Lecture 7

Effective population size

Effective population size is a translation of some of the characteristics of the population into a measure of “how big would the population need to be to...”.

There are several definitions of effective population size, we will see some more later. Now we deal with how quickly the frequency of a neutral allele changes.

Say we have a population with 50% A and 50% B. It splits into 100 different populations that evolve independently. How different will the frequencies be?

Say the population size of each small population is 50.

We will represent each population by the frequency of A

```r
> freqA = rep(0.5,100)
> next.gen = function( freq, N) {
  rbinom( length(freq), N, prob=freq )/N
}
> + + > next.gen( freqA, 50)
> res=matrix(0,20,100)
> freqA = rep(0.5,100)
> for(i in 1:20) {
  res[i,] = freqA
  freqA = next.gen( freqA, 500)
}
> + + > plot(c(0,20),0:1,type="n")
> for(i in 1:50) {
  lines( 1:20, res[,i])
}
> v()
>
This is a measure of how different the populations are.

Let us see how different they are depending on one generation

```r
[Om> Ns = (1:20)*10;Ns
> plot(sapply(Ns, function(N) var( next.gen(next.gen( freqA,N),N) ) ))
>
(This is two generations, because the first one just creates the populations.)

Maybe we should average over more values

```r
> Ns = matrix( Ns, 20, 20 )
> res = sapply(Ns, function(N) var( next.gen(next.gen( freqA,N),N) ) )
> length(res)
> dim(res)
> res = matrix( res, 20, 20)
> plot( Ns[,1],apply(res, 1, mean) )
> 
```
Now let us do the same, but instead of Wright-Fisher sampling, a different scheme: Each organism produces 2, and of those we sample N without replacement.

```r
freq = c(0.4, 0.6)
freq * 100 * 2
rhyper(1, (freq*100*2)[1], (freq*100*2)[2], 100)

next.gen.H = function(freqA, N, offs=2) {
  n.oA = freqA * N * offs
  n.oB = (1-freqA) * N * offs
  rhyper(length(freqA), n.oA, n.oB, N)/N
}

freqA = rep(0.5, 100)
next.gen.H(freqA, 10)
res.H = sapply(Ns, function(N) var(next.gen.H(freqA,N),N))
points(Ns[,1], apply(res.H, 1, mean), col=2)
apply(res.H, 1, mean)
plot(Ns[,1], apply(res, 1, mean), col=1, ylim=c(0, 0.13))
points(Ns[,1], apply(res.H, 1, mean), col=2)
```

**Vectors and data types**

Till now we only used vectors. Vectors only use one type of data:

```r
x=1:10
x
x[5] = "a"
x

To convert between the different types, we use the various `as` functions

```r
as.numeric(x)
as.character(1:10)
as.integer(x)
as.double(x)
```

**Lists**

To store data of different type, or various length, we use lists.

```r
x=list("Z",1:10, matrix(1:9,3,3))
x
```

**Accessing data in a list**

To access a certain element in a list, use `[[index]]`: 

```r
```
If you use a single braket, you get back a list:

```r
data <- c(1, 2, 3)
data[1:2]
data[2:3]
data[1]
```

This mistake is easy to make. You want to access an item of a list, and get this strange thing back.

```r
data[c(T,F)]
data[-1]
```

You can use `c()` to concatenate lists:

```r
data <- c(1, 2, 3)
data <- c(data, 4)
data <- c(data, 1)
data <- data
data[1:2] <- 4
data <- data
```

Items in lists can have names:

```r
data <- c(1, 2, 3)
names(data) <- c("first","second","third")
data <- data
```

For lists, you can also use `$` to access a certain item

```r
data$first
```

And, in this case, you don’t have to specify the whole name:

```r
data$fir
```

To remove an item from a list, set it to `c()`, or to `NULL`:

```r
data <- c(1, 2, 3)
data[[2]] <- c()
data <- data
```
An item that doesn’t exist in the list, will look the same:

```r
> x
> x[4]
> x["tenth"]
> names(x)
>
```

Lists can hold it all, even lists:

```r
> x
> x[[3]]=x
> x
> x[[1]]
> x[[2]]
> x[[3]]
>
```

## Data.frames

Most of the time, you’ll probably use lists without even noticing that you have a list. You’ll be using data.frames

```r
> x=data.frame( a=1:10, b=1:10)
> x
> x[3,]
> x[,1]
> x[,"a"]
>
```

Most of the time, you will not even notice that you have a data frame, and not a matrix. But - NEVER FORGET! It will help you avoid bugs.

```r
> x
> x$c = sample(c("G","T"),10,rep=T)
> x
[0m> x[3,]
> x[3,1]+3
> x[3,2]+5
> x[3,3]
> x[3,]
> x[3,]+1
[0m>

You see, even though the 3rd line of x looks like a matrix, it isn’t. It can hold different data types.

```r
[0m> x
>

When you work with data inside x, and you want a matrix, make a matrix

```r
> x[,1:2]
> as.matrix(x[,1:2])
```
That’s a matrix!

Look

```r
> not.Mat = x[,1:2]
> Mat = as.matrix(x[,1:2])
> not.Mat
> Mat
> not.Mat[1]
> Mat[1]
>
That is because `not.Mat` is a list, not a vector.

```r
> not.Mat[[1]]
> not.Mat[[2]]
>
```

Reading data

There are several ways to read data.

The easiest is `read.table()`

```r
> a=read.table("~/data.csv",sep="","")
> a
> a[,1]
> a[1,]
>
If you have non-numeric values in your table, then these will be read as factors

```r
> a[,2]

```r
[0m>

Normaly this is not what you want. (I almost never use factors.) The benefits of factors are that factors take up less space, and comparison of factors is faster, since R just gives each possible factor a number. To compare strings, R has to go through the whole string to make sure they are identical.

To read this as stings:

```r
[0m> a=read.table("~/data.csv",sep="", as.is=T)
> a[,2]
>
```r

```r
> a=read.table("~/data.csv",sep="","")
> a=read.table("~/data.csv",sep="",fill=T,as.is=T)
> a
> a[,1]
>
```
Coalescent simulation

```r
> m=t(replicate(20,sort(sample(1:10,rep=T))))
> plot(1:2,1:2,xlim=c(0,(dim(m)[2]+1)),ylim=c(0,(dim(m)[1]+1)),type="n")
> for(i in 20:2) {
>   for(j in 1:10) {
>     lines( c(m[i,j],j)+0.5, c(21,20)-i+2 )
>   }
> }
>
> pop = 1:10
> res = matrix(0,21,10)
> res[1,] = pop
> for(i in 2:21) {
>   res[i,] = res[i-1,] [m[i-1,]]
> }
> image(1:11,1:21,t(res[21:1,]),col=rainbow(length(res[1,])))
> for(i in 20:1) {
>   for(j in 1:10) {
>     lines( c(m[i,j],j)+0.5, c(21,20)-i+1 )
>   }
> }
>
> coalesce = function(x,m) {
>   lineage = list()
>   n.gen = dim(m)[1]
>   for(i in n.gen:1 ) {
>     lineage[[i]] = x
>     x = unique( m[i,x] )
>   }
>   lineage
> }
> plot.offspring = function( m, coal = list(), ... ) {
>   n.gen = dim(m)[1]
>   N = dim(m)[2]
>   for(i in n.gen:1) {
>     if( length(coal) ==0 ) { Js = N:1 }
>     else { Js = coal[[i]]}
>     for(j in Js) {
>       lines( c(m[i,j],j)+0.5, c(21,20)-i+1, ... )
>     }
>   }
> }
coal = coalesce(1:3,m)

plot.offspring(m,coal=coal,col=2,lwd=3)

This can be done by the program `ms`

```r
> system("./ms 20 1 -t 1000 >out.txt")
```

```
1.573 sec

> res=read.table("out.txt",skip=6, colClasses="character")
```

```
171 msec
```

```
22.995 sec
```

```
> res
```

```
V1
1 010111
2 101000
3 010111
4 010111
```

```
143 msec
```

```
> let us convert this into genotypes
```

```
> x=strsplit(res[,1],"
```

```
> y=matrix(x,ncol=length(x))
```

```
> y=matrix(as.numeric(unlist(x)),ncol=length(x))
```

```
y
dim(y)
table(apply(y,1,sum)).
hist(apply(y,1,sum),breaks=(1:20)-0.5);v()
dim(cor(y))
image(cor(y))
o=order(cor(y)[1,])
y=y[,o]
```

> image(cor(y))
>
> hamming = function(i,j) { sum( y[,i]!=y[,j])
> hamming(1,2)
> z=outer(1:20,1:20,hamming)
> hamming(1:2,3:4)
> a1
>
> [0m> dim(a)
>
> This doesn't work. The function that we supply to outer has to be able to work on vectors

> hamming = function(i,j) {
  A = y[,i]
  B = y[,j]
  M = A != B
  apply(M,2,sum)
}
>
> hamming(1,2)
> hamming(1:2,3:4)
> hamming(1,3)
> hamming(2,4)
> res=outer(1:20,1:20,hamming)
> image(res)
> res
> h=hclust(dist(t(y)))
> plot(h);v()
> hamming(3,4)
> hamming(2,3)
> hamming(2,4)
> hamming(1,13)
>
>
> m=t(replicate(20,sort(sample(1:10,rep=T)))))

97 msec
> plot(1:2,1:2,xlim=c(0,(dim(m)[2]+1)),ylim=c(0,(dim(m)[1]+1)),type="n")  

112 msec

> for(i in 20:2) {
    for(j in 1:10) {
        lines( c(m[i,j],j)+0.5, c(21,20)-i+2 )
    }
}

131 msec

+ + + +

1.679 sec

> + + + + pop = 1:10
    res = matrix(0,21,10)
    res[1,] = pop

100 msec

> > > for(i in 2:21) {
    res[i,] = res[i-1,] [m[i-1,]]
}

106 msec

> > > image(1:11,1:21,t(res[21:1,]),col=rainbow(length(res[1,])))

120 msec

> for(i in 20:1) {
    for(j in 1:10) {
        lines( c(m[i,j],j)+0.5, c(21,20)-i+1 )
    }
}

134 msec

+ + + + coalesce = function(x,m) {
    lineage = list()
    n.gen = dim(m)[1]
    for(i in n.gen:1) {
        lineage[[i]] = x
        x = unique( m[i,x] )
    }
    lineage
}

104 msec

+ + + + + + + + + + + plot.offspring = function( m, coal = list(), ... ) {
    n.gen = dim(m)[1]
    N = dim(m)[2]
    for(i in n.gen:1) {
        if( length(coal) ==0 ) { Js = N:1 }
        else { Js = coal[[i]] }
        for(j in Js) {
            lines( c(m[i,j],j)+0.5, c(21,20)-i+1, ... )
        }
    }
}

107 msec

+ + + + + + + + + + + coal = coalesce( 1:3,m)

72 msec
> plot.offspring(m, coal=coal, col=2, lwd=3)

109 msec

>