7. Evrimsel süreçlerin DNA dizi verisinden analizi

Uygulama

Working with DNA sequence data

For the first part of the tutorial, we learn about how to handle **DNA sequence** data in R. To keep things simple at this stage, we will use a relatively small dataset. Once we have some familiarity with sequence data and the statistics we can calculate from it, we will learn how to handle data derived from whole genome sequencing. For much of this first part of the tutorial, we will use a series of functions in `ape` and also some from `pegas`.

Reading sequence data into R.

The first thing we will learn to do today is to read seqeunce data into R. Unlike in the previous tutorials, it doesn't make much sense to store sequence data as a `data.frame`, so we need to use functions specifically designed for the purpose such as `read.dna`. We will be reading in a **FASTA** which is a standard file format for sequencing data (https://en.wikipedia.org/wiki/FASTA_format). FASTA files can vary slightly, but the basic format is the same. The file we will use today can be downloaded here (https://evolutionarygenetics.github.io/example.fas).

Before we actually read this file into R, let's take a look at it:

```
file.show("example.fas")
```

The `file.show` function is quite self-explanatory. You can also click on the FASTA in the `Files` pane in R Studio and have a look at it in the console. In this FASTA file, there are two lines for each sequence - the sequence name and then the sequence itself. The sequence name line starts with `>` and then after it are the base calls. Again, it is worth checking out the wikipedia page for FASTA (https://en.wikipedia.org/wiki/FASTA_format) as it contains more information on how these files are formatted.

Next we will actually read our FASTA file in. To do this, we will use `read.dna` like so:

```
mydna <- read.dna("example.fas", format = "fasta")
```

This will create an object, `mydna` which is an `ape` specific structure called a `DNAbin` - i.e. DNA data stored in binary. You don't need to worry too much about the specifics of this data structure although you might want to check it for yourself. Try `class(mydna)` and `?DNAbin` to get more information.

In the meantime, have a look at the `mydna` object and see what information is printed to the R console.

Exploring DNA sequence data

As you will have just seen, when you call the `mydna` object, you only get a summary in the R console. What if you actually want to look at the sequences? One way to do this is to use the `as.alignment` function - like so:

```
myalign <- as.alignment(mydna)
```

This function simply converts our `DNAbin` object into an `alignment` - which is essentially a list with different elements - i.e `nb` - the number of sequences, `seq` - the actual sequences themselves, `nam` - the names of the sequences.

You can look at the sequences as a character vector using the following R code:

```
myalign$seq
```

But this is not really that useful, other than showing you a way of looking at the sequences. Furthermore, since the DNA sequence is no longer stored in a binary manner - i.e. as a `DNAbin`, this is a very clunky method of looking at your sequences. A better method is to use the `alview` function - like so:

```
alview(mydna)
```

Note that in this case, we used `alview` directly on our `mydna` object - **not** on the `myalign` alignment object. This prints our alignment to the screen and it makes it immediately obvious where there are nucleotide polymorphisms in our data. You should note however that the `N` bases in the first sequence are bases which could not be called by the sequencing machine - they are not valid base calls.

NOTE it is important that we make clarify what we mean by alignment here. In this case, these sequences were read into R as an alignment already - meaning that each position in the sequences corresponds to one another. We did not actually align the sequences in R itself.

Calculating basic sequence statistics

Base composition

Using a couple of standard `pegas` functions, we can get some more information about our sequences. For example, we can calculate the frequency of the four nucleotides in our dataset:

```
base.freq(mydna)
```

We can also calculate the GC content - i.e. the proportion of the sequence that is either a G or C nucleotide.

```
GC.content(mydna)
```

To break down GC content even further, this is equivalent to:

```
sum(base.freq(mydna)[c(2, 3)])
```

Segregating sites

Looking at our aligned sequences, we can see that there are several positions whether there is a polymorphism. Using the `seg.sites` function, we can actually count:

```
seg.sites(mydna)
```

`seg.sites` returns the position in the sequence where polymorphisms occur. So these are essentially the indices of polymorphsims in our 40 base pair sequence alignment. There are just a few here, so you can easily count them but if there were many, you might want to wrap this command with the length function to get the actual number - i.e. `length(seg.sites(mydna))`.

As this function makes pretty clear, segregating sites is just the number of polymorphic positions in a sequence. However, it is also something that scales with sample size - i.e. the more sequences we add, the higher the probability of finding segregating sites in our data. This is a point we will return to later.

One last thing about the segrating sites we calculated here - it is not standardised to the length of sequences. To achieve that we need to do the following.

```
# get segregating sites
S <- length(seg.sites(mydna))
# set sequence length
L <- 40
# standardise segregating sites by sequence length
s <- S/L
```

Nucleotide diversity

Nucleotide diversity or Π is quite a straightforward statistic - it is simply the average number of differences between sequences in a population or sample. We can calculate it using the following formula:

$$\Pi = \frac{1}{[n(n-1)]/2} \sum_{i<j} \Pi_{ij}$$

Where Π_{ij} is the number of nucleotide differences between sequence i and sequence j and $[n(n-1)]/2$ is the number of possible pairwise sequence comparisons from all n sequences. This seems a little tricky but it is actually very simple.

From `seg.sites(mydna)`, we know there are two polymorphic positions at sites 35 and 36 in our 40 bp alignment. Using `alview(mydna)` we can visualise these and also actually count the differences. There are 2 nucleotide differences between sequences `No305` and `No304`, 1 between `304` and `306` and then finally, 1 again between `305` and `306`.

We can use R to calculate our nucleotide diversity by hand:

```
# set n sequences
n <- 3
# set differences
Pi_ij <- c(2, 1, 1)
# calculate the number of pairwise comparisons
np <- (n*(n-1))/2
# calculate the nucleotide diversity
Pi <- sum(Pi_ij)/np
```

You can compare this to the calculations in Chapter 7 of the textbook - as they are essentially identical. So what we have here is the average number of nucleotide differences between our three sequences. This is *not* standardised to the sequence length, so we need to do that next.

```
# calculate standarised nucleotide diversity
L <- 40
pi <- Pi/L
```

Of course, calculating nucleotide diversity by hand is not that useful when we have more than even a few sequences (as we will shortly) Luckily. we can also calculate nucleotide diversity using the function `nuc.div` from `pegas`.

```
nuc.div(mydna)
```

Unlike the number of segregating sites, this estimate is already standardised to the length of our sequence. Take a moment to compare the output previous command with our hand calculated value of `pi`. They are different! Why is this? Well using `alview` again we can see that there are actually two sites where in the first sequence, `No305`, the base call is `N`. As we learned earlier, this means we do not have a reliable call for this position in the sequence (perhaps a sequencing error or an ambigous base).

The `nuc.div` function therefore corrects for these missing bases and shortens the total length of our sequence by two. This therefore changes the final estimate of π it produces.

Working with a larger dataset

As we have seen in previous tutorials, R packages often come with nice, ready-to-use datasets. `ape` and `pegas` are no exception. We will turn to the `woodmouse` dataset next to learn how we can work with larger data than just 3, 40 bp sequences. To load the data, you need to do the following:

```
data(woodmouse)
woodmouse
```

What is this data? It is 15 sequences the mitochondrial cytochrome b gene of the woodmouse (*Apodemus sylvaticus*), a very cute and widespread rodent in Europe (https://en.wikipedia.org/wiki/Wood_mouse). The data is a subset of a study (https://onlinelibrary.wiley.com/doi/full/10.1046/j.1365-294X.2003.01752.x) by the authors of `pegas`.

We can easily view the `woodmouse` sequences using `alview`.

```
alview(woodmouse)
```

However, you can see already that with this much data, looking at the alignment directly is not that practical - and we certainly wouldn't ask you to count all the polymorphic sites here! Instead it makes much more sense to use the R functions we just covered to calculate the statistics of interest.

With a larger dataset, we can also manipulate our `woodmouse` `DNAbin` object a bit more directly. Although we do not see a matrix of aligned DNA sequences when we call `woodmouse`, we can still treat it like one - including using R indices. In this sense, the rows of our matrix are our individuals and the columns are the number of base pairs in our sequence. So for example:

```
woodmouse[1:10, ]
```

This command will return only the first 10 sequences. Manipulating the columns will also change how much of the sequence we retain:

```
woodmouse[, 1:100]
```

In the next section, we will use this to demonstrate how sample size can influence our estimates of nucleotide diversity and segregating sites.

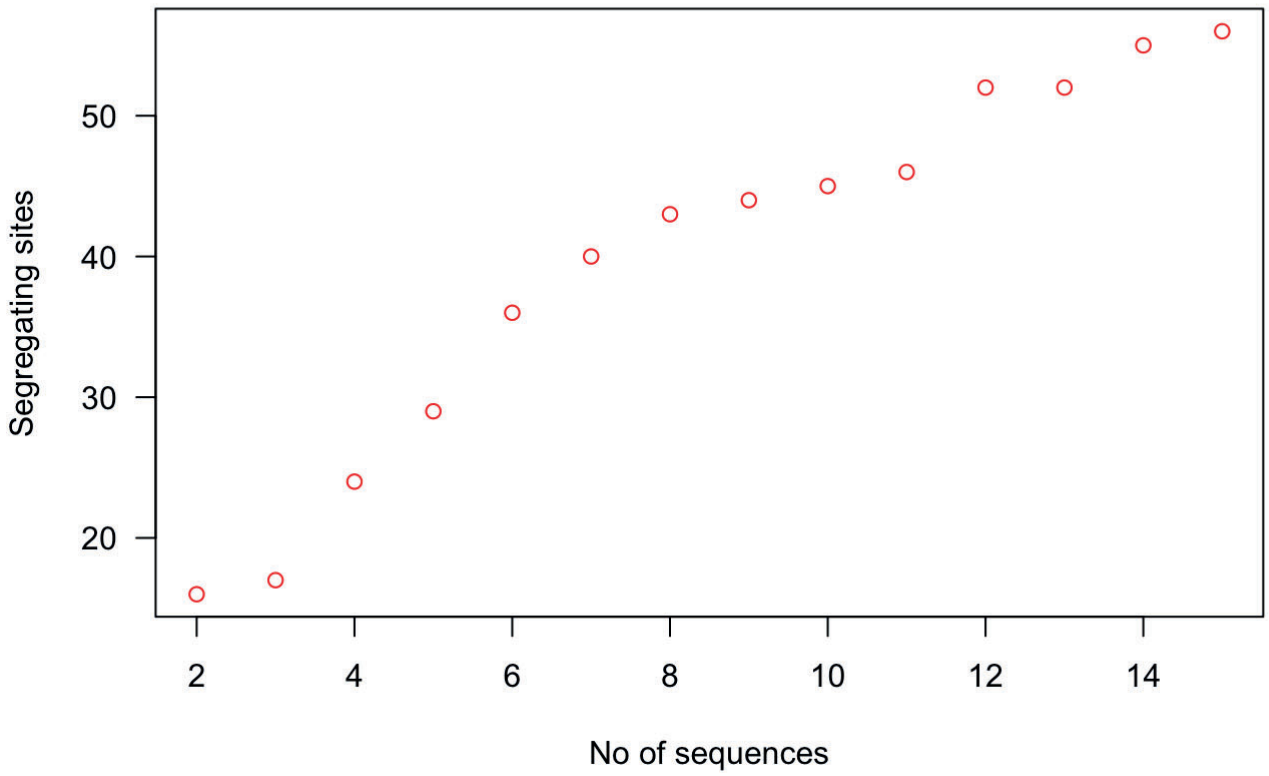
Sample size and sequence statistics

First of all, let's turn our attention to the number of segregating sites. How does using different numbers of sequences effect this? We can use `sapply` here to loop through different values of the maximum number of sequences. So we will first use 2 sequences, then 3, then 4 and so on, until we use all 15.

```
# use sapply to loop
ss <- sapply(2:15, function(z){
  length(seg.sites(woodmouse[1:z, ]))
})
```

With our `sapply` code, we set the `function(z)` argument, meaning that each value of 2, 3, 4 up to 15 is denoted by `z`. Then within the curly brackets we simply calculate the number of segregating sites each time. Have a look at the `ss` vector this creates - it is the number of segregating sites for each maximu number of sequences. However, the relationship is a lot easier if we plot it, like so:

```
# plot figure
plot(2:15, ss, col = "red", xlab = "No of sequences", ylab = "Segregating sites", las = 1)
```



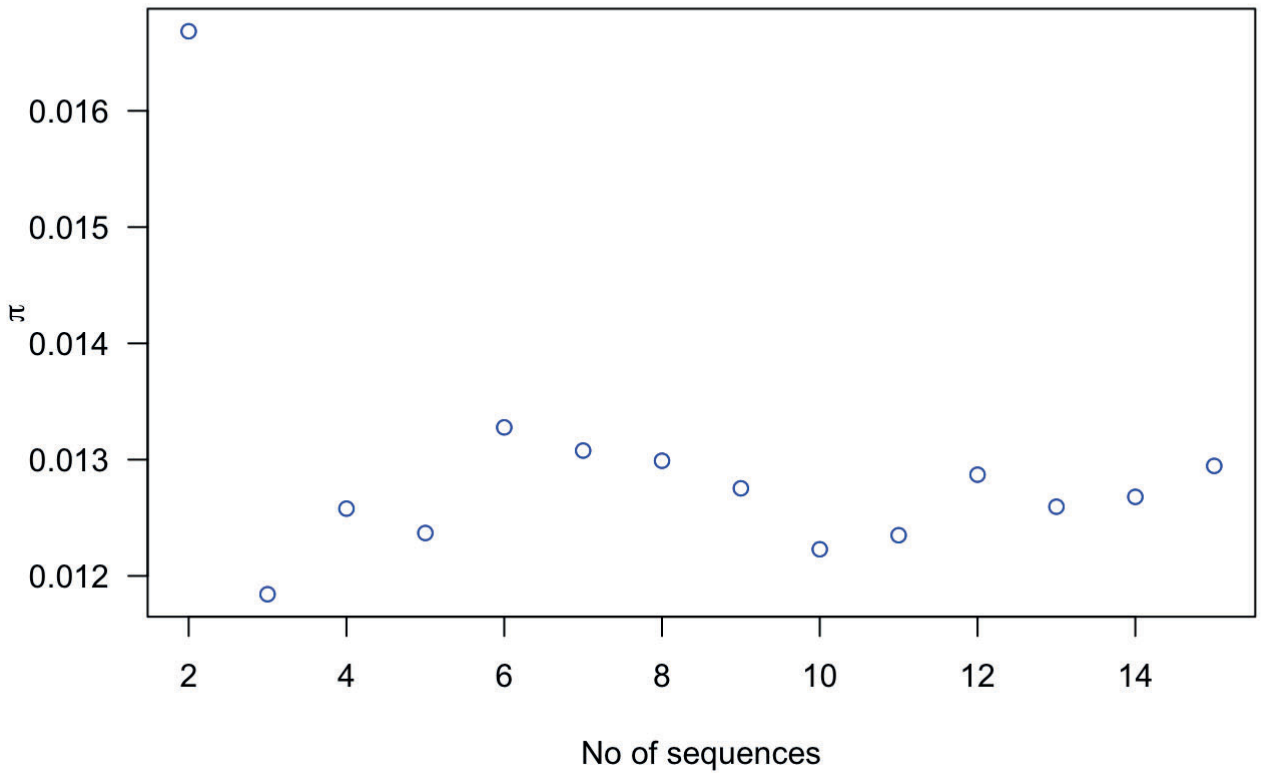
It should be immediately obvious from this figure that the number of segregating sites is biased by the number of sequences we include in the data. It increases our probability of observing a polymorphism and also since all polymorphisms are given equal weighting in the calculation of the number of segregating sites, any polymorphism will increase the value by 1.

So what do we see if we repeat the same code, but this time for nucleotide diversity or π?

```
# use sapply to loop
nd <- sapply(2:15, function(z){
  nuc.div(woodmouse[1:z, ])
})
```

Again, our `sapply` function is the same as above, except this time we use the `nuc.div` function within the curly brackets instead. We can also plot the relationship here:

```
# plot figure
plot(2:15, nd, col = "blue", xlab = "No of sequences", ylab = expression(pi), las = 1)
```



There is no obvious relationship here - nucleotide diversity is an estimate of the average difference among sequences so it only becomes more precise with increased numbers of sequences - it is not so easily biased. Unlike segregating sites, polymorphisms are also not equally weighted; rare polymorphisms only contribute slightly to nucleotide diversity as most sequences will be similar at this position, whereas more common polymorphisms mean most sequences will differ at a position.

Inferring evolutionary processes using Tajima's *D*

The number of segregating sites and π are essentially estimates of the population mutation rate θ - i.e. $4N_e\mu$. Under ideal conditions - i.e. neutrality, our two estimates of θ should be equivalent to one another. Differences between these estimates suggest either the action of selection or some kind of demographic change.

Tajima's *D* is a statistical test that allows us to actually investigate this. We can calculate it very easily in R using the `tajima.test` function in `pegas`.

```
# calculate Tajima's D for the woodmouse data
tajima.test(woodmouse)
```

This produces a list with three elements. The first is an estimate of Tajima's *D* and the other two are p-values assessing the significance of the statistic. They are both similar here and also both show that there is no significant deviation from zero.

Calculating statistics at the whole genome level

So far we have calculated descriptive statistics from two sets of sequence data. This is obviously useful to demonstrate how these statistics work and reinforcing our understanding of them. However, in modern evolutionary genetics, we are often working with much larger datasets than what we have dealt with so far. As

sequencing technology becomes cheaper, faster and more accessible, we are regularly working with genome-scale data. This has implications for how we handle data and also how we interpret it and use it to test hypotheses or make some form of evolutionary inference.

In this section of the tutorial, we will use a package called `PopGenome` to read variants called from **whole genome resequencing data** into R and then we will calculate nucleotide diversity within and among populations and also along a single chromosome. **A word of caution** `PopGenome` is a very useful package, but it is not always the most user friendly, so we have done our best to simplify matters as much as possible here.

Reading in variant data

The data we are going to be working with for this section is a set of **SNP** calls from whole-genome resequencing of *Passer* sparrows. It was originally used in a study on the evolution of human commensalism in the house sparrow by Ravinet *et al.* (2018) (http://rspb.royalsocietypublishing.org/content/285/1884/20181246). In the dataset there are SNPs from four sparrow species - the house sparrow, the Italian sparrow, the Spanish sparrow, the tree sparrow and also data from a house sparrow sub-species known as the Bactrianus sparrow.

The data comes in two parts. We'll deal with the actual SNP data first. This is stored in a file called a VCF, which stands for variant call format (https://samtools.github.io/hts-specs/VCFv4.2.pdf). This is a standard file format for storing SNP data and is produced by a lot of SNP calling pipelines. It is beyond the scope of todays tutorial to go into much detail about VCF files - all you really need to do today is understand that it is a format we need to read in to R in order to work with the SNP data.

One thing you should know about VCF files though is that they can quickly become very very big. The one we are using today is a much smaller, randomly sampled version of the true dataset **from a single chromosome only**, however it is still quite a large file size. Because it is large, the file is compressed and there are some preprocessing steps you will need to do before you can open in it in R.

- First, download the VCF (https://evolutionarygenetics.github.io/sparrow_chr8_downsample.vcf.gz)
- Next, make a directory in your working directory (use `getwd` if you don't know where that is) and call it `sparrow_snps`
- Move the downloaded VCF into this new directory and then uncompress it. If you do not have an program for this, you can either use the Unarchiver (https://theunarchiver.com/) (Mac OS X) or 7zip (https://www.7-zip.org/) (Windows).
- Make sure **only** the uncompressed file is present in the directory.

With these steps carried out, you can then easily read this data in like so:

```
sparrows <- readData("./sparrow_snps/", format = "VCF", include.unknown = TRUE, FAST = TRUE)
```

```
## |           :           |           :           | 100 %
## |=================================================
```

N.B This command does not always work on Windows machines. If you are using Windows, you might need to change the path to `"./sparrow_snps"` - i.e. so it is missing the last backslash.

First, we point the `readData` function at the directory where our decompressed VCF is. The following arguments are: * `format = VCF` function to tell it is a VCF (since `PopGenome` can read in other file types). *`include.unknown = TRUE` this is just an argument to `readData` to let it know that any SNPs where there are unknown calls should be included. We need this as not all individuals have data for each site.* `FAST = TRUE` to tell the function we want it to use fast computation.

So we just read a variant data from an entire chromsome into R. Before we move on, let's take a look at what we have actually created with our `sparrows` object. If you call the `sparrows` object, you will not really see anything that informative, since it is a very complex data structure. However with the `get.sum.data` function,

we can learn a bit more about it.

```
get.sum.data(sparrows)
```

This just gives us a quick summary of what we read in. **A word of warning here** - the `n.sites` value here is *NOT* the number of SNPs we read in - it is simply the position of the furthest SNP in our dataset on chromosome 8. We can essentially ignore this value. To get the number of variants or SNPs in our VCF, we need to add together `n.biallelic.sites` (i.e. SNP sites with only two alleles) and `n.polyallelic.sites` (i.e. SNP sites with three or four alleles).

You can do this by hand or using R code like so:

```
sparrows@n.biallelic.sites + sparrows@n.polyallelic.sites
```

This also demonstrates one other important point about `PopGenome`, the `sparrows` object is a complicated structure called a `GENOME` object (use `class(sparrows)` to see this). Unlike other R structures we have seen so far, we need to use `@` to access some parts of it. This is a little confusing, but you can think of it in acting in a similar way to the `$` operator we used to access columns of a `data.frame`.

Handling variant data

One thing we haven't done yet is set the populations in our dataset. We have data for 129 individuals, each from different species. Does reading in the variant data tell us anything about the populations?

```
# check for population data
sparrows@populations
```

If you look at this, you will only see a blank list. So we need to supply our population data to the `sparrows` object. To make naming our populations simple, we will read in some external data. This is easy, as it is just a `data.frame` which you can download here (https://evolutionarygenetics.github.io/sparrow_pops.txt)

```
sparrow_info <- read_delim("./sparrow_pops.txt", delim = "\t")
```

If you take a quick look at `sparrow_info`, you will see that it has two columns, one for an individual name and one for the population it belongs to. You can easily count the number of species with a `tidyverse` approach using `group_by` and `tally` or use `table` on the `pop` column if you want to use a base R approach.

What we are going to do next is create a list of the individuals in each of the populations. This is actually very easy to do using the split command, like so:

```
# now get the data for the popultions
populations <- split(sparrow_info$ind, sparrow_info$pop)
```

All `split` does is split the first argument (individual names in this case), by a factor variable in the second argument (population or species here). So, our use of `split` will create a list of 5 populations with the names of individuals in each one stored as character vectors in the list. We can then use this to set the populations in our sparrow dataset:

```
# now set
sparrows <- set.populations(sparrows, populations, diploid = T)
```

```
## |            :            |            :            | 100 %
## |=================================================
```

At first glance, there is no obvious clue this actually worked, but we can check it by accessing the `populations` data in the `sparrows` object again.

```
# check that it worked
sparrows@populations
```

Calculating nucleotide diversity statistics

So, let's recap so far. We have over 90,000 SNPs from chromsome 8 of 129 sparrows from five different species. With the `PopGenome` package, we can now very quickly ad easily calculate nucleotide diversity for every single one of these SNPs. We will do this like so:

```
# calculate nucleotide diversity
sparrows <- diversity.stats(sparrows, pi = TRUE)
```

```
## |           :           |           :           | 100 %
## |=================================================
```

A nice, simple function that just requires us to specify that we want to calculate π using `pi = TRUE`. Let's try and get at the data. Unfortunately, this is where things become a little more tricky since the `GENOME` object data structure is quite complicated. Use the following function to extract the data.

```
sparrow_nuc_div <- t(sparrows@region.stats@nuc.diversity.within[[1]])
```

Basically, we extracted the nucleotide data for each of the species from `GENOME` object and used `t` to transpose the data, so that each row is a SNP and each column is a species. But if you look at the `sparrow_nuc_div` matrix (tip: use the `head` function), you will see it is still not ideal for interpretation - the populations have no names and we don't have a proper data structure - i.e. the SNP positions are stored as row names.

We can use some code to get the data into a more suitable format for visualisation and interpretation.

```
# get the SNP positions from the rownames - note we need to make them numeric here
position <- as.numeric(rownames(sparrow_nuc_div))
# rename the matrix columns after the species
colnames(sparrow_nuc_div) <- c("bactrianus", "house", "italian", "spanish", "tree")
# combine into a data.frame and remove the row.names
sparrow_nd <- data.frame(position, sparrow_nuc_div, row.names = NULL)
# make a tibble for easy viewing
sparrow_nd <- as.tibble(sparrow_nd)
```

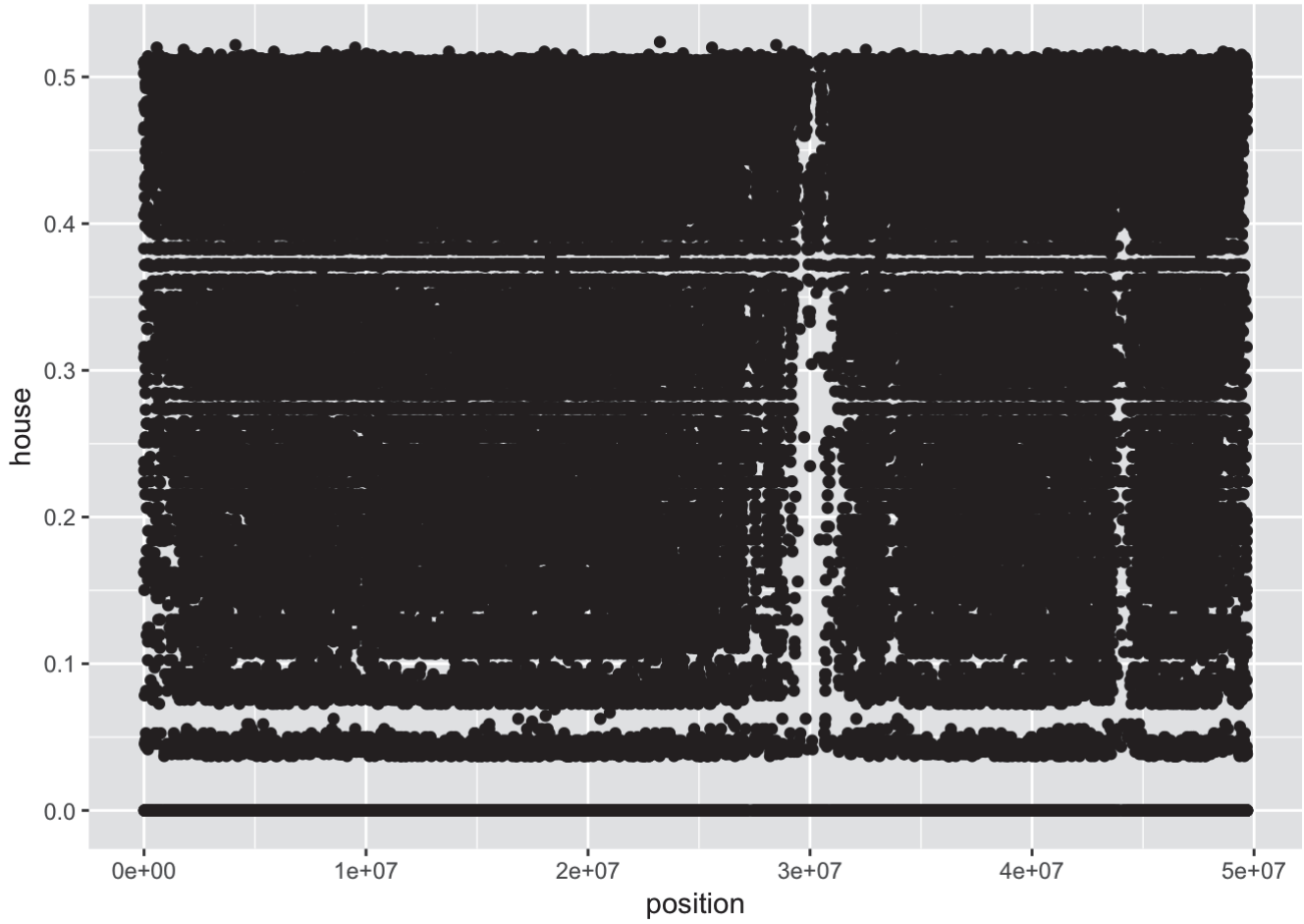
```
## Warning: `as.tibble()` is deprecated, use `as_tibble()` (but mind the new semantics).
## This warning is displayed once per session.
```

One thing to note here - our `sparrow_nd` data.frame is 91,312 rows - this is the same as the number of biallelic SNPs, which tells us that `PopGenome` only calculates nucleotide diversity on these positions, not those with more than two alleles.

Visualising nucleotide diversity along the chromosome

Lastly, we will actually visualise the nucleotide diversity for the house sparrow along the whole of chromosome 8. Since we set up a data.frame in the last section, this is very easy to do with `ggplot2`.

```
ggplot(sparrow_nd, aes(position, house)) + geom_point()
```



Hmm - this is not the most visually appealing figure and it is also not particularly informative. Why is that? Well one of the issues here is that we have calculated nucleotide diversity for each biallelic SNP position so there is a lot of noise and the signal from the data is not clear.

This is often the case when working with high-density genome data. One solution is to use a sliding window analyis in order to try and capture the average variation across a chromosome. We will learn how to do this in the next section.

Performing a sliding window analysis

Luckily, it is very easy indeed to perform a sliding window analysis using `PopGenome` . We just need to use the function `sliding.window.transform` to create a new `GENOME` object called `sparrows_sw`

```
# make a sliding window dataset
sparrows_sw <- sliding.window.transform(sparrows, width = 100000, jump = 25000, type = 2)
```

```
## |          :           |           :           | 100 %
## |==================================================| ;-)
```

The first argument to this function is our `sparrows` object. Next we specify the size of each window in the genome - here we will use `100000` basepairs or 100 Kb. We then set the step or `jump` for each window - in this case `25000` or 25 Kb. What this means is that we calculate a statistic in a 100 Kb window and then move along 25 Kb and calculate the statistic again and so on and so on until we have moved along the entire chromosome.

Dividing our chromsome up into windows, means that instead of 90,000 plus estimates of nucleotide diversity, we will get… as many as we have windows! And the easiest way to check how many windows we have is like so:

```
# check number of genome windows
length(sparrows_sw@region.names)
```

```
## [1] 1984
```

We can then recalculate nucleotide diversity in exactly the same way on our windowed data as we did before.

```
# calculate nucleotide diversity
sparrows_sw <- diversity.stats(sparrows_sw, pi = TRUE)
```

```
## |           :           |           :           | 100 %
## |=================================================| ;-)
```

It is also much easier to extract this data.

```
sparrow_nuc_div_sw <- sparrows_sw@nuc.diversity.within
```

However, a key point here is that our nucleotide diversity estimates are not scaled to the size of our genome windows. Use `head(sparrow_nuc_div_sw)` to see that the values we have calculated are very high. Remember, we need to standardise nucleotide diversity to the length of the sequence we estimated it from. In this case, we used 100 Kb windows, so we need to divide these values by 100000. We do this like so:

```
sparrow_nuc_div_sw <- sparrow_nuc_div_sw/100000
```

Finally, we can get it into a format ready for plotting. However, since we don't SNP positions for this data, since it is for sliding genome windows. Remember that we set our windows to be 100 Kb, but with a step of 25 Kb. So the midpoint for our first window is at 50 kb, the second window at 75 kb and so on. We can generate midpoint positions for our windows with `seq` .

```
# generate window midpoints
position <- seq(from = 1, to = 49575001, by = 25000) + 50000
```

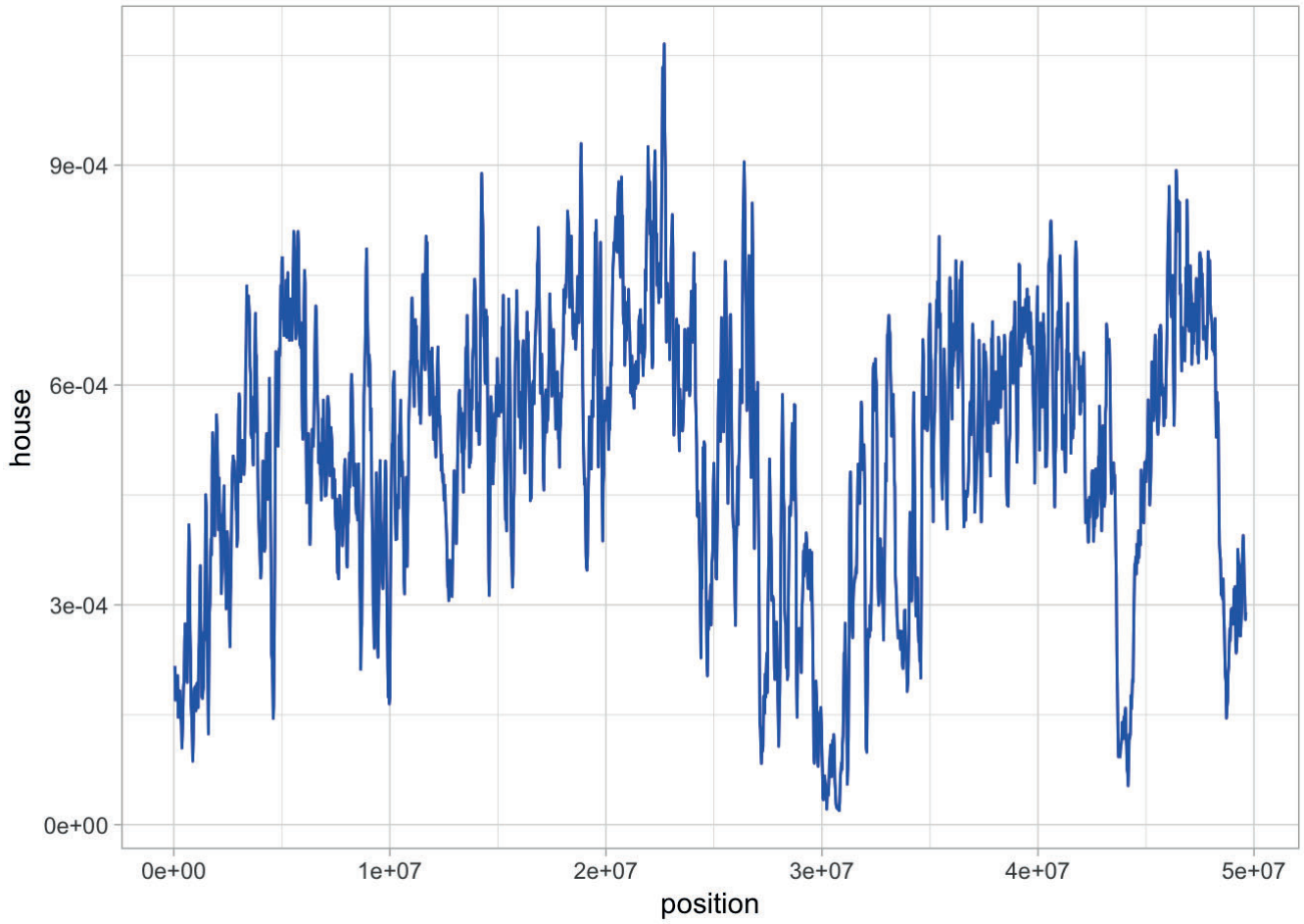
Here we set the first and last positions and then specify we want to generate values at intervals of 25000. Finally we add 50000 in order to make the midpoint of each window. Use `head(positions)` to examine the start of the vector and see.

Next all we need to do to create the final dataset for plotting is the same as we did for our per-snp analysis above.

```
# rename the matrix columns after the species
colnames(sparrow_nuc_div_sw) <- c("bactrianus", "house", "italian", "spanish", "tree")
# combine into a data.frame and remove the row.names
sparrow_nd_sw <- data.frame(position, sparrow_nuc_div_sw, row.names = NULL)
# make a tibble for easy viewing
sparrow_nd_sw <- as.tibble(sparrow_nd_sw)
```

Last but not least, we can actually plot our data, using `ggplot2` .

```
ggplot(sparrow_nd_sw, aes(position, house)) + geom_line(colour = "blue") + theme_light()
```



This is much more informative than our per SNP figure from before. What is more, we can see clearly there are several regions on this chromosome where there is a signficant reduction in nucleotide diversity, particularly around 30 Mb. We cannot say exactly what might be causing this without inferring other statistics or examining other data, but one possibility is that this is a region of reduced recombination where selection has led to a reduction in diversity. If it is shared with other sparrow species, it might suggest some kind of genome structure such as a centromere - where recombination rates are usually lower.

The point is that sliding window information like this can be extremely informative for evolutionary analysis. Although we only got a quick introduction to genomic data in today's tutorial, we will return to this sort of dataset again in the next session and explore more fully how we can use it to infer processes that might shape the distribution of these sorts of statistics in the genome.

Study questions

For study questions on this tutorial, download the `Chapter7_R_questions.R` from Canvas or find it here (https://evolutionarygenetics.github.io/Chapter7_R_questions.R).

Going further

- A simple tutorial on using PopGenome for inference in R (http://tonig-evo.github.io/workshop-popgenome/)
- Some more information on performing whole-genome analyses in PopGenome (https://cran.r-project.org/web/packages/PopGenome/vignettes/Whole_genome_analyses_using_VCF_files.pdf)
- More examples with PopGenome and also other R packages for sequence data (https://wurmlab.github.io/genomicscourse/2016-SIB/practicals/population_genetics/popgen)