Getting Started with R

James Holland Jones *
Department of Anthropological Sciences
Stanford University

July 13, 2005
v 0.95

1 A Brief Introduction to R

In this summer workshop we will use the R statistical computing language to facilitate teaching the concepts and methods of formal demography. R provides a convenient environment for manipulating lists of numbers (called vectors) and arrays of numbers (called matrices). While there are many more common applications that will allow you to manipulate lists of numbers (e.g., spreadsheet programs), R also allows for the easy calculation of a number of quantities that are of vital interest to population biologists. R also provides a powerful environment for performing numerical simulations, an important tool in the population biologist's arsenal.

What R lacks in apparent user-friendliness, it more than makes up for in power. While there is a learning curve associated with developing R skills, this is really true of any software package that you will use. Once you acquire the basics, you will find that R is logical and simple.

A couple questions naturally arise: (1) What is R? and (2) Why use it instead of, say, a spreadsheet application which is more typically used in introductory demography courses?

1.1 What is R?

- R is numerical software
- R is a "dialect" of the S statistical programming language
- R is free
- R is state-of-the-art in statistical computing. It is what many (most?) research statisticians use in their work

*Correspondence Address: Department of Anthropological Sciences, Building 360, Stanford, CA 94305-2117; phone: 650-723-4824, fax: 650-725-9996; email: jhj1@stanford.edu

1.2 Why Use R?

R has a number attractive features that recommend it for a course such as this.

- R is FREE! That, by itself, is almost enough. No complicated licensing. Broad dissemination of research methodologies and results, etc.
- R is available for a variety of computer platforms (e.g., Linux, MacOS, Windows).
- R is widely used by professional statisticians, biologists, demographers, and other scientists. This increases the likelihood that code will exist to do a calculation you might want to do.
- R has remarkable online help lists, tutorials, etc.
- R represents the state-of-the-art in statistical computing.

1.3 Why Not a Spreadsheet?

- While a spreadsheet is handy for doing many demographic calculations, it is not ideal for many of the problems we will be tackling.
- For example, for a variety of problems in formal demography and population biology, one must calculate an eigenvalue. This is a simple task in an environment such as R or MATLAB, but not possible in most common spreadsheet applications.
- For some applications, we will be numerically solving ordinary differential equations. R has a package to do this, spreadsheet applications lack such facilities.

2 Obtaining R

In this section, I am assuming that you would like to load R onto your laptop computer. We will have a computer lab with R-loaded machines available for the course, but assume that many people will want to use their own machines.

The first step toward successfully using R is to obtain the software. R is freely available and has been pre-compiled for a variety of platforms.

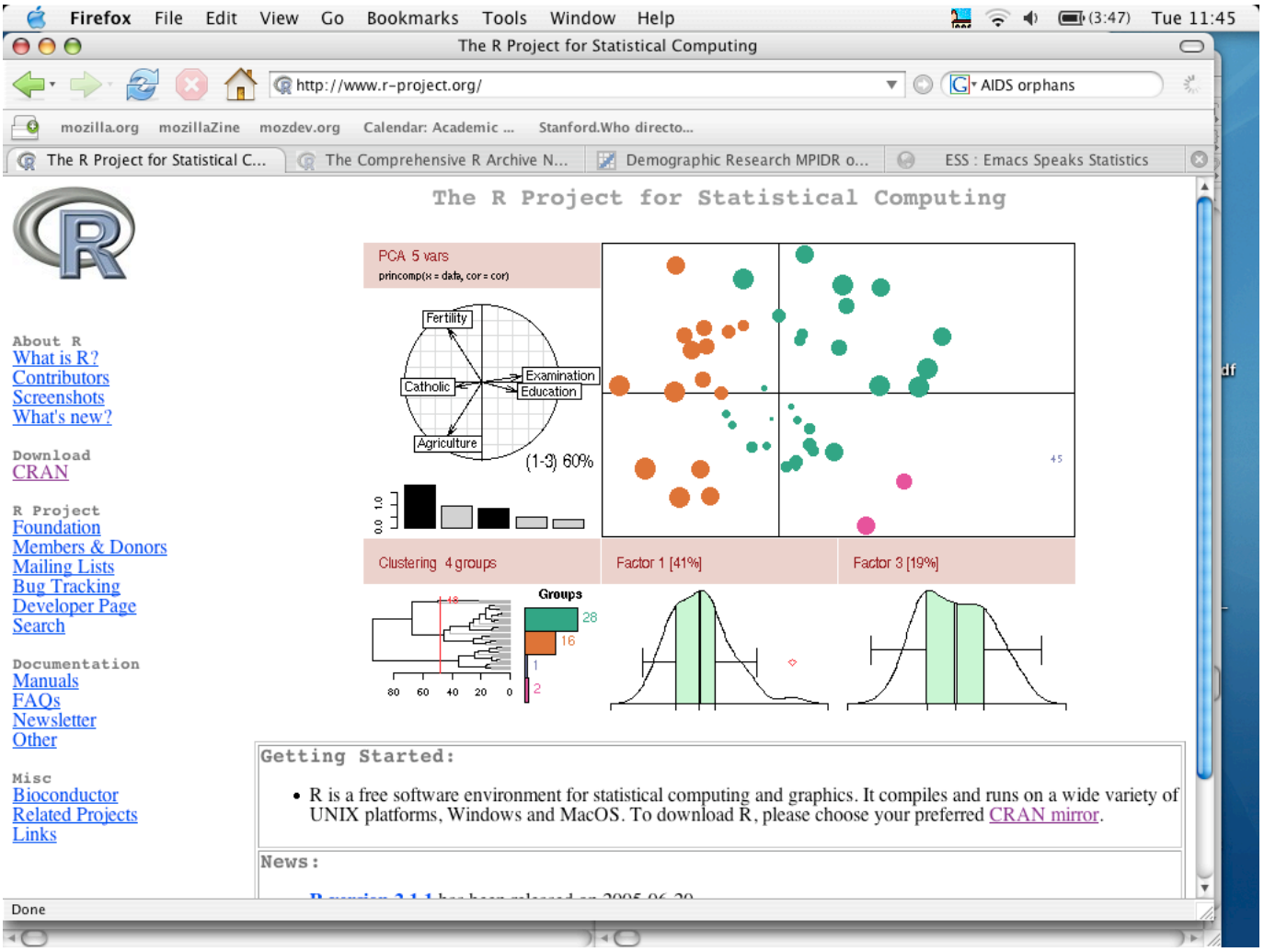

Figure 1: The R-Project home page.

Go to the R-Project home page (figure 1). The url is: `http://www.r-project.org`. From here, you can find a wealth of information on R, including extensive online documentation. Follow the link for CRAN (which stands for "The Comprehensive R Archive Network"). You will be given a list of mirrors from which to choose. I use the Berkeley mirror (figure 2), but it doesn't really matter which you use – probably one that's is geographically close to you is best. Here is the url for the Berkeley mirror: `http://cran.cnr.berkeley.edu/`.

Download the pre-compiled binary that is appropriate for your operating system and follow the instructions provided.

Aside for Geeks and Geek-Wannabes If you happen to already be an Emacs user, I recommend using ESS (Emacs Speaks Statistics), which allows you to run R through Emacs and thereby capitalize on all its editing functionality, syntax coloration, etc. For information on

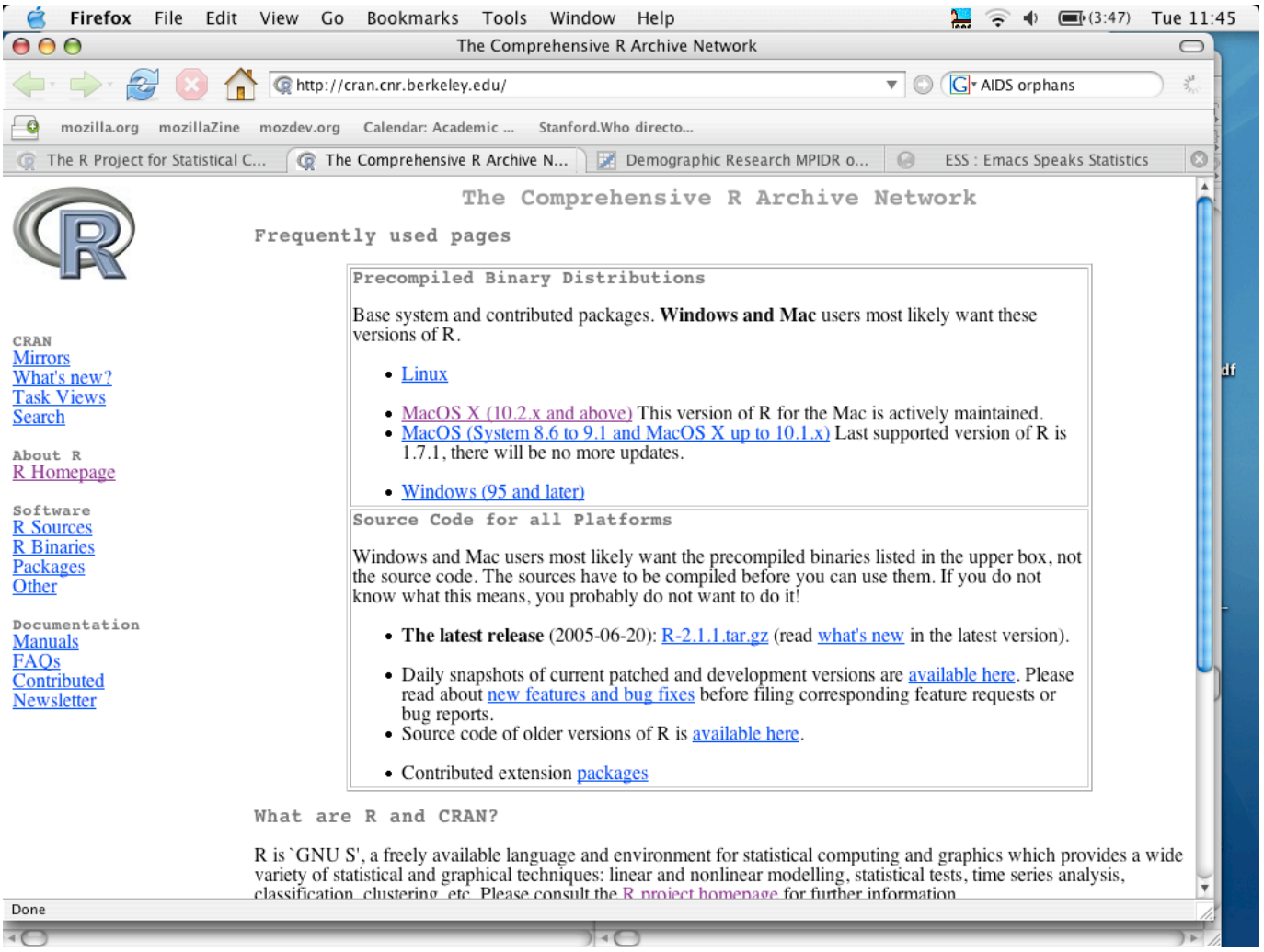

Figure 2: Screenshot of CRAN (Berkeley mirror).

using ESS, see `http://ess.r-project.org/`.

3 Starting R

Start R. Regardless of your platform, you will get a message that looks something like this:

```
R : Copyright 2005, The R Foundation for Statistical Computing
Version 2.1.1  (2005-06-20), ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for a HTML browser interface to help.
Type 'q()' to quit R.

>
```

Take it for a spin. Try the `demo()` option.

4 Some Notes to Help Get You Started

These are some notes that I put together for my class, "Human Population Biology," offered in the Department of Anthropological Sciences at Stanford. It has a decidedly evolutionary-anthropology flavor, but it should help get you started thinking about doing demographic work in R.

4.1 Obtaining Add-On Packages

New packages are created for R all the time. We will use a number of packages that, depending on your operating system, may or may not have come with the pre-compiled binary that you installed. Fortunately, it's easy to check and just as easy to add packages. First check to see what packages you have:

```
> library()
Packages in library '/sw/lib/R/library':

KernSmooth              Functions for kernel smoothing for Wand &
                        Jones (1995)
MASS                    Main Library of Venables and Ripley's MASS
```

```
base                    The R base package
boot                    Bootstrap R (S-Plus) Functions (Canty)
class                   Functions for classification
cluster                 Functions for clustering (by Rousseeuw et al.)
ctest                   Classical Tests
eda                     Exploratory Data Analysis
fields                  Tools for spatial data
foreign                 Read data stored by Minitab, S, SAS, SPSS,
                        Stata, ...
geoR                    geoR - functions for geostatistical data
                        analysis
.
.
.
```

The list continues. Let's now install the package `odesolve`. The on-screen details will depend on both the specific package being loaded and on the operating system under which you are running R. Nonetheless, the command is the same across all packages and platforms:

```
> install.packages("odesolve")
```

You're done. To use the tools in the library, you need to load it into R.

```
> library(odesolve)
```

You need to do this second bit every time you want to use a specific library. The `install.packages()` command is a one-time deal. Once you've got it, it's there; it just needs to be loaded.

Now, some time down the line, you may want to upgrade your packages. You can do this for all packages simultaneously by simply using the following command.

```
> update.packages()
```

4.2 R Objects

Some discussion of objects. Then we talk about vectors and matrices and eventually lists and data frames.

4.2.1 Manipulating Lists: Vectors and Matrices

A vector is simply a list of numbers arranged either as a row or a column. We represent population structure – whether by age, stage, geographic location, sex, or whatever – with vectors. Say that we perform a census on a community of chimpanzees. Nishida et al. (1990) suggest that

the chimpanzee life cycle can be meaningfully structured according to the following categories: (1) infant, (2) older infant, (3) juvenile, (4) adolescent, (5) older adolescent, (6) prime adult, and (7) old adult. In 1980, Nishida et al.'s census of the Mahale Mountain M-group revealed that the community is composed of 7 infants, 13 older infants, 8 juveniles, 13 adolescents, 5 older adolescents, 35 prime adults, and 9 old adults. We could represent this information in a compact manner by defining a column vector c_x:

$$c_x = \begin{bmatrix} 7 \\ 13 \\ 8 \\ 13 \\ 5 \\ 35 \\ 9 \end{bmatrix} \tag{1}$$

To define this vector in R, simply type:

```
> cx <- c(7, 13, 8, 13, 5, 35, 9)
```

Typing `cx` will cause R to echo the new variable:

```
> cx
[1]  7 13  8 13  5 35  9
```

Nearly everything that you type into R will be in the form of `lhs` ← `rhs`. What this means is that you name something on the left hand side (`lhs`) of the assignment operator (i.e., <−, a less-than sign followed by a minus sign to make a left-facing arrow) and enter the value for that thing on the right hand side (`rhs`). In this chimpanzee example, the `lhs` is `cx`, and the `rhs` is the vector of stage abundances. You can use any string (i.e., combination of letters and numbers and underscores) for a name as long as it begins with a letter. While you can give a variable any name, it is good practice to choose meaningful names that will help you remember what they stand for. If you have a vector of reproductive values, rather than calling it `stuff`, instead call it something like `vx`. R variables and functions are case sensitive, so the variable `c` is distinct from the variable `C`.

Vectors are formed by concatenating a list of numbers using the R command `c`. The elements of the list are separated by commas.

```
> cx <- c(7, 13, 8, 13, 5, 35, 9)
```

The default for R is not to echo input. To view your input simply type, `cx` at the command prompt:

```
> cx
[1]  7 13  8 13  5 35  9
```

Nishida et al. (1990) actually provide a time series of population counts. It is often convenient to represent such time series in a single array (e.g., if you wanted to plot changes in demographic composition over time). Using the Mahale census data from 1988, we can construct a matrix of population counts composed of two censuses separated by eight years. The matrix will contain seven rows corresponding to the seven stages of the chimpanzee life cycle. It will also contain two columns corresponding to the two different years, 1980 and 1988. To construct this matrix, we type:

```
> cx1980 <- c(7, 13, 8, 13, 5, 35, 9)
> cx1988 <- c(9, 11, 15, 8, 9, 38, 0)
> C <- cbind(cx1980, cx1988)
```

The output of these commands is:

```
> C
     cx1980 cx1988
[1,]      7      9
[2,]     13     11
[3,]      8     15
[4,]     13      8
[5,]      5      9
[6,]     35     38
[7,]      9      0
```

We used the command `cbind` to bind two vectors together in an array. Note that R is agnostic as to whether a vector of numbers that you enter is a row vector or column vector. By using the command `cbind`, we forced them to be read as column vectors.

What would happen is you typed in `C <- c(cx1980, cx1988)` ?

Now what would happen if you typed

`> C <- t(cbind(cx1980, cx1988))?`

The R command `t` is the transpose operator, which exchanges the rows and columns of a matrix.

When combining vectors and matrices in R, it is very important that you keep track of the sizes of your arrays. For example, Boesch and Boesch-Achermann (2000) categorize the chimpanzee life cycle in a different way from Nishida and colleagues. Instead of the seven stages, Boesch & Boesch-Achermann employ just four: (1) infant, (2) juvenile, (3) adolescent, and (4) adult. Their census data for 1982 is entered below:

```
> cx.boesch <- c(18,10,15,30)
> cx.boesch
[1] 18 10 15 30
```

What happens if we try to combine the Nishida and Boesch data?

```
> cbind(cx.boesch,C)
     cx.boesch cx1980 cx1988
[1,]        18      7      9
[2,]        10     13     11
[3,]        15      8     15
[4,]        30     13      8
[5,]        18      5      9
[6,]        10     35     38
[7,]        15      9      0
Warning message:
number of rows of result
        is not a multiple of vector length (arg 1) in: cbind(cx.boesch, C)
```

We get a warning message telling us that we tried to combine two things of different lengths, but we still get an array of numbers for output. R uses what is known as a recycling rule to combine arrays of differing lengths. This means that the elements of the shorter structure get re-used, starting over with the first, until it is the length of the longer structure. Note that the warning message is only given if the longer structure is not a multiple of the shorter one:

```
> a <- c(2,4)
> b <- c(1,3,1,3)
> cbind(a,b)
     a b
[1,] 2 1
[2,] 4 3
[3,] 2 1
[4,] 4 3
```

The recycling rule can be a powerful programming tool, but it can also cause serious (and potentially comical) problems. Beware.

4.2.2 Accessing Elements in Arrays

Arrays are indexed by their row and column addresses. Wood and Smouse (1982) conducted a demographic study of the Gainj of the New Guinea highlands. From their data, we can construct a Leslie matrix with five 10-year age classes. Remember that a Leslie matrix contains age-specific fertilities along the first row and age-specific survival probabilities along the subdiagonal. The matrix for the Gainj is:

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 1.0791 & 1.1676 & 0.25 \\ 0.939 & 0 & 0 & 0 & 0 \\ 0 & 0.995 & 0 & 0 & 0 \\ 0 & 0 & 0.981 & 0 & 0 \\ 0 & 0 & 0 & 0.973 & 0 \end{bmatrix}, \tag{2}$$

We can enter these data into R as follows:

```
> gainj <- c(0, 0, 1.0791, 1.1676, 0.25,0.939, 0, 0, 0, 0, 0, 0.995, 0,
+ 0, 0,0, 0, 0.981, 0, 0,0, 0, 0, 0.973, 0)

> gainj.leslie <- matrix(data=gainj,nrow=5,ncol=5,byrow=TRUE,dimnames=NULL)
> gainj.leslie
      [,1]  [,2]   [,3]   [,4] [,5]
[1,] 0.000 0.000 1.0791 1.1676 0.25
[2,] 0.939 0.000 0.0000 0.0000 0.00
[3,] 0.000 0.995 0.0000 0.0000 0.00
[4,] 0.000 0.000 0.9810 0.0000 0.00
[5,] 0.000 0.000 0.0000 0.9730 0.00
```

What did we just do? First, we entered the Leslie matrix data. We entered these data row-wise, meaning that our vector `gainj` has as it entries the entire first row, followed by the second, and so on. Note that by hitting `<return>` before closing the parentheses, R suspends input and allows you to continue typing your input on the next line. We then used the command `matrix` to re-shape the 25-element vector into a 5×5 matrix.

As an aside, what would we have done if we could not quite remember the syntax for forming a matrix from a data vector? Probably the easiest way to check on the syntax of an R function is by using the command line help facility. For our case, type `?matrix`. Depending on your system, you will either get a new window with the R Help Information of the `matrix` command or the information will appear on your command line.

Is is frequently useful to extract a specific element of the matrix we have constructed. R provides a variety of facilities for extracting elements of arrays (R Development Core Team 2005b; Venables and Ripley 1999). Probably the simplest is to use square brackets with the matrix index. Say we are interested in the age-specific fertility value of women between ages 20-30. This demographic rate is represented by the third element of the first row. To extract this, type:

```
> gainj.leslie[1,3]
[1] 1.0791
```

Vectors are indexed by a single number. Say we want to get the number of chimpanzee juveniles from the 1980 M-group at Mahale. Juveniles are the third stage of the developmental sequence, consequently, the number of juveniles will be the third element of the vector.

```
> cx[3]
[1] 8
```

Frequently, we want to extract more than a just a single element from an array we have made. For example, we might want all of the age-specific fertility values from a Leslie matrix. One way to do this is as follows:

```
> gainj.Fx <- gainj.leslie[1,]
> gainj.Fx
[1] 0.0000 0.0000 1.0791 1.1676 0.2500
```

By not specifying a column index following the comma, we instruct R to extract all columns corresponding to the specified row index.

To extract multiple rows or columns, we us a colon to specify the range. If we wanted, for example, the 4×4 submatrix of the Gainj Leslie matrix that had the survival probabilities along the diagonal, we would do the following:

```
> gainj.leslie[2:5,1:4]
      [,1]  [,2]  [,3]  [,4]
[1,] 0.939 0.000 0.000 0.000
[2,] 0.000 0.995 0.000 0.000
[3,] 0.000 0.000 0.981 0.000
[4,] 0.000 0.000 0.000 0.973
```

R allows the use of negative subscripts as well. This can be useful if, for example, we want all of a matrix but one column. To get only the first 4 columns of the Gainj Leslie matrix using negative indices:

```
> gainj.leslie[,-5]
      [,1]  [,2]   [,3]   [,4]
[1,] 0.000 0.000 1.0791 1.1676
[2,] 0.939 0.000 0.0000 0.0000
[3,] 0.000 0.995 0.0000 0.0000
[4,] 0.000 0.000 0.9810 0.0000
[5,] 0.000 0.000 0.0000 0.9730
```

Another powerful way to access matrix elements is by using a logical vector. Say that we wanted to access only the non-zero elements of the Gainj Leslie matrix. We can do this efficiently with a logical vector. The commonly used logical operators are given in table 4.2.2.

Logical vectors underlie the concept of conditional execution. That is, an operation is only executed if the value of the logical is true.

```
> q <- gainj.leslie != 0
> gainj.leslie[q]
[1] 0.9390 0.9950 1.0791 0.9810 1.1676 0.9730 0.2500
```

operator	function
==	equals
<=	less than or equal
>=	greater than or equal
!	not
&	and
|	or

What does the structure `q` look like?

Logical vectors and matrices are powerful tools for speeding up calculations. We can frequently use a logical matrix in place of a control loop when programming, saving valuable computational time. Note that for the previous example, the traditional way to assemble a vector of the nonzero elements of the Leslie matrix would be to use two nested `for` loops with a logical evaluation of the form if the ijth number is not equal to zero, then put it in the vector, otherwise move on.

Now that we have a projection matrix, we can project the population forward in time. Assume we start with a population structure of a single person in each of the five age classes. We therefore need to create an initial population vector which we will call `no`. We also need to create a matrix to hold the projected population. We will call that `N`. The length of our projection will be ten years, so with our initial population, we will need a matrix which contains 11 columns.

```
> no <- rep(1,5)
> N <- matrix(0,nrow=5,ncol=11)
> N[,1] <-no
> for (i in 2:11){
>   N[,i] <- gainj.leslie%*%N[,i-1]
> }
```

To plot the change in the total population size, we can sum the columns of our population and then use the R function `plot`. To sum along the columns of `N`, we use the R function `apply`.

```
> pop <- apply(N,2,sum)
> t <- 0:10
> plot(t,pop)
> lines(t,pop)
```

4.2.3 Basic Operations

Typically, the reason we want to construct vectors and matrices or extract elements from them is so we can perform operations on them. If we know the age-specific survival probability of 0-5 year-olds, and we have a census count of the number of newborns in a year, we might want to

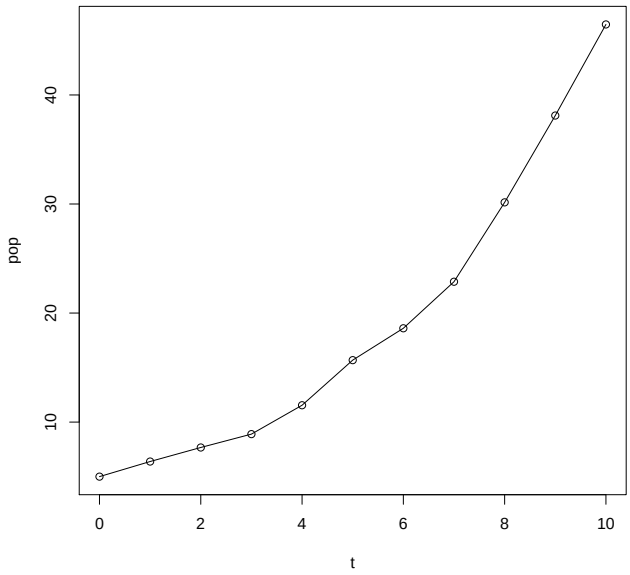

Figure 3: 100-year (10 interval) projection of an initial population of five females using the Gainj Leslie Matrix (Equation 2).

make a prediction about the number of these newborns who will still be alive at the end of five years. To do this, we multiply the p_x value for the 0-5 year period by the appropriate entry from the c_x vector. If $p_5 = .78$ and $c_1 = 123$, then the predicted number of survivors in five years is:

```
> n <- .78 * 123
> n
[1] 95.94
```

The operator (*) means multiplication. The other basic arithmetic operators are (+) for addition, (-) for subtraction, (/) for division, and (^) for power.

4.2.4 Plots and Graphs

One of the coolest things about R is its ability to make awesome graphics. We'll be keeping it simple for the time-being. A graph is made taking two (usually) vectors of the same length and plotting the elements of one against the elements of the other. The command for this is `plot()`, which takes three arguments, the third being optional.

Binford and Chasko (1976) present vital statistic data for the Nunamiut of Alaska, USA 1935-1968. Entering the time series of population counts given in their table 10, we get.

```
> nun <- c(66, 66, 65, 67, 68, 70, 69, 71, 71, 74, 74, 73, 74, 74, 76,
```

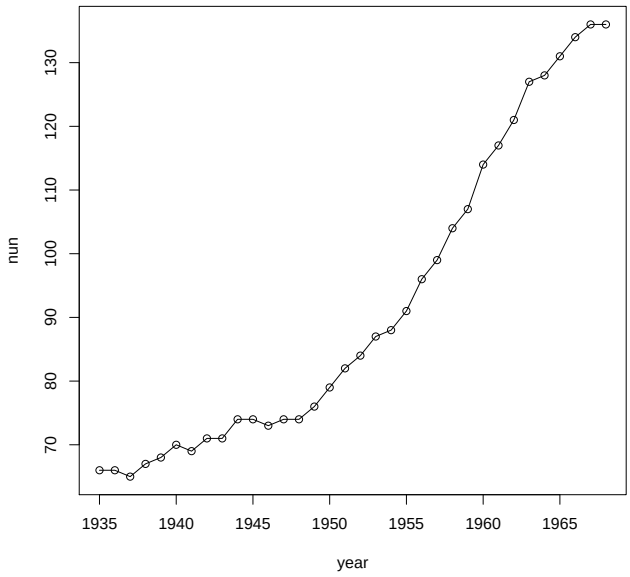

Figure 4: Changes in Total Population Size of the Nunamiut, 1935-1968.

```
+ 79, 82, 84, 87, 88, 91, 96, 99, 104, 107, 114, 117, 121, 127, 128,
+ 131, 134, 136, 136)
>
> year <-  1935:1968
```

To plot this time-series of population counts against the year, we simply type the following command:

```
> plot(year,nun)
> lines(year,nun)
```

This yields the plot presented in figure 4.

The plot in figure 4 allows us to visualize the time series of population counts well, but it is very basic. To improve the impact of the plot, we might want to add color to it. We should also give it more descriptive axis labels and possibly add a title.

R provides an incredible array of tools for creating professional graphics. We will focus on only a few graphics commands.

Let's re-plot the data with the points in color, the line a different color, and add both axis labels and a title to the plot. We can control the plotting symbol using the argument `pch=n`, where `n` is a number between 0 and 18. We define the plot as a scatter plot of points using `type="p"`. Other possible types of plots include "l" for lines, "b" for both, as well as a variety of others. We will use type "p" to facilitate making the points and lines different colors.

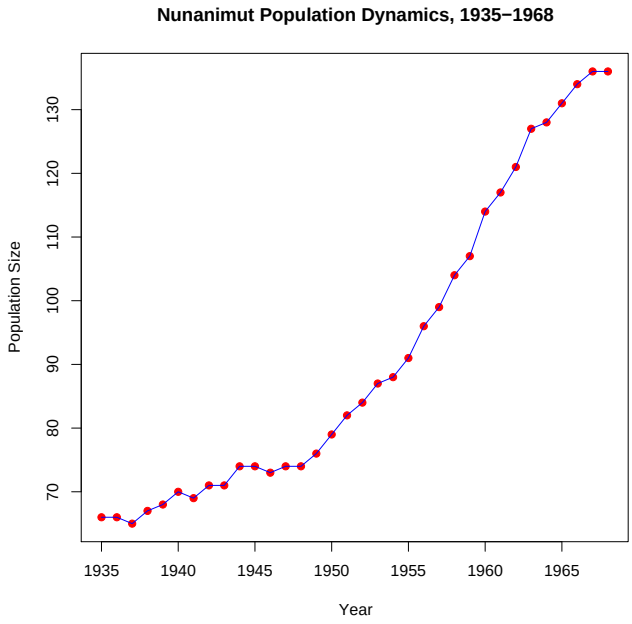

Figure 5: Changes in Total Population Size of the Nunamiut, 1935-1968, re-plotted.

```
> plot(year,nun,pch=16,col="red",type="p",
+     xlab="Year", ylab="Population Size")
> lines(year,nun,col="blue")
> title(main="Nunanimut Population Dynamics, 1935-1968")
```

The results are plotted in figure 5.

A graphical means for diagnosing exponential population growth is to see if the plot of the logarithm of population size against time produces a straight line. We can specify logarithmic axes using the argument `log`. This argument can take the value of "x", "y" or "xy" depending on which axis we want to be a logarithmic scale. Going back to the Gainj population projection, we can plot the data on semilogarithmic axes as follows:

```
> plot(t,pop,pch=16,col="red",type="p",log="y",
+      xlab="Year", ylab="Population Size")
> lines(t,pop,col="blue")
```

Figure 6 shows that the Gainj population projection shows exponential growth. The plot is approximately linear on the semilogarithmic axes.

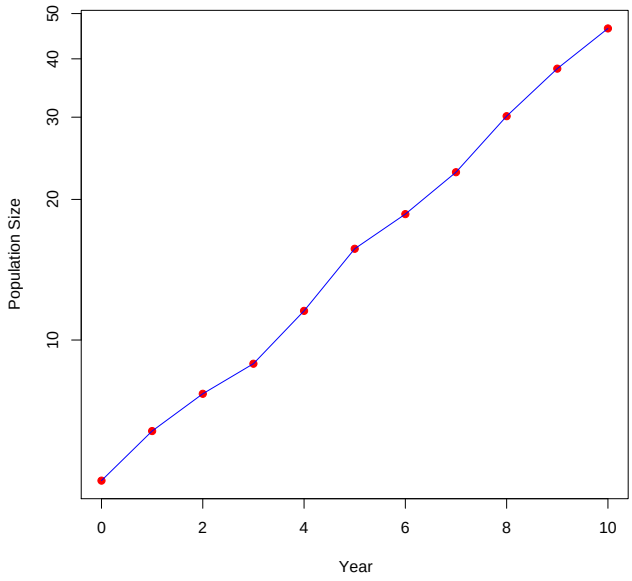

Figure 6: 100-year population projection of the Gainj, semilogarithmic axes.

4.2.5 Multiple Plots

We frequently want to plot multiple things on the same axes. For example, we may wish to simultaneously plot empirical observations or the results of a simulation along with the predictions of a theoretical model. R makes this very easy. In this example, we will generate a stochastic growth process and compare the results from the expected deterministic results. We begin by generating random normal variates with mean 1.02 and standard deviation 0.02.

```
> x <- rep(0,100)
> x[1] <- 20

> # generate normal random numbers for rate of increase
> w <- rnorm(100,mean=1.02,sd=0.02)

> # growth with noise
> for (i in 2:100) x[i] <- x[i-1]*w[i]
> plot(t,x,pch=3,col="red")

> #compare to plot without noise
> xx <- rep(0,100)
> xx[1] <- 20
> R <- mean(w)
> for (i in 2:100) xx[i] <- R*xx[i-1]
```

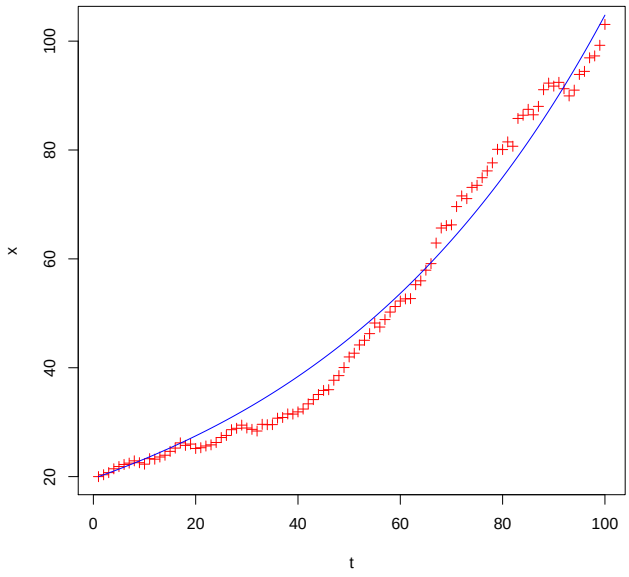

Figure 7: Comparison of simulated growth data with white noise and the deterministic model

```
> lines(t,xx,col="blue")
```

Note that we use a # to indicate a comment which R will ignore. Comments are essential when you write complex scripts, but they can also be handy when entered at the command line, particularly if you save your work history for future reference (a highly recommended practice!).

Not surprisingly, the model fits very well with the simulation (figure 7).

4.3 Other Plots

Key plots for EDA, data summary, etc.

Generate 1000 normally distributed pseudo-random numbers with zero mean and unit standard deviation. Contaminate that list with 100 variates with the same mean but five times the standard deviation. First, we can look at the histogram.

```
> x <- rnorm(1000,mean=0,sd=1)
> xx <- rnorm(100,mean=0,sd=5)
> y <- c(x,xx)
> # generate a histogram with 50 bins
> hist(y,50)
```

The distribution represented in figure 8 looks suspiciously leptokurtotic, suggesting there is a contamination. Another way to visualize this is to use a boxplot.

```
> # make another, uncontaminated vector for comparison
> yy <- rnorm(1100,mean=0,sd=1)
> boxplot(y,yy,boxwex=0.25, notch=TRUE)
```

Sure enough – though I guess we knew it already. `boxvex=0.25` and `notch=TRUE` are *arguments* passed to the command `boxplot()` command that help control the appearance of the plot. To find a complete list of arguments, type `?boxplot` at the command line.

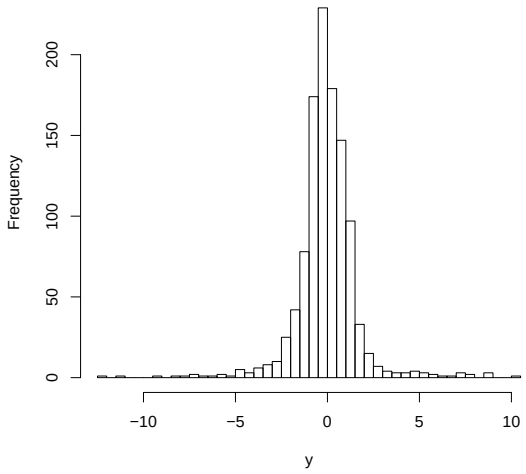

Figure 8: Histogram of 1100 pseudo-random numbers drawn from a standard normal distribution except for 100 which drawn from a normal distribution a standard deviation of 5.

4.3.1 Lists & Data Frames

So far, we have talked about data structures such as vectors and matrices that can be considered discrete units (i.e., we perform operations on them as complete structures even though we may wish to extract information from them). However, we also frequently want to work with data that are related, but will not necessarily be used together.

Lists A list is simply a collection of different items. Say that we were collecting data on the kinship networks of children. We interview heads of households about all the children currently residing in their home, and determine whether these children are natal or fostered, whether their mother and father are alive and the current ages of the child's parents or, if the child is fostered, their parents' ages at death. A list which would hold this information for a particular child might look like this:

```
> child1 <- list(name="mary", child.age=6, status="foster",
```

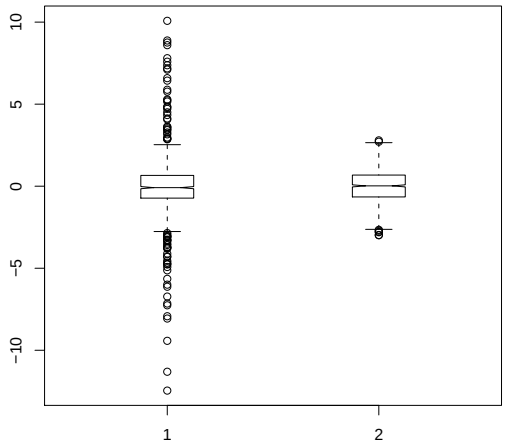

Figure 9: Boxplot comparing the same 1100 pseudo-random numbers drawn from the contaminated distribution with 1100 drawn from a standard normal.

```
+ mother.alive=F, father.alive=T, parents.ages=c(24,35))
```

This list contains character data, numeric data, and logical data. There are a number of options for accessing the components of a list. Probably the easiest is to use the component names. To extract components using the component names, we use the $ operator. For example, if we wanted to determine if the child's mother were alive, we could type:[1]

```
> child1$mother.alive
[1] FALSE
```

We can add components to a list using the concatenate function

```
> child1 <- c(child1, no.siblings=2, sib.ages=c(1,3))
```

Data Frames A data frame is an R object that holds a data matrix. It is essentially a list of variables, all with the same length.

Data frames can be entered at the R command line, but the most common means of creating data frames is to read them in from delimited text files. R also has a special library for reading in common binary formats (R Development Core Team 2005a).

[1]Excuse the spurious $ in wherever I simulate R commands that involve extracting elements of R objects. They represent an imperfect typesetting work-around.

We will use the `read.table` function to create a data frame holding life table information for the country of Venezuela in 1963 (Keyfitz and Flieger 1990). The columns of the data file are: age, number of women in age class (${}_nK_x$), number of deaths (${}_nM_x$), and the number of births (${}_nB_x$). Put the name of the text file you are reading in double quotes. If the file is not in your working directory, you need to specify the path. The path specification will vary by the machine that you use. The R documentation strongly recommends that you specify a header. It does seem to make the whole process go more smoothly, so let's do that.

```
> ven <- read.table("/home/jhj1/Teaching/summercourse/venezuela.deaths.dat",
+ header=TRUE)
>
> ven
   age    nKx  nMx    nBx
1    0 151377 7670      0
2    1 569468 3640      0
3    5 589891  926      0
4   10 489546  299    796
5   15 392537  334  49162
6   20 324897  565 104087
7   25 281655  572  88138
8   30 253215  665  59394
9   35 214278  637  38580
10  40 170131  801  10870
11  45 146577  820   2251
12  50 121309  998    268
13  55  90784 1132      0
14  60  74128 1811      0
15  65  52962 1107      0
16  70  36029 1504      0
17  75  25374 1055      0
18  80  15262 1345      0
19  85   6757 1412      0
```

As with lists, we can access individual columns of the data frame by using the $ operator:

```
> ven$age
[1]  0  1  5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85
```

When we are working intensively with one data frame, it frequently becomes inconvenient to have to type full variable names with the $ operator. To overcome this, we can use the R function `attach`, which will make the columns of the data frame visible as variables themselves.

```
> attach(ven)
```

```
> age
[1]  0  1  5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85

>
> detach(ven)
```

4.3.2 Getting Help

When you need help with, say, the syntax of an R function, probably the simplest way to get help is with the command line help facility. To create the stochastic growth model above, we needed to generate random normal variates. If we wanted to get help about the function `rnorm`, we could type at the command line `?rnorm`. Depending on the system, we would either get on the command line or in a newly spawned window a help file. Here is the beginning of the help file for `rnorm`:

```
Normal                  package:base                  R Documentation

The Normal Distribution

Description:

     Density, distribution function, quantile function and random
     generation for the normal distribution with mean equal to 'mean'
     and standard deviation equal to 'sd'.

Usage:

     dnorm(x, mean=0, sd=1, log = FALSE)
     pnorm(q, mean=0, sd=1, lower.tail = TRUE, log.p = FALSE)
     qnorm(p, mean=0, sd=1, lower.tail = TRUE, log.p = FALSE)
     rnorm(n, mean=0, sd=1)

Arguments:

     x,q: vector of quantiles.

       p: vector of probabilities.

       n: number of observations. If 'length(n) > 1', the length is
          taken to be the number required.

    mean: vector of means.
```

```
      sd: vector of standard deviations.

log, log.p: logical; if TRUE, probabilities p are given as log(p).

lower.tail: logical; if TRUE (default), probabilities are P[X <= x],
          otherwise, P[X > x].
```

In addition to the information shown above, this file contains more technical details of the function's implementation, a cross-referenced list of similar functions, and a series of examples. These examples can be particularly illuminating for functions that you have never used before.

R also maintains these files in html format. One advantage to this format is that the html-interface includes a searchable directory. Say, for example, that we could not remember the exact name of the function that generated normally-distributed pseudo-random numbers. We could then search the html-based help for the keyword "normal."

4.3.3 Keeping a Diary, Saving and Loading Data

It is always a good idea to keep a diary of your work.

A good way to work with R is to compose your commands in a text editor. This allows you to easily edit the R code you have written, and it keeps a log of your work that can be saved throughout your R session.

The text editor emacs has a mode called ESS (emacs speaks statistics). ESS allows you to compose all your commands in emacs, giving you all of its powerful editing and bug-checking capabilities, and send them to R to calculate. There are versions of ESS for both Unix (and therefore, Macintosh OS X) and Windows.

4.3.4 Saving Data

You can save your R session as a disk image. This has the benefit of maintaining all your variables, and sourced code intact and exactly as you left it. The major disadvantage is that these R disk images take up a lot of memory.

4.3.5 Writing Functions & Sourcing Code

If we wanted to perform a complex operation, we could type in the commands sequentially at the command line. This can become quite tedious if we want to re-run an analysis with a different variable or set of arguments. The most efficient way of performing complex operations (e.g., writing our own functions) is to compose them in a text editor and then "source" them in R.

Imagine that we had a reason to do the comparison of the simulated stochastic growth process to the mean deterministic process . Rather than re-entering the commands every time we wanted to run the simulation and generate a plot, it would make more sense to write a function to perform these tasks. In a text editor, we then would type:

```
stoch.plot <- function(x0=20,m,s,n=100)
   x <- rep(0,n)
```

```
  x[1] <- x0

 # generate normal random numbers for rate of increase
 w <- rnorm(n,mean=m,sd=s)

 # growth with noise
 for (i in 2:n) x[i] <- x[i-1]*w[i]
 plot(t,x,pch=3,col="red")

 #compare to plot without noise
 xx <- rep(0,n)
 xx[1] <- x0
 R <- mean(w)
 for (i in 2:n) xx[i] <- R*xx[i-1]
 lines(t,xx,col="blue")
```

In doing this, we have defined a function `stoch.plot` which takes four arguments (`x0, m, s, n`), and produces a plot similar to the one we made in section 4.2.5. Note that the first and last arguments are given default values (20 and 100). We can over-ride the defaults by specifying alternatives, but the function requires only two arguments to be specified by the user. All the commands which comprise the function are enclosed within curly braces.

To execute this function, we first save it. Suppose that we saved the file directly into our working directory, and called it `stoch.plot.r`. We could then source the file by typing

```
> source("stoch.plot.r")
```

To use this function, we then type the command:

```
> stoch.plot(m=1.05,s=.1)
```

When making complex graphics – particularly for presentation or publication – it is always a good idea to write a function to render the graphic for you. That way, you can tweak it repeatedly until it is exactly the way you want it without having to re-type all the commands. Furthermore, you can easily re-create the graphic long after you originally made the figure (e.g., when a submitted paper comes back from reviewers with comments and you've long since forgotten how you made the figures!).

References

Binford, L. and W. Chasko (1976). Nunamiut demographic history: A provocative case. In E. Zubrow (Ed.), *Demographic anthropology: Quantitative approaches*, pp. 63–144. Albuquerque: University of New Mexico Press.

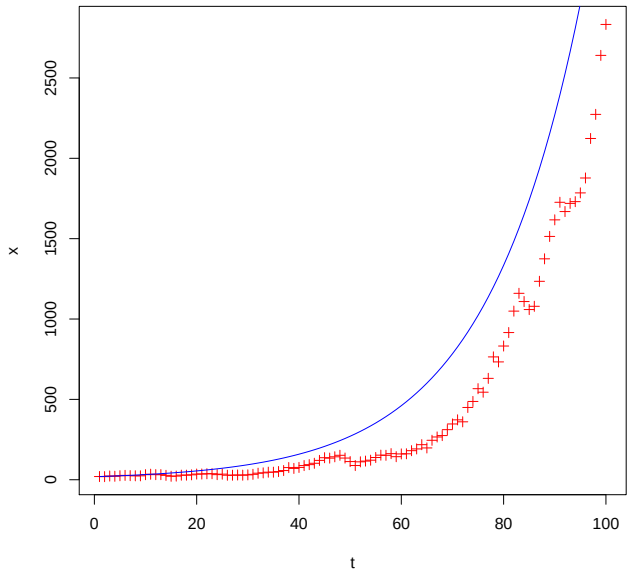

Figure 10: Comparison of simulated growth data with white noise and the deterministic model as generated by the function `stoch.plot`.

Boesch, C. and H. Boesch-Achermann (2000). *The chimpanzees of the Taï Forest: Behavioural ecology and evolution*. Oxford: Oxford University Press.

Keyfitz, N. and W. Flieger (1990). *World population growth and aging: Demographic treends in the late twentieth century*. Chicago: University of Chicago Press.

Nishida, T., H. Takasaki, and Y. Takahata (1990). Demography and reproductive profiles. In *The chimpanzees of the Mahale Mountains*, pp. 63–97. Tokyo: University of Tokyo Press.

R Development Core Team (2005a). *R Data Import/Export* (2.1.1 ed.). Vienna: R Foundation for Statistical Computing. ISBN: 3-900051-10-0.

R Development Core Team (2005b). *R Language Definition* (2.1.1 ed.). Vienna: R Foundation for Statistical Computing. ISBN: 3-900051-13-5.

Venables, W. and B. Ripley (1999). *Modern applied statistics with S-PLUS* (3rd ed.). New York: Springer.

Wood, J. W. and P. E. Smouse (1982). A method of analyzing density-dependent vital rates with an application to the Gainj of Papua New Guinea. *American Journal of Physical Anthropology 58*(4), 403–411.