2

Managing and Manipulating Genetic Data

Karl W. Broman[1] and Simon C. Heath[2]

[1] *Department of Biostatistics, Johns Hopkins University, Baltimore, MD, USA*

[2] *Centre National de Genotypage, Evry Cedex, France*

2.1 Introduction

Geneticists must learn to program: for efficiency, to avoid introducing errors into data, and to make simple what would otherwise be unfeasible. If a geneticist were to learn just one programming language, Perl would be an excellent choice; it is especially valuable for the manipulation of text files, which are the input and output of most statistical genetic software.

Our ability to learn from data relies upon the accuracy and integrity of such data. Thus, it is critical that data be stored and managed with great care. The continual growth in the size and complexity of genetic data has led to an increasing need for a formal approach to data management.

Many data are in the form of a rectangle: many individuals measured at many variables. Genetic data, however, are generally of more complex form, including pedigree information and genetic maps. Moreover, no standard data format has emerged, nor does there exist a comprehensive statistical genetic software package. The analysis of genetic data generally requires the use of multiple computer programs, each having a unique data input format.

A fundamental task in statistical genetic analyses is thus the manipulation of data files in order to conform to the variety of input formats required by the variety of software tools that must be used. Such data manipulation is cumbersome,

Bioinformatics for Geneticists, Second Edition. Edited by Michael R. Barnes
 ISBN 978-0-470-02619-9 (HB) ISBN 978-0-470-02620-5 (PB)

time-consuming, and error-prone, if not impossible, without the ability to program in a language like Perl. Programming also provides the ability to automate analyses and to perform computer simulations.

In this chapter, we describe the essential issues in the management and manipulation of genetic data, focusing on the case of human linkage data, although the basic principles apply to all types of data. Towards the end of the chapter, we provide some sample snippets of Perl code, to give the reader a flavour of the language and to emphasize certain features of Perl that are especially valuable for this type of work. We include examples of code with some trepidation, as we fear that readers will run in fright from learning to program. And so we hope that if the code frightens readers, they will ignore it, initially, and focus on the essential ideas. But we also hope that readers will be persuaded by our argument that geneticists must learn to program (or hire a programmer).

2.2 Basic principles

We begin with a brief set of guiding principles for the manipulation of genetic data. Our goals are, first, to maintain the integrity of the data; second, to be as efficient as possible; and third, to ensure that results are reproducible.

2.2.1 Never modify data 'by hand'

If certain genotypes are to be removed as likely to be in error, create a file of such, and write a program that creates a new version of the data with those genotypes removed. If the data must be reformatted for a particular software package, do not edit the files directly; write a program to do so. Why? One then avoids the introduction of errors, results can be easily reproduced, and the process can be automated so that, if the primary data should change, essentially no further effort must be expended to get back to the same point. Moreover, the computer program provides a record of what was done.

We would like to emphasize the value of command-line programs over point-and-click programs for this reason. Pointing and clicking can be useful for the occasional user of software, or for preliminary, interactive analyses, but if automation is needed, pointing and clicking is far too cumbersome, and if the analysis is to be repeated (and it usually is), how much easier is the repeated run of a program than repeated pointing and clicking!

2.2.2 Be organized; keep notes

When one leaves the laboratory and sits down in front of a computer, the importance of a laboratory notebook should not be forgotten. The procedures in data analysis

are not unlike those of a laboratory experiment: there are often many steps to be taken and many choices to be made at each step. Careful account must be taken of the particular steps and the particular choices, so that the results obtained may be understood, trusted, and reproduced. Such organization requires the investment of some effort, but this is made in order to minimize future effort.

Computer programs can serve as a useful record of one's analyses. However, it is often the case that multiple short programs are written, and that each includes some flexibility (and, indeed, we will emphasize the importance of both of these features subsequently). And so further notes on the particulars of one's analyses will be desired. If copious printouts are to be avoided, a short electronic notebook might be recommended.

It is unfortunate that statisticians have not adopted the laboratory notebook tradition, especially given the growth in the size and complexity of their computer simulations. (Statisticians' simulation results are notoriously irreproducible.) We hope that they soon do.

2.2.3 Reuse code

Few tasks are performed just once in a career, and so in writing a computer program, one should consider the possibility that it may be of some use in the future. Programs should be written in a modular and reasonably general form, and explanations ('comments') should be included in order to clarify any aspects of the program that are not obvious.

One must balance current versus future effort. If a program is written that is quite specific to the current task, it cannot be reused without modification. If the program is made somewhat more general (so that, for example, file names and parameter values are specified on the command line rather than within the program), there is a greater chance that it will be reused without modification in the future. But to write the program in more complete generality may require considerably more current effort without any guarantee that the added features will ever be put to use.

Modularity of software can increase the chance that one's programming effort will be put to future use. All of one's tasks might be solved by a single long, strung-out program, but it is unlikely that the same long sequence will be required unchanged in the future. If the long program is split into many small, independent modules, it is much more likely that some individual module will be of future use, unchanged.

Documentation of software is critical, even for code that is intended only for the programmer's own use. Think of yourself 3 months or 3 years hence; will you remember what you did and be able to modify or fix your code? That the program is written with some clarity is as important as proper documentation. If extensive explanations are required, perhaps it is best that the code be rewritten so that its use is more transparent. It is important that the documentation describe not only the operation of the program, but also the assumptions that the program makes about the input data. It is all too easy to write a program, that relies on a particular feature

of a data set (for example, that the records are sorted, or that the columns in the data file are in a particular order). If the software is subsequently reused on new data that do not have this feature, the results will be incorrect. Ideally, programs should perform extensive checking of any input data, particularly if the programs are intended for reuse, but further comparisons of input and output data are recommended to ensure that the data have not become garbled due to some subtle change in data format.

2.2.4 There's probably an easier way, but . . .

The first priority in programming should be to write code that works. There are generally many approaches to any program; do not concern yourself initially (if at all) with finding the optimal solution. Another trade-off arises here: time to construct the program versus time to run the program. For tasks in data manipulation, efficiency of computation is seldom of much importance. First solve the problem. If it is later seen to be important to reduce computation time, seek a more optimal solution, but retain your initial solution as a benchmark.

2.3 Data entry and storage

Data seldom begin their life within a computer; ideally, they are transmitted directly from the measuring instrument to the computer. If data are to be entered into the computer by hand, it is best done independently by at least two people, in order to reduce the possibility of errors. Any discrepancies between the two data sets may be checked against the original data.

Data sets of small or moderate size can reasonably be stored in an office spreadsheet program, such as Microsoft Excel. It is best to insert a value in every cell, using a standardized code (such as `NA`) in any cells for which the data are missing, rather than leave some cells empty. Empty cells are ambiguous: was the value missing, or was an error in data entry made? It is best not to use special fonts (such as boldface) or colours to encode important information, as such codes will be difficult to extract from the software. Consistency in the coding throughout the data will, of course, simplify its later use.

We routinely receive data as Excel files, but convert them to comma- or tab-delimited text files prior to their use, as such text files are easily manipulated via computer programs and are generally needed for input into statistical genetic software. For much of our work, it is sufficient to maintain the data in such text files.

The increasing size and complexity of genetic data argue for the abandonment of Excel or other spreadsheets as a solution for data storage, especially as Excel is limited in the number of columns (256) and rows (about 65 000) that are allowed. We continue to use plain text files for storing extremely large data sets (e.g. genotype data on 500K SNPs), but for complex data (particularly for the maintenance of

multiple projects whose data may be pooled, or for a project with a large number of individuals measured at many phenotypes longitudinally), a formal database may be preferred. The choice of database software depends on the size and complexity of the data (as well as the budget). For smaller projects, open-source solutions, such as MySQL or PostgreSQL, can work very well. For very large collections of data, however, it might be better to use one of the commercial offerings, such as Oracle or Sybase. In any case, if data storage and handling requirements are such that a database is required, it will generally be necessary either to hire a dedicated employee who is proficient in the design, implementation, and maintenance of databases, or to buy a complete solution where the database application has already been developed. The advantage of the latter solution is that these packages generally come with support from the supplier. The disadvantage is the cost, which in many cases can be substantial (although the cost of hiring a database programmer for the first solution must not be forgotten).

We hope it is unnecessary to emphasize that all data should be backed up regularly (and automatically), with backups kept off site so that, should a catastrophe occur, minimal data are lost.

2.4 Data manipulation

The analysis of genetic linkage data involves a sequence of tasks: verify and correct relationships between individuals, identify and resolve genotyping errors, identify and resolve errors in the phenotypes and any covariates, and perform the actual analysis. Sometimes one may then conduct computer simulations to assess the performance of the statistical methods or to obtain P values that properly account for test multiplicity.

As the different tasks involve the use of different programs, and as each such program may have its own data input format, the central problem concerns the manipulation of the data files to conform to the necessary input formats. The program Mega2 (Mukhopadhyay *et al.*, 2005) can be useful in this regard: once the data are put into Mega2, the program can be used to create files conforming to most, if not all, statistical genetic software of interest. We, however, have not made use of Mega2, but instead have written our own Perl programs to convert data between formats.

It is essential, for the manipulation of genetic data files, to define a single standard format for one's work. For almost every linkage project we are involved in, the primary data arrive in a unique format. One might be tempted to write new Perl programs to convert data from each such format into that needed for each analysis program of interest. If we are involved in 20 projects and there are 12 analysis programs we wish to use, we would then need to write 240 different Perl programs. A better approach is to define our own standard format, and write Perl programs to convert data from that format to each of the 12 analysis programs, and then for each project, we write just one Perl program to convert the data to our standard form. With 20 projects

and 12 analysis programs, we then have 32 Perl programs. And for each additional project, we write just one new Perl program, rather than 12.

A second important use of Perl in genetics is the automation of analyses. A particularly important example of this concerns single-marker linkage analysis (so-called two-point analysis), in which each of about 400 markers is investigated, one at a time, for linkage to a putative disease gene. We are aware of cases in which an investigator created, 'by hand', 400 input files (one for each marker), and then ran a linkage program 400 times, again 'by hand', writing down the one or two numbers that characterize the results for each marker. The problem with this approach should be obvious. More important than the enormous waste of effort is that the manual manipulation of data files, and the transcription of the results, can be extremely error-prone. With proficiency in Perl, it is a simple matter to write a program that reads all of the genetic data, steps through the markers one at a time, creates the required input files, runs the linkage program and extracts the essential pieces of information, and finally produces a table of the results for all markers.

Finally, Perl is extremely valuable for performing computer simulations with other genetic software, either to explore the performance of an analysis method or to obtain *P* values that make proper adjustment for the multiplicity of tests performed. This task is much like that of automating analyses: one simulates data (either with Perl or someone else's program), sends it through an analysis program, extracts the interesting bits from the output, and repeats the entire process many times. The greatest advantage of Perl for simulations is in the extraction and tabulation of the one or two interesting numbers at each replicate from the copious output produced by most analysis programs. This approach can be applied to essentially any statistical genetic software.

2.5 Examples of code

In this section, we provide some examples of Perl code, in order to give the reader a flavour of the language and to emphasize certain features of Perl that are especially useful for our work. We are unsure of the value of this section for a reader with no prior Perl programming experience; such readers may wish to skip this section.

Perl programs are generally run from a terminal window in Mac OS X or Unix, or from a command shell in Windows. The Perl interpreter will be pre-installed in Mac OS X and most Unix distributions. A Windows version of Perl may be obtained from http://www.activestate.com/ActivePerl.

2.5.1 The traditional first example

A traditional first example, and closest to the simplest possible Perl program, is displayed in Figure 2.1. This program simply prints 'Hello, world!' to the screen. The

```
#!/usr/bin/perl -w

print("Hello, world!\n");
```

Figure 2.1 A simple but complete Perl program

first line is necessary for Unix and MacOS, and indicates where the Perl interpreter is located. The `-w` indicates that the Perl interpreter should provide warnings regarding various constructions in Perl that, while being strictly legal, are more likely than not to be errors.

The second line prints the desired phrase. Note that `\n` is the 'newline' character. The semicolon indicates the end of the Perl statement.

One must create a text file containing the above code. To run the program in Unix or MacOS, the file must be made 'executable', by typing, from a terminal window, `chmod +x` *`filename`*, where *`filename`* is the name of the file. The program is then run by typing the name of the file. In Windows, `chmod` is not needed. Instead, the program file must be given a name of the form *`filename`*`.pl`. The program is then run from a command shell by typing the name of that file or by typing `perl filename.pl`.

2.5.2 Combining marker data

A common issue in genetic data manipulation is the combination of genotype data from multiple input files. In an extreme case, one may be confronted with a single file for each genetic marker. In Figure 2.2, we present a Perl program for reading all files in a directory in order to combine genotype data. We are imagining here that there is a single directory containing one file for each marker, with each file having a name like `D10S1123.txt`, where D10S1123 is the marker. The files are in LINKAGE PRE format, that is to say, each line contains the family identifier, individual identifier, dad, mom, sex, and disease status and then the two alleles for that subject at that marker. The aim of the first program is to read in all of the data, to store them in such a way that we can easily work with them. This may not appear so useful in itself, but we will show in subsequent examples how the program can be extended to perform recoding of marker alleles, estimation of allele frequencies and generation of input files for the LINKAGE programs.

The first line is the usual first line for a Perl program. The second and third lines instruct Perl to be stricter in terms of what it accepts and to issue warnings for unsafe code. This is highly recommended, as without these it is very easy to make errors that can be very difficult to detect.

The main inconvenience of this is that it is now necessary to *declare* each variable before use using the `my` command. For example, in line 5, `my $dir` declares that `$dir`

```
#!/usr/bin/perl
use strict;
use warnings;

my $dir = "data";
opendir DIR, $dir or die "Cannot open directory $dir:$!\n";
my (%ped, %gtypes, @markers);
while(my $file=readdir(DIR)) {
    next unless $file =~ /(.+)\.txt$/;
    my $mark = $1;
    push @markers, $mark;
    my $idx = $#markers;
    my $infile = "$dir/$file";
    open IN, $infile or die "Cannot open $infile:$!\n";
    my $line = 0;
    while(<IN>) {
        $line++;
        my @v=split;
        if(@v<8) {
            print "Short line at $line!\n";
            next;
        }
        my ($fam,$ind,$father,$mother,$sex,$status,$g1,$g2)=@v;
        my $id="$fam\_$ind";
        $ped{$ind} = [$fam,$ind,$father,$mother,$sex,$status];
        $gtypes{$ind}[$idx] = "$g1 $g2";
    }
    close IN;
}
```

Figure 2.2 A Perl program to read data from all data files with a `.txt` extension

is a scalar variable, indicated by the dollar sign, which is here assigned the character string `data`. The content will be just the bit between the double quotation marks. The advantage of having Perl enforce pre-declaration of variables is that it is very easy to mistype a variable name, and, by default, Perl will not complain but silently create a new variable with the mistyped name. This can lead to some extremely subtle and difficult to track down bugs in programs. For all but very short programs, therefore, it is generally advised to follow the practice here of adding the `use strict;` and `use warnings;` statements to the start of your programs.

In line 6, we open a directory using a 'directory handle' `DIR`. This allows us, from line 8, to 'loop' through each file in the directory; within this `while` loop, we read one file name at a time from the directory until there are no files remaining to be read. Note that if the `opendir` command fails, the `die` statement will be executed, which stops the program and prints the message `Could not open directory $dir: $!`. The variable `$dir` is expanded in the message to give the value we assigned in line 5. The odd-looking variable `$!` is a system variable, which gives the last error message from a system command, in this case `opendir`.

In line 9, we use pattern matching to check that the file name ends in `.txt`; otherwise, that file is skipped. (This is important, because the directories '.' and '..' will be included, but should be skipped.) The code for the pattern matching is a bit complicated at first glance. The first thing to note is that a period (.) matches any character and a plus sign (+) means one or more of the previous match. To match a literal period, it is necessary to escape the period with a backslash. The dollar sign at the end of the pattern matches the end of the string. If we ignore the brackets for the moment, the code in line 9 will therefore match one or more characters terminated by `.txt`. The brackets around the first part `.+` direct Perl to store the part of the input string which matched this part of the pattern, and store it in the variable `$1`, which is assigned to the variable `$mark` in line 10.

In line 11, the marker name is appended to the end of an array of all marker names, `@markers`. The `@` symbol indicates an array: an ordered list of values, indexed by 0, 1, 2, ... The index of the last item in an array is given by `$#`*name_of_array*, so line 12 sets `$idx` to the index of the last marker added, i.e., the current marker.

In line 13, `$infile` is assigned the full file name: the directory name followed by a / followed by the simple part of the file name. Note how we can use variables inside a quoted string, and they will be expanded to give the resulting string. We then open this file in line 14, producing a 'file handle', `IN`.

In line 15, we initialize the variable `$line` to zero; this will be used to track the line number of the input file, so that errors can be reported.

From line 16, we loop through each line in the input file. In a similar way to the while loop starting at line 8, this loop will exit when there are no more lines to be read.

In lines 17–18, we increment the line number and split the line into fields separated by white space (any combination of non-printing characters such as spaces or tabs), storing the results in the array `@v`. Lines 19–22 then check that there are at least eight columns of data; if there are fewer, we print an error message and skip to the next line.

In line 23, we assign the contents of the array `@v` to the individual variables, `$fam`, `$ind`, etc.

In lines 24–26, we store the information on the individuals' parents and sex, using 'hashes'. (This is rather difficult for beginning Perl programmers, but hashes are extremely valuable for this sort of work, as we will see in the next example.) A hash is like an array, but the hash is keyed by an arbitrary character string rather than indexed by numbers 0, 1, 2, ... Here we create a unique identifier for an individual by concatenating the family and individual identifiers together with an underscore character between them in line 24. Note here that we escape the underscore after `$fam` because otherwise Perl would take it as part of the variable name. We then store the pedigree information and genotype information in lines 25–26 keyed by his unique identifier. Note that for the genotype, we also index with the variable `$idx` (from line 8), which indicates which marker we are working on. We use braces {} for the variable `$id` and square brackets `[]` for the marker index at line 26 to indicate

to Perl that `$id` should be treated as a hash key and `$idx` should be treated as a conventional numeric index. It is not important to understand the details of how the data are stored in lines 25–26; the key point is that with the individuals' identifiers, we can access their pedigree information and genotype data.

We could avoid using hashes if we could assume that the same individuals appear in each input file in the same order. We could then just index the data by the line number. However, it is not always safe to make this assumption; in general, it is safer to use hashes.

At line 28, the input file has all been read in, so we close the file, and continue with the next file, if present.

2.5.3 Recoding alleles

The program in Figure 2.2 would be more useful if it could do some basic data manipulation. One such manipulation that is often required is allele recoding. Many programs for genetic analysis expect marker alleles to be coded from 1 up to the number of alleles present. The raw data, however, rarely come in this form. Microsatellite data come as allele sizes such as 180 or 225, and SNP data typically come as a series of nucleic acid codes (A, C, G or T). It is simple to use hashes in Perl to recode alleles, and this is a good illustration of the power of hashes. The strategy is to use the original allele code as the key to the hash. We can use this to check whether a numeric code has already been assigned to this allele and, if not, assign it the next available code.

In Figure 2.3, we provide a modification of the program in Figure 2.2 which will enable the program to recode the marker alleles into consecutive numeric codes starting from 1. The key additions are from lines 26–37. We start at line 26 by checking that the first allele is non-zero. (Zero typically indicates a missing value.) We then check whether this allele has already been encountered for this marker by checking the array `@recode`, which is indexed by the marker index `$idx` and the allele `$g1`. If not, then at line 28 we assign the next available code for this marker (stored in the array `@n_alleles`, and then at line 29 we change the original allele code to the numeric code. The same procedure is then followed for the second allele `$g2`. Note that doing this procedure without hashes would be a much more complex operation involving sorting and searching through the list of marker alleles.

2.5.4 Estimating allele frequencies

Another useful function of the program is to estimate allele frequencies, as most genetic analysis programs require these, and good estimates matched with the data set are not always available. In this case we can obtain allele estimates by simply counting the alleles in observed individuals. While marker allele frequencies are best estimated on the basis of unrelated individuals, such as the founding individuals in a set of pedigrees, genotypes of such founders are sometimes not available, and simple allele

```
#!/usr/bin/perl
use strict;
use warnings;

my $dir = "data";
opendir DIR, $dir or die "Cannot open directory $dir:$!\n";
my (%ped, %gtypes, @markers, @n_alleles, @recode);
while(my $file=readdir(DIR)) {
    next unless $file =~ /(.+)\.txt$/;
    my $mark = $1;
    push @markers, $mark;
    my $idx = $#markers;
    my $infile = "$dir/$file";
    open IN, $infile or die "Cannot open $infile:$!\n";
    my $line = 0;
    while(<IN>) {
        $line++;
        my @v=split;
        if(@v<8) {
            print "Short line at $line!\n";
            next;
        }
        my ($fam,$ind,$father,$mother,$sex,$status,$g1,$g2)=@v;
        my $id="$fam\_$ind";
        $ped{$ind} = [$fam,$ind,$father,$mother,$sex,$status];
        if($g1 != 0) {
            if(!$recode[$idx]{$g1}) {
              $recode[$idx]{$g1} = ++$n_alleles[$idx];
            }
            $g1 = $recode[$idx]{$g1};
        }
        if($g2 != 0) {
            if(!$recode[$idx]{$g2}) {
              $recode[$idx]{$g2} = ++$n_alleles[$idx];
            }
            $g2 = $recode[$idx]{$g2};
        }
        $gtypes{$ind}[$idx] = "$g1 $g2";
    }
    close IN;
}
```

Figure 2.3 A Perl program to read data from all data files with a `.txt` extension and recode marker alleles

counting provides unbiased estimates, without the great computational effort that can be required to account for the relationships between individuals (Broman, 2001).

Since we have already recoded the alleles to consecutive numbers in the previous example, it is simple to add a section to the program in Figure 2.3 to accumulate allele count information and to estimate allele frequencies. Figure 2.4 contains a snippet of Perl code which should go at the end of the previous program. It will estimate allele frequencies, and store them in the double indexed `@freq` so that `$freq[$i][$j]` will have the estimated frequency of allele `$j` of marker `$i`.

```
1  my (@freq, @count);
   for my $ind(keys %ped) {
       my $gt = $gtypes{$ind};
       for my $i(0..$#markers) {
5          my $g=$$gt[$i] || "0 0";
           my @all=split " ",$g;
           for my $j(0..2) {
               if($all[$j]) {
                   $freq[$i][$all[$j]]++;
10                 $count[$i]++;
               }
           }
       }
   }
15 for my $i(0..$#markers) {
       my @fq=@{$freq[$i]};
       for my $j(1..$#fq) {
           $fq[$j] /= $count[$i];
       }
20 }
```

Figure 2.4 A snippet of Perl for calculating marker allele frequencies

The first line of the snippet simply declares the arrays `@freq` and `@count`, where the former was described in the previous paragraph, and the latter will keep a count of the number of alleles observed for a given marker.

In line 2, we loop through all individuals for whom we have pedigree information, that is, every individual we read in previously, and then in line 4 we loop through the genotypes for each marker for this individual. If an individual did not appear in all of the input files, some of the genotypes will be undefined, and attempting to work with them will give a warning. We avoid this in line 5 by using the string '`0 0`' for any undefined genotype.

In line 6, we split the genotype on spaces to get the two alleles, and in lines 7–12 we loop through the two alleles, accumulating the counts for all non-zero alleles, and a total count for the marker. After this, it is just necessary to loop through each marker, and for each allele at each marker, and divide the allele counts by the total number of counts for that marker. This is done in lines 15–20.

We can see that the logic of the frequency estimation is very simple, but it is so simple because we have already recoded the alleles numerically, using hashes in the previous example. If we had not done this, the operation would have been much more complicated.

2.5.5 Automating single-marker analyses

Now that we have the alleles recoded and have obtained allele frequency estimates, there are many things we could do. For example, we could print out the number

```
1   my @lod;
    for my $i(0..$#markers) {
        my $datfile = "datafile.dat";
        open OUT, ">$datfile" or die "Cannot open $datfile for writing: $!\n";
5       print OUT "2 0 0 5\n0 0.0 0.0 0\n 1 2\n";
        print OUT "1 2 # Trait locus (2 alleles)\n";
        print OUT "0.999 0.001 # Disease allele frequency\n";
        print OUT "1 # Liability class\n";
        print OUT "0.0 0.0 1.0 # Recessive model\n";
10      print OUT "3 $n_alleles[$i] # $markers[$i]\n";
        my @fq=@{$freq[$i]};
        print OUT join (" ",@fq[1..$#fq]),"\n";
        print OUT "0 0\n0.0\n";
        print OUT "1 0.05 0.45 # Recombination varied, increment, last value\n";
15      close OUT;
        my $pedfile = "pedfile.pre";
        open OUT, ">$pedfile" or die "Cannot open $pedfile for writing: $!\n";
        for my $ind(keys %ped) {
            my $p = $ped{$ind};
20          print OUT join ("\t",@$p);
            my $gt = $gtypes{$ind}[$i] || "0 0";
            print OUT "\t$gt\n";
        }
        close OUT;
25      my $results_file = "tempout.txt";
        system("makeped $pedfile pedfile.dat n > $results_file");
        system("unknown >> $results_file");
        system("mlink >> $results_file");
        open IN, $results_file or die "Cannot open $results_file for input: $!\n";
30      my $theta;
        while(<IN>) {
            if(/^THETAS\s+(\S+)/) {
                $theta=$1;
            } elsif(/LOD SCORE =\s+(\S+)/) {
35              $lod[$i]{$theta} = $1;
                print "$markers[$i]\t$theta\t$1\n";
            }
        }
        close IN;
40  }
```

Figure 2.5 A snippet of Perl for running MLINK for each of many markers

of observations per marker, and obtain estimates of the success rate per marker. We could equally well count the number of observations per individual, and check whether a particular DNA sample appears to have worked less well than others. These are all important steps in the quality control of the genotyping process. We are not going to go into more details about these analyses, but instead we will finish with a demonstration of how we could use the previous examples to automate single-marker (i.e. two-point) linkage analysis with MLINK from the LINKAGE (Lathrop *et al.*, 1984) or FASTLINK (Cottingham *et al.*, 1993) packages.

The snippet of Perl in Figure 2.5 should go at the end of the previous examples in order to function properly. We first declare the array `@lod`, which will store the calculated LOD scores for each marker at each theta value. We loop over all possible markers (line 2), and then write the necessary information to a locus data file (lines

3–15), and a pedigree file (lines 16–24). In line 4, the greater-than sign in '`>$datfile`' is used to open the file for writing (as opposed to reading, as in Figure 2.2). In lines 13 and 20, `join` is used to write out each element of an array, in turn separated by a space character at line 13, and a tab character at line 20.

In lines 26–28, `system` is used to request the operating system to execute the specified commands; this is where the real work is done. Note that a greater-than sign is used to have the program output sent to a file, and two greater-than signs together indicate that the output should be appended to the file, rather than replace the file.

In the remainder of this Perl snippet, we read through the output of MLINK, pulling out the LOD score at each recombination fraction, and store this information in a hash. Hence we can run MLINK for each marker, one at a time, and distil and assemble the few essential numbers from its profuse output, which can then be written to a file, or form a part of subsequent calculations, as, for example, in the calculation of heterogeneity LOD scores.

We congratulate readers who have persevered through the sample Perl code and our brief explanations. We hope that several of the techniques and idioms that we have demonstrated in these examples can be adapted by readers for use in more general situations. While the code looks quite complicated, the language is not as difficult to learn as it may appear, and the great power that comes from knowledge of Perl well justifies the effort that must be made to acquire it.

2.6 Resources

There are numerous books on Perl; we recommend those published by O'Reilly: *Learning Perl* (Schwartz *et al.*, 2005) for the novice, *Programming Perl* (Wall *et al.*, 2000) as a reference, and *Perl Cookbook* (Christiansen and Torkington, 2003) for recipes encompassing many common tasks. These books, plus a couple of others, may be purchased together on a CD at a very good price: the *Perl CD Bookshelf*.

There are numerous online tutorials on Perl; links to some are available at http://www.biostat.jhsph.edu/∼kbroman/perlintro. This web page also contains a sample Perl program for genetic data manipulation, with line-by-line explanations. The Cold Spring Harbor Laboratory (CSHL) held a bioinformatics course in autumn 2004 that included a great deal on Perl programming; all of the lecture notes are available online at http://stein.cshl.org/genome_informatics.

Enormous amounts of useful Perl code may be obtained from the Comprehensive Perl Archive Network (CPAN) at http://cpan.perl.org. The CSHL lecture notes (mentioned above) provide good explanations of how to find and install code from CPAN. The reader may also be interested in Bioperl (http://www.bioperl.org): Perl tools for bioinformatics and genomics research, mostly for sequence data. Readers interested in the use of Perl for sequence data may wish to look at Tisdall (2001, 2003). Moorhouse and Barry (2004) will also be of interest.

Mega2 (Mukhopadhyay *et al.*, 2005), a program to facilitate the handling of genetic linkage data, is available at http://watson.hgen.pitt.edu/register.

2.7 Summary

The ever-increasing size and complexity of genetic data has led to an increasing need for geneticists to learn computer programming. As the most fundamental task for the genetic data analysis involves the manipulation of data files, proficiency in a computer language, such as Perl, with which such manipulation of text files is most natural, is recommended. For large, complex data sets, the use of a formal database, such as MySQL, in place of spreadsheet software, such as Microsoft Excel, may be important for the maintenance of data integrity and fidelity. Never modify data by hand, be organized and keep notes, and plan for the future but get the job done. Learn Perl!

Acknowledgements

Andrew Broman, Ken Manly, and Fernando Pineda generously provided comments to improve the manuscript. This work was supported in part by NIH/NHGRI grant GM074244 (to K.W.B.).

References

Broman, K. W. (2001). Estimation of allele frequencies with data on sibships. *Genet Epidemiol* **20**, 307–315.

Christiansen, T. and Torkington, N. (2003). *Perl Cookbook*, 2nd edn. Sebastopol, CA: O'Reilly Media.

Cottingham, R. W., Idury, R. M. and Schaffer, A. A. (1993). Faster sequential genetic linkage computations. *Am J Hum Genet* **53**, 252–263.

Dwyer, R. A. (2003). *Genomic Perl.* Cambridge: Cambridge University Press.

Lathrop, G. M., Lalouel, J. M., Julier, C. *et al.* (1984). Strategies for multilocus linkage analysis in humans. *Proc Natl Acad Sci USA* **81**, 3443–3446.

Moorhouse, M. and Barry, P. (2004). *Bioinformatics Biocomputing and Perl.* Chichester: Wiley.

Mukhopadhyay, N., Almasy, L., Schroeder, M. *et al.* (2005). Mega2: data-handling for facilitating genetic linkage and association analyses. *Bioinformatics* **21**, 2556–2557.

Reese, G., Yarger, R. J. and King, T. (2002). *Managing and Using MySQL*, 2nd edn. Sebastopol, CA: O'Reilly Media.

Schwartz, R. L., Phoenix, T. and Foy, B. D. (2005). *Learning Perl*, 4th edn. Sebastopol, CA: O'Reilly Media.

Tisdall, J. (2001). *Beginning Perl for Bioinformatics.* Sebastopol, CA: O'Reilly Media.

Tisdall, J. (2003). *Mastering Perl for Bioinformatics.* Sebastopol, CA: O'Reilly Media.

Wall, L., Christiansen, T. and Orwant, J. (2000). *Programming Perl*, 3rd edn. Sebastopol, CA: O'Reilly Media.