-likelihood does not improve, change step direction to be along gradient
# Take half steps until likelihood improves
# Continue taking half steps until log likelihood no longer improves
nhalf <- 0
if (llik < llikold) {
  llikprev <- -Inf
  while ((llik < llikold || llik > llikprev) && nhalf < maxhalf) {
    nhalf <- nhalf + 1

    # Set previous values

```

```

    llikprev <- llik
    thprev <- th
    f0prev <- f0
    infoinvBFGS.logprev <- infoinvBFGS.log

    f0 <- exp((log(f0) + log(f0old)) / 2)
    f0 <- f0 / sum(f0)
    f0 <- getTheta(spt=spt, f0=f0, mu=mu0, weights=weights, ySptIndex=1,
                    thetaStart=0, thetaControl=thetaControl)$fTilt[, 1]
    th <- getTheta(spt=spt, f0=f0, mu=mu, weights=weights, ySptIndex=ySptIndex,
                    thetaStart=th$theta, thetaControl=thetaControl)
    llik <- th$llik
    infoinvBFGS.log <- infoinvBFGS.log / 2
  }

  if (llik < llikprev) {
    nhalf <- nhalf - 1
    llik <- llikprev
    th <- thprev
    f0 <- f0prev
    infoinvBFGS.log <- infoinvBFGS.logprev
  }

  conv <- abs((llik - llikold) / (llik + 1e-100)) < eps
}

if (llik < llikold) {
  f0 <- f0old
  th <- thold
  llik <- llikold
  conv <- TRUE
}

if (trace) {
  printout <- paste0("iter ", iter, ": llik=", llik)
  if (nhalf > 0)
    printout <- paste0(printout, "; ", nhalf, " half steps")
  cat(printout, "\n")
}

# Final score calculation
smm <- outer(spt, th$bPrime, "-")
ymm <- y - th$bPrime
yeqmu <- which(abs(ymm) < 1e-15)

fTiltSums <- rowSums(th$fTilt)
fTiltSumsWeighted <- apply(th$fTilt, MARGIN=1, function(x) sum(weights * x))
smmfTilt <- smm * th$fTilt
ystd <- ymm / th$bPrime2
ystdWeighted <- weights * ystd
ystd[yeqmu] <- 0 # prevent 0/0
ystdWeighted[yeqmu] <- 0 # prevent 0/0
ystd[yeqmu] <- 0 # prevent 0/0

score.logT1 <- sptFreq.weighted
score.logT2 <- fTiltSumsWeighted
score.logT3 <- c(smmfTilt %*% ystdWeighted)
score.log <- score.logT1 - score.logT2 - score.logT3

```

```
# Final info calculation
info.logT1 <- diag(fTiltSumsWeighted)
info.logT2 <- tcrossprod(th$fTilt)
info.logT3 <- tcrossprod(smmfTilt, smmfTilt * rep(ystdWeighted, each=nrow(smmfTilt)))
info.log <- info.logT1 - info.logT2 - info.logT3
```

```
list(f0=f0, llik=llik, th=th, conv=conv, iter=iter, nhalf=nhalf,
      score.log=score.log, info.log=info.log)
```

```
}
```

```
◇
```

File defined by [1](#), [4](#), [6](#), [11](#).