Análisis de estrutura de la población y demográfico con *Lupinus*

Tabla de contenido:

Estructura de la población

PCA

Ahora que ya tenemos nuestro VCF listo, podemos empezar a hacer análisis. Primero, verificaremos la estructura de la población mediante un análisis de componentes principales (PCA). PCA es un método sin modelo y sin necesidad de una asignación de población previa que ayuda a identificar la estructura de la población y la ascendencia compartida. En el caso de los datos genéticos, PCA resumirá la variación en términos de frecuencias alélicas. Es decir, PCA identifica los principales ejes de variación basado en las frecuencias alélicas, con el primer componente explicando la variación principal observada, el segundo el siguiente más grande, y así sucesivamente. Al hacer esto, calculará las coordenadas de los individuos a lo largo de los ejes para posicionar las muestras en el conjunto de datos. Para realizar un PCA en los datos de *Lupinus*, usaremos el software *plink - versión 1.9*.

1. Vamos crear una carpeta llamada "demografia" donde vamos hacer estos analisis
2. dentro de esta carpeta, crea una otra llamada "PCA"
3. el archivo de input que vamos usar es el formato plink generado por Stacks en *populations* = **populations.plink.ped** y **populations.plink.map**. Entonces tenemos que obtener la ruta de estes archivos (`pwd`)
4. necesitamos cargar el módulo de *plink* en el cluster: `module load plink`
5. ahora podemos ejecutar plink usando el archivo de Stacks (`-file`) para calcular el PCA (`--pca`). Además, pediremos a plink que genere un formato "corregido" (`--recode12`) para ejecutar el análisis de *admixture* que sigue abajo

Output: 6 archivos: 2 con los resultados de los PCA (plink.eigenval y plink.eigenvec), 2 que vamos usar en *admixture* (plink.ped y plink.map) y 2 con información sobre la ejecución y los datos (plink.log y plink.nosex).

6. vamos descargar (`scp`) los outputs del PCA (plink.eigenval y plink.eigenvec) para graficar usando R

7. abra R Studio
8. primero vamos cargar los paquetes que necesitamos y setar el directorio

```
library(tidyverse)
library(ggplot2)

setwd("<camino_donde_esta_los_archivos>")
```

9. vamos ler los archivos generados por *plink* en R

```
pca <- read_table2("./plink.eigenvec", col_names = FALSE)

eigenval <- scan("./plink.eigenval")
```

10. necesitamos añadir nombres para las columnas del PCA

```
names(pca)[1] <- "Species"
names(pca)[2] <- "Ind"
names(pca)[3:ncol(pca)] <- paste0("PC", 1:(ncol(pca)-2))
```

11. y cambiar los valores de eigen para porcentaje

```
pve <- data.frame(PC = 1:length(eigenval), pve = eigenval/sum(eigenval)*100)
```

12. ahora si, ¡plotando! Primero, vamos plotar la variancia explicada por cada componente principal

```
a <- ggplot(pve, aes(PC, pve)) +
    geom_bar(stat = "identity") +
    ylab("Percentage de la variancia") +
    theme_light(); a
```

13. y vamos plotar el PCA

```
b <- ggplot(pca, aes(PC1, PC2, col = Species)) +
    geom_point(size = 3) +
    scale_colour_manual(values = c("red", "blue")) +
    coord_equal() +
    theme_light() +
    xlab(paste0("PC1 (", signif(pve$pve[1], 3), "%)")) +
    ylab(paste0("PC2 (", signif(pve$pve[2], 3), "%)")); b
```

¡Listo! :smile:

Admixture

El software *Admixture* - Manual - hace la estimación de máxima verosimilitud de la ancestralidad de los individuos basado en datos de genótipos multilocus (SNP).

Suposiciones del modelo: - los SNP no están vinculados - los individuos no san relacionados - los sitios son bialélicos y se eliminan los singletons - gran

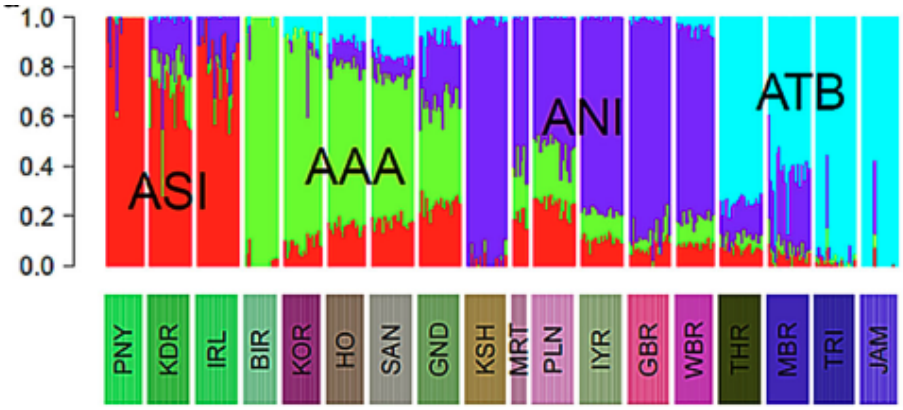

Figure 1: Un gráfico de *admixture* (imagen de: Lawson et al., 2018)

cantidad de SNP (~ 10,000) - poblaciones con un número similar de individuos (no se detectarán poblaciones subrepresentadas).

Es importante tener en cuenta que diferentes historias evolutivas pueden generar un patrón muy similar de estructura poblacional, por lo que se necesita mucho cuidado al interpretar los resultados (para más información, mire Lawson et al., 2018).

Otros softwares para detectar estructura poblacional: - STRUCTURE - FASTSTRUCTURE - CONSTRUCT - y muchos otros

1. Dentro de la carpeta "demografia", vamos crear una llamada "admixture"
2. Entra en la carpeta "admixture"
3. el archivo de input que vamos usar es el generado por *plink* cuando hicemos el PCA, que está en el camino relativo `../PCA/plink.ped`
4. necesitamos cargar el modulo en el cluster: `module load admixture`
5. la línea de comando para correr *admixture* és: `admixture <archivo> <#K>`
6. debido a que queremos comparar diferentes números de K (de 1 a 5), lo ejecutaremos usando validación cruzada, por lo que necesitamos generar un *for loop*:

```
for k in {1..5};
    do admixture --cv=10 <ruta_del_archivo> $k > log${k}.out;
done
```

Output: 3 archivos (log.out, .P y .Q) para cada valor de K.

7. vamos descargar (`scp`) los outputs del *admixture* para graficar en R
8. en R Studio, primero vamos cargar los paquetes que necesitamos y setar el directorio

```
library(ggplot2)
```

```
setwd("<camino_donde_esta_los_archivos>")
```

9. vamos mirar los likelihoods de cada K

```
#lista los archivos en la carpeta con extensión .out
f <- list.files(path = ".", pattern = ".out")

#crea un data.frame vacío
lh_K <- data.frame(K = numeric(),
                   CV_error = numeric())

#for loop para sacar el valor the K y CV error de cada archivo
for(i in f){
  temp <- readLines(i)[grep("CV", readLines(i))]
  k <- as.integer(sub(".*?K=*?(\\d+).*", "\\1", temp))
  cv <- as.numeric(sub(".*?): *?(\\d+(?:.\\d+)).*", "\\1", temp))
  lh_K[nrow(lh_K) + 1,] <- c(k, cv)
}

print(lh_K)
```

10. plotando los CV error

```
a <- ggplot(lh_K, aes(x = K, y = CV_error)) +
  geom_point() +
  geom_line() +
  ylab("Cross-validation error") +
  xlab("K") +
  theme_light(); a
```

Pregunta: ¿Cual valor de K es lo más probable para esto dato?

11. ahora que ya descobrimos cual K es lo numero de clusters mas probable, vamos plotar la ancestralidad estimada por admixture para cada individuo. Pero antes de plotar necesitamos ler la matrix Q.

```
#ler la matrix Q para el K más probable
q <- read.table("populations_cor.2.Q")

#plot
barplot(t(as.matrix(q)), col = c("#ef8a62", "#91bfdb"), xlab = "Individual #", ylab = "Ances
```

Demografia

En esta parte estimaremos algunos parámetros demográficos (tamanho poblacional, tiempo de divergencia) basados en nuestro conjunto de datos y com-

pararemos dos modelos demográficos de divergencia, aislamiento sin (A) y con migración (B), para verificar cuál de estos modelos es el más probable.

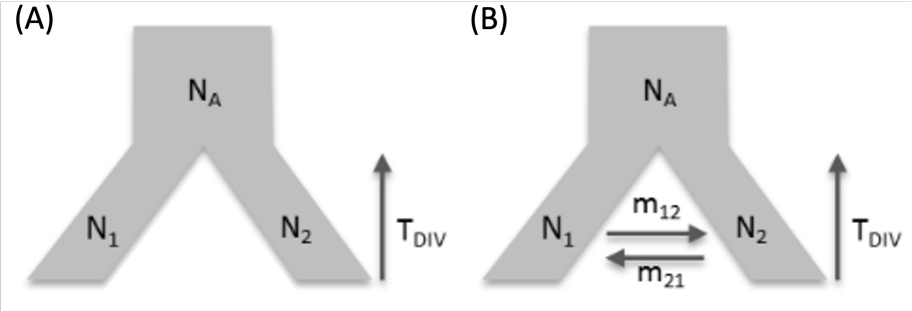

Figure 2: Los dos modelos de isolamento sin (A) y con migración (B) que vamos comparar (imagen modificada del manual de Fastsimcoal)

Inferindo el SFS

Antes de empezar las estimaciones demográficas, necesitamos transformar nuestros datos al *Site Frequency Spectrum* (SFS). Cada entrada en el SFS indica el número total de loci con la correspondiente frecuencia de alelos derivados.

Cálculo de **SFS** y **folded SFS** para 10bp y n = 6 individuos haploides. El SFS vás tener n-1 = 5 clases, porque cualquier variante de SNP podría ocurrir en 1/6, 2/6, ..., o 5/6 individuos: la base que ocurre en 6/6 los individuos serían invariantes. El SFS se obtiene contando el número de SNP derivados con respecto a los I ["1"s en recuadros grises] individuales. Cuando no está claro si el estado del carácter en el individuo #1 es ancestral o derivado, el SFS se "dobla" (**folded SFS**) combinando las clases "1" y "5" (ambas clases tienen un carácter de una manera y cinco de otra), y el clases "2" y "4" (ambas tienen dos en un sentido y tres en el otro). La clase '3' permanece sin cambios (combina los tipos equilibrados 000111 y 111000).

Es importante tener en cuenta que existen diferentes tipos de SFS (para una, dos o muchas poblaciones, si conoces el estado derivado o no) - mirar acá para una buena explicación. Esto dependerá de lo que tu quieres hacer con *fastsimcoal*. Para obtener más información al respecto, consulte el manual de fastsimcoal. Además, existen otros softwares que pueden generar el SFS, como Arlequin, angsd y dadi.

Acá, vamos utilizar un script llamado *easySFS* para generar el SFS de nuestro archivo vcf. Como tenemos dos poblaciones, vamos estimar el *joint* SFS o 2D SFS (= "minor allele frequency SFS" = *jointMAF*).

1. crear una carpeta "fastsimcoal" dentro de la carpeta "demografia"
2. entre en la carpeta "fastsimcoal"

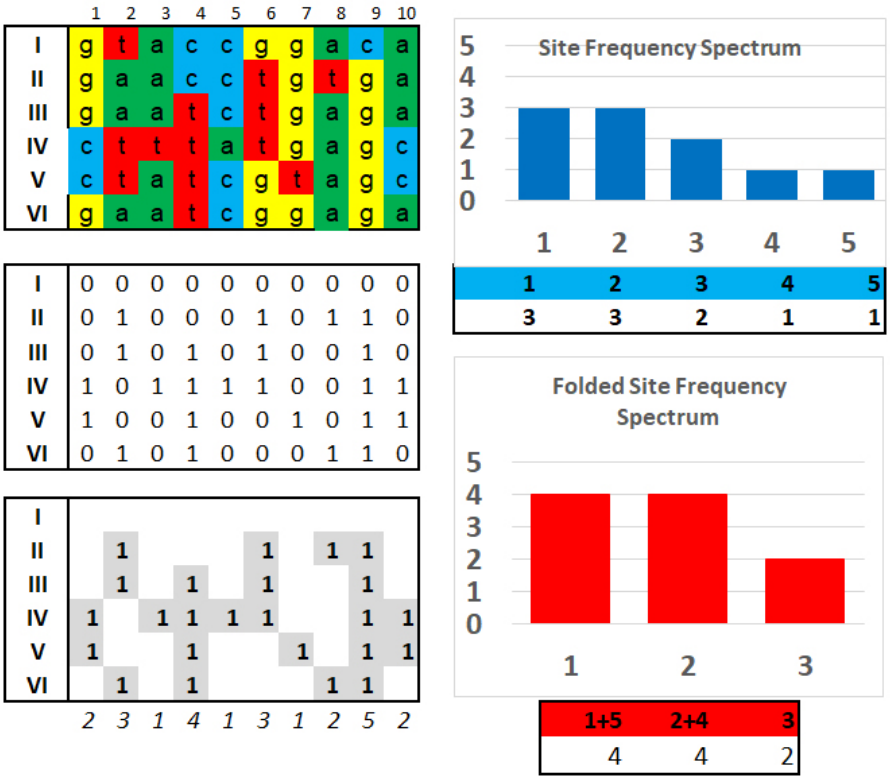

Figure 3: SFS y folded SFS (imagen de acá)

3. ahora vamos clonar el repositorio: `git clone https://github.com/isaacovercast/easySFS.git`
4. `cd easySFS`
5. tenemos que cargar un ambiente conda para lograr ejecutar el script `conda active SFS` (si algún día vas a ejecutar este programa en tu computadora, tendrás que instalar algunas dependencias que ya están instaladas aquí, para eso, sigue las instrucciones que están en el repositorio de github).
6. necesitamos de 2 archivos para hacer el SFS: el vcf (con uno solo SNP per loci) y el archivo de población que usamos con stacks, mira las rutas donde están estos archivos.
7. ahora estamos listos para generar el SFS en dos pasos: `./easySFS.py -i <ruta_vcf>/populations.snps.vcf -p <ruta_pop_map>/Lupinus_pops.txt --preview`

Que dará como resultado algo como esto:

```
L_alopecuroides
(2, 215)    (3, 322)    (4, 389)    (5, 435)    (6, 467)    (7, 491)    (8, 507)    (9, 209)

L_triananus
(2, 457)    (3, 685)    (4, 823)    (5, 916)    (6, 980)    (7, 1023)   (8, 1051)   (9, 538)
```

Cada columna es el número de muestras (2n) en la proyección y el número de sitios segregantes en ese valor de proyección. El manual dadi recomienda maximizar el número de sitios segregantes, pero al mismo tiempo, si tiene muchos datos faltantes, es posible que deba equilibrar el número de sitios segregados con el número de muestras para evitar reducir demasiado el muestreo.

A continuación, ejecute el script con los valores para proyectar para cada población (x,y), así:

```
./easySFS.py -i <ruta_vcf>/populations.snps.vcf -p <ruta_pop_map>/Lupinus_pops.txt
--proj x,y
```

Esto vas generar una carpeta `output` con archivos input para el software *dadi* y *fastsimcoal*. Usaremos el jointMAF producido para fastsimcoal (`populations_jointMAFpop1_0.obs`).

Fastsimcoal

Ahora que ya tenemos nuestro jointSFS, finalmente podemos ejecutar `fastsimcoal2` :) Fastsimcoal es un software de modelado demográfico permitiendo probar varios escenarios demográficos con diferentes niveles de complejidad dependiendo de sus datos. Utiliza el SFS para ajustar los parámetros del modelo a los datos observados mediante la realización de simulaciones coalescentes. El manual es bastante completo y se puede encontrar aquí.

Antes de empezar, vamos hacer el download del software:`wget <CAMINO>`

y descomprimir el archivo zip : `unzip <ARCHIVO>.zip`

Adentre en la carpeta `cd`, hace el archivo executable: `chmod +x fsc2702` y estamos listos :smile:

Archivos de entrada:

Para ejecutar fastsimcoal2 necesitamos 3 archivos en una mesma carpeta. Los 3 archivos deben tener el mismo prefijo pero diferente extensión. Todos son archivos de texto: - SFS observado = `${PREFIX}_jointMAFpop1_0.obs`, ya generado por `easySFS` - archivo que define el modelo demográfico = `${PREFIX}.tpl` - archivo que define los parámetros = `${PREFIX}.est`

Para los archivos `.tpl` y `.est` vamos modificar de un archivo ejemplo en la carpeta `example\ files/2PopDiv20Mb`. Para esto, vamos volver a nuestra carpeta inicial de `fastsimcoal` y crear una carpeta llamada `modelos`. Dentro de `modelos`, vamos generar dos carpetas: `div` y `div_mig` - una a cada modelo demográfico que probaremos.

Modelo de divergencia `div` En la carpeta `div` vamos a copiar de los archivos de ejemplo: `cp <RUTA>/dato_denovo/demographic/fastsimcoal/fsc27_linux64/example\ files/2PopDiv20Mb/2PopDiv20Mb.* .` También tenemos que copiar el SFS. Recuerden que todos los 3 archivos deben tener el mismo prefijo. Entonces cambiemos (`mv`) los prefijos de los archivos a algo significativo, como el nombre del organismo y el modelo: `Lupinus_div`. Todos los archivos deberían verse así:

```
Lupinus_div.est  Lupinus_div.tpl  Lupinus_div_jointMAFpop1_0.obs
```

Usando `nano`, vamos modificar los archivos `.est` y `.tpl`. En el `.tpl` tenemos que cambiar el numero de los individuos para lo que establecemos en el SFS y la taja de mutación (usaremos 7e-9). En el `.est`, podemos dejar los *priors* como están, pero es importante mirarlos y comprender lo que pasa.

Estamos casi listos para hacer nuestra primera prueba, pero necesitamos crear la línea de comando y el archivo *sbatch*.

Primero intentemos directamente en la línea de comando: 1. descubrir la ruta del fastsimcoal y generar una variable para esto: `ruta_fastsimcoal=<RUTA>/fsc2702` 2. en el terminal, crear una variable con el nombre del prefijo: `PREFIX="Lupinus_div"` 3. vamos ejecutar fastsimcoal: `<PATH_FASTSIMCOAL>/fsc2702` con estas con estas opciones: `-t ${PREFIX}.tpl -e ${PREFIX}.est -n 250000` número de simulaciones coalescentes para aproximar el SFS esperado en cada ciclo `-m` utilizar frecuencia de alelos menores (MAF) `--removeZeroSFS` ignorar los sitios monomórficos `-M` estimación de parámetros por máxima probabilidad compuesta de la SFS `-L 40` número de ciclos de optimización (ECM) para estimar los parámetros `-q` pocas mensajes a la consola (no detallado)

Ahora vamos hacer un *sbatch* por qué tenemos que ejecutar fastsimcoal mucho más de una vez. `fastsimcoal2` no debe ejecutarse una sola vez porque es posible que no encuentre la mejor combinación de estimaciones de parámetros de inmediato. Es mejor ejecutarlo 50 veces o más. De estas ejecuciones, seleccione la que tenga el *likelihood* más alto, que es la ejecución con las estimaciones de parámetros de

mejor ajuste para este modelo. Debido a limitaciones de tiempo, ejecutaremos este modelo solo 5 veces. En el *sbatch* vamos añadir una línea que indica ejecutar varias veces el mismo comando: `#SBATCH --array=1-5`. En este caso, *sbatch* ejecutará fastsimcoal de 1 a 5, generando la variable `${SLURM_ARRAY_TASK_ID}` con el número de ejecución. Usaremos esta variable para generar una carpeta para cada ejecución y evitar sobrescribir los resultados. Entonces, en el *sbatch* vamos poner:

```
ruta_fastsimcoal=<RUTA>/fsc2702
PREFIX="Lupinus_div"

cd <RUTA_MODELO_DIV>

mkdir Run${SLURM_ARRAY_TASK_ID}
cp ${PREFIX}.est Run${SLURM_ARRAY_TASK_ID}/
cp ${PREFIX}.tpl Run${SLURM_ARRAY_TASK_ID}/
cp ${PREFIX}_jointMAFpop1_0.obs Run${SLURM_ARRAY_TASK_ID}/
cd Run${SLURM_ARRAY_TASK_ID}/

$ruta_fastsimcoal -t ${PREFIX}.tpl -e ${PREFIX}.est -n 250000 -m --removeZeroSFS -M -L 40 -c

cp ${PREFIX}/${PREFIX}.bestlhoods ../bestlhoods/${PREFIX}_${SLURM_ARRAY_TASK_ID}.bestlhoods
```

Antes de ejecutar el *sbatch*, no se olvide de generar una nueva carpeta llamada `bestlhoods` que es donde vamos poner los resultados para mirar el mejor likelihood.

Memoria necesaria: ~70mb

Tiempo de ejecución: ~11min

Output: en cada carpeta Run* vamos tener una nueva carpeta con el nombre del ${PREFIX}. En esta carpeta tenemos 5 archivos. Los más importantes para nosotros son ${PREFIX}.bestlhoods y ${PREFIX}_maxL.par

Los archivos ${PREFIX}.bestlhoods ya copiamos a la carpeta `bestlhoods`. Naveguemos a esa carpeta y comparemos qué *run* presenta el mejor *likelihood*. Para esto, vamos ejecutar la línea: `cat ${PREFIX}_{1..5}.bestlhoods | grep -v MaxObsLhood | awk '{print NR,$5}' | sort -k 2`

En cada archivo, vamos sacar el encabezamiento con `grep`, después vamos mostrar en la pantalla al número de ejecución que acá corresponde el número de línea usando el comando `awk '{print NR,$5` donde $5 es la columna del *MaxEstLhood* y, por lo tanto, la probabilidad que queremos comparar entre diferentes ejecuciones.

Modelo de divergencia con migración `div_mig` Ahora que del modelo *div* ya esta siendo ejecutado, vamos generar los archivos para el modelo de divergencia con migración `div_mig`. Copie (`cp`) los archivos del `div` para `div_mig`. En este modelo, tenemos que modificar los archivos para añadir una matriz de migración

en el `.tpl` y los parámetros de migración en el `.est`. Vamos intentar hacer estas modificaciones en grupos. Un consejo es echar un vistazo a la página 77 del manual.

Memoria necesaria: ~70mb

Tiempo de ejecución: ~21min

Output: similar al modelo `div`, pero con más columnas en algunos archivos porque este modelo tiene más parámetros para estimar

De manera similar al modelo `div`, modifique el comando `cat` de arriba para comparar qué *run* presenta el mejor *likelihood*.

```
PREFIX="Lupinus_divMig"
cat ${PREFIX}_{1..5}.bestlhoods | grep -v MaxObsLhood | awk '{print NR,$7}' | sort -k 2
```

Comparación de modelos con AIC Vamos usar *Akaike information criterion* (AIC) para comparar los dos modelos que ejecutamos. Se calcula como:

Dónde: K: el número de parámetros del modelo *ln(L)*: la probabilidad logarítmica del modelo - fastsimcoal ya calcula este valor automáticamente.

El AIC encontra el modelo que explica la mayor variación en los datos, mientras que penaliza a los modelos que utilizan un número excesivo de parámetros. Cuanto menor sea el AIC, mejor se ajustará al modelo. Aquí porque solo estamos comparando dos modelos, calculemos manualmente AIC, en R:

```
#los valores de log(likelihood) para cada modelo
div <- <VALOR>
div_mig <- <VALOR>

#el numero de parametros para cada modelo
k_div <- <NUM_PARAM>
k_div_mig <- <NUM_PARAM>

#convertir de log10 a ln
ln_div <- div/log10(exp(1))
ln_div

ln_div_mig <- div_mig/log10(exp(1))
ln_div_mig

#calcular el AIC de cada modelo
AIC_div <- 2*k_div-2*ln_div
AIC_div
AIC_div_mig <- 2*k_div_mig-2*ln_div_mig
AIC_div_mig
```

PREGUNTA: ¿Qué modelo explica mejor los datos?

Bootstrap Ahora que sabemos cuál de los modelos es el mejor, podemos hacer *bootstrapping* para averiguar qué tan seguros estamos de nuestras estimaciones de parámetros. Para esto vamos obtener intervalos de confianza haciendo *parametric bootstraps* - en las páginas 58-60 del manual hay una buena explicación.

La idea és que utilizemos el archivo que contiene las estimaciones de los parámetros (*_maxL.par*) de la ejecución com mejor* likelihood* , para simular 100 SFS y ejecutar *fastsimcoal* basado en estos SFS simulados. De esta forma, descubriremos cómo nuestros datos tienen poder para inferir correctamente los parámetros estimados.

1. crear una carpeta `bootstrap` en la carpeta `fastsimcoal`
2. copiar el archivo `\*_maxL.par` de la mejor *run* del mejor modelo para la carpeta `bootstrap` - vamos modificar esto archivo
3. tenemos que mirar cuantos SNPs se usaron en *easySFS* para modificar el archivo `\*_maxL.par`. Para esto, miraamos el archivo `datadict.txt` creado por *easySFS*. Usando el comando `wc` podemos descubrir cuántas líneas hay en este archivo - ese es el número de SNP utilizados.
4. Modificar `\*_maxL.par`:

- En la línea abajo de "//Number of independent loci [chromosome]" sustituir 1 por el número de SNPs utilizados por *easySFS*
- sustituir la última línea por: FREQ por DNA y 1 por 100, que se quede así: `DNA 100 0 7e-9 OUTEXP`

5. retirar *_maxL* del nombre del archivo (`mv`)
6. Ejecute fastsimcoal con esto `.par` modificado para generar 10 SFS simulados. Por razones de tiempo, acá vamos simular solo 10 SFSs, pero el correcto es generar mucho más, como 100.

```
$ruta_fastsimcoal -i Lupinus_div.par -n10 -j -d -s0 -x -I -q
```

Esto generará 10 SFS (`-n`)en diez subdirectorios separados (`-j`) en el directorio `Lupinus_div`. Además,
`-d` calcula SFS para los sitios derivados (acá yo penso que tenemos que usar `-m`, pero la versión 2.7 esta con un *bug*. Yo escribí en el google groups preguntando a cerca de esto problema)
`-s0` generar SNPs de los datos de ADN - especificar el número máximo de SNP para generar (use 0 para generar todos los SNP)
`-x` no genera *Arlequin* output `-I` genera mutaciones de acuerdo con el modelo de mutación de sitio infinito (IS)
`-q` salida de mensaje mínima en el console

7. con este procedimiento, se puede estimar los parámetros de estos SFS simulados utilizando los mismos `.tpl` y `.est` de la ejecución inicial de *fastsimcoal*. Entonces tenemos que copiar `.tpl` y `.est` para cada una de las carpetas con los SFS simulados. En la carpeta del modelo que tenga los archivos `.tpl` y `.est`, hacer: `for i in {1..10}; do cp Lupinus_div.* <RUTA>/bootstrap/Lupinus_div/Lupinus_div_$i; done`

8. Ahora que ya tenemos los archivos listos para hacer el bootstrap, vamos copiar y modificar el `.sh` usado en la ejecución inicial de *fastsimcoal* para hacer los *bootstraps*. Primero copie el `.sh` para la carpeta `bootstrap`. Ahora vamos cambiar el archivo:
9. cambiar `job_name` y el `--array=1-10`
10. cambiar la ruta para la carpeta `boostrap/Lupinus_div/${PREFIX}_${SLURM_ARRAY_TASK_ID}_`
11. como ahora las carpetas ya están todas generadas de antemano, tenemos que remover las r líneas con los comandos `mkdir`, `cp` y `cd`
12. en la línea de comando del *fastsimcoal*, cambiar la opción `-m` por `-d`
13. cambiar la carpeta `bestlhoods` por una nueva que vamos crear llamada `results` dentro de la carpeta `bootstrap`

Memoria necesaria: ~70mb

Tiempo de ejecución: ~10min

Output: dentro de la carpeta `results` vamos tener todos los archivos `bestlhoods` para calcular los intervalos de confianza.

En `results`: `cat ${PREFIX}_{1..10}.bestlhoods | grep -v MaxObsLhood` Al hacer esto, generaremos una tabla fácil que podemos copiar y pegar en un archivo txt para leerlo en R. Si desea poner un encabezado en la tabla, simplemente cópielo de uno de los archivos, por ejemplo: `head -n 1 Lupinus_div_1.bestlhoods`

9. En R en tu computadora:

```
setwd("<RUTA>")
boot <- read.csv("fastsimcoal_bootstrap.txt", sep = "\t")

quantile(boot$NANC,c(.025,.975))
quantile(boot$NPOP1,c(.025,.975))
quantile(boot$NPOP2,c(.025,.975))
quantile(boot$TDIV,c(.025,.975))
```

y listo :smile: