San Francisco Estuary Regional Monitoring Program for Trace Substances

Estimation of Means, Totals, and Distribution Functions from Probability Survey Data

Submitted to the San Francisco Estuary Regional Monitoring Program for Trace Substances

Submitted by
Don L. Stevens, Jr.

SFEI Contribution 110
June 2002

San Francisco Estuary Institute

This report should be cited as:
Don L. Stevens. 2002. Estimation of Means, Totals, and Distribution Functions from Probability Survey Data. RMP Technical Report: SFEI Contribution #110. San Francisco Estuary Institute, Oakland, CA.

Estimation of Means, Totals, and Distribution Functions from Probability Survey Data

Techical Report
June 20, 2002

Submitted to
San Francisco Estuary Regional Monitoring Program for Trace Substances

Submitted by
Don L. Stevens, Jr.

Estimation of Means, Totals, and Distribution Functions from Probability Survey Data

Statistical Framework

We base the development here on the case of a response $z(s)$ defined on a region R that is a subset of a universe U, which we assume is a 2- or 3-dimensional continuum. Our objective is to estimate some properties of the response on R. In particular, we want estimates of the total Z_T, the mean value $\mu_z(R)$ and the distribution function $F_z(x)$ of the response over R. We define these by $Z_T = \int_R z(s)ds$, $\mu_z(R) = \frac{Z_T}{|R|}$, and $F_z(x) = \frac{1}{|R|}\int_R I_{\{s|z(s)\le x\}}(s)ds$ where $|R|$ denotes the size (length, area, volume) of R and $I_A(x)$ is the *indicator function* for A, that is, it indicates whether the point x is in the set A. Formally, $I_A(x)$ is defined as $I_A(x) = \begin{cases} 1, x \in A \\ 0, \text{otherwise} \end{cases}$. Because $\int_R I_{\{s|z(s)\le x\}}(s)ds$ is the size of the set for which the response meets the condition $z(s) \le x$, the distribution function $F_z(x)$ measures the fraction of R for which the condition is met.

A probability sample from R is a set $S = \{s_1, s_2, ..., s_n\}$ of n random points in U. The usual requirement for a probability sample that is the probability distribution of the sample be known. We are basing the development here on the assumption that we are sampling a spatial continuum, so we assume that we know (or can calculate) the *spatial sampling intensity* function (also called the *inclusion probability density* function) $\pi(s)$. The function $\pi(s)$ describes the average density of our sampling points, and has units of number of points per unit length or area. Thus, for a stream sample, $\pi(s)$ would have units of number of points per kilometer of stream.

Estimation of Totals and Means under Variable Probability Sampling Designs

Horvitz and Thompson (HT)(1952) provided an estimator of the population total for variable-probability, without-replacement, finite-population sampling designs, along with an expression for the variance of the estimated total and a related variance estimator. Cordy (1993) showed that a version of the HT theorem holds when sampling from a continuum U. The continuous version of the HT theorem provides estimators of the total (integral) of z over R An estimate of the mean is obtained by dividing the estimated total by $|R|$, the size of R. An alternative estimator of the mean, called a *ratio estimator,* uses the estimated size of R as the divisor. As in the finite population case, the ratio estimator of the mean (also known as the Hájek estimator (Hájek, 1971; Thompson, 1992)) tends to be nearly unbiased and less variable because of positive correlation between the numerator and denominator. It is also well-suited to subpopulation estimation, as the size of subpopulation domain need not be known.

Let $s_1, s_2, ..., s_n$ be a sample selected from a universe U according to a design with inclusion function $\pi(s)$. For an arbitrary region $R \subset U$, an unbiased estimator of $\int_R z(s)\, ds = Z_T$ is

(1) $$\hat{Z}_T = \sum_{i=1}^{n} \frac{I_R(s_i)\, z(s_i)}{\pi(s_i)}.$$

An (approximately) unbiased ratio estimator of mean value of *z*, i.e., of $\mu_z(R) = \int_R z(s)ds \,/\, |R|$ is

(2) $$\hat{\mu}_z = \sum_{i=1}^{n} \frac{I_R(s_i)\, z(s_i)}{\pi(s_i)} \,/\, |\hat{R}| = \frac{\hat{Z}_T}{|\hat{R}|},$$

where $|\hat{R}| = \sum_{i=1}^{n} \frac{I_R(s_i)}{\pi(s_i)}$.

The inclusion density specifies the number of points per unit of population, e.g., number of points per mile of stream. Its reciprocal, then, specifies the units of population per point, e.g., the miles of stream per point. Thus, the reciprocal of the inclusion density of a point specifies the amount or weight of the population represented by that point. We can use this observation to re-express **(1)** and **(2)** as weighted sums by letting $w(s_i) = \frac{1}{\pi(s_i)}$:

(3) $$\hat{Z}_T = \sum_{i=1}^{n} I_R(s_i)\, w(s_i) z(s_i)$$

and

(4) $$\hat{\mu}_z = \frac{\sum_{i=1}^{n} I_R(s_i) w(s_i) z(s_i)}{\sum_{i=1}^{n} I_R(s_i) w(s_i)} = \frac{\hat{Z}_T}{|\hat{R}|}.$$

The spatially restricted design used to select the ODFW sample has inclusion functions that are constant within Monitoring Areas (MAs). If the region *R* is an MA (or is contained within a single MA), then the ratio estimator of the mean value is identical to the usual estimator, i.e., it reduces to the sum of the observations divided by the number of observations. However, if *R* includes points from several MAs, that is, if all of the points do not have the same inclusion probability, then the general formula given above must be used.

Variance Estimation

The spatial balance inherent in the GRTS design will give a more precise (less variable) estimate of the mean than would a simple random sample of the same size, if the response has some spatial pattern. While the spatial balance will generally lead to more precision, it also complicates estimation of that precision. Intuitively, this happens because the locations of the sample points are not independent of one another, and that dependence must be taken into account in estimating variance. Horvitz and Thompson (1952) provided a general formula for the variance of the estimated mean from a probability sample that accounts for the pairwise dependence of the points, along with an unbiased estimator of the variance. Alternative expressions for the variance and its estimator have been provided by Yates and Grundy (1953) and Sen (1953). These estimators do not work well for a GRTS design because, although theoretically unbiased, they tend to be very unstable, sometimes even taking on negative values.

A simplified and stable estimator for the variance of the mean can be obtained by treating the sample as if it arose from independent random sampling (IRS), where the *n* points are selected independently from an arbitrary density *f(s)* over *U*. In the approximation, *n* is the number of points in the sample from the universe, not n_R, the number of points in the sample that fall in *R*. If we use this approximation as a variance estimator for a subpopulation, we need to recognize the fact that n_R is a random variable. The most straightforward way to do that is to view the variance estimator as conditional on the achieved sample size. This changes the interpretation slightly, but it makes the computation much simpler. The difference in interpretation is that sampling variance describes the variation in the estimator over repeated selections of the sample. The sampling variance conditional on the achieved sample size describes variation in the estimator over the restricted set of repeated selections of the sample that result in the same achieved sample size for *R*.

If we adopt this approach, and set the "*n*" equal to the achieved sample size in the subpopulation we are dealing with, then the IRS variance estimator for the total is

(5) $$\hat{V}_{IRS}(\hat{Z}_T) = nV_{SRS}(z/\pi) = nV_{SRS}(wz),$$

where $V_{SRS}(z/\pi)$ is the usual estimator of the population variance for an SRS design applied to $z(s_i)/\pi(s_i) = w(s_i)z(s_i)$. $V_{SRS}(X)$ is the default variance estimator available in most statistical software packages.

Furthermore, if the inclusion density is constant on the subpopulation (as it is within an MA), then the result further simplifies to

(6) $$\hat{V}_{IRS}(\hat{Z}_T) = nV_{SRS}(z)/\pi^2 = nw^2V_{SRS}(z)$$

as an estimator of the variance of the estimated total $\hat{Z}_T$. It follows that the corresponding variance estimator for $\hat{\mu}_z$ is the usual SRS estimator for the variance of the mean:

(7) $$\hat{V}_{IRS}(\hat{\mu}_z) = V_{SRS}(z)/n.$$

The IRS estimator does not account for the spatially constrained nature of the design. If the response has some spatial pattern, at least to the extent that two points close together tend to be more similar than two points far apart, then the spatially balanced design will lead to more precise estimates than independent random sampling. Thus, the IRS estimator will be conservative, i.e., it will tend to overstate the variance.

An approximately unbiased estimator of variance can be based on the observation that the spatially constrained nature of the design ensures that any arbitrary subset of the domain will have an achieved sample size nearly equal to its expected sample size (Stevens and Olsen, in review, 2002). If we were to split the population domain up into small neighborhoods, each with an expected sample size of, say, 4 to 5 points, then every replication of the design would place some points in each of the small neighborhoods. Because we can break down the estimated total into the sum of the estimated totals in each of the small neighborhoods, we can break down the variance of the total into the sum of variances of the neighborhood totals. This is the concept behind the neighborhood variance estimator $\hat{V}_{NB}$. The formula for $\hat{V}_{NB}$ is

(8) $$\hat{V}_{NB}(\hat{Z}_T)=\sum_{s_i\in R}\sum_{s_j\in D(s_i)} w_{ij}\left\{\frac{z(s_j)}{\pi(s_j)}-\sum_{s_k\in D(s_i)} w_{ik}\frac{z(s_k)}{\pi(s_k)}\right\}^2$$

In this formula, $D(s_i)$ is a neighborhood around the sample point s_i, and the w_{ij} are weights that depend on the design. The neighborhoods are defined so that each neighborhood contains at least 4 sample points, and satisfies $s_j \in D(s_i) \Leftrightarrow s_i \in D(s_j)$. The neighborhoods $D(s_i)$ are developed by initially including the point itself plus the next 3 nearest neighbors for each point, and then adding to $D(s_i)$ any points s_j such that $s_i \in D(s_j)$. This ensures the condition that $s_j \in D(s_i) \Leftrightarrow s_i \in D(s_j)$. The weights w_{ij} are selected using the following criteria:

1. The weight w_{ij} should be inversely proportional to $\pi(s_j)$ and decrease as the distance between s_i and s_j increases.

2. $\sum_j w_{ij} = \sum_i w_{ij} = 1$, so that the neighborhood totals are averages over the neighborhoods, and the sum of the neighborhood totals is equal to the estimated overall total.

The weights are developed by first assigning a value that decreases as the rank of the distance between s_j and s_i among the points in $D(s_i)$ increases and is inversely proportional to $\pi(s_j)$. The formula for the this first step is

$$w^*_{ij} = \frac{1-(\text{rank}(s_j)-1)/\text{count}(D(s_i))}{\pi(s_j)}$$

For example, if $D(s_1)$ contained 5 points, the points would be ranked 1 through 5 in order of their distance from s_1. Of course, s_1 receives rank 1, since it is the closest point to itself. The other 4 points would be ranked in terms of increasing distance from s_1. If all of the points have the same inclusion density, say $\pi(s_j) \equiv \pi$, then the point with rank 4 would get weight $\frac{(1-(4-1)/5)}{\pi} = \frac{2/5}{\pi}$. The weights are normalized to satisfy the constraint on the column totals by setting $\tilde{w}_{ij} = \frac{w^*_{ij}}{\sum_{s_k \in D(s_i)} w^*_{ik}}$. There is no unique way to satisfy both constraints in criterion (2), so we select the set of weights w_{ij} that minimize $\sum_{i,j} (w_{ij} - \tilde{w}_{ij})^2$ while satisfying criteria (2). The constrained minimization problem is solved using Lagrange multipliers. The unconstrained minimization problem is then

$$\min_{w_{ij}, \lambda_k, \gamma_l} \sum_{i,j} (w_{ij} - \tilde{w}_{ij})^2 + \sum_k \lambda_k (\sum_i w_{ij} - 1) + \sum_l \gamma_l (\sum_k w_{jk} - 1)$$

The w_{ij} are easily eliminated from the set of linear equations obtained by setting derivatives equal to 0. The resulting set of equations in λ_k and γ_l is singular, and the Moore-Penrose generalized inverse (Rao and Mitra, 1971) is used to obtain a unique solution for $\hat{\lambda}_k$ and $\hat{\gamma}_l$. The minimizing set of weights is

$$w_{ij} = w^*_{ij} + \frac{\hat{\lambda}_i + \hat{\gamma}_j}{2}.$$

Confidence Interval Estimation

A strictly correct confidence interval for an estimator would be based on the sampling distribution of the estimator, i.e., the distribution of the estimator over repeated sample selections. The sampling distribution depends on the sampling design, and on the distribution of the underlying population. There is no general, straightforward way to obtain sampling distributions when the underlying population distribution is unknown.

The standard approach to getting an approximate confidence interval is to appeal to the Central Limit Theorem, which, loosely speaking, says that the distribution of a sum of random variables becomes approximately normal as the number of terms in the sum increases. Fortunately, the estimators of primary interest (totals, means, and proportions) are sums of random variables, so the Central Limit Theorem is relevant. In practice, we assume that the sampling distribution of our estimator is approximately normal, and appeal to the Central Limit Theorem to justify our assumption. Given the assumption of approximate normality of the sampling distribution (not the underlying population distribution), we can obtain approximate confidence intervals by characterizing the shape of the distribution via the variance of the estimator.

The general form of the approximation is as follows. Let θ be a population characteristic we wish to estimate, $\hat{\theta}$ be our estimator, and $\hat{V}(\hat{\theta})$ be the estimated variance. An approximate p% confidence interval is given by

(9) $$\left(\hat{\theta} - Z(p)\sqrt{\hat{V}(\hat{\theta})}, \hat{\theta} + Z(p)\sqrt{\hat{V}(\hat{\theta})} \right)$$

where *Z(p)* denotes the appropriate percentile of the of the standard normal distribution. (For a 95% confidence interval, use *Z(p)* = 1.96; for a 90% confidence interval, use *Z(p)*

= 1.65.). The same formula holds whether θ is a total, mean, or proportion. The estimated variance will be calculated differently for the three cases, just as the estimators themselves are.

Subpopulation Estimation

The estimation equations **(1)** through **(9)** can be used to estimate the proportion of a population that meets some criteria or falls within some category, along with a corresponding confidence interval. For example, we may be interested in the proportion of the Central Bay target population that is shallow water, or the proportion with copper concentration less than *x*. To do this, we form a new response variable that takes on the value 1 if a sample site meets the criteria or is in the category, and 0 otherwise. We call this new response the *indicator variable* for the criteria or category. For the category {shallow water}, the indicator variable is $I_{shallow}(s_i) = \begin{cases} 1, \text{ if } s_i \text{ is in shallow water} \\ 0, \text{ if } s_i \text{ otherwise} \end{cases}$ The mean value of the indicator variable is the proportion we want, and we estimate it and its variance using the same method as for any other mean. Thus, for example, $\hat{p}_{\text{shallow}} = \dfrac{\sum_{\text{i}} I_{shallow}(s_i) w(s_i)}{\sum_i w(s_i)}$ would give the estimated proportion of shallow water in the Central Bay segment.

The indicator variable technique can be used to obtain an estimate of the entire population distribution via the *cumulative distribution function* or *cdf*. The cdf for a variable *z* , say $F_z(x)$, gives the proportion of the population with *z* value less than or equal to *x*. For example, if *z* is spawner density in spawners/km, then $F_Z(3)$ is the proportion of the population with 3 or fewer spawners per kilometer. We estimate the cdf of *z* by picking a set of levels $x_1, x_2, \ldots, x_k$ that span the range of *z*, and then estimating the mean values of the indicator variables $I_{z \le x_j}(s_i) = \begin{cases} 1, \text{ if } z(s_i) \le x_j \\ 0, \text{ otherwise} \end{cases}$, so that

(10) $$\hat{F}_z(x_j) = \frac{\sum_i I_{z \le x_j}(s_i) w(s_i)}{\sum_i w(s_i)}.$$

The concept of the indicator variable is very simple, but it is in fact a very powerful tool for doing exploratory and comparative analyses of a complex probability sample. For example, the formulae above show how to compute the cdf for the entire population, e.g., the entire San Francisco Estuary. But we can also use an indicator variable to estimate the cdf for a subset of the population. For example, suppose we want the cdf of copper concentration in shallow water. We use the "shallow" indicator variable in the cdf estimator equation to get

(11) $$\hat{F}_{z|shallow}(x_j) = \frac{\sum_i I_{z \le x_j}(s_i) I_{\text{shallow}}(s_i) w(s_i)}{\sum_i I_{\text{shallow}}(s_i) w(s_i)}.$$

At any particular value x_j, $\hat{F}_{z|shallow}(x_j)$ gives the estimated proportion of the shallow water of the Bay with copper concentration less than or equal to x_j. We could also calculate the cdf for deep water using a "deep" indicator variable, and compare the two cdfs. One way to make a quick and informative visual comparison is to calculate the two subpopulation cdfs at the same levels of the x-variable (spawner density in the example), and then plot corresponding values against one another, producing a plot known as a Q-Q plot (Q for "quantile"). If the two distributions are approximately equal, then they should plot on roughly a 1-1 line.

Computational Algorithms and Computer Software

Most widely available statistical analysis software packages do not have the capability to correctly analyze data from a complex probability survey. Some will handle weighted estimation properly, but none will properly calculate variance. The appendix provides S+ function source code for carrying out all the computations described above for probability-based surveys of the type used by EMAP that incorporate spatial balance. The code should also work under R, which is avialable for free download from http://cran.us.r-project.org/.

Electronic versions of the functions are available on request.

References

Cordy, C. (1993). ‘An extension of the Horvitz-Thompson theorem to point sampling from a continuous universe'. *Probability and Statistics Letters* **18**, 353–362.

Hájek, J. (1971). ‘Comment on a paper by D. Basu. In: Godambe, V. P., and Sprott, D. A. (eds.) *Foundations of Statistical Inference*. Toronto: Holt, Rinehart, and Winston, p. 236.

Horvitz, D.G. and D.J. Thompson. (1952). ‘A generalization of sampling without replacement from a finite universe'. *Journal of the American Statistical Association* **47**, 663–685.

Rao, C. R. and Mitra, S. K. 1971 *Generalized Inverse of Matrices and its Applications.* Wiley, New York

Sen, A.R. (1953). ‘On the estimate of the variance in sampling with varying probabilities’. *Journal of the Indian Society of Agricultural Statistics* **7**, 119–127.

Thompson, S.K. (1992). *Sampling*. New York: John Wiley & Sons.

Stevens, Jr., D.L., and A. R. Olsen. (In review, 2002). ‘Variance Estimation for Spatially Balanced Samples of Environmental Resources’

Yates, F. and P.M. Grundy. (1953). ‘Selection without replacement from within strata with probability proportional to size'. *Journal of the Royal Statistical Society* **B15**, 253–261.

Appendix: S+ Source Code for Analyzing Data fro Probability Surveys.

```
# PROGRAM:
```
pop.size.est.fcn
```
# Date:    April 20, 2001
# Description:
#   This function estimates the population size for each site status category
#   plus upper and lower confidence bounds.  For a discrete resource size is the
#   number of units in the resource.  For an extensive resource size is the
#   extent (measure) of the resource, i.e., length, area, or volume.  Size
#   estimates are calculated using the Horvitz-Thompson estimator.  Variance
#   estimates for the size estimates are calculated using either the neighborhood
#   variance estimator or the simple random sampling (SRS) variance estimator.
#   The choice of variance estimator is subject to user control.  The neighborhood
#   variance estimator requires the x-coordinate and the y-coordinate of each
#   site.  The SRS variance estimator uses the independent random sample
#   approximation to calculate joint inclusion probabilities.  Confidence bounds
#   are calculated using a Normal distribution multiplier.  In addition the
#   function estimates the population proportion for each site status category
#   plus upper and lower confidence bounds.  Proportion estimates are calculated
#   using the Horvitz-Thompson ratio estimator, i.e., the ratio of two Horvitz-
#   Thompson estimators.  The numerator of the ratio estimates the size of the
#   site status category. The denominator of the ratio estimates the size of the
#   resource.  Variance estimates for the proportion estimates are calculated
#   using either the neighborhood variance estimator or the simple random sampling
#   (SRS) variance estimator.  The function can accommodate a stratified sample.
#   For a stratified sample, separate estimates and confidence bounds are
#   calculated for each stratum and for all strata combined.  The function
#   checks for compatibility of input values, checks for a null site status
#   category and deletes the category as necessary, and removes missing values.
#   Input:
#     status = site status for each site
#     wgt = the weight (inverse of the sample inclusion probability) for each
#       site
#     x = x-coordinate of each site (the default is NULL)
#     y = y-coordinate of each site (the default is NULL)
#     stratum = the stratum for each site (the default is 1 for every site)
#     vartype = indicator for variance estimator ("Local"=neighborhood estimator
#       and "SRS"=SRS estimator, the default is "Local")
#     conf = the confidence level (the default is 95%)
#   Output:
#     An object in list format containing the estimates and confidence bounds.
#     If the sample was stratified, the list includes one element for each
#     stratum plus an additional element for all strata combined.  The elements
#     in the list are named Stratum1, Stratum2, etc., and the last element in
#     the list is named AllStrata.  For a stratified sample, each element in
#      the list contains two data frames: one data frame for the population size
#      estimates (named size.est) and a second data frame for the population
#      proportion estimates (named prop.est).  If the sample was not stratified,
#      the list contains two elements: a data frame for the  population size
#      estimates (named size.est) and a data frame for the population proportion
#      estimates (named prop.est).
#   Other Functions Required:
#      wnas.fcn - remove missing values
#      localmean.fcn - calculate the neighborhood variance estimator
#   Example Function Calls:
#      pop.size.est.fcn(mydata$status, mydata$weight, mydata$x,
#         mydata$y, mydata$stratum)
#      pop.size.est(mydata$status, mydata$weight, "mydata.status",
#         mydata$stratum, vartype="SRS")

pop.size.est.fcn <- function(status, wgt, x=NULL, y=NULL, stratum=1, vartype="Local",  conf=95)
{

# Calculate additional required values

   objname <- "Results"
   nstrata <- max(stratum)
# Remove missing values

   if(vartype == "Local") {
      temp <- wnas.fcn(list(status=status, wgt=wgt, x=x, y=y, stratum=stratum))
      status <- temp$status
      wgt <- temp$wgt
      x <- temp$x
      y <- temp$y
      stratum <- temp$stratum
   } else {
      temp <- wnas.fcn(list(status=status, wgt=wgt, stratum=stratum))
      status <- temp$status
      wgt <- temp$wgt
      stratum <- temp$stratum
   }

# Check for compatibility of input values

   if(length(status) != length(wgt))
      stop("\n\nNumber of status values must equal the number of weights.")
   if(min(wgt) <= 0)
      stop("\n\nWeights must be positive.")
   if(vartype == "Local") {
      if (length(x) == 0 || length(y) == 0)
         stop("\n\nx-coordinate and y-coordinate values are required for the neighborhood variance
estimator.")
      else if (length(z) != length(x) || length(z) != length(y))
         stop("\n\nNumber of response values must equal the number of x-coordinate and y-coordinate
values for the neighborhood variance estimator.")
   }
   if(nstrata > 1 && length(status) != length(stratum))
      stop("\n\nNumber of status values must equal the number of stratum values.")

# Determine levels of the status variable

   status.levels <- levels(category(status))
   nlevels <- length(status.levels)

# Calculate total of the weights

   tw <- sum(wgt)

# Calculate confidence bound multiplier

   mult <- qnorm(0.5 + (conf/100)/2)

# Branch to handle stratified and unstratified data

   if (nstrata > 1) {

# Begin the section for stratified data

# Create the object for the estimates

   eval(parse(text=paste(objname, " <- vector('list', nstrata+1)", sep="")))
   temp <- list(c(paste("Stratum", 1:nstrata, sep=""), "AllStrata"))
   eval(parse(text=paste("names(", objname, ") <- ", temp, sep="")))

# Begin the subsection for population size estimates

# Create the matrix for all strata combined

   rsltall <- array(0, c(5, nlevels+1))
   dimnames(rsltall) <- list(c("No. of Sites  ", "Pop. Size Estimate  ", "Standard Deviation  ",
paste("Upper ", conf, "% Conf. Bound  ", sep=""), paste("Lower ", conf, "% Conf. Bound  ",
sep="")), c(status.levels, "Total"))

# Begin the subsection for individual strata
```

```
  for (i in 1:nstrata) {

# Estimate the number of values and size of each site status category

  z <- category(status[stratum == i])
  n <- length(z)
  z.levels <- levels(z)
  m <- length(z.levels)
  w <- wgt[stratum == i]
  nval <- c(tapply(w, z, length), n)
  size <- c(tapply(w, z, sum), sum(w))

# Calculate the weighted indicator matrix

  im <- cbind(tapply(w, z) == matrix(rep(1:m, n), nrow=n, byrow=T), rep(1, n)) * matrix(rep(w,
m+1), nrow=n)

# Calculate the standard diviation

  if (vartype == "Local" && n < 4) {
    warning("\nThere are less than four site status values, the simple random sampling variance
estimator will be used\n\n")
    vartype <- "SRS"
  }
  if (vartype == "Local") {
    sdest <- sqrt(apply(im, 2, localmean.fcn, x=x[stratum == i], y=y[stratum == i], prb=(1/w)))
  } else {
    sdest <- sqrt(n * apply(im, 2, var))
  }

# Calculate confidence bounds

  ubound <- size + mult*sdest
  lbound <- pmax(size - mult*sdest, 0)

# Combine estimates and confidence bounds in a matrix

  rslt <- rbind(nval, size, sdest, ubound, lbound)
  dimnames(rslt)[[2]][m+1] <- "Total"
  if (m < nlevels) {
    temp <- array(0, c(5, nlevels+1))
    k <- 1
    for (j in 1:nlevels) {
      if (z.levels[k] != status.levels[j]) {
        temp[,j] <- rep(NA, 5)
       } else {
         temp[,j] <- rslt[,k]
         k <- k+1
       }
    }
    temp[,nlevels+1] <- rslt[,m+1]
    rslt <- temp
    dimnames(rslt)[[2]] <- list(c(status.levels, "Total"))
  }
  dimnames(rslt)[[1]] <- c("No. of Sites  ", "Pop. Size Estimate  ", "Standard Deviation  ",
paste("Upper ", conf, "% Conf. Bound  ", sep=""), paste("Lower ", conf, "% Conf. Bound  ",
sep=""))

# Append results to the object

  eval(parse(text=paste(objname, "$Stratum", i, "$size.est <- data.frame(rslt, check.names=F)",
sep="")))

# Add estimates to the matrix for all strata combined

  rsltall[1:2,][rslt[1:2,] != "NA"] <- rsltall[1:2,][rslt[1:2,] != "NA"] + rslt[1:2,][rslt[1:2,] != "NA"]
  rsltall[3,][rslt[3,] != "NA"] <- rsltall[3,][rslt[3,] != "NA"] + rslt[3,][rslt[3,] != "NA"]^2

# End the subsection for individual strata

  }

# Begin the subsection for all strata combined

  rsltall[3,] <- sqrt(rsltall[3,])

# Calculate confidence bounds

  rsltall[4,] <- rsltall[2,] + mult*rsltall[3,]
  rsltall[5,] <- pmax(rsltall[2,] - mult*rsltall[3,], 0)

# Append results to the object

  eval(parse(text=paste(objname, "$AllStrata", "$size.est <- data.frame(rsltall, check.names=F)",
sep="")))

# End the subsection for all strata combined

# End the subsection for population size estimates
# Begin the subsection for population proportion estimates

# Create the matrix for all strata combined

   rsltall <- array(0, c(5, nlevels))
   dimnames(rsltall) <- list(c("No. of Sites  ", "Pop. Proportion Estimate  ", "Standard Deviation  ",
paste("Upper ", conf, "% Conf. Bound  ", sep=""), paste("Lower ", conf, "% Conf. Bound  ",
sep="")), status.levels)

# Begin the subsection for individual strata

   for (i in 1:nstrata) {

# Calculate the proportion estimates

   z <- category(status[stratum == i])
   n <- length(z)
   z.levels <- levels(z)
   m <- length(z.levels)
   w <- wgt[stratum == i]
   nval <- tapply(w, z, length)
   prop <- tapply(w, z, sum) / tw

# Calculate the weighted residuals matrix

   im <- tapply(w, z) == matrix(rep(1:m, n), nrow=n, byrow=T)
   rm <- (im - matrix(rep(prop, n), nrow=n, byrow=T)) * matrix(rep(w, m), nrow=n)

# Calculate the standard deviation

   if (vartype == "Local" && n < 4) {
      warning("\nThere are less than four site status values, the simple random sampling variance
estimator will be used\n\n")
      vartype <- "SRS"
   }
   if (vartype == "Local") {
      sdest <- sqrt(apply(rm, 2, localmean.fcn, x=x[stratum == i], y=y[stratum == i], prb=(1/w)) /
(tw^2))
   } else {
      sdest <- sqrt(n * apply(rm, 2, var) / (tw^2))
   }

# Calculate confidence bounds
```

```
   ubound <- pmin(prop + mult*sdest, 1)
   lbound <- pmax(prop - mult*sdest, 0)

# Combine estimates and confidence bounds in a matrix

   rslt <- rbind(nval, prop, sdest, ubound, lbound)
   if (m < nlevels) {
      temp <- array(0, c(5, nlevels))
      k <- 1
      for (j in 1:nlevels) {
         if (z.levels[k] != status.levels[j]) {
            temp[,j] <- rep(NA, 5)
          } else {
             temp[,j] <- rslt[,k]
             k <- k+1
          }
      }
      rslt <- temp
      dimnames(rslt)[[2]] <- list(status.levels)
   }
   dimnames(rslt)[[1]] <- c("No. of Sites  ", "Pop. Proportion Estimate  ", "Standard Deviation  ",
paste("Upper ", conf, "% Conf. Bound  ", sep=""), paste("Lower ", conf, "% Conf. Bound  ",
sep=""))

# Append results to the object

   eval(parse(text=paste(objname, "$Stratum", i, "$prop.est <- data.frame(rslt, check.names=F)",
sep="")))

# Add estimates to the matrix for all strata combined

   rsltall[1:2,][rslt[1:2,] != "NA"] <- rsltall[1:2,][rslt[1:2,] != "NA"] + rslt[1:2,][rslt[1:2,] != "NA"]
   rsltall[3,][rslt[3,] != "NA"] <- rsltall[3,][rslt[3,] != "NA"] + rslt[3,][rslt[3,] != "NA"]^2

# End the subsection for individual strata

   }

# Begin the subsection for all strata combined

   rsltall[3,] <- sqrt(rsltall[3,])

# Calculate confidence bounds

   rsltall[4,] <- pmin(rsltall[2,] + mult*rsltall[3,], 1)
   rsltall[5,] <- pmax(rsltall[2,] - mult*rsltall[3,], 0)

# Append results to the object

   eval(parse(text=paste(objname, "$AllStrata", "$prop.est <- data.frame(rsltall, check.names=F)",
sep="")))

# End the subsection for all strata combined

# End the subsection for population proportion estimates

# End the section for stratified data

   } else {

# Begin the section for unstratified data

# Create the object for the estimates

   eval(parse(text=paste(objname, " <- vector('list', 2)", sep="")))
   temp <- list(c("size.est", "prop.est"))
   eval(parse(text=paste("names(", objname, ") <- ", temp, sep="")))

# Begin the subsection for population size estimates

# Estimate the number of values and size of each site status category

   z <- category(status)
   n <- length(z)
   z.levels <- levels(z)
   m <- length(z.levels)
   w <- wgt
   nval <- c(tapply(w, z, length), n)
   size <- c(tapply(w, z, sum), tw)

# Calculate the weighted indicator matrix

   im <- cbind(tapply(w, z) == matrix(rep(1:m, n), nrow=n, byrow=T), rep(1, n)) * matrix(rep(w,
m+1), nrow=n)

# Calculate the standard diviation

   if (vartype == "Local" && n < 4) {
      warning("\nThere are less than four site status values, the simple random sampling variance
estimator will be used\n\n")
```

```
      vartype <- "SRS"
   }
   if (vartype == "Local") {
      sdest <- sqrt(apply(im, 2, localmean.fcn, x=x, y=y, prb=(1/w)))
   } else {
      sdest <- sqrt(n * apply(im, 2, var))
   }

# Calculate confidence bounds

   ubound <- size + mult*sdest
   lbound <- pmax(size - mult*sdest, 0)

# Combine estimates and confidence bounds in a matrix

   rslt <- rbind(nval, size, sdest, ubound, lbound)
   dimnames(rslt)[[1]] <- c("No. of Sites  ", "Pop. Size Estimate  ", "Standard Deviation  ",
paste("Upper ", conf, "% Conf. Bound  ", sep=""), paste("Lower ", conf, "% Conf. Bound  ",
sep=""))
   dimnames(rslt)[[2]][m+1] <- "Total"

# Append results to the object

   eval(parse(text=paste(objname, "$size.est <- data.frame(rslt, check.names=F)", sep="")))

# End the subsection for population size estimates

# Begin the subsection for population proportion estimates

# Calculate the proportion of each site status category

   prop <- tapply(w, z, sum) / tw

# Calculate the weighted residuals matrix

   im <- tapply(w, z) == matrix(rep(1:m, n), nrow=n, byrow=T)
   rm <- (im - matrix(rep(prop, n), nrow=n, byrow=T)) * matrix(rep(w, m), nrow=n)

# Calculate the standard diviation

   if (vartype == "Local" && n < 4) {
      warning("\nThere are less than four site status values, the simple random sampling variance
estimator will be used\n\n")
      vartype <- "SRS"
   }
   if (vartype == "Local") {
      sdest <- sqrt(apply(rm, 2, localmean.fcn, x=x, y=y, prb=(1/w)) / (tw^2))
   } else {
      sdest <- sqrt(n * apply(rm, 2, var) / (tw^2))
   }

# Calculate confidence bounds

   ubound <- pmin(prop + mult*sdest, 1)
   lbound <- pmax(prop - mult*sdest, 0)

# Combine estimates and confidence bounds in a matrix

   rslt <- rbind(nval[-(m+1)], prop, sdest, ubound, lbound)
   dimnames(rslt)[[1]] <- c("No. of Sites  ", "Pop. Proportion Estimate  ", "Standard Deviation  ",
paste("Upper ", conf, "% Conf. Bound  ", sep=""), paste("Lower ", conf, "% Conf. Bound  ",
sep=""))

# Append results to the object

   eval(parse(text=paste(objname, "$prop.est <- data.frame(rslt, check.names=F)", sep="")))

# End the subsection for population proportion estimates

# End the section for unstratified data
   }

# Return the object

   eval(parse(text=objname))
}
```

```
# Program:
```
cat.prop.fcn

```
# Date:    April 20, 2001
# Description:
#   This function estimates proportion of the target population in sub-
#   categories plus upper and lower confidence bounds.  Proportions are
#   estimated using the Horvitz-Thompson ratio estimator, i.e., the ratio of two
#   Horvitz-Thompson estimators.  Variance estimates for the proportions are
#   calculated using either the neighborhood variance estimator or the simple
#   random sampling (SRS) variance estimator.  The choice of variance estimator
#   is subject to user control.  The neighborhood variance estimator requires the
#   x-coordinate and the y-coordinate of each site.  The SRS variance estimator
#   uses the independent random sample approximation to calculate joint
#   inclusion probabilities.  Confidence bounds are calculated using a Normal
#   distribution multiplier.  For a stratified sample, separate estimates and
#   confidence bounds are calculated for each stratum and for all strata
#   combined.  The function checks for compatibility of input values and removes
#   missing values.
#   Input:
#     catvar = the category variable value for each site
#     wgt = the weight (inverse of the sample inclusion probability) for each
#         site
#     x = x-coordinate of each site (the default is NULL)
#     y = y-coordinate of each site (the default is NULL)
#     stratum = the stratum for each site (the default is 1 for every site)
#     vartype = indicator for variance estimator ("Local"=neighborhood estimator
#       and "SRS"=SRS estimator, the default is "Local")
#     conf = the confidence level (the default is 95%)
#   Output:
#     An object in list format containing the estimates and confidence bounds.
#     If the sample was stratified, the list includes one element for each
#     stratum plus an additional element for all strata combined.  The elements
#     in the list are named Stratum1, Stratum2, etc., and the last element in
#     the list is named AllStrata.  For a stratified sample, each element in
#     the list contains a data frame (named estimates).  If the sample was not
#     stratified, the list contains a single data frame (named estimates).
#   Other Functions Required:
#     wnas.fcn - remove missing values
#     localmean.fcn - calculate the neighborhood variance estimator
#   Example Function Call:
#     cat.prop.fcn(mydata$group, mydata$weight, mydata$x, mydata$y,
#       mydata$stratum)

cat.prop.fcn <- function(catvar, wgt, x=NULL, y=NULL, stratum=1, vartype="Local", conf=95) {
```

```
# Calculate additional required values

   objname <- "Results"
   nstrata <- max(stratum)

# Remove missing values

   if(vartype == "Local") {
      temp <- wnas.fcn(list(catvar=catvar, wgt=wgt, x=x, y=y, stratum=stratum))
      catvar <- temp$catvar
      wgt <- temp$wgt
      x <- temp$x
      y <- temp$y
      stratum <- temp$stratum
   } else {
      temp <- wnas.fcn(list(catvar=catvar, wgt=wgt, stratum=stratum))
      catvar <- temp$catvar
      wgt <- temp$wgt
      stratum <- temp$stratum
   }
# Check for compatibility of input values

   if(length(catvar) != length(wgt))
      stop("\n\nNumber of category values must equal the number of weights.")
   if(min(wgt) <= 0)
      stop("\n\nWeights must be positive.")
   if(vartype == "Local") {
      if (length(x) == 0 || length(y) == 0)
         stop("\n\nx-coordinate and y-coordinate values are required for the neighborhood variance
estimator.")
      else if (length(catvar) != length(x) || length(catvar) != length(y))
         stop("\n\nNumber of response values must equal the number of x-coordinate and y-coordinate
values for the neighborhood variance estimator.")
   }
   if(nstrata > 1 && length(catvar) != length(stratum))
      stop("\n\nNumber of category values must equal the number of stratum values.")

# Determine levels of the category variable

   catvar.levels <- levels(category(catvar))
   nlevels <- length(catvar.levels)

# Calculate total of the weights

   tw <- sum(wgt)
```

```
# Calculate confidence bound multiplier

   mult <- qnorm(0.5 + (conf/100)/2)

# Branch to handle stratified and unstratified data

   if (nstrata > 1) {

# Begin the section for stratified data

# Create the object for the estimates

   eval(parse(text=paste(objname, " <- vector('list', nstrata+1)", sep="")))
   temp <- list(c(paste("Stratum", 1:nstrata, sep=""), "AllStrata"))
   eval(parse(text=paste("names(", objname, ") <- ", temp, sep="")))

# Create the matrix for all strata combined

   rsltall <- array(0, c(4, nlevels))
   dimnames(rsltall) <- list(c("Category Estimate  ", "Standard Deviation  ", paste("Upper ", conf, "%
Conf. Bound  ", sep=""), paste("Lower ", conf, "% Conf. Bound  ", sep="")), catvar.levels)

# Begin the subsection for individual strata

   for (i in 1:nstrata) {

# Calculate the proportion estimates

   z <- category(catvar[stratum == i])
   n <- length(z)
   z.levels <- levels(z)
   m <- length(z.levels)
   w <- wgt[stratum == i]
   est <- tapply(w, z, sum) / tw

# Calculate the weighted residuals matrix

   im <- tapply(w, z) == matrix(rep(1:m, n), nrow=n, byrow=T)
   rm <- (im - matrix(rep(est, n), nrow=n, byrow=T)) * matrix(rep(w, m), nrow=n)

# Calculate the standard deviation

   if (vartype == "Local" && n < 4) {
      warning("\nThere are less than four category values, the simple random sampling variance
estimator will be used\n\n")
      vartype <- "SRS"
   }
   if (vartype == "Local") {
      sdest <- sqrt(apply(rm, 2, localmean.fcn, x=x[stratum == i], y=y[stratum == i], prb=(1/w)) /
(tw^2))
   } else {
      sdest <- sqrt(n * apply(rm, 2, var) / (tw^2))
   }

# Calculate confidence bounds

   ubound <- pmin(est + mult*sdest, 1)
   lbound <- pmax(est - mult*sdest, 0)

# Combine estimates and bounds

   rslt <- rbind(est, sdest, ubound, lbound)
   if (m < nlevels) {
      temp <- array(0, c(4, nlevels))
      k <- 1
      for (j in 1:nlevels) {
         if (z.levels[k] != catvar.levels[j]) {
            temp[,j] <- rep(NA, 4)
          } else {
            temp[,j] <- rslt[,k]
            k <- k+1
          }
      }
      rslt <- temp
      dimnames(rslt)[[2]] <- list(catvar.levels)
   }
   dimnames(rslt)[[1]] <- c("Category Estimate  ", "Standard Deviation  ", paste("Upper ", conf, "%
Conf. Bound  ", sep=""), paste("Lower ", conf, "% Conf. Bound  ", sep=""))

# Append results to the object

   eval(parse(text=paste(objname, "$Stratum", i, "$estimates <- data.frame(rslt, check.names=F)",
sep="")))

# Add estimates to the matrix for all strata combined

   rsltall[1,][rslt[1,] != "NA"] <- rsltall[1,][rslt[1,] != "NA"] + rslt[1,][rslt[1,] != "NA"]
   rsltall[2,][rslt[2,] != "NA"] <- rsltall[2,][rslt[2,] != "NA"] + rslt[2,][rslt[2,] != "NA"]^2
```

```
# End the subsection for individual strata

   }

# Begin the subsection for all strata combined

   rsltall[2,] <- sqrt(rsltall[2,])

# Calculate confidence bounds

   rsltall[3,] <- pmin(rsltall[1,] + mult*rsltall[2,], 1)
   rsltall[4,] <- pmax(rsltall[1,] - mult*rsltall[2,], 0)

# Append results to the object

   eval(parse(text=paste(objname, "$AllStrata", "$estimates <- data.frame(rsltall, check.names=F)",
sep="")))

# End the subsection for all strata combined

# End the section for stratified data

   } else {

# Begin the section for unstratified data

# Create the object for the estimates

   eval(parse(text=paste(objname, " <- vector('list', 1)", sep="")))
   temp <- list("estimates")
   eval(parse(text=paste("names(", objname, ") <- ", temp, sep="")))

# Calculate the proportion estimates

   z <- category(catvar)
   n <- length(z)
   z.levels <- levels(z)
   m <- length(z.levels)
   w <- wgt
   est <- tapply(w, z, sum) / tw

# Calculate the weighted residuals matrix

   im <- tapply(w, z) == matrix(rep(1:m, n), nrow=n, byrow=T)
   rm <- (im - matrix(rep(est, n), nrow=n, byrow=T)) * matrix(rep(w, m), nrow=n)

# Calculate the standard deviation

   if (vartype == "Local" && n < 4) {
      warning("\nThere are less than four category values, the simple random sampling variance
estimator will be used\n\n")
      vartype <- "SRS"
   }
   if (vartype == "Local") {
      sdest <- sqrt(apply(rm, 2, localmean.fcn, x=x, y=y, prb=(1/w)) / (tw^2))
   } else {
      sdest <- sqrt(n * apply(rm, 2, var) / (tw^2))
   }

# Calculate confidence bounds

   ubound <- pmin(est + mult*sdest, 1)
   lbound <- pmax(est - mult*sdest, 0)

# Combine estimates and confidence bounds

   rslt <- rbind(est, sdest, ubound, lbound)
   dimnames(rslt)[[1]] <- c("Category Estimate  ", "Standard Deviation  ", paste("Upper ", conf, "%
Conf. Bound  ", sep=""), paste("Lower ", conf, "% Conf. Bound  ", sep=""))

# Append results to the object

   eval(parse(text=paste(objname, "$estimates <- data.frame(rslt, check.names=F)", sep="")))

# End the section for unstratified data

   }

# Return the object

   eval(parse(text=objname))
}
```

```
# Program:
```
total.fcn
```
# Date:    April 20, 2001
# Description:
#   This function calculates an estimate of the total of a discrete or an
#   extensive resource.  The Horvitz-Thompson estimator is used to calculate the
#   estimate.  Variance estimates are calculated using either the neighborhood
#   variance estimator or the simple random sampling variance estimator.  The
#   choice of variance estimator is subject to user control. The neighborhood
#   variance estimator requires the x-coordinate and the y-coordinate of each
#   site.  The simple random sampling variance estimator uses the independent
#   random sample approximation to calculate joint inclusion probabilities.
#   Confidence bounds are calculated using a Normal distribution multiplier.
#   The function can accommodate a stratified sample.  For a stratified sample,
#   separate estimates and confidence bounds are calculated for each stratum and
#   for all strata combined.  The function checks for compatibility of input
#   values and removes missing values.
#   Input:
#     z = the response value for each site
#     wgt = the weight (inverse of the sample inclusion probability) for each
#         site
#     x = x-coordinate of each site (the default is NULL)
#     y = y-coordinate of each site (the default is NULL)
#     stratum = the stratum for each site (the default is 1 for every site)
#     vartype = indicator for variance estimator ("Local"=neighborhood estimator
#       and "SRS"=SRS estimator, the default is "Local")
#     conf = the confidence level (the default is 95%)
#   Output:
#     An object in data frame format containing the total estimate, standard
#     deviation, and confidence bounds.  If the sample was stratified, the data
#     frame includes one row for each stratum plus an additional row for all
#     strata combined.

total.fcn <- function(z, wgt, x=NULL, y=NULL, stratum=1, vartype="Local", conf=95) {

# Calculate the number of strata

  nstrata <- max(stratum)

# Remove missing values

  if (vartype == "Local") {
    temp <- wnas.fcn(list(z=z, wgt=wgt, x=x, y=y))
    z <- temp$z
```

```
      wgt <- temp$wgt
      x <- temp$x
      y <- temp$y
   } else {
      temp <- wnas.fcn(list(z=z, wgt=wgt))
      z <- temp$z
      wgt <- temp$wgt
   }

# Check for compatability of input values

   if (length(z) != length(wgt))
      stop("\n\nNumber of response values must equal the number of weights.")
   if (min(wgt) <= 0)
      stop("\n\nWeights must be positive.")
   if(vartype == "Local") {
      if (length(x) == 0 || length(y) == 0)
         stop("\n\nx-coordinate and y-coordinate values are required for the neighborhood variance
estimator.")
      else if (length(z) != length(x) || length(z) != length(y))
         stop("\n\nNumber of response values must equal the number of x-coordinate and y-coordinate
values for the neighborhood variance estimator.")
   }
   if (nstrata > 1 && length(z) != length(stratum))
         stop("\n\nNumber of response values must equal the number of stratum values.")

# Calculate confidence bound multiplier

   mult <- qnorm(0.5 + (conf/100)/2)

# Branch to handle stratified and unstratified data

   if (nstrata > 1) {

# Begin the section for stratified data

# Create the object for results

   rslt <- as.data.frame(array(0, c(4, nstrata+1)))
   dimnames(rslt) <- list(c("Estimate", "Standard Deviation", paste("Upper ", conf, "% Conf. Bound",
sep=""), paste("Lower ", conf, "% Conf. Bound", sep="")), c(paste("Stratum", 1:nstrata, sep=""),
"AllStrata"))

# Calculate total estimates for individual strata
```

```
   z.wt <- wgt*z
   rslt[1, 1:nstrata] <- tapply(z.wt, stratum, sum)

# Calculate standard deviation estimates for individual strata

   for (i in 1:nstrata) {

      n <- length(z.wt[stratum == i])
      if (vartype == "Local" && n < 4) {
         warning(paste("\nThere are less than four response values for Stratum ", i, ", the simple
random sampling variance estimator will be used\n\n", sep=""))
         vartype <- "SRS"
      }
      if (vartype == "Local")
         rslt[2,i] <- sqrt(localmean.fcn(z=z.wt[stratum == i], x=x[stratum == i], y=y[stratum == i],
prb=1/wgt[stratum == i]))
      else
         rslt[2,i] <- sqrt(n * var(z.wt[stratum == i]))

   }

# Calculate the total estimate for all strata combined

   rslt[1, nstrata+1] <- sum(rslt[1, 1:nstrata])

# Calculate the standard deviation estimate for all strata combined

   rslt[2, nstrata+1] <- sqrt(sum(rslt[2, 1:nstrata]^2))

# Calculate confidence bounds for the estimates

   rslt[3,] <- rslt[1,] + mult*rslt[2,]
   rslt[4,] <- pmax(rslt[1,] - mult*rslt[2,], 0)

# End the section for stratified data

   } else {

# Begin the section for unstratified data

# Create the object for results

   rslt <- as.data.frame(rep(0, 4))
   dimnames(rslt) <- list(c("Estimate", "Standard Deviation", paste("Upper ", conf, "% Conf. Bound",
sep=""), paste("Lower ", conf, "% Conf. Bound", sep="")), " ")
```

```
# Calculate the total estimate

  z.wt <- wgt*z
  rslt[1,] <- sum(z.wt)

# Calculate the standard deviation estimate

  n <- length(z.wt)
  if (vartype == "Local" && n < 4) {
    warning("\nThere are less than four response values, the simple random sampling variance
estimator will be used\n\n")
    vartype <- "SRS"
  }
  if (vartype == "Local")
    rslt[2,] <- sqrt(localmean.fcn(z=z.wt, x=x, y=y, prb=1/wgt))
  else
    rslt[2,] <- sqrt(n * var(z.wt))

# Calculate confidence bounds for the estimates

  rslt[3,] <- rslt[1,] + mult*rslt[2,]
  rslt[4,] <- pmax(rslt[1,] - mult*rslt[2,], 0)

# End the section for unstratified data

  }

# Return the object

  rslt
}
```

```
# Program:
mean.fcn
# Date:    April 20, 2001
# Description:
#   This function calculates an estimate of the mean of a discrete or an
#   extensive resource.  The Horvitz-Thompson ratio estimator, i.e., the ratio
#   of two Horvitz-Thompson estimators, is used to calculate the estimate.
#   Variance estimates are calculated using either the neighborhood variance
#   estimator or the simple random sampling variance estimator.  The choice of
#   variance estimator is subject to user control. The neighborhood variance
#   estimator requires the x-coordinate and the y-coordinate of each site.  The
#   simple random sampling variance estimator uses the independent random sample
#   approximation to calculate joint inclusion probabilities.  Confidence bounds
#   are calculated using a Normal distribution multiplier.  The function can
#   accommodate a stratified sample.  For a stratified sample, separate
#   estimates and confidence bounds are calculated for each stratum and for all
#   strata combined.  If the known extent of the resource is provided for each
#   stratum,  those values are used to produce stratum weights for calculating
#   estimates for all strata combined.  Otherwise, estimated values of the
#   extent of each stratum are used to produce stratum weights.  The function
#   checks for compatibility of input values and removes missing values.
#   Input:
#     z = the response value for each site
#     wgt = the weight (inverse of the sample inclusion probability) for each
#         site
#     x = x-coordinate of each site (the default is NULL)
#     y = y-coordinate of each site (the default is NULL)
#     stratum = the stratum for each site (the default is 1 for every site)
#     R = the known extent of the resource for a stratified sample - the number
#       of units of a discrete resource or the measure of a continuous
#       resource (the default is NULL)
#     vartype = indicator for variance estimator ("Local"=neighborhood estimator
#       and "SRS"=SRS estimator, the default is "Local")
#     conf = the confidence level (the default is 95%)
#   Output:
#     An object in data frame format containing the mean estimate, standard
#     deviation, and confidence bounds.  If the sample was stratified, the data
#     frame includes one row for each stratum plus an additional row for all
#     strata combined.

mean.fcn <- function(z, wgt, x=NULL, y=NULL, stratum=1, R=NULL, vartype="Local", conf=95)
{

# Calculate the number of strata
```

```
   nstrata <- max(stratum)

# Remove missing values

   if (vartype == "Local") {
      temp <- wnas.fcn(list(z=z, wgt=wgt, x=x, y=y))
      z <- temp$z
      wgt <- temp$wgt
      x <- temp$x
      y <- temp$y
   } else {
      temp <- wnas.fcn(list(z=z, wgt=wgt))
      z <- temp$z
      wgt <- temp$wgt
   }

# Check for compatability of input values

   if (length(z) != length(wgt))
      stop("\n\nNumber of response values must equal the number of weights.")
   if (min(wgt) <= 0)
      stop("\n\nWeights must be positive.")
   if(vartype == "Local") {
      if (length(x) == 0 || length(y) == 0)
         stop("\n\nx-coordinate and y-coordinate values are required for the neighborhood variance
estimator.")
      else if (length(z) != length(x) || length(z) != length(y))
         stop("\n\nNumber of response values must equal the number of x-coordinate and y-coordinate
values for the neighborhood variance estimator.")
   }
   if (nstrata > 1) {
      if(length(z) != length(stratum))
         stop("\n\nNumber of response values must equal the number of stratum values.")
      if(length(R) > 0) {
         if(length(R) != nstrata)
            stop("\n\nThe known extent of the resource must be provided for each stratum")
         if(min(R) <= 0)
            stop("\n\nThe known extent of the resource must be positive for each stratum")
      }
   }

# Calculate confidence bound multiplier

   mult <- qnorm(0.5 + (conf/100)/2)
```

```
# Calculate additional required values

   if (nstrata > 1) {
      tw <- tapply(wgt, stratum, sum)
      if (length(R) > 0) {
         sumR <- sum(R)
      } else {
         Rhat <- tapply(wgt, stratum, sum)
         sumRhat <- sum(wgt)
      }
   } else {
      tw <- sum(wgt)
   }

# Branch to handle stratified and unstratified data

   if (nstrata > 1) {

# Begin the section for stratified data

# Create the object for results

   rslt <- as.data.frame(array(0, c(4, nstrata+1)))
   dimnames(rslt) <- list(c("Estimate", "Standard Deviation", paste("Upper ", conf, "% Conf. Bound",
sep=""), paste("Lower ", conf, "% Conf. Bound", sep="")), c(paste("Stratum", 1:nstrata, sep=""),
"AllStrata"))

# Calculate mean estimates for individual strata

   rslt[1, 1:nstrata] <- tapply(wgt*z, stratum, sum) / tw

# Caclulate standard deviation estimates for individual strata

   for (i in 1:nstrata) {

      n <- length(z[stratum == i])
      rm <- (z[stratum == i] - rep(rslt[1,i], n)) * wgt[stratum == i]
      if (vartype == "Local" && n < 4) {
         warning(paste("\nThere are less than four response values for Stratum ", i, ", the simple
random sampling variance estimator will be used\n\n", sep=""))
         vartype <- "SRS"
      }
      if (vartype == "Local")
         rslt[2,i] <- sqrt(localmean.fcn(z=rm, x=x[stratum == i], y=y[stratum == i], prb=1/wgt[stratum
== i]) / (tw[i]^2))
```

```
      else
         rslt[2,i] <- sqrt(n * var(rm) / (tw[i]^2))

   }

# Calculate the mean estimate and standard deviation estimate for all strata combined

      if (length(R) > 0) {
         rslt[1, nstrata+1] <- sum((R/sumR)*rslt[1, 1:nstrata])
         rslt[2, nstrata+1] <- sqrt(sum(((R/sumR)*rslt[2, 1:nstrata])^2))
      } else {
         rslt[1, nstrata+1] <- sum((Rhat/sumRhat)*rslt[1, 1:nstrata])
         rslt[2, nstrata+1] <- sqrt(sum(((Rhat/sumRhat)*rslt[2, 1:nstrata])^2))
      }

# Calculate confidence bounds for the estimates

   rslt[3,] <- rslt[1,] + mult*rslt[2,]
   rslt[4,] <- pmax(rslt[1,] - mult*rslt[2,], 0)

# End the section for stratified data

   } else {

# Begin the section for unstratified data

# Create the object for results

   rslt <- as.data.frame(rep(0, 4))
   dimnames(rslt) <- list(c("Estimate", "Standard Deviation", paste("Upper ", conf, "% Conf. Bound",
sep=""), paste("Lower ", conf, "% Conf. Bound", sep="")), " ")

# Calculate the total estimate

   rslt[1,] <- sum(wgt*z) / tw

# Calculate the standard deviation estimate

   n <- length(z)
   rm <- (z - rep(rslt[1,], n)) * wgt
   if (vartype == "Local" && n < 4) {
      warning("\nThere are less than four response values, the simple random sampling variance
estimator will be used\n\n")
      vartype <- "SRS"
   }
   if (vartype == "Local")
      rslt[2,] <- sqrt(localmean.fcn(z=rm, x=x, y=y, prb=1/wgt) / (tw^2))
   else
      rslt[2,] <- sqrt(n * var(rm) / (tw^2))
# Calculate confidence bounds for the estimates

   rslt[3,] <- rslt[1,] + mult*rslt[2,]
   rslt[4,] <- pmax(rslt[1,] - mult*rslt[2,], 0)

# End the section for unstratified data

   }

# Return the object

   rslt
}
```

```
# Program:
```
cdf.est.fcn
```
# Date:    April 20, 2001
# Description:
#   This function calculates an estimate of the cumulative distribution function
#   (CDF) for the proportion or the total of a discrete or a continuous
#   resource.  Optionally, for a discrete resource, the size-weighted CDF can be
#   calculated.  In addition the standard deviation of the estimated CDF and
#   confidence bounds are calculated.  The user supplies the set of values at
#   which the CDF is estimated.  For the CDF of a proportion, the Horvitz-
#   Thompson ratio estimator, i.e., the ratio of two Horvitz-Thompson
#   estimators, is used to calculate the CDF estimate.  For the CDF of a total,
#   the user can supply the known size of the resource or the known sum of the
#   size-weights for the resource, as appropriate.  For the CDF of a total when
#   either the size of the resource or the sum of the size-weights for the
#   resource is provided, the classic ratio estimator is used to calculate the
#   CDF estimate, where that estimator is the product of the known value and the
#   Horvitz-Thompson ratio estimator.   For the CDF of a total when neither the
#   size of the resource nor the sum of the size-weights for the resource is
#   provided, the Horvitz-Thompson estimator is used to calculate the CDF
#   estimate.  Variance estimates for the estimated CDF are calculated using
#   either the neighborhood variance estimator or the simple random sampling
#   variance estimator.  The choice of variance estimator is subject to user
#   control. The neighborhood variance estimator requires the x-coordinate and the
#   y-coordinate of each site.  The simple random sampling variance estimator
#   uses the independent random sample approximation to calculate joint
#   inclusion probabilities.  Confidence bounds are calculated using a Normal
#   distribution multiplier.  In addition the function uses the estimated CDF to
#   calculate percentile estimates.  Estimated confidence bounds for the
#   percentile estimates are calculated.  The user supplies the set of values at
#   which percentiles estimates are desired.  Optionally, the user can use the
#   default set of percentiles.  The function can accommodate a stratified
#   sample.  For a stratified sample, separate estimates and confidence bounds
#   are calculated for each stratum and for all strata combined.  For a
#   stratified sample when either the size of the resource or the sum of the
#   size-weights for the resource is provided for each stratum, those values are
#   used as stratum weights for calculating the estimates and confidence bounds
#   for all strata combined.  For a stratified sample when neither the size of
#   the resource nor the sum of the size-weights for the resource is provided
#   for each stratum, estimated values are used as stratum weights for
#   calculating the estimates and confidence bounds for all strata combined.
#   The function checks for compatibility of input values and removes missing
#   values.
#   Input:
#      z = the response value for each site
#     wgt = the weight (inverse of the sample inclusion probability) for each
#       site
#     cdfval = the set of values at which the CDF is estimated
#     x = x-coordinate of each site (the default is NULL)
#     y = y-coordinate of each site (the default is NULL)
#     stratum = the stratum for each site (the default is 1 for every site)
#     prop = indicator for proportion or total (T=proportion and F=total,
#       the default is T)
#     R = the known extent of the resource - the number of units of a discrete
#       resource or the measure of a continuous resource, which must be a
#       vector for a stratified sample (the default is NULL)
#     size = indicator for a size-weighted CDF (T=size-weighted and F=not
#       size-weighted, the default is F)
#     swgt = the size-weight for each response value (the default is NULL)
#     W = the known sum of the size-weights for the resource, which must be a
#       vector for a stratified sample (the default is NULL)
#     vartype = indicator for variance estimator ("Local"=neighborhood estimator
#       and "SRS"=SRS estimator, the default is "Local")
#     conf = the confidence level (the default is 95%)
#     pctval = the set of values at which percentiles are estimated (default =
#       {5, 25, 50, 75, 95})
#   Output:
#     An object in list format containing the estimates and confidence bounds.
#     If the sample was stratified, the list includes one element for each
#     stratum plus an additional element for all strata combined.  The elements
#     in the list are named Stratum1, Stratum2, etc., and the last element in
#     the list is named AllStrata.  For a stratified sample, each element in
#     the list contains two data frames: one data frame for the CDf estimates
#     (named cdf.est) and a second data frame for the percentile estimates
#     (named pct.est).  If the sample was not stratified, the list contains two
#     elements: a data frame for the CDF estimates (named cdf.est) and a data
#     frame for the percentile estimates (named pct.est).
#   Other Functions Required:
#     wnas.fcn - remove missing values
#     cdf.prop.fcn - calculate the CDF for the proportion
#     cdf.total.fcn - calculate the CDF for the total
#     cdf.size.prop.fcn - calculate the size-weighted CDF for the proportion
#     cdf.size.total.fcn - calculate the size-weighted CDF for the total
#     cdfvar.prop.fcn - calculate variance of the CDF for the proportion
#     cdfvar.total.fcn - calculate variance of the CDF for the total
#     cdfvar.size.prop.fcn - calculate variance of the size-weighted CDF for
#       the proportion
#     cdfvar.size.total.fcn - calculate variance of the size-weighted CDF for
#       the total
#     localmean.fcn - calculate the neighborhood variance estimator
#     dist2full.fcn - convert a vector of distance measures to a matrix
#   Example Function Call:
#     cdf.est.fcn(mydata$response, mydata$weight, seq(0, 200, length=25),
#        mydata$x, mydata$y, size=T, mydata$sizewt)

cdf.est.fcn <- function(z, wgt, cdfval, x=NULL, y=NULL, stratum=1, prop=T, R=NULL, size=F,
swgt=NULL, W=NULL, vartype="Local", conf=95, pctval=c(5, 25, 50, 75, 95)) {

# Calculate additional required values

  objname <- "Results"
  nvec <- 1:length(cdfval)
  ncdfval <- length(cdfval)
  nstrata <- max(stratum)
  npctval <- length(pctval)
  if (length(R) > 1) {
    sumR <- sum(R)
  } else if (length(R) == 0) {
    if (nstrata > 1) {
      Rhat <- tapply(wgt, stratum, sum)
      sumRhat <- sum(wgt)
    } else {
      Rhat <- sum(wgt)
    }
  }
  if (length(W) > 1) {
    sumW <- sum(W)
  } else if (length(W) == 0) {
    if (nstrata > 1) {
      What <- tapply(wgt*swgt, stratum, sum)
      sumWhat <- sum(wgt*swgt)
    } else {
      What <- sum(wgt*swgt)
    }
  }

# Remove missing values

  if(vartype == "Local") {
    if (size) {
      temp <- wnas.fcn(list(z=z, wgt=wgt, x=x, y=y, stratum=stratum, swgt=swgt))
      z <- temp$z
      wgt <- temp$wgt
      x <- temp$x
      y <- temp$y
      stratum <- temp$stratum
      swgt <- temp$swgt
    } else {
      temp <- wnas.fcn(list(z=z, wgt=wgt, x=x, y=y, stratum=stratum))
      z <- temp$z
      wgt <- temp$wgt
      x <- temp$x
      y <- temp$y
      stratum <- temp$stratum
    }
  } else {
    if (size) {
      temp <- wnas.fcn(list(z=z, wgt=wgt, stratum=stratum, swgt=swgt))
      z <- temp$z
      wgt <- temp$wgt
      stratum <- temp$stratum
      swgt <- temp$swgt
    } else {
      temp <- wnas.fcn(list(z=z, wgt=wgt, stratum=stratum))
      z <- temp$z
      wgt <- temp$wgt
      stratum <- temp$stratum
    }
  }

# Check for compatibility of input values

  if(length(z) != length(wgt))
    stop("\n\nNumber of response values must equal the number of weights.")
  if(min(wgt) <= 0)
    stop("\n\nWeights must be positive.")
  if(vartype == "Local") {
    if (length(x) == 0 || length(y) == 0)
      stop("\n\nx-coordinate and y-coordinate values are required for the neighborhood variance
estimator.")
    else if (length(z) != length(x) || length(z) != length(y))
      stop("\n\nNumber of response values must equal the number of x-coordinate and y-coordinate
values for the neighborhood variance estimator.")
  }
  if (nstrata > 1) {
    if(length(z) != length(stratum))
      stop("\n\nNumber of response values must equal the number of stratum values.")
    if(!prop && !size && length(R) > 0) {
      if(length(R) != nstrata)
        stop("\n\nThe known extent of the resource must be provided for each stratum")
      if(min(R) <= 0)
        stop("\n\nThe known extent of the resource must be positive for each stratum")
    }
    if(size && length(z) != length(swgt))
      stop("\n\nNumber of response value must equal the number of size-weights")
    if(size && min(swgt) <= 0)
      stop("\n\nSize-weights must be positive.")
    if(!prop && size && length(W) > 0) {
      if(length(W) != nstrata)
        stop("\n\nThe known sum of the size-weights for the resource must be provided for each
stratum")
      if(min(W) <= 0)
        stop("\n\nThe known sum of the size-weights for the resource must be positive for each
stratum")
    }
  } else {
    if(!prop && !size && length(R) > 0) {
      if(length(R) != 1)
        stop("\n\nOnly a single value should be provided for the known extent of the resource")
      if(R <= 0)
        stop("\n\nThe known extent of the resource must be positive")
    }
    if(size && length(z) != length(swgt))
      stop("\n\nNumber of response value must equal the number of size-weights")
    if(size && min(swgt) <= 0)
      stop("\n\nSize-weights must be positive.")
    if(!prop && size && length(W) > 0) {
      if(length(W) != 1)
        stop("\n\nOnly a single value should be provided for the known sum of the size-weights for
the resource")
      if(W <= 0)
        stop("\n\nThe known sum of the size-weights for the resource must be positive")
    }
  }

# Calculate confidence bound multiplier

  mult <- qnorm(0.5 + (conf/100)/2)

# Branch to handle stratified and unstratified data

  if (nstrata > 1) {

# Begin the section for stratified data
# Create the object for the estimates

   eval(parse(text=paste(objname, " <- vector('list', nstrata+1)", sep="")))
   temp <- list(c(paste("Stratum", 1:nstrata, sep=""), "AllStrata"))
   eval(parse(text=paste("names(", objname, ") <- ", temp, sep="")))

# Begin subsection for CDF estimates

# Create the matrix for all strata combined

   rsltall <- array(0, c(length(cdfval), 5))
   dimnames(rsltall) <- list(NULL, c("Value", "CDF Estimate", "Standard Deviation", paste("Lower
", conf, "% Conf. Bound", sep=""), paste("Upper ", conf, "% Conf. Bound", sep="")))
   rsltall[,1] <- cdfval

# Begin the subsection for individual strata

   for (i in 1:nstrata) {

# Calculate the CDF estimates, standard deviation estimates, and confidence bounds

   if (size) {
      if (prop) {
         cdfest <- cdf.size.prop.fcn(z[stratum == i], wgt[stratum == i], swgt[stratum == i], cdfval)
         sdest <- sqrt(cdfvar.size.prop.fcn(z[stratum == i], x[stratum == i], y[stratum == i], wgt[stratum
== i], swgt[stratum == i], cdfval, cdfest, vartype))
         ubound <- pmin(cdfest + mult*sdest, 1)
         lbound <- pmax(cdfest - mult*sdest, 0)
      } else {
         cdfest <- cdf.size.total.fcn(z[stratum == i], wgt[stratum == i], swgt[stratum == i], cdfval, W[i])
         sdest <- sqrt(cdfvar.size.total.fcn(z[stratum == i], x[stratum == i], y[stratum == i], wgt[stratum
== i], swgt[stratum == i], cdfval, cdfest, W[i], vartype))
         if (length(W) > 0)
            ubound <- pmin(cdfest + mult*sdest, W[i])
         else
            ubound <- cdfest + mult*sdest
         lbound <- pmax(cdfest - mult*sdest, 0)
      }
   } else {
      if (prop) {
         cdfest <- cdf.prop.fcn(z[stratum == i], wgt[stratum == i], cdfval)
         sdest <- sqrt(cdfvar.prop.fcn(z[stratum == i], x[stratum == i], y[stratum == i], wgt[stratum ==
i], cdfval, cdfest, vartype))
         ubound <- pmin(cdfest + mult*sdest, 1)
         lbound <- pmax(cdfest - mult*sdest, 0)
      } else {
         cdfest <- cdf.total.fcn(z[stratum == i], wgt[stratum == i], cdfval, R[i])
         sdest <- sqrt(cdfvar.total.fcn(z[stratum == i], x[stratum == i], y[stratum == i], wgt[stratum ==
i], cdfval, cdfest, R[i], vartype))
         if (length(R) > 0)
            ubound <- pmin(cdfest + mult*sdest, R[i])
         else
            ubound <- cdfest + mult*sdest
         lbound <- pmax(cdfest - mult*sdest, 0)
      }
   }
   rslt <- cbind(cdfval, cdfest, sdest, lbound, ubound)
   dimnames(rslt)[[2]] <- c("Value", "CDF Estimate", "Standard Deviation", paste("Lower ", conf,
"% Conf. Bound", sep=""), paste("Upper ", conf, "% Conf. Bound", sep=""))

# Append results to the object

   eval(parse(text=paste(objname, "$Stratum", i, "$cdf.est <- data.frame(rslt, check.names=F)",
sep="")))

# Add estimates to the matrix for all strata combined

   if (size) {
      if (prop) {
         if (length(W) > 0) {
            rsltall[,2] <- rsltall[,2] + (W[i]/sumW)*rslt[,2]
            rsltall[,3] <- rsltall[,3] + ((W[i]/sumW)*rslt[,3])^2
         } else {
            rsltall[,2] <- rsltall[,2] + (What[i]/sumWhat)*rslt[,2]
            rsltall[,3] <- rsltall[,3] + ((What[i]/sumWhat)*rslt[,3])^2
         }
      } else {
         rsltall[,2] <- rsltall[,2] + rslt[,2]
         rsltall[,3] <- rsltall[,3] + rslt[,3]^2
      }
   } else {
      if (prop) {
         if (length(R) > 0) {
            rsltall[,2] <- rsltall[,2] + (R[i]/sumR)*rslt[,2]
            rsltall[,3] <- rsltall[,3] + ((R[i]/sumR)*rslt[,3])^2
         } else {
            rsltall[,2] <- rsltall[,2] + (Rhat[i]/sumRhat)*rslt[,2]
            rsltall[,3] <- rsltall[,3] + ((Rhat[i]/sumRhat)*rslt[,3])^2
         }
      } else {
```

```
      rsltall[,2] <- rsltall[,2] + rslt[,2]
      rsltall[,3] <- rsltall[,3] + rslt[,3]^2
    }
  }

# End the subsection for individual strata

  }

# Begin the subsection for all strata combined

# Calculate confidence bounds

  rsltall[,3] <- sqrt(rsltall[,3])
  if (size) {
    if (prop) {
      rsltall[,4] <- pmax(rsltall[,2] - mult*rsltall[,3], 0)
      rsltall[,5] <- pmin(rsltall[,2] + mult*rsltall[,3], 1)
    } else {
      rsltall[,4] <- pmax(rsltall[,2] - mult*rsltall[,3], 0)
      if (length(W) > 0)
        rsltall[,5] <- pmin(rsltall[,2] + mult*rsltall[,3], sumW)
      else
        rsltall[,5] <- rsltall[,2] + mult*rsltall[,3]
    }
  } else {
    if (prop) {
      rsltall[,4] <- pmax(rsltall[,2] - mult*rsltall[,3], 0)
      rsltall[,5] <- pmin(rsltall[,2] + mult*rsltall[,3], 1)
    } else {
      rsltall[,4] <- pmax(rsltall[,2] - mult*rsltall[,3], 0)
      if (length(R) > 0)
        rsltall[,5] <- pmin(rsltall[,2] + mult*rsltall[,3], sumR)
      else
        rsltall[,5] <- rsltall[,2] + mult*rsltall[,3]
    }
  }

# Append results to the object

  eval(parse(text=paste(objname, "$AllStrata", "$cdf.est <- data.frame(rsltall, check.names=F)",
sep="")))

# End the subsection for all strata combined
# End subsection for CDF estimates

# Begin subsection for percentile estimates

# Create the matrix for percentile results

   rslt <- array(0, c(npctval, 4))
   dimnames(rslt) <- list(NULL, c("Value", "Percentile Estimate", paste("Lower ", conf, "% Conf.
Bound", sep=""), paste("Upper ", conf, "% Conf. Bound", sep="")))
   rslt[,1] <- pctval

# Begin the subsection for individual strata

   for (i in 1:nstrata) {

# Convert the input percentile values to proportions or to percentage of total, as appropriate

   if (prop) {
      val <- pctval/100
   } else if (size) {
      if (length(W) > 0)
         val <- (pctval/100)*W[i]
      else
         val <- (pctval/100)*What[i]
   } else {
      if (length(R) > 0)
         val <- (pctval/100)*R[i]
      else
         val <- (pctval/100)*Rhat[i]
   }

# Calculate percentile estimates

   if (min(z[stratum == i]) == max(z[stratum == i])) {
      rslt[, 2:4] <- max(z[stratum == i])
   } else {
      eval(parse(text=paste("cdfest <- ", objname, "$Stratum", i, "$cdf.est$CDF", sep="")))
      for (j in 1:npctval) {
         high <- min(nvec[cdfest >= val[j]])
         low <- max(nvec[cdfest <= val[j]])
         if (high == "NA" || low == "NA") {
            rslt[j,2] <- NA
         } else {
            if (high > low)
               ival <- (val[j] - cdfest[low]) / (cdfest[high] - cdfest[low])
```

```
          else
            ival <- 1
          rslt[j,2] <- ival*cdfval[high] + (1-ival)*cdfval[low]
        }
      }

# Calculate confidence bounds of the inverse percentile estimates

      temp <- rslt[,2] != "NA"
      sdest <- numeric(npctval)
      ubound <- numeric(npctval)
      lbound <- numeric(npctval)
      if (size) {
        if (prop) {
          sdest[temp] <- sqrt(cdfvar.size.prop.fcn(z[stratum == i], x[stratum == i], y[stratum == i],
wgt[stratum == i], swgt[stratum == i], rslt[temp,2], val[temp], vartype))
          ubound[temp] <- pmin(val[temp] + mult*sdest[temp], 1)
          lbound[temp] <- pmax(val[temp] - mult*sdest[temp], 0)
        } else {
          sdest[temp] <- sqrt(cdfvar.size.total.fcn(z[stratum == i], x[stratum == i], y[stratum == i],
wgt[stratum == i], swgt[stratum == i], rslt[temp,2], val[temp], W[i], vartype))
          if (length(W) > 0)
            ubound[temp] <- pmin(val[temp] + mult*sdest[temp], W[i])
          else
            ubound[temp] <- val[temp] + mult*sdest[temp]
          lbound[temp] <- pmax(val[temp] - mult*sdest[temp], 0)
        }
      } else {
        if (prop) {
          sdest[temp] <- sqrt(cdfvar.prop.fcn(z[stratum == i], x[stratum == i], y[stratum == i],
wgt[stratum == i], rslt[temp,2], val[temp], vartype))
          ubound[temp] <- pmin(val[temp] + mult*sdest[temp], 1)
          lbound[temp] <- pmax(val[temp] - mult*sdest[temp], 0)
        } else {
          sdest[temp] <- sqrt(cdfvar.total.fcn(z[stratum == i], x[stratum == i], y[stratum == i],
wgt[stratum == i], rslt[temp,2], val[temp], R[i], vartype))
          if (length(R) > 0)
            ubound[temp] <- pmin(val[temp] + mult*sdest[temp], R[i])
          else
            ubound[temp] <- val[temp] + mult*sdest[temp]
          lbound[temp] <- pmax(val[temp] - mult*sdest[temp], 0)
        }
      }

# Calculate confidence bounds of the percentile estimates
```

```
      for (j in 1:npctval) {
         high <- min(nvec[cdfest >= lbound[j]])
         low <- max(nvec[cdfest <= lbound[j]])
         if (high == "NA" || low == "NA") {
            rslt[j,3] <- NA
         } else {
            if (high > low)
               ival <- (lbound[j] - cdfest[low]) / (cdfest[high] - cdfest[low])
            else
               ival <- 1
            rslt[j,3] <- ival*cdfval[high] + (1-ival)*cdfval[low]
         }
         high <- min(nvec[cdfest >= ubound[j]])
         low <- max(nvec[cdfest <= ubound[j]])
         if (high == "NA" || low == "NA") {
            rslt[j,4] <- NA
         } else {
            if (high > low)
               ival <- (ubound[j] - cdfest[low]) / (cdfest[high] - cdfest[low])
            else
               ival <- 1
            rslt[j,4] <- ival*cdfval[high] + (1-ival)*cdfval[low]
         }
      }
   }

# Append results to the object

   eval(parse(text=paste(objname, "$Stratum", i, "$pct.est <- data.frame(rslt, check.names=F)",
sep="")))

# End the subsection for individual strata

   }

# Begin the subsection for all strata combined


# Convert the input percentile values to proportions or to percentage of total, as appropriate

   if (prop) {
      pctval <- pctval/100
   } else if (size) {
      if (length(W) > 0)
```

```
        pctval <- (pctval/100)*sumW
      else
        pctval <- (pctval/100)*sumWhat
    } else {
      if (length(R) > 0)
        pctval <- (pctval/100)*sumR
      else
        pctval <- (pctval/100)*sumRhat
    }

# Calculate percentile estimates

    if (min(z) == max(z)) {
      rslt[, 2:4] <- max(z)
    } else {
      eval(parse(text=paste("cdfest <- ", objname, "$AllStrata", "$cdf.est$CDF", sep="")))
      for (i in 1:npctval) {
        high <- min(nvec[cdfest >= pctval[i]])
        low <- max(nvec[cdfest <= pctval[i]])
        if (high == "NA" || low == "NA") {
          rslt[i,2] <- NA
        } else {
          if (high > low)
            ival <- (pctval[i] - cdfest[low]) / (cdfest[high] - cdfest[low])
          else
            ival <- 1
          rslt[i,2] <- ival*cdfval[high] + (1-ival)*cdfval[low]
        }
      }

# Calculate confidence bounds of the inverse percentile estimates

      temp <- rslt[,2] != "NA"
      sdest <- numeric(npctval)
      ubound <- numeric(npctval)
      lbound <- numeric(npctval)
      if (size) {
        if (prop) {
          sdest[temp] <- sqrt(cdfvar.size.prop.fcn(z, x, y, wgt, swgt, rslt[temp,2], pctval[temp],
vartype))
          ubound[temp] <- pmin(pctval[temp] + mult*sdest[temp], 1)
          lbound[temp] <- pmax(pctval[temp] - mult*sdest[temp], 0)
        } else {
          if (length(W) > 0) {
            sdest[temp] <- sqrt(cdfvar.size.total.fcn(z, x, y, wgt, swgt, rslt[temp,2], pctval[temp],
sumW, vartype))
            ubound[temp] <- pmin(pctval[temp] + mult*sdest[temp], sumW)
          } else {
            sdest[temp] <- sqrt(cdfvar.size.total.fcn(z, x, y, wgt, swgt, rslt[temp,2], pctval[temp],
NULL, vartype))
            ubound[temp] <- pctval[temp] + mult*sdest[temp]
          }
          lbound[temp] <- pmax(pctval[temp] - mult*sdest[temp], 0)
        }
      } else {
        if (prop) {
          sdest[temp] <- sqrt(cdfvar.prop.fcn(z, x, y, wgt, rslt[temp,2], pctval[temp], vartype))
          ubound[temp] <- pmin(pctval[temp] + mult*sdest[temp], 1)
          lbound[temp] <- pmax(pctval[temp] - mult*sdest[temp], 0)
        } else {
          if (length(R) > 0) {
            sdest[temp] <- sqrt(cdfvar.total.fcn(z, x, y, wgt, rslt[temp,2], pctval[temp], sumR, vartype))
            ubound[temp] <- pmin(pctval[temp] + mult*sdest[temp], sumR)
          } else {
            sdest[temp] <- sqrt(cdfvar.total.fcn(z, x, y, wgt, rslt[temp,2], pctval[temp], NULL,
vartype))
            ubound[temp] <- pctval[temp] + mult*sdest[temp]
          }
          lbound[temp] <- pmax(pctval[temp] - mult*sdest[temp], 0)
        }
      }

# Calculate confidence bounds of the percentile estimates

      for (i in 1:npctval) {
        high <- min(nvec[cdfest >= lbound[i]])
        low <- max(nvec[cdfest <= lbound[i]])
        if (high == "NA" || low == "NA") {
          rslt[i,3] <- NA
        } else {
          if (high > low)
            ival <- (lbound[i] - cdfest[low]) / (cdfest[high] - cdfest[low])
          else
            ival <- 1
          rslt[i,3] <- ival*cdfval[high] + (1-ival)*cdfval[low]
        }
        high <- min(nvec[cdfest >= ubound[i]])
        low <- max(nvec[cdfest <= ubound[i]])
        if (high == "NA" || low == "NA") {
         rslt[i,4] <- NA
      } else {
         if (high > low)
            ival <- (ubound[i] - cdfest[low]) / (cdfest[high] - cdfest[low])
         else
            ival <- 1
         rslt[i,4] <- ival*cdfval[high] + (1-ival)*cdfval[low]
      }
   }
  }

# Append results to the object

  eval(parse(text=paste(objname, "$AllStrata", "$pct.est <- data.frame(rslt, check.names=F)",
sep="")))

# End the subsection for all strata combined

# End the subsection for percentile estimates

# End the section for stratified data

  } else {

# Begin the section for unstratified data

# Begin subsection for CDF estimates

# Create the object for the estimates

  eval(parse(text=paste(objname, " <- vector('list', 2)", sep="")))
  temp <- list(c("cdf.est", "pct.est"))
  eval(parse(text=paste("names(", objname, ") <- ", temp, sep="")))

# Calculate the CDF estimates, standard deviation estimates, and confidence bounds

  if (size) {
    if (prop) {
      cdfest <- cdf.size.prop.fcn(z, wgt, swgt, cdfval)
      sdest <- sqrt(cdfvar.size.prop.fcn(z, x, y, wgt, swgt, cdfval, cdfest, vartype))
      ubound <- pmin(cdfest + mult*sdest, 1)
      lbound <- pmax(cdfest - mult*sdest, 0)
    } else {
      cdfest <- cdf.size.total.fcn(z, wgt, swgt, cdfval, W)
      sdest <- sqrt(cdfvar.size.total.fcn(z, x, y, wgt, swgt, cdfval, cdfest, W, vartype))
```

```
        if (length(W) > 0)
          ubound <- pmin(cdfest + mult*sdest, W)
        else
          ubound <- cdfest + mult*sdest
        lbound <- pmax(cdfest - mult*sdest, 0)
      }
    } else {
      if (prop) {
        cdfest <- cdf.prop.fcn(z, wgt, cdfval)
        sdest <- sqrt(cdfvar.prop.fcn(z, x, y, wgt, cdfval, cdfest, vartype))
        ubound <- pmin(cdfest + mult*sdest, 1)
        lbound <- pmax(cdfest - mult*sdest, 0)
      } else {
        cdfest <- cdf.total.fcn(z, wgt, cdfval, R)
        sdest <- sqrt(cdfvar.total.fcn(z, x, y, wgt, cdfval, cdfest, R, vartype))
        if (length(R) > 0)
          ubound <- pmin(cdfest + mult*sdest, R)
        else
          ubound <- cdfest + mult*sdest
        lbound <- pmax(cdfest - mult*sdest, 0)
      }
    }
    rslt <- cbind(cdfval, cdfest, sdest, lbound, ubound)
    dimnames(rslt)[[2]] <- c("Value", "CDF Estimate", "Standard Deviation", paste("Lower ", conf,
"% Conf. Bound", sep=""), paste("Upper ", conf, "% Conf. Bound", sep=""))

# Append results to the object

    eval(parse(text=paste(objname, "$cdf.est <- data.frame(rslt, check.names=F)", sep="")))

# End subsection for CDF estimates

# Begin subsection for percentile estimates

# Create the matrix for percentile results

    rslt <- array(0, c(npctval, 4))
    dimnames(rslt) <- list(NULL, c("Value", "Percentile Estimate", paste("Lower ", conf, "% Conf.
Bound", sep=""), paste("Upper ", conf, "% Conf. Bound", sep="")))
    rslt[,1] <- pctval

# Convert the input percentile values to proportions or to percentage of total, as appropriate

    if (prop) {
      pctval <- pctval/100
```

```
   } else if (size) {
      if (length(W) > 0)
         pctval <- (pctval/100)*W
      else
         pctval <- (pctval/100)*What
   } else {
      if (length(R) > 0)
         pctval <- (pctval/100)*R
      else
         pctval <- (pctval/100)*Rhat
   }

# Calculate percentile estimates

   if (min(z) == max(z)) {
      rslt[, 2:4] <- max(z)
   } else {
      for (i in 1:npctval) {
         high <- min(nvec[cdfest >= pctval[i]])
         low <- max(nvec[cdfest <= pctval[i]])
         if (high == "NA" || low == "NA") {
            rslt[i,2] <- NA
         } else {
            if (high > low)
               ival <- (pctval[i] - cdfest[low]) / (cdfest[high] - cdfest[low])
            else
               ival <- 1
            rslt[i,2] <- ival*cdfval[high] + (1-ival)*cdfval[low]
         }
      }

# Calculate confidence bounds of the inverse percentile estimates

      temp <- rslt[,2] != "NA"
      sdest <- numeric(npctval)
      ubound <- numeric(npctval)
      lbound <- numeric(npctval)
      if (size) {
         if (prop) {
            sdest[temp] <- sqrt(cdfvar.size.prop.fcn(z, x, y, wgt, swgt, rslt[temp,2], pctval[temp],
vartype))
            ubound[temp] <- pmin(pctval[temp] + mult*sdest[temp], 1)
            lbound[temp] <- pmax(pctval[temp] - mult*sdest[temp], 0)
         } else {
          sdest[temp] <- sqrt(cdfvar.size.total.fcn(z, x, y, wgt, swgt, rslt[temp,2], pctval[temp], W,
vartype))
          if (length(W) > 0)
            ubound[temp] <- pmin(pctval[temp] + mult*sdest[temp], W)
          else
            ubound[temp] <- pctval[temp] + mult*sdest[temp]
          lbound[temp] <- pmax(pctval[temp] - mult*sdest[temp], 0)
        }
      } else {
        if (prop) {
          sdest[temp] <- sqrt(cdfvar.prop.fcn(z, x, y, wgt, rslt[temp,2], pctval[temp], vartype))
          ubound[temp] <- pmin(pctval[temp] + mult*sdest[temp], 1)
          lbound[temp] <- pmax(pctval[temp] - mult*sdest[temp], 0)
        } else {
          sdest[temp] <- sqrt(cdfvar.total.fcn(z, x, y, wgt, rslt[temp,2], pctval[temp], R, vartype))
          if (length(R) > 0)
            ubound[temp] <- pmin(pctval[temp] + mult*sdest[temp], R)
          else
            ubound[temp] <- pctval[temp] + mult*sdest[temp]
          lbound[temp] <- pmax(pctval[temp] - mult*sdest[temp], 0)
        }
      }

# Calculate confidence bounds of the percentile estimates

      for (i in 1:npctval) {
        high <- min(nvec[cdfest >= lbound[i]])
        low <- max(nvec[cdfest <= lbound[i]])
        if (high == "NA" || low == "NA") {
          rslt[i,3] <- NA
        } else {
          if (high > low)
            ival <- (lbound[i] - cdfest[low]) / (cdfest[high] - cdfest[low])
          else
            ival <- 1
          rslt[i,3] <- ival*cdfval[high] + (1-ival)*cdfval[low]
        }
        high <- min(nvec[cdfest >= ubound[i]])
        low <- max(nvec[cdfest <= ubound[i]])
        if (high == "NA" || low == "NA") {
          rslt[i,4] <- NA
        } else {
          if (high > low)
            ival <- (ubound[i] - cdfest[low]) / (cdfest[high] - cdfest[low])
          else
```

```
            ival <- 1
          rslt[i,4] <- ival*cdfval[high] + (1-ival)*cdfval[low]
        }
      }
   }

# Append results to the object

   eval(parse(text=paste(objname, "$pct.est <- data.frame(rslt, check.names=F)", sep="")))

# End subsection for percentile estimates

# End section for unstratified data

   }

# Return the object

   eval(parse(text=objname))
}
```

# Program:
cdf.prop.fcn

```
# Date:    November 28, 2000
# Description:
#   This function calculates an estimate of the cumulative distribution function
#   (CDF) for the proportion of a discrete or an extensive resource.  The set of
#   values at which the CDF is estimated is supplied to the function.  The
#   Horvitz-Thompson ratio estimator, i.e., the ratio of two Horvitz-Thompson
#   estimators, is used to calculate the estimate.  The numerator of the ratio
#   estimates the total of the resource equal to or less than a specified value.
#   The denominator of the ratio estimates the size of the resource.  For a
#   discrete resource size is the number of units in the resource.  For an
#   extensive resource size is the extent (measure) of the resource, i.e.,
#   length, area, or volume.
#   Input:
#      z = the response value for each site
#      wgt = the weight (inverse of the sample inclusion probability) for each
#           site
#      val = the set of values at which the CDF is estimated
#   Output is the CDF estimate

cdf.prop.fcn <- function(z, wgt, val) {

# Calculate the cdf
```

```
  m <- length(val)
  cdf <- numeric(m)
  for(i in 1:m) {
    cdf[i] <- sum(ifelse(z <= val[i], wgt, 0))
  }
  cdf <- cdf/sum(wgt)

# Return the estimate

  cdf
}
```

```
# Program:
cdfvar.prop.fcn
# Date:    November 28, 2000
# Description:
#   This function calculates variance estimates of the estimated cumulative
#   distribution function (CDF) for the proportion of a discrete or a continuous
#   resource.  The set of values at which the CDF is estimated is supplied to
#   the function.  Either the simple random sampling (SRS) variance estimator or
#   the neighborhood variance estimator is calculated, which is subject to user
#   control.  The SRS variance estimator uses the independent random sample
#   approximation to calculate joint inclusion probabilities.
#   Input:
#      z = the response values
#      x = x-coordinate of each site
#      y = y-coordinate of each site
#      wgt = the weight (inverse of the sample inclusion probability) for each
#         response value
#      val = the set of values at which the CDF is estimated
#      cdfest = the CDF estimate
#      vartype = indicator for variance estimator ("Local"=neighborhood estimator
#         and "SRS"=SRS estimator, the default is "Local")
#   Output is the variance estimate
#   Other Functions Required:
#      localmean.fcn - calculate the neighborhood variance estimator

cdfvar.prop.fcn <- function(z, x, y, wgt, val, cdfest, vartype) {

# Calculate other required input values

   n <- length(z)
   prb <- 1/wgt
   tw2 <- (sum(wgt))^2
   m <- length(val)

# Calculate the weighted residuals matrix

   im <- ifelse(matrix(rep(z, m), nrow = n) <= matrix(rep(val, n), nrow = n, byrow = T), 1, 0)
   rm <- (im - matrix(rep(cdfest, n), nrow = n, byrow = T)) * matrix(rep(wgt, m), nrow = n)

# Calculate the variance estimate

   if (vartype == "Local" && n < 4) {
      warning("\nThere are less than four response values, the simple random sampling variance
estimator will be used\n\n")
      vartype <- "SRS"
   }
    if (vartype == "Local") {
      varest <- apply(rm, 2, localmean.fcn, x=x, y=y, prb=prb) / tw2
   } else {
      varest <- n * apply(rm, 2, var) / tw2
   }

# Return the variance estimate

   varest

}
```

```
# Program:
```
cdf.total.fcn
```
# Date:    December 4, 2000
# Description:
#   This function calculates an estimate of the cumulative distribution function
#   (CDF) for the total of a discrete or an extensive resource.  The set of
#   values at which the CDF is estimated is supplied to the function.  If the
#   known extent of the resource is provided, the classic ratio estimator is
#   used to calculate the estimate. That estimator is the product of the known
#   extent of the resource and the Horvitz-Thompson ratio estimator, where the
#   latter is the ratio of two Horvitz-Thompson estimators.  The numerator of
#   the ratio estimates the total of the resource equal to or less than a
#   specified value.  The denominator of the ratio estimates the extent of the
#   resource.  If the known extent of the resource is not provided, the Horvitz-
#   Thompson estimator of the total of the resource equal to or less than a
#   specified value is used to calculate the estimate. For a discrete resource,
#   extent is the number of units in the resource.  For an extensive resource,
#   extent is the measure of the resource, i.e., length, area, or volume.
#   Input:
#     z = the response value for each site
#     wgt = the weight (inverse of the sample inclusion probability) for each
#         site
#     val = the set of values at which the CDF is estimated
#     R = the known extent of the resource
#   Output is the CDF estimate

cdf.total.fcn <- function(z, wgt, val, R) {

# Calculate the cdf

  m <- length(val)
  cdf <- numeric(m)
  for(i in 1:m) {
    cdf[i] <- sum(ifelse(z <= val[i], wgt, 0))
  }
  if (length(R) > 0) cdf <- R*(cdf/sum(wgt))

# Return the estimate

  cdf
}
```

```
# Program:
```
cdfvar.total.fcn
```
# Date:    December 4, 2000
# Description:
#   This function calculates variance estimates of the estimated cumulative
#   distribution function (CDF) for the total of a discrete or a continuous
#   resource.  The set of values at which the CDF is estimated is supplied to
#   the function.  Either the simple random sampling (SRS) variance estimator or
#   the neighborhood variance estimator is calculated, which is subject to user
#   control.  The SRS variance estimator uses the independent random sample
#   approximation to calculate joint inclusion probabilities.
#   Input:
#     z = the response values
#     x = x-coordinate of each site
#     y = y-coordinate of each site
#     wgt = the weight (inverse of the sample inclusion probability) for each
#         response value
#     val = the set of values at which the CDF is estimated
#     cdfest = the CDF estimate
#     R = the known extent of the resource
#     vartype = indicator for variance estimator ("Local"=neighborhood estimator
#       and "SRS"=SRS estimator, the default is "Local")
#   Output is the variance estimate
#   Other Functions Required:
#     localmean.fcn - calculate the neighborhood variance estimator

cdfvar.total.fcn <- function(z, x, y, wgt, val, cdfest, R, vartype) {

# Calculate other required input values

  n <- length(z)
  prb <- 1/wgt
  m <- length(val)
  if (length(R) > 0) {
    tw2 <- (sum(wgt))^2
    cdfest <- cdfest/R
  }

# Calculate the weighted residuals matrix

  im <- ifelse(matrix(rep(z, m), nrow = n) <= matrix(rep(val, n), nrow = n, byrow = T), 1, 0)
  if (length(R) > 0)
    rm <- (im - matrix(rep(cdfest, n), nrow = n, byrow = T)) * matrix(rep(wgt, m), nrow = n)
  else
    rm <- im * matrix(rep(wgt, m), nrow = n)
# Calculate the variance estimate

   if (vartype == "Local" && n < 4) {
      warning("\nThere are less than four response values, the simple random sampling variance
estimator will be used\n\n")
      vartype <- "SRS"
   }
   if (length(R) > 0) {
      if (vartype == "Local")
         varest <- (R^2) * (apply(rm, 2, localmean.fcn, x=x, y=y, prb=prb) / tw2)
      else
         varest <- (R^2) * (n*apply(rm, 2, var) / tw2)
   } else {
      if (vartype == "Local")
         varest <- apply(rm, 2, localmean.fcn, x=x, y=y, prb=prb)
      else
         varest <- n*apply(rm, 2, var)
   }
# Return the variance estimate

   varest

}
```

```
# Program:
```
cdf.size.prop.fcn
```
# Date:    November 28, 2000
# Description:
#   This function calculates an estimate of the size-weighted cumulative
#   distribution function (CDF) for the proportion of a discrete resource. The
#   set of values at which the CDF is estimated is supplied to the function. The
#   Horvitz-Thompson ratio estimator, i.e., the ratio of two Horvitz-Thompson
#   estimators, is used to calculate the estimate.  The numerator of the ratio
#   estimates the size-weighted total of the resource equal to or less than a
#   specified value.  The denominator of the ratio estimates the sum of the
#   size-weights for the resource.
#   Input:
#      z = the response value for each site
#      wgt = the weight (inverse of the sample inclusion probability) for each
#          site
#      swgt = the size-weight for each site
#      val = the set of values at which the CDF is estimated
#   Output is the CDF estimate

cdf.size.prop.fcn <- function(z, wgt, swgt, val) {

# Calculate the cdf

  wgt <- wgt*swgt
  m <- length(val)
  cdf <- numeric(m)
  for(i in 1:m) {
    cdf[i] <- sum(ifelse(z <= val[i], wgt, 0))
  }
  cdf <- cdf/sum(wgt)

# Return the estimate

  cdf
}
```

```
# Program:
```

cdfvar.size.prop.fcn

```
# Date:    November 28, 2000
# Description:
#   This function calculates variance estimates of the estimated size-weighted
#   cumulative distribution function (CDF) for the proportion of a discrete
#   resource.  The set of values at which the CDF is estimated is supplied to
#   the function.  Either the simple random sampling (SRS) variance estimator or
#   the neighborhood variance estimator is calculated, which is subject to user
#   control.  The SRS variance estimator uses the independent random sample
#   approximation to calculate joint inclusion probabilities.
#   Input:
#      z = the response values
#      x = x-coordinate of each site
#      y = y-coordinate of each site
#      wgt = the weight (inverse of the sample inclusion probability) for each
#          response value
#      swgt = the size-weight for each site
#      val = the set of values at which the CDF is estimated
#      cdfest = the CDF estimate
#      vartype = indicator for variance estimator ("Local"=neighborhood estimator
#        and "SRS"=SRS estimator, the default is "Local")
#   Output is the variance estimate
#   Other Functions Required:
#      localmean.fcn - calculate the neighborhood variance estimator

cdfvar.size.prop.fcn <- function(z, x, y, wgt, swgt, val, cdfest, vartype) {

# Calculate other required input values

   n <- length(z)
   wgt <- wgt*swgt
   prb <- 1/wgt
   tw2 <- (sum(wgt))^2
   m <- length(val)

# Calculate the weighted residuals matrix

   im <- ifelse(matrix(rep(z, m), nrow = n) <= matrix(rep(val, n), nrow = n, byrow = T), 1, 0)
   rm <- (im - matrix(rep(cdfest, n), nrow = n, byrow = T)) * matrix(rep(wgt, m), nrow = n)

# Calculate the variance estimate

   if (vartype == "Local" && n < 4) {
      warning("\nThere are less than four response values, the simple random sampling variance
estimator will be used\n\n")
      vartype <- "SRS"
   }
   if (vartype == "Local") {
      varest <- apply(rm, 2, localmean.fcn, x=x, y=y, prb=prb) / tw2
   } else {
      varest <- n * apply(rm, 2, var) / tw2
   }

# Return the variance estimate

   varest

}
```

```
# Program:
```

cdf.size.total.fcn

```
# Date:    December 4, 2000
# Description:
#   This function calculates an estimate of the size-weighted cumulative
#   distribution function (CDF) for the total of a discrete resource.  The set
#   of values at which the CDF is estimated is supplied to the function.  If the
#   known sum of the size-weights of the resource is provided, the classic ratio
#   estimator is used to calculate the estimate. That estimator is the product
#   of the known sum of the size-weights of the resource and the Horvitz-
#   Thompson ratio estimator, where the latter is the ratio of two Horvitz-
#   Thompson estimators.  The numerator of the ratio estimates the size-weighted
#   total of the resource equal to or less than a specified value.  The
#   denominator of the ratio estimates the sum of the size-weights of the
#   resource.  If the known sum of the size-weights of the resource is not
#   provided, the Horvitz-Thompson estimator of the size-weighted total of
#   the resource equal to or less than a specified value is used to calculate
#   the estimate.
#   Input:
#      z = the response value for each site
#      wgt = the weight (inverse of the sample inclusion probability) for each
#          site
#      swgt = the size-weight for each site
#      val = the set of values at which the CDF is estimated
#      W = the known sum of the size-weights of the resource
#   Output is the CDF estimate

cdf.size.total.fcn <- function(z, wgt, swgt, val, W) {

# Calculate the cdf

  wgt <- wgt*swgt
  m <- length(val)
  cdf <- numeric(m)
  for(i in 1:m) {
    cdf[i] <- sum(ifelse(z <= val[i], wgt, 0))
  }
  if (length(W) > 0) cdf <- W*(cdf/sum(wgt))

# Return the estimate

  cdf
}
```

```
# Program:
```
cdfvar.size.total.fcn
```
# Date:    December 4, 2000
# Description:
#   This function calculates variance estimates of the estimated size-weighted
#   cumulative distribution function (CDF) for the total of a discrete
#   resource.  The set of values at which the CDF is estimated is supplied to
#   the function.  Either the simple random sampling (SRS) variance estimator or
#   the neighborhood variance estimator is calculated, which is subject to user
#   control.  The SRS variance estimator uses the independent random sample
#   approximation to calculate joint inclusion probabilities.
#   Input:
#     z = the response values
#     x = x-coordinate of each site
#     y = y-coordinate of each site
#     wgt = the weight (inverse of the sample inclusion probability) for each
#         response value
#     swgt = the size-weight for each site
#     val = the set of values at which the CDF is estimated
#     cdfest = the CDF estimate
#     W = the known sum of the size-weights of the resource
#     vartype = indicator for variance estimator ("Local"=neighborhood estimator
#       and "SRS"=SRS estimator, the default is "Local")
#   Output is the variance estimate
#   Other Functions Required:
#     localmean.fcn - calculate the neighborhood variance estimator

cdfvar.size.total.fcn <- function(z, x, y, wgt, swgt, val, cdfest, W, vartype) {

# Calculate other required input values

  n <- length(z)
  wgt <- wgt*swgt
  prb <- 1/wgt
  m <- length(val)
  if (length(W) > 0) {
    tw2 <- (sum(wgt))^2
    cdfest <- cdfest/W
  }

# Calculate the weighted residuals matrix

  im <- ifelse(matrix(rep(z, m), nrow = n) <= matrix(rep(val, n), nrow = n, byrow = T), 1, 0)
  if (length(W) > 0)
    rm <- (im - matrix(rep(cdfest, n), nrow = n, byrow = T)) * matrix(rep(wgt, m), nrow = n)
  else
    rm <- im * matrix(rep(wgt, m), nrow = n)

# Calculate the variance estimate

  if (vartype == "Local" && n < 4) {
    warning("\nThere are less than four response values, the simple random sampling variance
estimator will be used\n\n")
    vartype <- "SRS"
  }
  if (length(W) > 0) {
    if (vartype == "Local")
      varest <- (W^2) * (apply(rm, 2, localmean.fcn, x=x, y=y, prb=prb) / tw2)
    else
      varest <- (W^2) * (n*apply(rm, 2, var) / tw2)
  } else {
    if (vartype == "Local")
      varest <- apply(rm, 2, localmean.fcn, x=x, y=y, prb=prb)
    else
      varest <- n*apply(rm, 2, var)
  }

# Return the variance estimate

  varest

}
```

```
# Program:
```
cdf.test.fcn
```
# Date:    April 20, 2001
# Description:
#   This function calculates the Wald, mean eigenvalue-corrected, and
#   Satterthwaite-corrected statistics for testing differences between two
#   cumulative distribution functions (CDFs).  The user supplies the set of
#   upper bounds for defining the classes for the CDFs.  The Horvitz-Thompson
#   ratio estimator, i.e., the ratio of two Horvitz-Thompson estimators, is used
#   to calculate estimates of the class proportions for the CDFs.  Variance
#   estimates for the test statistics are calculated using either the neighborhood
#   variance estimator or the simple random sampling variance estimator.  The
#   choice of variance estimator is subject to user control.  The simple random
#   sampling variance estimator uses the independent random sample approximation
#   to calculate joint inclusion probabilities.  The function checks for
#   compatibility of input values and removes missing values.
#   Input:
#     sample1 = the sample from the first population
#     sample2 = the sample from the second population
#       where sample1 and sample2 are lists containing the following items:
#         z = the response value for each site
#         wgt = the weight (inverse of the sample inclusion probability) for
#           each site
#         x = x-coordinate of each site (may be NULL)
#         y = y-coordinate of each site (may be NULL)
#     bounds = upper bounds for calculating the classes for the cdf
#     vartype = indicator for variance estimator ("Local"=neighborhood estimator
#       and "SRS"=SRS estimator, the default is "Local")
#   Output:
#     An object in data frame format containing the test statistics, degrees of
        freedom, and p values for the Wald, mean eigenvalue, and Satterthwaite
#       test procedures
#   Other Functions Required:
#     wnas.fcn - remove missing values
#     localmean.cov.fcn - calculate the variance-covariance matrix using the
#       neighborhood estimator
#     dist2full.fcn - convert a vector of distance measures to a matrix
#   Example Function Call:
#     cdf.test.fcn(list(z=mydata1$response, wgt=mydata1$weight, x=mydata1$x,
#       y=mydata1$y), list(z=mydata2$response, wgt=mydata2$weight,
#       x=mydata2$x, y=mydata2$y), seq(10, 100, length=10))

cdf.test.fcn <- function(sample1, sample2, bounds, vartype="Local") {

# Remove missing values
```

```
   if(vartype == "Local") {
      temp <- wnas.fcn(list(z=sample1$z, wgt=sample1$wgt, x=sample1$x, y=sample1$y))
      sample1$z <- temp$z
      sample1$wgt <- temp$wgt
      sample1$x <- temp$x
      sample1$y <- temp$y
      temp <- wnas.fcn(list(z=sample2$z, wgt=sample2$wgt, x=sample2$x, y=sample2$y))
      sample2$z <- temp$z
      sample2$wgt <- temp$wgt
      sample2$x <- temp$x
      sample2$y <- temp$y
   } else {
      temp <- wnas.fcn(list(z=sample1$z, wgt=sample1$wgt))
      sample1$z <- temp$z
      sample1$wgt <- temp$wgt
      temp <- wnas.fcn(list(z=sample2$z, wgt=sample2$wgt))
      sample2$z <- temp$z
      sample2$wgt <- temp$wgt
   }

# Check for compatibility of input values

   if(length(sample1$z) != length(sample1$wgt))
      stop("\n\nNumber of response values must equal the number of weights for sample 1.")
   if(length(sample2$z) != length(sample2$wgt))
      stop("\n\nNumber of response values must equal the number of weights for sample 2.")
   if(min(sample1$wgt) <= 0)
      stop("\n\nWeights must be positive for sample 1.")
   if(min(sample2$wgt) <= 0)
      stop("\n\nWeights must be positive for sample 2.")
   if(vartype == "Local") {
      if (length(sample1$x) == 0 || length(sample1$y) == 0)
         stop("\n\nx-coordinate and y-coordinate values for sample 1 are required for the neighborhood
variance estimator.")
      else if (length(sample1$z) != length(sample1$x) || length(sample1$z) != length(sample1$y))
         stop("\n\nNumber of response values must equal the number of x-coordinate and y-coordinate
values for sample 1 for the neighborhood variance estimator.")
   }
   if(vartype == "Local") {
      if (length(sample2$x) == 0 || length(sample2$y) == 0)
         stop("\n\nx-coordinate and y-coordinate values for sample 2 are required for the neighborhood
variance estimator.")
      else if (length(sample2$z) != length(sample2$x) || length(sample2$z) != length(sample2$y))
```

```
         stop("\n\nNumber of response values must equal the number of x-coordinate and y-coordinate
values for sample 2 for the neighborhood variance estimator.")
   }

# Create the matrix for results

   rslt <- array(0, c(3, 3))
   dimnames(rslt) <- list(c("Wald  ", "Mean Eigenvalue  ", "Satterthwaite  "), c("Test Statistic",
"Degrees of Freedom", "p Value"))

# Assign the number of bins

   n.bin <- length(bounds)
   m <- n.bin - 1

# Assign length of the response vectors

   n1 <- length(sample1$z)
   n2 <- length(sample2$z)

# Calculate estimates for the first sample

   est1 <- cdfvar.test.fcn(sample1, bounds, vartype)
   sam1.phat <- as.array(est1$phat)
   sam1.phatvar <- as.matrix(est1$varest)

# Calculate estimates for the second sample

   est2 <- cdfvar.test.fcn(sample2, bounds, vartype)
   sam2.phat <- as.array(est2$phat)
   sam2.phatvar <- as.matrix(est2$varest)

# Calculate the Wald chi square statistic

   difr <- sam1.phat[-n.bin] - sam2.phat[-n.bin]
   tqr <- qr(sam1.phatvar[-n.bin,-n.bin] + sam2.phatvar[-n.bin,-n.bin])
   if (tqr$rank < m) {
      rslt[1, 1] <- NA
      rslt[1, 2] <- NA
      rslt[1, 3] <- NA
   } else {
      rslt[1, 1] <- difr %*% solve(tqr) %*% difr
      rslt[1, 2] <- m
      rslt[1, 3] <- 1 - pchisq(rslt[1, 1], rslt[1, 2])
   }
```

```
# Calculate the mean eigenvalue-corrected chi square statistic

  phatmean <- ((n1 * sam1.phat) + (n2 * sam2.phat)) / (n1 + n2)
  difr1 <- sam1.phat[-n.bin] - phatmean[-n.bin]
  difr2 <- sam2.phat[-n.bin] - phatmean[-n.bin]
  tqr <- qr(diag(phatmean[-n.bin]) - (phatmean[-n.bin] %o% phatmean[-n.bin]))
  if (tqr$rank < m) {
    rslt[2, 1] <- NA
    rslt[2, 2] <- NA
    rslt[2, 3] <- NA
  } else {
    rmatinv <- solve(tqr)
    chisqtmp <- (n1 * difr1 %*% rmatinv %*% difr1) + (n2 * difr2 %*% rmatinv %*% difr2)
    sam1.amat <- n1 * (rmatinv %*% sam1.phatvar[-n.bin,-n.bin])
    sam2.amat <- n2 * (rmatinv %*% sam2.phatvar[-n.bin,-n.bin])
    eigmat <- eigen(((n2*sam1.amat) + (n1*sam2.amat)) / (n1 + n2))
    eigmean <- mean(eigmat$values)
    rslt[2, 1] <- chisqtmp / eigmean
    rslt[2, 2] <- m
    rslt[2, 3] <- 1 - pchisq(rslt[2, 1], rslt[2, 2])
  }

# Calculate the Satterthwaite-corrected chi square statistic

  if (tqr$rank < m) {
    rslt[3, 1] <- NA
    rslt[3, 2] <- NA
    rslt[3, 3] <- NA
  } else {
    cvsquare <- var(eigmat$values) / (eigmean ^ 2)
    rslt[3, 1] <- rslt[2, 1] / (1 + cvsquare)
    rslt[3, 2] <- m / (1 + cvsquare)
    rslt[3, 3] <- 1 - pchisq(rslt[3, 1], rslt[3, 2])
  }

# Return the object

  data.frame(rslt, check.names=F)
}
```

```
# Program:
```

cdfvar.test.fcn

```
# Date:    December 15, 2000
# Description:
#   This function calculates unweighted and weighted estimates of the population
#   proportions in a set of bins and estimates of the variance-covariance matrix
#   of the weighted proportions using either the neighborhood variance estimator
#   or the simple random sampling variance estimator.  The choice of variance
#   estimator is subject to user control.  The simple random sampling variance
#   estimator uses the independent random sample approximation to calculate
#   joint inclusion probabilities.
#   Input:
#      sampl = sample values, which is a list containing the following items:
#         z = the response value for each site
#         x = x-coordinate of each site (may be NULL)
#         y = y-coordinate of each site (may be NULL)
#         wgt = the weight (inverse of the sample inclusion probability) for
#            each site
#      bounds = bounds for calculating the bins
#      vartype = indicator for variance estimator ("Local"=neighborhood estimator
#         and "SRS"=SRS estimator, the default is "Local")
#   Output is a list containing the following items:
#      phat = weighted estimator of the bin proportions
#      varest = estimator of the variance-covariance matrix
#   Other Functions Required:
#      localmean.cov.fcn - calculate the variance-covariance matrix using the
#         neighborhood estimator

cdfvar.test.fcn <- function(sampl, bounds, vartype) {

# Assign input values

   z <- sampl$z
   x <- sampl$x
   y <- sampl$y
   wgt <- sampl$wgt

# Calculate other required input values

   n <- length(z)
   tw <- sum(wgt)
   prb <- 1/wgt
   m <- length(bounds)

# Calculate the indicator value matrix
```

```
   zm <- matrix(rep(z, m), nrow = n)
   ubound <- matrix(rep(bounds, n), nrow = n, byrow = T)
   lbound <- matrix(rep(c(-1e10, bounds[-m]), n), nrow = n, byrow = T)
   im <- ifelse(zm <= ubound, 1, 0) - ifelse(zm <= lbound, 1, 0)

# Calculate the weight matrix

   wm <- matrix(rep(wgt, m), nrow = n)

# Calculate the class proportion estimate

   phat <- apply(im*wm, 2, sum)/tw

# Calculate the weighted residual matrices

   rm <- (im - matrix(rep(phat, n), nrow = n, byrow = T)) * wm

# Calculate the variance-covariance estimate

   if (vartype == "Local" && n < 4) {
      warning("\nThere are less than four response values, the simple random sampling
variance-covariance estimator will be used\n\n")
      vartype <- "SRS"
   }
   if (vartype == "Local") {
      varest <- localmean.cov.fcn(z=rm, x=x, y=y, prb=prb) / (tw^2)
   } else {
      varest <- n * var(rm) / (tw^2)
   }

# Return the estimates

   list(phat=phat, varest=varest)

}
```
# Program:
localmean.fcn
```
# Date:    April 20, 2001
# Description:
#   This function calculates the variance using the neighborhood estimator.
#   Input:
#      z = response values for the sample points
#      x = x-coordinates of the sample points
#      y = y-coordinates of the sample points
#      prb = inclusion probabilities for the sample points
#      nbh = number of neighboring points to use in the calculations
#   Output:
#      lmvar = neighborhood estimator of the variance

localmean.fcn <- function(z, x, y, prb, nbh=4) {

   n <- length(z)

# Calculate indices of nearest neighbors

   idx <- apply(dist2full.fcn(dist(cbind(x, y))), 2, order)[1:nbh,  ]

# Make neighbors symmetric

   jdx <- rep(1:n, rep(nbh, n))
   kdx <- unique(c((jdx - 1) * n + idx, (idx - 1) * n + jdx)) - 1
   ij <- cbind((kdx) %/% n + 1, (kdx) %% n + 1)
   ij <- ij[order(ij[, 1]),  ]

# Apply linear taper to the  inverse probability weights

   gct <- tabulate(ij[, 1])
   gwt <- numeric(0)
   for (i in 1:n)
      gwt <- c(gwt, 1 - (1:gct[i] - 1)/(gct[i]))
   gwt <- gwt/prb[ij[, 2]]

# Normalize to make true average

   smwt <- sapply(split(gwt, ij[, 1]), sum)
   gwt <- gwt/smwt[ij[, 1]]
   smwt <- sapply(split(gwt, ij[, 2]), sum)

# Make weights doubly stochastic

   hij <- matrix(0, n, n)
   hij[ij] <- 0.5
   a22 <- ginverse(diag(gct/2) - hij %*% diag(2/gct) %*% hij)
   a21 <-  - diag(2/gct) %*% hij %*% a22
   lm <- a21 %*% (1 - smwt)
   gm <- a22 %*% (1 - smwt)
   gwt <- (lm[ij[, 1]] + gm[ij[, 2]])/2 + gwt

# Calculate neighborhoods
```

```
  zb <- sapply(split(z[ij[, 2]] * gwt, ij[, 1]), sum)

# Calculate the variance estimate

  lmvar <- sum(gwt * (z[ij[, 2]] - zb[ij[, 1]])^2)

# Return the variance estimate

  lmvar
}
```

# Program:
localmean.cov.fcn

```
# Date:    April 20, 2001
# Description:
#   This function calculates the variance-covariance matrix using the neighborhood
#   estimator.
#   Input:
#      zmat = matrix of response values for the sample points
#      x = x-coordinates of the sample points
#      y = y-coordinates of the sample points
#      prb = inclusion probabilities for the sample points
#      nbh = number of neighboring points to use in the calculations
#   Output:
#      lmvar = neighborhood estimator of the variance

localmean.cov.fcn <- function(zmat, x, y, prb, nbh=4) {

  temp <- dim(zmat)
  n <- temp[1]
  m <- temp[2]

# Calculate indices of nearest neighbors

  idx <- apply(dist2full.fcn(dist(cbind(x, y))), 2, order)[1:nbh,  ]

# Make neighbors symmetric

  jdx <- rep(1:n, rep(nbh, n))
  kdx <- unique(c((jdx - 1) * n + idx, (idx - 1) * n + jdx)) - 1
  ij <- cbind((kdx) %/% n + 1, (kdx) %% n + 1)
  ij <- ij[order(ij[, 1]),  ]

# Apply linear taper to the  inverse probability weights

  gct <- tabulate(ij[, 1])
  gwt <- numeric(0)
  for (i in 1:n)
    gwt <- c(gwt, 1 - (1:gct[i] - 1)/(gct[i]))
  gwt <- gwt/prb[ij[, 2]]

# Normalize to make true average

  smwt <- sapply(split(gwt, ij[, 1]), sum)
  gwt <- gwt/smwt[ij[, 1]]
  smwt <- sapply(split(gwt, ij[, 2]), sum)
# Make weights doubly stochastic

  hij <- matrix(0, n, n)
  hij[ij] <- 0.5
  a22 <- ginverse(diag(gct/2) - hij %*% diag(2/gct) %*% hij)
  a21 <-  - diag(2/gct) %*% hij %*% a22
  lm <- a21 %*% (1 - smwt)
  gm <- a22 %*% (1 - smwt)
  gwt <- (lm[ij[, 1]] + gm[ij[, 2]])/2 + gwt

# Begin loops for variance-covariance calculations

  lmvar <- array(0, c(m, m))
  for (k in 1:m) {

    for (l in k:m) {

      z1 <- zmat[, k]
      z2 <- zmat[, l]

# Calculate neighborhoods

      zb1 <- sapply(split(z1[ij[, 2]] * gwt, ij[, 1]), sum)
      zb2 <- sapply(split(z2[ij[, 2]] * gwt, ij[, 1]), sum)

# Calculate the variance or covariance estimate

      lmvar[k, l] <- sum(gwt * (z1[ij[, 2]] - zb1[ij[, 1]]) * (z2[ij[, 2]] - zb2[ij[, 1]]))

    }

# Assign estimates that already have been calculated

    if (k > 1) {
      lmvar[k, 1:(k-1)] <- lmvar[1:(k-1), k]
    }
  }

# Return the variance-covariance estimate

  lmvar
}
```

```
# Program:
```
dist2full.fcn
```
# Date:    October 17, 2000
# Description:
#   This function takes the vector output from the S-Plus distance (dis)
#   function and reforms the output as a matrix.

dist2full.fcn <- function(dis)
{
  n <- attr(dis, "Size")
  full <- matrix(0, n, n)
  full[lower.tri(full)] <- dis
  full + t(full)
}
```

```
# Program:
```
wnas.fcn
```
# Date:    November 16, 2000
# Description:
#   For a vector this function returns the vector with missing values removed.
#   For a data frame this function returns the data frame with rows removed that
#   contain at least one missing value.  For a list with components of the same
#   length, this function returns the list with corresponding elements removed
#   when at least one list has a corresponding element that has a missing value.

wnas.fcn <- function(data) {
  if (is.list(data)) {
    n <- length(data)
    if (!is.data.frame(data)) data <- as.data.frame(data)
    for (i in 1:n) {
      wna <- which.na(data[,i])
      if (length(wna)) data <- data[-wna,]
    }
  } else {
    wna <- which.na(data)
    if (length(wna)) data <- data[-wna]
  }
  data
}
```

```
# Program:
```
write.object.fcn
```
# Date:    April 20, 2001
# Description:
#   This function writes an object to a plot.
#   Input:
#     obj = the object (either a data frame or a matrix)
#     n.digits = number of digits after the decimal point for numeric values
#       (the default is 2)
#     r.names = indicator for printing the row names (T=print the row names,
#       F=do not print the row names, the default is T)
#     c.names = indicator for printing the column names (T=print the column
#       names, F=do not print the column names, the default is T)
#     r.cex = character expansion parameter for the row labels (the default is
#       1)
#     c.cex = character expansion parameter for the column labels (the default
#       is 1)
#   Output: None
#   Other Functions Required: None

write.object.fcn <- function(obj, n.digits=2, r.names=T, c.names=T, r.cex=1, c.cex=1) {

# If the object is a matrix, convert to a data frame

  if(!is.data.frame(obj)) obj <- data.frame(obj, check.names=F)

# Assign the number of rows and number of columns for the plot

  dim.obj <- dim(obj)
  if (r.names)
    n.col <- dim.obj[2] + 1
  else
    n.col <- dim.obj[2]
  if (c.names)
    n.row <- dim.obj[1] + 1
  else
    n.row <- dim.obj[1]

# Create the plot area

  plot(seq(n.col), seq(n.col), xlim=c(0.5, n.col+0.5), ylim=c(0.5, n.row+0.5), type="n",
axes=F, xlab="", ylab="")

# Plot the object

  if (r.names) {
    if (c.names) {
      text(0.5, rev(1:(n.row-1)), dimnames(obj)[[1]], adj=0, cex=r.cex)
      text(2:n.col, n.row, dimnames(obj)[[2]], adj=0.5, cex=c.cex)
      for (i in 1:(n.col-1)) {
        text(i+1, rev(1:(n.row-1)), round(obj[,i], n.digits), adj=0.5)
      }
    } else {
      text(0.5, rev(1:n.row), dimnames(obj)[[1]], adj=0, cex=r.cex)
      for (i in 1:(n.col-1)) {
        text(i+1, rev(1:n.row), round(obj[,i], n.digits), adj=0.5)
      }
    }
  } else {
    if (c.names) {
      text(seq(n.col), n.row, dimnames(obj)[[2]], adj=0.5, cex=c.cex)
      for (i in 1:n.col) {
        text(i, rev(1:(n.row-1)), round(obj[,i], n.digits), adj=0.5)
      }
    } else {
      for (i in 1:n.col) {
        text(i, rev(1:n.row), round(obj[,i], n.digits), adj=0.5)
      }
    }
  }

# Place a box around the plot area

  box(2)

# Return a NULL object

  NULL
}
```