Introducción:

En este recurso vamos a aprender todo lo posible de la práctica 6. No solo aprender a hacerla, ya que seguramente muchos de vosotros ya la tendréis resuelta en vuestro script, también aprender a manejar perfectamente todas las funciones que se utilizan a lo largo de la práctica, aprender porqué ciertas cosas son de una forma y no de otra, aprender errores típicos que nos puedan ayudar a manejar R en diversas situaciones y consejo para entender mejor. Algunos de los errores más comunes **estarán señalados así, en color rojo y subrayados**

¡Vamos a por ello!

Antes de empezar: Esta práctica es la primera que involucra conceptos vistos en las clases teóricas, como son la interpolación de Lagrange y los Polinomios de base de Lagrange.

Aunque se podría realizar sin saber esto, ya que viene bien explicado en el guion, es recomendable revisarlo antes, para saber qué estamos haciendo y porqué, y que nos resulte más familiar a la hora de traspasarlo a lenguaje R.

También se va a dar por sabido algunas de las funciones más básicas y que ya hemos manejado con anterioridad, como pueden ser los bucles *for*.

Ejercicio 1:

Se conoce la producción de bioetanol que se obtiene en una planta de biocombustibles en función del tiempo. Los valores de la concentración (g/l) estan almacenados en el vector: B(10,20,35.33,35.5) y los instantes de tiempo (*horas*) en el vector s(1,3.5,5,10).

SE PIDE:

Realizar un script llamado **Bioetanol.R** para estimar, mediante interpolación de Lagrange, la concentración de bioetanol en los instantes $t = 2$ y $t = 6.5$. Para ello, se empleara la siguiente expresión del polinomio interpolador de Lagrange.

$$p = \sum_{i=1}^{n} B_i L_i$$

1) Comprender lo que se pide

Arriba tenemos el enunciado del primer ejercicio. En él se muestran los datos del problema en forma de vectores (‘B’ y ‘s’). Lo que queremos es estimar el valor de la concentración de Bioetanol en instantes distintos a los que vienen dados, y nos explica que para ello se empleará la siguiente expresión del polinomio interpolador de Lagrange.

Recomendamos poner los datos iniciales al comienzo del programa.

PASOS:

1. Programar la función Polbase para obtener las funciones de base la interpolación de Lagrange, que están dadas por la expresión

$$L_i = \prod_{\substack{j=1 \\ j \neq i}}^{n} \frac{t - s_j}{s_i - s_j}, (i = 1, \ldots, n)$$

L es un vector cuyas componentes son los Valores de cada pol de base en el punto t

2) Las Funciones de R

Lo primero que nos piden para comenzar es programar la función Polbase. No es la primera vez que vemos funciones en R, y aparecerán más adelante según avancemos en la práctica, por lo que es vital comprender bien en que consisten.

¿Cómo funcionan las *function* de R?

La estructura es bastante sencilla, pero es vital conocerla para poder usarlas correctamente. Primero, hace falta nombrarla la función y establecer los argumentos, valores que van a ir cambiando. (A lo mejor lo entendemos mejor si pensamos que *la function 'está en función' de los argumentos).*

```
> hipotenusa<-function(catetol, cateto2){
```

```
hipotenusa<-function(catetol, cateto2){
      result<-sqrt((catetol^2)+(cateto2^2))
return(result)
}
```

Aquí, la función que hemos llamado hipotenusa depende del valor de los catetos.

Falta poner el proceso, que en este caso es el Teorema de Pitágoras.

Tras la función hemos asignado a la variable *result* el cuerpo de la función, para que el resultado se guarde dicha variable. **Es importante no olvidar la función *return()*, que hace que la función devuelva el valor calculado.** Si no lo ponemos, el resultado no se mostrará:

```
> hipotenusa<-function(catetol, cateto2){
+ result<-sqrt((catetol^2)+(cateto2^2))
+  return(result)
+ }
> hipotenusa(3,4)
[1] 5
>
```

```
> hipotenusa<-function(catetol, cateto2){
+ result<-sqrt((catetol^2)+(cateto2^2))
+  }
> hipotenusa(3,4)
>
```

Si sustituimos los valores de los catetos por 3 y 4, la función nos devolverá el valor de la hipotenusa, porque en la función ya hemos definido las operaciones que debe hacer (observa como si no ponemos el *return()* no nos devuelve el valor).

Esto es lo básico de las *function* de R, pueden ser más o menos largas, con más o menos argumentos, pero al final es siempre lo mismo.

Cosas importantes a tener en cuenta:

Para que una función te devuelva un valor tienes que llamarla. Es decir, al comienzo del programa puedes dejar definida la *function*, en este caso, la función que calcula la hipotenusa a partir de los catetos. Esta función se queda definida y sólo va a 'responder' si se la llama, da igual en qué parte del programa sea.

También es importante saber usar y nombrar los argumentos. Si nosotros tenemos

```
Polbase<-function(Argumento1, Argumento2,Argumento3){
```

cualquier operación dentro de la function debe estar definida por los argumentos Argumento1, Argumento2 y Argumento3, si empleamos otros nombres, nos dará error.

```
Polbase<-function(Argumento1, Argumento2,Argumento3){

      L<-c(0)
      for(i in 1:Argumento3){
            L[i]=1
            for(j in 1:Argumento3){
                  if(j!=i){
                        L[i]=L[i]*(Argumento2-Argumento1[j])
                  }
            }
      }
return(L)
}
```

¡Dentro de la function, las variables deben tener siempre el mismo nombre que los argumentos del paréntesis!

Una vez definida la función, la cosa cambia. Si yo en el Argumento1 quiero introducir, por ejemplo, el vector *'s'* dado, en el Argumento2 el número 6.5 y en Argumento3 el comando *length(s),* ¡puedo hacerlo!

Eso sí, habrá que tener en cuenta el carácter numérico que introducimos. Es decir, si hemos programado Polbase para que en Argumento1 se introduzca un vector, si introducimos, por ejemplo, el número 4, saldrá error.

```
> Polbase(s,6.5,length(s))
[1] -15.750 -28.875 -57.750  24.750
> Polbase(4,6.5,length(s))
[1] NA NA NA NA
```

¡Continuemos con la práctica!

3) Programar la función

Ya sabemos lo básico sobre las funciones, ahora vamos a programar la nuestra.

Para la función Polbase, los argumentos (aquellos valores de los que depende la función) son tres: s, t y n.

El cuerpo de la función es bastante sencillo. Conviene al principio inicializar L=c(0). En el cuerpo nos encontramos un productorio, por lo que la operación quedaría algo así:

```
L[i]=L[i]*(t-s[j])/(s[i]-s[j])
```

Un error común es olvidar que tenemos L=0, o tratar el productorio como si fuese un sumatorio y pensar que hay que inicializarlo a 0. Como aquí estamos multiplicando, si dejamos L=0, ¡nos daría siempre 0, porque cualquier número por 0 es 0! **Por eso debemos inicializar antes a 1**. Quedaría algo así:

```
Polbase<-function(s,t,n){

L<-c(0)
      for(i in 1:n){
            L[i]=1
            for(j in 1:n){
                  if(j!=i){
                        L[i]=L[i]*(t-s[j])/(s[i]-s[j])
                  }
            }
      }
return(L)
}
```

Revisar la ortografía y ¡cuidado con las llaves! Tampoco debemos olvidar usar la función *return()* al final.

Aquí hay que tener **especial cuidado con la colocaciones de las llaves, ¡sobre todo cuando es una estructura condicional *if* !**

2. Programar la función PolInterp para obtener el polinomio interpolador en el punto t. Dicha función recibirá como argumentos de entrada:

- un vector B (que contiene los valores de la función que se interpola),
- el vector L (que contiene los polinomios de base de Lagrange evaluados en el punto t)
- la variable n (número de puntos del soporte de interpolación.

En el segundo paso tenemos que utilizar otra vez una *function*, pero ¡eso ya lo tenemos controlado! Además, en el enunciado nos están especificando cuáles serán los argumentos de entrada. ¡Está chupado!

Si llamamos *P* al vector que queremos que almacene los n polinomios de base, resultado de multiplicar un elemento del vector *L* (obtenido arriba) con el correspondiente del vector *B* (nos lo dan en el enunciado), deberíamos tener:

```
PolInterp<-function(B, L, n){
      B<-c(10,20,35.33,35.5)
      p=0
      for(i in 1:n){
            p=p+B[i]*L[i]
      }
return(p)
}
```

En este caso el vector *B* está puesto dentro de la función, pero también puede ponerse al comienzo del programa.

¡Vamos a hacer una prueba a ver cómo va!

Copia en la consola todo lo que llevamos hasta hora, y no te olvides de incluir los datos iniciales (vectores *s* y *B).*

Primero calcula el polinomio de base para tiempo=2 horas. Para ello llama a la función y sustituye en sus argumentos. Recuerda que 's' es un vector (hay que poner directamente *'s'* en el argumento), el tiempo son 2 horas y *n* corresponde a la longitud de *'s'*, en este caso 4. Debería salirte esto:

```
> Polbase(s,2,4)
[1]  0.40000000  0.98461538 -0.40000000  0.01538462
```

Ahora vamos a interpolar el valor para tiempo=2 horas. Para ello hay que hacer lo mismo, pero con la función PolInterp. En este caso, el segundo argumento corresponde con el polinomio de base para t=2 (calculado arriba), por lo que puedes asignar ese vector a una variable o directamente ponerlo en el argumento.

```
> PolInterp(B, Polbase(s,2,4), 4)
[1] 10.10646
```

¡Cuidado con la ortografía y con los corchetes!

Bueno, ¡continuemos!

4. Obtener 1001 abscisas equidistantes en el intervalo [s[1],s[n]] y almacenarlas en un vector x. (Calcularemos el valor del polinomio interpolador en cada abscisa para dibujar).

4) Obtener las abscisas

Este cuarto paso es bastante sencillo, pero hay varias maneras de hacerlo. ¡Vamos a repasarlas todas, que nunca viene mal!

La primera opción, posiblemente la más sencilla, es usar la función *seq()*. Esta función crea una secuencia desde un número inicial dado hasta otro final también dado, y podemos incluir la cantidad de números que queremos que haya en entre esos valores.

Se escribiría de la forma: `x=seq(s[1], s[n], length=1001)`

Otra forma es usar también la función *seq()*, pero en vez de poner al final la longitud, pondríamos un número correspondiente al número de intervalos (cambiamos *length=1001* por *'h'*, siendo *'h'* el número de intervalos *(h=(s[n]-s[1])/1000))*.

La última forma es haciendo un bucle. Esto te dejamos hacerlo a ti solo. Recuerda que estamos intentando conseguir un vector *'x'* con 1001 componentes que vayan desde *s[1]* hasta *s[n]*…

Ahora tenemos que calcular el valor del polinomio interpolador para cada valor almacenado en '*x'*.

Para ello habrá que crear un polinomio de base para cada una de las 1001 abscisas almacenadas en *'x'*, y luego obtener un vector que contenga los valores del polinomio interpolador en cada punto.

5. Llamamos 1001 veces a las funciones Polbase y PolInterp. La salida de Polinterp se guardará en un vector f[k], k=1,…,1001.

Para ello utilizaríamos un único bucle (desde 1 hasta 1001). Se puede hacer también usando dos bucles separados. Puede parecer un poco más complejo, pero solo hay que tener en cuenta que LL es un vector con 4 componentes. Habría que almacenar entonces en el vector LL 1001 grupos de 4 componentes… ¿Se te ocurre cómo?

(Pista…¡una matriz!)

Dejamos por aquí ambas formas de hacerlo

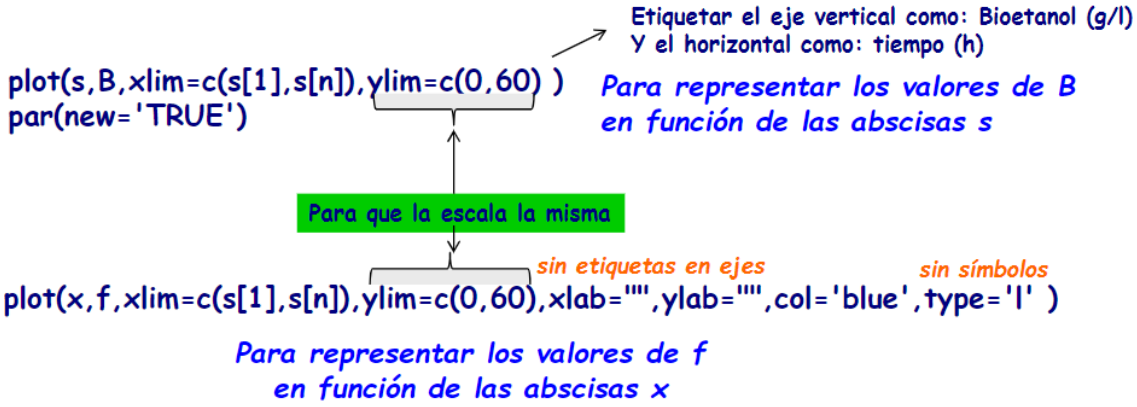
```
f=0
for(k in 1:1001){
      LL<-Polbase(s,x[k],n)
      f[k]<-PolInterp(B,LL,n)
}
f
```

(usando un único bucle)

```
f=0
LL<-matrix(0, nrow=1001, ncol=4)
for(k in 1:1001){
      LL[k,]<-Polbase(s,x[k],n)

}
LL
for(k in 1:1001){
      f[k]<-PolInterp(B,LL[k,],n)
}
f
```

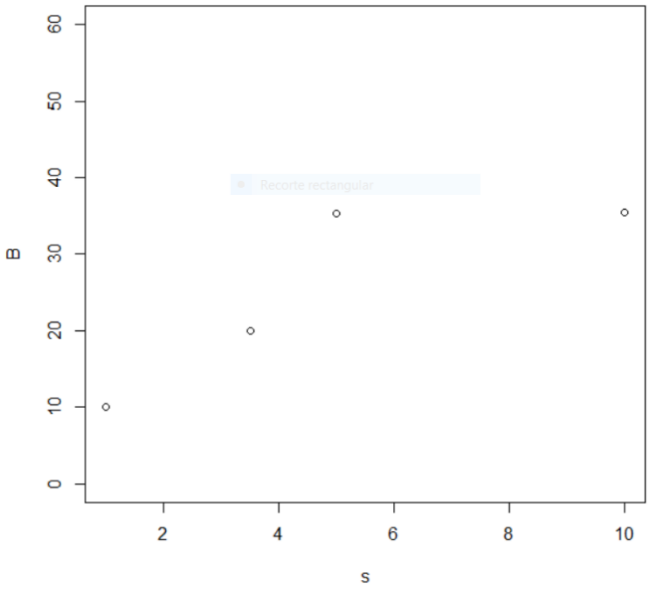
(usando dos bucles)



5) Hacer las gráficas

La última parte de este ejercicio consiste en dibujar y representar en la gráfica. ¡Vamos, que ya casi lo tenemos!

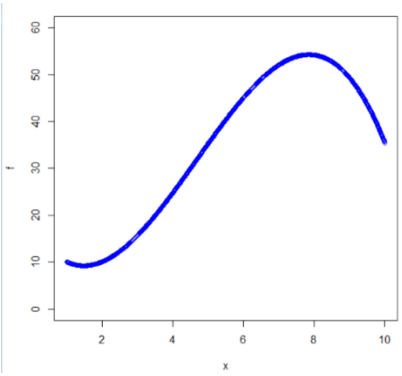
Vamos a ir poco a poco para entender mejor lo que estamos poniendo. Siempre que queramos hacer una gráfica tendremos que usar el comando *plot().* A continuación de este comando, entre los paréntesis, tendremos que indicar cuales son las variables. En la primera frase son los vectores *'s'* y '*B*', que, si recordamos, cada uno contiene 4 valores.



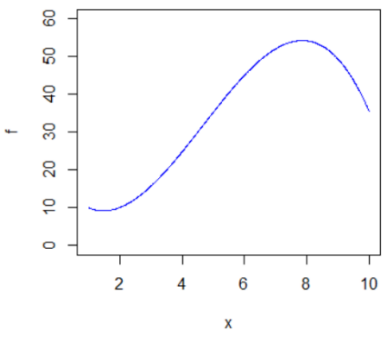
El orden es importante, ya que el primer argumento que introducimos en *plot()* va a ser el que aparezca a en el eje de abscisas.

A la izquierda tenemos representada únicamente la primera función

Ahora vamos con la segunda gráfica. En esta vamos a representar *'x'* (vector con los 1001 valores) con *'f'* (vector que almacena el valor obtenido al hacer la interpolación en cada punto).



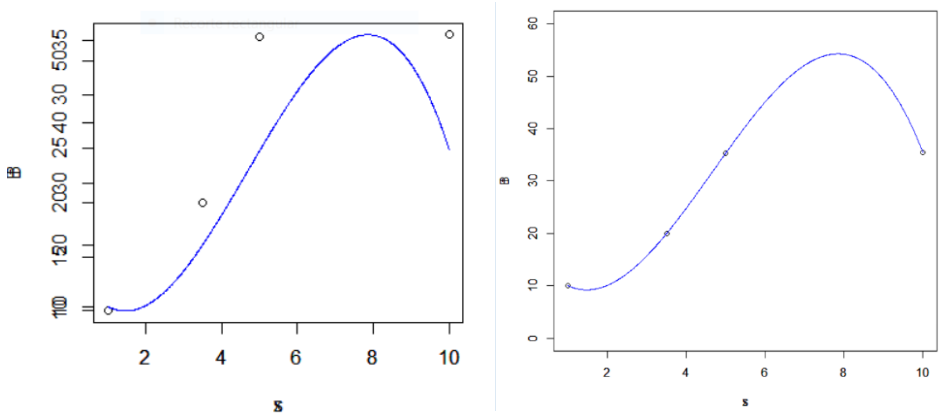
En este caso hemos añadido el comando *'type='blue''* para que los puntos sean de color azul. Ahora mismo en la gráfica hemos omitido el comando *'type=l'*, por lo que tenemos una gráfica con 1001 puntos (por eso la línea se va más gruesa, porque en realidad no es una línea sino muchos puntos).



Ahora ya hemos añadido el comando *'type=l'* y vemos claramente como la gráfica está representada por una línea y no por muchos puntos.

Para que ambas funciones se representen en la misma gráfica hace falta añadir el comando *'par(new='TRUE')'*, ¡Así de fácil!

Por último, para que ambas funciones estén representadas en la ‘misma escala’ y los puntos de la primera coincidan con los de la recta, hace falta añadir otro comando. Debemos añadir en la función *‘plot()’* los comandos *xlim(,)* e *ylim(,)*. Dentro de los paréntesis de estos comandos se introducirán los ‘valores iniciales y límites’ que van a dar la escala de la función. ¡Veamos como quedarían las gráficas con y sin estos comandos!



¿Ves la diferencia?

Pues… ¡ya está! Esto es todo lo que tienes que saber sobre el primer ejercicio de la práctica 6. No era tan difícil, ¿verdad? Acuérdate de revisar siempre la ortografía para evitos esos errores tontos.

Vuelve a leer el documento para quedarte con los conceptos más importantes y, si todavía no lo has hecho en tu ordenador, ¡a qué esperas!