Center for Tropical Forest Science R Package Manual
Pamela Hall, Suzanne Lao, Ellen Connell and Marie Massa
Version 1.00 March 29, 2006

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. Many contributed packages create new objects returned by their functions. This makes the results of complex functions easier to deal with as a single object type contains all of their output.

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 dealt with further.

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 is a two-dimensional object. 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, in deed, 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. The index [] can be added to address a specific row value in that column. The $ form cannot used for row names.

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

> example.dataframe$Uno[3]
[1] 3

> example.dataframe$Deux
NULL
```

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 here because all of the

other dataframes 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,]*") refer 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, but it build upon the same principles as described above.