CTFS R Manual Outline
August 9, 2004

1.0 Installing and Customizing the R environment
The R environment - what's attached, what functions are available, how to load up packages, (hopefully how to load up a CTFS package)

2.0 Objects and how to "address" them
Explanation of vectors, matrices, dataframes, arrays and lists; Discussion of components of objects such a names (row, column, list) and how to address the objects.

3.0 Using the R Help Facility
How to read the help pages and get somewhere with R help.

4.0 File organization
Recommendation to use a system which will help ensure some order in the file location and an ability to clean up files after an extensive analysis without fear of losing valuable information.

6.0 Reading and Writing datafiles
Explanation of general knowledge of how to read and write datafiles as well as specific CTFS file structures and CTFS functions

1.0 Installing and Customizing the R environment

This manual is written for use by CTFS colleagues using R in a Windows (version 95 or later) operating system. Some information is provided that is specific to Mac OSX. The vast majority of the user's experience of the R environment will be identical across platforms.

1.1 Installing R

All CTFS partners working on analyzing CTFS datasets should have the most recent version of R installed on their computer. The R installation file can be downloaded from the CRAN homepage http://cran.r-project.org/. Users should return to this web page to keep track of changes and new releases of R.

From this website R users can easily update their R program. The R program can either be installed from a binary (precompiled form) or from source files (compiled locally on a user's machine). If you have previously installed R as a binary, continue to update it in the same form. If you have installed R by compiling source files locally, then do this for updating. Do not mix locally compiled R with binary precompiled R updates or vice versa.

As of writing this manual (August 2004), the most recent version of R for Windows and MacOSX is 1.9.1 in both binary (precompiled) and source files.

The easiest and fastest method of installing R is to locate the link for the operating system that you are using under the *"Precompiled Binary Distributions"* section of CRAN's homepage. The user should click on *Windows (95 or later)* and will then be prompted to choose between two subdirectories. Choose *base*. The user should now see a directory containing various R 1.9.1 files. To install the R program, the user should select "rw1091.exe", the user should click on "rw1091.exe", and choose to open the file upon completion of its download. This executable file (~20 MB) contains all of the relevant information for completely installing R on a computer. A new box will open outside of the internet browser entitled "Setup - R for Windows." The user must now manually complete the installation and setup of R. Now the user will need to navigate through several steps in which all of the default settings are appropriate, for example we recommend that the user accept the default R directory location; choose to install all of R's components; accept the default selection of a Start Menu folder; and select all of the additional tasks.

If R has installed correctly, a shortcut named "R 1.9.1" will appear on the user's windows desktop. This shortcut points to the file RGui.exe which is located in a folder named bin in the R directory. The R directory is by default: *c:\Program Files\R\rw1091* but to eliminate any future confusion, the R directory location will henceforth be referred to as *$R_HOME*. The R directory also contains ten other folders which contain all of the files needed to run R. The folder named library contains sub-folders for each package that automatically came with this version of R. Any package that is installed in the future,

including the **CTFS package**, will be installed into its own subfolder in the library. More instructions on how to download and install R are available on the CRAN homepage.

Installing R for OS is much the same. If using a precompiled binary form, chose the *"Precompiled Binary Distributions"* section on CRAN's homepage. Chose *MacOS X (10.2.x and above)*. The chose *R.dmg (1.9.1)* and download this disk image (option-click). Open and double click on "R-1.9.1.pkg" icon on the R.dmg disk image. Please also read the **ReadMe.txt** file inside the disk image to understand if you have to install further components.

1.2 Running R and Quitting R

To run R in Windows, select the shortcut to RGui.exe or the file itself. This will open **RGui** which is a graphical user interface for running R. For MacOS you can put R into the dock or an alias to R in a dock folder and run it from there.

Run R. A window called the *R Console* will open. R is interactive mode such that lines of code are evaluated immediately upon input. Commands are entered at the R prompt below the bar and the results of the command appear above. In R, a **session** begins each time R Console opens and ends each time R is exited. The **R workspace** is all of the R objects (discussed in Chapter 2,) that were created in the current, and possibly R sessions. This is also known as the **GlobalEnv**.

To quit R, select Exit from the File menu or type *'q()'* into the command line. R will then ask the user whether he/she wants to save the workspace image. If you answer yes, all of the R objects are saved into the file ".RData" in the default directory. These will be available the next time R is started and you load in this save workspace. If you answer no, all objects that have not been explicitly saved to a folder on disk will be deleted. (See below for more details).

In most cases, the user should answer NO since starting a new session with an old workspace means that much of the memory available for R is already used. Note that R generally requires 512 RAM to run analyses of full CTFS datasets. The entire workspace image or just specific R objects from the workspace image into a *.rdata file(s) outside of the R environment. For example, the command:

> *> save(mydata,file="04Aug26.RData")*"

saves the R object *mydata* into a file called "04Aug26.RData" located in *$R_Home*, unlike ".Rdata", this file's R objects will not load and use memory automatically. The workspace saved into any *.RData file can be restored by opening the *.RData file.

1.3 Important R Terms:

Ambiguity and uncertainty about the definitions of many R related terms is a source of

confusion and frustration for many CTFS colleagues. This confusion is also evident in the R documentation produced by the R project; many concepts are given several names and the same names are given to several different concepts. This section will try to define some of these terms in a straightforward and clear manner. Let us know if these definitions work for you and if they are used consistently in this manual and other documentation.

The **R Environment** is meant to encompass everything that is currently accessible to the user when R is opened. This includes the R Console, the R Window, the objects, packages and help pages available to the user. It also includes files that are read into the current R session using the appropriate R functions.

The **R Environment** is divided into **search paths** which contain all of the objects (functions, variables, datasets, etc.) that the **R Environment** has available at any given time.

Search Path 1, which is also referred to as the **Global Environment** and the **Workspace Image**, is the location of all objects that are created at the R command line. Objects that are "sourced" and "loaded" into R are also put into this search path. (See Section 1.5 Useful Functions for explanation of *source()* and *load()*.)

Search Path 2, 3, 4, **etc.** are the location of all packages that are "installed" or "loaded" into the **R Environment** as well as R objects that are attached to the to it. (See Section X.x Useful Functions for explanation of *attach()*.)

1.4 R Commands:

R is an expression language with a very simple syntax. It is case sensitive, so "A" and "a" are considered different statements in R. For example,

```
> A = 2
> a = 1
> A
[1] 2
> a
[1] 1
```

All R commands can be classified as either expressions or assignments. An **expression** is a command that is evaluated and printed but does not change the value of an object.

```
> 2 + 2
> 4
```

An **assignment** is a command that is also evaluated and the result is saved into an object rather than displayed. The result can be displayed by properly addressing an object.

```
> a <- 2 + 2
> a
> 4
```

If a command is not complete at the end of a line, R will give a different prompt, by default '+' on second and subsequent lines and continue to read input until the command is syntactically complete.

```
>a <- 2 +
> +
```

In R, the user can use two assignment operators (almost) interchangeably. Above, the assignment operator used was "<-" but "=", another assignment operator, could have been used instead. There are differences in the behavior of "<-" and "=". The "<-" or "->" operators are pointers and the "=" is an assignment. But for most instances, they behave the same way.

1.5 Useful Functions:

The following functions are used to control the contents of the search paths, allow objects in the workspace to be saved to disk and provide summary information on the structure of an object.

search(): displays the names all attached packages and R objects in all of the search paths.

```
> search()
 [1] ".GlobalEnv"              "file:bci9095.full.rdata"
 [3] "file:bci95.mult.rdata"   "file:tst.bci9095.spp.rdata"
 [5] "package:CTFSAUG"         "package:methods"
 [7] "package:stats"           "package:graphics"
 [9] "package:utils"           "Autoloads"
[11] "package:base"
```

The **GlobalEnv** *is* the first search path. There are 3 datafiles in the next search paths (2 to 4). A test package of CTFS functions has been loaded into the 5th path. Then the standard packages that are loaded when R is run occupy the remaining paths (6 to 11).

ls(): displays the names of the objects in the search path specified by the argument, a number, passed to the function. When used without an argument (default) the objects in the **GlobalEnv** are listed. When used with a number, the objects in that search path are listed.

```
> ls()
[1] "mort.out"
```

```
> ls(2)
[1] "bci9095.full"

> ls(5)
 [1] "abundanceperquad"      "assemble.demography"   "fill.dimension"
 [4] "find.climits"          "growth"                "growth.dbh"
 [7] "growth.eachspp"        "growth.indiv"          "gxgy.to.hectindex"
[10] "gxgy.to.index"         "gxgy.to.rowcol"        "index.to.gxgy"
[13] "index.to.rowcol"       "merge.census"          "mortality"
[16] "mortality.calculation" "mortality.dbh"         "mortality.eachspp"
[19] "pop.change"            "recruitment"           "recruitment.eachspp"
[22] "rndown5"               "rowcol.to.index"       "select.dbhrange"
[25] "split.data"            "split.dbh"             "split.grform"
[28] "split.habitat"         "text.to.rdata"         "totalabund"
[31] "totalabund.spp"        "trim.growth"           "unwind.matrix"
```

load(): searches (in the working directory unless otherwise specified by the user) for a file matching the name (in quotation marks) and then copies the file into search path 1 (**GlobalEnv**). If the desired file is in the working directory then no path name is needed. Otherwise, the full path and filename must be provided.

```
> load("bci.spp.info.rdata")
> ls()
[1] "bci.spp.info" "mort.out"
```

After a file (*.rdata) is loaded, it can be addressed by its name and the “rdata” extension is not a part of its name. For more information about how to organize your R related files read chapter 4.

attach():searches (in the working directory unless otherwise specified by the user) for a file matching the name (in quotation marks) and then copies the file into search path 2. All objects in search path 2 and higher are shifted one search path higher. After a file or database (*.rdata) is attached, it can be referenced without its “.rdata” extension.

```
> attach("bci.spp.info.rdata")
> search()
 [1] ".GlobalEnv"            "file:bci.spp.info.rdata"   "file:bci9095.full.rdata"
 [4] "file:bci95.mult.rdata" "file:tst.bci9095.spp.rdata" "package:CTFSAUG"
 [7] "package:methods       " "package:stats"            "package:graphics"
[10] "package:utils"         "Autoloads"                 "package:base"
```

Note that *bci.spp.info.rdata* is now is search path 2 and all of the preexisiting objects have been “pushed down” one more search path. Compare with the use of *search()* above. After a file (*.rdata) is loaded, it can be addressed by its name and the “rdata” extension is not a part of its name. For more information about how to organize your R related files read chapter 4.

source(): This function reads R code from a file. In this way, r can automatically get large chunks of code or functions at once. The code is saved into search path 1.

```
>source("basal.area.r")
> ls()
[1] "ba"              "ba.ind"          "ba.perquad"      "ba.tot"
[5] "ba.tot.spp"      "bci.spp.info"    "mort.out"        "pop.change.ba"
```

Note that the file *basal.area.r* contained 6 functions. These functions are now all available in the GlobalEnv along with the preexisting objects. (See use of *ls()* above).

str(): displays the structure (i.e. its type and associate information about the type: list, data frame, vector, etc.) of an R object as well as parts of the object itself to allow the user to address and use the object properly. The name of the object that you want to see the structure of must be within the R environment and must be provided to the function as an argument.

```
> str(bci.spp.info)
`data.frame':   318 obs. of 8 variables:
 $ sp      : chr  "acacme" "acaldi" "acalma" "ade1tr" ...
 $ genus   : chr  "Acacia" "Acalypha" "Acalypha" "Adelia" ...
 $ species : chr  "melanoceras" "diversifolia" "macrostachya" "triloba" ...
 $ family  : chr  "Fabaceae:Mimos." "Euphorbiaceae" "Euphorbiaceae"
"Euphorbiaceae" ...
 $ grform  : chr  "U" "S" "U" "U" ...
 $ repsize : int  4 2 2 10 4 20 16 4 4 20 ...
 $ breedsys: chr  "B" "M" "M" "D" ...
 $ maxht   : chr  "6" "6" "5" "5" ...
```

save(): writes an R object(s) to an external file on disk. The argument(s) are the names of object(s) in the R environment to be saved as well as a specification of the file to which to save. The full path must be specified if the current directory is not where the file should be placed. The file name must be in quotes.

```
> save(bci.spp.info, tst.bci95.spp.rdata, myspp, file = "SppAug04.rdata")
> save(mort.out, file="mort.out.rdata")
```

rm(): removes R object(s) from the search path. By default, it removes from search path 1, but by passing the function a value for its parameter "pos", one can use the function to remove objects from the other search paths.

```
> rm(bci.spp.info, tst.bci95.spp.rdata)        #removes objects from search path 1
>rm(myspp,  pos=3)            #removes myspp from search path 3
```

detach(): detaches a database, i.e., removes it from the search() path of available R objects. Usually, this is either a data frame which has been attached or a package which was required previously. It does not detach objects from search path 1 nor can it be used to detach the package called base. By default the object in search path 2 is removed. Provide the search path number to remove an object from a specific location. When an object is removed, all remaining objects advance one search path.

> *detach()*

history(): is used to display a history of the previous commands in the workspace. By default, the function displays the last 25 commands in a separate window. For example,

> *history()*

If the user wants to see a specific number of commands, he/she must pass the number of commands as an argument in the function call. For example, the following will display the last 10 commands:

> *history(10)*

Previous commands can be displayed directly in the R console box with the use of the back arrow key. The previous commands can be edited and reentered.

gc(): causes a garbage collection, or memory reallocation, to take place.
Although memory allocation and reallocation takes place automatically, sometimes things can get "boloxed" up in memory by accidental overwriting or pointers to memory locations being lost. This is not often obvious to a user until file contents change unexpectedly or an object that was accessible becomes inaccessible without cause

> gc()

1.6 Using a Text Editor for Programming

R Console is very convenient when you wish to immediately see the results of a single command line. However, oftentimes the user may wish to write several lines of code before seeing any results. In this situation, the user could write a series of commands in a text editor, save the commands as a simple text file in $R_Home with the extension ".r" and then *source()* this file to make its contents available in R. This is how all R functions are written.

Windows comes with two simple text editors: WordPad and Notepad. There are also numerous additional freeware and shareware text editors that can be downloaded off of the internet, such as Alphatk (http://www.santafe.edu/~vince/Alphatk.html), Winedt (http://www.winedt.com/) and EMacs (http://www.gnu.org/software/emacs/emacs.html#Obtaining). For the MacOS X, AlphaX or Emacs work are both powerful text editors and have capacity to keep track of complex

function development projects.

1.7 Installing and Loading Extra Packages

A long-term goal of CTFS is to develop a package distribution of its functions, datasets and documentation. The package is not ready for distribution.

Packages that do not come pre-installed into R are generally distributed as compressed files for Windows users. In order to use one such package, a user should uncompress, or extract, the file into $R_HOME\library. This should create a folder under library named after the package name.

Occasionally, the extractive process creates an extra folder bearing the packages name into which the actual package folder and all of its files are placed. It is necessary to eliminate this extra folder so that the path to the package is: *$R_HOME\library\package name* rather than *$R_HOME\library\package name\package name* in order for R to recognize the package as installed.

Once the package is installed in the *$R_HOME\library*, the package must be attached to the current R session. Do this selecting the *Packages* menu. And then select the "*Load Package…"* option. A box will appear that lists each package contained in the user's $_Home\library folder. The user should highlight the name of the package (varies based on version) and select OK.

For MacOS X, there is the *Packages* menu that provides options for installing and loading packages and datasets. Packages that have been downloaded on the local machine can be installed either as compressed files, as source or as precompiled packages (see options in *Install from local files*). Installed packages are placed in *$R_HOME\library*. The *Package Manager* menu attaches an installed package. The same is true for *Datasets Manager*. Both provide a window with a checkbox and a single line (title) package description. Check the packages that you wish to attach.

library() loads a package from *$R_HOME/library* into the current R Environment.

> *> library("CTFSAUG")*

This loads the entire contents of the CTFSAUG package into the GlobalEnv. *library()* is a *load()* command not an *attach()* command.

Additional packages are maintained and distributed by CRAN (http://cran.us.r-project.org/src/contrib/PACKAGES.html)

1.8 Customizing your R Environment

The various characteristics of the R Environment are set in R preferences which is used each time R is run. This file can contain many user specific preferences including

commands to *load* certain package(s) is into one's workspace upon initiation of a new R session. A Using a text editor, the user should open the file $R_HOME\R\rw1091\etc\Rprofile. At the bottom of the file, he/she should add the following line: "*library("Package Name")*". We strongly recommend that users do not change anything else in *Rprofile*.

Alternatively, one can make a *mystartup.r* file that contains commands you would like to have executed for each new session of R. This can be the first file you source upon running R. Here is an example of some of useful things to do to make the R environment easy to work in. The lines in such a file can be easily changed to accommodate the changing needs of an ongoing analysis or function development project. All lines that begin with "#" are comments and are not executed.

File: *CTFS.development.setup.R*

```
# install and load most recent version of CTFS package AND survival
library(survival)
library(CTFSAUG)

# attach datasets by changing working directory and attaching desired files
setwd("/Users/phall/Documents/Research and Datasets/CTFS R&D/CTFS
datasets 2003/BCI")
attach("bci.quad.info.rdata")
attach("bci.spp.info.rdata")
attach("bci9095.full.rdata")
attach("bci95.mult.rdata")
attach("bci95.full.rdata")
attach("tst.bci95.full.rdata")
attach("tst.bci9095.full.rdata")

# run help for html files
help.start()

# set working directory to the one where functions under development are being
kept.
setwd("/Users/phall/Documents/CTFS_Package_Development/programs/R
programs development")
```

2.0 R objects and how to "address" them

R operates on objects which are named data structures. This section explains the main types of these data structures used by CTFS functions: vectors, matrices, arrays, data frames, and lists.

2.1 Vector

A vector is a set of contiguous cells containing data of a single type. R has five basic vector types: logical (true or false), integer (1,2,3…), real (65.354, 89.23, 12.3…), complex (2 + 3i, …), and string or character ("A", "B", "C", "termam", "vismma", "PSYDEF" ...). Single numbers, such as 4.2, and strings, such as "four point two" are vectors of length 1. The type "complex" will not be futher dealt with.

Vectors can be created using *c( ). c()* concatenates, or combines, all the arguments (i.e. objects inside the parentheses) forms them into a vector. The following commands will create a vector consisting of the five numbers:

```
> example.vector <- c(10.4, 5.6, 3.1, 6.4, 21.7)
> example.vector
 [1] 10.4  5.6  3.1  6.4 21.7
```

The cells of a vector are addressed through indexing operations. There are a variety of way to express the index of an R object. Some of them produce identical results but one form may be easier to understand in a given situation than another. Other forms are specific and can only be used in one manner.

The first index operator is *[ ]*, the square brackets. For example the *n*-th element can be accessed using the notation: *x[n]*.

```
>example.vector[5]
> [1] 21.7
```

The nth to the nth+x element can be address using ":" to indicate range.

```
> example.vector[2:5]
[1]  5.6  3.1  6.4 21.7
```

Note what happens when the range specified is invalid for the vector:

```
> example.vector[2:7]
[1]  5.6  3.1  6.4 21.7   NA   NA
```

Vectors can be used in arithmetic expressions. The arithmetic operations are performed element by element, i.e. are performed individually on each element on the vector. The elementary arithmetic operators are the usual +, -, *, / and ^. In addition all of the common arithmetic functions are also available: log, exp, sin, cos, tan, sqrt, etc..

```
> example.vector2 <- example.vector + 3
 > example.vector2
[1] 13.4  8.6  6.1  9.4 24.7
```

2.2 Matrix

A matrix can be thought of as a two-dimensional vector. Like a vector, a matrix can only contain a single type of data, either numeric or character. Like vectors, matrices can be used in arithmetic expressions and the operation is performed on the entire matrix as is done in matrix algebra. (This topic will not be dealt with here. See any linear algebra text book or the R manual: "Introduction to R; Arrays and Matrices".)

A matrix can be created with *matrix()*.

```
> matrix(data, nrow, ncol, byrow=F, dimnames)
```

data : value(s) with which the matrix will be filled.
nrow : the number of rows.
ncol : the number of columns
byrow : a logical value (either TRUE or FALSE). If TRUE , the matrix is filled by rows, i.e. values are inserted in order into the matrix row by row rather than by columns. Otherwise, by default the matrix is filled by columns.
dimnames : dimensions, i.e. a name for the rows and a name for the columns. By default, dimnames is set to NULL, i.e. no names are given to the rows and columns.

The following R command sets up a five element by five element matrix named *example.matrix* with values 1:25 filled by columns with the values 1 to 25. The contents of the matrix are displayed:

```
> example.matrix <-matrix(1:25, nrow = 5, ncol = 5, byrow = FALSE, dimnames
= NULL)
> example.matrix
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    2    7   12   17   22
[3,]    3    8   13   18   23
[4,]    4    9   14   19   24
[5,]    5   10   15   20   25
```

Here's what happens when the values of 1 to 25 are filled into the matrix when *byrow =TRUE*:

```
> example.matrix<-matrix(1:25,nrow=5,ncol=5,byrow=TRUE,dimnames=NULL)
> example.matrix
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
```

```
[2,]    6    7    8    9   10
[3,]   11   12   13   14   15
[4,]   16   17   18   19   20
[5,]   21   22   23   24   25
```

In the previous example, all five of the arguments were supplied in the function call to *matrix()*. However, because ncol, byrow and dimnames were passed their default values, they could have been omitted from the function call. The exact same matrix could have been made using the following command:

```
> example.matrix.col <-matrix(1:25, nrow = 5)
> example.matrix.col
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    2    7   12   17   22
[3,]    3    8   13   18   23
[4,]    4    9   14   19   24
[5,]    5   10   15   20   25
```

The cells of a matrix are addressed and displayed through indexing operations. Since a matrix has 2 dimensions, both rows and column have to be specified. The first value is for rows, the second for column. If ALL rows or ALL columns are displayed, then a "," is used.

Here are several examples of how to address *example.matrix*:

```
> example.matrix[1,3]          # 1st row, 3rd column
[1] 11
> example.matrix[2,]           # 2nd row, all columns
[1]  2  7 12 17 22
> example.matrix[,4]           # all rows, 4th column
[1] 16 17 18 19 20
> example.matrix[1:2,3]        # rows 1 to 2, 3rd column
[1] 11 12
```

```
> example.matrix[1:2,1:2]      The first 2 rows and first 2 columns
     [,1] [,2]
[1,]    1    6
[2,]    2    7
```

Matrices can be assigned row and column names. R recognizes two methods for assigning row and column names: 1) passing a list of the names (as character strings in quotation marks) as an argument to the matrix() function or 2) use the function *dimnames*() to alter a preexisting matrix. For example:

```
> cols<- c("Uno","Dos","Tres","Cuatro","Cinco")
```

```
> rows<- c("Un","Deux","Trois","Quatre","Cinq" )
> dimnames(example.matrix)<-list(rows,cols
> example.matrix
        Uno Dos Tres Cuatro Cinco
Un        1   6    11    16     21
Deux      2   7    12    17     22
Trois     3   8    13    18     23
Quatre    4   9    14    19     24
Cinq      5  10    15    20     25
```

Once assigned the contents of rows and columns can be assessed by the names of the row and column in quotes or by their position.

```
> example.matrix["Quatre","Dos"]
[1] 9
> example.matrix[4,2]
[1] 9
```

Another way of creating matrices is to bind vectors together using the function *rbind ()* which takes vectors as rows and "binds" them together to make a matrix. See the R help pages for further information on *rbind()* and *cbind()*.

2.3 Arrays:

An array is a more general data construct that can have anywhere from zero to eight dimensions. In fact, vectors are special cases of one-dimensional arrays and matrices of two-dimensional arrays. Arrays can be used in arithmetical expressions. The general syntax of the array function is as follows:

```
> array (data, dim, dimnames)
```

data : the value(s), in the form of a vector, with which the array will be filled.
Dim : a vector containing the dimensions of the array. Note that *dim()*will provide the number of dimensions on an array.
dimnames : the names of each of the dimensions. By default, *dimnames()* is set to NULL, i.e. no names are given to the dimensions.

As with matrices, it is not necessary to pass the function *array()* all of its arguments.

Here is an example of how to create a three-dimensional array in which the first dimension has 3 elements, the second dimension has 4 elements and the third dimension has 2 elements filled with the numbers 1:24:

```
> example.array<-array(1:24, c(3,4,2))      # (rows, columns, sets)
 > example.array                            # displays the array
, , 1      # first set
     [,1] [,2] [,3] [,4]
```

[1,] 1 4 7 10
[2,] 2 5 8 11
[3,] 3 6 9 12

, , 2 # second set
[,1] [,2] [,3] [,4]
[1,] 13 16 19 22
[2,] 14 17 20 23
[3,] 15 18 21 24

Data are addressed and displayed from an array by specifying the desired position of each dimension. Here are several examples of how to access certain positions within our array:

> example.array[1,4,2] # displays the value in the first row,
 # fourth column of the second set
[1] 22
> example.array[1,,1] # displays the values in the first dimension row of the first set
[1] 1 4 7 10

Like *matrix(), array()* allows the user to name the dimensions of an array. *array()*can be passed a list of the desired names or *dimnames()* can be used (see matrix explanation). Once assigned, the dimensions can be addressed and displayed by their names or by their position.

2.4 Data Frames:
A data frame consists of an ordered collection of objects known as its components. The components of a data frame can be of different types (vectors or matrices) but each must have the same length. This means the number of values in each row must be the same. Some value must be present for each column for each row. Data frames are differentiated from other two-dimensional arrays because variables of different types can be mixed and the length of the rows must be identical. Data frames are extremely useful for creating and storing CTFS datasets because of the variety of variables in these datasets and the ease with which a data.frame handles this variation.

The general syntax of the data.frame function is as follows:

> data.frame(data1, data2, …)

The notation (*data1, data2, …*) means that the function will accept as many datasets as it is passed as arguments.

In the following example, a data frame of six components is constructed. The components are the 5 columns of *example.matrix* and the single column of

example.vector created in the previous examples. Each column of *example.matrix* becomes a component of the data frame.

```
> example.dataframe <- data.frame(example.matrix, example.vector)
> example.dataframe          # displays the data frame
  X1 X2 X3 X4 X5 example.vector      # generated component names
1  1  6 11 16 21          10.4
2  2  7 12 17 22           5.6
3  3  8 13 18 23           3.1
4  4  9 14 19 24           6.4
5  5 10 15 20 25          21.7
```

As with the previous R objects, the *dim()* can be used to assess the dimensions of a data frame.

```
> dim(example.dataframe)
[1] 5 6
```

The first value returned is the number of rows, the second the number of columns. Note that *length()* returns only the length of the rows, that is the number of columns.

```
> length(example.dataframe)
[1] 6
```

When a data frame is created, names are created for its components based upon the names of the objects from which the data frame was constructed. In the above example, the dimensions of example.matrix and example.vector have already been named. Displaying the entire data frame results in:

```
> example.dataframe
      Uno Dos Tres Cuatro Cinco example.vector
Un      1   6   11     16    21          10.4
Deux    2   7   12     17    22           5.6
Trois   3   8   13     18    23           3.1
Quatre  4   9   14     19    24           6.4
Cinq    5  10   15     20    25          21.7
```

These names of the components can be modified using the assignment function *dimnames()* for both dimensions at the same time or with *rownames()* and *colnames()* for only rows and columns, respectively.

```
> rownames(example.dataframe)
[1] "Un"     "Deux"   "Trois"  "Quatre" "Cinq"

> colnames(example.dataframe)
[1] "Uno"          "Dos"          "Tres"         "Cuatro"       "Cinco"
"example.vector"
```

```
> dimnames(example.dataframe)
[[1]]
[1] "Un"     "Deux"   "Trois"  "Quatre" "Cinq"

[[2]]
[1] "Uno"          "Dos"          "Tres"         "Cuatro"       "Cinco"
"example.vector"
```

To get a quick version of the contents of a data frame, its names and structure (number of rows and column), use *str()*

```
> str(example.dataframe)
`data.frame':       5 obs. of  6 variables:
 $ Uno          : int  1 2 3 4 5
 $ Dos          : int  6 7 8 9 10
 $ Tres         : int  11 12 13 14 15
 $ Cuatro       : int  16 17 18 19 20
 $ Cinco        : int  21 22 23 24 25
 $ example.vector: num  10.4 5.6 3.1 6.4 21.7
```

str() tells you that the object is, indeed, a '*data.frame'* that it has 5 observations (rows) and 6 variables (columns). It provides the names for each column (but not the rows), the type of object in each column (*int* means an integer) and provides some of the values. In this case *example.dataframe* is small enough that its entire contents are displayed. But on larger data frames, all columns and their type will be provided but only a few rows of their values. Row names are not displayed. Rows are often not named other than by reference to their number (order) in the data frame.

Data in a data frame can be addressed and displayed in a greater variety of ways then from a matrix. Here are some examples using *example.dataframe.*

```
> example.dataframe[1:3,]
     Uno Dos Tres Cuatro Cinco example.vector
Un     1   6   11     16    21          10.4
Deux   2   7   12     17    22           5.6
Trois  3   8   13     18    23           3.1

> example.dataframe[,1:3]
      Uno Dos Tres
Un      1   6   11
Deux    2   7   12
Trois   3   8   13
Quatre  4   9   14
Cinq    5  10   15
```

```
> example.dataframe[1:3,1:3]
     Uno Dos Tres
Un     1   6   11
Deux   2   7   12
Trois  3   8   13
```

In addition the operator "$" can be used to address columns by column name. This cannot be used for row names.

```
> example.dataframe$Uno
[1] 1 2 3 4 5

> example.dataframe$Deux
NULL
> example.dataframe$Uno[3]
[1] 3
```

However, addressing a data frame as is more typical of some programming languages for addressing arrays is NOT valid. The incorrect and correct form is shown below.

```
> example.dataframe[1][2]
Error in "[.data.frame"(example.dataframe[1], 2) :
        undefined columns selected

> example.dataframe[1,2]
[1] 6
```

2.5 Lists:

A list is another R object consisting of an ordered collection of objects known as its components. Like data frames, the components of a list do not have to be of the same type. A list has is more flexible than a data frame because its components may also have different structures, i.e. each component may have a different length or row size. For example a list of 3 components could contain a vector of length 5 filled with logical values, a single character string, and a matrix of dimension 4 x 8 filled with numbers.

The general syntax of the list function is as follows:

```
> list(...)
```

Where the arguments supplied (…) are the objects which will become the components of the list.

Because lists can include many different structures, it is very important to learn how to address each component, to address elements of each component and to understand what portion of the list is being addressed. Single brackets "[ ]" and double brackets "[[ ]]" are used individually or together to address different components of a list and different

elements of its components, respectively. The components of lists also can have names including row and column names for elements where appropriate (when a component is a matrix, array or data frame). Lists can also be composed of lists which had components of their own. The "[[ ]]" address different "levels" of the large list.

This takes some practice, so let's work with an interesting list and see how to address its contents. The dataset *tst.bci9095.spp* is a list of data frames. If the user has the CTFS package loaded on his/her machine, he/she can directly load such an example dataset into his/her search path. Otherwise, attach the dataset which is located in *$R_HOME/library/CTFS/data.*

The following command and display use the functions *str()* to show the structure and basic organization of this list.

```
> str(tst.bci9095.spp)          # str() displays the structure of the object
List of 3
 $ alsebl:`data.frame':  11128 obs. of  13 variables:
  ..$ tag    : num [1:11128] -27784     47     49     68     71 ...
  ..$ gx     : num [1:11128]   -9  984  985  986 1000 ...
  ..$ gy     : num [1:11128]  -9 342 329 276 278 ...
  ..$ dbh0   : num [1:11128] NA 437 228 278 269 360 580 NA 311 348 ...
  ..$ dbh1   : num [1:11128] NA 426 228 277 318 368 580 NA 318 351 ...
  ..$ pom0   : num [1:11128] 0 2 1 1 1 2 2 0 2 2 ...
  ..$ pom1   : num [1:11128] 0 3 1 1 1 2 2 0 2 2 ...
  ..$ date0  : num [1:11128]    0 3702 3632 3627 3627 ...
  ..$ date1  : num [1:11128]    0 5382 5396 5390 5390 ...
  ..$ codes0 :Class 'AsIs'  chr [1:11128] "*" "B" "*" "*" ...
  ..$ codes1 :Class 'AsIs'  chr [1:11128] "*" "B" "*" "*" ...
  ..$ status0:Class 'AsIs'  chr [1:11128] NA "A" "A" "A" ...
  ..$ status1:Class 'AsIs'  chr [1:11128] NA "A" "A" "A" ...
 $ psycde:`data.frame':160 obs. of  13 variables:
  ..$ tag    : num [1:160]  9472 15954 20840 25559 31664 ...
        ...
 $ socrex:`data.frame': 1133 obs. of  13 variables:
  ..$ tag    : num [1:1133] 12330 12359 12444 12616 12636 ...
        ...
```

This list consists of 3 objects: *List of 3.*
These objects are each data frames: *$ alsebl:`data.frame'.*
The first data frame for *alsebl* has 11128 rows and 13 columns.
The second data frame for *psycde* has 160 rows and 13 columns and etc.

Note that only the complete description of the first data frame is provided because all of the other data frames have the same structure though their dimensions vary.

length () returns the number of components in the list which is the number of data frames.

```
> length(tst.bci9095.spp)
[1] 3
```

names() returns the names given to the components in a list in this case, each data frame contains the trees for a single species, so the names of the data frames are the species names.

```
> names(tst.bci9095.spp)
[1] "alsebl" "psycde" "socrex"
```

Here are some examples of how to access a component in this list:

```
> tst.bci9095.spp[2]
$psycde
      tag    gx    gy dbh0 dbh1 pom0 pom1 date0 date1 codes0 codes1 status0 status1
261491   9472 993.0 101.8   NA   NA    0    0  3613  5354     DN      *      D      D
261492  15954 843.6 126.0   NA   NA    0    0  3572  5318     DS      *      D      D
...
```

In this example [2] selects and displays the second component from the list *tst.bci9095.spp*, which is a data frame of the species psycde. This component is given the name *$psycde* in the list. Note that, in reality all 160 of the elements of this component would display rather than the abbreviated 2 elements that are displayed above. If the user asks R to list this data frame the entire R console would be filled with data. So take care with what is requested for display!

```
> tst.bci9095.spp[2:3]
$psycde
      tag    gx    gy dbh0 dbh1 pom0 pom1 date0 date1 codes0 codes1 status0 status1
261491   9472 993.0 101.8   NA   NA    0    0  3613  5354     DN      *      D      D
261492  15954 843.6 126.0   NA   NA    0    0  3572  5318     DS      *      D      D

$socrex
      tag    gx    gy dbh0 dbh1 pom0 pom1 date0 date1 codes0 codes1 status0 status1
288000  12330 964.4 421.0   NA   NA    0    0  3634  5380      *      *      D      D
288001  12359 964.1 435.0  128  117    1    1  3634  5380      *      *      A      A
```

The [2:3] selects a range of components in positions 2 through 3 in the list. Note that there is no "," inside the brackets because [2:3] represents only one dimension of components in the list: the names of the species. Again, this returns ALL of the rows, or trees, for each species; 160 for psycde and 1133 for socrex.

This is an incorrect use of the single brackets,

```
> tst.bci9095.spp[2,]
```

Error in tst.bci9095.spp[2,] : incorrect number of dimensions

because the single brackets refer to the components of the list, not the elements of the components.

Addressing the elements of the components within the list directly is done using "[[ ]]". For example,

```
> tst.bci9095.spp[[2]]
      tag    gx    gy dbh0 dbh1 pom0 pom1 date0 date1 codes0 codes1 status0 status1
261491  9472 993.0 101.8   NA   NA    0    0  3613  5354     DN      *       D       D
261492 15954 843.6 126.0   NA   NA    0    0  3572  5318     DS      *       D       D
...
```

refers to the second component in the list and returns the components of that component, which is in this case a data frame. Using "*[[ ]]*" refers directly to the object as if that object were not a component of the list at all. See how the output differs with the use of "*[ ]*" or "*[[ ]]*".

```
> tst.bci9095.spp[2]
$psycde
      tag    gx    gy dbh0 dbh1 pom0 pom1 date0 date1 codes0 codes1 status0 status1
261491   9472 993.0 101.8   NA   NA    0    0  3613  5354     DN      *       D   D
261492  15954 843.6 126.0   NA   NA    0    0  3572  5318     DS      *       D   D
…
```

tst.bci9095.spp[2], refers to the data frame as the second component within the list *tst.bci9095.spp*. This is evident because the name of the component is the first output. While, *tst.bci9095.spp[[2]]*, refers to the data frame as an individual and separate data frame. When the data frame is referenced with the [[ ]], its elements can now be accessed by the user. For example, the following commands display the first five values of each of the thirteen elements within the data frame which is the second component of the list:

```
> tst.bci9095.spp[[2]][1:5,]     # displays the 1st 5 values of all of the elements
      tag    gx    gy dbh0 dbh1 pom0 pom1 date0 date1 codes0 codes1 status0 status1
261491  9472 993.0 101.8   NA   NA    0    0  3613  5354     DN      *       D       D
261492 15954 843.6 126.0   NA   NA    0    0  3572  5318     DS      *       D       D
261493 20840 955.2 104.7   NA   NA    0    0  3620  5342     DS      *       D       D
261494 25559 920.8  62.8   NA   NA    0    0  3598  5340      *      *       D       D
261495 31664 901.4 199.0   NA   NA    0    0  3613  5369     DN      *       D       D
```

Again, the second set of brackets ("*[1:5,]*") refers to the components of the data frame. In this case the first 5 rows (1:5) with all columns (,) are returned.

If a user were to attempt to access the elements of a list's component without the use of the component's name or of double brackets, the user would get an error as follows:

```
> tst.bci9095.spp[2][1:5,]
Error in tst.bci9095.spp[2][1:5, ] : incorrect number of dimensions
```

The following example addresses the first through fifth values of the fourth and fifth elements of the data frame that is the second component of the list *tst.bci9095.spp:*

```
> tst.bci9095.spp[[1]][1:5,4:5]
     dbh0 dbh1
4218   NA   NA
4219  437  426
4220  228  228
4221  278  277
4222  269  318
```

Like with data frames, one may omit the use of "*[[ ]]*" in order to access the elements of the components within a list by substituting the use of "$" and the name of each object. For example:

```
> tst.bci9095.spp$psycde
      tag    gx    gy dbh0 dbh1 pom0 pom1 date0 date1 codes0 codes1 status0 status1
261491   9472 993.0 101.8   NA   NA    0    0  3613  5354     DN      *       D   D
261492  15954 843.6 126.0   NA   NA    0    0  3572  5318     DS      *       D   D
…
```

This call returns the same results as the call *tst.bci9095.spp[[2]]*. Similarly, the call *tst.bci9095.spp$psycde[1:5,4:5]* returns the same values as *tst.bci9095.spp[[2]][1:5,4:5]*.

Here's a way to see the first 5 rows of *alsebl*, all columns are displayed.

```
> tst.bci9095.spp$alsebl[1:5,]
      tag    gx    gy dbh0 dbh1 pom0 pom1 date0 date1 codes0 codes1 status0
status1
4218 -27784  -9.0  -9.0   NA   NA    0    0     0     0      *      *  <NA>    <NA>
4219     47 984.3 341.6  437  426    2    3  3702  5382      B      B     A       A
4220     49 985.3 328.9  228  228    1    1  3632  5396      *      *     A       A
4221     68 985.7 275.8  278  277    1    1  3627  5390      *      *     A       A
4222     71 999.9 277.6  269  318    1    1  3627  5390      M      *     A       A
```

Or only a given column can be displayed, using the column names. Note that this returns a vector (just *dbh0* of *alsebl*). If the "[ ]" were not used, all *dbh0* values for all 11128 trees would be displayed. This can be controlled by using the "[ ]", but without the ",", because the *dbh0* is only a vector.

```
> tst.bci9095.spp$alsebl$dbh0[1:5]
```

[1] NA 437 228 278 269

A list may have almost any type of object as a component. In these examples we address data frames only but a list can also contain a component of various lists. Accessing data from a list within a list is more complicated.

3.0 Using the R Help Facility

The purpose of this chapter is to orient the user to the format and terminology used in the R help facility so that the help pages for R functions become a substantial resource for the user. The R help pages are very similar in form and terminology to Unix man pages.

3.1 Help Pages

The R help facility contains a collection of help pages each of which provides a description and explanation of the use and arguments of a particular function. Help pages are an invaluable resource in R. As the library of CTFS functions has grown, workshop instructors have begun developing help pages which document each of the CTFS functions. At this time a set of stand along HTML pages are available that provide a quick reference guide to the use of many CTFS functions (http://ctfs.si.edu/workshop).

In the future, a CTFS R package will have help pages for CTFS functions and for general information on types of analyses. These manual chapters and teaching tools will also be integrated into the package. The help pages can be accessed within and R session or directly by using the html version of the files. The links among files should work under both circumstances.

When the future CTFS package is loaded into the R environment, its help pages are made available to the user through R just as any other loaded R package.

Help pages appear as simple text "man" pages by default. If *help.start()* is run (often included in the R preferences file or a startup file), then help pages in HTML format appear. From here it is assumed that the HTML format is available.

3.2 Accessing Help Pages

Help pages can be accessed from within the open R environment by:

(1) selecting "HTML help" from the HELP menu
(2) ?topic
(3) help(topic)
(4) help.search("concept")

Selecting R HELP from the Help Menu will open an html file with links to installed R packages (including the future CTFS package). Let's follow the links through a test run of the CTFS R package.

Open R Help, click on "Packages"

An index of all available R packages that came with the R installation and any locally installed packages appears.

Click on "CTFSAUG"

An index of all available functions in the test version of the CTFS package, CTFSAUG, appear with a short (title) description of each function. You can follow these links to help pages on individual files. Note that at the top of the page "overview" and "directory" information is provided.

Click on "overview"

An index to CTFSAUG "vignettes" appears. A "vignette" is a document that can take on just about any form and any content. These documents are (by default) viewed in pdf format. In the case of CTFSAUG, the vignettes are these manual chapters.

Explore the links and get an understanding of how the help pages are interconnected.

From the R Console, *?topic* or *help(topic)* will bring up the html pages for the specific function that is named *topic*. If the name is not correctly typed you get:

```
> help(mort)
Error in help(mort) : No documentation for 'mort' in specified packages and
libraries:  you could try 'help.search("mort")'
```

If you don't know the name of the function, try the *help.search("topic")* form of help. R will provide a list of help pages that contain information on *topic*. Chose the ones you want and each will appear in a separate browser window.

For a topic specified by special characters, the special characters must be enclosed in double or single quotes. The use of double quotes makes the special characters a character string. Examples of such special characters are: "!", "%%", "&" and "[[". See the help pages of these special characters to learn about what they do. The use of double quotes is also necessary to find the help pages for words which have special syntactic meaning in R. For example syntactic words include: "if", "for", "while", "repeat" and "function". See the help pages of these words to learn what they do. For example:

```
> ?"[["
> help("[[")
> ?"if"
> help("if")
```

3.3 Interpreting Help Pages

Help pages, including the CTFS help pages, follow a general format that closely resembles that of UNIX man pages. The names of the function and of its package are written at the top of each help page. Each help page contains a brief description of the function's application, and is divided into sections. The most important sections are: Description, Usage, Arguments, Details, Value and Examples. The names and number of sections vary from help page to help page depending on the author of the help page. The help page of the CTFS function *growth()* will be used to illustrate how to read a help page.

The **Title** is a single sentence description of the function and is what appears in the index lists for function description.
Annual Growth Rates by Categories (User defined groups)

The **Description** section consists of a short paragraph which details an explanation of the function and its purpose.

Description

Computes annual growth rate for all trees or any user defined categorization of trees. Two growth rates can be computed: simple change in dbh over time and relative growth rate. Growth rates can also be evaluated for unrealistically high and low values and removed from the summary values. Growth rate in mm dbh per year, relative growth rate in % change in dbh per year, standard deviation or 95% confidence limits, and sample size are provided. The dataset must have at least 2 censuses to compute growth.

The **Usage** section contains basic format for invoking a function which is called the usage line. It consists of the function name followed by all of the function's arguments inside parenthesis. For example, a usage line would appear in the following format:

function name (argument 1, argument 2, argument 3, …)

Usage
growth(datafile, cens1 = "0", cens2 = "1", rounddown = FALSE, method = "I", stdev = FALSE, err.limit=4, maxgrowth=75, split1 = rep("all", dim(get(datafile))[1]), split2 = rep("all", dim(get(datafile))[1]))

Argument(s) are the variable(s) provided to a function when it is used either when typed into the R console by the user or when used by another function ("called" from inside another function). Arguments provide functions with the values that are needed to perform their purposes. Oftentimes, some of the arguments specified in the usage line are set equal to specific values or variables. These are referred to as "default" values. If no default values are provided, then the function must be used with a name for the variable without a default value

Because the arguments *cens1*, *cens2*, *rounddown*, *method*, *stdev*, *split1* and *split2* are all set equal to values in the usage line, the user can infer that these arguments have default values. The function *growth ()* must, as a minimum, be passed the name of a *datafile.*

Here is the detailed description of some of the arguments in *growth()*:

datafile: "name" of file (in quotes) must be a dataframe with trees as rows and contain at least 2 censuses

cens1: census number, value used to identify measurements from the first census for

computing growth such as dbh0.

cens2: census number, value used to identify measurements from the second census for computing growth such as dbh1.

rounddown: logical value in caps. When TRUE, if either of census is < 55, then the floor of the dbh value / 5 is provided. When FALSE, no change in the dbh is made.

The *datafile* must be provided and no defaults exist. *cens1* and *cens2* are optional because defaults are provided. The defaults are the values that these variables are set to in usage. In this case, "0" and "1". The same is true for *rounddown.* Read through the rest of the arguments. A user can define values for arguments with "=" or use the defaults, which ever is appropriate.

Look down at the bottom of the page for examples on how to use the function and its arguments. Note that these examples (should be) can be run in the R session if the appropriate datafiles are available.

1. Default use of growth()

> growth("tst.bci9095.full") -> growth.out

The **Details** section generally describes the process and computations that a function undergoes as it runs. For example, if the function is computing a complex mathematical equation this section would explain the mathematics performed by the function. Help Page authors often include additional miscellaneous information in this section.

The **Value** section explains the value that the function returns, i.e. the output of the function. Many functions return complex data and this section is crucial to interpreting the different aspects of that data. Although the user might be able to guess what these variable names represent, this section of the help page clearly explains what each name actually represents

In this example, *growth()* returns a list and the component of the list and their names are provided so the user can address them properly to view the values that *growth()* computes. The names on the list are specified by the $.

In the case of *growth()* some of the complexities of the *split1* and *split2* variables are explored. The variables appear to be very complicated in the usage statement. However, this is only because the function is computing default, effectively "empty" values for these variables unless the user provides otherwise. Here is the example of providing user defined values for an argument of some complexity.

The **See Also** section contains links to functions that are related to *growth ()*. Reading these help pages will further the user's understanding of this function.

The **Examples** section provides examples of how the user would call the function in the R console. For example, if the user has the CTFS package, appropriate datasets and objects loaded in the R environment, which can be copied from CTFS help pages and paste them unmodified into the R console. Here is the example for creating a vector to use for calculating mean growth rates for each species instead of the default of an overall mean growth rate for all trees.

```
> tst.bci9095.full$sp->spp.vct
> dim(tst.bci9095.full)
[1] 12421    14
> length(spp.vct)
[1] 12421
```

A vector of species names is made from the dataset such that the vector has a row for the name of each tree and in the same order as the trees in the dataset.

```
> growth.szclass("tst.bci9095.full",split1=spp.vct) -> growth.out
```

The function is run using the user created value of *split1*. Note that all the user does is set *split1 = spp.vct*, the new variable created. All the information provided by the function the default value for *split1* is producing is ignored and only the contents of *spp.vct* are used in the function.

Here is what the function returns. Note that it is, indeed, a list.

```
> growth.out
$rate
            all
alsebl 0.5776014
psycde 0.1742765
socrex 1.6349520

$N
       all
alsebl 7111
psycde   20
socrex  483

$clim
             all
alsebl 0.03320377
psycde 0.16768391
socrex 0.25021546

$dbhmean
```

all
alsebl 49.93489
psycde 13.55000
socrex 91.02070

3.4 Viewing Functions in R

The source code of functions that are currently loaded into any R environment can be viewed in R by typing the function's name without the (). The source code for functions not currently loaded into R can be viewed by opening the file in which the function is saved using a text editor; i.e. Note Pad The individual functions of a locally installed package can be viewed in *$R_HOME\library/R-ex*

3.5 Resource:

The complete documentation of *growth()*

growth {CTFSAUG}
R Documentation

Annual Growth Rates by Categories (User defined groups)

Description

Computes annual growth rate for all trees or any user defined categorization of trees. Two growth rates can be computed: simple change in dbh over time and relative growth rate. Growth rates can also be evaluated for unrealistically high and low values and removed from the summary values. Growth rate in mm dbh per year, relative growth rate in % change in dbh per year, standard deviation or 95% confidence limits, and sample size are provided. The dataset must have at least 2 censuses to compute growth.

Usage

```
growth(datafile, cens1 = ""0", cens2 = "1", rounddown = FALSE,
method = "I", stdev = FALSE, err.limit=4, maxgrowth=75, split1 = rep("all",
dim(get(datafile))[1]), split2 = rep("all", dim(get(datafile))[1]))
```

Arguments

datafile
"name" of file (in quotes) must be a dataframe with trees as rows and contain at least 2 censuses

cens1
census number, value used to identify measurements from the first census for computing growth such as dbh0.

cens2
census number, value used to identify measurements from the second census for computing growth such as dbh1.

rounddown
logical value in caps. When TRUE, if either of census is < 55, then the floor of the dbh value / 5 is provided. When FALSE, no change in the dbh is made.

method

character indicating the type of growth rate to be calculated. When "I" the annual change in dbh is provided. When "E" the relative growth rate is provided. see CTFS.growth for more details on method.

stdev
logical value in caps. When TRUE, the standard deviation of growth is provided. When FALSE, a 95% condfidence interval is provided. The confidence limits are not computed.

err.limit
number of standard deviations: used for determining if an individual tree growth rate is too high or low for inclusion.

maxgrow
maximum absolute growth rate, mm per year: used for determining if an individual tree growth rate is too high for inclusion.

split1
a vector of categorical values of the same length as datafile which groups trees into classes of interest for which growth values are computed. This vector can be composed of charcters or numbers.

split2
a second vector of categorical values of the same length as datafile which groups trees into classes of interest for which growth values are computed. This vector can be composed of charcters or numbers.

Details

See CTFS.growth for details on the computation methods of growth rates and associated functions.

Any two censuses on a datafile can be used. They do not have to be sequential, only that cens1 has to be before cens2.

The name of the input datafile datafile must be in quotes which is different from many other functions.

The name of the vector for split1 or split2 must NOT be in quotes.

If you have problems running this function, make sure you have entered the argument values for datafile, cens1, cens2. properly.

Value

growth returns a list of lists with the following named components. Values for each category of the split vectors are provided.

$rate
the annual growth rate in mm per year or % per year, type determined by the argument method

$N
the number of trees used to compute growth rate

$clim
the 95% confidence limit based on a normal distribution

$dbhmean
the mean dbh in mm for trees used to compute growth rate

If the vector(s) split1 and split2 are provided by the user, then growth rates and associated statistics are

computed for each value of the vectors. The vectors are nested so that growth rates is computed for each category of split2 within each category of split1

Author(s)

Pamela Hall, documentation: Rick Condit, function

See Also

growth, growth.dbh, growth.eachspp, growth.indiv, rndown5, trim.growth, split.habitat, split.grform, split.dbh CTFS.mortality, CTFS.abundance, CTFS.recruitment, CTFS.growth

Examples
Not run:
1. Default use of growth()

```
> growth("tst.bci9095.full") -> growth.out
> growth.out
$rate
        all
all 0.6436158

$N
    all
all 7614

$clim
         all
all 0.03529664

$dbhmean
        all
all 52.44563
```

2. Create a vector of species names for each tree to compute growth
rates for each species.

```
> tst.bci9095.full$sp->spp.vct
> dim(tst.bci9095.full)
[1] 12421    14
> length(spp.vct)
[1] 12421

> growth.szclass("tst.bci9095.full",split1=spp.vct) -> growth.out
> growth.out
$rate
                all
alsebl      0.5776014
psycde      0.1742765
socrex      1.6349520

$N
                all
alsebl      7111
psycde      20
socrex      483
```

$clim

	all
alsebl	0.03320377
psycde	0.16768391
socrex	0.25021546

$dbhmean

	all
alsebl	49.93489
psycde	13.55000
socrex	91.02070

3. Create 2 vectors: the first of habitats based on the quad location
of each tree and the second of tree species names.

```
> split.habitat(tst.bci9095.full,bci.quad.info,by.col="hab")->habitat.vct
> length(habitat.vct)
[1] 12421
> tst.bci9095.full$sp->spp.vct
> length(spp.vct)
[1] 12421
> growth("tst.bci9095.full",split1=spp.vct,split2=habitat.vct)->growth.spp.hab.out
> growth.spp.hab.out[1]
```

$rate

	1	2	3	4	5	6	7	8
alsebl	0.4952	0.4877	0.6492	0.5578	0.6279	0.5222	0.6118	0.5503
psycde	0.0000	NA	0.7222	-0.2095	0.2067	NA	0.2019	0.0353
socrex	1.2016	1.7317	1.4579	3.38964	1.5580	1.8466	1.6783	2.0268

```
## End(Not run)
```

[Package Contents]

4.0 File organization, Data, Functions and Results

CTFS colleagues working with R generate many large datasets, graphs and tables. A lack of organization for this data leads to wasted memory, storage space and time as well as misplaced and lost files. Although there is not only one correct way to organize CTFS R files, we recommend here a system which will help ensure some order in the file location and an ability to clean up files after an extensive analysis without fear of losing valuable information.

4.1 Basic File Organization of R

The first step to organizing personal R files is to develop an understanding of where general R packages, functions, manuals and documentation are located.

The R directory is the directory into which all of the R files are placed upon installation, i.e. $R_HOME, contains 10 sub-folders with various files. The actual R program file, Rgui.exe, is located in $R_HOME. The file *.rData* which is created and modified when the user saves their workspace is also located in $R_HOME. In addition to the files needed to run R, $R_HOME also contains help documentation as well as various other files.

$R_HOME's sub-folder "library", contains all of the R packages installed on a given computer. Each installed package is contained within its own folder. As explained in Chapter 1, a package is a bundle of functions, documentation, and datasets. Upon installation, R includes more than 30 packages each with its own folder in the library folder. Each package folder contains a sub-folder called "R." This folder contains all of functions specific to a particular package. Oftentimes, all of a package's functions are contained within a single file. Most package folders also contain a sub-folder called "html" in which all of the package's html help pages are stored. These are the html help pages that the user can access using the R help facility described in chapter 3. The user can also directly open these files by selecting the files themselves. This will open a Windows Explorer window outside of R.

In the MacOS the file structure is substantially different and depends upon whether R was installed for a single or multiple users. In general, it is not necessary to know where R is actually installed. The object is now where the user's files are kept.

When R is installed for multiple users it will appear in a folder *sw* in the root directory. There are many layers of folders within this that contain programs, libraries, etc that R is dependent upon. R is usually located quite a few layers down in the *R.framesworks* folder. Multiple versions of R can be maintained. Further down is the *library* folder that contains the packages installed with a given version of R. Do not move nor alter any of these folders or files.

The "home" directory for R in MacOSX is not a useful directory to know. The "home" directory of a user is useful as it is where all the user's files are kept and where the files used by R, created by the user and locally installed packages is located. "Local" installation means that the package was only installed for a single user. Packages can

also be installed for all users but this involves using the Unix terminal and won't be covered here.

A locally installed R package (like the future CTFS package) appears in ~user/Library/R/library. ("~user" refers to the home directory of a given user). The installed package contains numerous folders. Explore them and see where the R functions are kept, the html pages, manual chapters and the source files for help pages and functions. The help pages and manual chapters can be directly accessed from here or used during an R session. The latter is much easier to do since this is what the software was intended for.

4.2 CTFS Specific Files

During a CTFS Analytical Workshops, functions, datasets, tables and output from analyses are created. Keeping track of these is critical to proper interpretation of results and learning how to use CTFS functions. It is important to be able to separate work in progress from more finished work that needs to be preserved. CLEAN UP! Keep the number of "temp" files down by frequently reviewing them and getting rid of them. If you use "tmp" or "tst" in the file name, then it will be much easier to find them in the directories.

We highly recommend that users name their datasets, functions and results files clearly and with consistency. Use a code to indicate the type of information in the file. "res" for results from a function, "mort.res" for results from a mortality analysis. Put a date in the file name to indicate when it was created. Don't rely on the date of the file as opening under some circumstances can change the file date.

Consider keeping a log of files created and how they were obtained. This can be done in the "comments" section in file names in the MacOSX.

As a suggestion: Create a directory system that is specific to the 2004 workshop, the functions that will be distributed, functions that you will be writing, results you will be generating, and non-R files that are part of a manuscript. We suggest a folder labeled "CTFS 2004". Within this folder consider the following folders as "topics".

Keep the FDP permanent datasets in a "remote" location from the directories where functions are being written and results being generated so that the datasets will not accidently get overwritten.

"Functions2004": This folder should contain at least three subfolders: that differentiate between the functions provided at the beginning of the workshop, functions that are created during the workshop and functions that are "finalized" during the workshop. For instance: **Initial, Development, Final** may serve.

The **Initial** folder should include functions distributed at the beginning of the workshop and could include documentation. When the user is creates a new version of a function

which already exists in this folder the user should delete the old version if it is no longer in use.

The user may also wish to have functions from previous workshop. These can be kept in folders with the workshop year in the title eg. **"Functions2003",** etc. Be very careful with redundant function names. Functions from previous years that have been revised are best removed if they are useless or have their name changed to indicate the year of their creation.

When a CTFS Package is created, the proliferation of functions and documentation will be considerably eased as the **Initial** version of each function will be kept in the package's folder in the R library. These functions will be available for revision, but not necessarily for immediate incorporation into the package.

"Results": This folder is meant to contain all of the results of the CTFS user's work. This folder should be further divided by topics of analysis, for example, mortality, growth, or neighborhood analysis. Each of these topic subfolders should contain all of the user's results pertaining to that topic, for instance, datasets (output from functions), graphs, tables, papers, etc. Unlike the other two folders, "Results" could contain files that aren't directly associated with R.

Here is an example of a directory structure. The file names are in italics. Note that not all folders have been filled in with file names.

- CTFS Datasets
  - Final Versions
    - Lambir
    - BCI
      - *bci9095.full.rdata*
      - *bci9095.spp.rdata*
      - *bci95.mult.rdata*
      - *bci.spp.info.rdata*
      - *bci.quad.info.rdata*
    - Yasuni
  - Working Versions

- CTFS2004
  - Functions
    - Initial
      - Functions
      - HTML
    - Development
    - Final
  - Results
    - Habitat_Dynamics
      - Mortality
        - *Mort.spp.res*
        - *Mort.spp.smalltree50.res*
        - *mort.habclass.res*
        - *Mort.habclass.smalltree50.res*
      - Growth
      - Graphs
        - *Hab8.plot*
        - *Hab6.plot*
        - *Specialists.plot*
      - Tables
        - *Species by Habitat6*
        - *Species by Habitat6 smalltrees*
        - *Specialists by Habitat6*
      - Summary
        - *Methods1.hab defined*
        - *Methods2.mort&growth rules*
        - *Synopsis specialists by hab6*
    - Density_Dependence
    - Similar_Species

6.0 Reading and Writing Datafiles

Data can be entered using the keyboard (short datasets or test files) or by use of read and write data functions. To use the read and write functions in R, a database created in data management software must be converted to a tab delimited text file without quotes. Include the column headings (variable names) as the first line of the file. Use database and spreadsheet software that provide this option, either as part of the "Save As" or "Export".

6.1 Reading Datafiles

There are several ways to read large text files into R datasets. In addition to the information provided here, please read the R help pages for each function. The named files to be read into R must be located by an explicit **path** name or be in the working directory of the current R session. Use *setwd()* to make the working directory the one containing the data file to be read in. The following examples assume that the working directory for the R session is the directory in which the data files are located. If not, be sure to use the entire path in quotes when specifying the file name.

scan()
reads data from a text file into a vector or a list of vectors directly from the console. The general syntax of scan() is:

>scan(file , what , sep , quote , skip)

file : name of the input file to read data from. The path should be included if it is not the default path, i.e. the working directory. If the file is specified as """, then input is taken from the keyboard.
what : specifies the type of data to be read: logical, integer, numeric, character, or list.
sep : specifies what the field separator is, which by default, is a blank space between 2 fields as known as "white space". Other common separators are tab ("\t") and commas (",")
quote : specifies the set of quoting characters used to define where one character string begins and ends.
skip : designates the number of lines of the input file to skip before beginning to read the data values, as in the case of column names, which you would skip.

Scan returns a list of vectors with the types given by the types of the elements in *'what'*. This provides a way of reading columnar data. For example:

```
scancensdata=function(alsebl.txt)
 {
 data.read=scan(censdata,skip=1,what=list("","","","","","",""))
  tag=as.numeric(data.read[[1]])
  dbh=as.numeric(data.read[[2]])
  status=data.read[[3]]
```

```
 pom=as.numeric(data.read[[4]])
 stems=as.numeric(data.read[[5]])
 date=as.numeric(data.read[[6]])
 codes=data.read[[7]]

return(data.frame(tag=tag,dbh=dbh,status=status,pom=pom,stems=stems,
date=date,codes=codes))
 }
```

This function reads in your text *datafile*, *censdata*, containing the fields *tag, dbh, status, pom, number of stems, date,* and *codes*. It skips the first row containing the names of the columns, and specifies that the data be read in as a list of character vectors. The vectors are converted to numeric when applicable and a data.frame is returned as output. Note that the column names as provided in the text file are not included in the output except with the explicit inclusion of the column names in the *data.frame* statement.

Here is a datafile called *alsebl.txt* that can be made into a dataset accessible by R that can be read in using *scan().* This file is white space delimited.

tag	dbh	status	pom	stems	date	codes
000047	426	A	3	1	5382	B
000049	228	A	1	1	5396	*
000068	277	A	1	1	5390	*
000071	318	A	1	1	5390	*
000073	368	A	2	1	5382	B
000089	580	A	2	1	5382	B
000092	-1	D	0	-1	5387	*
000109	318	A	2	1	5382	B
000122	351	A	2	1	5377	B
000138	246	A	2	1	5377	B
000169	404	A	2	1	5377	B
000172	308	A	2	1	5377	BQ

Using the above function, *scanscensdata()* to create a dataframe Rdata file.

```
> alsebl=scancensdata("alsebl.txt")
> alsebl[1:5,]

  tag dbh status pom stems date codes
1  47 426      A   3     1  5382      B
2  49 228      A   1     1  5396      *
3  68 277      A   1     1  5390      *
4  71 318      A   1     1  5390      *
5  73 368      A   2     1  5382      B

> str(alsebl)
```

```
`data.frame':     11128 obs. of  14 variables:
 $ tag    : num  -27784     47     49     68     71 ...
 $ sp     :Class 'AsIs'  chr [1:11128] "alsebl" "alsebl" "alsebl" "alsebl" ...
 $ gx     : num    -9  984  985  986 1000 ...
 $ gy     : num   -9 342 329 276 278 ...
 $ dbh0   : num  NA 437 228 278 269 360 580 NA 311 348 ...
 $ dbh1   : num  NA 426 228 277 318 368 580 NA 318 351 ...
 $ pom0   : num  0 2 1 1 1 2 2 0 2 2 ...
 $ pom1   : num  0 3 1 1 1 2 2 0 2 2 ...
 $ date0  : num     0 3702 3632 3627 3627 ...
 $ date1  : num     0 5382 5396 5390 5390 ...
 $ codes0 :Class 'AsIs'  chr [1:11128] "*" "B" "*" "*" ...
 $ codes1 :Class 'AsIs'  chr [1:11128] "*" "B" "*" "*" ...
 $ status0:Class 'AsIs'  chr [1:11128] NA "A" "A" "A" ...
 $ status1:Class 'AsIs'  chr [1:11128] NA "A" "A" "A" ...
```

Notice that the fields *tag, dbh, pom, stems,* and *date* are numeric, while *status* and *codes* are character fields. The left most numbers on each line are row numbers and serve as row names once the data frame is created.

read.table()

reads a file in table format and outputs a data frame. This be accomplished with a single line command line making it much easier to use than *scan()*. This function is appropriate for the majority of text datasets that have been created by a spread sheet. However, *read.table()* is an inefficient way to read in very large numerical files, especially those with many columns. The *scan()* function is faster and takes up less memory. In fact *read.table()* actually uses scan to read the file, and then processes the results of *scan*().The difference between these two functions includes the format the variables in the output file take on. By default in *read.table()*, numeric fields are read in as numeric variables and character fields are read in as factors. The general syntax of read.table() is:

>*read.table(file, header, sep, quote, dec, row.names, col.names, as.is, skip)*

file : name of the input file to read data from.

header : a logical value indicating whether the input file contains the names of the variables as its first line.

sep : specifies what the field separator is, which by default, is a blank space between 2 fields as known as "white space". Other common separators are tab ("\t") and commas (",")

quote : specifies the set of quoting characters used to define where one character string begins and ends.

dec : designates the character used for decimal points. In the CTFS datasets the period is used (".").

row.names : a vector of names, or a single number giving the column of the table which contains the names of the rows. The default is sequentially numbered rows.

col.names : a vector of names for the variables. The default is a "'V'" followed by the column number eg V1 V2 V3….

as.is : a logical value. FALSE defines the default behavior, which is to convert the character variables to factors. Designate *as.is* = TRUE to suppress conversion of character variables to factors, leaving them "as is".

skip : designates the number of lines of the input file to skip before beginning to read the data values, as in the case of column names, which you would skip.

Here is a text file called *bci.spp.info.txt* that can be made into a dataset accessible by R and can be read in using *read.table()* . The file is tab delimited text, with no quoting characters and contains a header row with the variable names. The character variables are to remain as characters and are not to be converted to factors.

Example:

sp	genus	species	family	grform	repsize	breedsys	maxht
acacme	Acacia	melanoceras	Fabaceae:Mimos.	U	4	B	6
acaldi	Acalypha	diversifolia	Euphorbiaceae	S	2	M	6
acalma	Acalypha	macrostachya	Euphorbiaceae	U	2	M	5
ade1tr	Adelia	triloba	Euphorbiaceae	U	10	D	5

>bcispp=read.table(file="/datasets/bci/spplist.txt", as.is=T, header=T, sep="\t", quote="")

6.2 Writing Datafiles

Once an R dataset has been created, you may want to continue using it in future analyses. If you quit an R session without saving the workspace or saving specific objects, you will lose all objects created in the current session. Instead of recreating the R datasets every time an R session begins, you will want to save them and be able to call them up in a later session.

The best way to accomplish this is to save the created file in your directory structure so that it exists when you exit R.

Alternatively, especially when in a hurry, save the current workspace and loading it again in the next session and the datafile will be available. Remember, that until the file is saved into your directory structure, it does not exist permanently on your disk

save()

saves an R object to a specified file on disk which can be called up at a later date using the *load()* or *attach().* The general syntax is:

> *save(…., file)*

…. : a list of the names of the objects to be saved.
file : name of the file on disk, in quotes with the suffix *.rdata*. By default the file is saves in the working directory. Use *setwd()* to change the directory or include the full path of the folder where you want to save the file.

Save the R file *alsebl* created above in the working directory
> *save(alsebl,file="alsebl.rdata")*

Or in an explicitly provided path /R/results/ folder:

> *save(alsebl, file="/R/results/alsebl.rdata")*

A binary file called *alsebl.rdata* will be created in the /R/results/ folder, that is portable across all R platforms (Windows, MacOS, Linux). The *name* of the file for the R session will be the file name used when it was created in R. The saved *name* is the name of the file on disk which always includes the *.rdata* suffix. These names DO NOT have to be the same, though it greatly eases remembering what each file contains (see Chapter 4 on file management).

load()

To call up this same file in the next R working session use:

> *load("/R/results/alsebl.rdata")*

or if the file is in the current working directory:

> *load("alsebl.rdata")*

The R file "alsebl" will be available in the first environment. You can also use the "Load Workspace" option under "File" in the menu bar to load the file.

attach()

this effectively does the same thing as *load()* but the file is placed in a different location within the R session. It is places in environment #2 also referred to as R search path #2.

> *attach("/R/results/alsebl.rdata")*

By default, the R file "alsebl" will be available in the second environment (*ls(2)*). Once attached or loaded, the variables in *alsebl* can be accessed by giving their names alone.

write.table()

This function is used to export a data frame to a text file for use in other software, such as MS Excel or Word. Remember, R datasets are binary files and are not readable by other software without conversation to a text file. If the R dataset to export is not a data frame, this function will try to convert it to a data frame first. The general syntax is:

>write.table(x, file = "", append = FALSE, quote = TRUE, sep = " ",
na = "NA", dec = ".", row.names = TRUE, col.names = TRUE)

x : name of the data frame to be exported. If *x* is not a data frame, the function will try to convert it to a data frame.

file : the name of the file (in quotes) where the data frame is to be saved. The path should be included if you want to save the file in a folder that is not the default.

append : a logical value which specifies what to do if *file* already exists. If 'TRUE', the output is appended to the existing file, if 'FALSE', the file is overwritten, destroying all previous data.

quote : a logical value or a numeric vector. If 'TRUE', all character and factor variables will be surrounded by double quotes. If 'FALSE', nothing is quoted. If it is a numeric vector, its elements are taken as the indices of the columns to quote.

sep : the separator to use to separate the field values within each row. Common field separators are blank spaces, commas, tabs, or semicolons.

na : the string to use for missing values in the data. By default, it is assigned the string "NA", but you may assign it any other character.

dec : the character to use for the decimal point. The most common characters used are "." or ",".

row.names : either a logical value indicating whether the row names of the data frame are to be written, or it can be assigned a character vector of row names to be written.

col.names : either a logical value indicating whether the column names of the data frame are to be used, or it can be assigned a character vector of column names to be written.

To export an R table called *bci.mort.spp* as a text file to the /bci/mortality/ folder use:

> write.table(bci.mort.spp, file="/bci/mortality/spp.mort.txt", quote=F, sep="\t")

This creates a tab-delimited text file, with no quotes surrounding the character variables, called *spp.mort.txt*. This text file can be called up in MS Word, MS Excel, or any other database management software.

6.3 Creating CTFS Datasets

6.3.1 Preparing Text Datasets

Primary Files

In previous workshops, the datasets at each site were set up as separate species text files under several folders:

- In the \census folder, the species files contained the location information for each tree: tag, gx, gy.
- In the \census0 folder, the species files contained the census information from the first census: tag, dbh (diameter at breast height), status code, pom (point of measure), number of stems, date of measurement (Julian dates), and other site-specific codes.
- Sites with recensus data had folders called \census1, \census2, and so on with measurement data from each recensus.

Currently, however, instead of separate species files, the datasets should be set up as one large text file for each census with all the trees together, in order of species. These files will be referred to as the full text datasets, named using the following pattern:

Siteyear(s).full.rdata

For example, the BCI 1990 enumeration is named: *bci90.full.rdata*
The BCI 1990 and 1995 merged enumeration file is named: *bci9095.full.rdata*
The BCI 1985 to 1995 (3 censuses) file is named: *bci85to90.full.rdata*

The files for each of the census should include the same exact number of records and should be in exactly the same tree order. This means that as trees die they are NOT removed from a census dataset. When trees are recruited, they are added into ALL previous censuses.

FORMAT

The columns in these full text datasets include:

tag : the number of the tree. Each record of this file should refer to one tree. All multiple stem measurements other than the main stem should be moved to the multiple stem file.
sp : the 4 to 6-letter species code. Every species code should appear in the species file described below.
gx, gy : the x and y coordinates within the plot.
dbh : the diameter measurement from the current census.
pom : point of measure of the diameter.
date : date of measurement in Julian dates, i.e. number of days since a fixed date (we suggest 1/1/80 when BCI, the first plot, was established).
codes : codes that each site wants to keep for later analyses or explanations.

Following are more detailed explanations of these variables.

tag : Each record in the dataset refers to one individual, identified by the tag number. There should be no duplicate tag numbers. Tag numbers should always be in identical and numerical order in all datasets.

sp : All species codes should be valid and included in the species dataset described below. There shouldn't be any extra species codes either in these datasets or in the species list. All valid morphospecies should appear in the species list.

gx, gy : All gx and gy coordinates should fall within the range of the coordinates of your plot. If a location is unknown, it should be given a code of –9. If any coordinate falls exactly on the rightmost or uppermost border, change it to 1 decimal less. For example, for a plot 1000 x 500 m, make sure that your gx coordinates go from 0 to 999.9 and your gy coordinates from 0 to 499.9.

dbh : Dbh should be above 10 mm in all cases where the tree is alive. For multiple-stemmed plants, the largest dbh should appear in this dataset, the rest of the measurements should appear in the multiple stem file (see below). The following dbh codes should be used otherwise:

0 - if the tree is alive, but its main stem broke off and its dbh is below 1 cm, or its stem is under 1.3 m. This is used in the case of resprouts also.
–1 - if the tree died in a later census.
–2 - if the tree had not entered the census yet but will in a later census.
–9 - if the dbh measurement was missed and is unknown.
5 - For Pasoh only, a dbh of 5 refers to trees that were alive and were >=10 but were not measured.

pom : Pom refers to the point of measurement of the diameter. Pom is used when we calculate growth rates. All trees ≥1 cm dbh should be given a pom code of 1 the first time they enter a census, and in all subsequent censuses if the height at which the dbh was measured does not change. If the diameter of the tree is measured at a different height or at another stem (as in the case of multiple-stemmed trees) in a subsequent census, the pom should be increased to the next number. This may happen in trees with buttresses where the height at which the diameter was measured changed, in trees whose main stem broke off or died and whose largest stem is another one, or in trees whose main stem died but have resprouts. Trees that have not entered the census yet, trees whose dbh is missing, or trees that have died should all be given a pom of 0 (zero).

date : Dates refer to the date when the individual was measured. They should be julian dates, calculated from the number of days since January 1st,1980 (the day the first census at BCI began). Make sure the dates are all in the correct format before converting them to julian dates. Some countries specify the month before the day, while others specify the day before the month. Make sure it is consistent throughout the dataset. Check that the dates fall within the range of the census interval. It is very common to find the wrong year entered.

codes : Codes refer to any codes you have in your database that you want to keep and may want to use in your analyses. They should not have spaces between them. Put in underscores, periods, or any other character if you want to separate the codes.

ERROR CHECKING

Following are other inconsistencies to check in those sites with recensus data.

Number and order of records:

Make sure all tag numbers are in exactly the same order in each of your full datasets. There should be the same number of records in all your census datasets.

Screen the dbh remeasurements:

You should recheck all dbhs that are smaller (allowing for some shrinkage or slight measurement error) or abnormally larger than the previous census. A dbh may be smaller than the previous census if the main stem broke off or died, or if the height at which the measurement was taken changed. If so, make sure that it is annotated and increase the pom to the next larger number.

If you do find an error in a previous dbh measurement, do not change the previous measurement in your main database, since you will be estimating. Keep the measurements, make a note of the problem, and let each scientist/analyst make their own decision on what to do with the error.

For plots with 3 or more censuses, verify that all **dead** trees remain dead in the later censuses. If a tree was found alive in a later census, the tag number may be incorrect or you may have to change the previous "dead" code to "alive" and change the dbh to –9 (missing).

For plots with 3 or more censuses, verify that all **recruit** trees are indicated as to be recruits in earlier censuses.

SUPPORT FILES

Besides the full text datasets, each site should also prepare the following files.

Species information file: site.spp.info.txt

This file contains the information for each species found at each site. Each species is a row and the columns are the information about each species. Every single species code that appears in your full datasets should appear in this list (including all morphospecies)

and every species code that appears in this species list should appear in your full datasets. The file should have the following columns:

sp : the four to six-letter species code,
genus : the genus name
species : the species name
family : the family the species belongs to
grform : the growth form of the species, i.e. shrub, understory tree, mid-sized tree, canopy tree.
repsize : the reproductive dbh (cm) of the species
breedsys : breeding system, i.e. bisexual, dioecious, monoecious, polygamous, etc.
maxht : maximum height (m) attained by the species.

All columns should have something filled in for every record and there should be no spaces within each column. If your species name consists of 2 or more words, for example, make sure to insert an underscore, dash or period in the space between the words.

The first 5 columns should be filled in as best as possible. No columns should be left blank. If a species has not been identified yet, put "Unidentified" or "Unknown" in the relevant columns (*family, genus, species*). In the cases where information is missing, put in a –1 or –9 in the columns for numeric fields (*repsize, maxht*) and a '*' in the character fields (*grform, breedsys*).

Any other information that is identified with a species can be placed into this file. For instance, *timber type, pioneer status* etc. Classification of species that is derived from analyses can also be kept here such as the *degree of habitat specificity* that can be determined by using the torus habitat analysis. These classifications of species can be used in further analyses.

An rdata file should be made from this text file and saved in the same directory that the site rdata census files are kept. See *read.table()* and *save()* functions above. Save the file as *site.spp.info.rdata.* eg. *bci.spp.info.rdata.*

Quadrate information file: site.quad.info.txt

This file contains information about each 20 by 20 m quadrate. Each quadrate is a row and the columns are the information about each quadrate. There must be a row for each quadrate. There are 1250 rows for a standard 1000 by 500 m CTFS plot. At a minimum this file should contain the elevation data (m) for each 5 meter interval of the plot in order of x and y. In addition, slope and convexity can be added for use in habitat analysis. The elevation file should be tab-delimited. The columns are:

x : x coordinate of tree (east-west axis of plot)
y : y coordinate of tree (north-south axis of plot)
elev : elevation in m a.s.l.

slope : degrees (see “torus” analysis for computation)
convex : degrees (see “torus” analysis for computation)

The results of any form of habitat classification can also be put into this file. For instance, the “torus” habitat analysis can be used to provide 8 *classes of habitat* and these are assigned to each quadrate.

An rdata file should be made from this text file and saved in the same directory that the site rdata census files are kept. See *read.table()* and *save()* functions above. Save the file as *site.quad.info.rdata.* eg. *bci.quad.info.rdata.*

Multiple stem file: siteyear.mult.txt

There should be one multiple stem file for each census. Each file should include at least a tag and dbh column. Each multiple stem file includes all stem measurements of all multiple-stemmed individuals for that census, **excluding** the measurement of the largest main stem, which is in the full database. The multiple stem files do not include the largest stem measurement of each tree. The multiple stem files from each census will not necessarily have the same number of records nor the same tag numbers, unlike the full datasets for each census. At this time the multiple stem files are named: *mult0* (first census), *mult1* (second census), etc.

An rdata file should be made from this text file and saved in the same directory that the site rdata census files are kept. See *read.table()* and *save()* functions above. To be consistent with the naming of other files, save the file as *siteyear.mult.rdata.* eg. *bci95.mult.rdata.*

6.3.1 Preparing R Datasets

Once your text datasets are cleaned up, checked for errors and inconsistencies, and in the correct format, make them into rdata files.

Once your tab-delimited text file (with no quotes) is ready with the appropriate column headings, create the full dataset in R with the following command:

> *> siteyear..full=read.table(file=”FILENAME”, as.is=T, header=T, sep="\t", quote="")*

where *site* refers to the site (bci, yasuni, sinharaja, hkk, etc.), *year* refers to the census year, and *FILENAME* refers to the name of your tab-delimited text file. Remember to include the path in the *FILENAME* if your file is not found in working directory.

Similarly, to create a tab-delimited species text file with no quotes to an R species dataset, use the following command:

> *> site.spp.info=read.table(file=”FILENAME”, as.is=T, header=T, sep="\t", quote="")*

To create an R multiple stem dataset:

> *siteyear.mult0= read.table(file="mult0.txt", as.is=T, header=T, sep="\t", quote="")*

To create the R elevation file:

> *site.quad.info=read.table(file="FILENAME", as.is.=T, header=T, sep="\t", quote="")*

You may then use this file to create an elevation matrix with the CTFS R function *readelevdata().*

Split datasets by species

After the R full datasets are created, you can create a list of dataframes separated by species with the CTFS R function called *split.fulldata().*

> *siteyear.spp=split.fulldata(PLOT0.full)*

This contains the same fields as the full dataset, but separated by species. The file *siteyear.spp* is a list of dataframes, one for each species.

To extract a single species' data frame from *siteyear.spp*, use the CTFS R function *load.species().*

Merged datasets from two censuses

To run the growth, mortality, and recruitment functions, you need to merge the full datasets from at least two censuses, using the CTFS R function *merge.census().* Following is an example using the 90 and 95 BCI censuses.

> *bci9095.full=merge.census("bci", census0=bci90.full, census1=bci95.full)*

Again, it is very important that the full datasets from the two censuses be in the exact same tag order, since the program just merges the two datasets, it does not verify that the tag numbers match before merging.

After merging the two datasets, you can create the split files on this merged dataset with the split.fulldata() function mentioned above:

> *bci9095.spp=split.data(bci9095.full)*

Save all these R datasets to a folder on your computer with the *save()* function described above.