ALGORITMOS DE INTERPOLACIÓN POLINÓMICA DE NEWTON

Índice:

*Diferencias divididas: Cálculo del polinomio interpolador lagrangiano usando la fórmula de interpolación de Newton

Para sistematizar el cálculo de las diferencias divididas, empleamos la siguiente tabla:

	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	
s_0	$f[s_0]$	$f[s_0,s_1]$	$f[s_0,s_1,s_2]$	$f[s_0,s_1,s_2,s_3]$	$f[s_0,s_1,s_2,s_3,s_4]$	$i=0$
s_1	$f[s_1]$	$f[s_1,s_2]$	$f[s_1,s_2,s_3]$	$f[s_1,s_2,s_3,s_4]$	0	$i=1$
s_2	$f[s_2]$	$f[s_2,s_3]$	$f[s_2,s_3,s_4]$	0	0	$i=2$
s_3	$f[s_3]$	$f[s_3,s_4]$	0	0	0	$i=3$
s_4	$f[s_4]$	0	0	0	0	$i=4$

$$p_n(x) = f[s_0] + f[s_0,s_1](x-s_0) + f[s_0,s_1,s_2](x-s_0)\,(x-s_1) + \cdots$$
$$+ f[s_0,s_1,s_2,s_3](x-s_0)\,(x-s_1)(x-s_2) + \cdots$$
$$+ f[s_0,s_1,s_2,s_3,s_4](x-s_0)\,(x-s_1)(x-s_2)(x-s_3) + \cdots$$

$$f[s_0,s_1,s_2,\cdots,s_n] = \frac{f[s_1,s_2,\cdots,s_n] - f[s_0,s_1,s_2,\cdots,s_{n-1}]}{s_n - s_0}$$

1º EJERCICIO: Implementa una función en R que tome las abscisas y ordenadas de una nube de puntos y retorne la tabla de diferencias divididas.

```
#En primer lugar se debe hacer un chequeo de los parámetros de entrada:
  x e y tienes que tener el mismo tamaño, por ejemplo

    diferencias_divididas <- function(x, y) { #con function indicamos que diferencias
    divididas va a ser una función que dependerá de los valores de x e y.
     n <- length(x) – 1 # n va a ser el grado del polinomio.

# Inicializar la matriz para almacenar las diferencias divididas

    tabla <- matrix(0, nrow = n + 1, ncol = n + 1) #Inicializamos la matriz a 0 y pongo n+1
porque he dicho que n era la longitud - 1, entonces para que la longitud de las columnas y las
filas coincida con la longitud que quiero le sumo 1.
```

```
# La primera columna son los valores de y

        tabla[, 1] <- y

# Calcular las diferencias divididas
      for (j in 2:(n + 1)) { #primero ponemos un for que va a recorrer las columnas y que empieza en la segunda ya que la primera ya la tenemos rellena.

        for (i in 1:(n + 2 - j)) { # para la i el primer elemento es el 1 y acaba en n+2-j: suma 2 porque n+1 es el número de filas que hay, entonces, como para ir calculando las diferencias divididas en necesario coger una fila más, se le suma +1 (cuando queremos calcular la diferencia dividida en una fila cogíamos el valor de debajo de la columna anterior menos el de la fila en la que estamos). Además se le resta j porque según avanzamos en las columnas, en las filas, empezando por abajo va quedando cero y el número de ceros que haya se corresponde con el valor que toma j.

          tabla[i, j] <- (tabla[i + 1, j - 1] - tabla[i, j - 1]) / (x[i + j - 1] - x[i]) # aquí ya aplico la fórmula de las diferencias divididas: pongo la i siguiente a la que estoy trabajando y le resto la i. En el caso de la j le resto 1 para que al haber empezado en j=2 el bucle se corresponda con la columna que estoy calculando. En el denominador pongo el último elemento del soporte – el 1º. Ponemos i + j – 1 porque así obtienes el Sn que aparece en la fórmula (Sn – So). La x del último menos la x de la primera posición: si tu tienes i=1 y j=2 te da 1+2-1=2 que corresponde con el punto del soporte en esa posición.
        }
      }

# Nombrar las filas y columnas

      rownames(tabla) <- x
       colnames(tabla) <- paste0("Orden ", 0:n)  #paste0 lo que va a hacer es concatenar cadenas de texto, va a ir poniendo en cada columna el orden del polinomio.
       return(tabla)
  }
```

2º EJERCICIO: Implementa una función en R que, a partir de la tabla de las diferencias divididas, devuelva el valor del polinomio interpolador evaluado en un vector x.

1ªFORMA

En este caso vamos a utilizar esta fórmula:

$$p_n(x) = \sum_{i=0}^{n} f[s_0, s_1, \dots, s_i] \prod_{j=0}^{i-1} (x - s_j)$$

Vemos que hay dos bucles: el del sumatorio y el del productorio. Sin embargo, hay que tener en cuenta que se necesita otro que recorra el vector x que se nos pide.

IMPORTANTE: Para realizar este programa vamos a aplicar la estrategia que usamos para programar bucles anidados: ir de dentro a fuera desglosando la fórmula en etapas más simples. Primero inicializo el sumatorio (llamado resultado en este caso) y como depende del valor del productorio, pongo este dentro de un bucle for y lo calculo. Una vez lo tengo ya pongo la fórmula del sumatorio.

```
polinomio_newton_sin_funcion_interna <- function(
            tabla_dif_div,
             x_orig, # es el vector en el que se almacenan los puntos del soporte
             x_eval) # vector en el que se guardan los elementos que se van a evaluar
{
 n <- ncol(tabla_dif_div) - 1
 coef <- as.numeric(tabla_dif_div[1, ]) #indico que coef son los valores de las diferencias divididas que se van situando en la primera fila, se corresponden con los coeficientes canónicos.

# Inicializar un vector para almacenar los resultados

 resultados <- numeric(length(x_eval)) # resultados es el vector donde voy a ir almacenando los distintos valores de resultado.

for (k in seq_along(x_eval)) {#sec_along es lo mismo que 1:lenght(x_eval)
x <- x_eval[k]
resultado<-coef[1] # resultado es la variable que va guardando cada resultado individual y la inicializo en el valor de f[So]: diferencia dividida de orden 0 (término constante). Este paso sirve para inicializar el sumatorio

for (i in 1:n) {

    producto <- 1 #inicializo el productorio a 1
    for (j in 1:i) {
     producto <- producto * (x - x_orig[j]) # aquí estoy calculando el productorio de x-Sj como indica la fórmula.
    }

resultado <- resultado + coef[i + 1] * producto } # coef está en función de i+1 porque he empezado en 1.

  resultados[k] <- resultado
 }
 return(resultados)
}
```

2ª FORMA

Cuando estamos haciendo un programa y sabemos que vamos a encadenar muchos bucles o repetir el código, nos puede ser útil definir funciones internas.

```
polinomio_newton_con_funcion_interna <- function(
                tabla_dif_div,
                x_orig,
                x_eval)
{
 n <- ncol(tabla_dif_div) - 1 #n es el grado y va a ser el número de columnas que tenga la tabla menos 1
 coef <- as.numeric(tabla_dif_div[1, ])

p <- function(x) { #indico que p va a ser una función que depende del valor de x
   resultado <- coef[1]
    for(i in 1:n){
                producto <- 1
                 for (j in 1:i) {
                            producto<-producto*(x-x_orig[j])
                          }
                 resultado <- resultado + coef[i + 1] * producto
                }
   return(resultado)
       }
 return ( sapply (x_eval, p)) #sapply permite aplicar una función p en el vector x_eval
}
```

3ª FORMA: para mejorar la inteligibilidad del código puedo ir recuperando cálculos que ya he realizado anteriormente.

```
polinomio_newton <- function(
                tabla_dif_div,
                x_orig,
                x_eval)
{
 n <- ncol(tabla_dif_div) - 1
 coef <- as.numeric(tabla_dif_div[1, ])

p <- function(x) {
   resultado <- coef[1]
   producto <- 1
   for(i in 2: (n+1)) {
              producto <- product * (x -  x_orig[i- 1])
              resultado <- resultado + coef[i] * producto
          }
   return(resultado)
     }
 return ( sapply (x_eval, p))
```

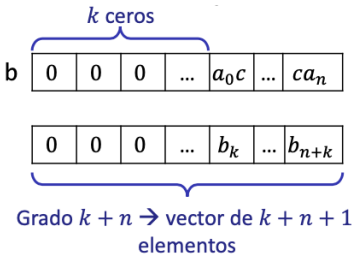
Pasar de polinomio de Newton a forma canónica-R

Los polinomios en forma canónica pueden estar representados mediante vectores, tales que:

a	a_0	a_1	...	...	a_n

Representa: $p_n(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$

Multiplicar el polinomio por una constante c y por x elevado a k es equivalente a desplazar el vector k veces a la izquierda y multiplicar todos sus coeficientes por c :



Representa: $p_n(x)cx^k = ca_0x^k + ca_1xx^k + c\,a_2x^2x^k + \cdots + c\,a_nx^n\,x^n = = ca_0x^k + ca_1x^{k+1} + ca_2x^{k+2} + \cdots + ca_nx^{n+k}$

- A su vez si $q_n(x)$ está representado por el vector d y $r_n(x) = p_n(x) + q_n(x)$ está representado por el vector e, entonces:

e	$a_0 + d_0$	$a_1 + d_1$	...	...	$a_n + d_n$

Representa: $r_n(x)$

prod_pol para cada iteración de i

$$p_n(x) = \underbrace{\sum_{i=0}^{n} f[s_0, s_1, \ldots, s_i] \overbrace{\prod_{j=0}^{i-1}(x - s_j)}}$$

pol para cada iteración de i

Por un lado, voy a acumular el vector que representa al polinomio canónico del producto y por otro, al polinomio final.

$$p_n(x) = f[s_0] + \\ + f[s_0, s_1](x - s_0) + \\ + f[s_0, s_1, s_2](x - s_0)\,(x - s_1) \\ + f[s_0, s_1, s_2, s_3](x - s_0)\,(x - s_1)(x - s_2) + \cdots \\ + f[s_0, s_1, s_2, s_3, s_4](x - s_0)\,(x - s_1)(x - s_2)(x - s_3) + \cdots$$

```
polinomio_newton_canonico <- function(
            tabla_dif_div,
            x_orig){

#Determinar el grado del polinomio
      n <- length(tabla_dif_div[1, ]) – 1

#Extraer los coeficientes de Newton desde la primera fila de la tabla de diferencias divididas
      coef_newton <- as.numeric(tabla_dif_div[1, ])

#Inicializar el polinomio con el término constante
       pol <- c(coef_newton[1]) #lo inicializamos a 1, el valor del coeficiente de f[So]
       prod_pol <- 1 #prod_pol es un sumatorio que inicializamos a 1
          for(i in 2: (n+1)) {
            prod_pol = c(0,prod_pol)- c(prod_pol,0)*x_orig[i-1] # poner un 0 delante
       corresponde a multiplicar por x,  como hemos añadido un 0, hay que añadir uno al
       final para que se mantenga el tamaño
            pol=c(pol,0) + coef_newton[i]*prod_pol
                }
        return (pol) #nos da el valor de los distintos coeficientes
       }
```

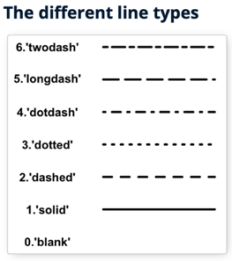
```
dibujar <- function(s, fS, x_sample, y_sample, y_pol) {

# Comprobamos que todos los vectores tienen las longitudes correctas
    if (length(s) != length(fS)) {
      stop("s y fS deben tener la misma longitud.")
    }
    if (length(x_sample) != length(y_sample) || length(x_sample) != length(y_pol) ||
   length(y_sample) != length(y_pol)) {
      stop("x_sample, y_sample y y_pol deben tener la misma longitud.")
    }
```

Progama para representar funciones (dibujar)

```
dibujar <- function(s, fS, x_sample, y_sample, y_pol) {

  # Comprobamos que todos los vectores tienen las longitudes correctas
      if (length(s) != length(fS)) {#este bucle if nos permite asegurarnos que la longitud de s es la misma que fs
        stop("s y fS deben tener la misma longitud.") #stop nos muestra el mensaje de error y deja de evaluar la función
      }
      if (length(x_sample) != length(y_sample) || length(x_sample) != length(y_pol) ||
     length(y_sample) != length(y_pol)) { #las dos rallas verticals permiten separar las distintas condiciones que queremos que se cumplan para poder seguir
        stop("x_sample, y_sample y y_pol deben tener la misma longitud.")
      }
  # Crear el gráfico
      plot(s, fS, #plot dibuja puntos
          type = "p", pch = 16, col = "black", cex=2, #type="p" va a dibujar puntos, pch especifica el estilo de los puntos, cex sirve para modificar el tamaño de los puntos
          xlab = "Eje X", ylab = "Eje Y", #xlab, ylab ponen los nombres de los ejes correspondintes
          ylim = range(c(fS, y_sample, y_pol))) #range nos entrega el valor mínimo y máximo de la variable que hemos ingresado
```




```
  # Agregar línea continua azul para x_sample frente a y_sample
      lines(x_sample, y_sample, type = "l", col = "red",lw =3) #ponemos los valores que queremos representar(x_sample, y _sample), luego type="l" es para que dibuje un línea, col para indicar el color del que queremos la línea, lw=3 indica el grosor de la línea
  # Agregar línea discontinua roja para x_sample frente a y_pol
      lines(x_sample, y_pol, type = "l", col = "black", lty = 2,lw=2) #lty=2 nos indica que va a ser una línea discontínua
  # Agregar una leyenda
      legend("topright", legend = c("f(s)", "f(x)", "p(x)"), #legend es para crear una leyenda, donde ponemos NA indicamos que no se va a dibujar lo que indique el comando para esa función.
```

```
        col = c("black", "red", "black"), pch = c(16, NA, NA), lty = c(NA, 1, 2), lw = c(NA, 2,
    2))
    }
```

PROGRAMA TRABLA DE DIFERENCIAS DIVIDIDAS+ GRÁFICA+ ECUACIÓN DEL POLINOMIO (FORMA CANÓNICA)

Ejemplo concreto

```
    cat("\014")
    rm(list=ls())

# Llamamos a los programas que necesitamos para crear la tabla de diferencias divididas y dibujar la gráfica del polinomio interpolador

    source("diferencias_divididas.R")  #para llamar a la función es importante saber dónde hemos guardado el programa y con qué nombre.
    source("dibujar.R")

# Tenemos unos puntos de soporte y los respectivos valores que toma la función en dichos puntos

    s <- c(1, 7, 13, 19, 25)
    fS <- c(500,300,415,313,251)

    tabla_dif <- diferencias_divididas(s,fS) #en el programa de diferencias divididas metemos los valores de s y fs
    print(tabla_dif)

    a = polinomio_newton_canonico (tabla_dif, s)

    y_15  <- polinomio_newton(tabla_dif,s,15) #pones el 15 porque es el valor que te pide el ejercicio evaluar

    print(paste("Habrá aproximadamente ", round(y_15,digits=0), " flamencos")) # round permite aproximar a un cierto número de decimales, en este caso 0 porque es el número de flamencos
    dibujar_canonicos(s, a)
```

Programas sin anotaciones:

- **Programa diferencias divididas (tabla)**

```
diferencias_divididas <- function(x, y) {
  n <- length(x) - 1

  # Inicializar la matriz para almacenar las diferencias divididas
  tabla <- matrix(0, nrow = n + 1, ncol = n + 1)

  # La primera columna son los valores de y
```

```
  tabla[, 1] <- y

  # Calcular las diferencias divididas
  for (j in 2:(n + 1)) {
    for (i in 1:(n + 2 - j)) {
      tabla[i, j] <- (tabla[i + 1, j - 1] - tabla[i, j - 1]) / (x[i + j - 1] - x[i])
    }
  }

  # Nombrar las filas y columnas
  rownames(tabla) <- x
  colnames(tabla) <- paste0("Orden ", 0:n)

  return(tabla)
}
```

- **Programa polinomio Newton con función interna**

```
polinomio_newton <- function(tabla_dif_div, x_orig, x_eval) {
  n <- ncol(tabla_dif_div) - 1
  coef <-  as.numeric(tabla_dif_div[1, ])

  p <- function(x) {
    resultado <- coef[1]
    producto <- 1
    for (i in 2:(n + 1)) {
      producto <- producto * (x - x_orig[i - 1])
      resultado <- resultado + coef[i] * producto
    }
    return(resultado)
  }

  return(sapply(x_eval, p))
}
```

- **Programa polinomio Newton sin función interna**

```
polinomio_newton_sin_funcion_interna <- function(tabla_dif_div, x_orig, x_eval) {
  n <- ncol(tabla_dif_div) - 1
  coef <- tabla_dif_div[1, ]

  # Inicializar un vector para almacenar los resultados
  resultados <- numeric(length(x_eval))

  for (k in seq_along(x_eval)) {
    x <- x_eval[k]
    resultado <- coef[1]  # Término constante
    for (i in 1:n) {
```

```
    producto <- 1
    for (j in 1:i) {
      producto <- producto * (x - x_orig[j])
    }
    resultado <- resultado + coef[i + 1] * producto
  }
  resultados[k] <- resultado
 }
 return(resultados)
}
polinomio_newton_con_funcion_interna <- function(
                    tabla_dif_div,
                    x_orig,
                    x_eval)
{
 n <- ncol(tabla_dif_div) - 1
 coef <- as.numeric(tabla_dif_div[1, ])

 p <- function(x) {
  resultado <- coef[1]
  for (i in 1:n) {
    producto <- 1
    for (j in 1:i) {
      producto <- producto * (x - x_orig[j])
    }
    resultado <- resultado + coef[i + 1] * producto
  }
  return(resultado)
 }

 return(sapply(x_eval, p))
}
```

- **Programa para representar la gráfica del polinomio canónico**

```
dibujar_canonicos <- function(x_orig, coeficientes) {
 # Crear una función para el polinomio
 polinomio <- function(x) {
  suma <- 0
  for (i in 1:length(coeficientes)) {
    suma <- suma + coeficientes[i] * x^(i-1)
  }
  return(suma)
 }
 # Determinar el rango para x
 x_min <- min(x_orig)
 x_max <- max(x_orig)
```

```
  rango <- x_max - x_min

  # Crear un vector de valores x para una curva suave
  x <- seq(x_min - 0.1*rango, x_max + 0.1*rango, length.out = 1000)

  # Calcular los valores y correspondientes
  y <- sapply(x, polinomio)

  # Crear la gráfica
  plot(x, y, type = "l", col = "blue",
       main = "Gráfica del Polinomio Canónico",
       xlab = "x", ylab = "y",
       xlim = c(min(x), max(x)), ylim = range(y))

  # Añadir puntos para el soporte original
  points(x_orig, sapply(x_orig, polinomio), col = "red", pch = 19)
}
```

- **Programa polinomio Newton en forma canónica**

```
polinomio_newton_canonico <- function(tabla_dif_div, x_orig) {
  # Determinar el grado del polinomio
  n <- length(tabla_dif_div[1, ]) - 1

  # Extraer los coeficientes de Newton desde la primera fila de la tabla de diferencias divididas
  coef_newton <- as.numeric(tabla_dif_div[1, ])

  # Inicializar el polinomio con el término constante
  pol <- c(coef_newton[1])
  prod_pol <- 1

  for (i in 2:(n+1))
  {
    prod_pol = c(0,prod_pol)-c(prod_pol,0)*x_orig[i-1]
    pol=c(pol,0)+coef_newton[i]*prod_pol
  }
  # Imprimir la ecuación del polinomio
  cat("Ecuación del polinomio:\n")
  cat("p(x) = ")
  for (i in 1:length(pol)) {
    if (i > 1 && pol[i]>0) cat(" + ")
    cat(sprintf("%.4f", pol[i]))
    if (i > 1) cat(sprintf("x^%d", i-1))
  }
  cat("\n")
  return(pol)
}
```