# 8. Türleşme genomiği ve genetiği

## Uygulama

- 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 %
## |=================================================
```

## Examining the variant data

Remember, you can look the data we have read in using the following command:

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

In this case, you can see that from the `n.sites` that the final site is at position 49,693,117. The actual chromosome is 49,693,984 long - so this confirms variants span the entire chromosome. Note that `n.sites` is a bit counter-intuitive here, it would only make sense as the number of sites if we had called nucleotides at **every single position in the genome** - but since this is a variant call format, only containing polymorphic positions then obviously this is not the case. Furthermore, the data has actually been subset in order to make it more manageable for our purposes today.

Nonetheless, it is still substantial, from the `n.biallelic.sites` we can see there are 91,312 bilallelic SNPs and from `n.polyallelic.sites` , there are 1092 positions with more than two alleles. So in total we have:

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

A total of 92,404 SNPs - a big dataset which requires some specific approaches to handling the data.

## Setting populations/species identifiers

Since we're going to be examining differences between species, we need to identify them in our dataset. In other words, we need to set some kind of information on the populations they belong to. We can use the following code to see if any population information is actually stored in the VCF.

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

Since at this point, the command only returns a blank list, we need to do something about it. As with the last session, we can read in some information to set the population names. If you ran through the last session, you will already have this data but if not, you can download it again here (https://evolutionarygenetics.github.io/sparrow_pops.txt) and then read it into R as a tab-delimited file.

```
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. Next, we will 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 populations
populations <- split(sparrow_info$ind, sparrow_info$pop)
```

Remember that 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
```

## Setting up sliding windows

So far, this will start to seem quite familiar! We learned in the last session that per-SNP estimates of statistics such as $\pi$ can often be extremely noisy when you are calculating them on very large numbers of markers. As well as this, there are issues with the fact that SNP positions in close proximity are not always independent due to recombination - this is a theme we will return too shortly. So for this reason, it is often better to use a **sliding-window** approach - i.e. split the genome into windows of a particular size and then calculate the mean for a statistic within that window.

We know already that chromosome 8 is 49,693,984 bp long, so we can get an idea of how many sliding windows we would generate by using some R code. We'll set our sliding window to be 100,000 bp wide - or 100 Kb. We will also set a **step** or **jump** for our window of 25,000 bp - or 25Kb.

```
# set chromosome size
chr8 <- 49693984

# set window size and window jump
window_size <- 100000
window_jump <- 25000

# use seq to find the start points of each window
window_start <- seq(from = 1, to = chr8, by = window_jump)
# add the size of the window to each start point
window_stop <- window_start + window_size
```

If you ran this code successfully, you can see that we have now generated two vectors `window_start` and `window_stop`. To create the windows, we used `seq` to go from 1 - i.e. the first basepair - up to the full chromosome 8 length in 25 Kb jumps or steps. The we just add the window size to each of these start points to get the stop points of the windows. So the first window runs from 1 to 100 Kb, the second window from 25 Kb to 125 Kb and so on.

*However*, there is an issue here. Some of the windows stop *after* the end of the chromosome, so we need to remove these. You can use the following code and logical operations to see that all windows start before the end of the chromosome but that because of how we generated the stop windows, this is not the case for the

stop positions.

```
# no windows start before the end of chromosome 8
sum(window_start > chr8)
# but some window stop positions do occur past the final point
sum(window_stop > chr8)
```

In fact, there are 4 windows that are beyond the end of the chromosome. To remove them, we can use the same logical operations as above, just this time within square brackets to drop those positions.

```
# remove windows from the start and stop vectors
window_start <- window_start[which(window_stop < chr8)]
window_stop <- window_stop[which(window_stop < chr8)]
```

Note that here we wrapped our logical operation `window_stop < chr8` in a `which` function - this basically tells R to return the position in the vector where this condition is `TRUE`. Also note that we have to remove the **start** windows that meet this condition too. Why? Well because we are using a sliding window and our window size is 100 kb, the window starting at 49,675,001 will come close to the end of the chromosome.

Actually, this highlights an important point, our final window actually falls **short** of the end of the chromosome. You can check this like so:

```
chr8 - window_stop[length(window_stop)]
```

Note that here, `length(window_stop)` in the square brackets just means we are evaluating the final value in the `window_stop` vector. This is something to be aware of, since our final window falls short of the end of the chromosome, we may not be including all our variants. This is not necessarily wrong, but it is important to note.

Anyway, although a little long-winded, this sliding window section is important as it will be useful for plotting later. For now, we will save our sliding window start/stop positions as a `data.frame`. We'll also calculate the midpoint for each window.

```
# save as a data.frame
windows <- data.frame(start = window_start, stop = window_stop,
                      mid = window_start + (window_stop-window_start)/2)
```

Finally, we can set our sliding windows for our sparrows dataset using the `PopGenome` function, `sliding.window.transform`

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

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

# Calculating sliding window estimates of nucleotide diversity and differentiation

Now that we have set up the data, the population information and the sliding windows, it is quite straightforward for us to calculate some statistics we are interested in. In this case, we are going to calculate nucleotide diversity (i.e. $\pi$) and $F_{ST}$. We will also generate a third statistic, *d*_XY_, which is the absolute nucleotide divergence between two populations.

First we will calculate $\pi$. Handily, the following command also sets up what we need for *d*_XY_.

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

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

Next we will calculate $F_{ST}$, which again is very straight forward with a single command.

```
# calculate diversity statistics
sparrows_sw <- F_ST.stats(sparrows_sw, mode = "nucleotide")
```

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

Note that here we use `mode = "nucleotide"` to specify we want it to be calculated sliding averages of nucleotides, rather than using haplotype data, which is the alternative. And that's it for calculating the statistics! As you will see in the next section, extracting them from the `sparrows_sw` object is actually more difficult than generating them…

## Extracting statistics for visualisation

Since we ran our analysis on a sliding-window basis, we should have estimates of $\pi$, $F_{ST}$ and *d*_XY_ for each window. What we want to do now is extract all our statistics and place them in a single `data.frame` for easier downstream visualisation - this will let us identify how these statistics are interrelated.

First of all, we will get the nucleotide diversity data.

```
# extract nucleotide diversity and correct for window size
nd <- sparrows_sw@nuc.diversity.within/100000
```

This is straightforward, but remember also that our estimates need to be corrected for window size - so we divide them by 100 Kb here. We should also add the population names to each of them, since they are not already set.

```
# make population name vector
pops <- c("bactrianus", "house", "italian", "spanish", "tree")
# set population names
colnames(nd) <- paste0(pops, "_pi")
```

We will learn about `paste0` in more detail shortly. For now, we can extract $F_{ST}$

```
# extract fst values
fst <- t(sparrows_sw@nuc.F_ST.pairwise)
```

Note that here, we need to use `t()` to transpose the *F*_ST matrix so that each column is a pairwise comparison and each row is an estimate for a genome window. Since *F*_ST is pairwise, the column names are also quite different and will also be the same for *d*_XY_, which is also a pairwise measure. For this reason, we will deal with these column names together shortly.

So now we are ready to extract our final statistic, *d*_XY_. We can do this in a similar way to how we handled the $F_{ST}$ data.

```
# extract dxy - pairwise absolute nucleotide diversity
dxy <- get.diversity(sparrows_sw, between = T)[[2]]/100000
```

As with nucleotide diversity, we also corrected *d*_XY_ for the window size.

Now we sort out the column names for our $F_{ST}$ and *d*_XY_ data. This is where our R skills come in use! We will need to use some R-based string manipulation. The column names are identical for both datasets, so we will take the first one and use the `sub` function to replace the population names.

```
# get column names
x <- colnames(fst)
# replace all occurrences of pop1 with house
x <- sub("pop1", "bactrianus", x)
# does the same thing as above but by indexing the pops vector
x <- sub("pop1", pops[1], x)
# look at x to confirm the replacement has occurred
x
```

All `sub` does is replace the first argument, `pop1` in this case, with the second, `bactrianus` here. In other words, it is a kind of find and replace for character strings. The second example in the code above does exactly the same thing, but instead calls the relevant population from the `pops` vector we created earlier.

Now we know what `sub` does, we can replace all the different populations at once. Like so:

```
# get column names
x <- colnames(fst)
# does the same thing as above but by indexing the pops vector
x <- sub("pop1", pops[1], x)
x <- sub("pop2", pops[2], x)
x <- sub("pop3", pops[3], x)
x <- sub("pop4", pops[4], x)
x <- sub("pop5", pops[5], x)
# replace forward slash
x <- sub("/", "_", x)
# look at x to confirm the replacement has occurred
x
```

```
##  [1] "bactrianus_house"   "bactrianus_italian" "bactrianus_spanish"
##  [4] "bactrianus_tree"    "house_italian"      "house_spanish"
##  [7] "house_tree"         "italian_spanish"    "italian_tree"
## [10] "spanish_tree"
```

Now all that we need to do is make clear these names are for either $F_{ST}$ or *d*_XY_. The best way to do this is to append a suffix to our vector of pairwise comparison names. We can do this using `paste0` :

```
paste0(x, "_fst")
paste0(x, "_dxy")
```

So this function allows us to join strings together in a character vector. Very useful. Next we will actually change the column names of our two data matrices, before we put everything together in our final dataset.

```
colnames(fst) <- paste0(x, "_fst")
colnames(dxy) <- paste0(x, "_dxy")
```

Ok so now that our $\pi$, $F_{ST}$ and *d*_XY_ datasets are ready, we can combine them all together with our windows information from earlier into a big dataset.

```
sparrow_data <- as.tibble(data.frame(windows, nd, fst, dxy))
```

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

# Visualising the data - distributions

For the purposes of this session, we will focus mainly on the difference between house and spanish sparrows. However, since we now have all our data in a tidy `data.frame`, it is very easy to calculate things like the mean values of our statistics among all the different species. For example, let's say we want to look at mean nucleotide diversity, we can do that like so:

```
# select nucleotide diversity data and calculate means
sparrow_data %>% select(contains("pi")) %>% summarise_all(mean)
```

A lot of this will be familiar from before but to clarify, we used `select` and `contains` to select columns from our main dataset that contain `pi` - i.e. nucleotide diversity columns. We then used `summarise_all` and `mean` to calculate the mean value for all of the columns we selected. From the output above, we can see that the house and the Italian sparrow have the highest levels of nucleotide diversity.

We could also quite easily plot if we wanted to. However, to do this, we need to use `gather` on the data.

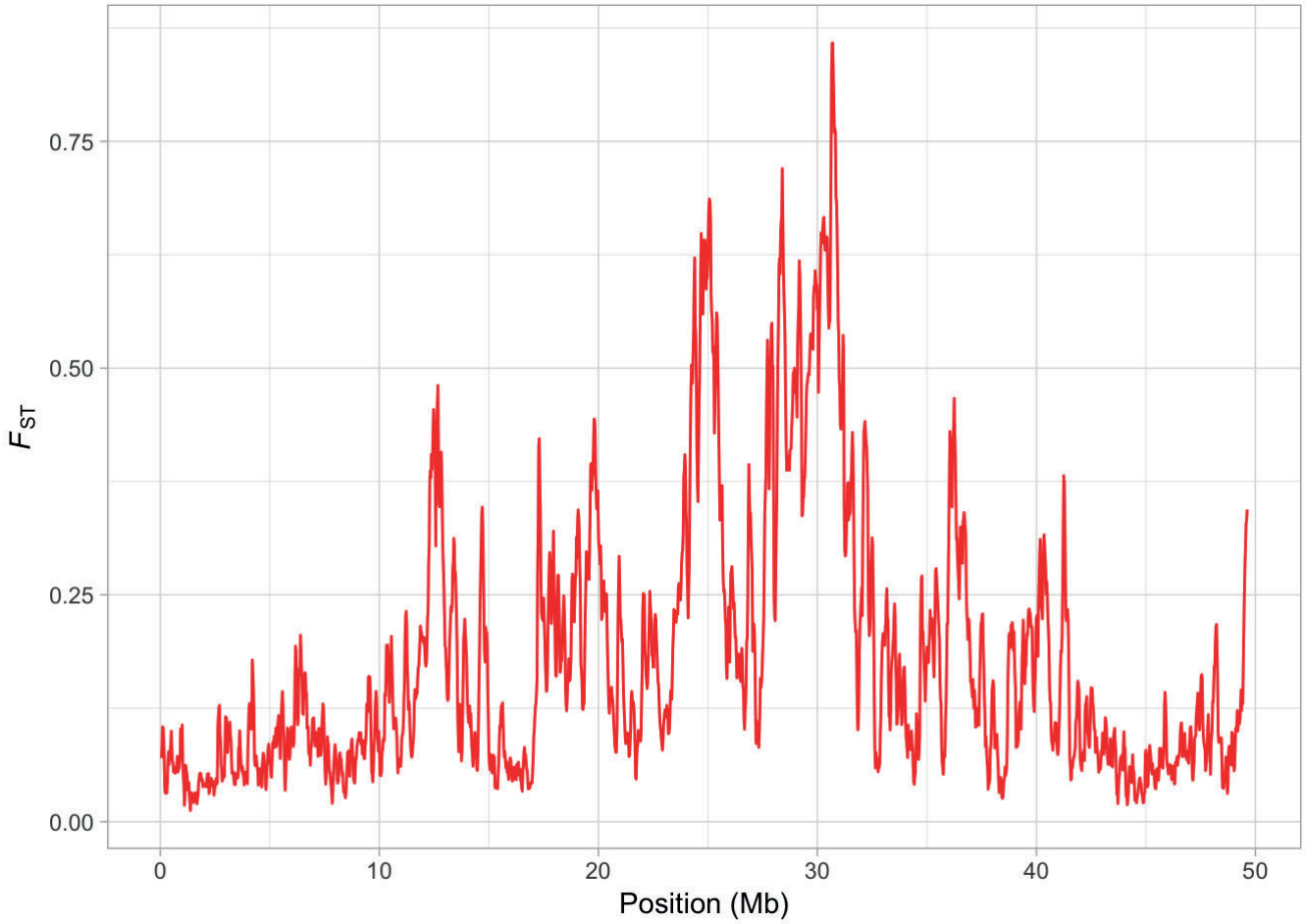
```
# gather the data
pi_g <- sparrow_data %>% select(contains("pi")) %>% gather(key = "species", value = "pi")
# make a boxplot
a <- ggplot(pi_g, aes(species, pi)) + geom_boxplot() + theme_light() + xlab(NULL)
```

This makes it much clearer how nucleotide diversity differs among the species.

# Visualising patterns along the chromosome

Let's have a look at how $F_{ST}$ between house and spanish sparrows varies along chromosome 8. We can do this very simply with `ggplot`.

```
a <- ggplot(sparrow_data, aes(mid/10^6, house_spanish_fst)) + geom_line(colour = "red")
a <- a + xlab("Position (Mb)") + ylab(expression(italic(F)[ST]))
a + theme_light()
```




From this plot, it is very clear there is a huge peak in $F_{ST}$ around 30 Mb. Actually, there are several large peaks on this genome but is this one a potential region that might harbour a speciation gene? Well you might recall from the previous session that there is a drop in nucleotide diversity in this region…

How can we investiage this? The easiest thing to do is to plot $\pi$, $F_{ST}$ and $d$_XY_ to examine how they co-vary along the genome. This requires a bit of data manipulation, but is relatively straightforward. We will break it down into steps.

First, let's get the data we are interested in:

```
# select data of interest
hs <- sparrow_data %>% select(mid, house_pi, spanish_pi, house_spanish_fst, house_spanish_dx
y)
```

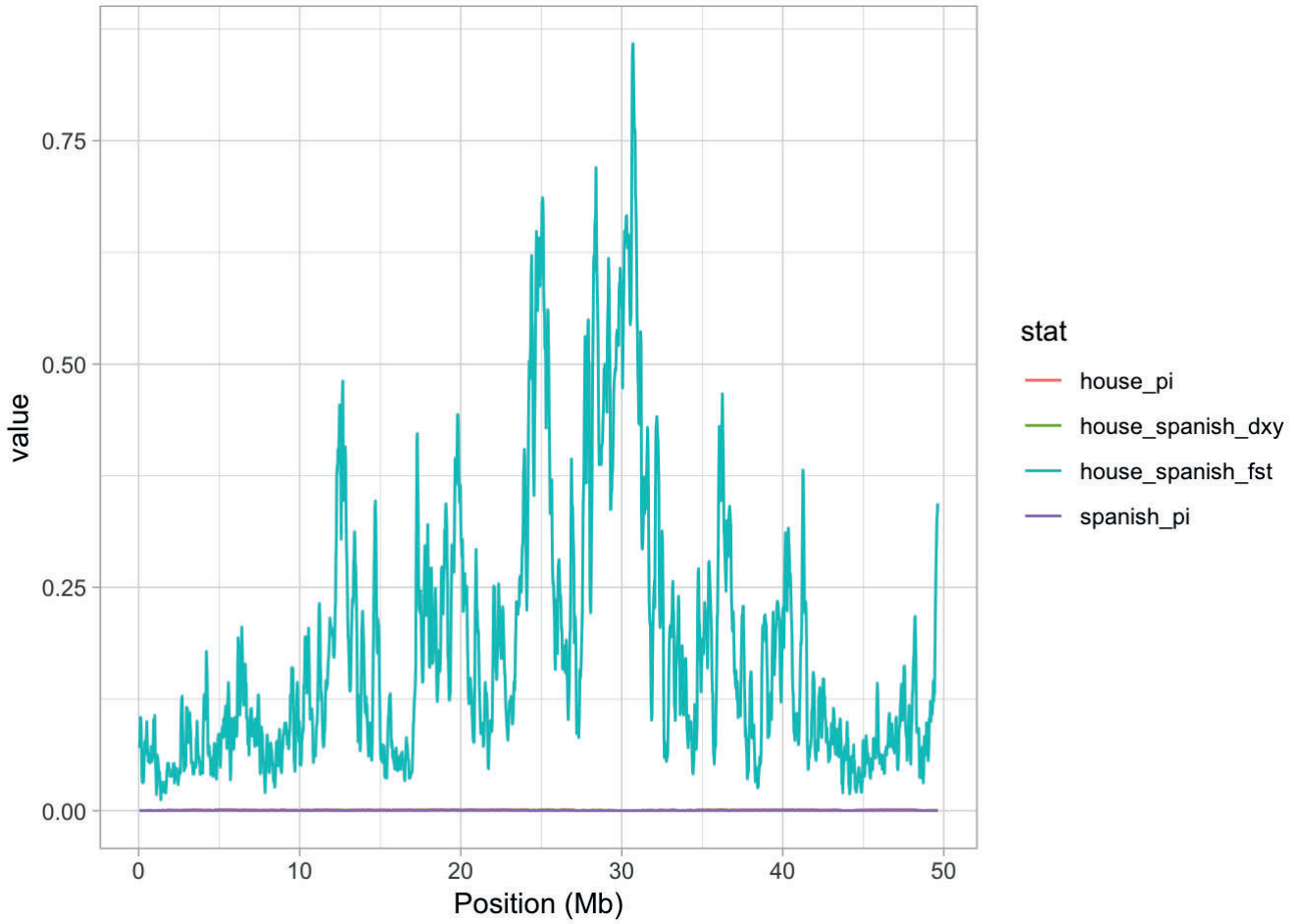
To keep things simple, we've thrown everything out we don't need. Next, we need to use `gather` in order to rearrange our `data.frame` so that we can plot it properly.

```
# use gather to rearrange everything
hs_g <- gather(hs, -mid, key = "stat", value = "value")
```

`gather` is a tricky function to explain, although we have encountered it back in Chapter 2 (https://evolutionarygenetics.github.io/Chapter2.html). All it does is collapse everything so we can plot them efficiently. We use `-mid` to tell the function we want to leave this out of the gathering and use `key = stat` to make it clear we are arranging our data by the statistics we have calculated, `value = value` is just a name for the values of each of our statistics.

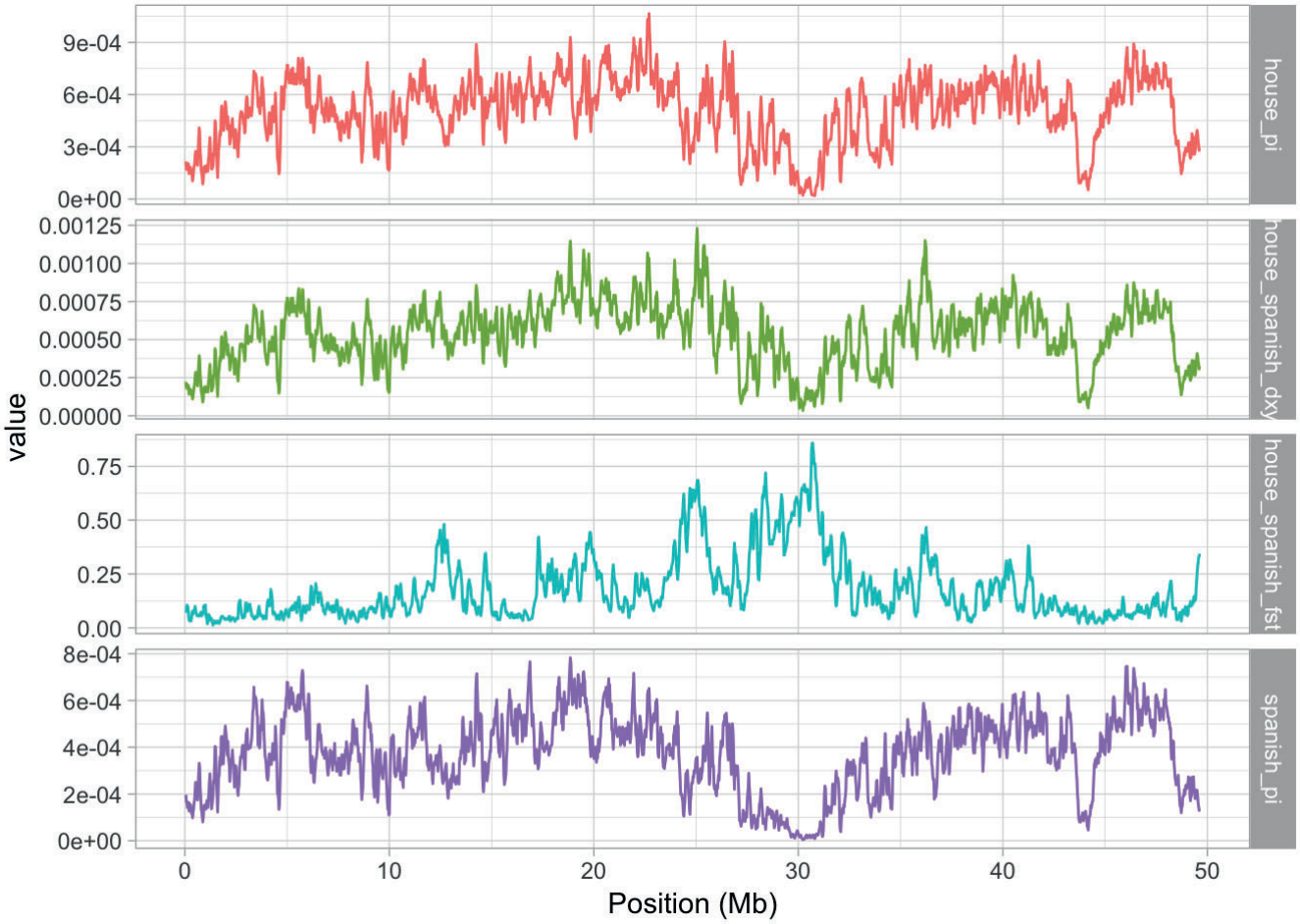
Now we can easily plot everything together like so:

```
a <- ggplot(hs_g, aes(mid/10^6, value, colour = stat)) + geom_line()
a <- a + xlab("Position (Mb)")
a + theme_light()
```




OK so it should be immediately obvious that this plot is really unhelpful. We see the $F_{ST}$ data again, but since that is on such a different scale to estimates of $\pi$ and $d\_XY\_$, we can't see anything! Instead, it would make a lot more sense to split our plot into facets - i.e. a plot panel for each statistic. This is simple with the `ggplot` function `facet_grid` . We will construct our plot first and then breakdown what `facet_grid` actually does.

```
# construct a plot with facets
a <- ggplot(hs_g, aes(mid/10^6, value, colour = stat)) + geom_line()
a <- a + facet_grid(stat~., scales = "free_y")
a <- a + xlab("Position (Mb)")
a + theme_light() + theme(legend.position = "none")
```



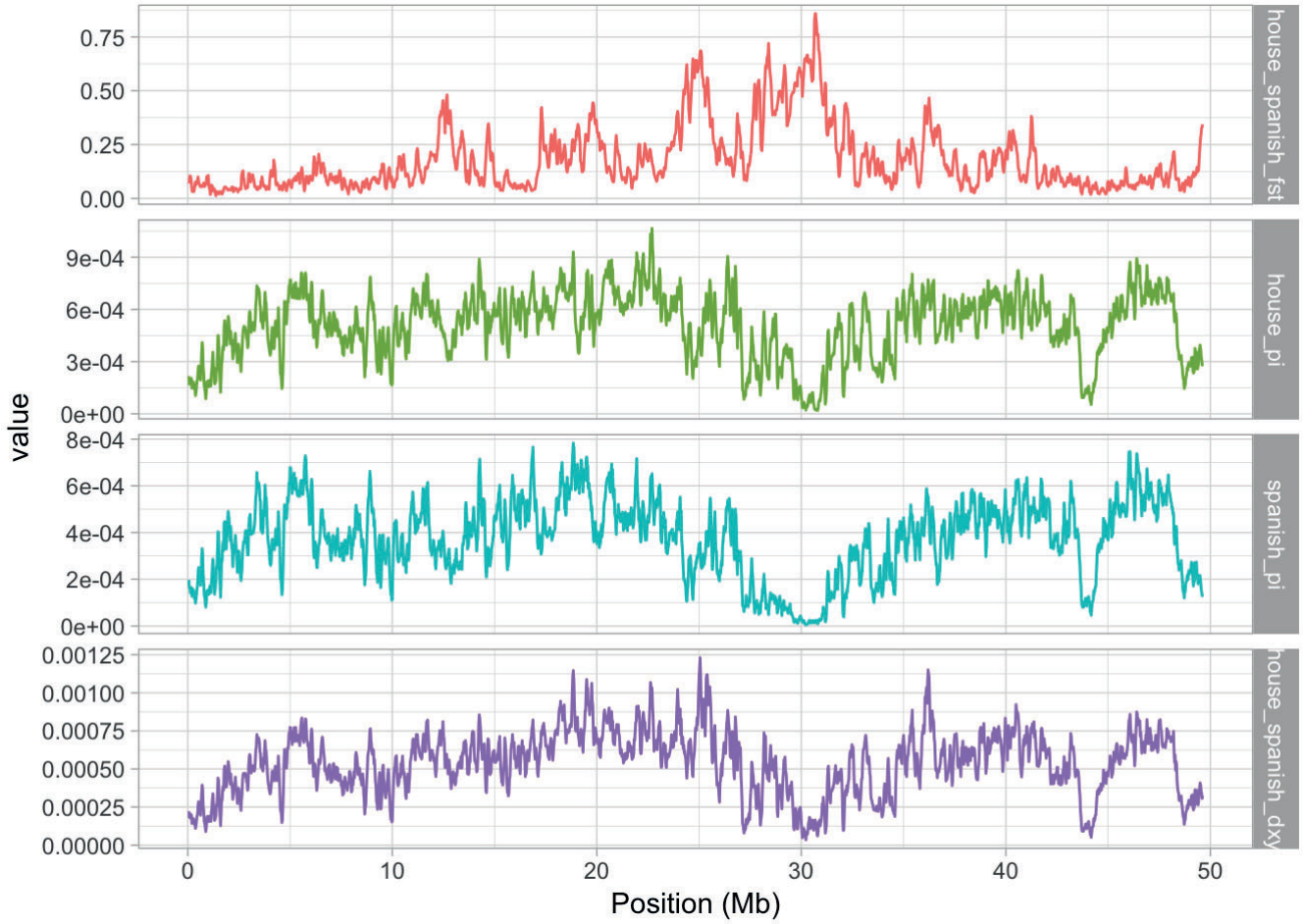

The `facet_grid` function allows us to split our data across panels for quick and easy visualisation. In this case, we split our data by the `stat` variable - we used `stat~.` to specify we want this done by rows (compare with `.~stat` for the column equivalent). We also specified that we wanted the scales on our y-axes to vary with `scales = free_y`.

However, before we examine our plot in detail, it would also be easier if we rearranged everything so $F_{ST}$ came at the top, $\pi$ beneath it and then finally, *d*_XY_. How can we do that? Well we need to reorder the `stat` factor in our `hs_g` dataset.

```
# first make a factor
x <- factor(hs_g$stat)
# then reorder the levels
x <- factor(x, levels(x)[c(3, 1, 4, 2)])
# add to data.frame
hs_g$stat <- x
```

This looks a little complicated, but in the square brackets above we simply rearranged what order our facets are displayed. We can replot our figure to demonstrate this:

```
# construct a plot with facets
a <- ggplot(hs_g, aes(mid/10^6, value, colour = stat)) + geom_line()
a <- a + facet_grid(stat~., scales = "free_y")
a <- a + xlab("Position (Mb)")
a + theme_light() + theme(legend.position = "none")
```




Examining the plot we created, it is pretty clear that the large peak in $F_{ST}$ on our chromosome is matched by two regions of low nucleotide diversity in the house and Spanish sparrow, *d*_XY_ is also very low in the same region.

The signal here is quite clear - what could explain it? Low recombination regions in the genome are one potential explanation. The reason for this is that in a low recombination region, **background selection** and also **selective sweeps** can remove variation at polymorphic positions that are closely linked to the target of selection. Selection of this kind in either the house or the Spanish lineages AFTER they have split into different species will reduce $\pi$ in these regions and since $F_{ST}$ is a relative measure of differentiation, it will potentially be inflated.

This is an important issue as it means that we cannot reliably use $F_{ST}$ to identify genes involved in reproductive isolation from a genome scan. By comparing $F_{ST}$ to *d*_XY_ here, we see the later is also reduced in this region, which again suggests it is likely that some sort of genome structure might be responsible for the peak in $F_{ST}$ we see. One way to investigate this is examine the recombination rate variation along chromosome 8 - which we will do in the next section.

# Investigating recombination rate variation

To check whether variation in recombination might explain the pattern we observed, we will read in the recombination rate estimated for 100 Kb windows with a 25 Kb step on chromoosme 8. This was originally estimated from a house sparrow linkage map, published by Elgvin et al (2018) (http://advances.sciencemag.org/content/3/6/e1602996) and you can download the data here (https://evolutionarygenetics.github.io/chr8_recomb.tsv). We will read the data in like normal

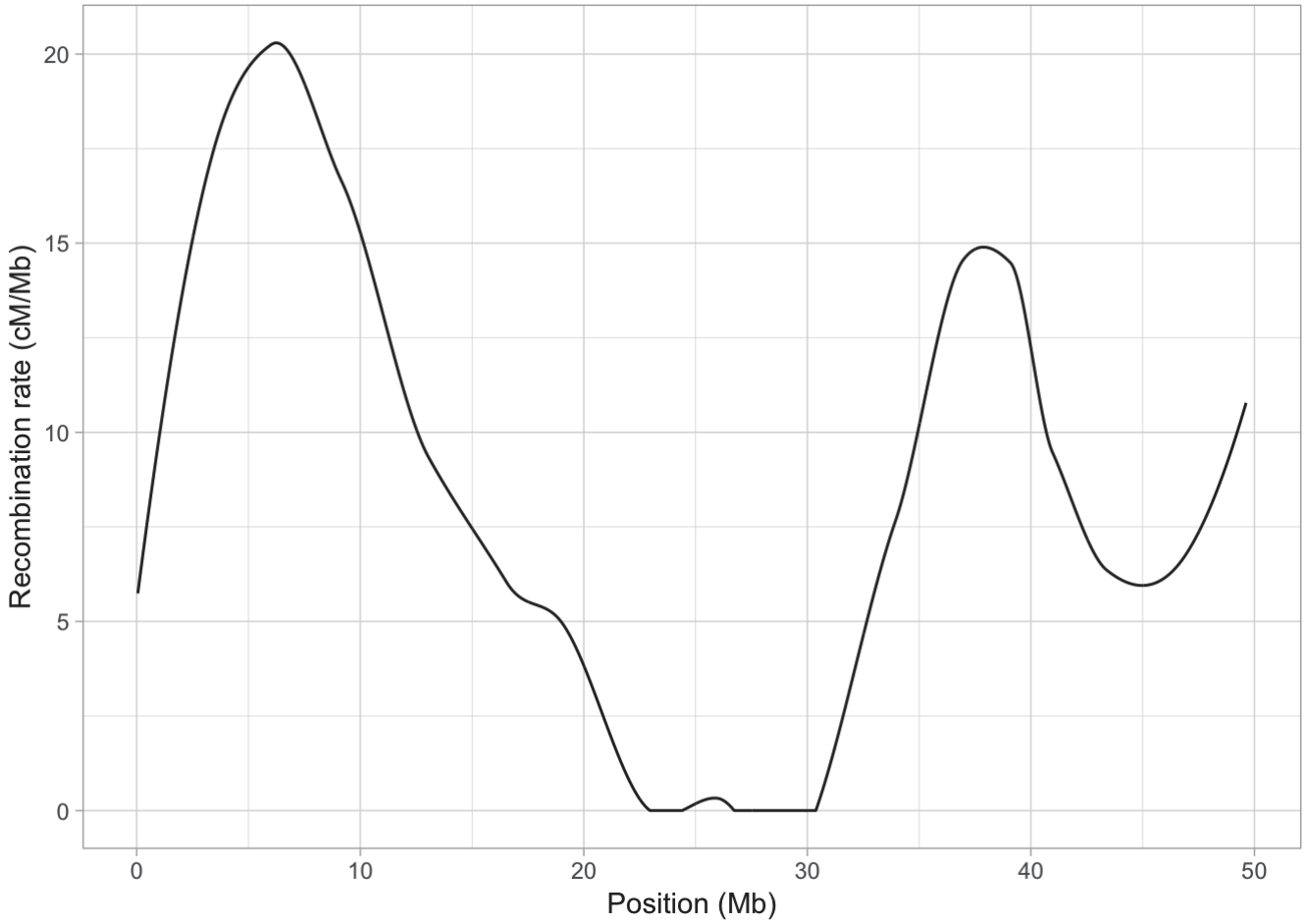
```
rrate <- read_delim("./chr8_recomb.tsv", delim = "\t")
```

Since the recombination rate is the same number of rows as our main dataset, we can just add it as a column.

```
# assign recombination rate to full sparrow dataset
sparrow_data$recomb <- rrate$recomb
```

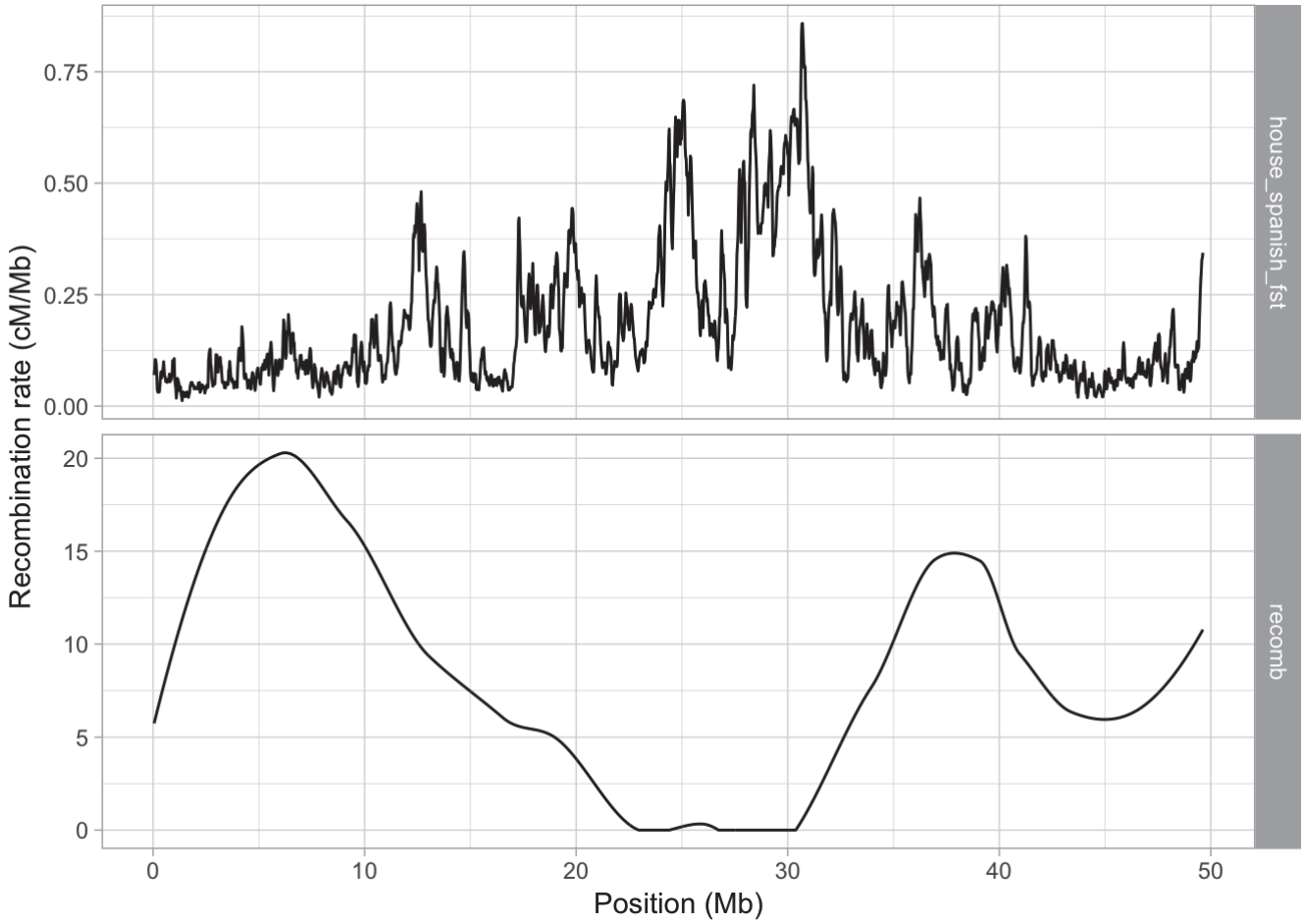
Now we are ready to see whether the variation in nucleotide diversity and $F_{ST}$ can be explained by recombination rate. Let's plot how it varies along the genome.

```
# construct a plot for recombination rate
a <- ggplot(sparrow_data, aes(mid/10^6, recomb)) + geom_line()
a <- a + xlab("Position (Mb)") + ylab("Recombination rate (cM/Mb)")
a + theme_light()
```




To explain this a little, we have plotted recombination rate in **centiMorgans per Megabase** - i.e. essentially the probability that a recombination event can occur. The higher this value is, the higher the probability of recombination. The first obvious point to take home from this figure is that our recombination rate varies quite significantly across the genome. Secondly, we see quite a drastic reduction in recombination rate between about 23 Mb and 30 Mb. This is exactly where our $F_{ST}$ peak occurs. to confirm this, we will plot both statistics together.

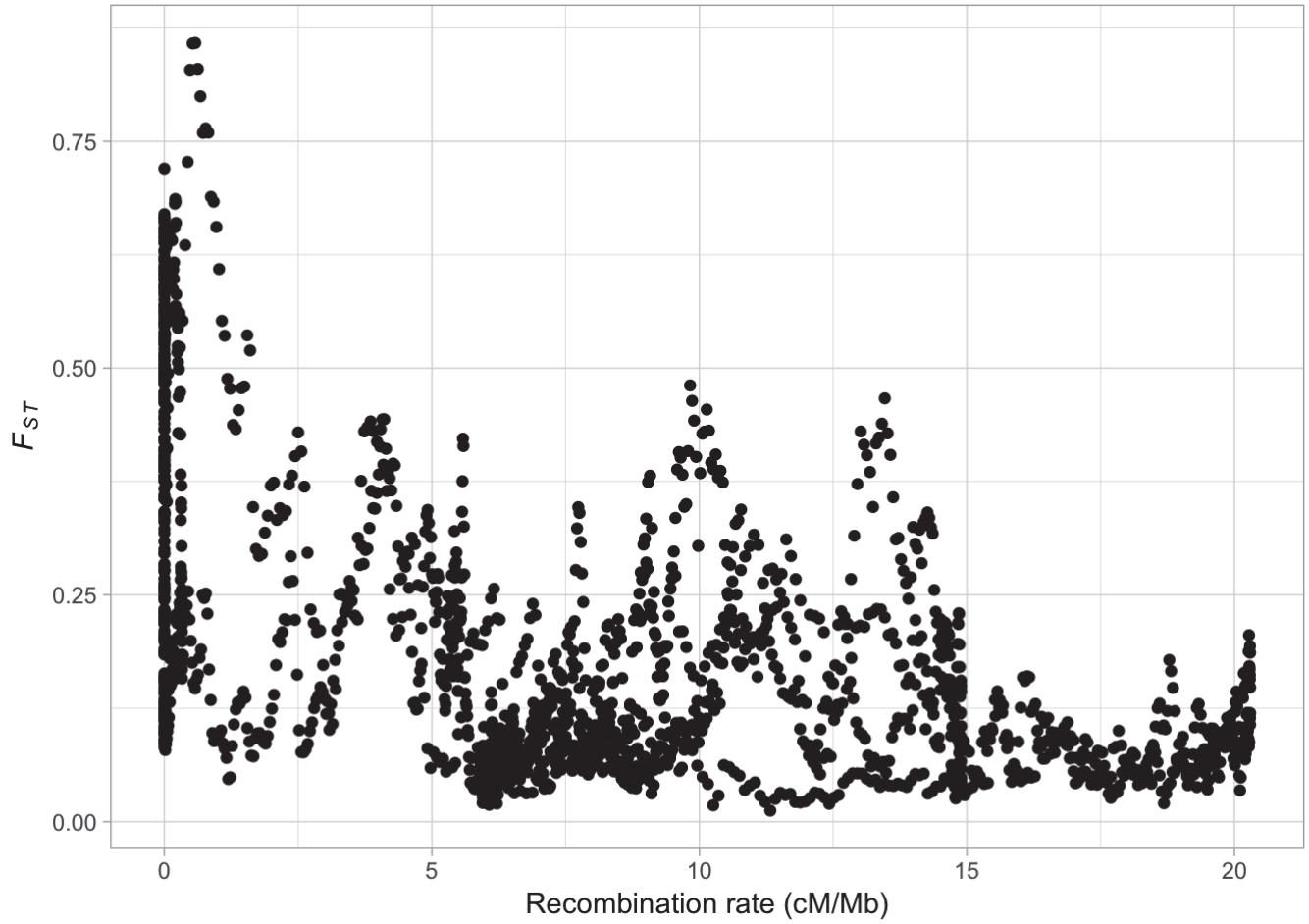
```
# subset data and gather
hr <- sparrow_data %>% select(mid, house_spanish_fst, recomb) %>% gather(-mid, key = "stat",
 value = "value")
# make a facet plot
a <- ggplot(hr, aes(mid/10^6, value)) + geom_line()
a <- a + facet_grid(stat~., scales = "free_y")
a <- a + xlab("Position (Mb)") + ylab("Recombination rate (cM/Mb)")
a + theme_light()
```




When we plot our data like this, it is actually more clear that perhaps both of the large peaks on chromosome 8 occur in an area of very low recombination. What could be causing such low recombination? Well one possibility is the centromere (https://en.wikipedia.org/wiki/Centromere) is likely to be present here.

Now that we have recombination data read into R, we can also explore the relationships between recombination rate and other statistics in more detail. To demonstrate this, we will plot the joint distribution of recombination rate and $F_{ST}$ between house and Spanish sparrows.

```
# plot recombination rate and fst
a <- ggplot(sparrow_data, aes(recomb, house_spanish_fst)) + geom_point()
a <- a + xlab("Recombination rate (cM/Mb)") + ylab(expression(italic(F[ST])))
a + theme_light()
```




Clearly there is a bias here - higher $F_{ST}$ values are found in regions of low recombination. Although this doesn't completely invalidate the use of $F_{ST}$ in speciation genomics, it does mean we must be cautious when using it to identify genes involved in speciation. If we had not done so here, it would have been quite easy to mistake the peak on chromosome 8 as having an important role in maintaining reproductive isolation between house and Spanish sparrows.

# Study questions

For study questions on this tutorial, download the `Chapter8_R_questions.R` from Canvas or find it here (https://evolutionarygenetics.github.io/Chapter8_R_questions.R).

# Going further

- A nice review on what we mean by speciation genomics (https://academic.oup.com/biolinnean/article/124/4/561/5035934)
- 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)
- Additional learning resources on speciation genomics using Python, R and Unix (http://evomics.org/learning/population-and-speciation-genomics/2018-population-and-speciation-genomics/)