Poppr 1.1.0.99: An R package for genetic analysis of populations with mixed (clonal/sexual) reproduction

Zhian N. Kamvar[1] and Niklaus J. Grünwald[1,2]

1) Department of Botany and Plant Pathology, Oregon State University, Corvallis, OR

2) Horticultural Crops Research Laboratory, USDA-ARS, Corvallis, OR

May 2, 2014

Abstract

Poppr provides open-source, cross-platform tools for quick analysis of population genetic data enabling focus on data analysis and interpretation. While there are a plethora of packages for population genetic analysis, few are able to offer quick and easy analysis of populations with mixed reproductive modes. *Poppr*'s main advantage is the ease of use and integration with other packages such as *adegenet* and *vegan*, including support for novel methods such as clone correction, multilocus genotype analysis, calculation of Bruvo's distance and the index of association.

Contents

1 Introduction

1.1 Purpose

Poppr is an R package with convenient functions for analysis of genetic data with mixed modes of reproduction including sexual and clonal reproduction. While there are many R packages in CRAN and other repositories with tools for population genetic analyses, few are appropriate for populations with mixed modes of reproduction. There are several stand alone programs that can handle these types of data sets, but they are often platform specific and often only accept specific data types. Furthermore, a typical analysis often involves switching between many programs, and converting data to each specific format.

Poppr is designed to make analysis of populations with mixed reproductive modes more streamlined and user friendly so that the researcher using it can focus on data analysis and interpretation. *Poppr* allows analysis of haploid and diploid dominant/co-dominant marker data including microsattelites, Single Nucleotide Polymorphisms (SNP), and Amplified Fragment Length Polymorphisms (AFLP). To avoid creating yet another file format that is specific to a program, *poppr* was created on the backbone of the popular R package *adegenet* and can take all the file formats that *adegenet* can take (Genpop, Genetix, Fstat, and Structure) and newly introduces compatibility with GenAlEx formatted files (exported to CSV). This means that anything you can analyze in *adegenet* can be further analyzed with *poppr*.

The real power of *poppr* is in the data manipulation and analytic tools. *Poppr* has the ability to define multiple population hierarchies, clone- censor, and subset data sets. With *poppr* you can also quickly calculate Bruvo's distance, the index of association, and easily determine which multilocus genotypes are shared across populations.

1.2 Installation

This manual assumes you have installed R. If you have not, please refer to The CRAN home page at http://cran.r-project.org/. We also recommend the Rstudio IDE (http://www.rstudio.com/), which allows the user to view the R console, environment, scripts, and plots in a single window.

1.2.1 From CRAN

To install *poppr* from CRAN, select "Package Installer" from the menu "Packages & Data" in the gui or type:

```
install.packages("poppr", dependencies = TRUE)
```

All dependencies (*adegenet, pegas, vegan, ggplot2, phangorn, ape, reshape2* and *igraph*) will also be installed. In the unfortunate case this does not work, consult http://cran.r-project.org/doc/manuals/R-admin.html#Installing-packages.

1.2.2 From Source

The tarball for *poppr* can be downloaded from CRAN: http://cran.r-project.org/package=poppr, under the RESOURCES tab in the Grünwald Lab website: http://http://grunwaldlab.cgrb.oregonstate.edu/, or via github at https://github.com/grunwaldlab/poppr.

Since *poppr* contains C code, it needs to be compiled, which means that you need a working C compiler. If you are on Linux, you should have that, but if you are on Windows or OSX, you might need to download some special tools:

Windows Download Rtools: http://cran.r-project.org/bin/windows/Rtools/

OSX Download Xcode: https://developer.apple.com/xcode/

If you choose to install *poppr* from a source file, you should first make sure to install all of the dependencies with the following command:

```
install.packages(c("adegenet", "pegas", "vegan", "ggplot2", "phangorn", "ape",
    "igraph"))
```

If you want to install from github, skip to the next section.
After installing dependencies, download the package to your computer and then install it with:

```
install.packages("/path/to/poppr.tar.gz", type = "source", repos = NULL)
```

1.2.3 From github

Github is a repository where you can find all stable and development versions of *poppr*. Installing from github requires a C compiler, so be sure to read the section above for instructions on how to obtain that if you aren't on a Linux system.

To install from github, you do not need to download the tarball since there is a package called *devtools* that will download and install the package for you directly from github. After you have installed all dependencies (see above section), you should download *devtools*:

```
install.packages("devtools")
```

Now you can execute the command `install_github` with the user and repository name:

```
library(devtools)
install_github(repo = "grunwaldlab/poppr")
```

If you are the adventurous type and are willing to test out unreleased versions of the package, you can also install the development version:

```
library(devtools)
install_github(repo = "grunwaldlab/poppr", ref = "devel")
```

Users who install this version do so at their own risk. Since it is a development version, documentation may be incomplete or nonexistant for new functions.

1.3 Quick start

The author assumes that if you have reached this point in the manual, then you have successfully installed R and *poppr*. Before proceeding, you should be aware that R is case sensitive. This means that the words "Case" and "case" are different. You should also know where your R package Library is located.

> **What or where is my R package library?**
> R is as powerful as it is through a community of people who submit extra code called "Packages" to help it do specific things. These packages live in a certain place on your computer called an R library. You can find out where this library is by typing `.libPaths()`

Importing a file into R involves you knowing the path to your file and then typing that into R's console. `getfile()` will help provide a point and click interface for selecting a file. There are two steps:

Tell your computer to search R's library to find the *poppr* and load the package:

```
library(poppr)
```

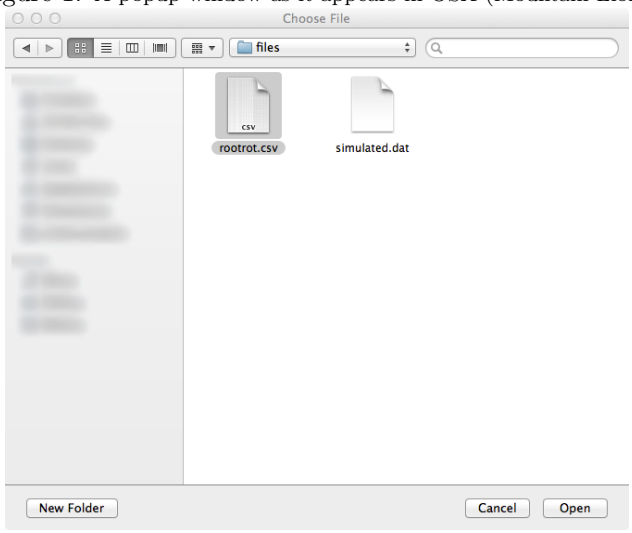
After that, you can use `getfile()`

```
x <- getfile()
```

A pop up window will appear like this[1]:

[1] This window sometimes appears behind your current session of R, depending on the GUI and you will have to toggle to this window

Figure 1: A popup window as it appears in OSX (Mountain Lion).



Hey! My window doesn't look like that!

Now, this window will not match up to your window on your computer because you will probably not be in the right directory. Remember the first path in `.libPaths()`? Move to a folder called **poppr** in that path. In that folder, you will find another folder called **files**. Move there and your window will match the one displayed.

We can navigate throughout your entire computer through this window and tell R where to go. The example I'm using goes to your R library directory where *poppr* is stored. If you don't know where that is, you can find it by typing `find.package('poppr')` into the R command line. Once we select a file, the file name and its path will be stored in the variable, x. We can confirm that by typing `x` into R's command line.

```
x

## $files
## [1] "/path/to/R/poppr/files/rootrot.csv"
##
## $path
## [1] "/path/to/R/poppr/files"
```

Here we can see that `x` is a list with two entries: `$files` shows the files you selected and `$path` shows the path to those files.

Not sure what I mean by path or working directory?

For anyone who has never used a command line, this is a new concept. You can think of the path as an address. So instead of `"/path/to/R"`, you could have `"/USA/Oregon/Corvallis"`. Or on your computer, it could be `"C:/users/poppr-user/R/win-library/3.1"` on Windows (where "poppr-user" is your username) or `"/Library/Frameworks/R.framework/Versions/3.1/Resources/library"` on OSX. Each slash represents a folder that you would click on when you are using the mouse.

A working directory is the folder that R is working in. It is where you can access and write files. When you tell R to read a file, it will only look for that file in your working directory. Note that you will not endanger your files by reading them into R. R works by making a copy of the file into memory. This means that you can manipulate the data in any way that you want without ever changing the original file.

To find what your current working directory is set to, type `getwd()` into the R console. Usually, you will start off a session in your "home" directory, which will look like this: `"~/"`. The command `setwd()` will change your working directory to any folder of your choice on your computer as indicated by the path that you provide. For more information, see Quick R at http://www.statmethods.net.

We will use `x$files` to access the file. The `poppr()` function provides a simple first analysis of your data directly from the file on the your disk (For information on importing your data into R, see section Get out of my dreams and into my R.)

```
popdata <- poppr(x$files)
```

```
## | Athena_1
## | Athena_2
## | Athena_3
## | Athena_4
## | Athena_5
## | Athena_6
## | Athena_7
## | Athena_8
## | Athena_9
## | Athena_10
## | Mt. Vernon_1
## | Mt. Vernon_2
## | Mt. Vernon_3
## | Mt. Vernon_4
## | Mt. Vernon_5
## | Mt. Vernon_6
## | Mt. Vernon_7
## | Mt. Vernon_8
## | Total
```

The output of `poppr()` was assigned to the variable `popdata`, so let's look at the data.

```
popdata
```

```
##            Pop   N MLG  eMLG    SE     H     G  Hexp   E.5    Ia rbarD        File
## 1     Athena_1   9   7  7.00 0.000 1.889  6.23 0.944 0.932  2.92 0.210 rootrot.csv
## 2     Athena_2  12  12 10.00   NaN 2.485 12.00 1.000 1.000  4.16 0.128 rootrot.csv
## 3     Athena_3  10   2  2.00 0.000 0.325  1.22 0.200 0.571  2.00 1.000 rootrot.csv
## 4     Athena_4  13   9  7.15 0.769 1.946  5.12 0.872 0.687  5.49 0.372 rootrot.csv
```

```
## 5      Athena_5  10   7  7.00 0.000 1.834  5.56 0.911 0.866  4.53 0.353 rootrot.csv
## 6      Athena_6   5   5  5.00 0.000 1.609  5.00 1.000 1.000  2.46 0.190 rootrot.csv
## 7      Athena_7  11  10  9.18 0.386 2.272  9.31 0.982 0.955  2.13 0.086 rootrot.csv
## 8      Athena_8   8   6  6.00 0.000 1.667  4.57 0.893 0.831  3.86 0.323 rootrot.csv
## 9      Athena_9  10  10 10.00 0.000 2.303 10.00 1.000 1.000  2.82 0.118 rootrot.csv
## 10    Athena_10   9   8  8.00 0.000 2.043  7.36 0.972 0.948  2.85 0.137 rootrot.csv
## 11 Mt. Vernon_1  10   9  9.00 0.000 2.164  8.33 0.978 0.952  7.13 0.276 rootrot.csv
## 12 Mt. Vernon_2   6   6  6.00 0.000 1.792  6.00 1.000 1.000 20.65 0.492 rootrot.csv
## 13 Mt. Vernon_3   8   6  6.00 0.000 1.667  4.57 0.893 0.831  2.12 0.106 rootrot.csv
## 14 Mt. Vernon_4  12   8  6.83 0.665 1.814  4.50 0.848 0.681  3.01 0.255 rootrot.csv
## 15 Mt. Vernon_5  17   7  5.54 0.828 1.758  5.07 0.853 0.848  2.68 0.340 rootrot.csv
## 16 Mt. Vernon_6  12  11  9.32 0.466 2.369 10.29 0.985 0.958 19.50 0.467 rootrot.csv
## 17 Mt. Vernon_7  12   9  7.82 0.649 2.095  7.20 0.939 0.870  1.21 0.153 rootrot.csv
## 18 Mt. Vernon_8  13   9  7.35 0.764 2.032  6.26 0.910 0.794  1.15 0.169 rootrot.csv
## 19        Total 187 119  9.61 0.612 4.558 68.97 0.991 0.720 14.37 0.271 rootrot.csv
```

The fields you see in the output include:

- `Pop` - Population name (Note that "Total" also means "Pooled").
- `N` - Number of individuals observed.
- `MLG` - Number of multilocus genotypes (MLG) observed.
- `eMLG` - The number of expected MLG at the smallest sample size ≥ 10 based on rarefaction. [8]
- `SE` - Standard error based on `eMLG` [7]
- `H` - Shannon-Wiener Index of MLG diversity. [16]
- `G` - Stoddart and Taylor's Index of MLG diversity. [18]
- `Hexp` - Nei's 1978 genotypic diversity (corrected for sample size), or Expected Heterozygosity. [11]
- `E.5` - Evenness, E_5. [15][10][4]
- `Ia` - The index of association, I_A. [2] [17] [1]
- `rbarD` - The standardized index of association, $\bar{r}_d$. [1]

These fields are further described in section I KNOW WHAT YOU DID LAST SUMMARY TABLE at the end of this vignette.

One thing to note about this output is the `NaN` in the column labeled `SE`. In R, `NaN` means "Not a number". This is produced from calculation of a standard error based on rarefaction analysis. Occasionally, this calculation will encounter a situation in which it must attempt to take a square root of a negative number. Since the root of any negative number is not defined in the set of real numbers, it must therefore have an imaginary component, i. Unfortunately, R will not represent the imaginary components of numbers unless you specifically tell it to do so. By default, R represents these as `NaN`.

1.4 Importing data into poppr {Get out of my dreams and into my R}

There are several ways of reading data into R. One way is using the function `getfile`.

1.4.1 Function: getfile

`getfile` gives the user an easy way to point R to the directory in which your data is stored. It is only meant for R GUIs such as Rstudio. Using this on the command line has little advantage over setting the working directory manually.

Default Command:

```
getfile(multi = FALSE, pattern = NULL, combine = TRUE)
```

- `multi` - This is normally set to `FALSE`, meaning that it will only grab the file you selected. If it's `TRUE`, it will grab all files within the directory, constrained only by what you type into the `pattern` field.
- `pattern` - A pattern that you want to filter the files you get. This accepts regular expressions, so you must be careful with anything that is not an alphanumeric character.
- `combine` - This tells `getfile` to combine the path and all the files. This is set to `TRUE` by default so that you can access your files no matter what working directory you are in.

This method works for a single file, but let's say you had a lot of data sets you wanted to import. Instead of doing these one-by-one, `getfile` has a flag called `multi` telling the computer that you want to grab multiple files in the folder:

```
x <- getfile(multi = TRUE)
```

A window would pop up again, and you should navigate to the same directory as you had before, and select any of the files in that directory.

```
x
```

```
## $files
## [1] "/path/to/R/poppr/files/rootrot.csv"   "/path/to/R/poppr/files/rootrot2.csv"
## [3] "/path/to/R/poppr/files/simulated.dat"
##
## $path
## [1] "/path/to/R/poppr/files"
```

As you can see, now all of the files that existed in that directory are there! Now you can look at all those files at once! We will use `poppr.all` to produce a summary table for all of your files[2]. Let's set `digits = 2` to only print 2 significant digits.

```
all_files <- poppr.all(x$files) print(all_files, digits = 2)
```

```
##  \
##   | File: rootrot.csv
##  /
## | Athena_1
## | Athena_2
## | Athena_3
## | Athena_4
## | Athena_5
## | Athena_6
```

[2]These files do not need to be similar in any way to do this analysis

```
## | Athena_7
## | Athena_8
## | Athena_9
## | Athena_10
## | Mt. Vernon_1
## | Mt. Vernon_2
## | Mt. Vernon_3
## | Mt. Vernon_4
## | Mt. Vernon_5
## | Mt. Vernon_6
## | Mt. Vernon_7
## | Mt. Vernon_8
## | Total
##  \
##   | File: rootrot2.csv
##  /
## | 1
## | 2
## | 3
## | 4
## | 5
## | 6
## | 7
## | 8
## | 9
## | 10
## | Total
##  \
##   | File: simulated.dat
##  /
## | Total
##             Pop   N MLG eMLG      SE    H    G Hexp  E.5    Ia rbarD          File
## 1      Athena_1   9   7  7.0 0.0e+00 1.89  6.2 0.94 0.93  2.92 0.210   rootrot.csv
## 2      Athena_2  12  12 10.0     NaN 2.48 12.0 1.00 1.00  4.16 0.128   rootrot.csv
## 3      Athena_3  10   2  2.0 0.0e+00 0.33  1.2 0.20 0.57  2.00 1.000   rootrot.csv
## 4      Athena_4  13   9  7.2 7.7e-01 1.95  5.1 0.87 0.69  5.49 0.372   rootrot.csv
## 5      Athena_5  10   7  7.0 0.0e+00 1.83  5.6 0.91 0.87  4.53 0.353   rootrot.csv
## 6      Athena_6   5   5  5.0 0.0e+00 1.61  5.0 1.00 1.00  2.46 0.190   rootrot.csv
## 7      Athena_7  11  10  9.2 3.9e-01 2.27  9.3 0.98 0.96  2.13 0.086   rootrot.csv
## 8      Athena_8   8   6  6.0 0.0e+00 1.67  4.6 0.89 0.83  3.86 0.323   rootrot.csv
## 9      Athena_9  10  10 10.0 0.0e+00 2.30 10.0 1.00 1.00  2.82 0.118   rootrot.csv
## 10    Athena_10   9   8  8.0 0.0e+00 2.04  7.4 0.97 0.95  2.85 0.137   rootrot.csv
## 11 Mt. Vernon_1  10   9  9.0 0.0e+00 2.16  8.3 0.98 0.95  7.13 0.276   rootrot.csv
## 12 Mt. Vernon_2   6   6  6.0 0.0e+00 1.79  6.0 1.00 1.00 20.65 0.492   rootrot.csv
## 13 Mt. Vernon_3   8   6  6.0 0.0e+00 1.67  4.6 0.89 0.83  2.12 0.106   rootrot.csv
## 14 Mt. Vernon_4  12   8  6.8 6.6e-01 1.81  4.5 0.85 0.68  3.01 0.255   rootrot.csv
## 15 Mt. Vernon_5  17   7  5.5 8.3e-01 1.76  5.1 0.85 0.85  2.68 0.340   rootrot.csv
## 16 Mt. Vernon_6  12  11  9.3 4.7e-01 2.37 10.3 0.98 0.96 19.50 0.467   rootrot.csv
## 17 Mt. Vernon_7  12   9  7.8 6.5e-01 2.09  7.2 0.94 0.87  1.21 0.153   rootrot.csv
## 18 Mt. Vernon_8  13   9  7.3 7.6e-01 2.03  6.3 0.91 0.79  1.15 0.169   rootrot.csv
## 19        Total 187 119  9.6 6.1e-01 4.56 69.0 0.99 0.72 14.37 0.271   rootrot.csv
## 20            1  19  16  9.2 7.0e-01 2.73 14.4 0.98 0.94 14.23 0.313  rootrot2.csv
## 21            2  18  18 10.0 5.4e-07 2.89 18.0 1.00 1.00  9.14 0.194  rootrot2.csv
## 22            3  18   8  5.3 1.0e+00 1.61  3.4 0.75 0.59 22.84 0.573  rootrot2.csv
## 23            4  25  17  7.9 1.1e+00 2.58  9.6 0.93 0.71 18.49 0.415  rootrot2.csv
## 24            5  27  14  7.5 1.1e+00 2.45  9.7 0.93 0.83 23.00 0.520  rootrot2.csv
## 25            6  17  15  9.3 6.3e-01 2.67 13.8 0.99 0.95 17.78 0.410  rootrot2.csv
## 26            7  23  19  9.2 7.6e-01 2.87 16.0 0.98 0.90 19.16 0.405  rootrot2.csv
## 27            8  21  15  8.3 9.8e-01 2.56 10.8 0.95 0.82 24.31 0.543  rootrot2.csv
## 28            9  10  10 10.0 0.0e+00 2.30 10.0 1.00 1.00  2.82 0.118  rootrot2.csv
## 29           10   9   8  8.0 0.0e+00 2.04  7.4 0.97 0.95  2.85 0.137  rootrot2.csv
## 30        Total 187 119  9.6 6.1e-01 4.56 69.0 0.99 0.72 14.37 0.271  rootrot2.csv
## 31        Total 100   6  6.0 0.0e+00 1.23  2.8 0.65 0.73  0.05 0.061 simulated.dat
```

You've seen examples of how to use `getfile` to extract a single file and all the files in a directory, but what if you wanted many files only of a certain type or with a certain name? This is what you would use the `pattern` argument for. For example, there are several data files with different formats in the *adegenet* folder in your R library. Let's take a look at the names of these files.

> For the rest of this section, remember that every time you invoke `getfile()`, a window will pop up and you should select a file before hitting enter.

```
getfile(multi = TRUE)
```

Navigate to the *adegenet* folder in your R library.

```
## $files
##  [1] "/path/to/R/adegenet/files/AFLP.txt"
##  [2] "/path/to/R/adegenet/files/exampleSnpDat.snp"
##  [3] "/path/to/R/adegenet/files/mondata1.rda"
##  [4] "/path/to/R/adegenet/files/mondata2.rda"
##  [5] "/path/to/R/adegenet/files/nancycats.dat"
##  [6] "/path/to/R/adegenet/files/nancycats.gen"
##  [7] "/path/to/R/adegenet/files/nancycats.gtx"
##  [8] "/path/to/R/adegenet/files/nancycats.str"
##  [9] "/path/to/R/adegenet/files/pdH1N1-data.csv"
## [10] "/path/to/R/adegenet/files/pdH1N1-HA.fasta"
## [11] "/path/to/R/adegenet/files/pdH1N1-NA.fasta"
## [12] "/path/to/R/adegenet/files/usflu.fasta"
##
## $path
## [1] "/path/to/R/adegenet/files"
```

We can see that we have a mix of files with different formats. If we tried to run all of these files using poppr, we would have a problem because some of the file formats have no direct import into a `genind` object (*.fasta, or *.snp), or just simply are not supported (eg. *.rda files). To filter these files, use the `pattern` argument. Let's say we only wanted the files that have the word "nancy" in them.

```
getfile(multi = TRUE, pattern = "nancy")
```

```
## $files
## [1] "/path/to/R/adegenet/files/nancycats.dat" "/path/to/R/adegenet/files/nancycats.gen"
## [3] "/path/to/R/adegenet/files/nancycats.gtx" "/path/to/R/adegenet/files/nancycats.str"
##
## $path
## [1] "/path/to/R/adegenet/files"
```

Now, let's exclude everything but genetix files (*.gtx).

```
getfile(multi = TRUE, pattern = "gtx")
```

```
## $files
## [1] "/path/to/R/adegenet/files/nancycats.gtx"
##
## $path
## [1] "/path/to/R/adegenet/files"
```

Now, let's only get FSTAT files (*.dat)

```
getfile(multi = TRUE, pattern = "dat")
```

```
## $files
## [1] "/path/to/R/adegenet/files/mondata1.rda"
## [2] "/path/to/R/adegenet/files/mondata2.rda"
## [3] "/path/to/R/adegenet/files/nancycats.dat"
## [4] "/path/to/R/adegenet/files/pdH1N1-data.csv"
##
## $path
## [1] "/path/to/R/adegenet/files"
```

Uh-oh. We've run into a problem. Three out of our four files are not FSTAT files. Why did this happen? It happened because they happen to have `"dat"` within their name. This problem can be solved, by using regular expressions. If you are unfamiliar with regular expressions, you can think of them as special characters that you can use to make your search pattern more strict or more flexible. Since the topic of regular expressions can take up several lectures, I will spare you the gory details. For this situation, the only one you need to know is "`$`". The dollar sign indicates the end of a word or string. If we want specific file extensions all we have to do is add this to the end of the search term like so:

```
getfile(multi = TRUE, pattern = "dat$")
```

```
## $files
## [1] "/path/to/R/adegenet/files/nancycats.dat"
##
## $path
## [1] "/path/to/R/adegenet/files"
```

Now we have our FSTAT file!

1.4.2 Function: read.genalex

A very popular program for population genetics is GenAlEx (http://biology.anu.edu.au/GenAlEx/Welcome.html) [14, 13]. GenAlEx runs within the Excel environment and can be very powerful in its analyses. *Poppr* has added the ability to read *.CSV files[3] produced in the GenAlEx format. It can handle data types containing regions and geographic coordinates, but currently cannot import allelic frequency data from GenAlEx. Using the *poppr* function `read.genalex` will import your data into *adegenet*'s `genind` object or *poppr*'s `genclone` object (more information on that below). For ways of formatting a GenAlEx file, see the manual here: http://biology.anu.edu.au/GenAlEx/Download_files/GenAlEx%206.5%20Guide.pdf

Below is an example of the GenAlEx format. We will use the data set called `microbov` from the *adegenet* package to generate it. The data contains three demographic factors: Country, Species and Breed contained within the `@other` slot (detailed in The other slot). We will combine these and save the file to the desktop. Details of these functions are presented elsewhere in this manual.

```
library(poppr)
data(microbov)
microbov <- as.genclone(microbov)
sethierarchy(microbov) <- data.frame(other(microbov))
setpop(microbov) <- ~coun/breed/spe
genind2genalex(microbov, file = "~/Desktop/microbov.csv")
```

```
## Extracting the table ... Writing the table to ~/Desktop/microbov.csv ... Done.
```

The GenAlEx format contains individuals in rows and loci in columns. Individual data begins at row 4. Column A always contains individual names and column B defines the population of each individual. Notice here that the three demographic factors from the data have been concatenated with a "_". This allows us to import more than one population factor to use as hierarchical levels in a GENCLONE OBJECT.

[3] *.CSV files are comma separated files that are easily machine readable.

Figure 2: The first 15 individuals and 4 loci of the microbov data set. The first column contains the individual names, the second column contains the population names, and each subsequent column represents microsatellite genetic data. Highlighted in red is a list of populations and their relative sizes.

	A	B	C	D	E	F	G	H	I	J
1	30	704	15	50	50	51	30	50	50	47
2	Unmodified Data			AF_BI_Borgou	AF_BI_Zebu	AF_BT_Lagunaire	AF_BT_NDama	AF_BT_Somba	FR_BT_Aubrac	FR_BT_Bazadais
3	Ind	Pop	INRA63		INRA5		ETH225		ILSTS5	
4	AFBIBOR9503	AF_BI_Borgou	183	183	137	141	147	157	190	190
5	AFBIBOR9504	AF_BI_Borgou	181	183	141	141	139	157	186	186
6	AFBIBOR9505	AF_BI_Borgou	177	183	141	141	139	139	194	194
7	AFBIBOR9506	AF_BI_Borgou	183	183	141	141	141	147	184	190
8	AFBIBOR9507	AF_BI_Borgou	177	183	141	141	153	157	184	186
9	AFBIBOR9508	AF_BI_Borgou	177	183	137	143	149	157	184	186
10	AFBIBOR9509	AF_BI_Borgou	177	181	139	141	147	157	184	190
11	AFBIBOR9510	AF_BI_Borgou	183	183	139	141	155	157	184	186
12	AFBIBOR9511	AF_BI_Borgou	177	183	139	141	139	143	182	190
13	AFBIBOR9512	AF_BI_Borgou	183	183	141	141	157	159	186	186
14	AFBIBOR9513	AF_BI_Borgou	177	177	141	141	147	157	184	190
15	AFBIBOR9514	AF_BI_Borgou	183	183	143	143	139	157	186	186
16	AFBIBOR9515	AF_BI_Borgou	183	183	137	137	143	157	0	0
17	AFBIBOR9516	AF_BI_Borgou	177	183	137	143	139	157	182	184
18	AFBIBOR9517	AF_BI_Borgou	177	183	141	141	157	157	186	194

The First three rows contain information pretaining to the global data set. The only important information for *poppr* is the information contained in row 3 and the first three columns of row 1.

	A	**B**	**C**	**D**
1	# of Loci	# of Individuals	# of Populations	Pop1 Size ...
2	-	-	-	Pop1 Name ...
3	-	-	Locus 1	...

Highlighted in red in figure 2 are definitions of the number of populations and their respective sizes. As this is redundant information, we can remove it. Below is an example of a valid data set that can be imported into *poppr*.

Figure 3: The first 15 individuals and 4 loci of the microbov data set. This is the same figure as above, however the populations and counts have been removed from the header row and the third number in the header has been replaced by 1.

	A	B	C	D	E	F	G	H	I	J
1	30	704	1	704						
2	Example Modified	Data		ALL						
3	Ind	Pop	INRA63		INRA5		ETH225		ILSTS5	
4	AFBIBOR9503	AF_BI_Borgou	183	183	137	141	147	157	190	190
5	AFBIBOR9504	AF_BI_Borgou	181	183	141	141	139	157	186	186
6	AFBIBOR9505	AF_BI_Borgou	177	183	141	141	139	139	194	194
7	AFBIBOR9506	AF_BI_Borgou	183	183	141	141	141	147	184	190
8	AFBIBOR9507	AF_BI_Borgou	177	183	141	141	153	157	184	186
9	AFBIBOR9508	AF_BI_Borgou	177	183	137	143	149	157	184	186
10	AFBIBOR9509	AF_BI_Borgou	177	181	139	141	147	157	184	190
11	AFBIBOR9510	AF_BI_Borgou	183	183	139	141	155	157	184	186
12	AFBIBOR9511	AF_BI_Borgou	177	183	139	141	139	143	182	190
13	AFBIBOR9512	AF_BI_Borgou	183	183	141	141	157	159	186	186
14	AFBIBOR9513	AF_BI_Borgou	177	177	141	141	147	157	184	190
15	AFBIBOR9514	AF_BI_Borgou	183	183	143	143	139	157	186	186
16	AFBIBOR9515	AF_BI_Borgou	183	183	137	137	143	157	0	0
17	AFBIBOR9516	AF_BI_Borgou	177	183	137	143	139	157	182	184
18	AFBIBOR9517	AF_BI_Borgou	177	183	141	141	157	157	186	194

All GenAlEx formatted data can be imported with the command `read.genalex`, detailed below:

Default Command:

```
read.genalex(genalex, ploidy = 2, geo = FALSE, region = FALSE,
    genclone = TRUE, sep = ",")
```

- `genalex` - a *.CSV file exported from GenAlEx on your disk (For example: `"my_genalex_file.csv"`).
- `ploidy` - a number indicating the ploidy for the data set (eg 2 for diploids, 1 for haploids).
- `geo` - GenAlEx allows you to have geographic data within your file. To do this for *poppr*, you will need to follow the first format outlined in the GenAlEx manual and place the geographic data AFTER all genetic and demographic data with one blank column separating it (See the GenAlEx Manual for details). If you have geographic information in your file, set this flag to `TRUE` and it will be included within the resulting genind object in the `@other` slot. (If you don't know what that is, don't worry. It will be explained later in The other slot.)
- `region` - To format your GenAlEx file to include regions, you can choose to include a separate column for regional data, or, since regional data must be in contiguous blocks, you can simply format it in the same way you would any other data (see the GenAlEx manual for details). If you have your file organized in this manner, select this option and the regional information will be stored in the `@other` slot of the resulting genind object or be incorporated into the hierarchy of the genclone object.
- `genclone` - This flag will convert your data into a `genclone` object (see Send in the clones for more info).
- `sep` - The separator argument for columns in your data. It defaults to ",".

If you are unfamiliar with exporting data from Excel

1. Click the Microsoft Office Button in the top left corner of Excel. (Or go to the File menu if you have an older version)
2. Click Save As...
3. In the "Save as type" drop down box, select CSV (comma delimited).

Note that regional data and geographic data are not mutually exclusive. You can have both in one file, just make sure that they are on the same sheet and that the geographic data is always placed after all genetic and demographic data.

We have a short example of genalex formatted data with no geographic or regional formatting. We will first see where the data is using the command `system.file()`

```
system.file("files/rootrot.csv", package = "poppr")
```

```
## [1] "/path/to/R/library/poppr/files/rootrot.csv"
```

Now import the data into *poppr* like so:

```
rootrot <- read.genalex(system.file("files/rootrot.csv", package = "poppr"))
```

Executing rootrot shows that this file is now in genclone format and can be used with any function in *poppr* and *adegenet*

```
rootrot

##
## This is a genclone object
## -------------------------
## Genotype information:
##
##    119 multilocus genotypes
##    187 diploid individuals
##     56 dominant loci
##
## Population information:
##
##      1 hierarchical level - Pop
##     18 populations defined - Athena_1 Athena_2 Athena_3 ... Mt. Vernon_6 Mt. Vernon_7
## Mt. Vernon_8
```

1.4.3 Other ways of importing data

Adegenet already supports the import of FSTAT, Structure, Genpop, and Genetix formatted files, so if you have data in those formats, you can import them using the function `import2genind`. For sequence data, check if you can use `read.dna` from the *ape* package to import your data. If you can, then you can use the *adegenet* function `DNAbin2genind`. If you don't have any of these formats handy, you can still import your data using R's `read.table` along with `df2genind` from *adegenet*. For more information, see *adegenet*'s "Getting Started" vignette.

1.4.4 Function: genind2genalex

Of course, being able to export data is just as useful as being able to import it, so we have this handy little function that will write a GenAlEx formatted file to wherever you desire.
WARNING: This will overwrite any file that exists with the same name.

Default Command:

```
genind2genalex(pop, filename = "genalex.csv", quiet = FALSE,
    geo = FALSE, geodf = "xy", sep = ",")
```

- `pop` - a `genind` object.
- `filename` - This is where you specify the path to the new file you wish to create. If you specify only a filename with no path, it will place the file in your current working directory.
- `quiet` - If this is set to `FALSE`, a status message will be printed to the console as the extraction progresses.
- `geo` - Set to `TRUE`, if you have a data frame or matrix in the `@other` slot of your genind object that contains geographic coordinates for all individuals or all populations. Setting this to `TRUE` means the resulting file will have two extra columns at the end of your file with geographic coordinates.
- `geodf` - The name of the data frame or matrix containing the geographic coordinates.

- `sep` - A separator to separate columns in the resulting file.

First, a simple example for the rootrot data we demonstrated in section 1.4.2:

```
genind2genalex(rootrot, "~/Desktop/rootrot.csv")
```

```
## Extracting the table ... Writing the table to ~/Desktop/rootrot.csv ... Done.
```

Here's an example of exporting the nancycats data set into GenAlEx format with geographic information. If we look at the nancycats geographic information, we can see it's coordinates for each population, but not each individual:

```
data(nancycats)
nancycats@other$xy
```

```
##         x      y
## P01 263.3 171.11
## P02 183.5 122.41
## P03 391.1 254.70
## P04 458.6  41.72
## P05 182.8 219.08
## P06 335.2 344.84
## P07 359.2 375.36
## P08 271.3  67.89
## P09 256.8 150.03
## P10 270.6  17.01
## P11 493.5 237.26
## P12 305.5  85.34
## P13 463.0  86.79
## P14 429.6 291.05
## P15 531.2 115.14
## P16 407.8  99.87
## P17 345.4 251.79
```

To export it:

```
genind2genalex(nancycats, "~/Desktop/nancycats_pop_xy.csv", geo = TRUE)
```

```
## Extracting the table ... Writing the table to ~/Desktop/nancycats_pop_xy.csv ... Done.
```

If we wanted to assign a geographic coordinate to each individual, we can use this trick knowing that there are 17 populations in the data set:

```
nan2 <- nancycats
nan2@other$xy <- nan2@other$xy[rep(1:17, table(pop(nan2))), ]
head(nan2@other$xy)
```

```
##         x     y
## P01 263.3 171.1
## P01 263.3 171.1
## P01 263.3 171.1
## P01 263.3 171.1
## P01 263.3 171.1
## P01 263.3 171.1
```

Now we can export it to a different file.

```
genind2genalex(nan2, "~/Desktop/nancycats_inds_xy.csv", geo = TRUE)
```

```
## Extracting the table ... Writing the table to ~/Desktop/nancycats_inds_xy.csv ... Done.
```

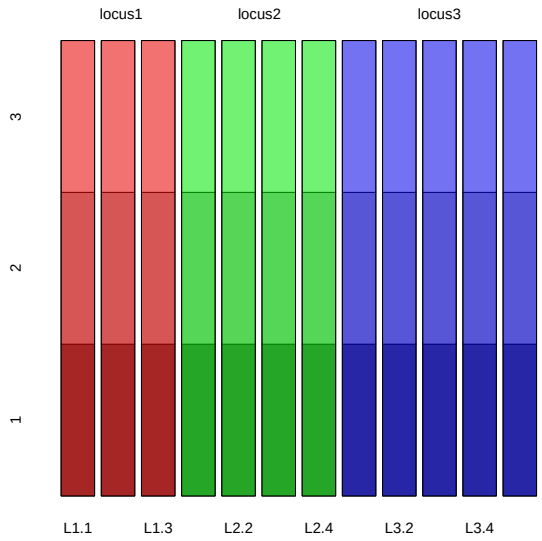
1.5 Getting to know *adegenet*'s genind object

Since *poppr* was built around *adegenet*'s framework, it is important to know how *adegenet* stores data in the genind object, as that is the object used by *poppr*. To create a genind object, *adegenet* takes a data frame of genotypes (rows) across multiple loci (columns) and converts them into a matrix of individual allelic frequencies at each locus [9].

For example, Let's say we had data with 3 diploid individuals each with 3 loci that had 3, 4, and 5 allelic states respectively:

	locus1	locus2	locus3
1	101/101	201/201	301/302
2	102/103	202/203	301/303
3	102/102	203/204	304/305

The resulting `genind` object would contain a matrix that has 3 rows and 12 columns. Below is a schematic of what that would look like. Each column represents a separate allele, each row represents an individual and each color represents a different locus.



When we look at the data derived from table 1.5, we see that we have a matrix of individual allele frequencies at each locus.

```
##   L1.1 L1.2 L1.3 L2.1 L2.2 L2.3 L2.4 L3.1 L3.2 L3.3 L3.4 L3.5
## 1    1  0.0  0.0    1  0.0  0.0  0.0  0.5  0.5  0.0  0.0  0.0
## 2    0  0.5  0.5    0  0.5  0.5  0.0  0.5  0.0  0.5  0.0  0.0
## 3    0  1.0  0.0    0  0.0  0.5  0.5  0.0  0.0  0.0  0.5  0.5
```

At each locus, the allele frequencies for each individual sum to one. Homozygotes are denoted as having an allele frequency of 1 at a particular allele while heterozygotes have their allele frequencies represented as $1/p$ wher p = ploidy. Along with this matrix, are elements that define the names of the individuals, loci, alleles, and populations. If you wish to know more, see the *adegenet* "Getting Started" manual.

1.5.1 The other slot

The other slot is a place in the genind object that can be used to store useful information about the data. We saw earlier that it could store demographic information, now let's explore a different example. Bruvo's distance is based off of a stepwise mutation model for microsatellites. This requires us to know the length of the repeat of each locus. We could store the repeat lengths in a separate variable in our R environment, but we are at risk of losing that. One wa to prevent it from being lost would be to place it in the "other" slot. For the purpose of this example, we will use the "nancycats" data set from the *adegenet* package and assume that it has dinucleotide repeats at all of its loci.

```
data(nancycats)  # Load the data
other(nancycats)  # geographical coordinates

## $xy
##         x      y
## P01 263.3 171.11
## P02 183.5 122.41
## P03 391.1 254.70
## P04 458.6  41.72
## P05 182.8 219.08
## P06 335.2 344.84
## P07 359.2 375.36
## P08 271.3  67.89
## P09 256.8 150.03
## P10 270.6  17.01
## P11 493.5 237.26
## P12 305.5  85.34
## P13 463.0  86.79
## P14 429.6 291.05
## P15 531.2 115.14
## P16 407.8  99.87
## P17 345.4 251.79

repeats <- rep(2, nLoc(nancycats))  #nLoc = number of loci
repeats

## [1] 2 2 2 2 2 2 2 2 2

other(nancycats)$repeat_lengths <- repeats
other(nancycats)  # two items named xy and repeat_lengths

## $xy
##         x      y
## P01 263.3 171.11
## P02 183.5 122.41
## P03 391.1 254.70
## P04 458.6  41.72
## P05 182.8 219.08
## P06 335.2 344.84
## P07 359.2 375.36
## P08 271.3  67.89
## P09 256.8 150.03
## P10 270.6  17.01
## P11 493.5 237.26
## P12 305.5  85.34
## P13 463.0  86.79
## P14 429.6 291.05
## P15 531.2 115.14
## P16 407.8  99.87
## P17 345.4 251.79
##
## $repeat_lengths
## [1] 2 2 2 2 2 2 2 2 2
```

1.6 The genclone object {send in the clones}

The `genclone` class was defined in order to make working with hierarchies more intuitive. It is built off of the `genind` object and has dedicated slots for the population hierarchy and defined multilocus genotypes. The name genclone refers to the fact that it has the ability to handle genotypes of clonal organisms (but it is also used for sexual populations).

The main difference between the genclone and genind objects is how they handle populations: With the `genind` object, the user must find the vector defining the population and set it using that vector. The `genclone` object already defines a data frame with different population factors or hierarchical levels in the object. The user simply supplies a formula defining the desired hierarchy with which to set the population. This formula driven method is also used for clone correction, combining hierarchical levels and conducting AMOVA. These will all be explained in later chapters. For examples and details, type `help("genclone")` in your R console.

The function `as.genclone` allows the user to convert a `genind` object to a `genclone` object. The following example will demonstrate that the `genclone` object is an extention of the `genind` object as well as the advantages of having populations pre-defined in your data set.

1.6.1 Function: as.genclone

Default Command:

```
as.genclone(x, hierarchy = NULL)
```

- **x -** a `genind` object to be converted.
- **hierarchy -** an optional data frame where each column represents a hierarchical level of the population hierarchy in the data set.

Let's show an example of a `genclone` object. First, we will take an existing `genind` object and convert it using the function `as.genclone` (We can also use the function `read.genalex` to import as `genclone` or `genind` objects). We will use the `Aeut` data set because it is a clonal data set that has a simple population hierarchy [5]. The data set is here: http://dx.doi.org/10.6084/m9.figshare.877104 and it is AFLP data of the root rot pathogen *Aphanomyces euteiches* collected from two different fields in NW Oregon and W Washington, USA. These fields were divided up into subplots from which samples were collected. The fields represent the population and the subplots represent the subpopulation. Let's take a look at what the `genind` object looks like:

```
library(poppr)
data(Aeut)
Aeut

##
##    ######################
##    ### Genind object ###
##    ######################
## - genotypes of individuals -
##
## S4 class:  genind
## @call: read.genalex(genalex = "rootrot.csv")
##
## @tab:  187 x 56 matrix of genotypes
```

```
##
## @ind.names: vector of  187 individual names
## @loc.names: vector of  56 locus names
## @loc.nall: NULL
## @loc.fac: NULL
## @all.names: NULL
## @ploidy:  2
## @type:  PA
##
## Optionnal contents:
## @pop:  factor giving the population of each individual
## @pop.names:  factor giving the population of each individual
##
## @other: a list containing: population_hierarchy
```

This gives us a lot of information about the object, and is useful once you become more comfortable with programming. Once we convert it to a `genclone` object, the multilocus genotypes will be defined and the population hierarchy (if a data frame is defined in the `@other` slot called "population_hierarchy") will be set.

```
agc <- as.genclone(Aeut)
agc

##
## This is a genclone object
## -------------------------
## Genotype information:
##
##    119 multilocus genotypes
##    187 diploid individuals
##     56 dominant loci
##
## Population information:
##
##      3 hierarchical levels - Pop_Subpop Pop Subpop
##      2 populations defined - Athena Mt. Vernon

# We can also manually set the hierarchy.
as.genclone(Aeut, hierarchy = other(Aeut)$population_hierarchy[-1])

##
## This is a genclone object
## -------------------------
## Genotype information:
##
##    119 multilocus genotypes
##    187 diploid individuals
##     56 dominant loci
##
## Population information:
##
##      2 hierarchical levels - Pop Subpop
##      2 populations defined - Athena Mt. Vernon
```

We can see here that it shows less information, but it gives us a very simple overview of our data. Don't be fooled, however, because it contains the same information as a `genind` object and the advantage of the `genclone` object is that setting the population from the hierarchy becomes **much** easier.

```
c(is.genind(Aeut), is.genclone(Aeut), is.genind(agc), is.genclone(agc))

## [1]  TRUE FALSE  TRUE  TRUE

# Adegenet functions work the same, too
c(nInd(Aeut), nInd(agc))
```

```
## [1] 187 187
```

```
# We'll look at the population names
Aeut$pop.names

##           P1           P2
##     "Athena" "Mt. Vernon"

# genind way: Extract the combined hierarchical levels.
pophier <- other(Aeut)$population_hierarchy$Pop_Subpop
pop(Aeut) <- pophier
Aeut$pop.names

##  [1] "Athena_1"     "Athena_2"     "Athena_3"     "Athena_4"     "Athena_5"
##  [6] "Athena_6"     "Athena_7"     "Athena_8"     "Athena_9"     "Athena_10"
## [11] "Mt. Vernon_1" "Mt. Vernon_2" "Mt. Vernon_3" "Mt. Vernon_4" "Mt. Vernon_5"
## [16] "Mt. Vernon_6" "Mt. Vernon_7" "Mt. Vernon_8"

# genclone way:
agc

##
## This is a genclone object
## -------------------------
## Genotype information:
##
##    119 multilocus genotypes
##    187 diploid individuals
##     56 dominant loci
##
## Population information:
##
##      3 hierarchical levels - Pop_Subpop Pop Subpop
##      2 populations defined - Athena Mt. Vernon

setpop(agc) <- ~Pop/Subpop
agc

##
## This is a genclone object
## -------------------------
## Genotype information:
##
##    119 multilocus genotypes
##    187 diploid individuals
##     56 dominant loci
##
## Population information:
##
##      3 hierarchical levels - Pop_Subpop Pop Subpop
##     18 populations defined - Athena_1 Athena_2 Athena_3 ... Mt. Vernon_6 Mt. Vernon_7
## Mt. Vernon_8

# Notice that you only see the first and last three population names.  Use the print
# function to display all population names.

print(agc)

##
## This is a genclone object
## -------------------------
## Genotype information:
##
```

```
##    119 multilocus genotypes
##    187 diploid individuals
##     56 dominant loci
##
## Population information:
##
##      3 hierarchical levels - Pop_Subpop Pop Subpop
##     18 populations defined - Athena_1 Athena_2 Athena_3 Athena_4 Athena_5 Athena_6
## Athena_7 Athena_8 Athena_9 Athena_10 Mt. Vernon_1 Mt. Vernon_2 Mt. Vernon_3 Mt. Vernon_4
## Mt. Vernon_5 Mt. Vernon_6 Mt. Vernon_7 Mt. Vernon_8
```

1.7 Accessing the population hierarchy

The hierarchy slot in the genclone object allows us to access and manipulate different levels of a population hierarchy without the complication of creating new data sets. We implemented the use of hierarchical nested formulae:

```
hier = ~Population/Subpopulation/Year
```

NOTE: The ~ symbol is absolutely required for formulas, even if you only are specifying one level.

The `"/"` symbolizes a hierarchical nesting. The above formula represents years nested within subpopulations, nested within populations. This allows the user to easily restructure the population hierarchy. Manipulation of the hierarchy is necessary for CLONE CORRECTION and AMOVA analysis. See the section CAN YOU TAKE ME HIER(ARCHY) for more details on manipulation of hierarchies.

2 Data Manipulation

One tedious aspect of population genetic analysis is the need for repeated data manipulation. *Adegenet* has some functions for manipulating data that are limited to replacing missing data and dividing data into populations, loci, or by sample size [9]. *Poppr* includes novel functions for clone-censoring your data sets or sub-setting a genind object by specific populations.

2.1 Replace or remove missing data {Inside the golden days of missing data}

A data set without missing data is always ideal, but often not achievable. Many functions in *adegenet* cannot handle missing data and thus the function `na.replace` exists [9]. It will replace missing data with with either "0" representing a mysterious extra allele in the data set resulting in more diversity or the mean of allelic frequencies at the locus. There is no set method, however, for simply removing missing data from analyses, which is why the *poppr* function `missingno` (see below) exists. If the name makes you uneasy it's because it should. Missing data can mean different things based on your data type. For microsatellites, missing data might represent any source of error that could cause a PCR product to not amplify in gel electrophoresis, which may or may not be biologically relevant. For a DNA alignment, missing data could mean something as simple as an insertion or deletion, which is biologically relevant. The choice to exclude or estimate data has very different implications for the type of data you have.

2.1.1 Function: missingno

`missingno` is a function that serves partially as a wrapper for *adegenet*'s `na.replace` to replace missing data and as a way to exclude specific areas that contain systematic missing data.

Default Command:

```
missingno(pop, type = "loci", cutoff = 0.05, quiet = FALSE)
```

- `pop` - a `genind` object.
- `type` - This could be one of four options:

  "mean" This replaces missing data with the mean allele frequencies in the entire data set.

  "zero" or "0" This replaces missing data with zero, signifying a new allele.

  "loci" This is to be used for a data set that has systematic problems with certain loci that contain null alleles or simply failed to amplify. This will remove loci with a defined threshold of missing data from the data set.

  "geno" This is to be used for genotypes (individuals) in your data set where many null alleles are present. Individuals with a defined threshold missing data will be removed.

- `cutoff` - This is a numeric value from 0 to 1 indicating the percent allowable missing data for either loci or genotypes. If you have, for example, two loci containing missing 5% and 10% missing data, respectively and you set `cutoff = 0.05`, `missingno` will remove the second locus. Percent missing data for genotypes is considered the percent missing loci over number of total loci.
- `quiet` - When this is set to `FALSE`, the number of missing values replaced will be printed to screen if the method is "zero" or "mean". It will print the number of loci or individuals removed if the method is "loci" or "geno".

Of course, seeing is believing. Let's take a look at what this does by focusing in on areas with missing data. Note that I will be using some sub-setting functions here that are described in *adegenet*'s *Getting Started* vignette. First, let's take a look at what the missing data in R looks like as well as how many loci and individuals the data set nancycats contains. We need to first tell R to look in its library for the package *poppr*.

```
library(poppr)
```

Next, we'll initialize the *adegenet* data set nancycats and load it into memory.

```
data(nancycats)
```

Now, we'll take a quick look at the nancycats data set using *adegenet*'s `summary()` function:

```
summary(nancycats)

##
##  # Total number of genotypes:  237
##
##  # Population sample sizes:
##  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
## 10 22 12 23 15 11 14 10  9 11 20 14 13 17 11 12 13
##
##  # Number of alleles per locus:
## L1 L2 L3 L4 L5 L6 L7 L8 L9
## 16 11 10  9 12  8 12 12 18
##
##  # Number of alleles per population:
## 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17
## 36 53 50 67 48 56 42 54 43 46 70 52 44 61 42 40 35
##
##  # Percentage of missing data:
```

```
## [1] 2.344
##
##  # Observed heterozygosity:
##     L1     L2     L3     L4     L5     L6     L7     L8     L9
## 0.6682 0.6667 0.6793 0.7083 0.6329 0.5654 0.6498 0.6184 0.4515
##
##  # Expected heterozygosity:
##     L1     L2     L3     L4     L5     L6     L7     L8     L9
## 0.8657 0.7929 0.7953 0.7603 0.8703 0.6885 0.8158 0.7603 0.6063
```

We can see here a lot of summary statistics about nancycats. Here we can see that there are 17 populations, 237 individuals, and 9 loci. Nancycats also has a little over 2.3% missing data. Let's take a look at the names of the loci and the structure of the data. In order to save space, I will only show you the first five individuals (rows) and a portion of the alleles in the first locus (columns).

```
locNames(nancycats)  # Names of the loci

##      L1      L2      L3      L4      L5      L6      L7      L8      L9
##  "fca8" "fca23" "fca43" "fca45" "fca77" "fca78" "fca90" "fca96" "fca37"

nancycats$tab[1:5, 8:13]

##     L1.08 L1.09 L1.10 L1.11 L1.12 L1.13
## 001    NA    NA    NA    NA    NA    NA
## 002    NA    NA    NA    NA    NA    NA
## 003   0.0   0.5     0     0     0   0.5
## 004   0.5   0.5     0     0     0   0.0
## 005   0.5   0.5     0     0     0   0.0
```

When looking at this data set, recall how a `genind` object is formatted. You have a matrix of 0's, 1's and 0.5's. For diploids, if you see 0.5, that means it is heterozygous at that allele, and a 1 means it's homozygous. Here we see three heterozygotes and two individuals with missing data (indicated by NA). Now, there are more places with missing data in the data set, but I'm only showing a little bit at one locus so it's easier to digest. Let's first replace it by zero and mean, respectively.

```
nanzero <- missingno(nancycats, type = "zero")

##
##  Replaced 617 missing values

nanmean <- missingno(nancycats, type = "mean")

##
##  Replaced 617 missing values

nanzero$tab[1:5, 8:13]

##     L1.08 L1.09 L1.10 L1.11 L1.12 L1.13
## 001   0.0   0.0     0     0     0   0.0
## 002   0.0   0.0     0     0     0   0.0
## 003   0.0   0.5     0     0     0   0.5
## 004   0.5   0.5     0     0     0   0.0
## 005   0.5   0.5     0     0     0   0.0

nanmean$tab[1:5, 8:13]

##       L1.08  L1.09  L1.10   L1.11   L1.12  L1.13
## 001 0.07604 0.2419 0.1912 0.06221 0.09447 0.1014
## 002 0.07604 0.2419 0.1912 0.06221 0.09447 0.1014
## 003 0.00000 0.5000 0.0000 0.00000 0.00000 0.5000
## 004 0.50000 0.5000 0.0000 0.00000 0.00000 0.0000
## 005 0.50000 0.5000 0.0000 0.00000 0.00000 0.0000
```

You notice how the values of NA changed, yet the basic structure stayed the same. These are the replacement options from *adegenet*. Let's look at the same example with the exclusion options (set to the default cutoff of 5%).

```
nanloci <- missingno(nancycats, "loci")

##
##  Found 617 missing values.
##  2 loci contained missing values greater than 5%.
##  Removing 2 loci : fca8 fca45

nangeno <- missingno(nancycats, "geno")

##
##  Found 617 missing values.
##  38 genotypes contained missing values greater than 5%.
##  Removing 38 genotypes : N215 N216 N188 N189 N190 N191 N192 N298 N299 N300
## N301 N302 N303 N304 N310 N195 N197 N198 N199 N200 N201 N206 N182 N184 N186 N282
## N283 N288 N291 N292 N293 N294 N295 N296 N297 N281 N289 N290

nanloci$tab[1:5, 8:13]

##     L1.08 L1.09 L1.10 L1.11 L2.01 L2.02
## 001     0   0.5     0     0     0     0
## 002     0   1.0     0     0     0     0
## 003     0   0.5     0     0     0     0
## 004     0   0.0     0     0     0     0
## 005     0   0.5     0     0     0     0
```

Notice how we now see columns named "L2.01" and "L2.02". This is showing us another locus because we have removed the first. Recall from the summary table that the first locus had 16 alleles, and the second had 11. Now that we've removed loci containing missing data, all others have shifted over.
Let's look at the loci names and number of individuals.

```
nInd(nanloci)  # Individuals

## [1] 237

locNames(nanloci)  # Names of the loci

##      L1      L2      L3      L4      L5      L6      L7
## "fca23" "fca43" "fca77" "fca78" "fca90" "fca96" "fca37"
```

You can see that the number of individuals stayed the same but the loci "fca8", "fca45", and "fca96" were removed.
Let's look at what happened when we removed individuals.

```
nangeno$tab[1:5, 8:13]

##     L1.08 L1.09 L1.10 L1.11 L1.12 L1.13
## 001   0.0   0.5     0     0     0   0.5
## 002   0.5   0.5     0     0     0   0.0
## 003   0.5   0.5     0     0     0   0.0
## 004   0.0   0.5     0     0     0   0.5
## 005   0.0   1.0     0     0     0   0.0

nInd(nangeno)  # Individuals

## [1] 199

locNames(nangeno)  # Names of the loci

##      L1      L2      L3      L4      L5      L6      L7      L8      L9
##  "fca8" "fca23" "fca43" "fca45" "fca77" "fca78" "fca90" "fca96" "fca37"
```

We can see here that the number of individuals decreased, yet we have the same number of loci. Notice how the frequency matrix changes in both scenarios? In the scenario with "loci", we removed several columns of the data set, and so with our sub-setting, we see alleles from the second locus. In the scenario with "geno", we removed several rows of the data set so we see other individuals in our sub-setting.

2.2 Population hierarchy construction {Can you take me hier(archy)?}

in *poppr*, the GENCLONE object contains a slot called `hierarchy`. This slot contains a data frame used to define hierarchical levels of population factors describing your data. The preferred way of defining these hierarchical levels is to concatenate them using '_' and use them to define a single population in your data before you import it into *poppr*. Examples of this format can be found in figures 2, 3, and at http://dx.doi.org/10.6084/m9.figshare.877104.

In this section, we will show you how to DEFINE hierarchical levels, VIEW those levels to ensure that they are correctly defined, MANIPULATE your hierarchical levels by adding and renaming them, and use these levels to SET THE POPULATION in your GENCLONE object using the following methods:

Method	Function	Input	Result
split	`splithierarchy`	formula	defined hierarchical levels
set	`sethierarchy`	data frame	new hierarchical levels
get	`gethierarchy`	formula	data frame
name	`namehierarchy`	formula	new hierarchical level names
add	`addhierarchy`	vector or data frame	new hierarchical level

A NOTE ABOUT FORMULAS
The formulas used by genclone objects always start with a ~ and are hierarchical levels are always separated by a /. Some examples are:
~Country/City/District
~Field/Year
Refer to ACCESSING HIERARCHIES for more details on how to access hierarchies.

In the next section, we'll explore two ways of defining hierarchical levels.

2.2.1 Defining hierarchies

As explained above, the best way to define hierarchical levels is to concatenate them using '_' and set that as your population factor. We will use the example data set from http://dx.doi.org/10.6084/m9.figshare.877104. It is an AFLP data set of the root rot pathogen *Aphanomyces euteiches* from two fields and multiple soil cores per field. First, we will follow the link and copy the download link from figshare.

```
aphan <- read.genalex("http://files.figshare.com/1314228/rootrot.csv")
aphan
```

```
##
## This is a genclone object
## -------------------------
## Genotype information:
##
##    119 multilocus genotypes
##    187 diploid individuals
##     56 dominant loci
##
## Population information:
##
##      1 hierarchical level - Pop
##     18 populations defined - Athena_1 Athena_2 Athena_3 ... Mt. Vernon_6 Mt. Vernon_7
## Mt. Vernon_8
```

The supplemental information in the data defined two hierarchical leves, yet we only see one here labeled 'Pop'. This is how populations are automatically defined when importing to a GENCLONE object since it does not know how many hierarchical levels you have defined. To define these levels present in the data set, we will need to split them up using the function `splithierarchy`:

```
splithierarchy(aphan) <- ~field/sample
aphan

##
## This is a genclone object
## -------------------------
## Genotype information:
##
##    119 multilocus genotypes
##    187 diploid individuals
##     56 dominant loci
##
## Population information:
##
##      2 hierarchical levels - field sample
##     18 populations defined - Athena_1 Athena_2 Athena_3 ... Mt. Vernon_6 Mt. Vernon_7
## Mt. Vernon_8
```

Now we have sucessfully defined our hierarchies. If you have imported your data in this manner, you may skip to the SETTING POPULATION HIERARCHIES , VIEWING HIERARCHIES , or MANIPULATING HIERARCHICAL LEVELS.

If you have imported your data with a single population and want to add hierarchical levels separately, you can use the function `sethierarchy` with a data frame containing your hierarchical levels. For this example, we will use the data set `H3N2`, which contains SNP data from the H3N2 virus. This data set holds a data frame in the the OTHER SLOT that contains many variables including country, year, and month of collection. We will first load the data and write that data frame to a file on the desktop.

```
data(H3N2)
write.table(other(H3N2)$x, file = "~/Desktop/virus_info.csv", row.names = FALSE)
```

Now we have our data and we have a separate table in a file on our desktop defining our hierarchical levels. Let's import those levels into R with `read.table` and see what they are:

```
virus_info <- read.table("~/Desktop/virus_info.csv", header = TRUE)
names(virus_info)
```

```
##  [1] "accession"     "length"        "host"          "segment"       "subtype"
##  [6] "country"       "year"          "Virus name"    "Misc info"     "Age"
## [11] "Gender"        "date"          "usePreciseLoc" "localisation"  "lon"
## [16] "lat"           "month"
```

From here we will convert our genind object to a genclone object and use `sethierarchy` to define the hierarchical levels with the table we just imported.

```
virus <- as.genclone(H3N2)  # Converting it to a genclone object.
sethierarchy(virus) <- virus_info  # Setting the hierarchy
virus

##
## This is a genclone object
## -------------------------
## Genotype information:
##
```

```
##     752 multilocus genotypes
##    1903 haploid individuals
##     125 codominant loci
##
## Population information:
##
##      17 hierarchical levels - accession length host ... lon lat month
##       0 populations defined.
```

In this data, levels such as host and segment are unimportant levels because they are all the same. Let's say that we are only interested in year and country. To make things easier to view, we will set the hierarchical levels to these two columns:

```
sethierarchy(virus) <- virus_info[c("country", "year")]
virus
```

```
##
## This is a genclone object
## -------------------------
## Genotype information:
##
##     752 multilocus genotypes
##    1903 haploid individuals
##     125 codominant loci
##
## Population information:
##
##       2 hierarchical levels - country year
##       0 populations defined.
```

Notice that there are no populations defined. Now that we have the hierarchical levels in place, we can use it to define the population hierarchy. We will use the function `setpop` to define the population as year with respect to country:

```
setpop(virus) <- ~year/country
virus
```

```
##
## This is a genclone object
## -------------------------
## Genotype information:
##
##     752 multilocus genotypes
##    1903 haploid individuals
##     125 codominant loci
##
## Population information:
##
##       2 hierarchical levels - country year
##     102 populations defined - 2002_Japan 2002_USA 2002_Finland ... 2006_Algeria
## 2006_Sweden 2006_Greece
```

2.2.2 Viewing hierarchies

If you wanted to view your hierarchies to make sure that you made no spelling errors in your population definitions, you can extract the data frame from your genclone object by using the function `gethierarchy`:

Default Command:

```
gethierarchy(x, formula = NULL, combine = TRUE)
```

Where **x** is the genclone object, **formula** defines the hierarchical levels, and **combine** indicates whether or not you want the lower levels of the hierarchy combined with the higher levels. For example, in the root rot data above, the hierarchical levels are explicitly hierarchical and should be combined. Note, if you don't supply a formula argument, the data frame as it exists will be returned.

```
head(gethierarchy(aphan))
```

```
##    field sample
## 1 Athena      1
## 2 Athena      1
## 3 Athena      1
## 4 Athena      1
## 5 Athena      1
## 6 Athena      1
```

```
head(gethierarchy(aphan, ~field/sample))
```

```
##    field   sample
## 1 Athena Athena_1
## 2 Athena Athena_1
## 3 Athena Athena_1
## 4 Athena Athena_1
## 5 Athena Athena_1
## 6 Athena Athena_1
```

If the hierarchical levels are not nested, or you simply want to use this hierarchy for another data set, you might want to set the **combine** flag to `FALSE`. Let's use the virus data as an example:

```
head(gethierarchy(virus, ~year/country))
```

```
##   year    country
## 1 2002 2002_Japan
## 2 2002 2002_Japan
## 3 2002 2002_Japan
## 4 2002 2002_Japan
## 5 2002 2002_Japan
## 6 2002 2002_Japan
```

```
head(gethierarchy(virus, ~year/country, combine = FALSE))
```

```
##   year country
## 1 2002   Japan
## 2 2002   Japan
## 3 2002   Japan
## 4 2002   Japan
## 5 2002   Japan
## 6 2002   Japan
```

It will return only the levels you ask it to return:

```
head(gethierarchy(virus, ~country))
```

```
##   country
## 1   Japan
## 2   Japan
## 3   Japan
## 4   Japan
## 5   Japan
## 6   Japan
```

2.2.3 Manipulating hierarchical levels

Once we have our hierarchies set in place, we want to be able to rename and add to them. For this example, we will revisit the VIRUS EXAMPLE from above. We have set the population hierarchy to be based on year and country, but we've noticed that we left out month. And let's say that we accidentally overwrote the data object like this:

```
virus_info <- virus_info[["month"]]
names(virus_info)

## NULL
```

If we were saving our script the whole time, we could just go back and retrieve the data frame, but that defeats the purpose of this section where we imagine that we've recieved new information and wanted to add it to our hierarchy. If we want to add this to our hierarchy, we just use the function `addhierarchy` defined as thus:

Default Command:

```
addhierarchy(x, value, name = "NEW")
```

We can use this function to add on a new column to the data frame.

```
addhierarchy(virus) <- virus_info
virus

##
## This is a genclone object
## -------------------------
## Genotype information:
##
##     752 multilocus genotypes
##    1903 haploid individuals
##     125 codominant loci
##
## Population information:
##
##       3 hierarchical levels - country year NEW
##     102 populations defined - 2002_Japan 2002_USA 2002_Finland ... 2006_Algeria
## 2006_Sweden 2006_Greece
```

Notice that the new hierarchical level is simply labeled as `NEW`. We will customize the name of the hierarchical levels with the function `namehierarchy`.

```
namehierarchy(virus) <- ~country/year/month
virus

##
## This is a genclone object
## -------------------------
## Genotype information:
##
##     752 multilocus genotypes
##    1903 haploid individuals
##     125 codominant loci
##
## Population information:
##
##       3 hierarchical levels - country year month
##     102 populations defined - 2002_Japan 2002_USA 2002_Finland ... 2006_Algeria
## 2006_Sweden 2006_Greece
```

Of course, perhaps a better way still would be to use a data frame:

```
addhierarchy(virus) <- data.frame(month = virus_info)
```

2.2.4 Defining populations with hierarchies

Now that we have defined the hierarchical levels in the data set, setting the population hierarchy allows us to group our data according to the hierarchical level of your choice. This is a necessary step. For this example, we will use a data set of *Phytophthora infestans* collected from North America and South America.

```
data(Pinf)
Pinf

##
## This is a genclone object
## -------------------------
## Genotype information:
##
##    72 multilocus genotypes
##    86 tetraploid individuals
##    11 codominant loci
##
## Population information:
##
##     2 hierarchical levels - Continent Country
##     2 populations defined - South America North America
```

Above we have two hierarchies for Continent and Country, but the populations only show Continent level populations. If we wanted to investigate each country separately, we would need to reset the population to be represented by Country. This can be done with the function `setpop`. This function utilizes the defined population hierarchies to set the population. We'll use our data set above to illustrate this:

```
setpop(Pinf) <- ~Country
Pinf  # Now set by country

##
## This is a genclone object
## -------------------------
## Genotype information:
##
##    72 multilocus genotypes
##    86 tetraploid individuals
##    11 codominant loci
##
## Population information:
##
##     2 hierarchical levels - Continent Country
##     4 populations defined - Colombia Ecuador Mexico Peru
```

The beauty about it is the fact that it will also combine all the hierarchical levels you want to use. Let's see when we ask it to set the population of Country with respect to Continent.

```
setpop(Pinf) <- ~Continent/Country
Pinf

##
## This is a genclone object
## -------------------------
## Genotype information:
##
```

```
##    72 multilocus genotypes
##    86 tetraploid individuals
##    11 codominant loci
##
## Population information:
##
##     2 hierarchical levels - Continent Country
##     4 populations defined - South America_Colombia South America_Ecuador
## North America_Mexico South America_Peru
```

Nice!

2.3 Extract populations {Divide (populations) and conquer (your analysis)}

Adegenet provides methods for subsetting data by individual or splitting all of the data into a list of populations. If you only want one or two populations, these methods become tedious. The *poppr* function `popsub` makes this easier:

2.3.1 Function: popsub

The command `popsub` is powerful in that it allows you to choose exactly what populations you choose to include or exclude from your analyses. As with many R functions, you can also use this within a function to avoid creating a new variable to keep track of.

Default Command:

```
popsub(gid, sublist = "ALL", blacklist = NULL, mat = NULL, drop = TRUE)
```

- `pop` - a `genind` object.
- `sublist` - vector of populations or integers representing the populations in your data set you wish to **retain**. For example: `sublist = c("pop_z", "pop_y")` or `sublist = 1:2`.
- `blacklist` - vector of populations or integers representing the populations in your data set you wish to **exclude**. This can take the same type of arguments as sublist, and can be used in conjunction with sublist for when you want a range of populations, but know that there is one in there that you do not want to analyze. For example: `sublist = 1:15, blacklist = "pop_x"`. One very useful thing about the blacklist is that it allows the user to be extremely paranoid about the data. You can set the blacklist to contain populations that are not even in your data set and it will still work!
- `mat` - (see section Multilocus Genotype Analysis for more information) A matrix produced from the `mlg.table` function. This overrides the *pop* argument and subsets this table instead.

To demonstrate this tool, let's revisit the virus data set that we saw in Defining hierarchies. Let's say we wanted to analyze only the data in North America. To make sure we are all on the same page, we will reset the population factor to "country".

```
setpop(virus) <- ~country
virus$pop.names  # Only two countries from North America.
```

```
##  [1] "Japan"          "USA"            "Finland"        "China"          "South Korea"
##  [6] "Norway"         "Taiwan"         "France"         "Latvia"         "Netherlands"
## [11] "Bulgaria"       "Turkey"         "United Kingdom" "Denmark"        "Austria"
## [16] "Canada"         "Italy"          "Russia"         "Bangladesh"     "Egypt"
## [21] "Germany"        "Romania"        "Ukraine"        "Czech Republic" "Greece"
## [26] "Iceland"        "Ireland"        "Sweden"         "Nepal"          "Saudi Arabia"
## [31] "Switzerland"    "Iran"           "Mongolia"       "Spain"          "Slovenia"
## [36] "Croatia"        "Algeria"
```

```
vna <- popsub(virus, sublist = c("USA", "Canada"))
vna$pop.names
```

```
##       P1       P2
##    "USA" "Canada"
```

If we want to see the population size, we can use the *adegenet* function `nInd()`:

```
c(NorthAmerica = nInd(vna), Total = nInd(virus))
```

```
## NorthAmerica        Total
##          665         1903
```

You can see that the population factors are correct and that the size of the data set is considerably smaller. Let's see the data set without the North American countries.

```
vnaminus <- popsub(virus, blacklist = c("USA", "Canada"))
vnaminus$pop.names
```

```
##              P01              P02              P03              P04              P05
##          "Japan"        "Finland"          "China"    "South Korea"         "Norway"
##              P06              P07              P08              P09              P10
##         "Taiwan"         "France"        "Latvia"    "Netherlands"       "Bulgaria"
##              P11              P12              P13              P14              P15
##         "Turkey" "United Kingdom"        "Denmark"        "Austria"          "Italy"
##              P16              P17              P18              P19              P20
##         "Russia"     "Bangladesh"          "Egypt"        "Germany"        "Romania"
##              P21              P22              P23              P24              P25
##        "Ukraine" "Czech Republic"         "Greece"        "Iceland"        "Ireland"
##              P26              P27              P28              P29              P30
##         "Sweden"          "Nepal"   "Saudi Arabia"    "Switzerland"           "Iran"
##              P31              P32              P33              P34              P35
##       "Mongolia"          "Spain"       "Slovenia"        "Croatia"        "Algeria"
```

Let's make sure that the number of individuals in both data sets is equal to the number of individuals in our original data set:

```
(nInd(vnaminus) + nInd(vna)) == nInd(virus)
```

```
## [1] TRUE
```

Now we have data sets with and without North America. Let's try something a bit more challenging. Let's say that we want the first 10 populations in alphabetical order, but we know that we still don't want any countries in North America. We can easily do this by using the *base* function `sort`.

```
vsort <- sort(virus$pop.names)[1:10]
vsort
```

```
##  [1] "Algeria"        "Austria"        "Bangladesh"     "Bulgaria"       "Canada"
##  [6] "China"          "Croatia"        "Czech Republic" "Denmark"        "Egypt"
```

```
valph <- popsub(virus, sublist = vsort, blacklist = c("USA", "Canada"))
valph$pop.names
```

```
##              P1              P2              P3              P4              P5
##         "China"       "Bulgaria"       "Denmark"       "Austria"    "Bangladesh"
##              P6              P7              P8              P9
##         "Egypt" "Czech Republic"       "Croatia"       "Algeria"
```

And that, is how you subset your data with poppr!

2.4 Clone-censor data sets {Attack of the clone correction}

Clone correction refers to the ability of keeping one observation of each MLG in a given population (or sub-population). Clone correcting can be hazardous if its done by hand (even on small data sets) and it requires a defined population hierarchy to get relevant results. *Poppr* has a clone correcting function that that will correct down to the lowest level of any defined population hierarchy. Note that clone correction in *poppr* is sensitive to missing data, as it treats all missing data as a single extra allele.

This function will create new data sets, but it is also utilized by the function `poppr` natively.

2.4.1 Function: clonecorrect

This function will return a clone corrected data set corrected for the lowest population level. Population levels are specified with the `hier` flag. You can choose to combine the population hierarchy to analyze at the lowest population level by choosing `combine = TRUE`.

Default Command:

```
clonecorrect(pop, hier = 1, dfname = "population_hierarchy",
    combine = FALSE, keep = 1)
```

- `pop` - a `genclone` object with a defined hierarchy or a `genind` object that has a population hierarchy data frame in the `@other` slot. Note, the `genind` object does not necessarily require a population factor to begin with.
- `hier` - A hierarchical formula (eg. ~`Pop/Subpop`), representing the hierarchical levels in your data.
- `dfname` - **Only for use in genind objects, otherwise, deprecated** The name of a data frame you have in the `@other` slot with the population factors.
- `combine` - Do you want to combine the population hierarchy? If it's set to `FALSE` (default), you will be returned an object with the top most hierarchical level as a population factor unless the **keep** argument is defined. If set to `TRUE`, the hierarchy will be returned combined.
- `keep` - This flag is to be used if you set `combine = FALSE`. This will tell clone correct to return a specific combination of your hierarchy defined as integers. For example, imagine a hierarchy that needs to be clone corrected at three levels: *Population* by *Year* by *Month*. If you wanted to only run an analysis on the *Population* level, you would set `keep = 1` since *Population* is the first level of the hierarchy. On the other hand, if you wanted to run analysis on *Year* by *Month*, you would set `keep = 2:3` since those are the second and third levels of the hierarchy.

Let's look at ways to clone-correct our data. We'll look at our *A. euteichies* data that we loaded earlier in the section Can you take me hier(archy) since that data set is known to include clonal populations [5]. Try playing around with the data and see what different combinations of the `hier`, and `keep` flags produce. Below, I will give three examples of clone corrections at the sample level with respect to field, at the field level, and finally, at the level of the entire data set.

First, we will examine the original data set.

```
aphan  # Original object

##
## This is a genclone object
## -------------------------
## Genotype information:
##
##    119 multilocus genotypes
##    187 diploid individuals
##     56 dominant loci
##
## Population information:
##
##      2 hierarchical levels - field sample
##     18 populations defined - Athena_1 Athena_2 Athena_3 ... Mt. Vernon_6 Mt. Vernon_7
## Mt. Vernon_8
```

Now we correct by sample with respect to field and keep the field as the population.

```
clonecorrect(aphan, hier = ~field/sample)

##
## This is a genclone object
## -------------------------
## Genotype information:
##
##    119 multilocus genotypes
##    141 diploid individuals
##     56 dominant loci
##
## Population information:
##
##      2 hierarchical levels - field sample
##      2 populations defined - Athena Mt. Vernon

# Your turn: Use the same hierarchy and use combine = TRUE and then keep = 1:2. Is there
# any difference?
```

Correcting by field. Notice how the number of MLG is much closer to our census.

```
clonecorrect(aphan, hier = ~field)

##
## This is a genclone object
## -------------------------
## Genotype information:
##
##    119 multilocus genotypes
##    120 diploid individuals
##     56 dominant loci
##
## Population information:
##
##      2 hierarchical levels - field sample
##      2 populations defined - Athena Mt. Vernon
```

Correcting over whole data set. Our MLG is equal to our census.

```
clonecorrect(aphan, hier = NA)

##
## This is a genclone object
## -------------------------
```

```
## Genotype information:
##
##    119 multilocus genotypes
##    119 diploid individuals
##     56 dominant loci
##
## Population information:
##
##      2 hierarchical levels - field sample
##     18 populations defined - Athena_1 Athena_2 Athena_3 ... Mt. Vernon_6 Mt. Vernon_7
## Mt. Vernon_8
```

2.5 Permutations and bootstrap resampling {every day I'm shuffling (data sets)}

A common null hypothesis for populations with mixed reproductive modes is panmixia, or to put it simply: lots of sex. *Poppr* randomly shuffles data sets in order to calculate P-values for the index of association (I_A and $\bar{r}_d$) [1] using 4 different methods:

method	strategy	units sampled
1	permutation	alleles
2	simulation	alleles
3	simulation	alleles
4	permutation	genotypes

These methods are detailed below. We will create a dummy data set to be shuffled by each example below. Let's assume a single diploid locus with four alleles (1, 2, 3, 4) with the frequencies of 0.1, 0.2, 0.3, and 0.4, respectively:

	A1/A2
1	4/4
2	4/1
3	4/3
4	2/2
5	3/3

Table 1: Original

The 4 methods are detailed below.

2.5.1 Function: shufflepop

Default Command:

```
shufflepop(pop, method = 1)
```

- `pop` - a `genind` object.
- `method` - a number indicating the method of sampling you wish to use. The following methods are available for use:

	A1/A2
1	3/4
2	2/3
3	4/4
4	2/1
5	3/4

Table 2: Permute 1

	A1/A2
1	1/3
2	2/4
3	3/4
4	4/3
5	4/2

Table 3: Permute 2

1. **Permute Alleles (default)** This is a sampling scheme that will **permute alleles within the locus.**

   The example above might become tables 2 and 3.

   As you can see, The heterozygosity has changed, yet the allelic frequencies remain the same. Overall this would show you what would happen if the sample you had underwent panmixis within this sample itself.

2. **Parametric Bootstrap** The previous scheme reshuffled the observed sample, but the parametric bootstrap **draws samples from a multinomial distribution using the observed allele frequencies as weights.** Tables 4 and 5 are examples of what I mean.

	A1/A2
1	1/3
2	3/3
3	3/2
4	4/4
5	4/2

Table 4: Parametric 1

	A1/A2
1	3/4
2	2/3
3	4/2
4	4/4
5	4/2

Table 5: Parametric 2

   Notice how the heterozygosity has changed along with the allelic frequencies. The frequencies for alleles 3 and 4 have switched in the first data set, and we've lost allele 1 in the second data set purely by chance! This type of sampling scheme attempts to show you what the true population would look like if it were panmictic and your original sample gave you a basis for estimating expected allele frequencies. Since estimates are made from the observed allele frequencies, small samples will produce skewed results.

3. **Non-Parametric Bootstrap** The third method is sampling with replacement, again **drawing from a multinomial distribution, but with no assumption about the allele frequencies** (tables 6 and 7).

	A1/A2
1	1/3
2	3/3
3	3/1
4	2/2
5	3/1

Table 6: Non-parametric 1

	A1/A2
1	1/3
2	3/1
3	2/3
4	2/1
5	4/3

Table 7: Non-parametric 2

   Again, heterozygosity and allele frequencies are not maintained, but now all of the alleles have a 1 in 4 chance of being chosen.

4. **Multilocus permutation** This is called Multilocus permutation because it does the same thing as the permutation analysis in the program *multilocus* by Paul Agapow and Austin Burt [1]. This

will shuffle the genotypes at each locus. Using our example above, tables 8 and 9 are shuffled with method 4.

	A1/A2
1	3/3
2	4/1
3	2/2
4	4/4
5	4/3

Table 8: ML 1

	A1/A2
1	4/4
2	2/2
3	3/3
4	4/3
5	4/1

Table 9: ML 2

Note that you have the same genotypes after shuffling, so at each locus, you will maintain the same allelic frequencies and heterozygosity. So, in this sample, you will only see a homozygote with allele 2. This also ensures that the P-values associated with I_A and $\bar{r}_d$ are exactly the same (for an explanation, see the end of FUNCTION: IA in this manual). Unfortunately, if you are trying to simulate a sexual population, this does not make much biological sense as it **assumes that alleles are not independently assorting within individuals.**

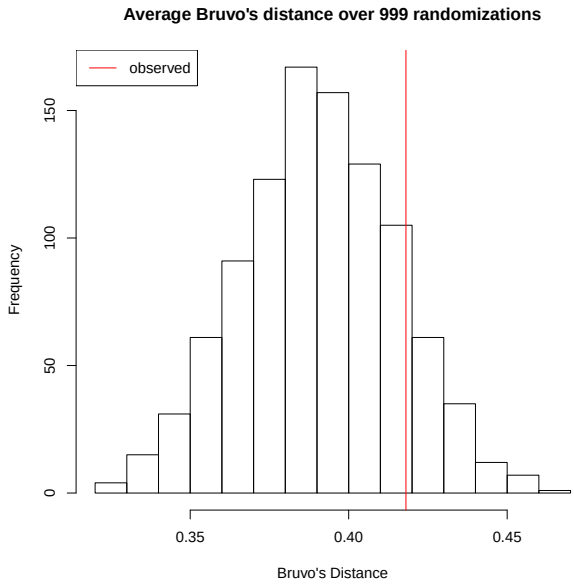
These shuffling schemes have been implemented for the index of association, but there may be other summary statistics you can use `shufflepop` for. All you have to do is use the function `replicate`. Let's use average BRUVO'S DISTANCE with the first population of the data set `nancycats` as an example:

```
data(nancycats)
nan1 <- popsub(nancycats, 1)
reps <- rep(2, 9)  # Assuming dinucleotide repeats.
observed <- mean(bruvo.dist(nan1, replen = reps))
observed

## [1] 0.4181
```

```
set.seed(9999)
bd.test <- replicate(999, mean(bruvo.dist(shufflepop(nan1, method = 2), replen = reps)))
```

You could use this method to replicate the resampling 999 times and then create a histogram to visualize a distribution of what would happen under different assumptions of panmixia.

```
hist(bd.test, xlab = "Bruvo's Distance", main = "Average Bruvo's distance over 999 randomizations")
abline(v = observed, col = "red")
legend("topleft", legend = "observed", col = "red", lty = 1)
```



2.6 Removing uninformative loci {Cut It Out!}

Phylogenetically uninformative loci are those that have only one sample differentiating from the rest. This can lead to biased results when using multilocus analyses such as the index of association (See THE MISSING LINKAGE DISEQUILIBRIUM and I KNOW WHAT YOU DID LAST SUMMARY TABLE). These nuisance loci can be removed with the following function.

2.6.1 Function: informloci

Default Command:

```
informloci(pop, cutoff = 2/nInd(pop), quiet = FALSE)
```

- `pop` - a `genind` object.
- `cutoff` - this represents the minimum fraction of individuals needed for a locus to be considered informative. The default is set to $2/n$ with n being the number of individuals in the data set (represented by the *adegenet* function `nInd`). Essentially, this means that any locus with fewer than 2 observations differing will be removed. The user can also specify a fraction of observations for the cutoff (eg. 0.05).
- `quiet` - if `TRUE`, nothing will be printed to the screen, if `FALSE` (default), the cutoff value in percentage and number of individuals will be printed as well as the names of the uninfomrative loci found.

Here's a quick example using the H3N2 virus SNP data set. We will only retain loci that have a minor allele frequency of $\geq 5\%$

```
data(H3N2)
H.five <- informloci(H3N2, cutoff = 0.05)
```

```
## cutoff value: 5 percent ( 95 individuals ).
##  47 uninfomative loci found: 157
## 177 233 243 262 267 280 303 313 327 357 382 384 399 412 418 424 425 429 433 451
## 470 529 546 555 557 564 576 592 595 597 602 612 627 642 647 648 654 658 663 667
## 681 717 806 824 837 882
```

Now what happens when you have all informative loci. We'll use the nancycats data set, which has microsatellite loci. It is important to note that this is searching for loci with a specified genotype frequency as fixed heterozygous sites are also uninformative:

```
data(nancycats)
naninform <- informloci(nancycats, cutoff = 0.05)

## cutoff value: 5 percent ( 12 individuals ).
## No sites found with fewer than 12 different individuals.
```

3 Multilocus Genotype Analysis

In populations with mixed sexual and clonal reproduction, it is not uncommon to have multiple samples from the same population have the same genotype across multiple loci (multilocus genotype, MLG). Here, we introduce tools for tracking MLGs within and across populations in GENIND objects from the *adegenet* package. We will be using the VIRUS data set which SNP data from isolates of the H3N2 virus from 2002 to 2006. Note that genclone objects are optimal for these analyses.

3.1 How many multilocus genotypes are in our data set? {Just a peek}

Counting the number of multilocus genotypes in a population is the first step for these analyses as they allow one to quickly see how many clones exist. With the GENCLONE object, This information is already displayed when we view the object.

```
virus

##
## This is a genclone object
## -------------------------
## Genotype information:
##
##     752 multilocus genotypes
##    1903 haploid individuals
##     125 codominant loci
##
## Population information:
##
##       3 hierarchical levels - country year month
##      37 populations defined - Japan USA Finland ... Slovenia Croatia Algeria
```

If we need to store the number of MLGs as a variable, we can simply run the `mlg` command.

```
virus_mlg <- mlg(virus)

## ###############################
## # Number of Individuals:  1903
## # Number of MLG:  752
## ###############################

virus_mlg

## [1] 752
```

We can see that since the number of individuals exceeds the number of multilocus genotypes, this data set contains clones. Let's take a look at where those clones are with respect to populations.

3.2 MLGs across populations {clone-ing around}

Since you have the ability to change the population structure of your data set freely, it is quite possible to see some of the same MLGs across different populations. Tracking them by hand can be a nightmare with large data sets. Luckily, `mlg.crosspop` has you covered in that regard.

3.2.1 Function: mlg.crosspop

Analyze the MLGs that cross populations within your data set. This has three output modes. The default one gives a list of MLGs, and for each MLG, it gives a named numeric vector indicating the abundance of that MLG in each population. Alternate outputs are described with `indexreturn` and `df`.

Default Command:

```
mlg.crosspop(pop, sublist = "ALL", blacklist = NULL, mlgsub = NULL,
    indexreturn = FALSE, df = FALSE, quiet = FALSE)
```

- `pop` - a `genind` object.
- `sublist` - see `popsub`. Analyze specified populations.
- `blacklist` - see `popsub`. Do not include specified populations.
- `mlgsub` - see `mlg.table`. Only analyze specified MLGs. The vector for this flag can be produced by this function as you will see later in this vignette.
- `indexreturn` - return a vector of indices of MLGs. (You can use these in the `mlgsub` flag, or you can use them to subset the columns of an MLG table).
- `df` - return a data frame containing the MLGs, the populations they cross, and the number of copies you find in each population. This is useful for making graphs in *ggplot2*.
- `quiet` - `TRUE` or `FALSE`. Should the populations be printed to screen as they are processed? (will print nothing if `indexreturn` is `TRUE`)

We can see what Multilocus Genotypes cross different populations and then give a vector that shows how many populations each multi-population MLG crosses.

```
setpop(virus) <- ~country
v.dup <- mlg.crosspop(virus, quiet = TRUE)
```

Here is a snippet of what the output looks like when `quiet` is `FALSE`. It will print out the MLG name, the total number of individuals that make up that MLG, and the populations where that MLG can be found.

```
## MLG.3: (12 inds) USA Denmark
## MLG.9: (16 inds) Japan USA Finland Denmark
## MLG.31: (9 inds) Japan Canada
## MLG.75: (23 inds) Japan USA Finland Norway Denmark Austria Russia Ireland
## MLG.80: (2 inds) USA Denmark
## MLG.86: (7 inds) Denmark Austria
## MLG.95: (2 inds) USA Bangladesh
## MLG.97: (8 inds) USA Austria Bangladesh Romania
## MLG.104: (3 inds) USA France
## MLG.110: (16 inds) Japan USA China
```

The output of this function is a list of MLGs, each containing a vector indicating the number of copies in each population. We'll count the number of populations each MLG crosses using the function `sapply` with `length`.

```
head(v.dup)

## $MLG.3
##     USA Denmark
##       4       8
##
## $MLG.9
##   Japan     USA Finland Denmark
##       1      13       1       1
##
## $MLG.31
##  Japan Canada
##      2      7
##
## $MLG.75
##   Japan     USA Finland  Norway Denmark Austria  Russia Ireland
##       2       8       2       1       6       2       1       1
##
## $MLG.80
##     USA Denmark
##       1       1
##
## $MLG.86
## Denmark Austria
##       3       4

v.num <- sapply(v.dup, length)  # count the number of populations each MLG crosses.
v.num

##   MLG.3   MLG.9  MLG.31  MLG.75  MLG.80  MLG.86  MLG.95  MLG.97 MLG.104 MLG.110
##       2       4       2       8       2       2       2       4       2       3
```

3.3 Producing MLG tables and graphs {bringing something to the table}

We can also create a table of multilocus genotypes per population as well as bar graphs to give us a visual representation of the data. This is achieved through the function `mlg.table`

3.3.1 Function: mlg.table

Produce a matrix containing counts of MLGs (columns) per population (rows). If there is no population structure to your data set, a vector will be produced instead.

Default Command:

```
mlg.table(pop, sublist = "ALL", blacklist = NULL, mlgsub = NULL,
    bar = TRUE, total = FALSE, quiet = FALSE)
```

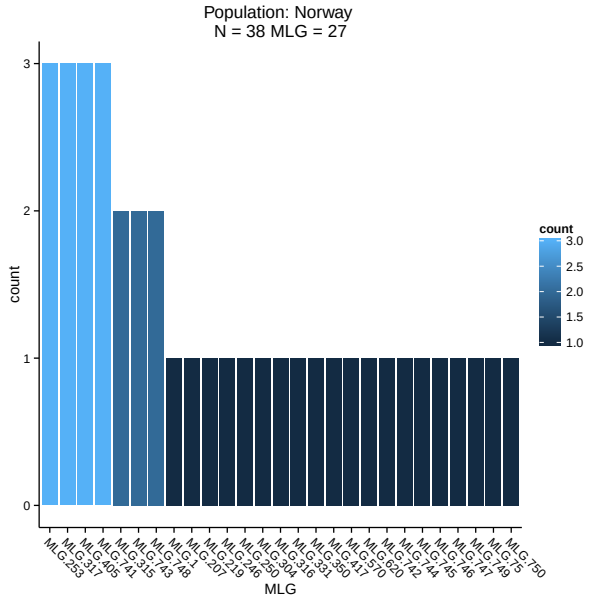
- `pop` - a `genind` object.
- `sublist` - a vector indicating which specific populations you want to produce a table for. This can be a numeric or character vector. See `popsub` for details.
- `blacklist` - a vector indicating which specific populations you do not want to include in your table. This can be a numeric or character vector, and does not necessarily have to be the same type as `sublist`. eg. `sublist=1:10, blacklist="USA"`. See `popsub` for details.
- `mlgsub` - a vector containing the indices of MLGs you wish to subset your table with.

- `bar` - `TRUE` or `FALSE`. If `TRUE`, a bar plot will be printed for each population with more than one individual.
- `total` - `TRUE` or `FALSE`. Should the entire data set be included in the table? This is equivalent to evoking `colSums` on the table.
- `quiet` - `TRUE` or `FALSE`. When `bar` is `TRUE`, should the populations be printed to screen as they are processed?

```
v.tab <- mlg.table(virus, quiet = TRUE, bar = TRUE)
v.tab[1:10, 1:10]  # Showing the first 10 columns and rows of the table.
```

```
##             MLG.1 MLG.2 MLG.3 MLG.4 MLG.5 MLG.6 MLG.7 MLG.8 MLG.9 MLG.10
## Japan           0     0     0     0     0     0     1     2     1      0
## USA             0     2     4     1     1     0     0     0    13      0
## Finland         0     0     0     0     0     0     0     0     1      0
## China           0     0     0     0     0     0     0     0     0      0
## South Korea     0     0     0     0     0     1     0     0     0      0
## Norway          1     0     0     0     0     0     0     0     0      0
## Taiwan          0     0     0     0     0     0     0     0     0      0
## France          0     0     0     0     0     0     0     0     0      0
## Latvia          0     0     0     0     0     0     0     0     0      0
## Netherlands     0     0     0     0     0     0     0     0     0      0
```

Figure 4: An example of a bar-chart produced by `mlg.table`. Note that this data set would produce several such charts.



The MLG table is not restricted for use with just *poppr*. One of the main advantages of the function `mlg.table` is that it allows easy access to diversity functions present in the package *vegan* [12]. One very simple example is to create a rarefaction curve for each population in your data set giving the number of expected MLGs for a given sample size. For more information, type `help("diversity", package="vegan")` in your R console.

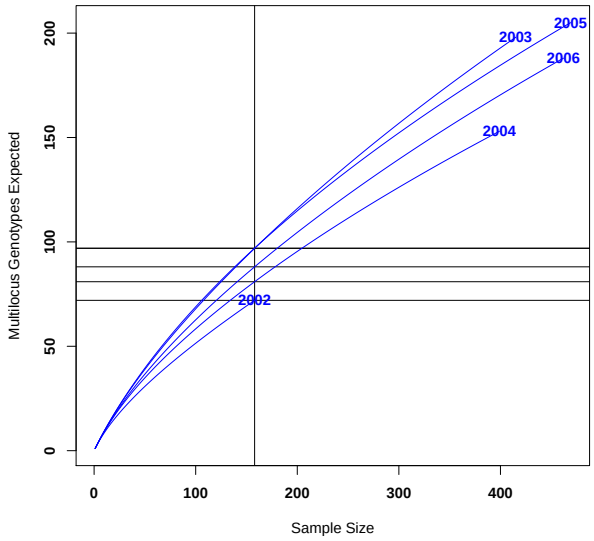
For the sake of a simple example, instead of drawing a curve for each of the 37 countries represented in this sample, let's change the population structure to be the different years of the epidemics.

```
setpop(virus) <- ~year
summary(virus)  # Check the data to make sure it's correct.
```

```
##
##  # Total number of genotypes:  1903
##
##  # Population sample sizes:
## 2002 2003 2004 2005 2006
##  158  415  399  469  462
##
##  # Number of alleles per locus:
## L001 L002 L003 L004 L005 L006 L007 L008 L009 L010 L011 L012 L013 L014 L015 L016 L017 L018
##    3    3    4    2    4    2    3    2    4    3    4    2    4    3    2    2    3    3
## L019 L020 L021 L022 L023 L024 L025 L026 L027 L028 L029 L030 L031 L032 L033 L034 L035 L036
##    2    2    3    3    3    2    2    2    2    2    2    2    2    2    2    4    4    3
## L037 L038 L039 L040 L041 L042 L043 L044 L045 L046 L047 L048 L049 L050 L051 L052 L053 L054
##    3    3    4    2    2    2    4    3    2    3    4    2    3    2    3    2    2    2
## L055 L056 L057 L058 L059 L060 L061 L062 L063 L064 L065 L066 L067 L068 L069 L070 L071 L072
##    4    2    2    2    2    2    2    2    4    4    4    3    3    2    3    4    3    2
## L073 L074 L075 L076 L077 L078 L079 L080 L081 L082 L083 L084 L085 L086 L087 L088 L089 L090
##    3    3    3    3    2    3    2    4    2    3    2    2    3    3    3    3    2    2
## L091 L092 L093 L094 L095 L096 L097 L098 L099 L100 L101 L102 L103 L104 L105 L106 L107 L108
##    2    2    3    2    3    2    3    2    3    2    3    3    2    2    2    3    2    2
## L109 L110 L111 L112 L113 L114 L115 L116 L117 L118 L119 L120 L121 L122 L123 L124 L125
##    2    3    3    3    2    2    3    3    3    3    4    2    3    3    4    3    2
##
##  # Number of alleles per population:
##   1   2   3   4   5
## 203 255 232 262 240
##
##  # Percentage of missing data:
## [1] 2.363
##
##  # Observed heterozygosity:
## [1] 0
##
##  # Expected heterozygosity:
## [1] 0
```

```
library(vegan)
H.year <- mlg.table(virus, bar = FALSE)
rarecurve(H.year, ylab = "Multilocus genotypes expected", sample = min(rowSums(H.year)), border = NA,
    fill = NA, font = 2, cex = 1, col = "blue")
```

Figure 5: An example of a rarefaction curve produced using a MLG table.



The minimum value from the *base* function `rowSums()` of the table represents the minimum common sample size of all populations. Setting the "sample" flag draws the horizontal and vertical lines you see on the graph. The intersections of these lines correspond to the numbers you would find if you ran the function `poppr` on this data set (under the column "`eMLG`").

3.4 Combining MLG functions {getting into the mix}

Alone, the different functionalities are neat. Combined, we can create interesting data sets. Let's say we wanted to know which MLGs were duplicated across the regions of the United Kingdom, Germany, Netherlands, and Norway. All we have to do is use the `sublist` flag in the function:

```
setpop(virus) <- ~country
UGNN.list <- c("United Kingdom", "Germany", "Netherlands", "Norway")
UGNN <- mlg.crosspop(virus, sublist = UGNN.list, indexreturn = TRUE)
```

OK, the output tells us that there are three MLGs that are crossing between these populations, but we do not know how many are in each. We can easily find that out if we subset our original table, `v.tab`.

```
UGNN  # Note that we have three numbers here. This will index the columns for us.

## MLG.315 MLG.317 MLG.620
##     315     317     620

UGNN.list  # And let's not forget that we have the population names.

## [1] "United Kingdom" "Germany"        "Netherlands"    "Norway"

v.tab[UGNN.list, UGNN]

##                MLG.315 MLG.317 MLG.620
## United Kingdom       1       0       0
## Germany              0       1       1
## Netherlands          0       0       0
## Norway               2       3       1
```

Now we can see that Norway has a higher incidence of nearly all of these MLGs. We can investigate the incidence of these MLGs throughout our data set. One thing that the GENCLONE object keeps track of is a single vector defining the unique multilocus genotypes within the data. These are represented as integers and can be accessed with the function `mlg.vector`. This is useful for finding what MLGs correspond to which individuals or populations. Let's use `mlg.vector` to find the individual names corresponding to the MLGs. First we'll investigate what the output of this function looks like.

```
v.vec <- mlg.vector(virus)
str(v.vec)  # Analyze the structure.

##  int [1:1903] 605 605 672 675 674 673 670 671 670 678 ...

length(unique(v.vec))  # count the number of MLGs

## [1] 752

virus  # equal to the first number in this output.

##
## This is a genclone object
## -------------------------
## Genotype information:
##
##     752 multilocus genotypes
##    1903 haploid individuals
##     125 codominant loci
##
## Population information:
##
##       3 hierarchical levels - country year month
##      37 populations defined - Japan USA Finland ... Slovenia Croatia Algeria
```

We can see that the numbers are all mixed up, but the order of the MLG vector corresponds to the order of individuals in our data set. We can see immediately that the first two individuals are both designated as MLG 605, indicating that they both have the exact same genetic profile. We will take our vector of MLGs crossing UK, Germany, Netherlands, and Norway and match them to the MLG vector to see where else they occur.

```
UGNN  # Show what we are looking for

## MLG.315 MLG.317 MLG.620
##     315     317     620

UGNN_match <- v.vec %in% UGNN
table(UGNN_match)  # How many individuals matched to those three MLGs?

## UGNN_match
## FALSE  TRUE
##  1881    22
```

22 individuals matched to those three MLGs. We can use this vector to analyze what individuals they come from.

```
indNames(virus)[UGNN_match]

##       0329       0330       0331       0332       0341       0342       0345       0556
## "CY026119" "CY026120" "CY026121" "CY026122" "CY026131" "CY026132" "CY026135" "EU502462"
##       0557       0558       0870       0974       1112       1113       1114       1122
## "EU502463" "EU502464" "EU501513" "AB243868" "DQ883618" "DQ883619" "DQ883620" "DQ883628"
##       1193       1209       1210       1281       1288       1426
## "EU501609" "EU501642" "EU501643" "EU501735" "EU501742" "EU502513"
```

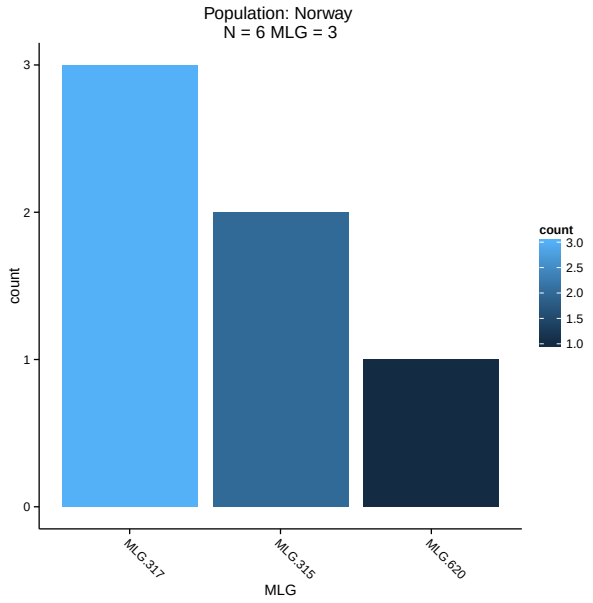
We can also use the vector of MLGs to subset `mlg.table` with the `mlgsub` flag.

```
mlg.table(virus, mlgsub = UGNN, bar = TRUE)
```

```
##                MLG.315 MLG.317 MLG.620
## Japan                4       1       0
## Norway               2       3       1
## United Kingdom       1       0       0
## Austria              0       0       7
## Germany              0       1       1
## Greece               0       0       1
```

And we can see where exactly these three MLGs fall within our data set.

Figure 6: An example of the same bar-chart as *Figure 1*, but focusing on three MLGs.



So, we've gotten this far, yet we haven't actually seen what the genotypes look like! For analyses where the genotypic signature is important, this is a crucial identification step. Lucky for us, the `genind` object retains all of the genotypic information and can be accessed using the `genind2df` function. Let's take a look at the three genotypes we specified above utilizing the vector of MLGs we created above, `H.vec`.

```
v.df <- genind2df(virus)
v.df[v.vec %in% UGNN, 1:15]  # Showing only 15 columns becaus it is a large dataset.
```

```
##                    pop 6 17 39 42 45 51 60 72 73 90 108 123 129 134
## CY026119        Austria a  a  g  c  g  c  g  g  c  g   a   g   t   g
## CY026120        Austria a  a  g  c  g  c  g  g  c  g   a   g   t   g
## CY026121        Austria a  a  g  c  g  c  g  g  c  g   a   g   t   g
## CY026122        Austria a  a  g  c  g  c  g  g  c  g   a   g   t   g
## CY026131        Austria a  a  g  c  g  c  g  g  c  g   a   g   t   g
## CY026132        Austria a  a  g  c  g  c  g  g  c  g   a   g   t   g
## CY026135        Austria a  a  g  c  g  c  g  g  c  g   a   g   t   g
## EU502462        Germany a  a  g  c  g  c  g  g  c  g   a   g   t   g
## EU502463         Greece a  a  g  c  g  c  g  g  c  g   a   g   t   g
## EU502464         Norway a  a  g  c  g  c  g  g  c  g   a   g   t   g
## EU501513          Japan a  a  g  c  t  c  a  g  a  g   t   g   t   g
## AB243868          Japan a  a  g  c  t  c  a  g  a  g   t   g   t   g
## DQ883618         Norway a  a  g  c  t  c  a  g  a  g   t   g   t   g
## DQ883619         Norway a  a  g  c  t  c  a  g  a  g   t   g   t   g
## DQ883620         Norway a  a  g  c  t  c  a  g  a  g   t   g   t   g
## DQ883628         Norway a  a  g  c  t  c  a  g  a  g   t   g   t   g
## EU501609        Germany a  a  g  c  t  c  a  g  a  g   t   g   t   g
## EU501642          Japan a  a  g  c  t  c  a  g  a  g   t   g   t   g
## EU501643          Japan a  a  g  c  t  c  a  g  a  g   t   g   t   g
## EU501735 United Kingdom a  a  g  c  t  c  a  g  a  g   t   g   t   g
## EU501742          Japan a  a  g  c  t  c  a  g  a  g   t   g   t   g
## EU502513         Norway a  a  g  c  t  c  a  g  a  g   t   g   t   g
```

Notice that there seems to be a clear separation between the SNPs of the first 10 isolates and the rest? This is no coincidence. Take a look at the output of our earlier sub-setting.

```
v.vec[UGNN_match]

##  [1] 620 620 620 620 620 620 620 620 620 620 317 315 315 317 317 315 317 315 315 315 315
## [22] 317
```

We have the MLGs 315, 317, and 620, and the result of the sub-setting shows us that 620 occurs earlier in our data set, and that MLGs 315 and 317 are mixed in together. The reason why we do not see a mixture of three different sets of SNP calls in our little window is because `mlg.vector` creates the MLGs by first concatenating and then sorting the genotypes. This way, the closer two MLG indexes are to each other, the fewer differences they will have between one another.

3.5 Alternative data visualization {do you see what I see?}

The graphs that are output by *poppr* are simply aids for the user to make data analysis easier. We want to better visualize how these MLGs cross populations by MLG or population. We also want to see exactly what MLGs are in which populations, and how prevalent they are. As the package *ggplot2* is based on data frames, we have to give ourselves a data frame to work with. We can do this using the `df = TRUE` flag.

```
df <- mlg.crosspop(virus, df = TRUE, quiet = TRUE)
names(df)

## [1] "MLG"        "Population" "Count"
```

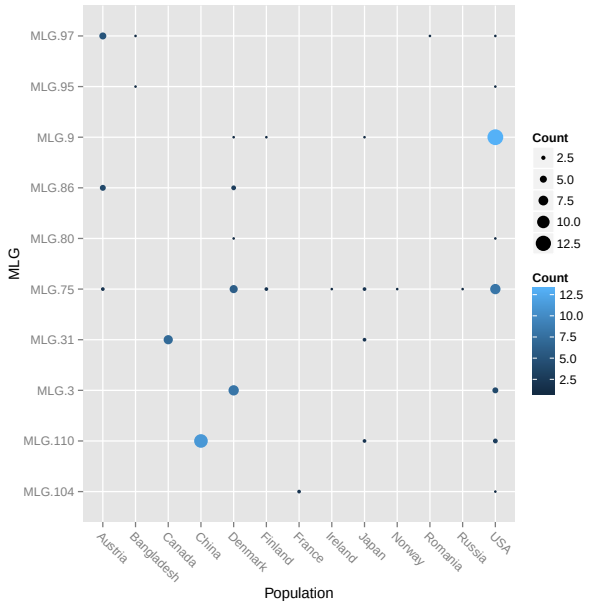
Now that we have our data frame, we can do a couple of things. We can first see where the most omnipresent MLG occurs. After that, we will plot the top ten MLGs using ggplot2. Remember that `v.num` is the number of populations per multilocus genotype.

```
v.max <- names(sort(v.num, decreasing = TRUE)[1:10])
# Showing the data frame by the largest MLG complex.
df[df$MLG %in% v.max[1], ]

##       MLG Population Count
## 9  MLG.75      Japan     2
## 10 MLG.75        USA     8
## 11 MLG.75    Finland     2
## 12 MLG.75     Norway     1
## 13 MLG.75    Denmark     6
## 14 MLG.75    Austria     2
## 15 MLG.75     Russia     1
## 16 MLG.75    Ireland     1
```

And now we can visualize the largest ten MLG complexes using *ggplot2*'s `qplot` function.

Figure 7: An example of the versatility of the MLG information.

```
df2 <- df[df$MLG %in% v.max, ]
library(ggplot2)
qplot(y = MLG, x = Population, data = df2, color = Count, size = Count) + theme(axis.text.x = element_text(size = 10,
    angle = -45, hjust = 0))
```



4 Data Analysis

4.1 Calculating the index of association, I_A and $\bar{r}_d$ {the missing linkage disequilibrium}

The index of association was originally developed as a measure of multilocus linkage disequilibrium [2] and was found to be able to detect signatures of sexual reproduction and population structure [2, 17]. Unfotunately, I_A was found to increase with the number of loci, and was not suitable to comparisons across studies [1]. To remedy this, $\bar{r}_d$ was developed that corrects for this scaling and forces the index to lie

between 0 (linkage equilibrium) and 1 (full disequilibrium). I_A has previously been implemented in a couple of programs including *multilocus* [1] and *LIAN* [6]. While both of these programs are still available for download, *multilocus* is no longer actively supported, and *LIAN*, despite its speed, is only appropriate for haplotypic data. Both of these programs each require one specific file format, and, until recently[4], neither of these programs had an internal ability to run in batch across multiple populations within a file or multiple files within a directory in the same way that poppr can (see footnote).

It is important to note that for this algorithm, all missing values are treated in the same way as *multilocus* in that all missing alleles are imputed to be the same as the alleles they are being compared to. Depending on the percent missing data in your data set, this might influence the statistic. If you have a lot of missing data, consider using the `missing` flag in this function.

4.1.1 Function: ia

This function is a quick look at a single data set. It can do almost everything that `poppr` can do except for sorting through populations.

Default Command:

```
ia(pop, sample = 0, method = 1, quiet = FALSE, missing = "ignore",
    hist = TRUE, valuereturn = FALSE)
```

- `pop` - a `genind` object.
- `sample` - You should use this flag whenever you want to reshuffle your data set. Indicate how many times you want to reshuffle your data set to obtain a P-value.
- `method` - a number from 1 to 4 indicating the sampling method:
  1. permutation over alleles.
  2. parametric bootstrap.
  3. non-parametric bootstrap.
  4. *multilocus* style permutation [1].

  The methods are detailed in `shufflepop`.
- `quiet` - If set to `TRUE`, nothing will be printed to the screen as the sampling progresses. If `FALSE` will produce a progress bar.
- `missing` - This will preprocess your missing values. It is set to ignore missing data, so that they do not contribute to the distance measure. It can also be set to `"loci", "geno", "zero", or "mean"`. For details, see MISSINGNO.
- `hist` - This will produce a pair of histograms for each population showing the distribution of I_A and $\bar{r}_d$ across the sampled data sets, and plot the observed value as a single vertical line.
- `valuereturn` - If set to `TRUE` and the number of samples is greater than zero, it will return all values generated from the permuted data.

Running the analysis is as simple as this:

[4] LIAN 3.6 allows the user to run multiple contiguous data sets within a single file or across multiple files. It is impossible to run MULTILOCUS in batch.

```
ia(nancycats)

##      Ia   rbarD
## 0.17207 0.02179
```

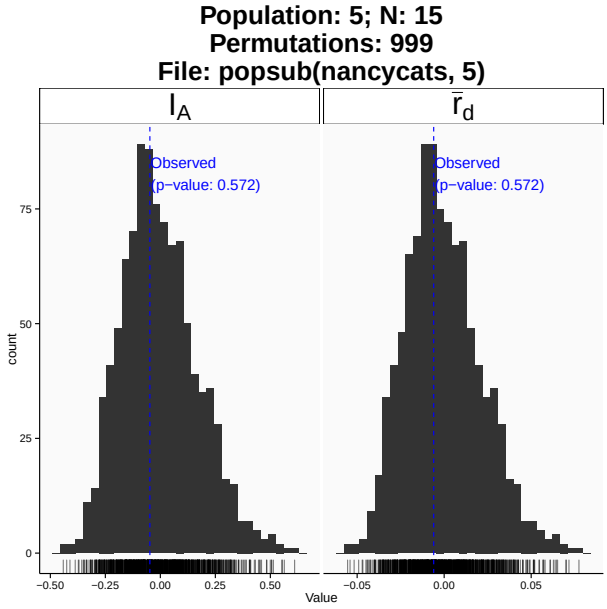
We can use `popsub` to subset for specific populations. Here, we'll also demonstrate the sampling flag and show you what the histogram looks like.

```
set.seed(1009)
ia(popsub(nancycats, 5), sample = 999)
```

```
## |======================================================================| 100%
##        Ia      p.Ia     rbarD      p.rD
## -0.047540  0.572000 -0.006004  0.572000
```

This analysis produced the histograms you see below. What these histograms represent are 999 resamplings of the data under the null hypothesis (H_0) of sexual reproduction. The way that H_0 is created is determined by the sampling method chosen. In this case, the method was to shuffle genotypes at each locus to simulate unlinked loci. Since the P = 0.572, we would fail to reject H_0 and we therefore might conclude that this population is sexually reproducing [2] [17] [1].

Figure 8: Histograms of 999 values of I_A and $\bar{r}_d$ calculated from 999 resamplings of population 5 from the data set "nancycats". The observed values of I_A and $\bar{r}_d$ are represented as vertical blue lines overlaid on the distributions. The ticks at the bottom of each histogram represent individual observations.



If we wanted to, we could also have set the flag `valuereturn = TRUE` to get back our permuted data if we wanted to make our own histograms (we'll set the number of samples to 9 for demonstrative purposes):

```
set.seed(1009)
ia(popsub(nancycats, 5), sample = 9, hist = FALSE, valuereturn = TRUE)
```

```
## |======================================================================| 100%
## $index
```

```
##        Ia      p.Ia     rbarD      p.rD
## -0.047540  0.500000 -0.006004  0.500000
##
## $samples
##         Ia     rbarD
## 1 -0.17418 -0.022057
## 2  0.11100  0.014208
## 3 -0.05166 -0.006611
## 4 -0.22706 -0.028948
## 5 -0.29079 -0.036685
## 6  0.17143  0.021764
## 7  0.12743  0.016054
## 8 -0.04898 -0.006224
## 9 -0.01005 -0.001270
```

There, are, of course a couple of caveats that need to be mentioned regarding our P-values. First, while we have equivalent P-values for I_A and $\bar{r}_d$, they might not always be equal due to the difference in calculation. Details about that can be found in the Appendix section I_A AND $\bar{r}_d$. Second, the P-values are calculated by comparing how many permuted values are greater than or equal to the observed value. This includes the observed value (which is why setting the randomizations to 999 will give you a round P-value) which means that the lowest P-value you will ever have is $1/(n+1)$ where n is the number of permutations you select. Take for example this population of a clonal root rot pathogen, *Aphanomyces euteiches*:

```
data(Aeut)
set.seed(1001)
ia(popsub(Aeut, 1), sample = 999, method = 2, quiet = TRUE, hist = FALSE)
```

```
##      Ia    p.Ia   rbarD    p.rD
## 2.90603 0.00100 0.07237 0.00100
```

If you want to be able to report $P < 0.001$ in this situation, then you can simply increase the number in sample: `sample = 1999`

4.2 Genetic distances {may I have distance?}

Genetic distances are great tools for analyzing diversity in populations as they are the basis for creating dendrograms with bootstrap support and also for AMOVA. This section will simply present different genetic distances along with a few notes about them. Most of these distances are derived from the *ade4* and *adegenet*packages, where they were implemented as distances between populations. *Poppr*extends the implementation to individuals as well (with the exception of Bruvo's distance).

Method	Function	Assumption	Euclidean
PROVESTI	`provesti.dist`	-	No
	`diss.dist`		
NEI	`nei.dist`	Infinite Alleles	No
		Genetic Drift	
EDWARDS	`edwards.dist`	Genetic Drift	Yes
REYNOLDS	`reynolds.dist`	Genetic Drift	Yes
ROGERS	`rogers.dist`	-	Yes
BRUVO	`bruvo.dist`	Stepwise Mutation	No

Note on these distances: It is valuable for the user to know which distance is appropriate for use within his/her system. All of these distances except for Bruvo's distance are based on allele frequencies. Bruvo's distance takes into account the value of the allele, and thus may be more appropriate for questions of finding clonal descendents as opposed to a distance that will estimate the number of mutations that occured.

Currently only Bruvo's distance and Provesti's distance perform as expected with polyploid data. The other distances appear to give spurious results.

All of the distances are based on the table found in a GENIND OBJECT. I have translated the following from math jargon into English with a few minor modifications:

> Let **A** a table containing allelic frequencies with t populations or individuals (rows) and m alleles (columns).
> Let ν be the number of loci. The locus j gets $m(j)$ alleles.

$$m = \sum_{j=1}^{\nu} m(j) \tag{1}$$

Again, this is describing the table present in the GENIND OBJECT where, instead of having the number of columns equal the number of loci, the number of columns equals the number of observed alleles in the entire data set. So, if you had a data set with 5 loci that had 2 alleles each, your table would have ten columns. Of course, codominant loci like microsatellites have varying numbers of alleles.

> For the row i and the modality k of the variable j, notice the value

$$a_{ijk}(1 \leq i \leq t, 1 \leq j \leq \nu, 1 \leq k \leq m(j)) \tag{2}$$

> the value of the initial table.

$$a_{ij\cdot} = \sum_{k=1}^{m(j)} a_{ijk} \tag{3}$$

> and

$$p_{ijk} = \frac{a_{ijk}}{a_{ij\cdot}} \tag{4}$$

The above couple of equations are basically defining the allele frequency (p_{ijk}). Remember that i is individual, j is locus, and k is allele. The following continutes to describe properties of the table:

$$p_{ij\cdot} = \sum_{k=1}^{m(j)} p_{ijk} = 1 \tag{5}$$

The sum of all allele frequencies for a single population (or individual) at a single locus is one.

$$p_{i\cdot\cdot} = \sum_{j=1}^{\nu} p_{ij\cdot} = \nu \tag{6}$$

The sum of all allele frequences over all loci is equal to the number of loci.

$$p_{\cdots} = \sum_{j=1}^{\nu} p_{i\cdot\cdot} = t\nu \tag{7}$$

The the sum of the entire table is the sum of all loci multiplied by the number of populations (or individuals).

4.2.1 Non-Euclidean distances

Dissimilarity distance

One of the least known distances and yet more conceptually satisfying is Provesti's (Also spelled Prevosti) distance. It's conceptually satisfying because it can be thought of as a percentage of dissimilar sites between two individuals or populations or the absolute genetic distance. It is implemented in two functions, `provesti.dist` and `diss.dist`.

$$D_P(a,b) = \frac{1}{2\nu}\sum_{k=1}^{\nu}\sum_{j=1}^{m(k)}|p_{ajk} - p_{bjk}| \tag{8}$$

Note: for AFLP data, the 2 is dropped.

Nei's distance

$$D_1(a,b) = -\ln\left(\frac{\sum_{k=1}^{\nu}\sum_{j=1}^{m(k)} p_{ajk}p_{bjk}}{\sqrt{\sum_{k=1}^{\nu}\sum_{j=1}^{m(k)}(p_{ajk})^2}\sqrt{\sum_{k=1}^{\nu}\sum_{j=1}^{m(k)}(p_{bjk})^2}}\right) \tag{9}$$

4.2.2 Euclidean distance

Edward's Angular distance

$$D_2(a,b) = \sqrt{1 - \frac{1}{\nu}\sum_{k=1}^{\nu}\sum_{j=1}^{m(k)}\sqrt{p_{ajk}p_{bjk}}} \tag{10}$$

Reynold's Coancestry distance

$$D_3(a,b) = \sqrt{\frac{\sum_{k=1}^{\nu}\sum_{j=1}^{m(k)}(p_{ajk} - p_{bjk})^2}{2\sum_{k=1}^{\nu}(1 - \sum_{j=1}^{m(k)} p_{ajk}p_{bjk})}} \tag{11}$$

Roger's distance

$$D_4(a,b) = \frac{1}{\nu}\sum_{k=1}^{\nu}\sqrt{\frac{1}{2}\sum_{j=1}^{m(k)}(p_{ajk} - p_{bjk})^2} \tag{12}$$

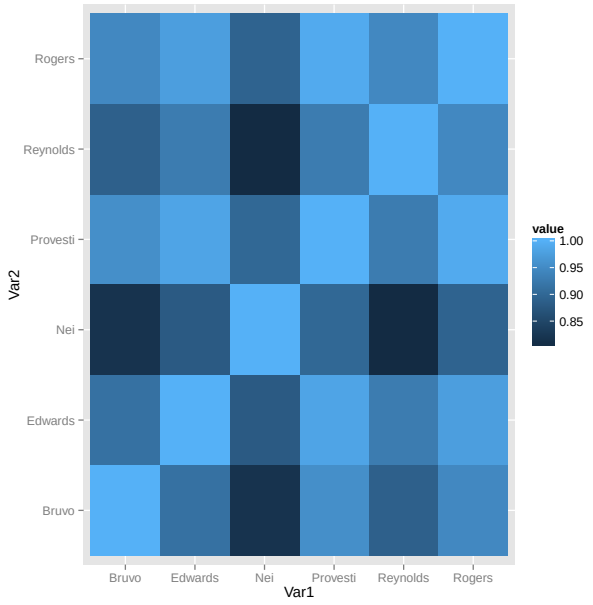
4.2.3 Bruvo's distance for microsatellites

This is based off of a specific model where it is necessary to specify the repeat lengths of the markers at each locus. Eg a repeat of $(CAT)^n$ would have a repeat length of 3.

Comparing all distances (WIP)

```
data(nancycats)
nancy9 <- popsub(nancycats, 9)
repeats <- rep(2, 9)
pro_nan <- provesti.dist(nancy9)
nei_nan <- nei.dist(nancy9)
edw_nan <- edwards.dist(nancy9)
rey_nan <- reynolds.dist(nancy9)
rog_nan <- rogers.dist(nancy9)
bru_nan <- bruvo.dist(nancy9, replen = repeats)
distlist <- list(Provesti = pro_nan, Nei = nei_nan, Edwards = edw_nan, Reynolds = rey_nan,
    Rogers = rog_nan, Bruvo = bru_nan)
combs <- combn(x = 1:length(distlist), m = 2)
mtest <- apply(combs, 2, function(x) mantel.randtest(distlist[[x[1]]], distlist[[x[2]]])$obs)
attr(mtest, "class") <- "dist"
attr(mtest, "Size") <- 6
attr(mtest, "Labels") <- names(distlist)
```

```
mtest <- as.matrix(mtest)
diag(mtest) <- 1
library(reshape2)
library(ggplot2)
ggplot(melt(mtest)) + geom_tile(aes_string(x = "Var1", y = "Var2", fill = "value"))
```



4.3 Bootstrap analysis {give 'em the boot(strap)}

4.3.1 Bootstrapping any distance

4.3.2 Bootstrapping Bruvo's Distance

This is optimized for Bruvo's distance.

4.4 Minimum spanning networks {nothing but net(works)}

4.5 Analysis of MOlecular VAriance {Come here! AMOVA here!}

4.6 Going the distance {dissimilarity distance}

Since *poppr* is still in its infancy, the number of distance measures it can offer are few. Bruvo's distance is well supported and allows you to quickly visualize your data, but it only allows for microsatellites. The index of association, above, utilizes a discreet dissimilarity distance matrix. It is with this matrix that we have constructed a relative dissimilarity distance where the distance is the ratio of the number of dissimilarities to the number of dissimilarities possible. The number of dissimilarities possible is the number of loci multiplied by the ploidy, so if you have 10 loci from a diploid population, then there are 20 dissimilarities possible. For details, see equations (14) and (15) in I_A AND $\bar{r}_d$.

4.6.1 Function: diss.dist

Use this function to calculate relative dissimilarity between individuals and return a distance matrix for use in creating cladograms or minimum spanning networks. A note: missing alleles will be imputed to be the same as the challenging allele, decreasing the distance between some individuals. If you want to consider all missing data as special alleles, treat your data with `missingno(pop, type = "zero")` beforehand.

Default Command:

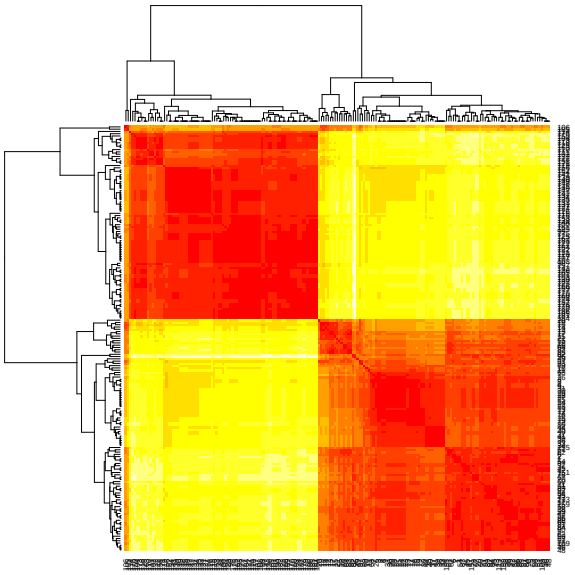
```
diss.dist(x, diff = TRUE, frac = TRUE, mat = FALSE)
```

- `pop` - a `genind` object.

Since we have a data set that we know is very clonal, let's analyze the *A. euteiches* data set [5] and create a heatmap to visualize the degree of difference between populations.

```
data(Aeut)
A.dist <- diss.dist(Aeut)
heatmap(as.matrix(A.dist), symm = TRUE)
```

Figure 9: Heatmap representation of a dissimilarity distance for the data set "Aeut"



4.7 Bruvo's distance {step by stepwise mutation}

Bruvo's distance is a genetic distance measure for microsatellite markers utilizing a stepwise mutation model that allows for differing ploidy levels [3]. As *adegenet's* `genind` object has an all or none approach to missing data, any genotypes not exhibiting full ploidy will be treated as missing. This means that only non-special cases will be considered for the calculation and missing data will be ignored [3].

This is true only if missing data is treated as missing when importing to *adegenet*. If missing data in the initial data frame has missing microsatellite alleles coded as 0, then the genind object will not treat them as missing (unless the genotype has no alleles in it). An example adapted from the *adegenet* "Getting Started" vignette shows how this can be done:

```
set.seed(5001)
temp <- lapply(1:30, function(i) sample(0:9, 4, replace = TRUE))
temp <- sapply(temp, paste, collapse = "/")
temp <- matrix(temp, nrow = 10, dimnames = list(paste("ind", 1:10), paste("loc", 1:3)))
temp

##        loc 1     loc 2     loc 3
## ind 1  "9/7/7/1" "5/6/8/7" "1/3/9/6"
## ind 2  "5/6/7/0" "1/8/1/8" "2/7/6/1"
## ind 3  "0/7/7/9" "2/4/4/0" "8/8/9/2"
## ind 4  "4/8/6/4" "8/1/0/2" "6/2/5/4"
## ind 5  "2/0/3/1" "6/9/1/8" "1/0/0/1"
## ind 6  "5/3/0/5" "5/5/0/0" "8/2/6/2"
## ind 7  "0/7/4/7" "8/5/8/5" "1/3/8/0"
## ind 8  "7/0/7/5" "2/1/5/4" "9/5/2/8"
## ind 9  "4/8/0/1" "8/8/8/4" "9/0/3/4"
## ind 10 "4/4/3/3" "2/0/5/6" "8/7/1/5"

obj <- df2genind(temp, ploidy = 4, sep = "/")
pop(obj) <- paste("ind", 1:10)
```

We will save this object for a later demonstration of the addition and loss models of Bruvo's distance. It is important to note that this is a distance between individuals, not populations, unlike Nei's 1978 distance [11]. For distances between populations, see the *adegenet* function `dist.genpop`

4.7.1 Function: bruvo.dist

Bruvo's distance requires knowledge of the repeat lengths of each locus, so take care to read the description below.

Default Command:

```
bruvo.dist(pop, replen = 1, add = TRUE, loss = TRUE)
```

- `pop` - a `genind` object.
- `replen` - This is a vector of numbers indicating the repeat length for each locus in your sample. If you have two dinucleotide repeats and five tetranucleotide repeats, you would put `c(2,2,4,4,4,4,4)` in this field. If you have imported data where that represents the raw number of steps, all you would have to type is `rep(1, n)`, replacing n with the number of loci in your sample. It is important that you place something in this field because this function will attempt to estimate the repeat length based on the minimum difference of the alleles represented; with variability of position calls, relying on this estimation is NOT recommended.
- `add` - For missing data: use the genome addition model (see Special cases of Bruvo's distance.)

- `loss` - For missing data: use the genome loss model (see Special cases of Bruvo's distance.)

To illustrate why it is important to specify the repeat lengths, let's imagine a locus that contains 5 alleles and the true repeat length is 4. Note that Bruvo's distance between alleles is calculated as $1 - 2^{-[x]}$, where x is the difference in repeat lengths:

```
locus1 <- c(244, 248, 256, 240, 236)
locus1/4

## [1] 61 62 64 60 59

1 - 2^-dist(locus1/4)

##        1      2      3      4
## 2 0.5000
## 3 0.8750 0.7500
## 4 0.5000 0.7500 0.9375
## 5 0.7500 0.8750 0.9688 0.5000
```

We can see that the distance between them ranges from 1 to 5. Let's say, that we accidentally wrote 2 or 8 instead of 4:

```
locus1/2

## [1] 122 124 128 120 118

1 - 2^-dist(locus1/2)  # Distance increase

##        1      2      3      4
## 2 0.7500
## 3 0.9844 0.9375
## 4 0.7500 0.9375 0.9961
## 5 0.9375 0.9844 0.9990 0.7500

locus1/8

## [1] 30.5 31.0 32.0 30.0 29.5

1 - 2^-dist(locus1/8)  # Distance decrease

##        1      2      3      4
## 2 0.2929
## 3 0.6464 0.5000
## 4 0.2929 0.5000 0.7500
## 5 0.5000 0.6464 0.8232 0.2929
```

While we will still get results from this analysis with the incorrect repeat length, they will be inherently wrong as they do not represent the true distance. That being said, it's important to note that the repeat lengths we represent for the rest of the manual are not known by the authors, but are used as a simple example.

This function will return a distance matrix (displaying the smallest population in the data set "nancycats"):

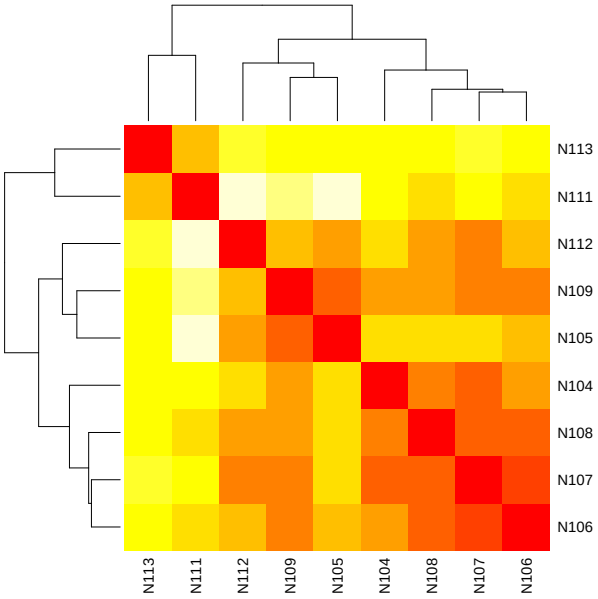
```
dist9 <- bruvo.dist(popsub(nancycats, 9), replen = rep(1, 9))
dist9

##        N104   N105   N106   N107   N108   N109   N111   N112
## N105 0.5779
## N106 0.4008 0.4563
## N107 0.2203 0.5093 0.1806
## N108 0.3270 0.5534 0.2352 0.2179
## N109 0.4016 0.2760 0.3192 0.3331 0.4295
```

```
## N111 0.6150 0.8708 0.5533 0.6167 0.5219 0.7331
## N112 0.5492 0.4086 0.4392 0.3289 0.3886 0.4529 0.8037
## N113 0.5926 0.6228 0.6203 0.6586 0.6186 0.6100 0.5060 0.7026
```

You can visualize this better with a simple heatmap:

```
heatmap(as.matrix(dist9), symm = TRUE)
```

Figure 10: Heatmap representation of Bruvo's distance for population 9 of the data set "nancycats"



Let's take a closer look at the two individuals, N113 and N111. They seem to have large distances between everyone else and themselves. The names and columns of the matrix contain the names of individuals, but not the population information. We can make a comparison of Bruvo's distance across populations easier by editing the "Labels" attribute of the distance object. Let's take a look at the labels attribute using the `attr()` command. ¡¡attrLabels¿¿= attr(dist9, "Labels") @ Remember that they all came from population 9, so let's append that to each label using the `paste()` command.

```
dist9.attr <- attr(dist9, "Labels")
attr(dist9, "Labels") <- paste(rep("P09", 9), dist9.attr)
dist9
```

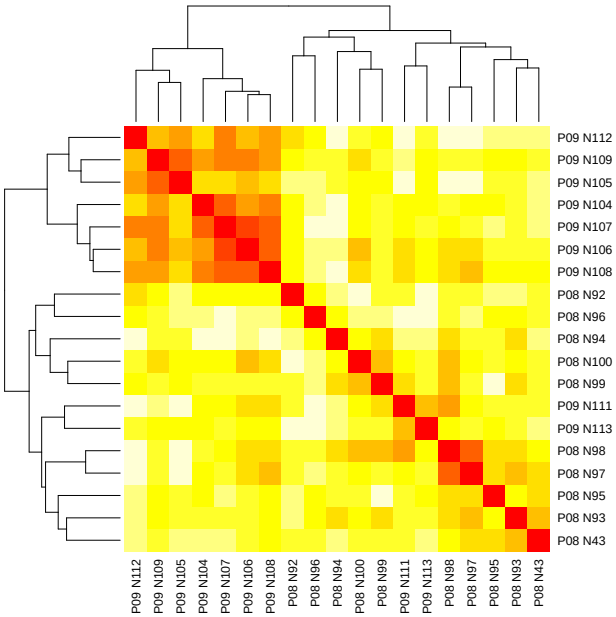
```
##          P09 N104 P09 N105 P09 N106 P09 N107 P09 N108 P09 N109 P09 N111 P09 N112
## P09 N105   0.5779
## P09 N106   0.4008   0.4563
## P09 N107   0.2203   0.5093   0.1806
## P09 N108   0.3270   0.5534   0.2352   0.2179
## P09 N109   0.4016   0.2760   0.3192   0.3331   0.4295
## P09 N111   0.6150   0.8708   0.5533   0.6167   0.5219   0.7331
## P09 N112   0.5492   0.4086   0.4392   0.3289   0.3886   0.4529   0.8037
## P09 N113   0.5926   0.6228   0.6203   0.6586   0.6186   0.6100   0.5060   0.7026
```

Now we can see that all of the labels are corresponding to population 9. Let's calculate Bruvo's distance between populations 8 and 9.

```
dist9to8 <- bruvo.dist(popsub(nancycats, 8:9), replen = rep(1, 9))
dist9to8.attr <- attr(dist9to8, "Labels")
```

```
nan9to8pop <- nancycats@pop[nancycats@pop %in% c("P08", "P09")]
attr(dist9to8, "Labels") <- paste(nan9to8pop, dist9to8.attr)
heatmap(as.matrix(dist9to8), symm = TRUE)
```

Figure 11: Heatmap representation of Bruvo's distance for populations 8 and 9 of the data set "nancycats"



Remember N113 and N111? Take a look at where they fall on the heatmap. They don't cluster together with population 9 anymore, but somewhere in population 8.

4.8 See the forest for the trees {visualizing distances with dendrograms and networks}

Staring at a raw distance matrix might be able to tell you something about your data, but it also might be able to ruin your eyesight. In this section, we present functions to display this data in trees and networks.

4.8.1 Function: bruvo.boot

This function provides the ability to draw a dendrogram based on Bruvo's distance including bootstrap support.

Default Command:

```
bruvo.boot(pop, replen = 1, add = TRUE, loss = TRUE, sample = 100,
    tree = "upgma", showtree = TRUE, cutoff = NULL, quiet = FALSE,
    ...)
```

- `pop` - a `genind` object.
- `replen` - see `bruvo.dist`, above.
- `add` - For missing data: use the genome addition model (see Special cases of Bruvo's distance.)

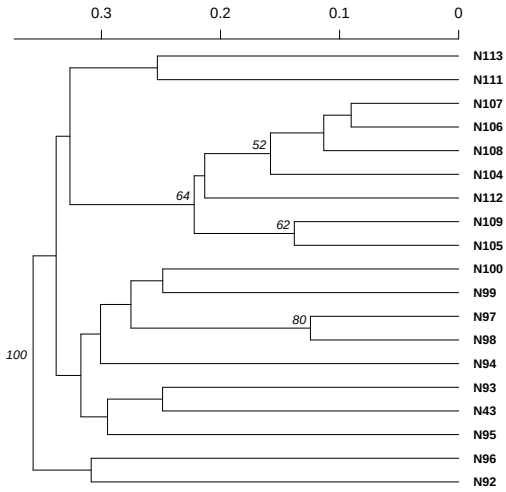
- `loss` - For missing data: use the genome loss model (see Special cases of Bruvo's distance.)
- `sample` - How many bootstraps do you want to perform?
- `tree` - Two trees are available, Neighbor-Joining `"nj"` or UPGMA `"upgma"`.
- `showtree` - if `TRUE`, a tree will be plotted automatically.
- `cutoff` - This is a number between 0 and 100 indicating the cutoff value for the bootstrap nodelables. If you only wanted to see the the boostrap values for nodes that were present more than 75% of the time, you would use `cutoff = 75`. If you don't put anything for this parameter, all values will be shown.
- `quiet` - if `quiet = TRUE`, no standard messages will be printed to screen. If `quiet = FALSE` (default), then a progress bar and standard message will be printed to the screen.

For this example, let's set the cutoff to 50%.

```
set.seed(410)
nan9tree <- bruvo.boot(popsub(nancycats, 8:9), replen = rep(1, 9), sample = 1000, cutoff = 50)
```

```
##
## Bootstrapping...
## (note: calculation of node labels can take a while even after the progress bar is full)
## |================================================================================| 100%
```

Figure 12: UPGMA Tree of Bruvo's distance for population 9 of the data set "nancycats" with 1000 Bootstrap Replicates. Node labels represent percentage of bootstrap replicates that contained that node.



4.8.2 Function: greycurve

Use this function to display a gradient of grey values based on user- defined parameters. The following functions will display a minimum spanning network that utilize a grey scale to display the weight of the lines (referred to as "edges") that connect two or more individuals. The darker the line the closer the distance. Since this is based off of a linear grey scale, what happens when you have a distance matrix comprised of values all below 0.2 or all above 0.8?

With linear grey scaling, it becomes very difficult to detect the differences in these ranges. The following function allows you to visualize and manipulate a gradient from black to white so that you can use it in *poppr*'s msn functions below to maximize the visual differences in your data.

Default Command:

```
greycurve(data = seq(0, 1, length = 1000), glim = c(0, 0.8),
    gadj = 3, gweight = 1, scalebar = FALSE)
```

This function does not return any values. It will print a visual gradient from black to white horizontally. On this gradient, it will plot the adjustment curve (in opposing grey values), yellow horizontal lines bounding the maximum and minimum values, and the equation used to calculate the correction in red. By default it will use a sequence from 1 to 0 to plot the curve, but you can use any sequence. Let's take 1000 random normal draws:

```
set.seed(9999)
xnorm <- rnorm(1000)
```

First, we'll see what happens when we change the weight parameter.

Figure 13: Default for `greycurve()`, weighted for small values.

```
greycurve(xnorm)

 ## Error:  error in evaluating the argument 'x'
 in selecting a method for function 'as.matrix':
 Error in gray(sort(adj)) :  invalid gray level,
                must be in [0,1].
```

Figure 14: weighting for large values.

```
greycurve(xnorm, gweight = 2)

 ## Error:  error in evaluating the argument 'x'
 in selecting a method for function 'as.matrix':
 Error in gray(sort(adj)) :  invalid gray level,
                must be in [0,1].
```

If you change the adjustment parameter, `gadj`, the shape of the curve will change, and if you change the limit parameter, `glim`, the bounds of the curve will change.

This function also will plot scalebars for use with minimum spanning networks produced in previous versions of poppr. Just use the scalebar argument and you will get a scalebar using the quantiles from the data and a scalebar with the data smoothed from the minimum to the maximum.

Figure 15: We can see that more data lies around zero (as expected with a random normal distribution)

```
greycurve(xnorm, scalebar = TRUE)

 ## Error:  error in evaluating the argument 'x'
 in selecting a method for function 'as.matrix':
 Error in gray(sort(adj)) :  invalid gray level,
                must be in [0,1].
```

Figure 16: Same as the figure on the left, but weighting heavily toward larger values.

```
greycurve(xnorm, gweight = 2, scalebar = TRUE)

 ## Error:  error in evaluating the argument 'x'
 in selecting a method for function 'as.matrix':
 Error in gray(sort(adj)) :  invalid gray level,
                must be in [0,1].
```

4.8.3 Function: bruvo.msn

This function will automatically draw a minimum spanning network of MLGs based on Bruvo's distance. It's important to note that this will recalculate Bruvo's distance each time it is run, but the amount of time it takes to run is on the order of seconds. It will return a list containing the network, the populations and the related colors in the network so you can export or redraw it with the legend if you wanted to using the package *igraph* (type `help("plot.igraph")` for details).

Default Command:

```
bruvo.msn(pop, replen = 1, add = TRUE, loss = TRUE, palette = topo.colors,
    sublist = "All", blacklist = NULL, vertex.label = "MLG",
    gscale = TRUE, glim = c(0, 0.8), gadj = 3, gweight = 1, wscale = TRUE,
    showplot = TRUE, ...)
```

- `pop` - a `genind` object.
- `replen` - see `bruvo.dist`, above.
- `add` - For missing data: use the genome addition model (see Special cases of Bruvo's distance.)
- `loss` - For missing data: use the genome loss model (see Special cases of Bruvo's distance.)
- `palette` - this is a **function** defininining a color palette to use. The default is `topo.colors`. There are different palettes, which you can search by typing `?rainbow`. If you want a custom color palette, an easy way is to use the function `colorRampPalette`.
- `sublist` - The populations you wish to analyze. This defaults to "All". See section 2.3.1 for details.
- `blacklist` - Populations you do not want to include in the graph. See section 2.3.1 for details.
- `vertex.label` - This is an option that is passed on to *igraph*'s `plot` function. *Poppr* has added two arguments specific to *poppr*. If you want to label the graph with the multilocus genotypes from the whole data set, use the argument `vertex.label = "mlg"`. If you want to display the representative individual names, you can use the argument `vertex.label = "inds"`. I say representative individual names because, only one representative from each MLG will be present in the clone corrected data set used to calculate the distance. For no labels, you can choose `vertex.label = NA`.
- `gscale` - If this is set to `TRUE`, the edge color will be converted to greyscale based on Bruvo's distance. If two nodes are closely related, the edge will appear darker. The limits of the scale can be set by the argument `glim`. If this is set to `FALSE`, all edge colors will be black.
- `glim` - This is a vector of numbers between 0 and 1. This lets you set the limits of the grey scaling based on R's internal `grey` function. For example, if you wanted a maximum of 50% white saturation (for use if you have distantly related nodes) and a minimum of 1%, you would use `glim = c(0.01, 0.5)`.
- `gadj` - This is an integer greater than zero used to adjust the scaling factor for the grey curve. Since very small changes in the grey scale are not easily precieved, it's useful to be able to adjust the grey scale to be able to show you the weights of each edge. For example, a population with most weights less than 0.3, you might want to set `gadj = 10` to exaggerate the grey scale.
- `gweight` - If `gweight = 1`, the grey scale adjustment will be weighted towards separating out smaller values of Bruvo's distance. If `gweight = 2`, the grey scale ajustment will be weighted towards separating out larger values of Bruvo's distance.

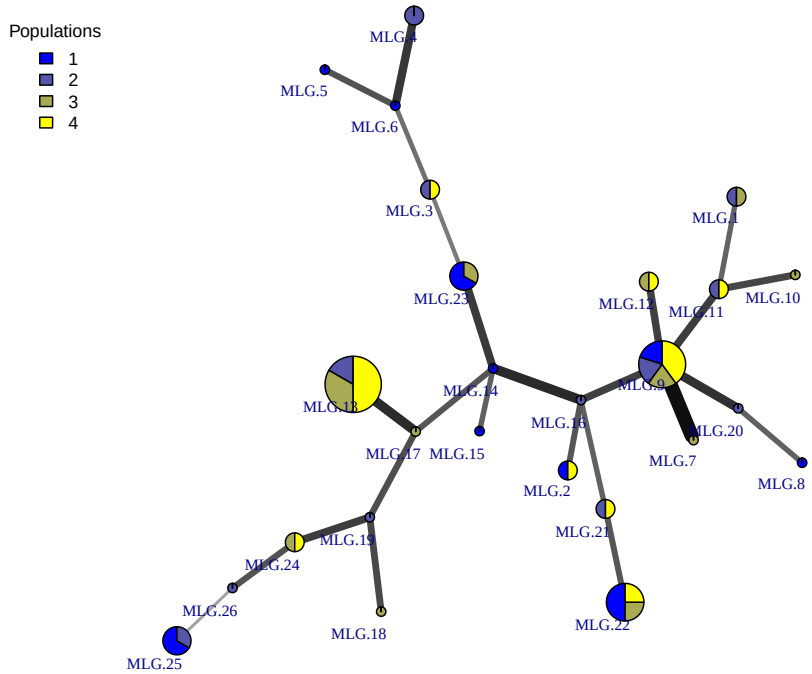
- `wscale` – If this is set to `TRUE`, edge widths will be displayed corresponding to Bruvo's distance in that thicker edges will represent a smaller distance between nodes. If this is set to `FALSE`, all edges will be set to a width of 2.

- `...` – This is a placeholder for any other arguments that you want to supply to *igraph*. Useful arguments are `vertex.label.cex` to adjust the size of the labels, `vertex.label.dist` to adjust the position of the labels, and `vertex.label.color` to adjust the color of the labels.

Often, minimum spanning networks are the preferred way to visualize Bruvo's distance. *Poppr* offers an easy way to plot these. For a demonstration, let's analyze a simulated data set of 50 individuals from populations that reproduce at a 99.9% rate of clonal reproduction.

```
data(partial_clone)
set.seed(9005)
pc.msn <- bruvo.msn(partial_clone, replen = rep(1, 10), vertex.label.cex = 0.7, vertex.label.dist = -0.5,
    palette = colorRampPalette(c("blue", "yellow")))
```

Figure 17: Minimum Spanning Network representing 4 simulated populations. Each node represents a different multi locus genotype (MLG). Node sizes and colors correspond to the number of individuals and population membership, respectively. Edge thickness and color are proportional to Bruvo's distance. Edge lengths are arbitrary.



The output, as mentioned earlier, is a list containing the graph constructed via the *igraph* package, a vector of the population names and a vector of colors representing the populations.

```
library(igraph)
pc.msn

## $graph
## IGRAPH UNW- 26 25 --
## + attr: name (v/c), size (v/n), shape (v/c), pie (v/x), pie.color (v/x), label
##   (v/c), weight (e/n), color (e/c), width (e/n)
##
## $populations
## [1] "1" "2" "3" "4"
##
## $colors
## [1] "#0000FF" "#5555AA" "#AAAA55" "#FFFF00"
```

Note that the thickness of the edges (the lines that are connecting the dots) is representative of relatedness between individuals, but the lengths do not necessarily mean anything due to the fact that with a larger data sets, displaying lengths proportional to relatedness would be impossible to draw on a 2D surface. Interpreting these data would show that MLG 9 has 5 individuals from all four populations and that it is most closely related to MLG 7, whereas the most distantly related connection exists between MLG 25 and MLG 26.

Since a graph can be represented in many ways, you might want to play around with different layouts using the `layout()` function in *igraph*. Type `help("layout", package = igraph)` for details. Below is the code for reconstructing the previous graph using the output:

```
set.seed(9005)
library(igraph)
plot(pc.msn$graph, vertex.size = V(pc.msn$graph)$size * 3, vertex.label.cex = 0.7, vertex.label.dist = -0.5,
    )
legend(-1.55, 1, bty = "n", cex = 0.75, legend = pc.msn$populations, title = "Populations",
    fill = pc.msn$colors, border = NULL)
```

4.8.4 Function: poppr.msn

Use this function to draw a minimum spanning network from your data set and a distance matrix derived from your data set. Since there are hundreds of distances that can be calculated for genetic data, and since I want to be able to graduate at some point in this decade, functions to automatically calculate distances and draw the minimum spanning networks will be few and far between. This function is an attempt to meet the user halfway and draw a minimum spanning network provided that the user has supplied two things:

1. A distance matrix over all individuals.
2. The original data set containing demographic information.

That's it. For the most part, this function is functionally the same as `bruvo.msn`, except that instead of being exclusive to microsatellite markers, you can now visualize distances in any marker type provided that you have the two items listed above.

Default Command:

```
poppr.msn(pop, distmat, palette = topo.colors, sublist = "All",
    blacklist = NULL, vertex.label = "MLG", gscale = TRUE, glim = c(0,
        0.8), gadj = 3, gweight = 1, wscale = TRUE, showplot = TRUE,
    ...)
```

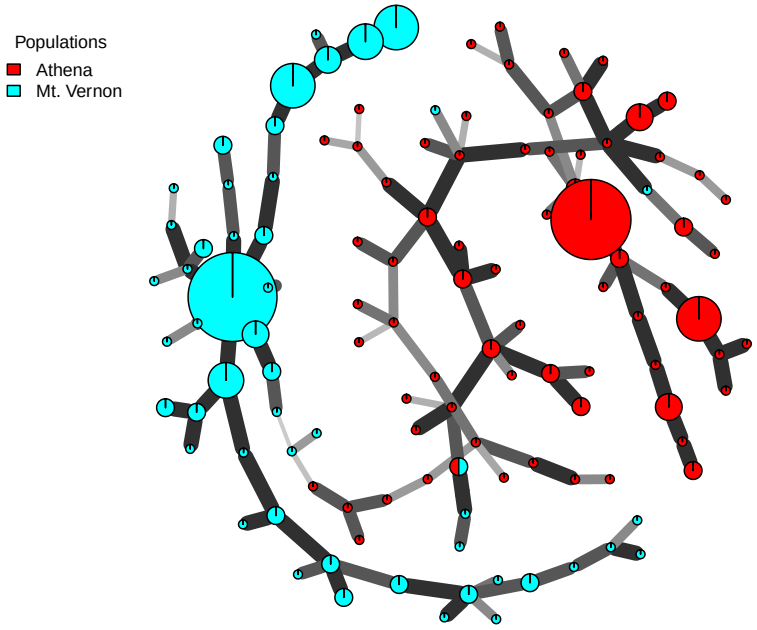
- `pop` - a `genind` object.

- `distmat` - a dissimilarity distance matrix derived from your data with distances between zero and one.
- `palette` - this is a **function** definining a color palette to use. The default is `topo.colors`. There are different palettes, which you can search by typing `?rainbow`. If you want a custom color palette, an easy way is to use the function `colorRampPalette`.
- `sublist` - The populations you wish to analyze. This defaults to "All".
- `blacklist` - Populations you do not want to include in the graph.
- `vertex.label` - This is an option that is passed on to *igraph*'s `plot` function. *Poppr* has added two arguments specific to *poppr*. If you want to label the graph with the multilocus genotypes from the whole data set, use the argument `vertex.label = "mlg"`. If you want to display the representative individual names, you can use the argument `vertex.label = "inds"`. I say representative individual names because, only one representative from each MLG will be present in the clone corrected data set used to calculate the distance. For no labels, you can choose `vertex.label = NA`.
- `gscale` - If this is set to `TRUE`, the edge color will be converted to greyscale based on the distance. If two nodes are closely related, the edge will appear darker. The limits of the scale can be set by the argument `glim`. If this is set to `FALSE`, all edge colors will be black.
- `glim` - This is a vector of numbers between 0 and 1. This lets you set the limits of the grey scaling based on R's internal `grey` function. For example, if you wanted a maximum of 50% white saturation (for use if you have distantly related nodes) and a minimum of 1%, you would use `glim = c(0.01, 0.5)`.
- `gadj` - This is an integer greater than zero used to adjust the scaling factor for the grey curve. Since very small changes in the grey scale are not easily precieved, it's useful to be able to adjust the grey scale to be able to show you the weights of each edge. For example, a population with most weights less than 0.3, you might want to set `gadj = 10` to exaggerate the grey scale.
- `gweight` - If `gweight = 1`, the grey scale adjustment will be weighted towards separating out smaller values of the distance. If `gweight = 2`, the grey scale ajustment will be weighted towards separating out larger values of Bruvo's distance.
- `wscale` - If this is set to `TRUE`, edge widths will be displayed corresponding to Bruvo's distance in that thicker edges will represent a smaller distance between nodes. If this is set to `FALSE`, all edges will be set to a width of 2.
- `...` - This is a placeholder for any other arguments that you want to supply to *igraph*. Useful arguments are `vertex.label.cex` to adjust the size of the labels, `vertex.label.dist` to adjust the position of the labels, and `vertex.label.color` to adjust the color of the labels.

Since we have the ability, let's visualize the *A. euteiches* data set [5].

```
data(Aeut)
A.dist <- diss.dist(Aeut)
set.seed(9005)
A.msn <- poppr.msn(Aeut, A.dist, vertex.label = NA, palette = rainbow, gadj = 15)
```

Figure 18: Minimum Spanning Network representing 4 simulated populations. Each node represents a different multi locus genotype (MLG). Node sizes and colors correspond to the number of individuals and population membership, respectively. Edge thickness and color are proportional to Bruvo's distance. Edge lengths are arbitrary.



5 Diversity table {I know what you did last summary table}

Remember the summary function that you used to get all the diversity statistics in section 1.3? In this section, we will flesh out all that you can do with this function. This was the very first function that was written for *poppr* to make it easy for the user to manipulate and summarize the data in one function.

5.1 Function: poppr

This function is quite daunting with all its possibilities. You have the option to subset your data for specific populations, correct for missing data, and clone correct. With each of these possibilities, comes the need to provide all the arguments for their various functions.

Default Command:

```
poppr(dat, total = TRUE, sublist = "ALL", blacklist = NULL, sample = 0,
    method = 1, missing = "ignore", cutoff = 0.05, quiet = FALSE,
```

```
    clonecorrect = FALSE, hier = 1, dfname = "population_hierarchy",
    keep = 1, hist = TRUE, minsamp = 10, legend = FALSE)
```

- `dat` – A `genind` object, `genclone` object, or a path to a file on your machine that contains genetix, structure, fstat, genpop, or genalex formatted data.
- `total` – This is also a synonym for "pooled". This will calculate all diversity statistics on the entire data set if set to `TRUE` or if there is no population structure.
- *popsub functions:* See section 2.3

  sublist - A list of populations you want to include in your analysis.

  blacklist - A list of populations you want to exclude from your analysis.
- *shufflepop functions:* See section 2.5
  Note that this only affects the calculation for I_A and $\bar{r}_d$.

  sample - The number of samples you desire (eg. 999)

  method - Which sampling method? 1: permute, 2: parametric bootstrap, 3: non-parametric bootstrap, 4: multilocus.
- *missingno functions:* See Section 2.1
  Note that all analyses in this function ignore/impute missing data by default.

  missing - How to deal with missing data. This feeds into the `type` flag of `missingno`.

  cutoff - Allowable percentage of missing data per genotype or locus.
- `quiet` – If set to `TRUE`, nothing will be printed to the screen as the sampling progresses. If `FALSE` (default) a progress bar will be produced.
- *clonecorrect functions:* See section 2.4

  clonecorrect - if this is set to `TRUE`, then you will need to set the next two parameters.

  hier - A list of the population hierarchy, or names of columns in the data frame noted below.

  dfname - A data frame in the `@other` slot of the `genind` object containing all of the population factors in different columns. For an example, see sections 2.2 and 2.4.

  keep - A vector of integers as indexes for the `hier` flag indicating which levels of the hierarchy you want to analyze. See section 2.4 for details.
- `hist` – if `TRUE`, a histogram of distributions of I_A and $\bar{r}_d$ will be displayed with each population if there is sampling.
- `minsamp` – The minimum number of individuals you want to use to calculate the expected number of MLGs. The default is set to 10.

This function produces a table that contains the population name, number of individuals observed, number of MLGs observed, number of MLGs expected at the lowest common sampling size within the data set [8] [7], the Shannon-Wiener index [16], Stoddart and Taylor's index for expected MLGs [18], Nei's 1987 genotypic diversity [11], evenness [15][10][4], the index of association [2][17], the standardized index of association [1], and the file name. Most of these indices are calculated by converting the population into an MLG table with `mlg.table` (see section 3.3) and using the *vegan* package's `diversity` function (To see details, type `?vegan::diversity` into the R console).

To begin, let's revisit our example data set of *Aphanomyces euteiches* [5].

```
data(Aeut)
poppr(Aeut)

## | Athena
## | Mt. Vernon
## | Total
##          Pop   N MLG eMLG   SE    H    G  Hexp   E.5    Ia  rbarD File
## 1     Athena  97  70 66.0 1.25 4.06 42.2 0.986 0.721  2.91 0.0724 Aeut
## 2 Mt. Vernon  90  50 50.0 0.00 3.67 28.7 0.976 0.726 13.30 0.2816 Aeut
## 3      Total 187 119 68.5 2.99 4.56 69.0 0.991 0.720 14.37 0.2706 Aeut
```

OK, so we were able to get a table out of this. Now let's see what happens when we do some sampling to see if this is reproducing clonally or not. We will turn quiet on and the histogram off to save space.

```
poppr(Aeut, sample = 999, hist = FALSE, quiet = TRUE)
```

```
##          Pop   N MLG  eMLG    SE     H     G   Hexp    E.5     Ia  p.Ia   rbarD  p.rD
## 1     Athena  97  70 65.98 1.246 4.063 42.19 0.9865 0.7210  2.906 0.001 0.07237 0.001
## 2 Mt. Vernon  90  50 50.00 0.000 3.668 28.72 0.9760 0.7259 13.302 0.001 0.28164 0.001
## 3      Total 187 119 68.45 2.989 4.558 68.97 0.9908 0.7201 14.371 0.001 0.27062 0.001
##   File
## 1 Aeut
## 2 Aeut
## 3 Aeut
```

From now on, we'll set `quiet = TRUE` to save space on our vignette. Let's clone correct at different levels to see if that affects the index of association. First, we'll clone correct at the sub population level.

```
poppr(Aeut, sample = 999, clonecorrect = TRUE, hier = c("Pop", "Subpop"), dfname = "population_hierarchy",
    quiet = TRUE, hist = FALSE)
```

```
##          Pop   N MLG  eMLG    SE     H     G   Hexp    E.5     Ia  p.Ia   rbarD  p.rD
## 1     Athena  76  70 60.62 1.017 4.221 65.64 0.9979 0.9630  2.535 0.001 0.06222 0.001
## 2 Mt. Vernon  65  50 50.00 0.000 3.796 36.74 0.9880 0.8214 14.310 0.001 0.29775 0.001
## 3      Total 141 119 59.63 1.854 4.705 96.98 0.9968 0.8762 13.802 0.001 0.26003 0.001
##   File
## 1 Aeut
## 2 Aeut
## 3 Aeut
```

And at the population level.

```
poppr(Aeut, sample = 999, clonecorrect = TRUE, hier = "Pop", dfname = "population_hierarchy",
    quiet = TRUE, hist = FALSE)
```

```
##          Pop   N MLG  eMLG        SE     H   G   Hexp   E.5     Ia  p.Ia   rbarD  p.rD
## 1     Athena  70  70 50.00 3.147e-06 4.248  70 1.0000 1.000  2.438 0.001 0.05955 0.001
## 2 Mt. Vernon  50  50 50.00 0.000e+00 3.912  50 1.0000 1.000 13.856 0.001 0.28508 0.001
## 3      Total 120 119 49.83 3.770e-01 4.776 118 0.9999 0.995 12.497 0.001 0.23426 0.001
##   File
## 1 Aeut
## 2 Aeut
## 3 Aeut
```

As you can see, clone correction doesn't always have to involve creation of new data sets!

You might notice that the P-values for both I_A and $\bar{r}_d$ are often equal to each other. This will always be the case with the sampling method utilized in method 4 [1]. Here, we show examples where they are not equal and why it's okay.

```
set.seed(2002)
poppr(nancycats, sublist = 5:6, total = FALSE, sample = 999, method = 2, quiet = TRUE, hist = FALSE)
```

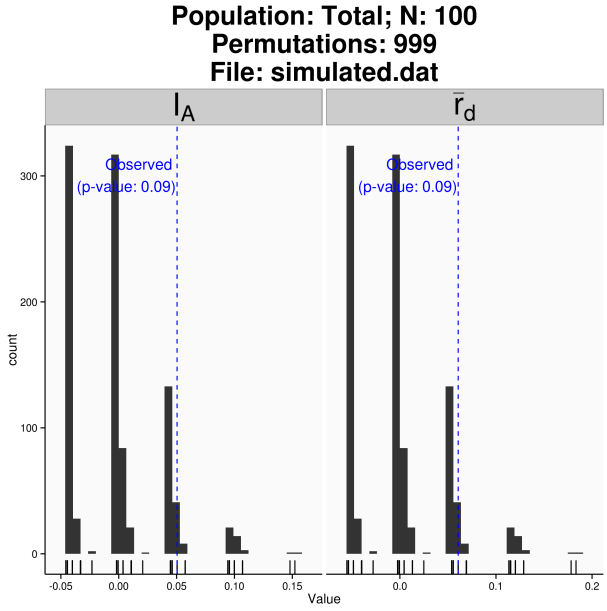
```
##   Pop  N MLG eMLG        SE     H  G Hexp E.5       Ia  p.Ia     rbarD  p.rD      File
## 1   5 15  15   11 1.192e-07 2.708 15    1   1 -0.04754 0.576 -0.006004 0.576 nancycats
## 2   6 11  11   11 0.000e+00 2.398 11    1   1  0.33370 0.070  0.042616 0.071 nancycats
```

The reason why the P-values would be different is described at the end of section 4.1.1. The differences in P-values are normally not very far off. It's important to note this because of what can happen in extremely clonal populations. You can end up with a large enough sample size consisting of very few MLGs. Upon shuffling using method 4, you find that there are very few values of I_A and $\bar{r}_d$ that can be obtained. Observe with this simulated data set:

```
set.seed(2004)
poppr(system.file("files/simulated.dat", package = "poppr"), sample = 999, method = 4, quiet = TRUE)
```

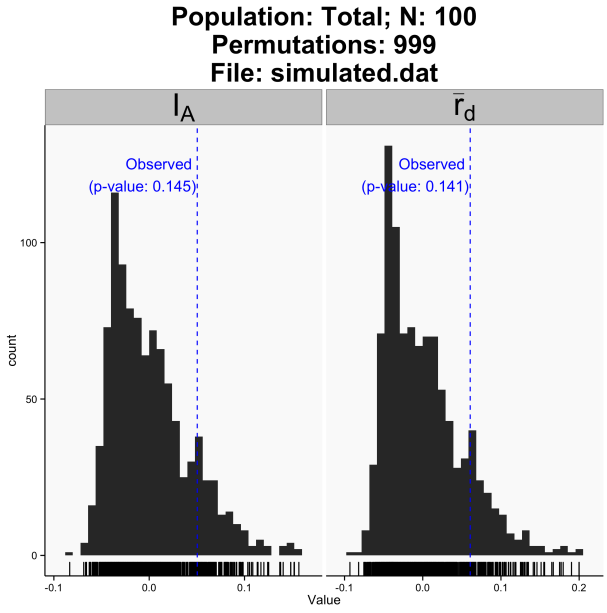
```
##     Pop   N MLG eMLG SE     H    G   Hexp    E.5     Ia p.Ia   rbarD p.rD          File
## 1 Total 100   6    6  0 1.235 2.79 0.6481 0.7346 0.0504 0.09 0.06068 0.09 simulated.dat
```

Figure 19: Output of multilocus-style sampling. Note the multi- modal distribution.



Take a look a these two histograms. The number of ways you can recombine the data with the default sampling method is very small. Other sampling methods could give a more theoretical distribution. Let's try the parametric bootstrap (For details, see section 2.5).

Figure 20: Output for parametric bootstrap sampling.



As you can see, the distribution is much closer to a distribution we would expect if this were a small sample of a larger population.

6 Appendix

6.1 General hierarchy method use

To reiterate, there are currently 5 methods that manipulate population hierarchies in genclone objects: NOTE: Refer to Accessing hierarchies for more details on how to access hierarchies.

Method	Function	Input	Result
split	splithierarchy	formula	defined hierarchical levels
set	sethierarchy	data frame	new hierarchy
get	gethierarchy	formula	data frame
name	namehierarchy	formula	new hierarchy names
add	addhierarchy	vector or data frame	new hierarchical level

These functions all have a syntax that looks like this:

```
newobject <- FUNCTION(object, input)
```

Let's take a data set of *Phytophthora infestans* collected from North America and South America and use that as an example. It has two population hierarchies defined, Continent and Country:

```
data(Pinf)
Pinf

##
## This is a genclone object
## -------------------------
## Genotype information:
##
##    72 multilocus genotypes
##    86 tetraploid individuals
##    11 codominant loci
##
## Population information:
##
##     2 hierarchical levels - Continent Country
##     2 populations defined - South America North America
```

Let's say I wanted to change the hierarchy names to Spanish. I can do that using `namehierarchy`.

```
elPinf <- namehierarchy(Pinf, ~continente/pais)  # Don't forget the formula syntax!
elPinf

##
## This is a genclone object
## -------------------------
## Genotype information:
##
##    72 multilocus genotypes
##    86 tetraploid individuals
##    11 codominant loci
##
## Population information:
##
##     2 hierarchical levels - continente pais
##     2 populations defined - South America North America

Pinf

##
## This is a genclone object
## -------------------------
## Genotype information:
##
##    72 multilocus genotypes
##    86 tetraploid individuals
##    11 codominant loci
##
## Population information:
##
##     2 hierarchical levels - Continent Country
##     2 populations defined - South America North America
```

The original data set has stayed the same and we now have a new data set with the names we want.

Of course, it would be silly to create a new data set every time we wanted to do something like change names. This is why all of the above functions (with the exception of `gethierarchy`) all have the replacement syntax of:

```
FUNCTION(object) <- input
```

NOTE: This is the preferred syntax.
This allows the object to be edited *in place* and makes things generally easier. Let's revisit our previous example:

```
Pinf

##
## This is a genclone object
## -------------------------
## Genotype information:
##
##    72 multilocus genotypes
##    86 tetraploid individuals
##    11 codominant loci
##
## Population information:
##
##     2 hierarchical levels - Continent Country
##     2 populations defined - South America North America

namehierarchy(Pinf) <- ~continente/pais
Pinf

##
## This is a genclone object
## -------------------------
## Genotype information:
##
##    72 multilocus genotypes
##    86 tetraploid individuals
##    11 codominant loci
##
## Population information:
##
##     2 hierarchical levels - continente pais
##     2 populations defined - South America North America
```

While we will mainly be using the replacement syntax in this vignette, the advantage to having both systems is that with the `function(object, input)` syntax, you can test manipulation without affecting your object.

6.2 Algorithmic Details

6.2.1 I_A and $\bar{r}_d$

The index of association was originally developed by A.H.D. Brown analyzing population structure of wheat [2]. It has been widely used as a tool to detect clonal reproduction within populations [17]. Populations whose members are undergoing sexual reproduction, whether it be selfing or out-crossing, will produce gametes via meiosis, and thus have a chance to shuffle alleles in the next generation. Populations whose members are undergoing clonal reproduction, however, generally do so via mitosis. This means that the most likely mechanism for a change in genotype is via mutation. The rate of mutation varies from species to species, but it is rarely sufficiently high to approximate a random shuffling of alleles. The index of association is a calculation based on the ratio of the variance of the raw number of differences between individuals and the sum of those variances over each locus [17]. You can also think of it as the observed variance over the expected variance. If they are the same, then the index is zero after subtracting one (from Maynard-Smith, 1993 [17]):

$$I_A = \frac{V_O}{V_E} - 1 \tag{13}$$

Since the distance is more or less a binary distance, any sort of marker can be used for this analysis. In the calculation, phase is not considered, and any difference increases the distance between two individuals. Consider the genotypes of the dummy data frame we created earlier:

```
##    locus1  locus2  locus3
## 1 101/101 201/201 301/302
## 2 102/103 202/203 301/303
## 3 102/102 203/204 304/305
```

Now, consider the first locus represented in the genind object:

```
##   L1.1 L1.2 L1.3
## 1    1  0.0  0.0
## 2    0  0.5  0.5
## 3    0  1.0  0.0
```

Remember that each column represents a different allele and that each entry in the table represents the fraction of the genotype made up by that allele at that locus. Notice also that the sum of the rows all equal one. *Poppr* uses this to calculate distances by simply taking the sum of the absolute values of the differences between rows.

The calculation for the distance between two individuals at a single locus with a allelic states and a ploidy of k is as follows[5]:

$$d = \frac{k}{2} \sum_{i=1}^{a} \mid ind_{Ai} - ind_{Bi} \mid \tag{14}$$

```
abs(dfg@tab[1, 1:3] - dfg@tab[2, 1:3])

## L1.1 L1.2 L1.3
##  1.0  0.5  0.5

abs(dfg@tab[1, 1:3] - dfg@tab[3, 1:3])

## L1.1 L1.2 L1.3
##    1    1    0

abs(dfg@tab[2, 1:3] - dfg@tab[3, 1:3])

## L1.1 L1.2 L1.3
##  0.0  0.5  0.5
```

As you can see, these values of d at locus one add up to 2, 2, and 1, respectively.

To find the total number of differences between two individuals over all loci, you just take d over m loci, a value we'll call D:

$$D = \sum_{i=1}^{m} d_i \tag{15}$$

These values are calculated over all possible combinations of individuals in the data set, $\binom{n}{2}$ after which you end up with $\binom{n}{2} \cdot m$ values of d and $\binom{n}{2}$ values of D. Calculating the observed variances is fairly straightforward (modified from Agapow and Burt, 2001) [1]:

$$V_O = \frac{\sum_{i=1}^{\binom{n}{2}} D_i^2 - \frac{(\sum_{i=1}^{\binom{n}{2}} D_i)^2}{\binom{n}{2}}}{\binom{n}{2}} \tag{16}$$

[5]Individuals with Presence / Absence data will have the $k/2$ term dropped.

Calculating the expected variance is the sum of each of the variances of the individual loci. The calculation at a single locus, j is the same as the previous equation, substituting values of D for d [1]:

$$var_j = \frac{\sum_{i=1}^{\binom{n}{2}} d_i^2 - \frac{(\sum_{i=1}^{\binom{n}{2}} d_i)^2}{\binom{n}{2}}}{\binom{n}{2}} \tag{17}$$

The expected variance is then the sum of all the variances over all m loci [1]:

$$V_E = \sum_{j=1}^{m} var_j \tag{18}$$

Now you can plug the sums of equations (16) and (18) into equation (13) to get the index of association. Of course, Agapow and Burt showed that this index increases steadily with the number of loci, so they came up with an approximation that is widely used, $\bar{r}_d$ [1]. For the derivation, see the manual for *multilocus*. The equation is as follows, utilizing equations (16), (17), and (18) [1]:

$$\bar{r_d} = \frac{V_O - V_E}{2\sum_{j=1}^{m}\sum_{k\neq j}^{m}\sqrt{var_j \cdot var_k}} \tag{19}$$

6.2.2 Bruvo's distance

Bruvo's distance between two individuals calculates the minimum distance across all combinations of possible pairs of alleles at a single locus and then averaging that distance across all loci [3]. The distance between each pair of alleles is calculated as [3]:

$$m_x = 2^{-|x|} \tag{20}$$

$$d_a = 1 - m_x \tag{21}$$

Where x is the number of steps between each allele. So, let's say we were comparing two haploid ($k = 1$) individuals with alleles 228 and 244 at a locus that had a tetranucleotide repeat pattern $(CATG)^n$. The number of steps for each of these alleles would be $228/4 = 57$ and $244/4 = 61$, respectively. The number of steps between them is then $\mid 57 - 61 \mid = 4$. Bruvo's distance at this locus between these two individuals is then $1 - 2^{-4} = 0.9375$. For samples with higher ploidy (k), there would be k such distances of which we would need to take the sum [3].

$$s_i = \sum_{a=1}^{k} d_a \tag{22}$$

Unfortunately, it's not as simple as that since we do not assume to know phase. Because of this, we need to take all possible combinations of alleles into account. This means that we will have k^2 values of d_a, when we only want k. How do we know which k distances we want? We will have to invoke parsimony for this and attempt to take the minimum sum of the alleles, of which there are $k!$ possibilities [3]:

$$d_l = \frac{\left(\min_{i\ldots k!} s_i\right)}{k} \tag{23}$$

Finally, after all of this, we can get the average distance over all loci [3].

$$D = \frac{\sum_{i=1}^{l} d_i}{l} \tag{24}$$

This is calculated over all possible combinations of individuals and results in a lower triangle distance matrix over all individuals.

6.2.3 Special Cases of Bruvo's distance

As shown in the above section, ploidy is irrelevant with respect to calculation of Bruvo's distance. However, since it makes a comparison between all alleles at a locus, it only makes sense that the two loci need to have the same ploidy level. Unfortunately for polyploids, it's often difficult to fully separate distinct alleles at each locus, so you end up with genotypes that appear to have a lower ploidy level than the organism [3].

To help deal with these situatons, Bruvo has suggested three methods for dealing with these differences in ploidy levels [3]:

- Infinite Model - The simplest way to deal with it is to count all missing alleles as infinitely large so that the distance between it and anything else is 1. Aside from this being computationally simple, it will tend to inflate distances between individuals.
- Genome Addition Model - If it is suspected that the organism has gone through a recent genome expansion, the missing alleles will be replace with all possible combinations of the observed alleles in the shorter genotype. For example, if there is a genotype of [69, 70, 0, 0] where 0 is a missing allele, the possible combinations are: [69, 70, 69, 69], [69, 70, 69, 70], and [69, 70, 70, 70]. The resulting distances are then averaged over the number of comparisons.
- Genome Loss Model - This is similar to the genome addition model, except that it assumes that there was a recent genome reduction event and uses the observed values in the full genotype to fill the missing values in the short genotype. As with the Genome Addition Model, the resulting distances are averaged over the number of comparisons.
- Combination Model - Combine and average the genome addition and loss models.

As mentioned above, the infinite model is biased, but it is not nearly as computationally intensive as either of the other models. The reason for this is that both of the addition and loss models requires replacement of alleles and recalculation of Bruvo's distance. The number of replacements required is equal to the multiset coefficient: $\left(\binom{n}{k}\right) == \binom{(n-k+1)}{k}$ where n is the number of potential replacements and k is the number of alleles to be replaced. So, for the example given above, The genome addition model would require $\left(\binom{2}{2}\right) = 3$ calculations of Bruvo's distance, whereas the genome loss model would require $\left(\binom{4}{2}\right) = 10$ calculations.

To reduce the number of calcuations and assumptions otherwise, Bruvo's distance will be calculated using the largest observed ploidy. This means that when comparing [69,70,71,0] and [59,60,0,0], they will be treated as triploids.

6.3 Exporting Graphics

R has the ability to produce nice graphics from most any type of data, but to get these graphics into a report, presentation, or manuscript can be a bit challenging. It's no secret that the R Documentation pages are a little difficult to interpret, so I will give the reader here a short example on how to export graphics from R. Note that any code here that will produce images will also be present in other places in this vignette. The default installation of the R GUI is quite minimal, and for an easy way to manage your plots and code, I strongly encourage the user to use Rstudio http://www.rstudio.com/.

6.3.1 Basics

Before you export graphics, you have to ask yourself what they will be used for. If you want to use the graphic for a website, you might want to opt for a low-resolution image so that it can load quickly. With printing, you'll want to make sure that you have a scalable or at least a very high resolution image. Here, I will give some general guidelines for graphics (note that these are merely suggestions, not defined rules).

- **What you see is not always what you get** I have often seen presentations where the colors were too light or posters with painfully pixellated graphs. Think about what you are going to be using a graphic for and how it will appear to the intended audience given the media type.
- $\geq$ **300 dpi unless its for a web page** For any sort of printed material that requires a raster based image, 300dpi (dots per inch) is the absolute minimum resolution you should use. For simple black and white line images, 1200dpi is better. This will leave you with crisp, professional looking images.
- **If possible, save to SVG, then rasterize** Raster images (bmp, png, jpg, etc...) are based off of the number of pixels or dots per inch it takes to render the image. This means that the raster image is more or less a very fine mosaic. Vector images (SVG) are built upon several interconnected polygons, arcs, and lines that scale relative to one another to create your graphic. With vector graphics, you can produce a plot and scale it to the size of a building if you wanted to. When you save to an SVG file first, you can also manipulate it in programs such as Adobe Illustrator or Inkscape.
- **Before saving, make sure the units and dimensions are correct** Unless you really wanted to save a graph that's over 6 feet wide.

6.3.2 Image Editors

Often times, fine details such as labels on networks need to be tweaked by hand. Luckily, there are a wide variety of programs that can help you do that. Here is a short list of image editors (both free and for a price) that you can use to edit your graphics.

- Bitmap based editors (for jpeg, bmp, png, etc...)

The GIMP	Free, cross-platform. http://www.gimp.org
Paint.net	Free, Windows only. http://www.getpaint.net
Adobe Photoshop	Pricey, Windows and Mac. http://www.adobe.com/products/photoshop.html

- Scalable Vector Graphics based editors (for svg, pdf)

Inkscape	Free, cross-platform http://inkscape.org
Adobe Illustrator	Pricey, Windows and Mac. http://www.adobe.com/products/illustrator.html

6.3.3 Exporting ggplot2 graphics

ggplot2 is a fantastic package that *poppr* uses to produce graphs for the `mlg.table`, `poppr`, and `ia` functions. Saving a plot with *ggplot2* is performed with one command after your plot has rendered:

```
data(nancycats)  # Load the data set.
poppr(nancycats, sublist = 5, sample = 999)  # Produce a single plot.
ggsave("nancy5.pdf")
```

Note that you can name the file anything, and `ggsave` will save it in that format for you. The details are in the documentation and you can access it by typing `help("ggsave")` in your R console. The important things to note are that you can set a `width`, `height`, and `unit`. The only downside to this function is that you can only save one plot at a time. If you want to be able to save multiple plots, read on to the next section.

6.3.4 Exporting any graphics

Some of the functions that *poppr* offers will give you multiple plots, and if you want to save them all, using `ggsave` will require a lot of tedious typing and clicking. Luckily, R has Functions that will save any plot you generate in nearly any image format you want. You can save in raster images such as png, bpm, and jpeg. You can also save in vector based images such as svg, pdf, and postscript. The important thing to remember is that when you are saving in a raster format, the default units of measurement are "pixels", but you can change that by specifying your unit of choice and a resolution.

For raster images and svg files, you can only save your plots in multiple files, but pdf and postscript plots can be saved in one file as multiple pages. All of these functions have the same basic form. You call the function to specify the file type you want (eg. `pdf("myfile.pdf")`), create any graphs that you want to create, and then make sure to close the session with the function `dev.off()`. Let's give an example saving to pdf and png files.

```
data(H3N2)
pop(H3N2) <- H3N2$other$x$country
####
png("H3N2_barchart%02d.png", width = 14, height = 14, units = "in", res = 300)
H.tab <- mlg.table(H3N2)
dev.off()
####
```

Since this data set is made up of 30 populations with more than 1 individual, this will save 30 files to your working directory named "H3N2_barchart01.png...H3N2_barchart30.png". The way R knows how to number these files is because of the `%02d` part of the command. That's telling R to use a number that is two digits long in place of that expression. All of these files will be 14x14" and will have a resolution of 300 dots per inch. If you wanted to do the same thing, but place them all in one file, you should use the pdf option.

```
pdf("H3N2_barcharts.png", width = 14, height = 14, compress = FALSE)
H.tab <- mlg.table(H3N2)
dev.off()
```

Remember, it is important not to forget to type `dev.off()` when you are done making graphs. Note that I did not have to specify a resolution for this image since it is based off of vector graphics.

6.4 Todo

- locus_table
- missing_table
- amova
- private_alleles
- anyboot
- *.dist

6.5 Table of Functions

Below is a table of functions found in *poppr*. These functions are linked within the document. Simply click on a funciton name to go to its definition and description.

Function	Description
Import/Export	
getfile	Provides a quick GUI to grab files for import
read.genalex	Read *GenAlEx* formatted csv files to a genind object
genind2genalex	Converts genind objects to *GenAlEx* formatted csv files
as.genclone	Converts genind objects to genclone objects
Manipulation	
setpop	Set the population using defined hierarchies
splithierarchy	Split a concatenated hierarchy imported as a population
sethierarchy	Define a population hierarchy of a genclone object
gethierarchy	Extract the hierarchy data frame
addhierarchy	Add a vector or data frame to an existing hierarchy
namehierarchy	Rename a population hierarchy
missingno	Handles missing data
clonecorrect	Clone censors at a specified population hierarchy
informloci	Detects and removes phylogenetically uninformative loci
popsub	Subsets genind objects by population
shufflepop	Shuffles genotypes at each locus using four different shuffling algorithms
splitcombine*	Manipulates population hierarchy *Deprecated
Analysis	
bruvo.boot	Produces dendrograms with bootstrap support based on Bruvo's distance
bruvo.dist	Calculates Bruvo's distance
diss.dist	Calculates the percent allelic dissimilarity
ia	Calculates the index of association
mlg	Calculates the number of multilocus genotypes
mlg.crosspop	Finds all multilocus genotypes that cross populations
mlg.table	Returns a table of populations by multilocus genotypes
mlg.vector	Returns a vector of a numeric multilocus genotype assignment for each individual
poppr	Returns a diversity table by population
poppr.all	Returns a diversity table by population for all compatible files specified
Visualization	
greycurve	Helper to determine the appropriate parameters for adjusting the grey level for msn functions
bruvo.msn	Produces minimum spanning networks based off Bruvo's distance colored by population
poppr.msn	Produces a minimum spanning network for any pairwise distance matrix related to the data

References

[1] Paul-Michael Agapow and Austin Burt. Indices of multilocus linkage disequilibrium. *Molecular Ecology Notes*, 1(1-2):101–102, 2001.

[2] A.H.D. Brown, M.W. Feldman, and E. Nevo. Multilocus structure of natural populations of hordeum spontaneum. *Genetics*, 96(2):523–536, 1980.

[3] Ruzica Bruvo, Nicolaas K. Michiels, Thomas G. D'Souza, and Hinrich Schulenburg. A simple method for the calculation of microsatellite genotype distances irrespective of ploidy level. *Molecular Ecology*, 13(7):2101–2106, 2004.

[4] Niklaus J. Grünwald, Stephen B. Goodwin, Michael G. Milgroom, and William E. Fry. Analysis of genotypic diversity data for populations of microorganisms. *Phytopathology*, 93(6):738–46, 2003.

[5] N.J. Grünwald and G. Hoheisel. Hierarchical analysis of diversity, selfing, and genetic differentiation in populations of the oomycete aphanomyces euteiches. *Phytopathology*, 96(10):1134–1141, 2006.

[6] Bernhard Haubold and Richard R. Hudson. Lian 3.0: detecting linkage disequilibrium in multilocus data. *Bioinformatics*, 16(9):847–849, 2000.

[7] Kenneth L.Jr. Heck, Gerald van Belle, and Daniel Simberloff. Explicit calculation of the rarefaction diversity measurement and the determination of sufficient sample size. *Ecology*, 56(6):pp. 1459–1461, 1975.

[8] S H Hurlbert. The nonconcept of species diversity: a critique and alternative parameters. *Ecology*, 52(4):577–586, 1971.

[9] Thibaut Jombart. adegenet: a r package for the multivariate analysis of genetic markers. *Bioinformatics*, 24(11):1403–1405, 2008.

[10] J.A. Ludwig and J.F. Reynolds. *Statistical Ecology. A Primer on Methods and Computing*. New York USA: John Wiley and Sons, 1988.

[11] Masatoshi Nei. Estimation of average heterozygosity and genetic distance from a small number of individuals. *Genetics*, 89(3):583–590, 1978.

[12] Jari Oksanen, F. Guillaume Blanchet, Roeland Kindt, Pierre Legendre, Peter R. Minchin, R. B. O'Hara, Gavin L. Simpson, Peter Solymos, M. Henry H. Stevens, and Helene Wagner. *vegan: Community Ecology Package*, 2012. R package version 2.0-5.

[13] R. Peakall and P. E. Smouse. GENALEX 6: genetic analysis in excel. population genetic software for teaching and research. *Molecular Ecology Notes*, 6(1):288–295+, 2006.

[14] Rod Peakall and Peter E. Smouse. Genalex 6.5: genetic analysis in excel. population genetic software for teaching and research—an update. *Bioinformatics*, 28(19):2537–2539, 2012.

[15] E.C. Pielou. *Ecological Diversity*. Wiley, 1975.

[16] Claude Elwood Shannon. A mathematical theory of communication. *Bell Systems Technical Journal*, 27:379–423,623–656, 1948.

[17] J M Smith, N H Smith, M O'Rourke, and B G Spratt. How clonal are bacteria? *Proceedings of the National Academy of Sciences*, 90(10):4384–4388, 1993.

[18] J.A. Stoddart and J.F. Taylor. Genotypic diversity: estimation and prediction in samples. *Genetics*, 118(4):705–11, 1988.